

We have got the company!

- **Fruit flies** were the first living creatures to be sent into space. Lucky one!
- A **bee's** wings beat 190 times a second, that's 11,400 times a minute. OMG!
- **Caterpillars** have 12 eyes!
- One **dung beetle** can drag 1,141 times its weight – that's like a human pulling six double-decker buses!
- **Grasshoppers** existed before dinosaurs!

<https://www.natgeokids.com/uk/discover/animals/insects/15-facts-about-bugs/>

DotNetCore

Prepared for Vth semester DDU-CE students
2022-23 WAD

Apurva A Mehta

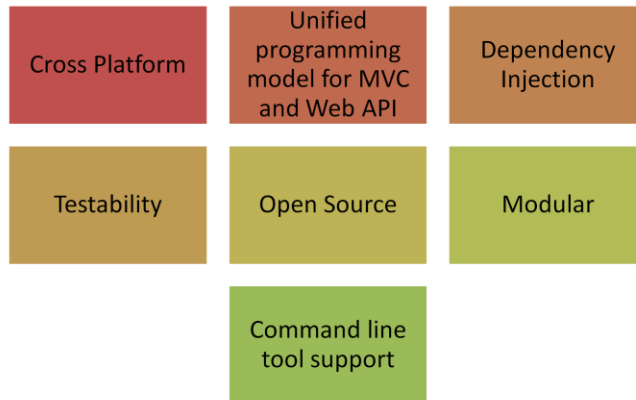
.NET Core

- .NET Core is a *software development framework* which is used to create different types of applications.
- There are many frameworks which are written on top of .NET Core for creating various applications.
- .NET Core is a **cross-platform**, **high-performance**, **open-source** framework for building modern, cloud-based, internet-connected applications.

.NET has a long history, but .NET Core is very young.

.NET **Core** 3.1

Benefits and Features



Cross Platform

- ASP.NET 4.x applications → windows platform
- .NET Core applications can be **developed** and **run** across different platforms like Windows, macOS, or Linux.
- ASP.NET 4.x applications can be hosted only on IIS.
- .NET Core applications can be hosted on IIS, Apache, Docker, or even self-host in your own process.
- From a development standpoint, you can either use Visual Studio or Visual Studio Code, Sublime, Bracket, Vim, Etc... for building .NET Core applications.

ASP.NET 4.x applications run only on windows platform, where as ASP.NET Core applications can be developed and run across different platforms like Windows, macOS, or Linux.

ASP.NET 4.x applications can be hosted only on IIS, where as ASP.NET Core applications can be hosted on IIS, Apache, Docker, or even self-host in your own process.

From a development standpoint, you can either use Visual Studio or Visual Studio Code for building .NET Core applications. You can also use third party editors like Sublime.

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and deploy it as one package.

Unified programming model

- With ASP.NET core, we use the same unified programming model to create MVC style web applications and ASP.NET Web API's.

With ASP.NET core, we use the same unified programming model to create MVC style web applications and ASP.NET Web API's. In both the cases, the Controller that we create inherits from the same Controller base class and returns IActionResult. As the name implies IActionResult is an interface and it has got several implementations. ViewResult and JsonResult are just 2 examples of the built-in result types that implement IActionResult interface. So, in the case of a Web API, the controller returns a JsonResult and in the case of an MVC style web application it returns a ViewResult. If this does not make much sense at the moment, do not worry, it will be crystal clear as we progress through the course.

Dependency Injection

- ASP.NET Core has built-in support for dependency injection.

Out of the box, ASP.NET Core has built-in support for dependency injection. If you are new to this powerful concept, please do not worry, we will discuss it in detail as we progress through this course.

Dependency injection is basically providing the objects that an object needs (its dependencies) instead of having it construct them itself. It's a very useful technique for testing, since it allows dependencies to be mocked or stubbed out.

Dependencies can be injected into objects by many means (such as constructor injection or setter injection).

Testability

- With built-in dependency injection and the unified programming model for creating Web Applications and Web API's, unit testing ASP.NET Core applications is straight forward.

With built-in dependency injection and the unified programming model for creating Web Applications and Web API's, unit testing ASP.NET Core applications is easy.

Open-source and community-focused

- <https://github.com/dotnet/core>
- ASP.NET Core is fully open source and is being actively developed by the .NET team in collaboration with a vast community of open source developers.
- ASP.NET core is continually evolving as the vast community behind it is suggesting ways to improve it and help fix bugs and problems.
- This means we have a more secure and better quality software.
- MIT Licence (Private and Commercial use)

ASP.NET Core is fully open source and is being actively developed by the .NET team in collaboration with a vast community of open source developers. So, ASP.NET core is continually evolving as the vast community behind it is suggesting ways to improve it and help fix bugs and problems. This means we have a more secure and better quality software.

Modular HTTP Request Pipeline

- ASP.NET Core Provides Modularity with Middleware Components in ASP.NET Core
- We compose the request and response pipeline using the middleware components.
- It includes a rich set of built-in middleware components.
- We can also write our own custom middleware components.

ASP.NET Core Provides Modularity with Middleware Components. In ASP.NET Core, we compose the request and response pipeline using the middleware components. It includes a rich set of built-in middleware components. We can also write our own custom middleware components. As we progress through the course we will be discussing, what middleware components are and using them to compose request and response pipeline.

.NET Core is designed in a modular approach. The framework splits up into a large list of NuGet packages. So that you don't have to deal with all the packages, metapackages are used that reference the smaller packages that work together.

Command line tool support

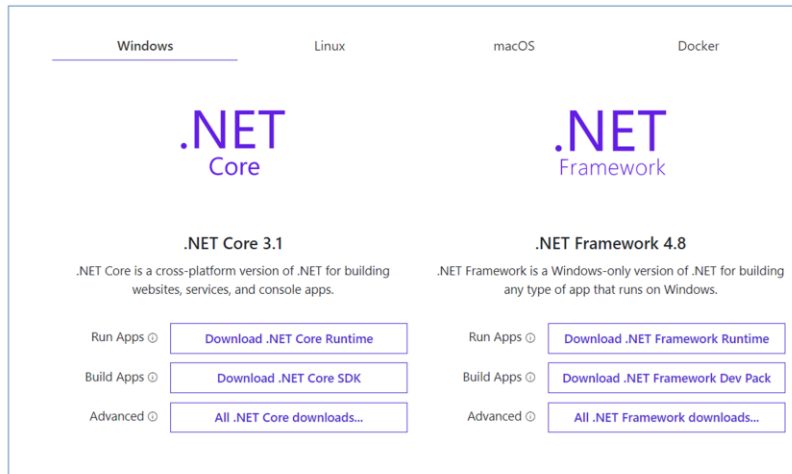
- .NET Core fully supports command line tool which is useful in complete cycle of development.
 - Create new project
 - Add package
 - Build
 - Run
 - Test
 - Deploy
 - Etc...

.NET Core CLI

- .NET CLI helps us to perform almost all the tasks which are required in order to work with .NET Core application.
- .NET CLI works with the command and these commands are applicable on all types of application of .NET Core.
- .NET CLI is a cross platform tool for developing .NET applications.

Download .NET

- <https://dotnet.microsoft.com/download>



Downloads for .NET Framework and .NET Core, including ASP.NET and ASP.NET Core For Windows, you can download an executable that installs the SDK.

With Linux, you need to select the Linux distribution to get the corresponding command:

With Red Hat and CentOS, install the .NET SDK using yum.

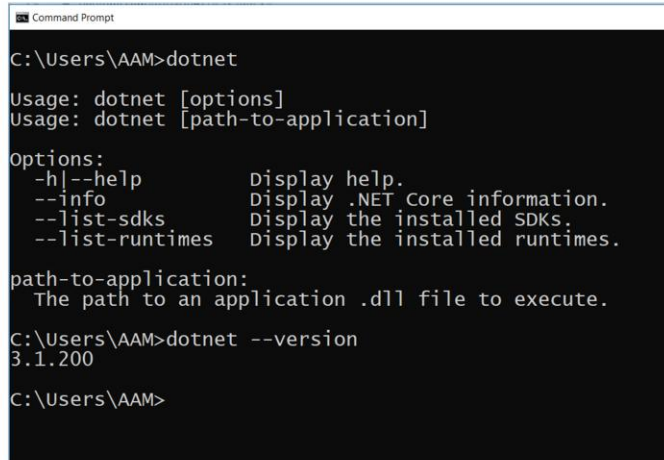
With Ubuntu and Debian, use apt-get.

With Fedora, use dnf install.

With SLES/openSUSE, use zipper install.

To install the .NET SDK on the Mac, you can download a .pkg file.

Let's Rain...



```
Command Prompt
C:\Users\AAM>dotnet

Usage: dotnet [options]
Usage: dotnet [path-to-application]

Options:
  -h|--help           Display help.
  --info              Display .NET Core information.
  --list-sdks          Display the installed SDKs.
  --list-runtimes      Display the installed runtimes.

path-to-application:
  The path to an application .dll file to execute.

C:\Users\AAM>dotnet --version
3.1.200

C:\Users\AAM>
```

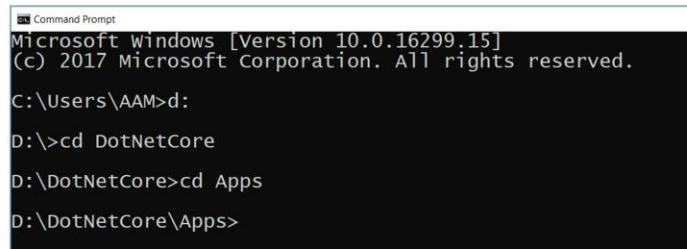
.NET Core Command Line Interface (CLI).

Installing .NET Core CLI tools, you have the dotnet tools as an entry point to start all these tools. Just start

➤ dotnet --help

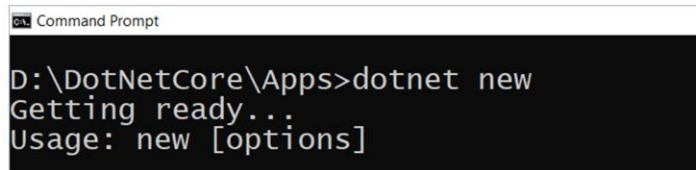
to see all the different options of the dotnet tools available. Many of the options have a shorthand notation. For help, you can type

➤ dotnet -h



```
Command Prompt
Microsoft Windows [Version 10.0.16299.15]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\AAM>d:
D:\>cd DotNetCore
D:\DotNetCore>cd Apps
D:\DotNetCore\Apps>
```



```
Command Prompt

D:\DotNetCore\Apps>dotnet new
Getting ready...
Usage: new [options]
```

Going to a directory, I want to create programs

Wait for templates

Templates	Short Name	Language

Console Application	console	[C#], F#, VB
Class library	classlib	[C#], F#, VB
WPF Application	wpf	[C#]
WPF Class library	wpflib	[C#]
WPF Custom Control Library	wpfcustomcontrollib	[C#]
WPF User Control Library	wpfusercontrollib	[C#]
Windows Forms (WinForms) Application	winforms	[C#]
Windows Forms (WinForms) Class library	winformslib	[C#]
Worker Service	worker	[C#]
Unit Test Project	mstest	[C#], F#, VB

Dotnet new command output


```
D:\DotNetCore\Apps>dotnet new console
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on D:\DotNetCore\Apps\Apps.csproj...
  Restore completed in 177.25 ms for D:\DotNetCore\Apps\Apps.csproj.

Restore succeeded.

D:\DotNetCore\Apps>dir
Volume in drive D is D
Volume Serial Number is 4684-394D

Directory of D:\DotNetCore\Apps

08-07-2020  04:11 PM    <DIR>          .
08-07-2020  04:11 PM    <DIR>          ..
08-07-2020  04:11 PM             178 Apps.csproj
08-07-2020  04:11 PM    <DIR>          obj
08-07-2020  04:11 PM             186 Program.cs
                2 File(s)              364 bytes
                3 Dir(s)  325,871,345,664 bytes free

D:\DotNetCore\Apps>
```

➤ dotnet new console --output HelloWorld

This command creates a new HelloWorld directory and adds the source code file Program.cs and the project file HelloWorld.csproj.

```
D:\DotNetCore\Apps>type Apps.csproj
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
</Project>
D:\DotNetCore\Apps>
```

What is inside Apps.csproj

With the project file, the `OutputType` defines the type of the output. With a console application, this is `Exe`. The `TargetFramework` specifies the framework and the version that is used to build the application. With the sample project, the application is built using .NET Core 3.1.

The `Sdk` attribute specifies the SDK that is used by the project. Microsoft ships two main SDKs: `Microsoft.NET.Sdk` for console applications, and `Microsoft.NET.Sdk.Web` for ASP.NET Core web applications.

```
D:\DotNetCore\Apps>type Program.cs
using System;

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

D:\DotNetCore\Apps>
```

What is inside program.cs

Since the 1970s, when Brian Kernighan and Dennis Ritchie wrote the book *The C Programming Language*, it's been a tradition to start learning programming languages using a "Hello World!" application. With the .NET Core CLI, this program is automatically generated.

Let's get into the syntax of this program. The Main method is the entry point for a .NET application. The CLR invokes a static Main method on startup. The Main method needs to be put into a class. Here, the class is named Program, but you could call it by any name.

Console.WriteLine invokes the WriteLine method of the Console class. You can find the Console class in the System namespace. You don't need to write System.Console.WriteLine to invoke this method; the System namespace is opened with the using declaration on top of the source file. After writing the source code, you need to compile the code to run it.

```
D:\DotNetCore\Apps>dotnet build
Microsoft (R) Build Engine version 16.5.0+d4cbfca49 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

Restore completed in 41.58 ms for D:\DotNetCore\Apps\Apps.csproj.
Apps -> D:\DotNetCore\Apps\bin\Debug\netcoreapp3.1\Apps.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

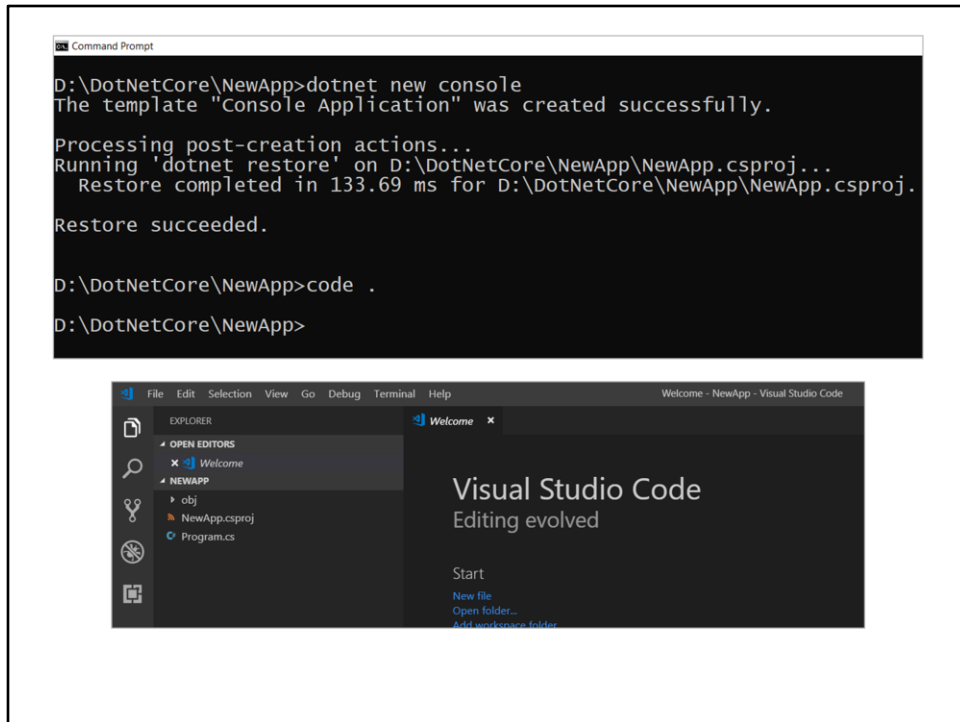
Time Elapsed 00:00:06.88

D:\DotNetCore\Apps>dotnet run
Hello world!

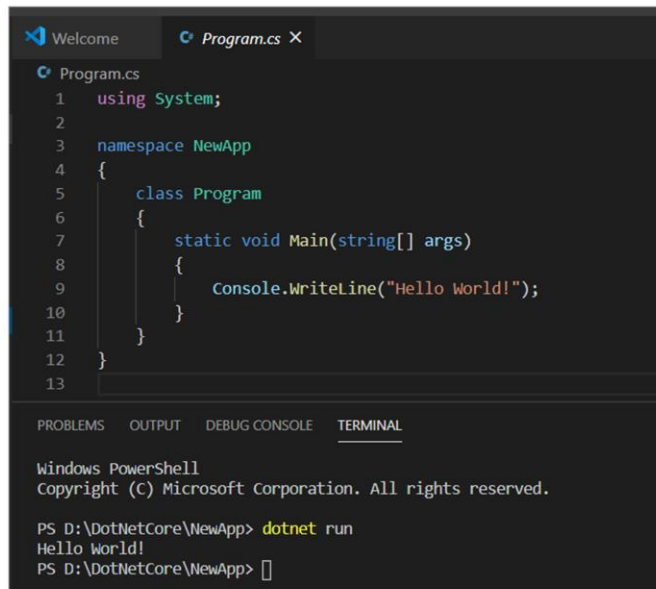
D:\DotNetCore\Apps>
```

Build and run the program

To build the application, you need to change the current directory to the directory of the application and start dotnet build.



Open everything in current directory in visual studio code editor



The image shows a screenshot of the Visual Studio Code editor. The top part displays a C# file named `Program.cs` with the following code:

```
1 using System;
2
3 namespace NewApp
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             Console.WriteLine("Hello World!");
10        }
11    }
12 }
13
```

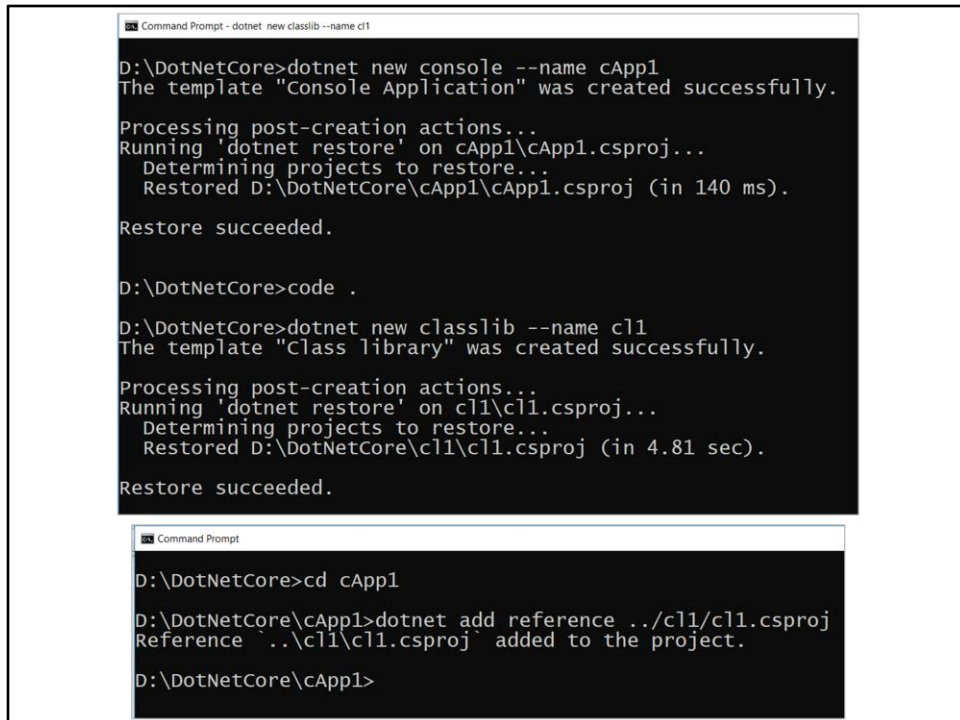
The bottom part of the screenshot shows the `TERMINAL` panel. It contains the following text:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS D:\DotNetCore\NewApp> dotnet run
Hello World!
PS D:\DotNetCore\NewApp> 
```

Run from terminal

Can also run using F5 in visual studio code



```
Command Prompt - dotnet new classlib --name c11

D:\DotNetCore>dotnet new console --name cApp1
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on cApp1\cApp1.csproj...
  Determining projects to restore...
  Restored D:\DotNetCore\cApp1\cApp1.csproj (in 140 ms).

Restore succeeded.

D:\DotNetCore>code .

D:\DotNetCore>dotnet new classlib --name c11
The template "Class library" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on c11\c11.csproj...
  Determining projects to restore...
  Restored D:\DotNetCore\c11\c11.csproj (in 4.81 sec).

Restore succeeded.

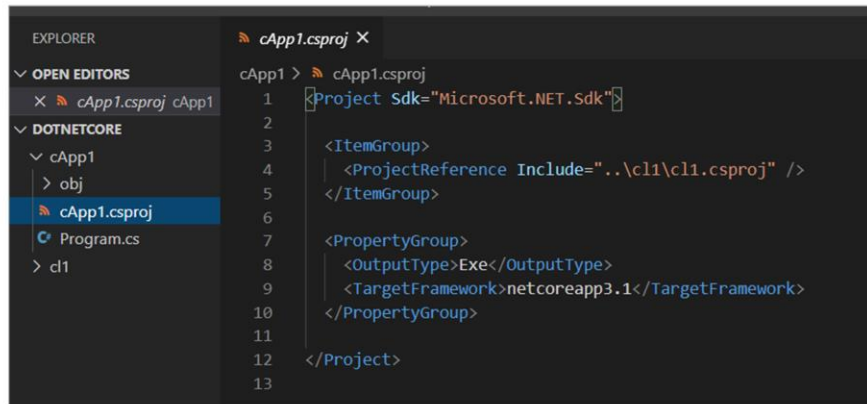
Command Prompt

D:\DotNetCore>cd cApp1

D:\DotNetCore\cApp1>dotnet add reference ../c11/c11.csproj
Reference '..\c11\c11.csproj' added to the project.

D:\DotNetCore\cApp1>
```

add project reference in .NET Core 3.0 using CLI



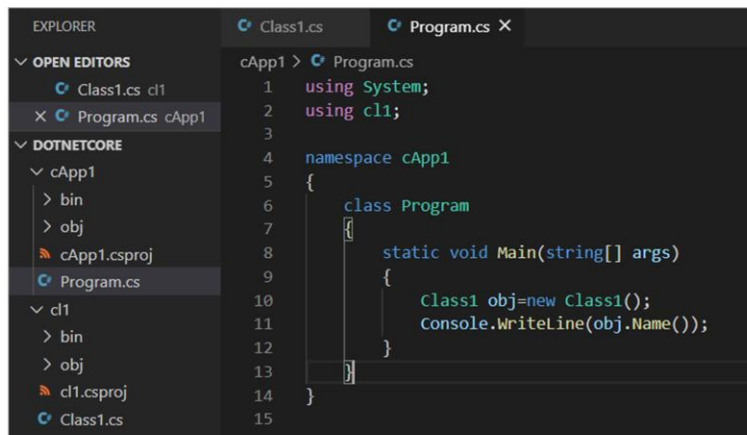
The screenshot displays the Visual Studio interface. On the left, the Explorer window shows a project structure with a folder named 'cApp1' containing subfolders 'obj' and 'cl1'. The 'cApp1.csproj' file is selected. The main editor window shows the content of 'cApp1.csproj', which is an XML file defining a .NET Core project. The XML includes the SDK, a project reference to 'cl1.csproj', and properties for the output type and target framework.

```
1 <Project Sdk="Microsoft.NET.Sdk">
2
3   <ItemGroup>
4     <ProjectReference Include="..\cl1\cl1.csproj" />
5   </ItemGroup>
6
7   <PropertyGroup>
8     <OutputType>Exe</OutputType>
9     <TargetFramework>netcoreapp3.1</TargetFramework>
10  </PropertyGroup>
11
12 </Project>
13
```



```
EXPLORER
v OPEN EDITORS
  x Class1.cs c1
v DOTNETCORE
  v cApp1
    > obj
    cApp1.csproj
    Program.cs
  v c1
    > obj
    c1.csproj
    Class1.cs

Class1.cs
1  using System;
2
3  namespace c1
4  {
5      public class Class1
6      {
7          public string Name()
8          {
9              return "DDU";
10         }
11     }
12 }
13 |
```



The screenshot shows the Visual Studio IDE. On the left, the 'EXPLORER' pane displays the project structure under 'DOTNETCORE'. The 'cApp1' folder contains 'bin', 'obj', 'cApp1.csproj', and 'Program.cs'. The 'c1' folder contains 'bin', 'obj', 'c1.csproj', and 'Class1.cs'. The 'Program.cs' file is open in the editor, showing the following code:

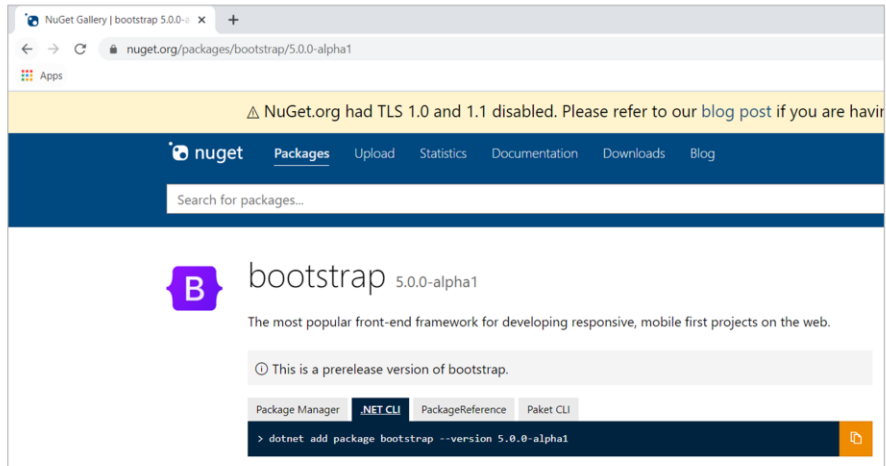
```
1 using System;
2 using c1;
3
4 namespace cApp1
5 {
6     class Program
7     {
8         static void Main(string[] args)
9         {
10             Class1 obj=new Class1();
11             Console.WriteLine(obj.Name());
12         }
13     }
14 }
15
```

```
D:\DotNetCore\cApp1>dotnet run
DDU
D:\DotNetCore\cApp1>
```

In similar manner references can be added w/o going into specific directory. Even we can add multiple references as well in a single command.

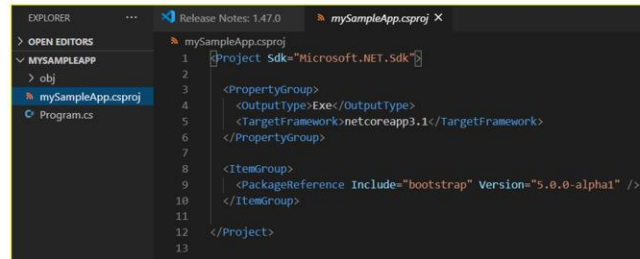
```
D:\DotNetCore>dotnet new console --name mySampleApp
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on mySampleApp\mySampleApp.csproj...
  Determining projects to restore...
  Restored D:\DotNetCore\mySampleApp\mySampleApp.csproj (in 145 ms).
Restore succeeded.
```



Add package from nuget to an existing application

```
D:\DotNetCore>cd mySampleApp  
D:\DotNetCore\mySampleApp>dotnet add package bootstrap --version 5.0.0-alpha1
```



.NET Core vs .NET Framework^{server apps}

- There are two supported .NET implementations for building server-side apps
 - .NET Framework
 - .NET Core.
 - Both share many of the same components and you can share code across the two.
 - However, there are fundamental differences between the two and your choice depends on what you want to accomplish.

<https://docs.microsoft.com/en-us/dotnet/standard/choosing-core-framework-server>

.NET Core^{When}

- You have cross-platform needs.
- You're targeting microservices.
- You're using Docker containers.
- You need high-performance and scalable systems.
- You need side-by-side .NET versions per application.

.NET Framework^{When}

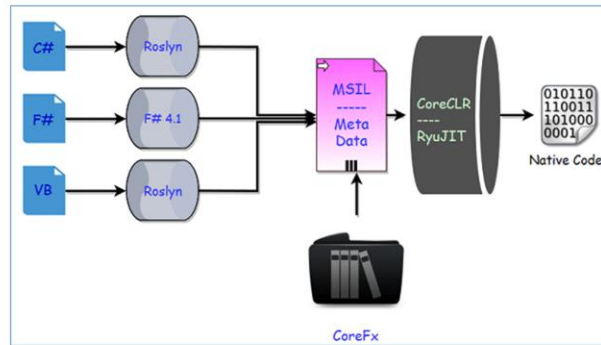
- Your app currently uses .NET Framework (recommendation is to extend instead of migrating).
- Your app uses third-party .NET libraries or NuGet packages not available for .NET Core.
- Your app uses .NET technologies that aren't available for .NET Core.
- Your app uses a platform that doesn't support .NET Core.
 - Windows, macOS, and Linux support .NET Core.

What is the difference between SDK and Runtime in .NET Core?

The SDK is all of the stuff that is needed/makes developing a .NET Core application easier, such as the CLI and a compiler.

The runtime is the "virtual machine" that hosts/runs the application and abstracts all the interaction with the base operating system.

What is CoreCLR?



CoreCLR provides the common language runtime environment for .NET Core applications, and manages the execution of the complete application life cycle. It performs various operations when the program is running. Operations such as memory allocation, garbage collection, exception handling, type safety, thread management, and security are part of CoreCLR.

.NET Core's runtime provides the same **Garbage Collection (GC)** as .NET Framework and a new **Just In Time (JIT)** compiler that is more optimized, codenamed *RyuJIT*.
NET Core Libraries (CoreFX)

[RyuJIT](#) is the code name for the .NET just-in-time compiler, one of the foundational components of the .NET runtime. In contrast, the [Roslyn C# compiler](#) compiles C# code to IL byte code. The RyuJIT compiler compiles IL byte code to machine code for multiple processors.

In .NET Core now we have a new series of compilers, like we have Roslyn for C# and VB.

You can also make use of the new F# 4.1 compiler if you want to use F# with .NET Core.

Actually these tools are different and we can use Roslyn with .NET Framework as well if we are using C# 6 or later, because C# compiler can only support up to C# 5.

In .NET Core, we don't have a framework class libraries (FCL), so a different set of libraries are used and we now have CoreFx.

CoreFx is the reimplement of the class libraries for .NET Core.

We also have a new run time with .NET Core known as CoreCLR and leverages a JIT Compiler.

https://www.tutorialspoint.com/dotnet_core/dotnet_core_code_execution.htm