CSc 346
Object Oriented Programming

Ken Gamradt

Spring 2024

Chapter 1

Using Visual Studio Code for cross-platform development

1. The most modern and lightweight code editor to choose from, and the only one from Microsoft that is cross-platform, is Microsoft Visual Studio Code
   - It can run on all common operating systems, including Windows, macOS, and many varieties of Linux, including Red Hat Enterprise Linux (RHEL) and Ubuntu
2. Visual Studio Code is a good choice for modern cross-platform development because it has an extensive and growing set of extensions to support many languages beyond C#
   - The most important extension for C# and .NET developers is the C# Dev Kit that was released in preview in June 2023 because it turns Visual Studio Code from a general-purpose code editor into a tool optimized for C# and .NET developers

Using Visual Studio Code for cross-platform development

3. Being cross-platform and lightweight, it can be installed on all platforms that your apps will be deployed to for quick bug fixes and so on
   - Choosing Visual Studio Code means a developer can use a cross-platform code editor to develop cross-platform apps
   - Visual Studio Code is supported on ARM processors so that you can develop on Apple Silicon computers and Raspberry Pi computers
4. Visual Studio Code has strong support for web development, although it currently has weak support for mobile and desktop development
5. Visual Studio Code is by far the most popular code editor or IDE, with over 73% of professional developers selecting it in the Stack Overflow 2023 survey that you can read at the following link: https:// survey.stackoverflow.co/2023/

Using GitHub Codespaces for development in the cloud

- GitHub Codespaces is a fully configured development environment based on Visual Studio Code that can be spun up in an environment hosted in the cloud and accessed through any web browser
- It supports
  - Git repos
  - Extensions
  - Built-in command-line interface so you can edit, run, and test from any device

What I use

- I use the following hardware:
  - Apple iMac (Intel) desktop
  - Apple MacBook Pro (Apple Silicon – M1 Pro) laptop
- I use the following software:
  - Visual Studio Code on:
    - macOS on an Apple desktop (Intel) and laptop (M1)
    - Windows 11 ARM virtual machine running with Parallels

Deploying cross-platform

- .NET 8 supports the following platforms for deployment:
  - **Windows**:
    - Windows 10 version 1607 or later
    - Windows 11 version 22000 or later
  - **Mac**:
    - macOS Catalina (version 10.15) or later
  - **Linux**
    - Alpine 3.17 or later, Fedora 37 or later, openSUSE 15 or later, Ubuntu 20.04 or later
    - Debian 11 or later, Oracle 8 or later, RHEL 8 or later, SUSE Enterprise 12 SP2 or later
  - **Android**:
    - API 21 or later
  - **iOS**:
    - 10.0 or later

Deploying cross-platform

- Windows ARM64 support in .NET 5 and later means you can develop on, and deploy to, Windows ARM devices like Microsoft Surface Pro X
- Developing on an Apple M1 Mac using Parallels and a Windows 11 ARM virtual machine is reportedly twice as fast!

Download and install VS Code

1. Download and install either the Stable build or the Insiders edition of Visual Studio Code from the following link: https://code.visualstudio.com/
2. Download and install the .NET SDK for version 8.0 and at least one other version like 6.0 or 7.0 from the following link: https://www.microsoft.com/net/download
3. To install the C# Dev Kit extension with a user interface, you must first launch the Visual Studio Code application
4. In Visual Studio Code, click the Extensions icon or navigate to View | Extensions
5. C# Dev Kit is one of the most popular extensions available, so you should see it at the top of the list, or you can enter C# in the search box
6. Click Install and wait for supporting packages to download and install

Download and install VS Code

Extension name and identifier	Description
C# Dev Kit `ms-dotnettools.csdevkit`	Official C# extension from Microsoft. Manage your code with a solution explorer and test your code with integrated unit test discovery and execution. Includes the **C#** and **IntelliCode for C# Dev Kit** extensions.
C# `ms-dotnettools.csharp`	C# editing support, including syntax highlighting, IntelliSense, Go To Definition, Find All References, debugging support for .NET, and support for `csproj` projects on Windows, macOS, and Linux.
IntelliCode for C# Dev Kit `ms-dotnettools.vscodeintellicode-csharp`	Provides AI-assisted development features for Python, TypeScript/JavaScript, C#, and Java developers.
MSBuild project tools `tintoy.msbuild-project-tools`	Provides IntelliSense for MSBuild project files, including autocomplete for `<PackageReference>` elements.
Polyglot Notebooks `ms-dotnettools.dotnet-interactive-vscode`	This extension adds support for using .NET and other languages in a notebook. It has a dependency on the **Jupyter** extension (`ms-toolsai.jupyter`), which itself has dependencies.
ilspy-vscode `icsharpcode.ilspy-vscode`	Decompile MSIL assemblies – support for modern .NET, .NET Framework, .NET Core, and .NET Standard.
REST Client `humao.rest-client`	Send an HTTP request and view the response directly in Visual Studio Code.

Understanding .NET support

- **Long Term Support (LTS)**
  - Releases are a good choice for applications that you do not intend to update frequently, although you must update the .NET runtime for your production code monthly
  - LTS releases are supported by Microsoft for
    - 3 years after General Availability (GA)
    - 1 year after the next LTS release ships
    - whichever is longer

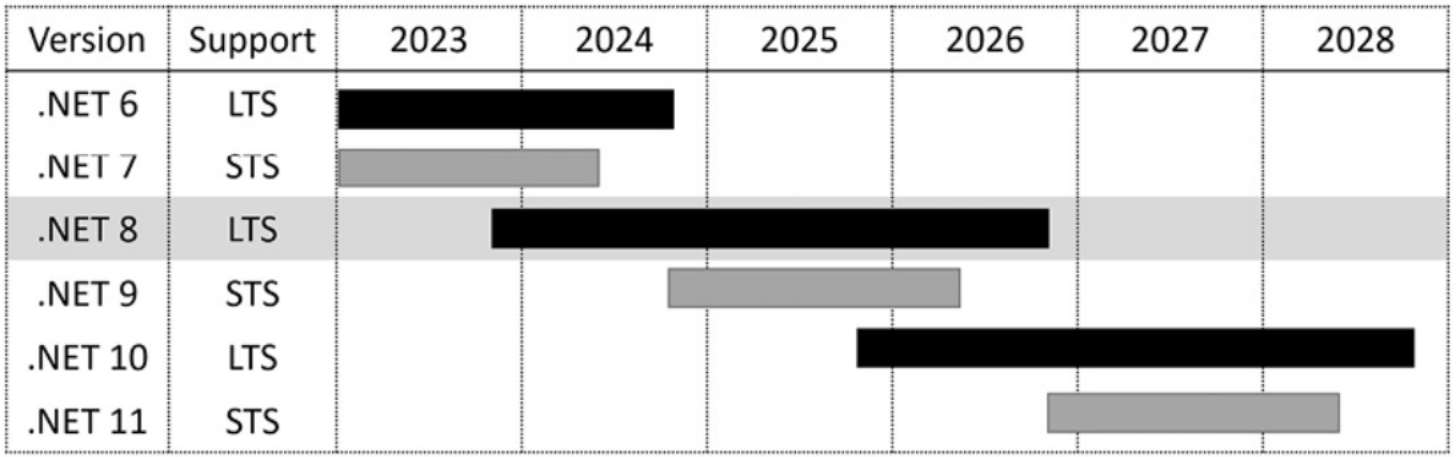
Understanding .NET support

- **Standard Term Support (STS) – formerly Current**
  - Releases include features that may change based on feedback
  - These are a good choice for applications that you are actively developing because they provide access to the latest improvements
  - STS releases are supported by Microsoft for
    - 18 months after GA
    - 6 months after the next STS or LTS release ships
    - whichever is longer

Understanding .NET support

- **Preview**
  - Releases are for public testing
  - These are a good choice for adventurous programmers who want to
    - live on the bleeding edge
    - programming book authors who need to have early access to new language features, libraries, and app and service platforms
  - These not usually supported by Microsoft, but some preview or Release Candidate (RC) releases may be declared Go Live
    - meaning they are supported by Microsoft in production

Understanding .NET support



Understanding .NET support

- All versions of modern .NET have reached their end of life except those shown in the following list that are ordered by their end-of-life dates:
  - .NET 7.0 – end of life 05-14-24
  - .NET 6.0 – end of life 11-12-24
  - .NET 8.0 – end of life 11-10-26

Understanding .NET Runtime and .NET SDK versions

- If you have not built a standalone app, then the .NET Runtime is the minimum needed to install so that an operating system can run a .NET application
- The .NET SDK includes the .NET Runtime as well as the compilers and other tools needed to build .NET code and apps
- .NET Runtime versioning follows semantic versioning, that is, a major increment indicates breaking changes, minor increments indicate new features, and patch increments indicate bug fixes
- .NET SDK versioning does not follow semantic versioning
- The major and minor version numbers are tied to the runtime version it is matched with
- The patch number follows a convention that indicates the major and minor versions of the SDK.

Understanding .NET Runtime and .NET SDK versions

Change	Runtime	SDK
Initial release	8.0.0	8.0.100
SDK bug fix	8.0.0	8.0.101
Runtime and SDK bug fix	8.0.1	8.0.102
SDK new feature	8.0.1	8.0.200

Listing and removing versions of .NET

- .NET Runtime updates are compatible with a major version such as 8.x, and updated releases of the .NET SDK maintain the ability to build applications that target previous versions of the runtime, which enables the safe removal of older versions
- You can see which SDKs and runtimes are currently installed using the following commands:

```
dotnet --list-sdks
dotnet --list-runtimes
dotnet --info
```

Understanding intermediate language

- The C# compiler (named **Roslyn**) used by the dotnet CLI tool converts your C# source code into **intermediate language** (**IL**) code and stores the IL in an **assembly** (a DLL or EXE file)
- IL code statements are like assembly language instructions, which are executed by .NET's virtual machine, known as **CoreCLR**
- At runtime, CoreCLR loads the IL code from the assembly, the **just-in-time** (**JIT**) compiler compiles it into native CPU instructions, and then it is executed by the CPU on your machine
- The benefit of this two-step compilation process is that Microsoft can create CLRs for Linux and macOS, as well as for Windows
- The same IL code runs everywhere because of the second compilation step, which generates code for the native operating system and CPU instruction set

Comparing .NET technologies

Technology	Description	Host operating systems
Modern .NET	A modern feature set, with full C# 8 to C# 12 language support. It can be used to port existing apps or create new desktop, mobile, and web apps and services. It can target older .NET platforms.	Windows, macOS, Linux, Android, iOS, tvOS, Tizen
.NET Framework	A legacy feature set with limited C# 8 support and no C# 9 or later support. It should be used to maintain existing applications only.	Windows only
Xamarin	Mobile and desktop apps only.	Android, iOS, macOS

Understanding top-level programs

- If you have seen older .NET projects before, then you might have expected more code, even just to output a simple message
- This project has minimal statements because some of the required code is written for you by the compiler when you target .NET 6 or later
- If you had created the project with .NET SDK 5 or earlier, or if you had selected the check box labeled **Do not use top-level statements**, then the Program.cs file would have more statements, as shown in the following code:

```
using System;

namespace HelloCS {

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

}
```

```
using System;

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

Writing top-level programs

- During compilation with .NET SDK 6.0 or later, all the boilerplate code to define a namespace, the Program class, and its Main method, is generated and wrapped around the statements you write
- This uses a feature introduced in .NET 5 called **top-level programs**, but it was not until .NET 6 that Microsoft updated the project template for console apps to use it by default
- Then in .NET 7 or later, Microsoft added options to use the older style if you prefer:

```
dotnet new console --use-program-main
```

Writing top-level programs

- Key points to remember about top-level programs include the following list:
  - There can be only one file like this in a project
  - Any using statements still must go at the top of the file
  - If you declare any classes or other types, they must be at the bottom of the file
  - Although you should name the method Main if you explicitly define it, the method is named <Main>$ when created by the compiler

Implicitly imported namespaces

- The trick is that we do still need to import the System namespace, but it is now done for us using a combination of features introduced in C# 10 and .NET 6
- In Solution Explorer, expand the obj folder, expand the Debug folder, expand the net8.0 folder, and open the file named **HelloCS.GlobalUsings.g.cs**
- The following is automatically created by the compiler for projects that target .NET 6 or later, and that it uses a feature introduced in C# 10 called **global namespace imports**, which imports some commonly used namespaces like **System** for use in all code files

```
// <autogenerated />
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.Net.Http;
global using global::System.Threading;
global using global::System.Threading.Tasks;
```

Practice C# anywhere

- You can go to .NET Fiddle – https://dotnetfiddle.net/ – and start coding online