CSc 346
Object Oriented Programming

Ken Gamradt

Spring 2024

Chapter 2

Discovering your C# compiler versions

- The .NET language compiler for C# and Visual Basic, also known as **Roslyn**, along with a separate compiler for F#, is distributed as part of the .NET SDK
- To use a specific version of C#, you must have at least that version of the .NET SDK installed dotnet --version

.NET SDK	Roslyn compiler	Default C# language
1.0.4	2.0-2.2	7.0
1.1.4	2.3-2.4	7.1
2.1.2	2.6-2.7	7.2
2.1.200	2.8-2.10	7.3
3.0	3.0-3.4	8.0
5.0	3.8	9.0
6.0	4.0	10.0
7.0	4.4	11.0
8.0	4.8	12.0

Enabling a specific language version compiler

- Developer tools assume that you want to use the latest major version of a C# language compiler dy default
- To use the improvements in a C# point release like 7.1, 7.2, or 7.3, you had to add a **<LangVersion>** configuration element to the project file
- After the release of C# 12 with .NET 8, if Microsoft releases a C# 12.1 compiler and you want to use its new language features, then you will have to add a configuration element to your project file, as shown in the following markup

<LangVersion>	Description
`7`, `7.1`, `7.2`, `7.3`, `8`, `9`, `10`, `11`, `12`	Entering a specific version number will use that compiler if it has been installed.
`latestmajor`	Uses the highest major number, for example, 7.0 in August 2019, 8 in October 2019, 9 in November 2020, 10 in November 2021, 11 in November 2022, and 12 in November 2023.
`latest`	Uses the highest major and highest minor number, for example, 7.2 in 2017, 7.3 in 2018, 8 in 2019, and perhaps 12.1 in H1 2024.
`preview`	Uses the highest available preview version, for example, 12.0 in July 2023 with .NET 8 Preview 6 installed.

Enabling a specific language version compiler

Gamradt1.csproj

Gamradt1 > Gamradt1.csproj

```
1   <Project Sdk="Microsoft.NET.Sdk">
2
3     <PropertyGroup>
4       <OutputType>Exe</OutputType>
5       <TargetFramework>net8.0</TargetFramework>
6       <ImplicitUsings>enable</ImplicitUsings>
7       <Nullable>enable</Nullable>
8       <LangVersion>latestmajor</LangVersion>
9     </PropertyGroup>
10
11  </Project>
12
```

Importing namespaces

- **System** is a namespace, which is like an address for a type
- **System.Console.WriteLine** tells the compiler to look for a method named WriteLine in a type named Console in a namespace named System
- To simplify our code, the **Console Application** project template for every version of .NET before 6.0 added a statement at the top of the code file to tell the compiler to always look in the System namespace for types that haven't been prefixed with their namespace, as shown in the following code:

```
using System;              // import the System namespace
```

- We call this **importing the namespace**
- The effect of importing a namespace is that all available types in that namespace will be available to your program without needing to enter the namespace prefix and will be seen in IntelliSense while you write code

Implicitly and globally importing namespaces

- Traditionally, every **.cs** file that needs to import namespaces would have to start with **using** statements to import those namespaces
- Namespaces like **System** and **System.Linq** are needed in almost all **.cs** files, so the first few lines of every **.cs** file often had at least a few **using** statements, as shown in the following code:

```
using System;
using System.Linq;
using System.Collections.Generic;
```

- When creating websites and services using ASP.NET Core, there are often dozens of namespaces that each file would have to import

Implicitly and globally importing namespaces

- C# 10 introduces some new features that simplify importing namespaces
- The **global using** statement means you only need to import a namespace in one .cs file and it will be available throughout all .cs files
- You could put global using statements in the Program.cs file but I recommend creating a separate file for those statements named something like GlobalUsings.cs or GlobalNamespaces.cs, as shown in the following code:

```
global using System;
global using System.Linq;
global sing System.Collections.Generic;
```

- Projects that target .NET 6.0 or later, and use the C# 10 compiler or later, generate a **<ProjectName>.GlobalUsings.g.cs** file in the **obj\Debug\net8.0** folder to implicitly globally import some common namespaces like **System**

Implicitly and globally importing namespaces

SDK	Implicitly imported namespaces

`Microsoft.NET.Sdk`	`System`
	`System.Collections.Generic`
	`System.IO`
	`System.Linq`
	`System.Net.Http`
	`System.Threading`
	`System.Threading.Tasks`

`Microsoft.NET.Sdk.Web`	Same as `Microsoft.NET.Sdk` and:
	`System.Net.Http.Json`
	`Microsoft.AspNetCore.Builder`
	`Microsoft.AspNetCore.Hosting`
	`Microsoft.AspNetCore.Http`
	`Microsoft.AspNetCore.Routing`
	`Microsoft.Extensions.Configuration`
	`Microsoft.Extensions.DependencyInjection`
	`Microsoft.Extensions.Hosting`
	`Microsoft.Extensions.Logging`

`Microsoft.NET.Sdk.Worker`	Same as `Microsoft.NET.Sdk` and:
	`Microsoft.Extensions.Configuration`
	`Microsoft.Extensions.DependencyInjection`
	`Microsoft.Extensions.Hosting`
	`Microsoft.Extensions.Logging`

Verbs are methods

- In English, verbs are doing or action words, like run and jump
- In C#, doing or action words are called **methods**
- There are hundreds of thousands of methods available to C#
- In C#, methods such as **WriteLine** change how they are called or executed based on the specifics of the action
- This is called overloading, which we'll cover in more detail in Chapter 5
  - Building Your Own Types with Object-Oriented Programming

Nouns are types, variables, fields, and properties

- In English, nouns are names that refer to things
  - E.g., Fido is the name of a dog
- The word "dog" tells us the type of thing that Fido is, and so in order for Fido to fetch a ball, we would use his name
- In C#, their equivalents are **types**, **variables**, **fields**, and **properties**
  - E.g., **Animal** and **Car** are types
  - They are nouns for categorizing things
  - **Head** and **Engine** might be fields or properties
    - Nouns that belong to **Animal** and **Car**
  - Fido and **Bob** are variables
    - Nouns for referring to a specific object

Nouns are types, variables, fields, and properties

- There are tens of thousands of types available to C#
- The language of C# only has a few keywords for types, such as **string** and **int**
- Keywords such as string that look like types are **aliases**
  - Which represent types provided by the platform on which C# runs
- C# cannot exist alone, it's a language that runs on variants of .NET
- Someone could write a compiler for C# that uses a different platform, with different underlying types
- The platform for C# is .NET, which provides tens of thousands of types to C#
- It's worth taking note that the term **type** is often confused with **class**
- In C#, every **type** can be a **class**, **struct**, **enum**, **interface**, or **delegate**
  - The C# keyword **string** is a **class**, but **int** is a **struct**.
- It is best to use the term **type** to refer to both

Working with variables

- All applications process data
- Data comes in, data is processed, and then data goes out
- Data comes into our program from files, databases, or user input
- Data is temporarily stored in variables, which are are stored in memory
- When the program ends, the data in memory is lost
- Data is usually output to files and databases, or to the screen or a printer
- When using variables, you should think about
  - how much space the variable takes up in the memory
  - how fast it can be processed
- We control this by picking an appropriate type
- Common types such as int and double are different-sized storage boxes where a
  - smaller box would take less memory
  - smaller box may not be as fast at being processed

Naming things

Naming convention	Examples	Used for
Camel case	`cost`, `orderDetail`, and `dateOfBirth`	Local variables, private fields.
Title case aka Pascal case	`String`, `Int32`, `Cost`, `DateOfBirth`, and `Run`	Types, non-private fields, and other members like methods.

- **Good Practice**: Following a consistent set of naming conventions will enable your code to be easily understood by other developers
  - Yourself in the future!

Storing text

- **Literal string**: Characters enclosed in double-quote characters
  - They can use escape characters like \t for tab
  - To represent a backslash, use two: \\
- **Raw string literal**: Characters enclosed in three or more double-quote characters
- **Verbatim string**: A literal string prefixed with @ to disable escape characters so that a backslash is a backslash
  - It also allows the string value to span multiple lines because the whitespace characters are treated as themselves instead of instructions to the compiler
- **Interpolated string**: A literal string prefixed with $ to enable embedded formatted variables

Storing dynamic types

- There is another special type named **dynamic** that can also store any type of data, even more than **object**, its flexibility comes at the cost of performance
- The **dynamic** keyword was introduced in C# 4.0
- Unlike **object**, the value stored in the variable can have its members invoked without an explicit cast

```
// storing a string in a dynamic object
// string has a Length property
dynamic something = "Ahmed";
// This compiles but might throw an exception at run-time
Console.WriteLine($"The length of something is {something.length}");
// Output the type of the something variable
Console.WriteLine($ "something is a {something.GetType()}");
```

Storing dynamic types

- One limitation of **dynamic** is that code editors cannot show **IntelliSense** to help you write the code
- This is because the compiler cannot check what the type is during build time
- Instead, the CLR checks for the member at runtime and throws an exception if it is missing
- Exceptions are a way to indicate that something has gone wrong at runtime
- Dynamic types are most useful when interoperating with non-.NET systems
- For example, you might need to work with a class library written in F#, Python, or some JavaScript
- You might also need to interop with technologies like the **Component Object Model** (**COM**), for example, when automating Excel or Word

Inferring the type of a local variable

- You can use the **var** keyword to declare local variables
- The compiler will infer the type from the value that you assign after the assignment operator, =
- A literal number without a decimal point is inferred as an **int** variable
- A literal number with a decimal point is inferred as double unless you add the
  - M suffix, in which case, it infers a decimal variable
  - F suffix, in which case, it infers a float variable

- L: Compiler infers `long`
- UL: Compiler infers `ulong`
- M: Compiler infers `decimal`
- D: Compiler infers `double`
- F: Compiler infers `float`

Understanding format strings

- A variable or expression can be formatted using a format string after a comma or colon
- An No (Nzero) format string means a number with thousand separators and no decimal places, while a C format string means currency
- The currency format will be determined by the current thread
- For instance, if you run code that uses the number or currency format on a PC
  - in the UK, you'll get pounds sterling with commas as the thousand separators
  - in Germany, you will get euros with dots as the thousand separators
- The full syntax of a format item is:

{ index [, alignment] [: formatString] }

Understanding format strings

```
string applesText = "Apples";
int applesCount = 1234;
string bananasText = "Bananas";
int bananasCount = 56789;

Console.WriteLine();
Console.WriteLine(format: "{0,-10} {1,6}", arg0: "Name", arg1: "Count");
Console.WriteLine(format: "{0,-10} {1,6:N0}", arg0: applesText, arg1: applesCount);
Console.WriteLine(format: "{0,-10} {1,6:N0}", arg0: bananasText, arg1: bananasCount);
```

```
Name          Count
Apples        1,234
Bananas      56,789
```

Understanding format strings

Format code	Description
`0`	Zero placeholder. Replaces the zero with the corresponding digit if present; otherwise, it uses zero. For example, `0000.00` formatting the value `123.4` would give `0123.40`.
`#`	Digit placeholder. Replaces the hash with the corresponding digit if present; otherwise, it uses nothing. For example, `####.##` formatting the value `123.4` would give `123.4`.
`.`	Decimal point. Sets the location of the decimal point in the number. Respects culture formatting, so it is a `.` (dot) in US English but a `,` (comma) in French.
`,`	Group separator. Inserts a localized group separator between each group. For example, `0,000` formatting the value `1234567` would give `1,234,567`. Also used to scale a number by dividing by multiples of 1,000 for each comma. For example, `0.00,,` formatting the value `1234567` would give `1.23` because the two commas mean divide by 1,000 twice.
`%`	Percentage placeholder. Multiplies the value by 100 and adds a percentage character.
`\`	Escape character. Makes the next character a literal instead of a format code. For example, `\##,###\#` formatting the value `1234` would give `#1,234#`.
`;`	Section separator. Defines different format strings for positive, negative, and zero numbers. For example, `[0];(0);Zero` formatting: `13` would give `[13]`, `-13` would give `(13)`, and `0` would give `Zero`.
Others	All other characters are shown in the output as is.

Understanding format strings

Format code	Description
C or c	Currency. For example, in US culture, C formatting the value 123.4 gives $123.40, and C0 formatting the value 123.4 gives $123.
N or n	Number. Integer digits with an optional negative sign and grouping characters.
D or d	Decimal. Integer digits with an optional negative sign but no grouping characters.
B or b	Binary. For example, B formatting the value 13 gives 1101 and B8 formatting the value 13 gives 00001101.
X or x	Hexadecimal. For example, X formatting the value 255 gives FF and X4 formatting the value 255 gives 00FF.
E or e	Exponential notation. For example, E formatting the value 1234.567 would give 1.234567000E+003 and E2 formatting the value 1234.567 would give 1.23E+003.

Passing arguments to a console app

- You might have been wondering how to get any arguments that might be passed to a console application
- In every version of .NET prior to version 6.0, the console application project template made it obvious:

```
using System;  // Not needed in .NET 6 or later

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

Passing arguments to a console app

- When you run a console app, you often want to change its behavior by passing arguments
- E.g., with the dotnet command-line tool, you can pass the name of a new project template

```
dotnet new console
dotnet new mvc
```

Passing arguments to a console app

- The **string[] args** arguments are declared and passed in the **Main** method of the **Program** class
- They're an array used to pass arguments into a console application
- In top-level programs, as used by the console application project template in .NET 6.0 and later, the Program class and its Main method are hidden, along with the declaration of the **args** string array
- The trick is knowing that it still exists
- Command-line arguments are separated by spaces
  - Characters like hyphens and colons are treated as part of an argument value

```
dotnet run firstarg second-arg third:arg "fourth arg"
```

Handling platforms that do not support an API

- **API:** Application Programmer Interface

```
try {
  CursorSize = int.Parse(args[2]);
}
catch (PlatformNotSupportedException) {
  WriteLine("The current platform does not support changing the size of the cursor.");
}


if (OperatingSystem.IsWindows()) {
  // execute code that only works on Windows
}
else if (OperatingSystem.IsWindowsVersionAtLeast(major: 10)) {
  // execute code that only works on Windows 10 or later
}
else if (OperatingSystem.IsIOSVersionAtLeast(major: 14, minor: 5)) {
  // execute code that only works on iOS 14.5 or later
}
else if (OperatingSystem.IsBrowser()) {
  // execute code that only works in the browser with Blazor
}
```

Handling platforms that do not support an API

- **API:** Application Programmer Interface

```
#if NET8_0_ANDROID
// Compile statements that only work on Android
#elif NET8_0_IOS
// Compile statements that only work on iOS
#else
// Compile statements that work everywhere else
#endif
```

Data Types

Microsoft Visual Studio Debug Console

Type	Byte(s) of memory	Min	Max
sbyte	1	-128	127
byte	1	0	255
short	2	-32768	32767
ushort	2	0	65535
int	4	-2147483648	2147483647
uint	4	0	4294967295
long	8	-9223372036854775808	9223372036854775807
ulong	8	0	18446744073709551615
Int128	16	-170141183460469231731687303715884105728	170141183460469231731687303715884105727
UInt128	16	0	340282366920938463463374607431768211455
Half	2	-65500	65500
float	4	-3.4028235E+38	3.4028235E+38
double	8	-1.7976931348623157E+308	1.7976931348623157E+308
decimal	16	-79228162514264337593543950335	79228162514264337593543950335