Chapter 1 : Hello, C#! welcome, .NET!

Understanding .NET

- NET Framework is a development platform that includes a Common Language Runtime (CLR), which manages the execution of code, and a Base Class Library (BCL), which provides a rich library of classes to build applications from.
- All of the apps on a computer written for the .NET Framework share the same version of the CLR and libraries stored in the Global Assembly Cache (GAC), which can lead to issues if some of them need a specific version for compatibility.
- NET Core includes a cross-platform implementation of the CLR known as `CoreCLR` and a streamlined library of classes known as `CoreFX`.

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 Core'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.
- Another .NET initiative is called .NET Native. This compiles C# code to native CPU instructions ahead of time (`AoT`), rather than using the CLR to compile IL code JIT to native code later. .NET Native improves execution speed and reduces the memory footprint for applications because the native code is generated at build time and then deployed instead of the IL code.

Chapter 2

Comments : // or /* */

Naming Convention	Examples	Use for
camel Case	`cost`, `orderDetail`, `dateOfBirth`	Local variables, private fields.
Pascal Case	`String`, `Int32`, `Cost`, `DateOfBirth`, `Run`	Types, non-private fields, and other members like methods.

- **Literal string**: Characters enclosed in double-quote characters. They can use escape characters like \t for tab.
- **Verbatim string**: A literal string prefixed with @ to disable escape characters so that a backslash is a backslash. (`@"\t"` will show `\t` and not a tab)
- **Interpolated string**: A literal string prefixed with $ to enable embedded formatted variables. You will learn more about this later in this chapter.

- **decimal Type** : The decimal type is accurate because it stores the number as a large integer and shifts the decimal point. For example, 0.1 is stored as 1, with a note to shift the decimal point one place to the left. 12.75 is stored as 1275, with a note to shift the decimal point two places to the left.

```
int population = 66_000_000; // 66 million in UK
double weight = 1.88; // in kilograms
decimal price = 4.99M; // in pounds sterling
string fruit = "Apples"; // strings use double-quotes
char letter = 'Z'; // chars use single-quotes
bool happy = true; // Booleans have value of true or false
Console.WriteLine($"{name} has {length2} characters."); // interpolated string
```

- `Var` 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.
- `dynamic` : type changes with each assignment
- **Default values for types** :
    - references : null
    - bool : False
    - Datetime : 01/01/0001 00:00:00

Arrays :

- Arrays are always of a fixed size at the time of memory allocation, so you need to decide how many items you want to store before instantiating them.
- Arrays are useful for temporarily storing multiple items, but collections are a more flexible option when adding and removing items dynamically.

```
string[] names; // can reference any array of strings
// allocating memory for four strings in an array
names = new string[4];
// storing items at index positions
names[0] = "Kate";
names[1] = "Jack";
names[2] = "Rebecca";
names[3] = "Tom";
// looping through the names
for (int i = 0; i < names.Length; i++)
{
 // output the item at index position i
 Console.WriteLine(names[i]);
}
```

Making a value type nullable (for `int`)

- Sometimes, for example, when reading values stored in a database that allows empty, missing, or null values, it is convenient to allow a value type to be null, we call this a nullable value type.

```
int thisCannotBeNull = 4;
thisCannotBeNull = null; // compile error!
int? thisCouldBeNull = null; // all good
```

Checking for null

```
string authorName = null;
// the following throws a NullReferenceException
int x = authorName.Length;
// instead of throwing an exception, null is assigned to y
int? y = authorName?.Length;

// result will be 3 if authorName?.Length is null
var result = authorName?.Length ?? 3;
Console.WriteLine(result);
```

Understanding format strings

```
string applesText = "Apples";
int applesCount = 1234;
string bananasText = "Bananas";
int bananasCount = 56789;
Console.WriteLine(
 format: "{0,-8} {1,6:N0}",
 arg0: "Name",
 arg1: "Count");
 format: "{0,-8} {1,6:N0}",
 arg0: applesText,
 arg1: applesCount);
Console.WriteLine(
 format: "{0,-8} {1,6:N0}",
 arg0: bananasText,
 arg1: bananasCount);
```

- Result :

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

Getting text input from the user

```
Console.Write("Type your first name and press ENTER: ");
string firstName = Console.ReadLine();
Console.Write("Type your age and press ENTER: ");
string age = Console.ReadLine();
Console.WriteLine(
 $"Hello {firstName}, you look good for {age}.");
```

Importing a namespace

```
using <Namespace> ;
```

Main arguments

- Command-line arguments are separated by spaces. Other characters like hyphens and colons are treated as part of an argument value. To include spaces in an argument value, enclose the argument value in single or double quotes

Controlling Flow and converting types

- `nameof` : name of the variable
- `sizeof` : size of the type

Looping with the `foreach` statement

- Technically, the `foreach` statement will work on any type that follows these rules:
    1. The type must have a method named `GetEnumerator` that returns an object.
    2. The returned object must have a property named Current and a method named `MoveNext`.
    3. The `MoveNext` method must return true if there are more items to enumerate through or false if there are no more items.
- There are interfaces named `IEnumerable` and `IEnumerable` that formally define these rules but technically the compiler does not require the type to implement these interfaces

```
string[] names = { "Adam", "Barry", "Charlie" };
foreach (string name in names)
{
 WriteLine($"{name} has {name.Length} characters.");
}
```

Casting and converting between types

Casting numbers implicitly and explicitly

```
using static System.Convert;
/*...*/
int d = (int)c;
double g = 9.8;
int h = ToInt32(g);
WriteLine($"g is {g} and h is {h}");
```

Rounding numbers

- the cast operator trims the decimal part of a real number and that the `System.Convert` methods round up or down.