Chapter 1: Introduction to C# .NET

1.1 Introduction to C# - its features and applications:

What is C# ?

- C# is pronounced "C-Sharp".
- It is an object-oriented programming language created by Microsoft that runs on the .NET Framework.
- The first version was released in year 2002. The latest version, C# 11.0, was released in November 8, 2022.
- It combines the features of C and C++ with the simplicity and ease of use of Visual Basic.

C# is used for:

- Mobile applications
- Desktop applications
- Web applications
- Web services
- Web Sites
- Games
- VR
- Database applications
- And much, much more!

Why Use C#?

1. It is one of the most popular programming language in the world.
2. It is easy to learn and simple to use.
3. It has a huge community support.
4. C# is an object-oriented language which gives a clear structure to programs and allows code to be reused.
5. lowering development costs.
6. As C# is close to C, C++ and Java, it makes it easy for programmers to switch to C# or vice versa.

Development of C# .Net

History of C# language is interesting to know. Here we are going to discuss brief history of C# language.

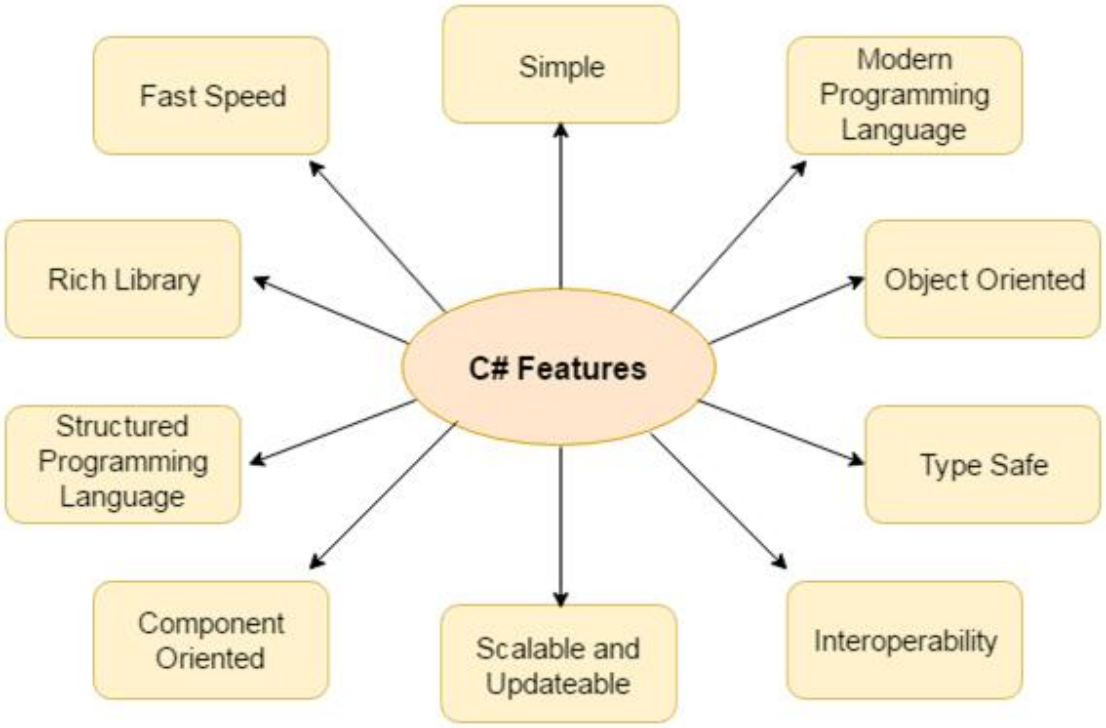
- C# is pronounced as "C-Sharp". It is an object-oriented programming language provided by Microsoft that runs on .NetFramework.
- **Anders Hejlsberg is known as the founder of C# language.**
- It is based on C++ and Java, but it has many additional extensions used to perform component oriented programming approach.
- C# has evolved much since their first release in the year 2002. It was introduced with .NET Framework 1.0 and the current version of C# is 11.0 (Until This Note Is Prepare)

Version	Details	Year Released
1.0	C# v1.0 came with .NET framework 1.0,1.1 having CLR version 1.0 and Microsoft Visual Studio 2002.	2002
2.0	C# v2.0 came with .NET framework 2.0 having CLR version 2.0, and Microsoft Visual Studio 2005.	2005
3.0	C# v3.0 came with .NET framework 3.0,3.5 having CLR version 2.0, and Microsoft Visual Studio 2008.	2007
4.0	C# v4.0 came with .NET framework 4.0 having CLR version 4.0, and Microsoft Visual Studio 2010.	2010
5.0	C# v5.0 came with .NET framework 4.5 having CLR version 4.0, and Microsoft Visual Studio 2012, 2013.	2012
6.0	C# v6.0 came with .NET framework 4.6 having CLR version 4.0 and Microsoft Visual Studio 2013, 2015.	2015
7.0	C# v7.0 came with .NET framework 4.6, 4.6.1, 4.6.2 having CLR version 4.0, and Microsoft Visual Studio 2015 and 2017.	2017
8.0	C# v8.0 came with .NET framework 4.8 having CLR version 4.0, and Microsoft Visual Studio 2019.	2019
9.0	C# v9.0 came with .NET framework 5.0 having CLR version >4.*, and Microsoft Visual Studio 2020.	2020
10.0	C# v10.0 came with .NET framework6.0 having CLR version >4.*, and Microsoft Visual Studio 2021.	2021
11.0	C# v11.0 came with .NET framework7.0 having CLR version >4.*, and Microsoft Visual Studio 2022.	2022

What is use of CLR?

The Common Language Runtime (CLR) is a component of the Microsoft .NET Framework that manages the execution of .NET applications. It is responsible for loading and executing the code written in various .NET programming languages, including C#, VB.NET, F#, and others.

Introduction to C# .net & it's features:



1. Simple

C# is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.

2. Modern Programming Language

C# programming is based upon the current trend, and it is very powerful and simple for building scalable, interoperable and robust applications.

3. Object Oriented

C# is object-oriented programming language. OOPs, makes development and maintenance easier whereas in Procedure-oriented programming language it is not easy to manage if code grows as project size grow.

4. Type Safe

C# type safe code can only access the memory location that it has permission to execute. Therefore, it improves a security of the program.

5. Interoperability

Interoperability process enables the C# programs to do almost anything that a native C++ application can do.

6. Scalable and Updatable

C# is automatic scalable and updatable programming language. For updating our application we delete the old files and update them with new ones.

7. Component Oriented

C# is component oriented programming language. It is the predominant software development methodology used to develop more robust and highly scalable applications.

8. Structured Programming Language

C# is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

9. Rich Library

C# provides a lot of inbuilt functions that makes the development fast.

10. Fast Speed

The compilation and execution time of C# language is fast.

1.2 Structure of C#:

The structure of a C# program consists of various elements organized in a specific manner. A basic structure of a C# program includes:

- Namespace declaration
- A class
- Class methods
- Class attributes
- A Main method
- Statements and Expressions
- Comments

```
using System;

  namespace HelloWorld
  {
     class Program
     {
        static void Main(string[] args)
        {
           /* comment */  or //comment
           Console.WriteLine("Hello World!");
        }
     }
  }
```

Program.cs

Line 1:
using System means that we can use classes from the System namespace.
Line 2:
A blank line. C# ignores white space. However, multiple lines makes the code more readable.
Line 3:
namespace is used to organize your code, and it is a container for classes and other namespaces.
Line 4:
The curly braces {} marks the beginning and the end of a block of code.
Line 5:
class is a container for data and methods, which brings functionality to your program. Every line of code that runs in C# must be inside a class. In our example, we named the class Program.
Line 7:
Another thing that always appear in a C# program, is the Main method. Any code inside its curly brackets will be executed.
Line 9:
The next line /*...*/ is ignored by the compiler and it is put to add comments in the program.

Line 10:
Console is a class of the System namespace, which has a WriteLine() method that is used to output/print text. In our example it will output "Hello World!".

If you omit the using System line, you would have to write **System.Console.WriteLine()** to print/output text.

Note: *Every C# statement ends with a semicolon ;.*
Note: *C# is case-sensitive: "MyClass" and "myclass" has different meaning.*
Note: *Unlike Java, the name of the C# file does not have to match the class name, but they often do (for better organization). When saving the file, save it using a proper name and add ".cs" to the end of the filename. To run the example above on your computer, make sure that C# is properly installed: Go to the Get Started Chapter for how to install C#. The output should be:*

1. **Namespace Declaration:** It defines the namespace in which the program resides, helping to organize and group related code.

In C#, a namespace is a way to organize and group related classes, structures, interfaces, enums, and other types within a logical container. It provides a hierarchical naming structure to avoid naming conflicts, improve code readability and helps in organizing code into manageable units.

Ex:

```
namespace MyApplication
{
    namespace Utilities
    {
        // Utility classes and functions
    }

    namespace Models
    {
        // Data models and entities
    }
}
```

2. **Class Declaration:** It defines a class, which is a blueprint for creating objects. Classes encapsulate data and behaviour into a single unit.

```
namespace MyNamespace
{
   class MyClass
   {
      // Class members go here
   }
}
```

3. **Main Method:** It serves as the entry point for the program and contains the code that will be executed first.

```
namespace MyNamespace
{
   class MyClass
   {
      static void Main(string[] args)
      {
         // Code to be executed goes here
      }
   }
}
```

1.3 Variables in C#:

Variables in C# are used to store and manipulate data. They have a data type, a name, and a value.
Each variable in C# has a specific type, which determines the size and layout of the variable's memory the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

The basic value types provided in C# can be categorized as:

Type	Example
Integral types	sbyte, byte, short, ushort, int, uint, long, ulong, and char
Floating point types	float and double
Decimal types	decimal
Boolean types	true or false values, as assigned
Nullable types	Nullable data types

The sizes of integral types, floating point type and decimal types in C# are as follows:
sbyte: 1 byte,byte: 1 byte,short: 2 bytes, ushort: 2 bytes, int: 4 bytes, uint: 4 bytes, long: 8 bytes, ulong: 8 bytes, char: 2 bytes
float: 4 bytes ,double: 8 bytes
decimal: 16 bytes, Boolean Type:
bool: The size of a bool value is implementation-specific, but it is typically 1 byte.

Defining Variables:

Syntax for variable definition in C# is:

<data_type> <variable_list>;

Here, data_type must be a valid C# data type including char, int, float, double, or any user-defined data type, and variable_list may consist of one or more identifier names separated by commas.

Some valid variable definitions are shown here:

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

Initializing Variables:

Variables are initialized (assigned a value) with an equal sign followed by a constant expression.
The general form of initialization is:
variable_name = value;

Variables can be initialized in their declaration. The initializer consists of an equal sign followed by a constant expression as −

<data_type> <variable_name> = value;

Some examples are:

a. int age; // Declaration of an integer variable
b. age = 25; // Assignment of a value to the age variable
c. double height = 1.75; // Declaration and assignment of a floating-point variable
d. char grade = 'A'; // Declaration and assignment of a character variable
e. bool isStudent = true; // Declaration and assignment of a boolean variable
f. string name = "John"; // Declaration and assignment of a string variable

C# Identifiers:

Identifiers are the name given to entities such as variables, methods, classes, etc. They are tokens in a program which uniquely identify an element. For example,

int value;

Here, value is the name of variable. Hence it is an identifier. Reserved keywords can not be used as identifiers unless @ is added as prefix:

example @if is valid Identifier but if is not valid identifier because it is a keyword, int break;
This statement will generate an error in compile time.

Rules for Naming an Identifier

- An identifier can not be a C# keyword.
- An identifier must begin with a letter, an underscore or @ symbol. The remaining part of identifier can contain
- letters, digits and underscore symbol.
- Whitespaces are not allowed. Neither it can have symbols other than letter, digits and underscore.
- Identifiers are case-sensitive. So, getName, GetName and getname represents 3 different identifiers.

Here are some of the valid and invalid identifiers:

Identifier	Remarks
number	valid
calculateMarks	valid
hello$	Invalid (Contains $)
name1	Valid
@if	Valid (Keyword with prefix @)
my_name	Valid
My Name	Invalid (Contains whitespace)
_hello_hi	Valid
my@name	Invalid
my-name	Invalid

Example:

```
using System;

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

Keywords	Identifiers
using	System
namespace	HelloWorld (namespace)
class	Hello (Class)
static	Main (Method)
void	args
string	Console

Constants

If we don't want to overwrite existing values, you can add the const keyword in front of the variable type.This will declare the variable as "constant", which means unchangeable and read-only.

Ex: const int myNum = 15;

Data types :

A data type specifies the size and type of variable values.

- **Numeric Data Types:**

- **Integer types** stores whole numbers, positive or negative (such as 123 or -456), without decimals.

  Ex: int myNum = 100000; long myNum = 15000000000L;

- **Floating point types** represents numbers with a fractional part, containing one or more decimals. Valid types are float ,double and decimal.

Ex:
float f1 = 123456.5F;
double d1 = 12345678912345.5d;
decimal d2 = 1.12345678913456791234567891 23m;

❖ **Character Data Types:**

It represents a single 16-bit Unicode character. Enclosed in single quotes.
e.g.char grade = 'A';

❖ **Boolean Data Type:**

It represents a Boolean value, either true or false.
bool isDone = false;

❖ **String Data Type:**

It represents a sequence of characters, such as words or sentences. Enclosed in double quotes, e.g., string name = "John Doe";

❖ **Date and Time Data Types:**

It represents a date and time value. Ex: DateTime currentDate = DateTime.Now;

❖ **Enum Data Type:**

It represents an enumerated type, which defines a set of named constant values.
Ex: enum DaysOfWeek { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
DaysOfWeek today = DaysOfWeek.Friday;

C# Type Casting:

Type casting is when you assign a value of one data type to another type.

Two types of casting are:

Implicit Casting - Implicit casting is done automatically when passing a smaller size type to a larger size type.
char -> int -> long -> float -> double
Ex: int myInt = 9;
double myDouble = myInt;

Explicit Casting - Explicit casting must be done manually by placing the type in parentheses in front of the value
double -> float -> long -> int -> char

Example:
double myDouble = 9.78;
int myInt = (int) myDouble

Type Conversion Methods:
It is also possible to convert data types explicitly by using built-in methods, such as Convert.ToBoolean, Convert.ToDouble, Convert.ToString, Convert.ToInt32 (int) and Convert.ToInt64 (long):

Ex:
```
int myInt = 10;
double myDouble = 5.25;
bool myBool = true;

Console.WriteLine(Convert.ToString(myInt));    // convert int to string
Console.WriteLine(Convert.ToDouble(myInt));    // convert int to double
Console.WriteLine(Convert.ToInt32(myDouble));  // convert double to int
Console.WriteLine(Convert.ToString(myBool));   // convert bool to string
```

1.4 Identifiers in C#:

Identifiers in C# are used to give names to various program elements such as variables, classes, methods, and namespaces. They must follow specific rules and conventions. Here are some guidelines for naming identifiers in C#:

- They must start with a letter or an underscore.
- They can contain letters, digits, and underscores.
- They are case-sensitive, meaning "myVariable" and "myvariable" are considered different.
- They cannot be a reserved keyword.
- It is recommended to use descriptive and meaningful names.

Examples of valid identifiers:

```
int age;
string firstName;
double _salary;
```

1.5 Keywords in C#:

Keywords in C# are reserved words that have a special meaning in the language and cannot be used as identifiers. They represent language constructs and control the behavior of the program. Examples of C# keywords include if, else, switch, for, while, class, int, string, and many others.

```
int age = 25;
if (age > 18)
{
   Console.WriteLine("You are an adult.");
}
else{
 Console.WriteLine("You are a kid.");

}
```

In the example above, if and else are keywords used for conditional statements, while int is a keyword used for declaring an integer variable.

1.6 Data Types in C#:

Data types in C# specify the type of data that can be stored in variables. C# supports two categories of data types: value types and reference types.

- **Value types** represent simple data types, such as integers, floating-point numbers, booleans, and characters. They are stored directly in memory and occupy a fixed size.

Examples of value types:

```
int age = 25;
double height = 1.75;
bool isStudent = true;
char grade = 'A';
```

- **Reference types** represent complex data types, such as classes, interfaces, arrays, and delegates. They store a reference (memory address) to the actual data, which is allocated on the heap. Examples of reference types:

```
string name = "John";
object obj = new object();
int[] numbers = new int[] { 1, 2, 3 };
```

1.7 C# Type Conversion:

Type conversion, also known as type casting, is the process of converting a value from one data type to another. C# supports two types of type conversions: implicit and explicit.

- **Implicit conversion** is done automatically by the compiler when there is no risk of data loss or precision. For example:

```
int number = 10;
double result = number; // Implicit conversion from int to double
```

- **Explicit conversion** requires the use of casting operators and may result in data loss or exceptions. For example:

```
double value = 3.14;
int approximation = (int)value; // Explicit conversion from double to int
```

1.8 C# Operators:

Operators in C# are symbols or keywords that perform specific operations on operands (variables or values). C# supports a wide range of operators for arithmetic, assignment, comparison, logical operations, and more.

1. **Arithmetic operators** are used for performing mathematical calculations:

```
int a = 5;
int b = 2;
int sum = a + b; // Addition
int difference = a - b; // Subtraction
int product = a * b; // Multiplication
```

```
int quotient = a / b; // Division
int remainder = a % b; // Modulus (remainder)
```

2. **Assignment operators** are used to assign values to variables:

```
int x = 10;
x += 5; // Equivalent to x = x + 5
x -= 3; // Equivalent to x = x - 3
```

3. **Comparison operators** are used to compare values:

```
int a = 5;
int b = 3;
bool isEqual = (a == b); // Equality comparison
bool isGreaterThan = (a > b); // Greater than comparison
bool isLessThan = (a < b); // Less than comparison
```

4. **Logical operators** are used for logical operations:

```
bool condition1 = true;
bool condition2 = false;
bool result = (condition1 && condition2); // Logical AND
bool result2 = (condition1 || condition2); // Logical OR
bool result3 = !condition1; // Logical NOT
```

These are just a few examples of C# operators. Understanding and using operators effectively is crucial for performing various calculations, comparisons, and logical operations in C#.