**Basic Concepts of C#:**

* **Classes [**[**Reference**](https://www.c-sharpcorner.com/UploadFile/0c1bb2/types-of-classes-in-C-Sharp1/)**]**
  + Classes are user-defined data types that represent the state and behavior of an object. The state represents the properties, and behavior is the action that objects can perform.
  + Classes are reference types that hold the object created dynamically in a heap.
  + All classes have a base type of System.Object.
  + The default access modifier of a class is Internal.
  + The default access modifier of methods and variables is Private.
  + Directly inside the namespaces, declarations of private classes are not allowed.
  + Types of Classes:
    - Abstract
    - Partial
    - Sealed
    - Static
* **Access Modifiers [**[**Reference**](https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers)**]**
  + Access Modifiers are keywords in C# which are used to restrict availability of object, method, class, and its members into the program or in application.
  + **Types:**

| **Caller's location** | **public** | **protected internal** | **protected** | **internal** | **private protected** | **private** |
| --- | --- | --- | --- | --- | --- | --- |
| Within the class | ✔️️ | ✔️ | ✔️ | ✔️ | ✔️ | ✔️ |
| Derived class (same assembly) | ✔️ | ✔️ | ✔️ | ✔️ | ✔️ | ❌ |
| Non-derived class (same assembly) | ✔️ | ✔️ | ❌ | ✔️ | ❌ | ❌ |
| Derived class (different assembly) | ✔️ | ✔️ | ✔️ | ❌ | ❌ | ❌ |
| Non-derived class (different assembly) | ✔️ | ❌ | ❌ | ❌ | ❌ | ❌ |

* **Virtual Keyword [**[**Reference**](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/virtual)**]**
  + The virtual keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class. For example, this method can be overridden by any class that inherits it.
  + By default, methods are non-virtual. You cannot override a non-virtual method.
  + You cannot use the virtual modifier with the static, abstract, private, or override modifiers.
* **Generics [**[**Reference**](https://learn.microsoft.com/en-us/dotnet/standard/generics/)**]**
  + Generics let you tailor a method, class, structure, or interface to the precise data type it acts upon.
  + Generics are classes, structures, interfaces, and methods that have placeholders (type parameters) for one or more of the types that they store or use.
  + A generic collection class might use a type parameter as a placeholder for the type of objects that it stores.
  + Syntax:
    - public class SimpleGenericClass<T>

{

public T Field;

}

* **Delegates/ Events [**[**Reference**](https://learn.microsoft.com/en-us/dotnet/standard/delegates-lambdas)**]**
  + A delegate defines a type that represents references to methods that have a particular parameter list and return type.
  + A method (static or instance) whose parameter list and return type match can be assigned to a variable of that type, then called directly (with the appropriate arguments) or passed as an argument itself to another method and then called.
  + Anonymous delegate -> without method name , usually inline delegate.
  + **Types:**
    - Action<> is used when there is a need to perform an action using the arguments of the delegate. The method it encapsulates does not return a value.
    - Func<> is used usually when you have a transformation on hand, that is, you need to transform the arguments of the delegate into a different result. Projections are a good example. The method it encapsulates returns a specified value.
    - Predicate<> is used when you need to determine if the argument satisfies the condition of the delegate. It can also be written as a Func<T, bool>, which means the method returns a Boolean value.
  + **EVENT:** 
    - An event is a notification sent by an object to signal the occurrence of an action. Events in .NET follow the observer design pattern.
    - The class who raises events is called Publisher, and the class who receives the notification is called Subscriber.
    - In C#, an event is an encapsulated delegate. It is dependent on the delegate. The delegate defines the signature for the event handler method of the subscriber class.
* **Out Keyword [**[**Reference**](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out)**]**
  + Out keyword is used in the following contexts: [[Reference](https://www.geeksforgeeks.org/out-parameter-with-examples-in-c-sharp/)]
    - As a parameter modifier, which lets you pass an argument to a method by reference rather than by value.
      * For using out keyword as a parameter both the method definition and calling method must use the out keyword explicitly
    - In generic type parameter declarations for interfaces and delegates, which specifies that a type parameter is covariant.
* **Reference Types [**[**Reference**](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/reference-types)**]**
  + Contains reference to an object.
  + If values are changed using one variable, it effects other too.
  + Class, object, interface, string, array, delegate
* **Value Types [**[**Reference**](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-types)**]**
  + Variable contains object with the value.
  + Primitives, struct, Enum.
* **Operators**
  + **Arithmetic:**
    - Unary plus +
    - Unary minus -
    - Binary plus +
    - Binary minus -
    - Multiplication \*
    - Division /
    - Modulus %
    - Increment (Pre & Post) ++
    - Decrement (Pre & Post) --
  + **Relational**
    - Equal to (==)
    - Not Equal to (!=)
    - Greater than (>)
    - Less than (<)
    - Greater than or equal to (>=)
    - Less than or equal to (<=)
  + **Logical**
    - AND &&
    - OR ||
    - NOT !
  + **Bitwise**
    - AND &
    - OR |
    - XOR ^
    - Left Shift <<
    - Right Shift >>
  + **Assignment**
    - Simple Assignment =
    - Add Assignment +=
    - Subtract Assignment -=
    - Multiple Assignment \*=
    - Division Assignment /=
    - Modulus Assignment %=
    - Left Shift Assignment <<=
    - Right Shift Assignment >>=
    - Bitwise AND Assignment &=
    - Bitwise OR Assignment |=
    - Bitwise XOR Assignment ^=
  + **Conditional**
    - Condition ? trueStatments : falseStatments;
  + Is Operator
  + As Operator
  + Cast Expression
  + Typeof Operator
* **Assemblies**
  + Assemblies are the fundamental units of deployment, version control, reuse, activation scoping, and security permissions for .NET-based applications.
  + An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality.
  + Assemblies take the form of executable (.exe) or dynamic link library (.dll) files, and are the building blocks of .NET applications.
* **Parameter Types [**[**References**](https://www.geeksforgeeks.org/c-sharp-method-parameters/)**]**
  + Named Parameters
  + Ref Parameters
  + Out Parameters
  + Default or Optional Parameters
  + Dynamic Parameters
  + Value Parameters
  + Params
* **Interfaces**
  + An interface contains definitions for a group of related functionalities that a non-abstract class or a struct must implement.
  + An interface may define static methods, which must have an implementation.
  + An interface may define a default implementation for members.
  + Used to achieve complete abstraction.
  + Supports multiple inheritance.
* **Casting [**[**Reference**](https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/types/casting-and-type-conversions)**]**
  + Converting one type to another type.
  + Implicit Conversion
  + Explicit Conversion
  + User-defined Conversion
  + Conversion with Helper Classes
* **Strings [**[**Reference**](https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/strings/)**]**
  + A string is an object of type String whose value is text. Internally, the text is stored as a sequential read-only collection of Char objects.
  + There's no null-terminating character at the end of a C# string; therefore a C# string can contain any number of embedded null characters ('\0').
  + “string” is alias for “System.String”.
  + String objects are immutable: they can't be changed after they've been created.
  + All the String methods and C# operators that appear to modify a string actually return the results in a new string object.
* **Statements**
  + A statement can consist of a single line of code that ends in a semicolon, or a series of single-line statements in a block.
  + A statement block is enclosed in {} brackets and can contain nested blocks.
  + The actions that a program takes are expressed in statements.
  + Common actions include declaring variables, assigning values, calling methods, looping through collections, and branching to one or another block of code, depending on a given condition.
  + The order in which statements are executed in a program is called the flow of control or flow of execution.
* **Expressions**
  + Expression body definitions let you provide a member's implementation in a concise, readable form.
  + You can use an expression body definition whenever the logic for any supported member, such as a method or property, consists of a single expression.
  + Syntax:
    - member => expression;
* **Collections**
  + Similar data can often be handled more efficiently when stored and manipulated as a collection.
  + the classes in the System.Collections, System.Collections.Generic, System.Collections.Concurrent, and System.Collections.Immutable namespaces to add, remove, and modify either individual elements or a range of elements in a collection.
  + Concurrent collections are useful to increase performance.
    - The Queue class, as well as the Queue<T>, ConcurrentQueue<T>, and ImmutableQueue<T> generic classes all offer FIFO access. For more information, see When to Use a Thread-Safe Collection.
    - The Stack class, as well as the Stack<T>, ConcurrentStack<T>, and ImmutableStack<T> generic classes all offer LIFO access. For more information, see When to Use a Thread-Safe Collection.
    - The LinkedList<T> generic class allows sequential access either from the head to the tail, or from the tail to the head.
    - The Queue class, as well as the Queue<T>, ConcurrentQueue<T>, and ImmutableQueue<T> generic classes all offer FIFO access. For more information, see When to Use a Thread-Safe Collection.
    - The Stack class, as well as the Stack<T>, ConcurrentStack<T>, and ImmutableStack<T> generic classes all offer LIFO access. For more information, see When to Use a Thread-Safe Collection.
    - The LinkedList<T> generic class allows sequential access either from the head to the tail, or from the tail to the head.
* **Boxing / Unboxing**
  + Boxing is the process of converting a value type to the type object or to any interface type implemented by this value type.
  + When the common language runtime (CLR) boxes a value type, it wraps the value inside a System.Object instance and stores it on the managed heap.
  + Unboxing extracts the value type from the object. Boxing is implicit; unboxing is explicit. The concept of boxing and unboxing underlies the C# unified view of the type system in which a value of any type can be treated as an object.
  + Example:
    - int num = 23; // value type is int and assigned value 23

Object Obj = num; // Boxing

int i = (int)Obj; // Unboxing

* **Async**
  + Async and Await are the two keywords that help us to program asynchronously.
  + An async keyword is a method that performs asynchronous tasks such as fetching data from a database, reading a file, etc, they can be marked as “async”.
  + Whereas await keyword making “await” to a statement means suspending the execution of the async method it is residing in until the asynchronous task completes.
  + Classes used: using System.Threading; using System.Threading.Tasks;
  + Async will support parallel execution of tasks.
  + Await will wait till the task is completed.
* **Using Keyword**
  + Using statement is used for importing assemblies.
  + The using statement ensures the correct use of an IDisposable instance:
  + Example C#

var numbers = new List<int>();

using (StreamReader reader = File.OpenText("numbers.txt"))

{

string line;

while ((line = reader.ReadLine()) is not null)

{

if (int.TryParse(line, out int number))

{

numbers.Add(number);

}

}

}

* **Arrays**
  + Store collection of similar or different types under a single name.
  + Elements store in Contiguous memory locations.
  + To store multiple type data, we can use Object as type.

**.NET 7 and .NET 8**

* Performance is improved in .Net8 compared to .Net7
* Better cross platform compatibility in .Net 8.
* Better integration with MS tech like Azure, Visual Studio and Windows.

**Donet CLI**

* The .NET command-line interface (CLI) is a cross-platform toolchain for developing, building, running, and publishing .NET applications.
* [.NET CLI | Microsoft Learn](https://learn.microsoft.com/en-us/dotnet/core/tools/) reference

**NuGet**

* NuGet is the package manager for .NET. The NuGet client tools provide the ability to produce and consume packages.
* The NuGet Gallery is the central package repository used by all package authors and consumers.
* We can work with NuGet either through NuGet manager under Tools or right click on project and explore packages under NuGet.
* We can use NuGet CLI to work with NuGet packages using CND.