Basics of C#

API Basic Training

Prince Goswami

Table of Contents

Introduction to C#

It is an object-oriented programming language created by Microsoft that runs on the .NET Framework.

C# has roots from the C family, and the language is close to other popular languages like C++ and Java.

The first version was released in 2002. The latest version C# 12, , was released in November 2023.

C# is used for:

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

Create first C# program ‘Hello world’

```
// Hello World! program
namespace HelloWorld
{
    class Hello {
        static void Main(string[] args)
```

```
            {
                System.Console.WriteLine("Hello World!");
            }
        }
    }
```

Understanding C# program structure

Let's look into various parts of the above C# program.

- /* Comments */
    - /* Print, some string in C# */, is the comment statement that does not get executed by the compiler and is used for code readers or programmers' understanding.
- using
    - Here, using keyword is employed for fetching all the associated methods that are within the System namespace. Any C# program can have multiple using statements.
- namespace
    - This next statement is used for declaring a namespace, which is a collection of classes under a single unit; here, "printHelloCsharp".
- Class
    - Next, you have declared a class "HelloCsharp" using the keyword class, which is used for preserving the concept of encapsulation.
- main()
    - Inside a class, you can have multiple methods and declare variable names. The static/void is a return value, which will be explained in a while.
- Console.WriteLine()

- Console.WriteLine() is a predefined method used for displaying any string or variable's value in a C# program.

- Console.ReadKey()
    - Console.ReadKey() is another predefined method used to make the program wait for any key-press.

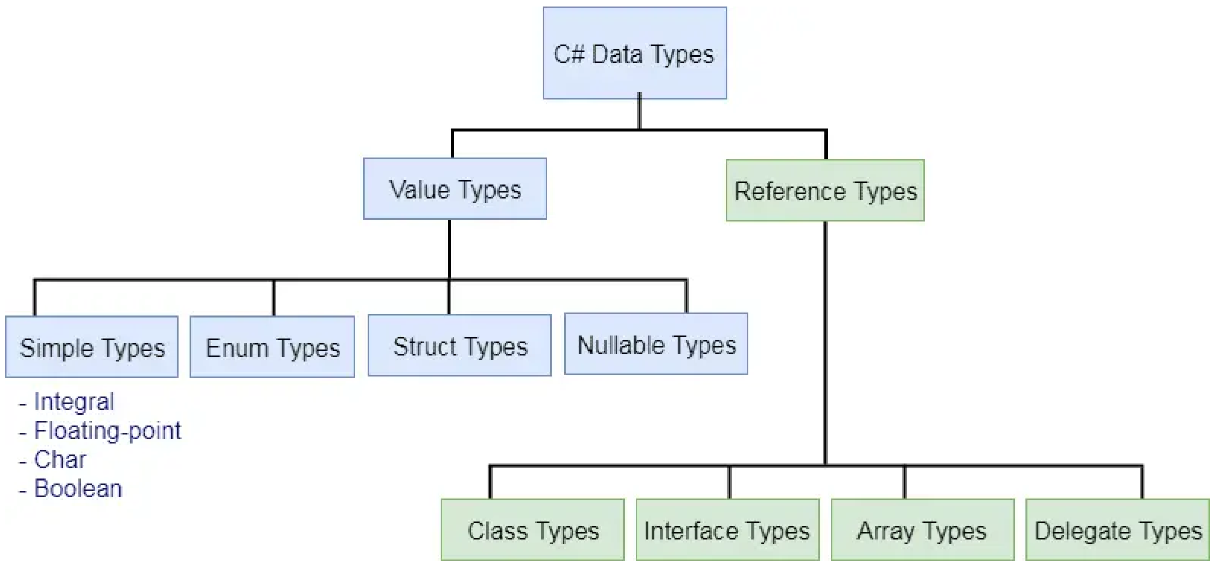
Working with Code files, Projects & Solutions

—-------------------------------------PENDING---

Datatypes & Variables with type conversion

C# mainly categorized data types in two types: Value types and Reference types.

- Value types include simple types (such as int, float, bool, and char), enum types, struct types, and Nullable value types.
- Reference types include class types, interface types, delegate types, and array types.



Predefined Data Types in C#

C# includes some predefined value types and reference types. The following table lists predefined data types:

Type	Description	Range	Suffix
byte	8-bit unsigned integer	0 to 255	
sbyte	8-bit signed integer	-128 to 127	
short	16-bit signed integer	-32,768 to 32,767	
ushort	16-bit unsigned integer	0 to 65,535	
int	32-bit signed integer	-2,147,483,648 to 2,147,483,647	
uint	32-bit unsigned integer	0 to 4,294,967,295	u
long	64-bit signed integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	l
ulong	64-bit unsigned integer	0 to 18,446,744,073,709,551,615	ul
float	32-bit Single-precision floating point type	-3.402823e38 to 3.402823e38	f

double	64-bit double-precision floating point type	-1.79769313486232e308 to 1.79769313486232e308	d
decimal	128-bit decimal type for financial and monetary calculations	(+ or -)1.0 x 10e-28 to 7.9 x 10e28	m
char	16-bit single Unicode character	Any valid character, e.g. a,*, \x0058 (hex), or\u0058 (Unicode)	
bool	8-bit logical true/false value	True or False	
object	Base type of all other types.		
string	A sequence of Unicode characters		
DateTime	Represents date and time	0:00:00am 1/1/01 to 11:59:59pm 12/31/9999	

C# Type Conversion

The process of converting the value of one type (int, float, double, etc.) to another type is known as type conversion.

In C#, there are two basic types of type conversion:

1. Implicit Type Conversions
2. Explicit Type Conversions

1. Implicit Type Conversion in C#

In implicit type conversion, the C# compiler automatically converts one type to another.

Generally, smaller types like int (having less memory size) are automatically converted to larger types like double (having larger memory size).

// Implicit Conversion

double numDouble = numInt;

2. C# Explicit Type Conversion

In explicit type conversion, we explicitly convert one type to another.

Generally, larger types like double (having large memory size) are converted to smaller types like int (having small memory size).

// Explicit casting

int numInt = (int) numDouble;

Operators & Expression

Operators are the foundation of any programming language. Thus the functionality of C# language is incomplete without the use of operators. Operators allow us to perform different kinds of operations on operands. In C#,

operators Can be categorized based upon their different functionality :

no.	Name	Operators
1	Arithmetic Operators	+, -, *, /, %
2	Relational Operators	==, =!, >, <, <=, >=
3	Logical Operators	&&, ||, !
4	Bitwise Operators	&, ^, |, <<, >>

no.	Name	Operators
5	Assignment Operators	==,+=,-=,*=,/=,etc.
6	Conditional Operator	con ? f_exp : s_exp

Operators can also categorized based upon Number of Operands :

1. Unary Operator: Operator that takes one operand to perform the operation.
2. Binary Operator: Operator that takes two operands to perform the operation.
3. Ternary Operator: Operator that takes three operands to perform the operation.

C# Statements

A statement is a basic unit of execution of a program. A program consists of multiple statements.

For example:

```
int age = 21;
Int marks = 90;
```

In the above example, both lines above are statements.

There are different types of statements in C#. In this tutorial, we'll mainly focus on two of them:

1. Declaration Statement

2. Expression Statement

Declaration Statement

Declaration statements are used to declare and initialize variables.

```
char ch;
int maxValue = 55;
```

Expression Statement

An expression followed by a semicolon is called an expression statement.

```
/* Assignment */
area = 3.14 * radius * radius;
/* Method call is an expression*/
```

Understanding Arrays

You can store multiple variables of the same type in an array data structure. You declare an array by specifying the type of its elements.
If you want the array to store elements of any type, you can specify an object as its type. In the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from Object.

```
type[] arrayName;
```

An array has the following properties:

- An array can be single-dimensional, multidimensional, or jagged.
- The number of dimensions are set when an array variable is declared. The length of each dimension is established when the array instance is

created. These values can't be changed during the lifetime of the instance.

- A jagged array is an array of arrays, and each member array has the default value of null.
- Arrays are zero indexed: an array with n elements is indexed from 0 to n-1.
- Array elements can be of any type, including an array type.
- Array types are reference types derived from the abstract base type Array. All arrays implement IList and IEnumerable. You can use the foreach statement to iterate through an array. Single-dimensional arrays also implement IList<T> and IEnumerable<T>.

There are many different kinds of arrays available :

- Single-dimensional arrays - []
- Multidimensional arrays - [,,]
- Jagged arrays - [][]
- Implicitly typed arrays - new[] {,,,}

Defining & Calling Methods

Define :

When you define a method, you basically declare the elements of its structure. The syntax for defining a method in C# is as follows −

<Access Specifier> <Return Type> <Method Name>(Parameter List) {

Method Body

}

Following are the various elements of a method −

- **Access Specifier** − This determines the visibility of a variable or a method from another class.

- **Return type** − A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is void.
- **Method name** − Method name is a unique identifier and it is case sensitive. It cannot be the same as any other identifier declared in the class.
- **Parameter list** − Enclosed between parentheses, the parameters are used to pass and receive data from a method. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.
- **Method body** − This contains the set of instructions needed to complete the required activity.

Calling :

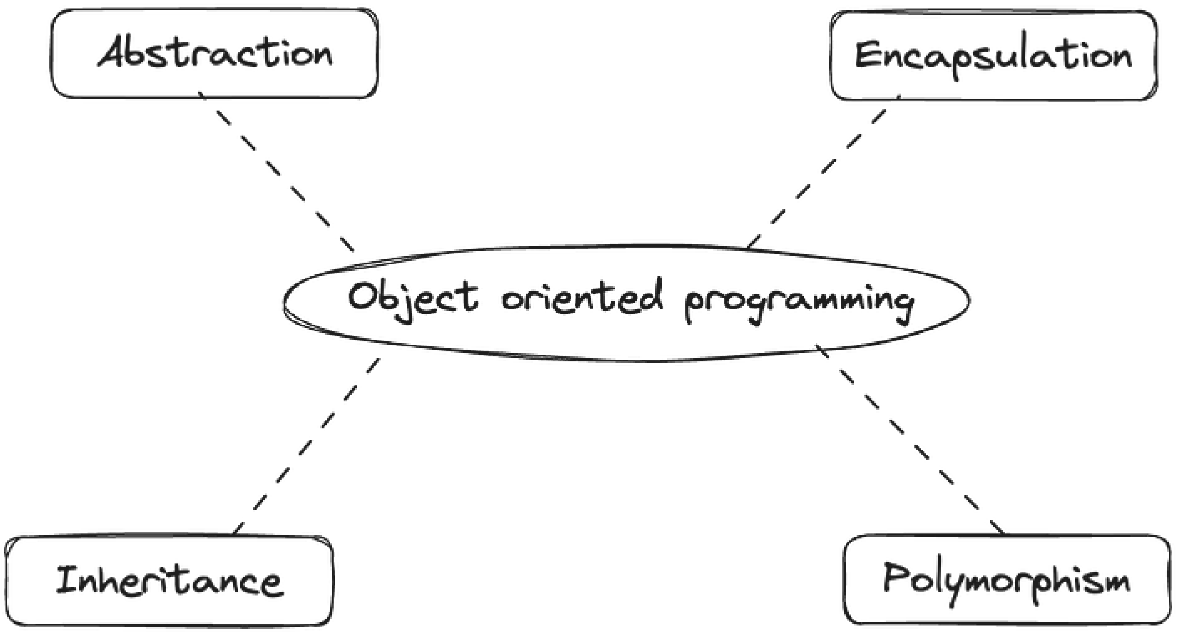
You can call a method using the name of the method.

When a method with parameters is called, you need to pass the parameters to the method. There are three ways that parameters can be passed to a method −

1. Value parameters:
    a. This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2. Reference parameters
    a. This method copies the reference to the memory location of an argument into the formal parameter. This means that changes made to the parameter affect the argument.
3. Output parameters
    a. This method helps in returning more than one value.

Understanding Classes & OOP concepts

OOP - Object Oriented Programing



Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C# code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Main Pillars of Object-Oriented Programming :

- **Abstraction**: Show only the necessary things.
- **Polymorphism**: More than one form: An object can behave differently at different levels.
- **Inheritance**: Parent and Child Class relationships.
- **Encapsulation**: Hides the Complexity.
- **Classes and Objects**: A class is a type, or blueprint which contains the data members and methods , An Object is an instance.

Abstraction :

Abstraction is an important part of object oriented programming. It means that only the required information is visible to the user and the rest of the information is hidden. Abstraction can be implemented using abstract classes in C#. Abstract classes are base classes with partial implementation.

Encapsulation :

Wrapping up data into a single unit. Encapsulation hides the complexity of the way the object was implemented . Encapsulation in object-oriented programming (OOP) is a mechanism of hiding the internal details (implementation) of an object from other objects and the outside world.

Inheritance (Derived and Base class)

Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and speeds up implementation time.

Polymorphism

Polymorphism means assigning a single name but there can be multiple behaviours.

Polymorphism means single name & multiple meaning. So there are two types of binding to call that function.

1. Method Overloading — Compile time
2. Method Overriding — Runtime

Class and Objects

A class in C# is a blueprint or template that is used for declaring an object. However, there is no need to declare an object of the static class. A class consists of member variables, functions, properties etc. A method is a block of code in C# programming. The function makes the program modular and easy to understand.

In object oriented programming, classes and methods are the essential thing. It provides reusability of code and makes c# programming is more secure.

Interface & Inheritance

Interface - define behavior for multiple types

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. An interface may not declare instance data such as fields, auto-implemented properties, or property-like events.

By using interfaces, you can, for example, include behavior from multiple sources in a class. That capability is important in C# because the language doesn't support multiple inheritance of classes. In addition, you must use an interface if you want to simulate inheritance for structs, because they can't actually inherit from another struct or class.

```
interface IEquatable<T>
{
   bool Equals(T obj);
}
```

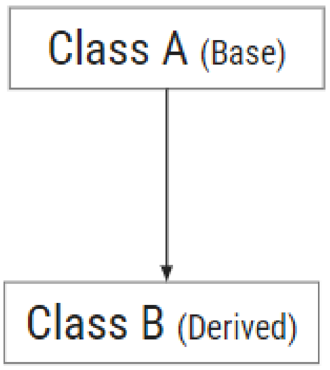
Inheritance

- Inheritance is where one class (child class) inherits the properties of another class (parent class).

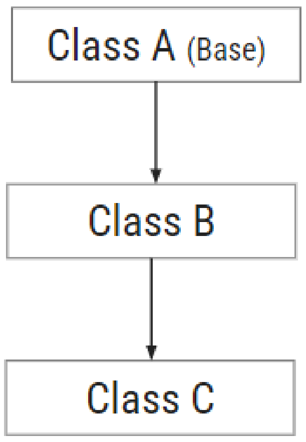
- Inheritance is a mechanism of acquiring the features and behaviors of a class by another class.
- The class whose members are inherited is called the base class, and the class that inherits those members is called the derived class.
- Inheritance implements the IS-A relationship.
- Why Inheritance?
    - Reduce code redundancy
    - Provides code reusability
    - Reduces source code size and improves code readability
    - Code is easy to manage and divided into parent and child classes

Types of Inheritance :

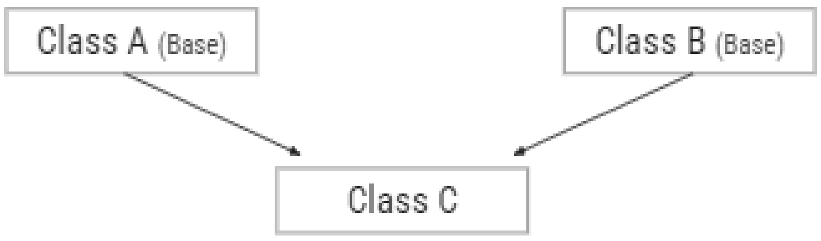
1. Single Inheritance
    a. In single inheritance, a derived class is created from a single base class.



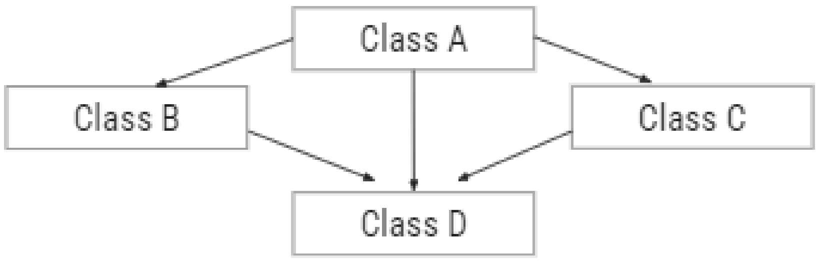
2. Multi-Level Inheritance
    a. In Multi-level inheritance, a derived class is created from another derived class



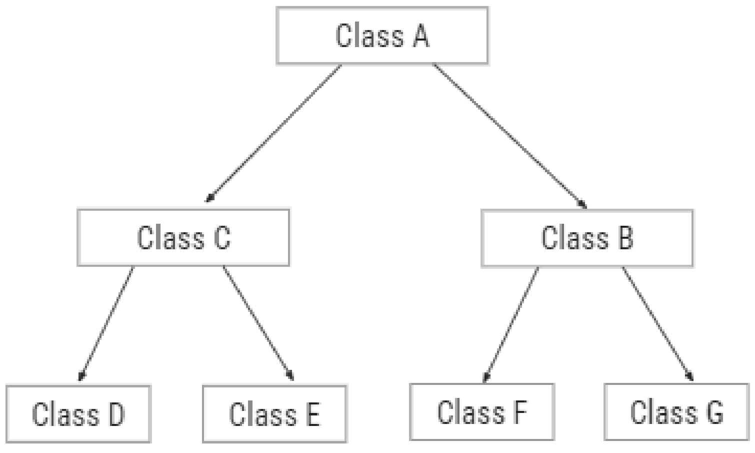
3. Multiple Inheritance
    a. In Multiple inheritance, a derived class is created from more than one base class.
    b. This type of inheritance is not supported by .NET Languages like C#, F# etc.



4. Multipath Inheritance
    a. In Multipath inheritance, a derived class is created from another derived classes and the same base class of another derived classes.
    b. This type of inheritance is not supported by .NET Languages like C#, F# etc.

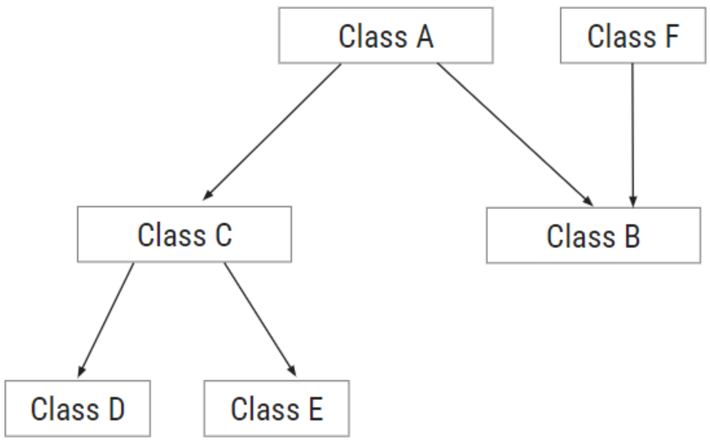


5. Hierarchical Inheritance
    a. Hierarchical inheritance describes a situation in which a parent class is inherited by multiple subclasses.
    b. A type of inheritance in which more than one class is inherited from a single parent or base class is known as hierarchical inheritance.



6. Hybrid Inheritance

a. Hybrid Inheritance, As the name suggests, combines multiple types of inheritance, such as single, multiple, hierarchical, or multilevel inheritance.
b. In C#, hybrid inheritance enables us to create complex class hierarchies by inheriting implementation details from multiple base classes (Including classes and interfaces).



Scope & Accessibility modifier

- Access modifiers are keywords used to specify the declared accessibility of a member or a type.
- Why to use access modifiers?
  - Access modifiers are an integral part of object-oriented programming.
  - They support the concept of encapsulation, which promotes the idea of hiding functionality.
  - Access modifiers allow you to define who does or doesn't have access to certain features.
- In C# Modifiers can be divided in five categories.
  1. Public
  2. Private
  3. Protected
  4. Internal
  5. Protected internal

Public :

- Public members are accessible anywhere in program or application.
- Public access is the most permissive access level.
- There are no restrictions on accessing public members.
- The public keyword is used for it.

- Accessibility:
  - Can be accessed by objects of the class

  - Can be accessed by derived classes

Private :

- Private members are accessible only within the body or scope of the class or the structure in which they are declared.
- Private access is the least permissive access level.
- The private keyword is used for it.

- Accessibility:
  - Cannot be accessed by object
  - Cannot be accessed by derived classes

Protected :

- A protected member is accessible from within the class in which it is declared and from within any class derived from the class that declared this member.
- A protected member of a base class is accessible in a derived class only if the access takes place through the derived class type.
- The protected keyword is used for it.

- Accessibility:
  - Cannot be accessed by object
  - By derived classes

Internal :

- We can declare a class as internal or its member as internal.
- Internal members are accessible only within the same assembly.
- In other words, access is limited exclusively to classes defined within the current project assembly.
- The internal keyword is used for it.

- Accessibility:
  - The variable or classes that are declared with internal can be access by any member within application.
  - It is the default access specifiers for a class in C# programming.

Protected Internal :

- The protected internal accessibility means protected OR internal, not protected AND internal.
- In other words, a protected internal member is accessible from any class in the same assembly, including derived classes.
- The protected internal access specifier allows its members to be accessed in derived class, containing classes within same application.

- Accessibility:
    - Within the class in which they are declared
    - Within the derived classes of that class available within the same assembly
    - Outside the class within the same assembly
    - Within the derived classes of that class available outside the assembly

Namespace & .Net Library

Namespace

The namespace keyword is used to declare a scope that contains a set of related objects. You can use a namespace to organize code elements and to create globally unique types.

Within a namespace, you can declare zero or more of the following types:

1. class
2. interface
3. struct
4. enum
5. delegate
6. nested namespaces can be declared except in file scoped namespace declarations

.Net Library

—---Pending—---

Creating & Adding Reference to Assemblies

—---Pending—---

Working with Collections

- We mostly use arrays to store and manage data.However, arrays can store a particular type of data, such as integer and characters.To resolve this issue, the .NET framework introduces the concept of collections for data storage and retrieval, where collection means a group of different types of data.
- Collections provide a more flexible way of working with a group of objects.In arrays, you need to define elements at declaration time.
- But in a collection, you don't need to define the size of the collection in advance. You can add elements or even remove elements from the collection at any point of time.

  Most useful collection classes defined are as follows.

    1. ArrayList Class

2. Stack Class
3. Queue Class
4. Hashtable Class
5. SortedList Class
6. BitArray Class

ArrayList Class

- Using an arrays, the main problem arises is that their size always remain fixed by the number that you specify when declaring an arrays.In addition, you cannot add items in the middle of array.
- Also you can not deal with different data types.The solution to these problems is the use of ArrayList Class.Similar to an array, the ArrayList class allows you to store and manage multiple elements.
- With the ArrayList class, you can add and remove items from a list of array items from a specified position, and then the array items list automatically resizes itself accordingly.

```
ArrayList A1 = new ArrayList();
A1.Add(10);
A1.Add("Hello");
Console.WriteLine(A1[0]);
Console.WriteLine(A1[1]);
```

Stack Class

- The stack class represents a last in first out (LIFO) concept.let's take an example. Imagine a stack of books with each book kept on top of each other.
- The concept of last in first out in the case of books means that only the top most book can be removed from the stack of books.
- It is not possible to remove a book from between, because then that would disturb the setting of the stack. Hence in C#, the stack also works in the same way.
- Elements are added to the stack, one on the top of each other.
- The process of adding an element to the stack is called a push operation.The process of removing an element from the stack is called a pop operation.

```
Stack st = new Stack();
        st.Push(1);
        st.Push(2);
        st.Push(3);

        foreach (Object obj in st)
        {
           Console.WriteLine(obj);
        }
```

```
Console.WriteLine("Total elements " + st.Count);
```

Queue Class:

- The Queue class represents a first in first out (FIFO) concept.let's take an example. Imagine a queue of people waiting for the bus.
- Normally, the first person who enters the queue will be the first person to enter the bus.
- Similarly, the last person to enter the queue will be the last person to enter into the bus.The process of adding an element to the queue is the Enqueue operation.
- The process of removing an element from the queue is the Dequeue operation.

```
Queue qt = new Queue();
 qt.Enqueue(1);
 qt.Enqueue(2);
 foreach (Object obj in qt)
 {
    Console.WriteLine(obj);
 }
 Console.WriteLine("Total elements =" + qt.Count);
```

Hashtable Class

- The Hashtable class is similar to the ArrayList class, but it allows you to access the items by a key.
- Each item in a Hashtable object has a key/value pair.
- So instead of storing just one value like the stack, array list and queue, the Hashtable stores 2 values that is key and value.
- The key is used to access the items in a collection and each key must be unique, whereas the value is stored in an array.
- The keys are short strings or integers.

```
Hashtable ht = new Hashtable();
ht.Add("1", "Darshan");
ht.Add("2", "Institute");
ICollection keys = ht.Keys;
foreach (String k in keys)
{
   Console.WriteLine(ht[k]);
}
```

Sorted List :

- The SortedList class is a combination of the array and hashtable classes.
- The SortedList<TKey, TValue>, and SortedList are collection classes that can store key-value pairs that are sorted by the keys based on the associated IComparer implementation.

- For example, if the keys are of primitive types, then sorted in ascending order of keys.
- C# supports generic and non-generic SortedList.
- It is recommended to use generic SortedList<TKey, TValue> because it performs faster and less error-prone than the non-generic SortedList.

- SortedList Characteristics
    1. SortedList<TKey, TValue> is an array of key-value pairs sorted by keys.
    2. Sorts elements as soon as they are added. Sorts primitive type keys in ascending order and object keys based on IComparer<T>.
    3. It comes under System.Collection.Generic namespace.
    4. A key must be unique and cannot be null but a value can be null or duplicate.
    5. A value can be accessed by passing associated key in the indexer mySortedList[key]

```
SortedList<int,string>numberNames=new SortedList<int, string>();
numberNames.Add(3, "Three");
numberNames.Add(1, "One");
numberNames.Add(2, "Two");
foreach (var vr in numberNames)
{
        Console.WriteLine("key:{0},value:{1}",vr.Key,vr.Value;
 }
```

BitArray Class

- The BitArray class is used to manage an array of the binary representation using the value 1 and 0, where 1 stands for true and 0 stands for false.
- A BitArray object is resizable, which is helpful in case if you do not know the number of bits in advance. You can access items from the BitArray collection class by using an integer index.
- A BitArray can be used to perform Logical AND, XOR, OR operations.

```
 BitArray ba = new BitArray(4);
 ba[0] = true;
 ba[1] = false;
foreach (Object obj in ba)
 {
    Console.WriteLine(obj);
 }
```

Enumeration

An enumeration type (or enum type) is a value type defined by a set of named constants of the underlying integral numeric type. To define an enumeration type, use the enum keyword and specify the names of enum members:

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

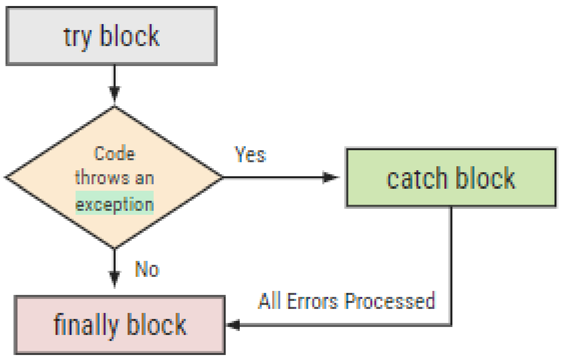
By default, the associated constant values of enum members are of type int; they start with zero and increase by one following the definition text order. You can explicitly specify any other integral numeric type as an underlying type of an enumeration type.

Data Table :

——————————————————-pending——————————————————-

Exception Handling :

A try block is used by C# programmers to partition code that might be affected by an exception. Associated catch blocks are used to handle any resulting exceptions. A finally block contains code that is run whether or not an exception is thrown in the try block, such as releasing resources that are allocated in the try block. A try block requires one or more associated catch blocks, or a finally block, or both.



- try

-
    - The try block encloses the statements that might throw an exception.
- catch
    - The catch block handles the exceptions thrown by the try block.
    - In a program, a try block can contain one or multiple catch blocks.
- finally
    - It is an optional block and is always executed.
    - If no exception occurs inside the try block, the program control is transferred directly to the finally block.

Example :

```
try
{
   // Code to try goes here.
}
catch (SomeSpecificException ex)
{
   // Code to handle the exception goes here.
}
finally
{
   // Code to execute after the try (and possibly catch) blocks
   // goes here.
}
```

Different Project types

Project Type	Description
Class library	Component library with no user interface
Console application	Command line application
Database project	SQL script storage
Device application	Windows application for a smart device
Empty project	Blank project
SQL Server project	Management of stored procedures and SQL Server objects
Web service	ASP.NET Web application with no user interface; technically, no longer a project type
Web site	ASP.NET Web application; technically, no longer a project type
Windows	Windows application with a user interface application
Windows service	Windows application with no user interface
WPF Browser Application	Windows Presentation Foundation browser application

Working with String Class

- In C#, a string is a sequence of Unicode characters or array of characters.
- The range of Unicode characters will be U+0000 to U+FFFF. The array of characters is also termed as the text. So the string is the representation of the text. A string is represented by a class System.String.
- The String class is defined in the .NET base class library. In other words, a String object is a sequential collection of System.Char objects which represent a string. The maximum size of the String object in memory can be 2GB or about 1 billion characters.

Characteristics of String Class:

- The System.String class is immutable, i.e once created its state cannot be altered.
- With the help of length property, it provides the total number of characters present in the given string.
- String objects can include a null character which counts as the part of the string's length.
- It provides the position of the characters in the given string.

Constructor	Description
String(Char*)	Initializes a new instance of the String class to the value indicated by a specified pointer to an array of Unicode characters.
String(Char*, Int32, Int32)	Initializes a new instance of the String class to the value indicated by a specified pointer to an array of Unicode characters, a starting character position within that array, and a length.
String(Char, Int32)	Initializes a new instance of the String class to the value indicated by a specified Unicode character repeated a specified number of times.

String(Char[])	Initializes a new instance of the String class to the value indicated by an array of Unicode characters.
String(Char[], Int32, Int32)	Initializes a new instance of the String class to the value indicated by an array of Unicode characters, a starting character position within that array, and a length.
String(SByte*)	Initializes a new instance of the String class to the value indicated by a pointer to an array of 8-bit signed integers.
String(SByte*, Int32, Int32)	Initializes a new instance of the String class to the value indicated by a specified pointer to an array of 8-bit signed integers, a starting position within that array, and a length.
String(SByte*,Int32,Int32,encoding)	Initializes a new instance of the String class to the value indicated by a specified pointer to an array of 8-bit signed integers, a starting position within that array, a length, and an Encoding object.

Working with DateTime Class

The DateTime class is the most commonly used date class in C#. It represents a specific date and time and has many properties and methods that allow you to perform operations on it. DateTime objects can be created using various constructors, which accept parameters such as year, month, day, hour, minute, second, and millisecond.

Here's an example:

DateTime date1 = new DateTime(2023, 3, 11, 10, 30, 0);
(This code creates a DateTime object representing March 11, 2023, 10:30 AM.)

DateTime Static Fields and Methods

DateTime struct includes static fields, properties, and methods. Static methods come in handy for quick time operations. Some static methods are listed below with their explanation.

Examples :

```
DateTime currentDateTime = DateTime.Now;
 //returns current date and time

DateTime todaysDate = DateTime.Today;
 // returns today's date

DateTime currentDateTimeUTC = DateTime.UtcNow;
// returns current UTC date and time

DateTime maxDateTimeValue = DateTime.MaxValue;
// returns max value of DateTime

DateTime minDateTimeValue = DateTime.MinValue;
// returns min value of DateTime
```

Basic File operations

- Generally, the file is used to store the data.
- The term File Handling refers to the various operations like creating the file, reading from the file, writing to the file, appending the file, etc.
- There are two basic operation which is mostly used in file handling is reading and writing of the file.
- The file becomes stream when we open the file for writing and reading. A stream is a sequence of bytes which is used for communication. Two stream can be formed from file one is input stream which is used to read the file and another is output stream is used to write in the file.
- There are mainly 2 class which are useful for reading and writing from the text file.
    - StreamWriter Class
    - StreamReader Class

StreamWriter Class :

The StreamWriter class implements TextWriter for writing character to stream in a particular format.

Method	Description
Close()	Closes the current StreamWriter object and stream associate with it.
Flush()	Clears all the data from the buffer and write it in the stream associate with it.
Write()	Write data to the stream. It has different overloads for different data types to write in stream.
WriteLine()	It is same as Write() but it adds the newline character at the end of the data.

StreamReader Class

The StreamReader class implements TextReader for reading character from the stream in a particular format.

Method	Description

Close()	Closes the current StreamReader object and stream associate with it.
Peek()	Returns the next available character but does not consume it.
Read()	Reads the next character in input stream and increment characters position by one in the stream
ReadLine()	Reads a line from the input stream and return the data in form of string
Seek()	It is use to read/write at the specific location from a file

Introduction to Web Development

- Even though C# is a language that's relatively easy to learn and maintain, it isn't just for beginners. Its scalability and large support community make C# the language of choice for Microsoft app developers and video game developers working with the Unity Engine. Like C++ and Java,
- C# is a high-level object-oriented programming language.
- It is generally more efficient than Java and has useful features such as operator overloading.
- C# is based on C++ but has several advantages over this older language:
    - it is type-safe, more comprehensively object-oriented, and the syntax has been simplified in several important ways.
- Most importantly, C# interoperates exceptionally well with other languages on the .NET platform.
- For this reason, C# is a better choice for building applications for .NET.