Basic C#

**Introduction**

C# is an object-oriented programming language developed by Microsoft as part of the .NET framework. It is used for building a wide variety of applications, including desktop software, web applications, mobile apps, and cloud-based services. C# integrates well with the .NET framework, allowing for seamless development of modern applications.

**C# Execution**

* compiler converts .cs file to an intermediate language (IL or MSIL)
* this IL is stored in form of .dll (library) or .exe
* Common Lannguage Runtime(CLR) converts IL to machine code using JIT-Compiler specific to os and hardware.
* CLR invokes Main method to start execution.
* After execution CLR releases occupied resources (like memory)

**DataTypes in C#**

Value type: byte, sbyte, short, ushort, int, uint, long, ulong, float, double, decimal, char, bool, enum

Reference type: string, object, array

Nullable type: int?, double?, etc.

Pointer type: int\*, char\*, etc. (unsafe code only)

Custom type: class, struct, interface, enum

**TypeCasting in C#**

Typecasting is the process of converting one type of data to another. It is commonly done in two main ways:

1. Implicit Typecasting (or Automatic Typecasting):

Implicit typecasting occurs when a smaller type is automatically converted to a larger type without requiring any explicit action from the programmer. This type of casting is safe and does not lose data.

int num1 = 10;

double num2 = num1;

1. Explicit Typecasting (or Manual Typecasting):

Explicit typecasting occurs when you manually convert a type to another type, typically when narrowing down the range of the data. This process may result in data loss, so the compiler requires explicit instructions.

double num1 = 9.78;

int num2 = (int)num1;

1. Using Convert class:

It provides methods to convert common types like int, double, string, bool, etc. Unlike explicit casting, Convert methods handle some common conversions safely and can throw exceptions when the conversion is not possible.

string str = "123";

int num = Convert.ToInt32(str)

**Operators in C#**

Operators are special symbols or keywords used to perform operations on variables and values. In C#, operators are classified into several categories:

* Arithmetic Operators: +, -, \*, /, %
* Comparison Operators: ==, !=, <, >, <=, >=
* Logical Operators: &&, ||, !
* Assignment Operators: =, +=, -=, \*=, /=, %=
* Unary Operators: ++, --, +, -, !
* Ternary Operator: ?:
* Bitwise Operators: &, |, ~, <<, >>
* Type Comparison Operators: is( if(obj is int){} ), as(string s = obj as string)
* Null-Coalescing Operators: ?? (string str = name ?? “default”)

**Expressions in C#**

An expression is a combination of operators, operands, and functions that evaluates to a value. Below are the types of expressions in C#:

* Literal Expressions: int x = 5;
* Variable Expressions: int sum = a+b;
* Method Call Expressions: int max = Math.Max(10, 20);
* Conditional Expressions: string result = (10 > 5) ? "Big" : "Small";
* Lambda Expressions: Func<int, int, int> add = (a, b) => a + b;

**Statements**

A statement represents a complete instruction that the program executes.

* Conditional (if-else) Statements

Conditional statements are used to execute a block of code based on whether a condition is true or false.

* Switch Statement

The switch statement allows a variable or expression to be compared against multiple values, executing different code based on the match.

* Iteration Statements (Loops)

C# provides several loop constructs for repeated execution of a block of code. Eg. for, while, do-while

* Jump Statements

Jump statements are used to transfer control to another part of the program. Eg. break, continue, return

**Arrays**

Arrays are a data structure that allows you to store multiple values of the same type in a single variable. Arrays are commonly used to work with collections of data when the number of elements is known beforehand.

type[] arrayName = new type[size];

Multidimensional Arrays: type[,] arrayName = new type[rows, columns];

Jagged Arrays: A jagged array is an array of arrays, where each element in the main array is itself an array.

type[][] arrayName = new type[rows][];

Defining and Calling of Methods

Methods are blocks of reusable code that perform specific tasks.

**Understanding Classes & OOP Concepts**

Classes

A class is a blueprint or prototype from which objects are created. It can contain:

* Fields (variables that hold data).
* Methods (functions that define the behaviors or actions).
* Properties (attributes that represent the state of an object).

OOP Principles

Encapsulation

Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on the data into a single unit or class. This ensures that the internal state of an object is protected from outside interference and misuse.

Inheritance

Inheritance is the mechanism by which one class can derive from another, inheriting its fields and methods. The derived class can also add its own fields and methods or override inherited ones.

Polymorphism

Polymorphism allows methods to behave differently based on the object that calls them. This means a subclass can provide a specific implementation of a method that is already defined in its superclass.

Abstraction

Abstraction is the concept of hiding the complex implementation details and showing only the essential features of an object. It helps in reducing complexity and focusing on high-level functionalities.

Interfaces

An interface defines a contract that classes must adhere to. It can declare methods and properties, but it does not contain any implementation. Classes that implement the interface must provide their own implementations for the methods defined in the interface.

Access Modifiers

Public, private, protected, internal

**Namespace & .NET Library**

Namespaces help organize code and avoid name conflicts.

The .NETLibrary is a collection of pre-built classes that handle common tasks like file I/O, networking, and UI development.

**Collections In C#**

Collections are data structures that store and manage groups of objects.

There are two main types of collections in C#:

* **Non-generic Collections:** These collections store elements of type Object. They are part of the System.Collections namespace.

1. **ArrayList**: An ArrayList is a collection that dynamically resizes itself when elements are added or removed. It is similar to an array but can grow or shrink as needed.
2. **Hashtable**: A Hashtable is a collection of key-value pairs, where each key is unique, and you can retrieve its associated value using the key.
3. Stack: A Stack represents a collection of objects that follows the LIFO (Last In, First Out) principle. It is commonly used for situations like undo functionality.
4. Queue: A Queue represents a collection that follows the FIFO (First In, First Out) principle.

* **Generic Collections:** These collections store elements of a specific type. They are part of the System.Collections.Generic namespace and are more type-safe and efficient than non-generic collections.

1. List<T>: A List<T> is a collection that can store elements of a specified type. It allows random access and dynamic resizing.
2. Dictionary<TKey, TValue>: A Dictionary<TKey, TValue> is a collection of key-value pairs, where each key is unique, and you can retrieve its associated value using the key. It is similar to a Hashtable but type-safe.
3. HashSet<T>: A HashSet<T> is a collection that contains no duplicate elements. It is useful for ensuring that only unique elements are stored.
4. SortedList<TKey, TValue>: A SortedList<TKey, TValue> is a collection of key-value pairs that are automatically sorted by the key. It provides fast lookup, insertion, and removal of elements.
5. Queue<T>: A Queue<T> is a generic collection that follows the FIFO (First In, First Out) principle.

**ArrayList (System.Collections)**

* Add()
* Remove()
* RemoveAt()
* Insert()
* InsertRange()
* Clear()
* Contains()
* IndexOf()
* Sort()
* ToArray()
* Reverse()

**Hashtable (System.Collections)**

* Add()
* Remove()
* ContainsKey()
* ContainsValue()
* Clear()
* GetEnumerator()
* Keys()
* Values()
* TryGetValue()

**Stack (System.Collections)**

* Push()
* Pop()
* Peek()
* Clear()
* Contains()
* ToArray()
* GetEnumerator()

**Queue (System.Collections)**

* Enqueue()
* Dequeue()
* Peek()
* Clear()
* Contains()
* ToArray()
* GetEnumerator()

**List<T> (System.Collections.Generic)**

* Add()
* Remove()
* RemoveAt()
* Insert()
* InsertRange()
* Clear()
* Contains()
* IndexOf()
* Sort()
* ToArray()
* Reverse()
* GetEnumerator()
* Find()
* FindAll()
* AddRange()
* Capacity()

**Dictionary<TKey, TValue> (System.Collections.Generic)**

* Add()
* Remove()
* ContainsKey()
* ContainsValue()
* TryGetValue()
* Clear()
* Keys()
* Values()
* GetEnumerator()
* Count()

**HashSet<T> (System.Collections.Generic)**

* Add()
* Remove()
* Contains()
* Clear()
* UnionWith()
* IntersectWith()
* ExceptWith()
* SymmetricExceptWith()
* ToArray()
* GetEnumerator()

**SortedList<TKey, TValue> (System.Collections.Generic)**

* Add()
* Remove()
* ContainsKey()
* ContainsValue()
* Clear()
* Keys()
* Values()
* GetEnumerator()
* TryGetValue()

**Queue<T> (System.Collections.Generic)**

* Enqueue()
* Dequeue()
* Peek()
* Clear()
* Contains()
* ToArray()
* GetEnumerator()

**LinkedList<T> (System.Collections.Generic)**

* AddLast()
* AddFirst()
* AddBefore()
* AddAfter()
* Remove()
* RemoveFirst()
* RemoveLast()
* Find()
* FindLast()
* Clear()
* Contains()
* ToArray()
* GetEnumerator()

**Enumerations:**

An enum (short for enumeration) is a special "class" that represents a group of constants (unchangeable variables). Enums are used to assign human-readable names to numeric values.

An enum is declared using the enum keyword, followed by the name of the enum, and then the set of constants within curly braces {}. We can explicitly assign values to the constants in an enum.

**Enum Methods**

There are built-in methods in C# to work with enums:

* Enum.GetValues(typeof(var)): Returns an array of all values in the enum.
* Enum.GetNames(): Returns an array of all names in the enum.
* Enum.IsDefined(): Checks if a value is defined in the enum.
* Enum.Parse(): Converts a string to its corresponding enum value.
* Enum.TryParse(): Tries to convert a string to its corresponding enum value.

Enums can be used to represent a combination of values using bitwise operations, often used for settings or options. This is done by applying the [Flags] attribute to the enum.

[Flags]

enum Permissions { Read = 1, Write = 2, Execute = 4, Delete = 8 }

Permissions myPermissions = Permissions.Read | Permissions.Write; // Combination of Read and Write

Console.WriteLine(myPermissions); // Output: Read, Write

**DataTable**

A DataTable is an in-memory representation of a table of data. It allows you to store, manipulate, and query data in a structured format, similar to a table in a database.

Basic Concept of DataTable

* Rows: A DataTable contains rows, each of which can hold data for one record.
* Columns: A DataTable contains columns, each of which holds data for a specific field or attribute (e.g., Name, Age, Salary).
* Data: The actual data in the DataTable is organized into cells in a grid format (rows and columns)

Methods:

Clear(), Select(), NewRow(), AcceptChanges(), GetChanges(), ImportRow(), Merge(), Clone(), Load(), RejectChanges(), GetErrors(), HasErrors()

Properties:

Columns, Rows, TableName, PrimaryKey, Constraints, Prefix, CaseSensitive, DataSet, HasErrors, ColumnsCount, RowCount, IsInitialized

**Exception Handling**

Exceptionhandling is a mechanism that allows you to handle runtime errors, also known as exceptions, in a structured way. It enables your program to continue executing instead of terminating abruptly when an error occurs.

Components: try, catch, finally, throw statement

ExceptionClass

The base class for all exceptions in C# is the Exception class. This class has several useful properties:

* Message: A description of the exception.
* StackTrace: A string that describes the sequence of method calls that led to the exception.
* InnerException: The exception that caused the current exception (if any).
* Source: The name of the application or object that caused the exception.
* TargetSite: The method that threw the exception.

**Different Project Types**

C# supports different types of projects, including:

* ConsoleApplication: For CLI-based programs.
* WindowsForms: For GUI-based applications.
* WebApplications: Using ASP.NET for building web apps.
* ClassLibraries: Reusable code that can be shared across projects.

**Strings in C#**

Common String Methods:

* ToUpper(), ToLower(): Converts to uppercase or lowercase.
* Trim(), TrimStart(), TrimEnd(): Removes leading and/or trailing whitespace.
* Substring(): Extracts part of the string.
* Replace(): Replaces occurrences of a substring.
* IndexOf(), LastIndexOf(): Finds the index of a substring.
* Contains(): Checks if a substring exists.
* StartsWith(), EndsWith(): Checks if the string starts or ends with a substring.
* Split(): Splits a string into an array of substrings.
* Join(): Combines multiple strings into a single string.

Common String Properties:

* Length: Gets the number of characters in the string.
* IsNullOrEmpty: A static property that checks if a string is either null or has a length of 0.
* IsNullOrWhiteSpace**:** A static property that checks if a string is null, empty, or consists only of white-space characters.

**DateTime in C#**

The DateTime structure in C# represents an instance in time, typically expressed as a date and time of day. It provides various methods and properties for working with dates and times.

StaticProperties

* MaxValue  
  Represents the largest possible value of DateTime, which is December 31, 9999 at 23:59:59.9999999.
* MinValue  
  Represents the smallest possible value of DateTime, which is January 1, 0001 at 00:00:00.

Properties

* Date: Gets the date component (year, month, and day) of the DateTime.
* Day: Gets the day of the month represented by the DateTime object.
* DayOfWeek: Gets the day of the week (e.g., Sunday, Monday) represented by the DateTime.
* DayOfYear  
  Gets the day of the year (from 1 to 366) for the specified DateTime.
* Hour  
  Gets the hour component of the DateTime object (from 0 to 23).
* Minute  
  Gets the minute component of the DateTime.
* Month  
  Gets the month component (1 to 12) of the DateTime.
* Second  
  Gets the second component of the DateTime.
* Ticks  
  Gets the number of ticks (100-nanosecond intervals) since January 1, 0001 at 00:00:00.000.
* Year  
  Gets the year component of the DateTime.

Methods

* AddDays(double value)  
  Adds the specified number of days to the current DateTime.
* AddHours(double value)  
  Adds the specified number of hours to the current DateTime.
* AddMinutes(double value)  
  Adds the specified number of minutes to the current DateTime.
* AddMonths(int months)  
  Adds the specified number of months to the current DateTime.
* AddSeconds(double value)  
  Adds the specified number of seconds to the current DateTime.
* AddTicks(long value)  
  Adds the specified number of ticks to the current DateTime.
* AddYears(int value)  
  Adds the specified number of years to the current DateTime.
* Compare(DateTime t1, DateTime t2)  
  Compares two DateTime objects and indicates if one is earlier, later, or the same as the other.
* ToString()  
  Returns a string representation of the DateTime object, formatted based on the current culture.
* ToString(string format)  
  Returns a string representation of the DateTime object, formatted using the specified format string.
* Parse(string s)  
  Converts the string representation of a date and time to its DateTime equivalent.
* TryParse(string s, out DateTime result)  
  Tries to convert a string representation of a date and time to its DateTime equivalent and returns a boolean indicating success.
* IsLeapYear(int year)  
  Determines if the specified year is a leap year.
* Now  
  Gets the current date and time.
* UtcNow  
  Gets the current date and time in UTC (Coordinated Universal Time).

**File Operations in C#**

In C#, file operations are generally performed using the System.IO namespace, which provides classes for working with files and directories.

1. File Creation

* File.Create(string path)  
  Creates a new file at the specified path. If the file already exists, it will be overwritten.

2. File Writing

* File.WriteAllText(string path, string contents)  
  Writes the specified text to a file. If the file exists, it overwrites the file.
* File.AppendAllText(string path, string contents)  
  Appends text to an existing file. If the file doesn’t exist, it creates the file.

3. File Reading

* File.ReadAllText(string path)  
  Reads the entire content of a file as a string.
* File.ReadAllLines(string path)  
  Reads all lines of a file and returns them as an array of strings.
* File.ReadLines(string path)  
  Reads lines of a file lazily, which is useful for large files.

4. File Deletion

* File.Delete(string path)  
  Deletes the specified file if it exists.

5. File Copying

* File.Copy(string sourceFileName, string destFileName)  
  Copies a file from one location to another. If the destination file exists, an exception will be thrown unless you specify true for the overwrite parameter.
* File.Copy(string sourceFileName, string destFileName, bool overwrite)  
  Copies a file and allows you to specify whether to overwrite an existing file at the destination.

6. File Moving

* File.Move(string sourceFileName, string destFileName)  
  Moves a file from one location to another.

7. Checking File Existence

* File.Exists(string path)  
  Checks if a file exists at the specified path.

8. File Information

* FileInfo class  
  Provides properties and methods for obtaining file properties and performing operations on a file, such as file size, creation time, and last access time. (To be covered in Advance API part)

9. File Streams (Advanced Reading and Writing)

* FileStream  
  Used for more granular control over file reading and writing, such as reading in chunks or handling binary data. (To be covered in Advance API part)

10. Directory Operations (Basic)

* Directory.CreateDirectory(string path)  
  Creates a directory at the specified path if it does not exist.
* Directory.Exists(string path)  
  Checks if a directory exists at the specified path.
* Directory.GetFiles(string path)  
  Retrieves the list of files in a directory. (To be covered in Advance API part)