Basics of C#

• Introduction to C#

C# (pronounced "C-sharp") is a modern, object-oriented programming language developed by Microsoft. It was introduced in the early 2000s as part of the Microsoft .NET framework and has since become one of the most popular programming languages for building a wide range of applications, including desktop, web, mobile, and cloud-based applications. The first version was released in year 2002. The latest version, C# 12, was released in November 2023.

C# is a versatile and powerful language suitable for a wide range of application development scenarios. Whether you're building desktop applications, web services, mobile apps, or cloud-based solutions, C# provides a robust foundation for creating modern, scalable, and maintainable software.

C# is used for:

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

• Create first C# program ‘Hello world’

In code file which is saved with “.cs” extension, write code which is given below to print “Hello world”.

```
using System;

namespace HelloWorld

{

  class Program

  {

    static void Main(string[] args)

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

  }

}
```

- Understanding C# program structure
  - **using System** means that we can use classes from the System namespace.
  - **namespace** is used to organize your code, and it is a container for classes and other namespaces.
  - The **curly braces {}** marks the beginning and the end of a block of code.
  - **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.
  - Another thing that always appear in a C# program, is the **Main method**. Any code inside its curly brackets {} will be executed. You don't have to understand the keywords before and after Main. You will get to know them bit by bit while reading this tutorial.
  - **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 ;.**

- Working with Code files, Projects & Solutions
  - Code files: -

    Files with . cs extension are source code files for C# programming language. Introduced by Microsoft for use with the . NET Framework, the file format provides the low-level programming language for writing code that is compiled to generate the final output file in the form of EXE or a DLL.

  - Projects: -

    When you create a C# application in Visual Studio Code, you start with a project. A project contains all files (such as source code, images, etc.) that are compiled into an executable, library, or website. All of your related projects can then be stored in a container called a solution.

  - Solution: -

    A project is contained within a solution. Despite its name, a solution isn't an "answer." It's simply a container for one or more related projects, along with build information, Visual Studio window settings, and any miscellaneous files that aren't associated with a particular project. Visual Studio uses two file types for solution file (.sln and .suo) to store settings for solutions:

Extension	**Name**	**Description**
.sln	Visual Studio Solution	Organizes projects, project items, and solution items in the solution.
.suo	Solution User Options	Stores user-level settings and customizations, such as breakpoints.

- Datatypes & Variables with conversion
  - Datatypes: -

Data Type	Size	Description
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
bool	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter, surrounded by single quotes
string	2 bytes per character	Stores a sequence of characters, surrounded by double quotes

  - Variables: -

Variables are named storage locations that hold values of a particular data type. Here's how you declare and use variables:

```
// Declaration

int age;

double price = 19.99;

string name = "John";

bool isStudent = true;


// Assignment

age = 30;

name = "Jane";


// Usage

Console.WriteLine($"Name: {name}, Age: {age}, Price: {price}, Is Student:
{isStudent}");
```

- Type conversion: -

Type conversion allows you to convert values from one data type to another. In C#, there are two types of casting:

- **Implicit Casting (automatically)** - converting a smaller type to a larger type size
  char -> int -> long -> float -> double

```
int myInt = 9;
double myDouble = myInt;      // Automatic casting: int to double

Console.WriteLine(myInt);      // Outputs 9
Console.WriteLine(myDouble);   // Outputs 9
```

- **Explicit Casting (manually)** - converting a larger type to a smaller size type
  double -> float -> long -> int -> char

```
double myDouble = 9.78;
int myInt = (int) myDouble;    // Manual casting: double to int

Console.WriteLine(myDouble);   // Outputs 9.78
Console.WriteLine(myInt);      // Outputs 9
```

- **Type conversion method:** 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):

```
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
```

- Operators and Expression
    - Arithmetic operator:

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	x++
--	Decrement	Decreases the value of a variable by 1	x--

    - Assignment operator:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

    - Comparison operator:

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

    - Logical operator:

Operator	Name	Description	Example
&&	Logical and	Returns True if both statements are true	x < 5 && x < 10
||	Logical or	Returns True if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns False if the result is true	!(x < 5 && x < 10)

- Statements

Category	**C# keywords / notes**
Declaration statements	A declaration statement introduces a new variable or constant. A variable declaration can optionally assign a value to the variable. In a constant declaration, the assignment is required.
Expression statements	Expression statements that calculate a value must store the value in a variable.
Selection statements	Selection statements enable you to branch to different sections of code, depending on one or more specified conditions. if switch

Iteration statements	Iteration statements enable you to loop through collections like arrays, or perform the same set of statements repeatedly until a specified condition is met.
	do
	for
	foreach
	while

Jump statements	Jump statements transfer control to another section of code. break continue goto return yield
Exception-handling statements	Exception-handling statements enable you to gracefully recover from exceptional conditions that occur at run time. throw ,try-catch try-finally try-catch-finally

checked and unchecked	The checked and unchecked statements enable you to specify whether integral-type numerical operations are allowed to cause an overflow when the result is stored in a variable that is too small to hold the resulting value.
The await statement	If you mark a method with the async modifier, you can use the await operator in the method. When control reaches an await expression in the async method, control returns to the caller, and progress in the method is suspended until the awaited task completes. When the task is complete, execution can resume in the method.
The yield return statement	An iterator performs a custom iteration over a collection, such as a list or an array. An iterator uses the yield return statement to return each element one at a time. When a yield return statement is reached, the current location in code is remembered. Execution is restarted from that location when the iterator is called the next time.
The fixed statement	The fixed statement prevents the garbage collector from relocating a movable variable.
The lock statement	The lock statement enables you to limit access to blocks of code to only one thread at a time.
Labeled statements	You can give a statement a label and then use the goto keyword to jump to the labeled statement.
The empty statement	The empty statement consists of a single semicolon. It does nothing and can be used in places where a statement is required but no action needs to be performed.

- Understanding Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value. To declare an array, define the variable type with square brackets:

DataType [] name;

- 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>.

  - **Single-dimensional Array: -**

```
int[] array = new int[5];

string[] weekDays = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];
```

  - **Multidimensional Array: -**

```
int[,] array2DDeclaration = new int[4, 2];


int[,,] array3DDeclaration = new int[4, 2, 3];


// Two-dimensional array.
int[,] array2DInitialization =  { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };
```

  - **Jagged Array: -**

A jagged array is an array whose elements are arrays, possibly of different sizes. A jagged initialized to null.

```
int[][] jaggedArray = new int[3][];
jaggedArray[0] = [1, 3, 5, 7, 9];
jaggedArray[1] = [0, 2, 4, 6];
jaggedArray[2] = [11, 22];
```

```
int[][] jaggedArray2 =
[
    [1, 3, 5, 7, 9],
    [0, 2, 4, 6],
    [11, 22]
];
```

- Define & calling of Methods

  A method is a block of code which only runs when it is called.You can pass data, known as parameters, into a method. Methods are used to perform certain actions, and they are also known as functions. Why use methods? To reuse code: define the code once, and use it many times.

  - Define a Method

    A method is defined with the name of the method, followed by parentheses (). C# provides some pre-defined methods, which you already are familiar with, such as Main(), but you can also create your own methods to perform certain actions:

```
class Program
{
  static void MyMethod()
  {
    // code to be executed
  }
}
```

  - Call a Method

    To call (execute) a method, write the method's name followed by two parentheses () and a semicolon; In the following example, MyMethod() is used to print a text (the action), when it is called:

```
static void MyMethod()
{
  Console.WriteLine("I just got executed!");
}

static void Main(string[] args)
{
  MyMethod();
}

// Outputs "I just got executed!"
```

- Understanding classes & OOP concepts
  - Class
    - The class is like a blueprint of a specific object that has certain attributes and features.
    - In c# class is nothing but a collection of various data members (fields, properties, etc.) and member functions.
    - Class is a data structure, and it will combine various types of data members such as fields, properties, member functions, and events into a single unit.
    - It defines the kinds of data and the functionality their objects will have.

      ```
      public class users
      {
         // Properties, Methods, Events, etc.
      }
      ```

  - Objects
    - Objects are real-world entities and instance of a class.
    - For example, chair, car, pen, mobile, laptop etc.
    - We can create one or more objects of a class.
    - Each object can have different values of properties and field but, methods and events behaves the same.

  - Constructor
    - A constructor is a special type of method which will be called automatically when you create an instance of a class.
    - A constructor is defined by using an access modifier and class name
    - <access-modifier> <class-name>(){ }.
    - A constructor name must be the same as a class name.
    - A constructor can be public, private, or protected.
    - The constructor cannot return any value so, cannot have a return type.
    - A class can have multiple constructors with different parameters but can only have one parameter less constructor.
    - If no constructor is defined, the C# compiler would create it internally

- Interface & Inheritance
  - Interface

Another way to achieve abstraction in C#, is with interfaces. An interface is a completely "abstract class", which can only contain abstract methods and properties (with empty bodies):

```
interface Animal
{
  void animalSound(); // interface method (does not have a body)
  void run(); // interface method (does not have a body)
}
```

- In C# versions earlier than 8.0, an interface is like an abstract base class with only abstract members. A class or struct that implements the interface must implement all its members.
- Beginning with C# 8.0, an interface may define default implementations for some or all of its members. A class or struct that implements the interface doesn't have to implement members that have default implementations. For more information, see default interface methods.
- An interface can't be instantiated directly. Its members are implemented by any class or struct that implements the interface.
- A class or struct can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.

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.

  - **Single inheritance**

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

  - **Multi-level inheritance**

    In Multi-level inheritance, a derived class is created from another derived class.
  - **Multiple inheritance**

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

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

- **Hierarchical inheritance**

```
Class A
    Class C
        Class D
        Class E
    Class B
        Class F
        Class G
```

- **Hybrid inheritance**

```
Class A
    Class C
        Class D
        Class E
Class F

        Class B
```

- Scope & Accessibility modifier

Access modifiers in C# are used to specify the scope of accessibility of a member of a class or type of the class itself. For example, a public class is accessible to everyone without any restrictions, while an internal class may be accessible to the assembly only. Access modifiers are an integral part of object-oriented programming. Access modifiers are used to implement encapsulation of OOP. Access modifiers allow you to define who does or who doesn't have access to certain features.

public	There are no restrictions on accessing public members.
private	Access is limited to within the class definition. This is the default access modifier type if none is formally specified
protected	Access is limited to within the class definition and any class that inherits from the class
internal	Access is limited exclusively to classes defined within the current project assembly
protected internal	Access is limited to the current assembly and types derived from the containing class. All members in current project and all members in derived class can access the variables.
private protected	Access is limited to the containing class or types derived from the containing class within the current assembly.

- public modifier

The public keyword is an access modifier for types and type members. Public access is the most permissive access level. There are no restrictions on accessing public members.

- Accessibility
  - Can be accessed by objects of the class
  - Can be accessed by derived classes

- private modifier

Private access is the least permissive access level. Private members are accessible only within the body of the class or the struct in which they are declared.

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

- protected modifier

  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.

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

- internal modifier

  The internal keyword is an access modifier for types and type members. We can declare a class as internal or its member as internal. Internal members are accessible only within files in the same assembly (.dll). In other words, access is limited exclusively to classes defined within the current project assembly.

    - Accessibility
      In same assembly (public)
        - Can be accessed by objects of the class
        - Can be accessed by derived classes

      In other assembly (internal)
        - Cannot be accessed by object
        - Cannot be accessed by derived classes

- protected internal modifier

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 modifier seems to be a confusing but is a union of protected and internal in terms of providing access but not restricting. It allows:

- Inherited types, even though they belong to a different assembly, have access to the protected internal members.
- Types that reside in the same assembly, even if they are not derived from the type, also have access to the protected internal members.

- Namespace & .Net Library
  - Namespace:

    Namespaces are used to organize the classes. It helps to control the scope of methods and classes in larger .Net programming projects. In simpler words you can say that it provides a way to keep one set of names(like class names) different from other sets of names. The biggest advantage of using namespace is that the class names which are declared in one namespace will not clash with the same class names declared in another namespace. It is also referred as named group of classes having common features. The members of a namespace can be namespaces, interfaces, structures, and delegates.

  - .Net library:

    .NET Framework Class Library is the collection of classes, namespaces, interfaces and value types that are used for .NET applications. It contains thousands of classes that supports the following functions.

    - Base and user-defined data types
    - Support for exceptions handling
    - input/output and stream operations
    - Communications with the underlying system
    - Access to data
    - Ability to create Windows-based GUI applications
    - Ability to create web-client and server applications
    - Support for creating web services

  - .NET Framework Class Library Namespaces

Namespaces	**Description**
System	It includes all common datatypes, string values, arrays and methods for data conversion.

System.Data, System.Data.Common, System.Data.OleDb, System.Data.SqlClient, System.Data.SqlTypes	These are used to access a database, perform commands on a database and retrieve database.
System.IO, System.DirectoryServices, System.IO.IsolatedStorage	These are used to access, read and write files.
System.Diagnostics	It is used to debug and trace the execution of an application.
System.Net, System.Net.Sockets	These are used to communicate over the Internet when creating peer-to-peer applications.
System.Windows.Forms, System.Windows.Forms.Design	These namespaces are used to create Windows-based applications using Windows user interface components.
System.Web, System.WebCaching, System.Web.UI, System.Web.UI.Design, System.Web.UI.WebControls, System.Web.UI.HtmlControls,	These are used to create ASP. NET Web applications that run over the web.

System.Web.Configuration, System.Web.Hosting, System.Web.Mail, System.Web.SessionState	
System.Web.Services, System.Web.Services.Description, System.Web.Services.Configuration, System.Web.Services.Discovery, System.Web.Services.Protocols	These are used to create XML Web services and components that can be published over the web.
System.Security, System.Security.Permissions, System.Security.Policy, System.WebSecurity, System.Security.Cryptography	These are used for authentication, authorization, and encryption purpose.
System.Xml, System.Xml.Schema, System.Xml.Serialization, System.Xml.XPath, System.Xml.Xsl	These namespaces are used to create and access XML files.

- 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 are similar to arrays, it provide a more flexible way of working with a group of objects.
    - In arrays, you need to define the number of 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.
    - The .NET Framework provides you a variety of collection classes, which are available in the System.Collections namespace.
    - Most useful collection classes defined are as follows.

Collection	**Description**
ArrayList	The ArrayList collection is similar to the Arrays data type in C#. The biggest difference is the dynamic nature of the array list collection.
Stack	The stack is a special case collection which represents a last in first out (LIFO) concept
Queues	The Queue is a special case collection which represents a first in first out concept
Hashtable	A hash table is a special collection that is used to store key-value items
SortedList	The SortedList is a collection which stores key-value pairs in the ascending order of key by default.
BitArray	A bit array is an array of data structure which stores bits.

    - ArrayList:
        - 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.

- Stack:
    - 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.

- Queues:
    - 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.

- Hashtable:
    - 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.
    - There is no direct way to display the elements of a Hashtable.
        - { "1" , “Darshan" }
        - { "2" , “Institute " }
        - { "3" , "of Engg. & Technology" }
    - This is done via the ICollection interface. (This is a special data type which can be used to store the keys of a Hashtable collections)

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

- BitArray:
    - 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.

- Enumerations

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:

```
public enum DaysOfWeek
{
  Sunday,    // 0
  Monday,    // 1
  Tuesday,   // 2
  Wednesday, // 3
  Thursday,  // 4
  Friday,    // 5
  Saturday   // 6
}

DaysOfWeek today = DaysOfWeek.Wednesday;
Console.WriteLine($"Today is {today}"); // Outputs: Today is Wednesday
```

- Data Table

The DataTable class in C# ADO.NET is a database table representation and provides a collection of columns and rows to store data in a grid form. The code sample in this artilce explains how to create a DataTable at run-time in C#. How to create a DataTable columns and rows, add data to a DataTable and bind a DataTable to a DataGridView control using data binding.

  - DataTable class Properties:

PROPERTY	DESCRIPTION
Columns	Represents all table columns
Constraints	Represents all table constraints
DataSet	Returns the dataset for the table
DefaultView	Customized view of the data table
ChildRelation	Return child relations for the data table
ParentRelation	Returns parent relations for the data table
PrimaryKey	Represents an array of columns that function as primary key for the table

Rows	All rows of the data table
TableName	Name of the table

- DataTable class Methods:

METHOD	**DESCRIPTION**
AcceptChanges	Commits all the changes made since last AcceptChanges was called
Clear	Deletes all data table data
Clone	Creates a clone of a DataTable including its schema
Copy	Copies a data table including its schema
NewRow	Creates a new row, which is later added by calling the Rows.Add method
RejectChanges	Reject all changed made after last AcceptChanges was called
Reset	Resets a data table's original state
Select	Gets an array of rows based on the criteria

- Exception Handling

It is very common for software applications to get errors and exceptions when executing code. If these errors are not handled properly, the application may crash and you may not know the root cause of the problem. Exception handling is the method of catching and recording these errors in code so you can fix them. Usually, errors and exceptions are stored in log files or databases. In C#, the exception handling method is implemented using the **try, catch, finally and throw** statement. In this article, learn how to implement exception handling in C#.

- Basic Try- catch block:

```
try
{
   // Code that might throw an exception
   int result = Divide(10, 0);
   Console.WriteLine(result);
}
catch (DivideByZeroException ex)
{
   // Handle the specific exception
   Console.WriteLine($"Error: {ex.Message}");
}
catch (Exception ex)
```

```
{
    // Handle other types of exceptions
    Console.WriteLine($"Unexpected Error: {ex.Message}");
}
finally
{
    // Code that will always be executed, regardless of whether an exception
occurred
    Console.WriteLine("Finally block executed");
}

int Divide(int a, int b)
{
    return a / b;
}
```

- catch Blocks:
  - Multiple catch blocks can be used to handle different types of exceptions.
  - The order of catch blocks matters; the first matching block will be executed.
- finally Block:
  - The finally block contains code that will always be executed, regardless of whether an exception occurred.
  - It is useful for cleanup operations (closing files, releasing resources, etc.).
- throw Statement:
  - The throw statement is used to explicitly throw an exception.
- Custom exceptions:
  - You can create custom exception classes by deriving from the Exception class.

- Exception Properties:

  Exceptions provide properties like Message, StackTrace, and InnerException for detailed error information.

```
try

{

    // ...

}

catch (Exception ex)
```

```
{
    Console.WriteLine($"Error: {ex.Message}");
    Console.WriteLine($"Stack Trace: {ex.StackTrace}");
    Console.WriteLine($"Inner Exception: {ex.InnerException?.Message}");
}
```

- Working with String Class
  - 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.
    - It allows empty strings. Empty strings are the valid instance of String objects that contain zero characters.
    - A string that has been declared but has not been assigned a value is null. Attempting to call methods on that string throws a NullReferenceException.
    - It also supports searching strings, comparison of string, testing of equality, modifying the string, normalization of string, copying of strings, etc.
    - It also provides several ways to create strings like using a constructor, using concatenation, etc.

  - Constructor:

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.

- Methods of String Class:

Method	**Description**
Clone()	Returns a reference to this instance of String.
Compare()	Used to compare the two string objects.
CompareOrdinal(String, Int32, String, Int32, Int32)	Compares substrings of two specified String objects by evaluating the numeric values of the corresponding Char objects in each substring.
CompareOrdinal(String, String)	Compares two specified String objects by evaluating the numeric values of the corresponding Char objects in each string.
CompareTo()	Compare the current instance with a specified Object or String object.
Concat()	Concatenates one or more instances of String, or the String representations of the values of one or more instances of Object.
Contains(String)	Returns a value indicating whether a specified substring occurs within this string.
Copy(String)	Creates a new instance of String with the same value as a specified String.
CopyTo(Int32, Char[], Int32, Int32)	Copies a specified number of characters from a specified position in this instance to a specified position in an array of Unicode characters.

EndsWith()	Determines whether the end of this string instance matches a specified string.
Equals()	Determines whether two String objects have the same value.
Format()	Converts the value of objects to strings based on the formats specified and inserts them into another string.
GetEnumerator()	Retrieves an object that can iterate through the individual characters in this string.
GetHashCode()	Returns the hash code for this string.
GetType()	Gets the Type of the current instance. (Inherited from Object)
GetTypeCode()	Returns the TypeCode for class String.
IndexOf()	Reports the zero-based index of the first occurrence of a specified Unicode character or string within this instance. The method returns -1 if the character or string is not found in this instance.
IndexOfAny()	Reports the index of the first occurrence in this instance of any character in a specified array of Unicode characters. The method returns -1 if the characters in the array are not found in this instance.
Insert(Int32, String)	Returns a new string in which a specified string is inserted at a specified index position in this instance.
Intern(String)	Retrieves the system's reference to the specified String.
IsInterned(String)	Retrieves a reference to a specified String.
IsNormalized()	Indicates whether this string is in a particular Unicode normalization form.
IsNullOrEmpty(String)	Indicates whether the specified string is null or an Empty string.
IsNullOrWhiteSpace(String)	Indicates whether a specified string is null, empty, or consists only of white-space characters.
Join()	Concatenates the elements of a specified array or the members of a collection, using the specified separator between each element or member.
LastIndexOf()	Reports the zero-based index position of the last occurrence of a specified Unicode character or string within this instance. The

	method returns -1 if the character or string is not found in this instance.
MemberwiseClone()	Creates a shallow copy of the current Object. (Inherited from Object)
Normalize()	Returns a new string whose binary representation is in a particular Unicode normalization form.
PadLeft()	Returns a new string of a specified length in which the beginning of the current string is padded with spaces or with a specified Unicode character.
PadRight()	Returns a new string of a specified length in which the end of the current string is padded with spaces or with a specified Unicode character.
Remove()	Returns a new string in which a specified number of characters from the current string are deleted.
Replace()	Returns a new string in which all occurrences of a specified Unicode character or String in the current string are replaced with another specified Unicode character or String.
Split()	Returns a string array that contains the substrings in this instance that are delimited by elements of a specified string or Unicode character array.
StartsWith(String)	Determines whether the beginning of this string instance matches a specified string.
Substring(Int32)	Retrieves a substring from this instance.
ToCharArray()	Copies the characters in this instance to a Unicode character array.
ToLower()	Returns a copy of this string converted to lowercase.
ToLowerInvariant()	Returns a copy of this String object converted to lowercase using the casing rules of the invariant culture.
ToString()	Converts the value of this instance to a String.
ToUpper()	Returns a copy of this string converted to uppercase.
ToUpperInvariant()	Returns a copy of this String object converted to uppercase using the casing rules of the invariant culture.

Trim()	Returns a new string in which all leading and trailing occurrences of a set of specified characters from the current String object are removed.
TrimEnd(Char[])	Removes all trailing occurrences of a set of characters specified in an array from the current String object.
TrimStart(Char[])	Removes all leading occurrences of a set of characters specified in an array from the current String object.

- ## Working with DateTime Class

  C# DateTime is a structure of value Type like int, double etc. It is available in System namespace and present in mscorlib.dll assembly. It implements interfaces like IComparable, IFormattable, IConvertible, ISerializable, IComparable, IEquatable.

  - ### DateTime formatting:

    Different users need different kinds of format date. For instance some users need date like "mm/dd/yyyy", some need "dd-mm-yyyy". Let's say current Date Time is "12/8/2015 3:15:19 PM" and as per specifier you will get below output.

    - `DateTime tempDate = new DateTime(2015, 12, 08); // creating date object with 8th December 2015`
    - `Console.WriteLine(tempDate.ToString("MMMM dd, yyyy")); //December 08, 2105.`

Specifier	Description	Output
d	Short Date	12/8/2015
D	Long Date	Tuesday, December 08, 2015
t	Short Time	3:15 PM
T	Long Time	3:15:19 PM
f	Full date and time	Tuesday, December 08, 2015 3:15 PM
F	Full date and time (long)	Tuesday, December 08, 2015 3:15:19 PM
g	Default date and time	12/8/2015 15:15
G	Default date and time (long)	12/8/2015 15:15
M	Day / Month	8-Dec
r	RFC1123 date	Tue, 08 Dec 2015 15:15:19 GMT
s	Sortable date/time	2015-12-08T15:15:19
u	Universal time, local timezone	2015-12-08 15:15:19Z

Y	Month / Year	December, 2015
dd	Day	8
ddd	Short Day Name	Tue
dddd	Full Day Name	Tuesday
hh	2 digit hour	3
HH	2 digit hour (24 hour)	15
mm	2 digit minute	15
MM	Month	12
MMM	Short Month name	Dec
MMMM	Month name	December
ss	seconds	19
fff	milliseconds	120
FFF	milliseconds without trailing zero	12
tt	AM/PM	PM
yy	2 digit year	15
yyyy	4 digit year	2015
:	Hours, minutes, seconds separator, e.g. {0:hh:mm:ss}	9:08:59
/	Year, month , day separator, e.g. {0:dd/MM/yyyy}	8/4/2007

- DateTime Constructor

It initializes a new instance of DateTime object. At the time of object creation we need to pass required parameters like year, month, day, etc. It contains around 11 overload methods. More details available here.

```
// 2015 is year, 12 is month, 25 is day
DateTime date1 = new DateTime(2015, 12, 25);
Console.WriteLine(date1.ToString()); // 12/25/2015 12:00:00 AM

// 2015 - year, 12 - month, 25 – day, 10 – hour, 30 – minute,
50 - second
DateTime date2 = new DateTime(2012, 12, 25, 10, 30, 50);
Console.WriteLine(date1.ToString());// 12/25/2015 10:30:00 AM }
```

- Basic File Operation

Basic file operations in C# involve tasks such as creating, reading, writing, copying, moving, and deleting files. Here's a brief description of these fundamental file operations:

- **Creating Files:**

  To create a new file, you can use the File.Create method. This method returns a FileStream that allows you to write data to the file.

- **Reading Files:**

  Reading files involves opening an existing file and retrieving its content. Common methods include File.ReadAllText and File.ReadAllLines for reading the entire content or lines of text from a file.

- **Writing to Files:**

  To write data to a file, you can use methods like File.WriteAllText or File.WriteAllLines. These methods allow you to overwrite the existing content or create a new file if it doesn't exist.

- **Appending to Files:**

  If you want to add content to an existing file without overwriting its contents, you can use methods like File.AppendAllText or File.AppendAllLines.

- **Copying Files:**

  Copying files involves creating a duplicate of an existing file. The File.Copy method is commonly used for this purpose.

- **Moving or Renaming Files:**

  The File.Move method is used to move or rename a file. Moving involves changing the file's location, while renaming keeps the file in the same directory but changes its name.

- **Deleting Files:**

  To delete a file, you can use the File.Delete method. This permanently removes the file from the file system.

- **Checking File Existence:**

  The File.Exists method checks whether a file exists at a specified path. It returns true if the file exists and false otherwise.

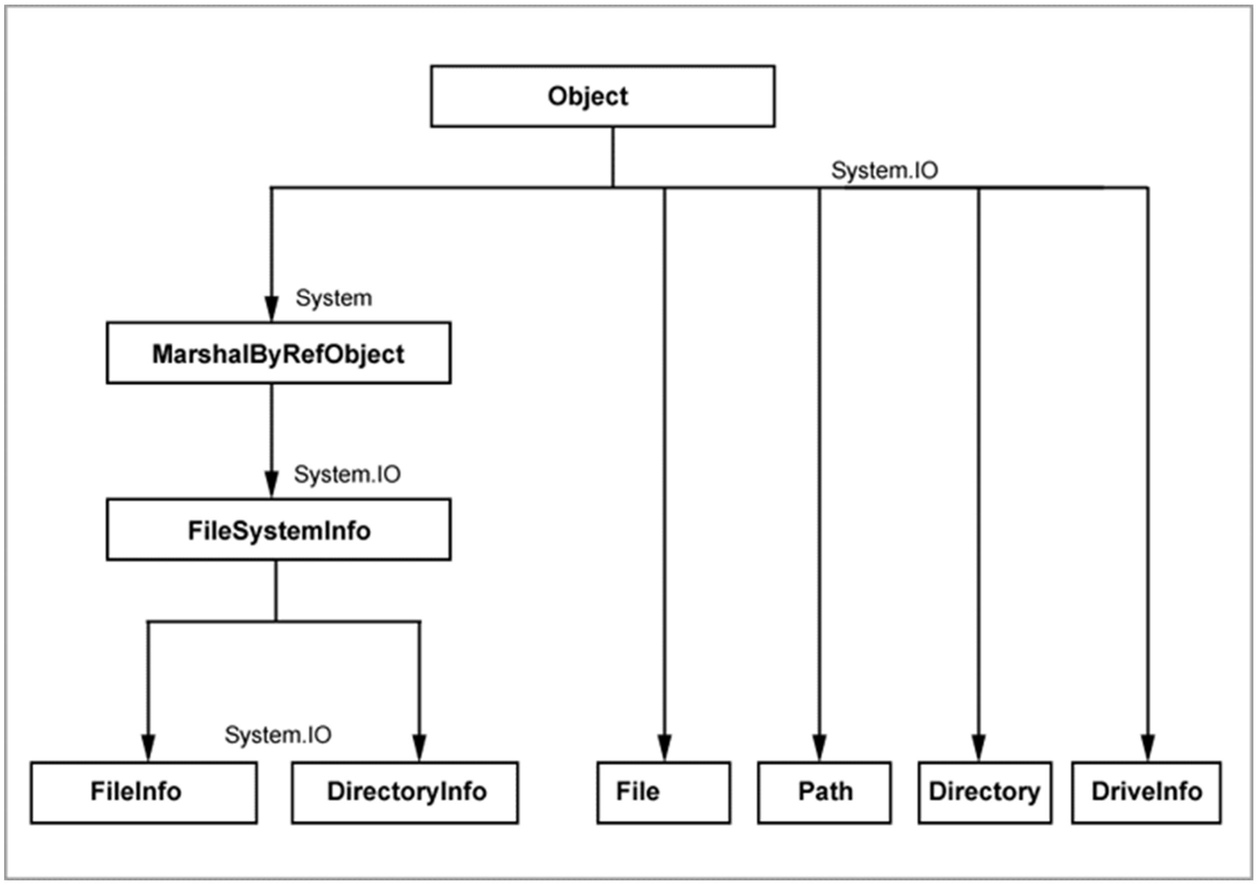
- **File Information:**

  The FileInfo class provides additional information about a file, such as its size, creation time, last access time, and last write time.

- **Working with Directories:**

  In addition to individual files, C# provides methods and classes for working with directories. The Directory class offers operations like creating directories, getting file lists, and deleting directories.

  - Diagram to represent file-handling class hierarchy :



- Common Class used in System.io namespace:

Class Name	**Description**
FileStream	It is used to read from and write to any location within a file
BinaryReader	It is used to read primitive data types from a binary stream
BinaryWriter	It is used to write primitive data types in binary format
StreamReader	It is used to read characters from a byte Stream
StreamWriter	It is used to write characters to a stream.
StringReader	It is used to read from a string buffer
StringWriter	It is used to write into a string buffer
DirectoryInfo	It is used to perform operations on directories
FileInfo	It is used to perform operations on files