C#

1. What is a Computer?

- An **electronic device** that performs operations/calculations based on user instructions.
- Basic process: **Input** → **Process** → **Output** → **Storage**.
- Components:
  - **Hardware** – physical parts (CPU, RAM, storage, I/O devices)
  - **Software** – OS & applications that instruct hardware.
- Types: Smartphones, PCs, Servers, Supercomputers.

---

2. Major Components of a Computer

(a) CPU – Central Processing Unit (Brain of Computer)

- Executes programs step by step.
- Components:
  - **ALU (Arithmetic Logic Unit):** Performs calculations & logical operations.
  - **CU (Control Unit):** Fetches, decodes, and executes instructions, controls data flow.

(b) Main Memory (RAM)

- Temporary/volatile storage (erased when power is off).
- Stores data & programs while running.
- Called **working memory**; CPU can access only via RAM, not directly from disk.

(c) Hard Disk (Secondary Storage)

- Permanent/non-volatile storage.
- Stores OS, applications, and user files.
- Two types of files:
  - **Program Files** – executable code (e.g., `.exe`, `.dll`, `.py`).
  - **Data Files** – documents, images, videos, etc. (e.g., `.txt`, `.jpg`, `.mp3`).

(d) Input Devices

- Provide data/commands (keyboard, mouse, scanner, mic).
- Data first goes to **input buffer** before CPU reads it.

(e) Output Devices

- Display results (monitor, printer, speakers).
- Use **output buffer** to hold data before showing it.

---

3. How It All Works Together

1. OS loads the program from Hard Disk → RAM.
2. CPU fetches instructions from RAM and executes them.
3. Inputs are taken (if any).
4. Processing is done.
5. Output is sent to screen/printer/speakers or stored back.

---

4. Number Systems in Computers

- Humans use **Decimal (Base 10: 0–9)**. Computers understand **Binary (Base 2: 0,1)**.
- Other systems:
  - **Octal (Base 8: 0–7)**
  - **Hexadecimal (Base 16: 0–9, A–F)**

Examples:

- Decimal **10 = Binary 1010**.
- Decimal **29 → Binary 11101** (by repeated division by 2).
- Binary **10101 → Decimal 21** (by place value expansion).

✅ **Summary**:
A computer works by taking input → processing through CPU & memory → producing output → and storing data.
It operates only in **binary**, so number system concepts are essential before learning programming.

Introduction to Programming Languages – Short Notes

1. What is a Language?

- A way of communication using rules/instructions.
- Human languages: English, Hindi, Odia, Telugu, etc.

2. What is Computer Language?

- A **formal language** to communicate with computers.
- Computer understands only **binary (0 & 1)**.
- We use **programming languages** to write instructions → compiler/translator converts them into machine code.

3. Why do we need Programming Languages?

- Directly writing in binary is difficult. High-level languages (C, C++, Java, Python, etc.) make programming easier.
- Programs = set of instructions → converted → executed by CPU.

4. What is an Interface?

- End-users interact with machines through **interfaces** (e.g., ATM screen, apps).
- Programmer's role = **develop applications & interfaces** so users can use machines easily.

5. Types of Computer Languages

- **Low-Level Languages** (closer to machine):
  - *Machine Language*: binary (0,1), hardware-dependent, hard to write.
  - *Assembly Language*: symbolic (ADD, MOV), easier than binary, requires assembler.
- **High-Level Languages** (closer to human):
  - English-like syntax, easy to learn, portable.
  - Examples: C, C++, Java, C#, Python.

Difference:

- High-Level → easy for humans, needs translator.
- Low-Level → easy for machines, hard for humans.

6. What is Software?

- **Software = collection of programs**. Example: Calculator = many small programs.

- Types:
  - **System Software** → runs hardware & platform (OS, compiler, assembler, linker).
  - **Application Software** → user-specific tasks (MS Office, Oracle, games).

✅ **Summary:**
Programming languages bridge humans & computers.
We write instructions in high-level languages → translators convert to binary → computer executes.
Software is built from these programs to make machines useful for both programmers & end-users.

How Computer Programs Work – Short Notes

1. What is a Program?

- A **set of instructions** for a computer to perform tasks (open app, calculate, etc.).
- Written in **high-level languages** (C, Java, Python).
- Computer understands only **binary (0,1)** → needs **translators**.

---

2. Translators (System Software)

Convert **source code** → **machine code**.
Types:

- **Compiler** → Converts entire program at once.
  - Fast execution.
  - Errors shown after full compilation.
  - Examples: C, C++.
- **Interpreter** → Converts line by line.
  - Stops at error.
  - Slower but good for debugging.
  - Examples: Python, Java, .NET.
- **Assembler** → Converts **assembly language** → machine code.

Compiler	Interpreter
Scans the whole program and generates machine code in one go	Scans each statement one by one
Takes time to analyze the program However, execution time is faster.	Translation time is faster but overall execution time is slower.
Generates intermediate code called object code	No object code is generated.

---

3. Operating System (OS)

- **Interface between user & hardware**.
- Handles memory, files, processes, I/O, devices.
- Examples: Windows, Linux, MacOS, Android, iOS.
- Auto-loaded when computer starts.Performs all the basic tasks like file management, memory management, process management, handling input and output.

---

4. Loader & Linker

- **Loader** → Loads machine code into memory.
- **Locator** → Assigns memory addresses.
- **Linker** → Combines smaller modules/programs into one complete program.

✅ **Summary:**
Programs = instructions → written in high-level language → translators (compiler/interpreter/assembler) convert to binary → OS manages execution → loader loads → linker links → computer runs program.

Different Types of Applications – Brief Notes

1. Types of Applications

- **Standalone Applications**
- **Web Applications**

---

2. Standalone Applications

- Installed on a computer.
- Dependent on Operating System (OS-specific)
- Examples: VLC, MS Office, Google Chrome.
- **File extensions**:
  - `.txt` → Notepad
  - `.mp4` → VLC
- **OS extensions**:
  - Windows → `.exe`
  - Mac → `.dmg`
  - Linux → `.rpm`

---

3. Web Applications

- Run on a **web browser**, no installation needed.
- Independent of OS.
- Examples: Gmail, YouTube, Facebook, Google.

---

4. Programming Languages

- **All programming languages are standalone applications** (need installation).
- **Platform-dependent languages (C, C++)** → Can develop only standalone apps.
- **Platform-independent languages (Java, C#, PHP, .NET)** → Can develop both standalone & web applications.

---

5. Use Cases

- **C** → Embedded systems.
- **C++** → Gaming libraries.
- **Java / .NET** → Enterprise & Web apps (e.g., Banking, IRCTC, Facebook).

Programming Methodologies – Brief Notes

1. What are Programming Methodologies?

- Different **styles/paradigms** of writing code.
- Major types:
    - **Monolithic Programming**
    - **Modular / Procedural Programming**
    - **Object-Oriented Programming (OOPs)**

---

2. Monolithic Programming

- Entire program written in **one block (main function)**.
- **Advantages**:
    - Simple, fast to code.
- **Disadvantages**:
    - Complex & messy for large programs.
    - Difficult debugging, testing, and maintenance.
    - No code reusability, redundancy issues.
    - Only one programmer can handle at a time.

---

3. Modular / Procedural Programming

- Code is divided into **functions (modules)**.
- **Language Example**: C.
- **Advantages**:
    - Code reusability & readability.
    - Easy debugging & maintenance.
    - Large projects possible (team collaboration).
    - Reduces complexity by dividing tasks.
- **Main Function**: Acts as **manager**, controls execution order.

---

4. Object-Oriented Programming (OOP)

- Code organized into **classes** (data + functions together).
- Objects are created from classes.
- **Languages**: C++, Java, C#, Python.
- **Advantages**:
    - Encapsulation → keeps data & functions together.
    - Reusability via classes.
    - Easier upgrades & scalability.
    - Higher productivity & lower maintenance cost.
    - Supports abstraction & modularity.

---

5. Key Difference in Paradigms

- **Monolithic** → One block, simple but messy.
- **Modular** → Functions divide tasks, easier to manage.
- **OOP** → Classes & objects, more structured, reusable, scalable.
- *Logic remains same, only style of writing code changes.*

✅ **Summary:**

- Early programming used **Monolithic style** → messy for big projects.
- Shifted to **Modular/Procedural** → functions improved readability & reusability.
- Now widely using **OOP** → classes & objects provide better structure, scalability, and efficiency.

Algorithms, Pseudocode, Programs & Flowcharts – Brief Notes

1. Algorithm

- Step-by-step procedure to solve a problem.
- Independent of any programming language.
- Example: Add all numbers in a list → divide by count → average.

---

2. Program

- Set of instructions written in a programming language.
- Follows strict **syntax rules** (compiler must understand).
- Converts algorithm into **machine-executable form**.

Difference:

- **Algorithm =** plan/steps (logic).
- **Program =** actual code (syntax-based).

---

3. Pseudocode

- Informal, text-based representation of algorithm.
- Written in plain English-like language, not tied to syntax.
- Helps in designing before actual coding.

---

4. Flowchart

- **Diagrammatic representation** of an algorithm/process.
- Shows flow of control → easy to understand program logic.
- Especially useful for **big/complex programs**.

Basic Flowchart Steps

1. **Input** → take data.
2. **Process** → perform operations.
3. **Output** → display results.

Flowchart Symbols

- **Oval (Terminal)** → Start/Stop.
- **Parallelogram** → Input/Output.
- **Rectangle** → Processing (calculations).
- **Diamond** → Decision (Yes/No, True/False).
- **Arrows** → Flow direction.

✅ **Summary:**

- **Algorithm** → Logical steps to solve a problem.
- **Pseudocode** → Informal design of algorithm.
- **Program** → Actual coded instructions with syntax.
- **Flowchart** → Visual representation of program logic.

🔹 Before .NET Framework

- **COM (Component Object Model)** was Microsoft's earlier framework. Used **VB** for Windows apps and **ASP** for Web apps.
- **Disadvantages of COM**:
    1. Incomplete Object-Oriented Programming.
    2. Platform dependent (Windows only).

🔹 What is .NET?

- **.NET = Network Enabled Technology**.
- A **free, open-source, cross-platform developer platform** for building Web, Mobile, Desktop, IoT, Cloud, and Games.
- Supports multiple **languages** (C#, VB, F#, etc.), **editors**, and **libraries**.

🔹 What is a Framework?

- A **collection of technologies** combined together to develop applications.
- Provides ready-made libraries and runtime support.

🔹 What .NET Framework Provides

1. **BCL (Base Class Library)**
    - Predefined classes for application development.
    - Located at: `C:\Windows\assembly`
    - Basic building blocks of .NET programs.
2. **CLR (Common Language Runtime)**
    - Core component that converts **MSIL (Intermediate Code)** → **Native Code**.
    - Handles **memory management, security, and exception handling**.

🔹 Compilation in .NET

1. **First Compilation**: Source code → MSIL (Managed Code).
2. **Second Compilation**: MSIL → Native Code (via CLR).
    - Done using **JIT (Just-In-Time Compiler)**.
    - 1st compilation is slow, 2nd is fast.

🔹 Flavors of .NET

1. **.NET Framework** – Original version, Windows-only.
2. **.NET (formerly .NET Core)** – Cross-platform (Windows, Linux, macOS).
3. **Xamarin/Mono** – For mobile (iOS, Android).

Note: .NET Framework = Platform Dependent,
.NET/.NET Core = Platform Independent.

🔹 What .NET is NOT

- Not an OS
- Not a Database
- Not an ERP or Testing tool
- Not a Programming Language

🔹 What .NET Actually Is

- A **Framework Tool** supporting many **languages** (60+ total, 9 by Microsoft):
    - **VB.NET, C#.NET, VC++.NET, J#.NET, F#.NET, JScript.NET, PowerShell, IronPython, IronRuby**
- Supports many **technologies**:
    - **ASP.NET** (Web, MVC, Web API, Blazor)
    - **ADO.NET** (Database access)
    - **WCF, WPF, WWF, AJAX, LINQ, Entity Framework**, etc.

🔹 Language vs Technology

- **Language**:
  - Bridge between programmer & system.
  - Provides **syntax + libraries**. (e.g., C#, VB, F#)
- **Technology**:
  - Designed for a **specific purpose**.
  - Example: **ASP.NET** for Web Apps.
  - Needs a language to be implemented.

✅ In short:

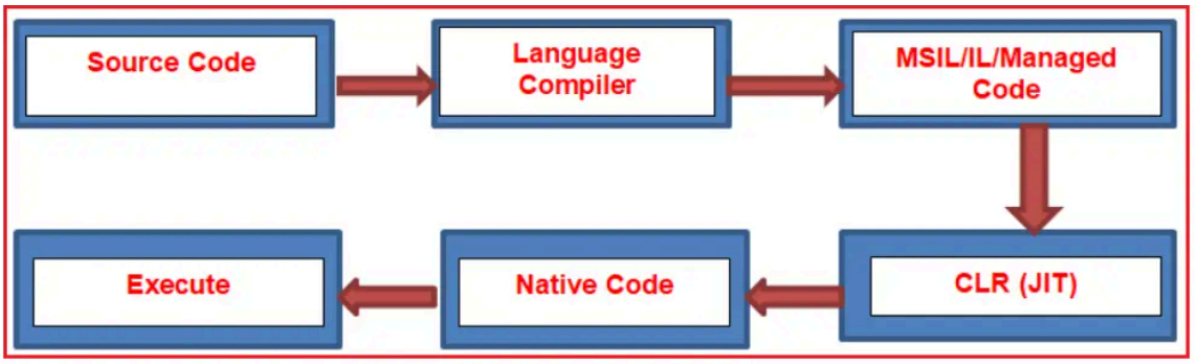
- **.NET Framework** = Microsoft's solution to build applications with **OOP support, cross-platform ability, and strong libraries**.
- Provides **BCL + CLR** as the foundation.
- Supports **multiple languages and technologies** to create different types of apps.

Differences Between .net core and .net framework:-

.NET Core	**.NET Framework**
• It is open Source • Works in all platforms • Robust classes use a redefined common language runtime called CoreCLR (.Net execution Engine or Compiler) and have a modular collection of libraries CoreFX. • ASP.NET Core used for both web and cloud applications.(available in windows,mac and linux) • Deploy ASP.NET Core applications directly in the cloud or self-host the applications by creating their own hosting process. • Mobile application development can be developed by using Xamarin • Easier for building microservices. • .net core is better than .net framework	• It is Licensed and proprietary software • Works only on windows • Robust Classes use CLR. • ASP.NET is a web application framework used to develop web application with .NET framework • Deploy web applications only on Internet Information server • Don't have any framework to simplify Mobile application development • Not much easier to build microservices.

CORE CLR :- Core CLR is the Common Language Runtime for multiplatform and cloud-based deployments. It is the .NET execution engine and performs functions such as garbage collection and Intermediate compilation to machine code. It includes the garbage collection, primitive data types and basic classes.

Roslyn is a compiler platform that includes the C# and VB compilers and other tools. These compilers generate Common Intermediate Language (CIL) code.
To run this code, CIL needs to be compiled into binary code that the target computer architecture can execute.
While the app is running, the CIL code is compiled into binary code using a JIT(Just In Time) compiler. This model is implemented by Core CLR. Core CLR was a copy of CLR and it has been modified to support different operating systems. They are maintained separately and in parallel.



.NET Native Runtime :- It is also referred to as Managed Runtime. It contains the native windows-based libraries. It also contains the Ahead Of Time (AOT) compilation in place of the Just In Time (JIT compilation that improves the performance of the applications.

Unified BCL :- The Unified Base Class Library is also referred to as CoreFX. It consists of the Basic and fundamental classes that form the core of the.Net Core platform.

App Models :- Various App Models sit on top of the BCL. Developers use these models to develop platform-specific applications. .NET core has the ASP.NET Model and Windows Store Model. ASP.NET model is used for web development and Windows Store Model for windows application development.

How .NET Works

WCF (Windows Communication Foundation): It is a framework for building service-oriented applications. Using WCF, you can send data as asynchronous messages from one service endpoint to another.

LINQ (Language Integrated Query): It is a query language, introduced in .NET 3.5 framework. It is used to make the query for data sources with C# or Visual Basics programming languages.

Entity Framework: It is an ORM-based open-source framework that is used to work with a database using .NET objects. It eliminates a lot of developers' effort to handle the database. It is Microsoft's recommended technology to deal with the database.

Parallel LINQ: Parallel LINQ or PLINQ is a parallel implementation of LINQ to objects. It combines the simplicity and readabilitySource Code (C#, VB, F#) → Compiled into **IL/MSIL/CIL (Intermediate Language)** → Stored in **Assemblies (.DLL / .EXE)**.

1. At runtime, **CLR + JIT compiler** converts IL → **Native Machine Code**.
2. Native code runs on the OS.

First compilation is slower, but execution becomes faster.

Applications Developed using .NET Framework

The types of applications that can be built in the .Net framework are classified broadly into the following categories.

WinForms – This is used for developing Forms-based applications, which would run on an end-user machine. Notepad is an example of a client-based application. Windows Forms is a smart client technology for the .NET Framework, a set of managed libraries that simplify common application tasks such as reading and writing to the file system.

ASP.NET – This is used for developing web-based applications, which are made to run on any browser such as Edge, Chrome or Firefox. ASP.NET is a web framework designed and developed by Microsoft. It is used to develop websites, web applications, and web services. It provides a fantastic integration of HTML, CSS, and JavaScript. It was first released in January 2002.

1. The Web application would be processed on a server, which would have Internet Information Services Installed.
2. Internet Information Services or IIS is a Microsoft component that is used to execute an ASP.NET application.
3. The result of the execution is then sent to the client machines, and the output is shown in the browser.

ADO.NET: This technology is used to develop applications to interact with databases such as Oracle or Microsoft SQL Server. ADO.NET is a module of the .Net Framework, which is used to establish a connection between applications and data sources. Data sources can be such as SQL Server and XML. ADO .NET consists of classes that can be used to connect, retrieve, insert, and delete data. of LINQ and provides the power of parallel programming. It can improve and provide fast speed to execute the LINQ query by using all available computer capabilities.

🔹 Applications Built with .NET Framework

1. **WinForms** – Desktop/Forms-based apps (e.g., Notepad).
2. **ASP.NET** – Web apps & services (runs via IIS).
3. **ADO.NET** – Database connectivity (SQL Server, Oracle, XML, etc.).
4. **WCF (Windows Communication Foundation)** – Service-oriented apps.
5. **LINQ** – Query data sources directly with C#/VB.
6. **Entity Framework** – ORM framework for database operations with objects.
7. **PLINQ (Parallel LINQ)** – LINQ with parallel execution for better performance.

Introduction to C# Programming Language

C# (pronounced C-Sharp) is a modern, object-oriented programming language developed by Microsoft in 2002 with .NET Framework 1.0. It is designed to overcome the drawbacks of C and C++, provide platform independence, and develop web and enterprise applications.

Why C#?

- **Simple & Familiar** – Based on C/C++, automatic memory management, safer than pointers.
- **Portable** – Source and IL code work across different OS.
- **Architecturally Neutral** – Same behavior on all systems (due to CLR & CTS).
- **Secure & Robust** – Runs in CLR, strict type checking.
- **Distributed & Multithreaded** – Supports services, concurrency, and multiple tasks.
- **Dynamic** – Supports runtime type resolution (`dynamic` keyword).
- **Compiled & Interpreted** – Uses MSIL + JIT.
- **Object-Oriented** – Encapsulation, Abstraction, Inheritance, Polymorphism.
- **Platform Independent** – Runs on Windows, Linux, Mac using .NET/.NET Core.
- **Automatic Memory Management** – Uses Garbage Collector.
- **Exception Handling** – Prevents program crashes with structured handling.

Features of C#

- Simple, Modern, Object-Oriented
- Type-Safe, Secure, Component-Oriented
- Structured, Scalable, and Interoperable
- Rich library with fast execution

Types of Applications in C#

- Windows Apps
- Web Apps & Web APIs
- Distributed and Cloud Apps
- Mobile Apps (iOS, Android)
- Database Applications
- Gaming, IoT, AI/ML, Blockchain

Console Application (what & why)

- **Console app** runs in a command prompt (CUI).
- Ideal for beginners to learn C# basics (logic, I/O, flow).
- No GUI elements (buttons, menus). Simple, fast to run/test.

Basic Structure of a C# Console Program (minimal example)

```
using System;

namespace WebApplication1{
    class Program{
        static void Main(string[] args){
            Console.WriteLine("Hello World");
            Console.ReadLine();
        }
    }
}
```

- Notes: C# is **case-sensitive**. Statements end with a **semicolon**.
- Console.WriteLine() // Gives Output
- Console.ReadLine() // Gives Output and wait until we press a key on Keyboard.

Short Explanations of Sections

- **using**: imports namespaces (BCL or user namespaces) so you can use their classes.
- **namespace**: groups related classes/types.
- **class**: blueprint that contains data (fields) and behavior (methods).
- **Main()**: entry point where program execution starts.

Advantages of .NET from C# Developer's View

- Full GUI support (WinForms/WPF).
- Easy DB access (ADO.NET, Entity Framework).
- Web development (ASP.NET/ASP.NET Core).
- Strong tooling (Visual Studio, debugging, profiling).
- Large ecosystem (NuGet packages, libraries).

Quick: How to create & run a Console App (Visual Studio)

1. **New Project → Console App (.NET/.NET Framework)**
2. Give project name, select framework, Create.
3. Edit `Program.cs` (Main method) → write code, which is a **static file.**
4. Run with **Start** or **Ctrl+F5**.
5. **Main is class Program(Starting point of program) -> Like int main(). Using system** (Imports packages from classes).

📘 Console Class in C#

🔹 What is Console Class?

- Belongs to **System namespace**.
- Used for **input/output** in Console Applications.
- **Static class** → only has static members, so you call directly with `Console.Method()` or `Console.Property`.
- Represents **standard input, output, and error streams**.

🔹 Properties of Console Class

Some important ones:

Property	Description

Title	Gets/sets console window title.
BackgroundColor	Gets/sets console background color. Default: Black.
ForegroundColor	Gets/sets text color. Default: Gray.
CursorSize	Gets/sets cursor height (1–100%).

Example:

```
Console.Title = "My Console App";

Console.BackgroundColor = ConsoleColor.DarkBlue;

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Clear(); // Apply color changes
```

🔹 Methods of Console Class

Some commonly used ones:

Method	Description
Clear()	Clears the console screen.
Beep()	Plays a beep sound.
ResetColor()	Restores default colors.

Write()	Prints text, stays on the same line.
WriteLine()	Prints text, moves to the next line.
Read()	Reads a single character, returns ASCII value (**int**).
ReadLine()	Reads entire line, returns **string**.
ReadKey()	Reads a single key press, returns **ConsoleKeyInfo**.

Output Statements - WriteLine() and Write() :-

Console.WriteLine() - Displays information on the next line in the console screen
Console.Write() - Displays information on the current line in the console screen

```
Console.WriteLine("Displays on ");
Console.WriteLine("New line");
Console.Write(" Displays on ");
Console.Write("Current line");
```

When we execute the above statements, the output would be,

```
Displays on
New line
Displays on Current line
```

- `Console.WriteLine ("Hello" + "Thomas");` // Output would be, Hello Thomas
- `string empName="Thomas"; Console.WriteLine("Hello" + empName);` // Output would be, Hello Thomas
- Formatting Console Outputs :- We can use placeholders for variables to format console outputs
  For example :- `string empName = "Thomas";`
  `Console.WriteLine("Hello {0}", empName);`
  Output would be :- Hello Thomas
  {0} is the placeholder for the variable empName. It is replaced by the value of the variable empName on execution.

Multiple variables can be included in the formatted string., using multiple place holders.
For example :- `string empName = "Thomas";`
`decimal salary = 4500.85m;`
`Console.WriteLine("Hello {0}, Your salary is {1}", empName, salary);`
Output :- Hello Thomas, Your salary is 4500.85

- **Number formatting can be done using string.Format() or ToString() methods.**
- Use the format specifier "." (point) to set the position of the decimal point and "," (comma) for thousand separator
- The format specifier 0(zero) represents a digit. If a digit is missing at this position of the result, a zero is printed instead
- The format specifier # also represents a digit. But it does not print anything If a digit is missing at this position of the result
- Examples are :- `string empName = "Thomas";`
  `decimal salary = 42786500.85m;`
  `Console.WriteLine("Hello {0}, Your salary is {1}", empName, salary); //` Hello Thomas, Your salary is 42786500.85
- `Console.WriteLine("Hello {0}, Your salary is {1}", empName, string.Format("{0:#,0.00}", salary)); //` Here {0} is actually placeholder where salary can be filled. {0 : #,0.00} so before : is placeholder after : is formatter where we are saying round upto 2 decimal places. #,0 whereas 0 and # represents 1 digit so if value has more digits from formatter then formatter won't effect hence only thousand will get divided.
  Output is :- *Hello Thomas, Your salary is 42,786,500.85*
- `Console.WriteLine("Hello {0}, Your salary is {1}", empName, salary.ToString("#,0.00")); //` No need of Placeholder and same output will be there as formatter is same. (Hello Thomas, Your salary is 42,786,500.85)
- If number in formatter is more than digits in value then what will happen :-
  `string empName = "Thomas";`
  `decimal salary = 4253.876m;`
- `Console.WriteLine("Hello {0}, Your salary is {1} ", empName, salary.ToString("####,0.00")); //` Hello Thomas, Your salary is 4,253.88 (There are 3 decimal values in the given value and it is rounded up to 2 decimal places. Left of
  "." in formatter we have 5 decimal places ####,0 which is more than value so it formats one by one as a thousand separator is also there it includes the comma 4,253 and one more hash is left as we know if there is no value hash leaves place empty).
- `Console.WriteLine("Hello {0}, Your salary is {1}", empName, salary.ToString("0000,0.00"));`

- `Console.ReadLine();//` Hello Thomas, Your salary is 04,253.88 (One will be access like before if 0 is there then empty place is filled with 0)

Aligning Numbers :-

To align a string, the alignment component can be used with the placeholder Consider the statement, `Console.WriteLine("{0,7}", 1234);`

{0 // says about place holder : 7 // says about how many string can be placed it can be positive or negative } , 1234 here only 4 places the value takes so other 3 places are filled with spaces}

- Positive values indicate alignment to the right and the negative means alignment to the left
- `Console.WriteLine("{0,3}", 1234);//` As 3 places to be fillers there are 4 places in value hence no need to consider 3 as value size is more. o/p :- 1234
- `Console.WriteLine("{0,7}", 12345); //` as only 5 places are filled 2 places should be filled with spaces and it should be written in right as it is a positive integer. O/p:- 12345
- `Console.WriteLine("{0,7}", 123); //` as only 3 places are filled 4 places should be filled with spaces and it should be written in right as it is a positive integer. O/p:- 123
- `Console.Write("{0,-7}", 1234); //`as only 4 places are filled 3 places should be filled with spaces and it should be written on the left as it is a negative integer. O/p:- 1234 .
- If the string to be aligned has a length greater than or equal to the value of the alignment component, then the alignment component is ignored.
- If the length of the string to be aligned is less than the alignment component, then the unfilled positions are filled in with spaces.

Currency format for numbers :-

The format specifier 'C' or 'c' can be used to format numbers with currency symbol, This 'c' already uses thousand Seperator.

```
decimal salary;
salary = 4500.856m;
Console.WriteLine("{0:C2}", salary);
```

C2 : c = Currency Format, 2 = Precision

Output would be, $4,500.86

- **Currency format with alignment:-** We can align currency values using format string as shown below
- ```
  double salary1, salary2;
  salary1 = 4500.856;
  salary2 = 320.654;
  Console.WriteLine("{0,10}", string.Format("{0:c2}",
  salary1)); //
  ```
  {0,10} alignment upto 10 digits. string.Format("{0:c2}", salary1)
  ```

Currency Format c2 where decimal places upto 2 and format of currency O/p :- $4,500.86

```
Console.WriteLine("{0,10}", string.Format("{0:c2}", salary2)); // {0,10} alignment upto 10 digits. string.Format("{0:c2}", salary1) Currency Format c2 where decimal places upto 2 and format of currency O/p :- $320.65
```

- **How to change the currency symbol ?** The currency symbol is defined by CultureInfo class. We can call the GetCultureInfo method of the CultureInfo class to change the currency symbol.

```
using System; // For importing Console classes for Input and output.
using System.Globalization; // For importing CultureInfo class

namespace AppNumFormat
{

class Program
{
static void Main(string[] args)
{
decimal salary = 4253.876m;
CultureInfo us = CultureInfo.GetCultureInfo("en-US");
Console.WriteLine(string.Format(us, "{0:C)", salary)); // $4,253.88

CultureInfo uk = CultureInfo.GetCultureInfo("en-GB");
Console.WriteLine(string.Format(uk, "(0
:C)", salary)); // £4,253.88

CultureInfo ca = CultureInfo.GetCultureInfo("en-CA");
Console.WriteLine(string.Format(ca, "(0:C)", salary)); // $4,253.88
Console.ReadLine();}}
```

Custom Date Formats:-

Format Specifier

- d = Day - from 1 to 31
- dd = Day - from 01 to 31
- M = Month - from 1 to 12
- MM = Month from 01 to 12
- yy = Last two digits of the year - 20
- yyyy = Year in four digits - 2020
- hh = Hour - from 00 to 11
- HH = Hour - from 00 to 23

- m = Minutes - from 0 to 59
- mm = Minutes - from 00 to 59
- s = Seconds - from 0 to 59
- ss = Seconds - from 00 to 59
- Eg :- `DateTime dateofJoining = DateTime.Now;`
  `Console.WriteLine("{0:dd/MM/yyyy HH:mm:ss}", dateofJoining);`
  `Console.WriteLine("{0:d/M/yy}", dateofJoining);`
- Output would be,
  02/01/2020 15:58:03
  2/1/20

Displaying special characters on the Console (Escape Sequence) :

- An escape sequence is a combination of characters beginning with a back slash (\) and followed by letters or digits.
- They represent non-printable and special characters in character and literal strings.
- **Escape Sequence Character - Representation**
- Single quote mark used in string literal - \'
- Double quote mark used in string literal - \"
- Backslash used to denote file path - \\
- New line - \n
- Carriage return - \r
- Horizontal tab - \t
- Vertical tab - \v
- Unicode character with a hex value - \u
- Eg :- `Console.WriteLine("Your name is \"Thomas\"");` // Your name is "Thomas" //
  `Console.WriteLine("First Line\nSecond Line\tThird Line");` //
  First Line
  Second Line Third Line //
- `Console.WriteLine("First Line" + Environment. NewLine + "Second Line");` // First Line
  Second Line //
- `Console.WriteLine("The path to the file is C:\\users\\tomspm");` // The path to the file is C:\users\tomspm //
- A backslash in a string introduces an escape sequence character in C#.
- If you want to output a backslash, then you need to double it. It is useful in displaying file paths.
- \n introduces a new line. Environment. NewLine is a human readable alternative for \n.

Setting Properties of Console Window:-

- To change the title of the console window, Title property of the Console class can be used.
- The maximum length of the title string is 24500 characters Console.WriteLine(Console.Title); - Displays the default title of the console like C:\Program Files\dotnet\dotnet.exe
- We can assign new title by typing this :-
  `Console.Title = "Tom's Console"; - Changes the console title to 'Tom's Console'`

  `Consider the below code lines,`
- `Console.WriteLine("Default Console Title: {0}", Console.Title); // o/p : Default Console Title:` C:\Program Files\dotnet\dotnet.exe //
- `Console.Title = "Tom's Console";`
- `Console.WriteLine("New Console Title: {0}", Console.Title); // o/p : New Console Title: Tom's Console //`
- `Console.WriteLine("Please press Enter to continue."); // Please press Enter to continue. //`
- `Console.ReadLine();`
- Following properties can be changed in this way :-
- SetWindowSize - method is used to change the width and height of the console window. `Console.SetWindowSize(100, 50);`
- Window height - specifies the number of rows that can be visible on the console window at a particular moment
- Window width - specifies the number of characters that can be visible on the console window at a particular moment
- BufferHeight - defines the number of rows of text that can be accessed on the console window.
- BufferWidth - defines the number of characters that can be accessed on a row
- The CursorVisible property is used to control the cursor visibility on the console window :- `Console.CursorVisible = false;`
- The BackgroundColor property can be used to change the background color of the console window
- The ForegroundColor property can be used to change the foreground color of the console window
- `Console.BackgroundColor = ConsoleColor.Blue;`
- `Console.ForegroundColor = ConsoleColor.Yellow;`

Taking Inputs from Keyboard :-

Use Console.Write() method when we need the user to take input and say regarding what we are taking input. Then use console.readline() so it takes input from the keyboard.

- Console.ReadLine() : It takes a Line from Keyboard in form of String
- Read(): The Read() method reads the (Single character) next character from the keyboard and it returns the ascii value of the character.
- ReadKey(): The ReadKey() method reads the next key pressed by user. It will return the original key. This method is commonly used to pause the screen until user press a key

Input is Default string we can convert it using Converting classes.

Rules for Variable Names in C# :-

- Variable name can contain letters, digits, and the underscore character (_).
- The first character of a variable name can be a letter, or an underscore character. But underscore is not recommended at the beginning of a variable name.
- C# is case-sensitive and so upper case and lower-case variable names are treated as different variables.
- C# keywords cannot be used as variable names.

Naming Conventions :-
Pascal notation: Instead of using spaces in a multi-word variable name, the first letter of each word is capitalised. Eg:- TotalPrice
Camel notation: is like Pascal notation. But the first letter of the variable name is lowercase. Eg: totalPrice

C# Naming Conventions
Class Name = PascalCase
Constructor Name = PascalCase
Method Name = PascalCase
Method Arguments = camelCase
Local Variables = camelCase
Constants Name = PascalCase

Declaring Variable :- DataType variable1, variable2,.....
Eg : `int a1, a2, a3;`
`decimal d1;`

Initializing Variable :- a1 = 11; // = is assignment operator
d1 = 12.009787m; // as it is decimal data type

Types of Variables in C#

1. Instance / Non-Static Variables

- Declared **without** `static`.
- Belongs to each **object instance**. Until and unless we create instance of class Memory will not be created so we can't access it.
- Different copies for each object.
- We can initialise non static variables with a constructor. Each time we change an object a new value will be stored like Obj1.y = 200, Obj2.y = 300.

Example:

```
class Student {
    int marks = 90;  // Instance variable
}
```

2. Static Variables

- Declared with `static`.
- Belongs to the **class, not objects**.
- Only **one copy** exists for all objects. That is When initialise the variable of static it won't create new memory each time but it will override the existing value
- If any variables which is defined inside static method will also be static variables

Example:

```
class Student {
    static int count = 0;  // Static variable
}
```

3. Constant Variables

- Declared with `const`.
- Value must be assigned **at declaration** and **cannot change** later.
- Behaves like static (one copy for whole class).

Example:

```
const float PI = 3.14f;
```

Constants :-

- A constant is like a variable as it provides a named location in memory to store a data value. But the value that is assigned to a constant cannot be changed later in the program.
- The keyword **'const'** is used to define a constant
- As with variables, constants have a type, a name and a value
- Unlike variables, **constants must be initialized while they are declared**

- **const datatype constant_name=value;**
- Example: `const double PI = 3.14159;`

4. Readonly Variables

- Declared with `readonly`.
- Value assigned **at declaration or inside constructor**.
- Cannot change later.
- Different value possible for each object.
- Behaves like Non static only difference is you cannot modify the value of Readonly variable after it's initialised.

Example:

```
readonly int id;
public Student(int i) { id = i; }
```

5. Local Variables

- Declared **inside a method**.
- Scope limited to the method/block.
- Must be initialized before use.

Example:

```
void Display() {
    int x = 100;  // local variable
    Console.WriteLine(x);
}
```

Key Differences

- **Instance vs Static**: Instance → per object, Static → one copy for all.
- **Static vs Constant**: Static → can change, Constant → cannot change.
- **Readonly vs Constant**: Readonly → can assign in constructor, Constant → fixed at declaration.
- **Local**: Exists only inside method, destroyed when method ends.

🔹 What are Data Types in C#?

- **Define what kind of data a variable can store (int, bool, string, etc.).**
- **Provide size, range, operations allowed, and type of result in expressions.**
- **Important for memory optimization and performance.**

🔹 Categories of Data Types

1. **Value Types**
    - **Store data directly in memory.**
    - **Examples: int, float, bool, char, struct, enum.**
2. **Reference Types**
    - **Store a reference (address) to the actual data.**
    - **Examples: string, class, array, object, interface.**
3. **Pointer Types**
    - **Hold memory addresses (used in unsafe code).**

Value Types are simple types(pre-defined data types) which include integral, floating point, boolean and char type

Value Data Types (Predefined / User defined) :-

- byte = 8 bit unsigned integer (0 to 255)
- sbyte = 8 bit signed integer (-128 to 127)
- short/Int16 = 16 bit signed integer
- ushort = 16-bit unsigned integer
- int/Int32 = 32-bit signed integer
- uint = 32-bit unsigned integer
- long/Int64 = 64-bit signed integer
- ulong = 64-bit unsigned integer
- float = 32-bit Single-precision floating point type (Suffix = F/f)
- double = 64-bit double-precision floating point type
- decimal = 128-bit decimal type. (Commonly used for storing financial and monetary values) (Suffix = M/m)
- char = 16-bit single Unicode character (Any valid character covered with ‘ ’)
- string = A sequence of Unicode characters (Covered with “ ”)
- DateTime = Date and time values (0:00:00am 1/1/01 to 11:59:59pm 12/31/9999)
- bool = 8-bit logical true/false value
- Struct → user-defined value type.
- Enum → named constants.(user defined data type)

Eg :- `Console.WriteLine("Total :-" + a1 );` // Displays **Total :- 11**

Creating DateTime variable and Initializing it :-

```
DateTime dOB;
dOB = Convert.ToDateTime("11/22/1995");
```

// as it is string we are converting it into DateTime.

Reference Data Types

- **string** → collection of characters.
- **class** → blueprint for objects.
- **object** → base type for all types in .NET.

- **array** → collection of same type.
- **interface** → defines contract.

Pointer Data Types

- Hold memory addresses (unsafe code).
- Example:

```
unsafe {

    int x = 10;

    int* p = &x;

    Console.WriteLine((int)p);

}
```

We can get default value of datatypes in this way

```
Console.WriteLine($"Default Value of Double: {default(double)}"); //0

Console.WriteLine($"Default Value of Character: {default(char)}"); //

Console.WriteLine($"Default Value of Boolean: {default(bool)}");
//False
```

Literals in C#

- **Definition**: Fixed values assigned directly to variables.
- **Property**: Cannot be changed during program execution.

Example: int x = 100; // here 100 is a literal

Types of Literals in C#

1. Integer Literals

- Used with int, uint, long, ulong.
- **Forms**:
    - **Decimal (Base 10)** → int a = 101;
    - **Hexadecimal (Base 16)** → int b = 0x1A3F;
    - **Binary (Base 2)** → int c = 0b1011;

- **Suffixes**:
  - U/u → Unsigned
  - L/l → Long
  - UL/ul → Unsigned Long
- ❌ No Octal support in C#.

---

2. Floating-Point Literals

- Used with float, double, decimal.
- By default → **double**.
- **Suffixes**:
  - F/f → float
  - M/m → decimal
  - D/d → double (optional)

Example:

```
double a = 10.15;   // default double

float b = 12.5F;    // float

decimal c = 99.99M; // decimal
```

---

3. Character Literals

- Enclosed in single quotes ' '.
- **Ways to specify**:
  1. **Single character** → char ch = 'A';
  2. **Unicode** → char ch = '\u0041'; // 'A'
  3. **Escape sequences** → '\n', '\t', '\\'

Example:

```
char ch1 = 'A';

char ch2 = '\u0041'; // Unicode A

Console.WriteLine("Hello\nWorld");
```

---

4. String Literals

- Enclosed in double quotes " ".
- **Types**:
  - **Regular String** → "Hello\nWorld"
  - **Verbatim String** → @"Hello\nWorld" (ignores escape sequences).

Example:

```
string str1 = "Dot Net Tutorials";

string str2 = @"C:\Users\Name\Docs"; // Verbatim
```

---

5. Boolean Literals

- Only two values: true and false.
- ❌ Not equal to 1 or 0.

Example:

```
bool isActive = true;

bool isDone = false;
```

---

6. Binary Literals

- Introduced with **C# 7.0**.
- Prefix: 0b (only 0 and 1 allowed).

Example:

```
int num1 = 0b1001;   // 9

int num2 = 0b01000011; // 67
```

Type Casting in C#

- **Definition**: Type Casting (or Type Conversion) is the process of converting one data type value into another.
- Only possible when both data types are **compatible** (e.g., numeric types).
- Types of Casting in C#:
    1. **Implicit Casting (safe conversion, automatic)**
    2. **Explicit Casting (manual conversion, may cause data loss)**

---

1. Implicit Casting (Type Conversion)

- Done automatically by the compiler.
- Converts **smaller data type** → **larger data type** (no data loss).

Example:

```
int num = 1500;
double result = num;   // Implicit casting int → double
```

```
Console.WriteLine(result);  // 1500
```

- Works only if **types are compatible**.
- Example: `int → long`, `float → double`.
- Not allowed: `int → bool`, `char → bool`,

---

2. Explicit Casting

- Conversion is **forced by the programmer** using **cast operator** `(type)`.
- Used when converting **larger type → smaller type**.
- Risk of **data loss**.

Example:

```
double d = 1.23;
int i = (int)d;   // Explicit casting
Console.WriteLine(i); // 1 (fraction lost)
```

Example with data loss:

```
int num = 500;
byte b = (byte)num;   // byte range: 0–255
Console.WriteLine(b); // 244 (data loss)
```

---

3. Conversion Between Non-Compatible Types

- Example: `string → int` is not possible with cast operator.
- Use **Helper Methods**:

(a) Convert Class

```
string str = "100";
int i1 = Convert.ToInt32(str);   // String → Int
double d = 123.45;
int i2 = Convert.ToInt32(d);     // Double → Int
```

May throw **runtime error** if incompatible:

```
string s = "Hello";
int i = Convert.ToInt32(s);  // Runtime error
```

Converting Classes Helps to convert one data type to Other datatype:-

1. ToInt16() - Converts a value to the corresponding 16-bit signed integer
2. ToInt32() - Converts a value to the corresponding 32-bit signed integer
3. ToInt64() - Converts a value to the corresponding 64-bit signed integer

4. ToBoolean() - Converts a value to the corresponding Boolean value
5. ToByte() - Converts a value to the corresponding 8-bit unsigned integer
6. ToChar() - Converts a value to the corresponding Unicode character
7. ToDateTime() - Converts a value to the corresponding DateTime value
8. ToDecimal() - Converts a value to the corresponding decimal number
9. ToSingle() - Converts a value to the corresponding single-precision floating-point number
10. ToDouble() - Converts a value to the corresponding double-precision floating-point number
11. ToSByte() - Converts a value to the corresponding 8-bit signed integer
12. ToUInt16() - Converts a value to the corresponding 16-bit unsigned integer
13. ToUInt32() - Converts a value to the corresponding 32-bit unsigned integer
14. ToUInt64() - Converts a value to the corresponding 64-bit unsigned integer
15. Eg : `feesPaid = Convert.ToDecimal(Console.ReadLine());` // Input which we get is string by default so we need to convert it into decimal in form of use it. //

(b) Parse() Method

- Available in built-in types like `int.Parse()`, `bool.Parse()`.

Example:

```
string str1 = "100";
int i = int.Parse(str1);   // String → Int
string str2 = "TRUE";
bool b = bool.Parse(str2); // String → Bool
```

- Throws **runtime error** if not convertible.

(c) TryParse() Method

- **Safe conversion** (does not throw exception).
- Returns **true if success**, false otherwise.

Example:

```
string str1 = "100";
bool ok = int.TryParse(str1, out int num1);
Console.WriteLine(ok ? $"Converted: {num1}" : "Conversion
failed");

string str2 = "Hello";
bool ok2 = int.TryParse(str2, out int num2);
```

```
Console.WriteLine(ok2 ? $"Converted: {num2}" : "Conversion failed");
```

Summary

- **Implicit Casting**: Safe, automatic, smaller → larger.
- **Explicit Casting**: Manual `(type)`, larger → smaller, may lose data.
- **Helper Methods**:
  - `Convert.ToXxx()` → Flexible, may throw runtime error.
  - `Parse()` → Converts string to target type, runtime error on failure.
  - `TryParse()` → Safe, returns `true/false` without exceptions.

Operators :

- Operators are symbols that perform operations on operands (values/variables).
- Arithmetic operators : +, -, *, /, and %
- Relational Operators : ==, >=, <=, >, < and != (Gives only true/false)
- Logical Operators : ||(OR), &&(AND), ! (Logical NOT) (work with boolean values)
- Unary Operators : +(unary plus), -(unary minus), ++(Post increment and pre increment), - -(Post decrement and pre decrement)
- Ternary Operator(?:): Conditional expression ? Exp 1 : Exp 2 If Conditional expression is true then Exp1 is evaluated otherwise exp2 is evaluated.
- Compound / Assignment operators: +=, -=. *=, /=, %=, and =
- Bitwise Operators – work at bit level => & (AND), | (OR), ^ (XOR), ~ (NOT), << (Left shift), >> (Right shift)
- Null-coalescing Assignment (??=) : variable ??= expression;
  If the variable is currently null, it will be assigned with the value of the expression on the right side of the operator. If the variable is not null, its value remains unchanged, and the expression on the right side of the operator is not evaluated.
  Nullable value can be also be declared as int? i = null;
  i?? = 10; // i is null so 10 gets assigned
  Console.Writeline(i); // 10;
  i??= 20; // as i = 10 so it won't assign 20 in it as it already consist a value //
  Console.Writeline(i); // 10;

Control Flow Statements in C#

Definition

- Control flow statements alter the **normal top-to-bottom execution** of a program.

- They allow **decision making, looping, and branching**.
- Useful for writing **flexible and complex programs**.

Types of Control Flow Statements

1. Selection (Branching) Statements

Used to **choose between multiple paths**.

- **if** – executes a block if condition is true.
- **if-else** – chooses one block when condition true/false.
- **else if ladder** – multiple conditions.
- **nested if** – if inside another if.
- **switch** – selects one case among many.

---

2. Iteration (Looping) Statements

Used to **repeat execution** of code.

- **for** – known number of iterations.
- **while** – repeats while condition true.
- **do-while** – executes at least once.
- **foreach** – iterates through collections.

---

3. Jump Statements

Used to **transfer control** from one point to another.

- **break** – exits loop/switch.
- **continue** – skips current iteration, goes to next.
- **goto** – jumps to a labeled statement.
- **return** – exits from method and returns value.

IF ELSE STATEMENTS :

- If(Condition 1){ // write statements set1
  }
  else if (condn 2) { // write statements set2
  }
  …
  …
  else { // write statements set n

}

- Nested If statement :

```
If(Condn 1) {
        if(Condn 2){
        // Write statement
        }}
```

Switch Statement :

```
switch(Expression){
        case value1:
                Statement 1;
                break;
        case value2:
                Statement 2;
                break;
        …
        …
        default:
                Statement n;
                break;
}
```

LOOP STATEMENTS :

While Loop :

```
Int i =0; //intialization
while(i<=10 //Condn statement){
        Statements that need to run inside loop
        i++; //Increment or decrement
}
```

- while loops should terminate at some point, otherwise it will result in infinite looping
- **C**ommon mistake is to forget a statement that causes the conditional expression to become false. In the previous example, the statement "i++;" accomplishes it
- The variable i is referred to as a **sentinel**
- While loop is an entry controlled loop because the condition is evaluated in the beginning

```
while (i <= n)
            {
                Console.Write($"{i} ");
                i = i + 2;
            }
```

Do while loop :

do{
// statements that needed to be executed
}while(condn statement); // semicolon is must in this do while

- **It will executed at once even the condn doesnot satisfy.**

For Loops :

for(control variable initialization; conditional expression; control variable update) {
statement set;
}
We can intialize multiple variable names at a time : (i,j,k common control variable names)
for(int i = 1 , j = 10; i <= j; i++, j++)
{ //statements
}

Break and Continue statements :

- The 'break' statement is used when a loop statement jumps out of the loop. It can also be used along with IF conditional statements in a loop.
- The 'continue' statement jumps to the beginning of a loop or continuation of a loop without executing the remaining statements in the loop.It can also be used along with IF conditional statements in a loop.
- Goto Statement = Used to **transfer control** unconditionally to a **labeled statement** within the same function. A **label** is a valid identifier followed by a colon (`label1:`).
  Syntax:
  ```
  goto labelName;
  labelName:
      // code
  ```
  Considered an **unstructured control flow** → makes code less readable, harder to debug & maintain.

What is a Function?

- A **function** (or method) is a group of related statements that perform a specific task. Functions can take **input (parameters)** and return an **output (return value)**. They promote **code reusability, modularity, and maintainability**.

🔹 **Why Functions?**

- Avoid **monolithic programming** (all code in `Main`).
- Break large programs into **smaller, manageable pieces** (modular programming).
- Enable **teamwork** (different developers can work on different functions).
- Improve **debugging, readability, and reusability**.

🔹 **Types of Functions in C#**

1. **Built-in Functions**
   - Predefined in .NET libraries.
   - Example: `Console.WriteLine()`, `Math.Sqrt(25)` → Output: `5`.
2. **User-Defined Functions**
   - Created by developers as per project needs.
   - Provide flexibility and reusability.

🔹 **Advantages**

- **Readable & maintainable code**
- **Code reusability** (write once, use multiple times)
- **Smaller program size** (avoid duplicate code)
- **Easier debugging**

🔹 **Structure of a Function**

```
[access_specifier] [modifier] return_type
FunctionName(parameter_list)
{
    // function body
    return value;   // if return_type is not void
}
```

- **Function Name** – identifier for the method
- **Return Type** – `int`, `string`, `void`, etc.
- **Parameters** – optional (can take 0 or more)
- **Access Specifier** – `public`, `private`, etc. If we do not specify any modifier, then it is private by default. If the access level is private, then the method will not be accessible outside the class in which it is defined. If the access level is public, then the method will be accessible to other classes.
- **Modifier** – `static`, `virtual`, etc.
- **Body** – code statements

🔹 **Example: User-Defined Function**

```
using System;
class Program
{
    static void Main(string[] args)
    {
        int sum = Add(10, 15);   // Function Call
        Console.WriteLine($"Sum is {sum}");
    }
    static int Add(int a, int b)   // Function Definition
    {
        return a + b;
    }
}
```

Output: `Sum is 25`

🔹 **Function Signature**

- Combination of **function name + parameter list**.
- Example: `Add(int a, int b)`

🔹 **Parameters**

- **Actual Parameters** → Values passed during function call (e.g., `10, 15`).
- **Formal Parameters** → Variables inside function that receive the values (`a, b`).

🔹 **Return Statement**

- Ends function execution and sends value back to caller.
- If no value is returned → use `void`.

User-Defined Functions in C#

👉 In C#, a function is a block of code that performs a specific task.
👉 Functions allow **code reusability, modularity, and easier maintenance**.
👉 User-defined functions are those written by the programmer.

Types of User-Defined Functions in C#

1. **No Arguments and No Return Type**
    - Function does not take parameters and does not return a value.
    - Data transfer: **None**.
    - Return type: `void`.

```
static void Sum()

{

    int x = 10, y = 20;

    Console.WriteLine($"Sum = {x + y}");

}
```

2. ✅ Output: `Sum = 30`

---

2. **No Arguments but Return a Value**
    - Function takes no input but returns a value.
    - Data transfer: From **called function → calling function**.

```
static int Sum()

{

    int x = 10, y = 20;

    return x + y;

}
```

3. ✅ Output: `Sum = 30`

---

3. **Arguments Passed but No Return Value**
    - Function takes input but does not return a value.
    - Data transfer: From **calling function → called function**.

```
static void Sum(int x, int y)

{

    Console.WriteLine($"Sum = {x + y}");

}
```

4. ✅ Output: `Sum = 30`

---

4. **Arguments Passed and Return a Value**
    - Function takes input and also returns a value.
    - Data transfer: **Two-way** (input + output).

```
static int Sum(int x, int y)

{

    return x + y;

}
```

5. ✅ Output: `Sum = 30`

Function Overloading in C#

- Writing multiple functions with the **same name** but **different parameter lists**.
- Compiler decides which function to call based on **arguments passed**.
- Overloading is allowed if:
    1. Number of parameters differs
    2. Type of parameters differs
- Return type **alone** cannot differentiate functions.

Example:

```
static int add(int x, int y) => x + y;

static int add(int x, int y, int z) => x + y + z;

static float add(float x, float y) => x + y;
```

✅ Valid overloads:

- `add(int, int)`

- `add(int, int, int)`
- `add(float, float)`

❌ Invalid overload:

- `int add(int, int)` and `float add(int, int)` → Same parameter list.

Advantages of Function Overloading

- Code readability → same name for similar tasks.
- No need to remember multiple function names.
- More intuitive and flexible coding.

Quick Example with All Overloads

```
Console.WriteLine(add(10, 20));          // int add(int, int)

Console.WriteLine(add(10, 20, 30));      // int add(int, int, int)

Console.WriteLine(add(10.5f, 25.6f));    // float add(float, float)
```

✅ Output:

```
30

60

36.1
```

Call by Value and Call by Reference in C#

🔹 Call by Value (Default in C#)

- A **copy** of the actual value/reference is passed to the method.
- Changes inside the method **do not affect** the original variable.
- Works for both **value types** and **reference types**.
- **No `ref`/`out` keyword** required.

Example with Value Type:

```
static void UpdateValue(int b) { b = 30; }
int a = 15;
UpdateValue(a);
Console.WriteLine(a); // 15 (unchanged)
```

Example with Reference Type:

```
public class Employee { public string Name; }
Employee emp1 = new Employee { Name = "James" };
UpdateName(emp1);
Console.WriteLine(emp1.Name); // Smith (object updated)
void UpdateName(Employee e) { e.Name = "Smith"; }
```

👉 Reference is copied → both variables point to the same object.
👉 But if you reassign (`e = null`), original reference remains intact.

🔹 Call by Reference

- The **actual memory address** is passed to the method.
- Changes inside the method **directly affect** the original variable.
- Achieved using `ref` or `out` keyword.

Example with Value Type:

```
static void UpdateValue(ref int b) { b = 30; }
int a = 15;
UpdateValue(ref a);
Console.WriteLine(a); // 30 (changed)
```

Example with Reference Type:

```
public class Employee { public string Name; }
Employee emp1 = new Employee { Name = "James" };
UpdateName(ref emp1);
Console.WriteLine(emp1.Name); // ❌ NullReferenceException
void UpdateName(ref Employee e) { e = null; }
```

👉 Because `emp1` and `e` point to the **same memory location**, setting `e = null` makes `emp1` null too.

🔹 Key Differences

Feature	Call by Value	Call by Reference
What is passed?	Copy of variable	Actual memory address
Effect on original variable	No change	Changes reflected
Keywords needed	None	`ref` or `out`

Memory address	Different for formal & actual parameters	Same for both
Use case	When original data must be protected	When original data must be modified

---

✅ **Summary**:

- In C#, **all values are passed by value by default**, even reference types.
- To truly modify the caller's variable, you must use `ref` **or** `out` → **Call by Reference**.

Recursion in C# – Brief Notes

🔹 What is Recursion?

- **Recursion**: A process in which a function calls itself.
- Similar to a loop: repeats until a **base condition** is satisfied.
- Without a base condition → leads to **infinite recursion / stack overflow**.

🔹 Recursive Function

- A function that **calls itself**.
- General form:

```
void Fun(int n)
{
    if (n > 0)   // Base condition
    {
        // Statements
        Fun(n - 1); // Recursive call
    }
}
```

🔹 How Recursion Works

- **Calling phase** → function keeps calling itself with smaller input.
- **Returning phase** → function returns results back after reaching base condition.
- Each call creates a **new activation record** in the stack (new set of local variables).

🔹 Examples

Example 1: Print Numbers (Descending)

```
static void fun1(int n)
{
    if (n > 0)
    {
        Console.Write($"{n} ");
        fun1(n - 1);
    }
}
// Output: 3 2 1
```

Example 2: Print Numbers (Ascending)

```
static void fun2(int n)
{
    if (n > 0)
    {
        fun2(n - 1);
        Console.Write($"{n} ");
    }
}
// Output: 1 2 3
```

Example 3: Factorial

```
static int factorial(int number)
{
    if (number == 1)
        return 1;   // Base condition
    else
        return number * factorial(number - 1);
}
// factorial(5) → 120
```

Example 4: Sum of 1 to n

```
static int fun(int n)
{
    if (n > 0)
        return fun(n - 1) + n;
    return 0;
}
// fun(5) → 15
```

- **Tracing a Recursive Function**
    - Example: `fun1(3)` → `fun1(2)` → `fun1(1)` → `fun1(0)` (stop).
    - Prints **on calling phase** (3 2 1) or **on returning phase** (1 2 3).

🔹 **Advantages**

- Simplifies code for problems like factorial, Fibonacci, tree/graph traversal.
- Easier to read for problems naturally recursive.
- Useful for prefix, postfix, infix evaluations.

🔹 **Disadvantages**

- **Slower** due to repeated function calls.
- **Stack overflow** risk if recursion is too deep.
- May lead to **infinite recursion** if no proper base condition.

🔹 **Time Complexity**

- Each function call takes **1 unit time**.
- For n calls → **O(n)** time complexity.
- Example: factorial recursion runs n times → O(n).

✅ **Summary**:

- Recursion = function calling itself until base condition.
- Two phases: **calling phase** & **returning phase**.
- Must always include a **base condition**.
- Often less efficient than loops, but **clearer for hierarchical problems**.

STRINGS : -

- In C#, string is referred as an array of characters.
- A string is represented by the .NET class **System.String**

string firstName;
firstName = "Thomas";

- A string variable can be assigned with a collection of characters surrounded by double quotes
- Unlike `int`, `double` (value types), **string is a class → reference type**.
- Stored in the **heap** instead of stack.
- The string firstName is an array of 6 characters with index from 0 to 5 Ex :- firstName[0] = 'T'
- The Length property of the String class gets the number of characters in the string. Example: firstname.Length.

- for loop to retrieve each character of a string :-
  string firstName="Thomas";
  for (int i = 0 ; i < firstName.Length; ++i)
  Console.Write(firstName[i]);
- foreach loop to retrieve each character of a string :-
  string firstName="Thomas";
  foreach (char ch in firstName)
  Console.Write(ch);

- **`string` vs `String`**
  - `string` (small s) → **alias** for `System.String`.
  - `String` (capital S) → actual class name.
  - Convention:
    - Use **`string`** for variable declarations.
    - Use **`String`** for calling static methods (`String.Compare()`, `String.Concat()`, etc.).

Methods of Strings :- syntax: stringName.Method(arguments);

- The IndexOf method of the string class returns an integer that represents the position of the first occurrence of the specified character/string in the source string. If the character/string isn't found in the source string -1 is returned.
  Example:
  string firstName="John";
  int position;
  position = firstName.IndexOf('o');
  Console.WriteLine("The position is: {0}", position); // Returns 1 //
  position = firstName.IndexOf("hn");
  Console.WriteLine("The position is: {0}", position); // Returns 2 //
  position = firstName.IndexOf("H");
  Console.WriteLine("The position is: {0}", position); // Returns -1 as it is case sensitive //
  position = firstName.IndexOf("p");
  Console.WriteLine("The position is: {0}", position); // Returns -1 as it is not present in john
- The LastIndexOf method returns the index of the last occurrence of a character/string in a source string
  Example :-
  int position;
  string title="Programming in C#";
  position = title. LastIndexOf(');
  Console.WriteLine("The position is: {0}", position); // Returns 14
  Returns the index of the last occurrence of the character ' '(blank space) in the string title, which is 14 //

- The StartsWith method checks if a string starts with the specified character/string. If it starts with the specified string, then this method returns true, otherwise returns false
  Example :
  bool result;
  string title="Programming in C#";
  result = title.StartsWith("Programming");
  Console.WriteLine("Result: {0}", result);
  // Returns true //
  result = title.StartsWith("Web");
  Console.WriteLine("Result: {0}", result);
  // Returns false //
- The EndsWith method checks if a string ends with the specified character/string. If it ends with the specified string, then this method returns true, otherwise returns false
  Example :
  bool result;
  string title="Programming in C#";
  result = title. EndsWith("C#");
  Console.WriteLine("Result: {0}", result);
  // Returns true //
  bool result;
  string title="Programming in C#";
  result = title. EndsWith("Java");
  Console.WriteLine("Result: {0}", result);
  // Returns false //
- The Remove method removes all the characters from the specified position of a string.
  Syntax: string.Remove(start_index, number_of_char)
  - ☐ start_index - specifies the index of the position from where the characters need to be removed.
  - ☐ number_of_char - specifies the number of characters to be removed. If the number_of_char is not specified, then all the characters from the start_index will be removed

  Example :
  string title = "Programming in C#";
  string newTitle;
  newTitle = title. Remove (11, 3); Console.WriteLine("Revised Title: {0}", newTitle);
  // Revised Title: Programming C# //
  newTitle = title. Remove (11);
  #Removes all characters from the index 11 of the string title
  Console.WriteLine("Revised Title: {0}", newTitle);

// Revised Title: Programming //

- The Insert method inserts a new string at the specified position of a string.
  Syntax: string.Insert(position, new_string)
  - ☐ position- specifies the index of the position where the new string to be inserted
  - ☐ new_string - specifies the new string to be inserted

  Example :
  string title = "Programming in C#";
  string newTitle;
  newTitle
  = title. Insert(0, "Basic "); #Inserts the string 'Basic 'at the 0th index position
  Console.WriteLine("Revised Title: {0}", newTitle);
  // Revised Title: Basic Programming in C#//
- The Replace method replaces a string with a specified string
  Syntax: string. Replace(old_string, new_string)
  - ☐ old_string - specifies the string that needs to be replaced
  - ☐ new_string-specifies the replacement string

  Example :
  string title = "Programming in C#";
  string newTitle;
  newTitle = title. Replace("C#","Java");
  # Replaces the string 'C#' with the string 'Java'
  Console.WriteLine("Revised Title: {0}", newTitle);
  // Revised Title: Programming in Java //
- The ToUpper method converts every character of a string to upper-case. If a character does not have an upper-case equivalent character, it remains unchanged. For example, special symbols like #, $ remain unchanged
  Example :
  string title = "Programming in C#";
  string newTitle;
  newTitle = title. ToUpper();
  Console.WriteLine("Revised Title: {0}", newTitle);
  // Revised Title: PROGRAMMING IN C# //
- The ToLower method converts every character of a string to lower-case. If a character does not have a lower-case equivalent character, it remains unchanged. For example, special symbols like #, $ remain unchanged
  Example :
  string title = "Programming in C#";
  string newTitle;
  newTitle = title. ToLower();
  Console.WriteLine("Revised Title: {0}", newTitle);

// Revised Title: programming in c# //

- The Substring method retrieves a substring(portion) from a string.
  Syntax: string.Substring(start_index, number_of_char)
  - ☐ start_index - specifies the index of the position from where the substring needs to be extracted
  - ☐ number_of_char - specifies the number of characters to be extracted. If the number_of_char is not specified, then all the characters from the start_index will be extracted

  Example :
  string title = "Programming in C#";
  string resultString;
  resultString = title. Substring(12, 2); Console.WriteLine("Result String: {0}", resultString);
  // Result String: “in” 2 characters will be extracted starting from the index position 12 //
  resultString = title. Substring(15)
  // All characters will be extracted starting from the index position 15 //
  Console.WriteLine("Result String: {0}", resultString);
  // Result String: C# //
- The Trim method removes all trailing and leading whitespaces from a string
  string firstName = “ Mark ";
  string lastName = “ Peter ";
  string fullName;
  fullName = firstName.Trim() + lastName.Trim(); Console.WriteLine("Full Name: {0}", fullName);
  // Full Name: MarkPeter //
- The Split method splits a string into an array of strings separated by a delimeter
  Syntax: string.Split(delimeter)
  delimeter - is the character on occurrences of which the string needs to be split
  Example :
  string address = “Unit 31,186 Parker Avenue, London, Ontario, Canada";
  address = address.Trim();
  string[] adressColumns = address.Split(',');
  string apartment adressColumns[0];
  string street adressColumns[1];
  string city adressColumns[2];
  string province= adressColumns[3];
  string country = adressColumns[4];
  Console.WriteLine("Address");
  Console.WriteLine("Apartment: {0}", apartment);

Console.WriteLine("Street : {0}", street); Console.WriteLine("City : {0}", city);
Console.WriteLine("Province: {0}", province);
Console.WriteLine("Country: {0}", country);
// Address
Apartment : Unit 31
Street : 186 Parker Avenue
City : London
Province: Ontario
Country : Canada //

- The Equals method is used to check whether two string objects are having the same value or not. If both string object values are equal, then the Equals() method will return true otherwise false.
  Syntax: string1.Equals(string2)
  Example :
  string firstName, lastName;
  firstName = "Mark";
  lastName = "Mark";
  if(firstName.Equals(lastName)==true)
  Console.WriteLine("First Name {0} is same as Last Name {1}", firstName, lastName);
  else
  Console.WriteLine("First Name {0} is not same as Last Name {1}", firstName, lastName);
  firstName = "Mark";
  lastName = "mark";
  if (firstName.Equals(lastName))
  Console.WriteLine("First Name {0} is same as Last Name {1}", firstName, lastName);
  else
  Console.WriteLine("First Name {0} is not same as Last Name {1}", firstName, lastName);
  // First Name Mark is same as Last Name Mark
  First Name Mark is not same as Last Name mark //
- The OrdinalIgnoreCase Property of the Equals method is used to check whether two string objects are having the same value or not ignoring the case. If both string object values are equal, then the Equals() method will return true otherwise false.
  Syntax: string1.Equals(string2, StringComparison.OrdinalIgnoreCase)
  Example :
  string firstName, lastName;
  firstName = "Mark";
  lastName = "mark";
  if (firstName. Equals (lastName, StringComparison.OrdinalIgnoreCase) == true)

```
Console.WriteLine("First Name {0} is same as Last Name {1}", firstName,
lastName);
else
Console.WriteLine("First Name {0} is not same as Last Name {1}",
firstName, lastName);
// First Name Mark is same as Last Name mark //
```

Immutability of string : In c#, string is immutable. That is string object cannot be modified once it is created. If we make any changes to the string like adding or modifying an existing value, then it discards the old instance in memory and creates a new instance to hold the new value. If we do modifications many times on same string, then every time new instance is created on the heap memory this will effect program's performance.

StringBuilder Class : is available in the System.Text namespace.

It is mutable. If we modify the value of a StringBuilder object, it will not create a new instance in the heap memory. Instead it will modify the value in the original instance itself.

The namespace System.Text should be imported to the application in order to use the StringBuilder using System.Text;

To declare a StringBuilder object, the new keyword is used.

StringBuilder sb = new StringBuilder();

- The Append method of a StringBuilder is used to append a string object at the end of a string represented by the StringBuilder.

```
StringBuilder sb = new StringBuilder();
sb.Append("Thomas");
sb.Append("Mathew");
// The content of the StringBuilder object will be now “Thomas Mathew” //
```

INTERPOLATED STRINGS :-

- An interpolated string starts with a $ sign.
- The string that follows can contain on or more interpolated expressions

Example :

```
string empName = "Mark";
decimal salary = 2400;
// Composite formatting:
Console.WriteLine("Hello, {0}! Your salary is {1}", empName, salary);
// String interpolation:
Console.WriteLine($"Hello, {empName}! Your salary is {salary}");
```

ARRAYS :

Single dimensional array :-

Total 100 elements index starts from 0 to 99. Arr[100] will be invalid reference.

```
Declaring array : syntax :- datatype[] arrayname
Eg : int[] arr;
```
To initialise array we need new keyword in order to store the elements as array is reference type.
```
arrayName = new datatype[arraysize];
Example:
arr = new int[100]
```

We can declare and initialize an array in a single statement too
```
Syntax:
datatype[] arrayName= new datatype[arraySize];
int[] arr = new int[100];
```

We can assign value to specific index in this way:
```
arr[0] = 4500;
arr[100] = 1500; //
```
we can't assign to index 100 as maximum index is 99 not 100 it gives error message.

We can assign values to the array at the time of declaration
Example 1:
```
double[] taxPayable = new double[5] {200.50, 100.75,
250.50, 140.75, 190.80 };
```
When we assign values to array elements at the time of declaration, we need to assign values to all elements if array size is specified. If we keep array size as 5 then we should declare 5 elements.

We can assign values to the array at the time of declaration omitting the size, as below
Example 1:
```
double[] taxPayable = new double[] { 200.50, 100.75,
250.50, 140.75};
```
Declares and initialises an array 'taxPayable' of the 'double' type with 4 elements.Since size is not specified, the size will be assumed as the number of initialization values.

We can copy an array into another array.The source and the target array will point to the same memory location.

Example 1:

```
int[] employeeNumber = new int[] {
123,145,189,231,110};
int[] registerNumber = employeeNumber;
```

The target array 'registerNumber' is created and initialised with values as available in the source array 'employee Number'.

Access array element : `arrayname[index value to be accessed];`

For accessing every element in array use for loop:

```
int[] employeeNumber = new int[] { 123, 145, 189,
231, 110 };
for (int i = 0; i <= 4; ++i)
{
Console.WriteLine("{0}", employeeNumber[i]);
}
```

For finding length of array we use :- `employeeNumber.Length;`

FOR EACH LOOP : The foreach loop can be used to iterate through the items c arrays/collections.It iterates through each item, hence called foreach loop. The iterable-item can be an array or a collection

Syntax: `foreach (dataType element in iterable-item) {`

```
body of foreach loop
}
```

Example:

```
double[] salary = new double[10] { 1200.50, 1400,
1800, 2300, 1900, 1500, 1600, 2000, 2100, 1750 };

foreach(double empSalary in salary)
{ Console.WriteLine("{0}", empSalary);
}
```

int[] arr = new int[10] ; // as we are not initialising with values by default all indexes from 0 to 9 will have 0 value.

Multi dimensional array :-

To declare a two-dimensional array in C#, we can use the following syntax: `datatype[,] arrayname;`

Example : `int[,] matrix;`

For initialising array we can use :

```
arrayName = new datatype[row size, col size];
Example: matrix= new int[3,2]; // 3 rows 2 cols
```

Accessing element in array of 0th row and 1th col : `matrix[0,1];`

We can assign values to the array at the time of declaration

Example 1:

```
int[,] matrix = new int[3, 2]{{2,3},{4,5},{4,5}};
```

Using Nested For loop to access individual elements in array:

```
for(int i=0;i<=2;++i){ // i for row index value
for(int j=0;j<=1;++j) {  // j for col index value
Console.Write("{0}", matrix[i,j]); } }
```

Using for each loop :

```
foreach (int element in matrix2) {
Console.WriteLine("{0}", element);
} // goes through each element in the matrix
```

The foreach loop when used in a 2D array iterates through each row.

EXCEPTION HANDLING :

The exception classes in C# are derived from the System.Exception

1. System.FormatException = Thrown when the format of an argument/data is invalid
2. System.DivideByZeroException = Thrown when dividing by zero
3. System.IndexOutOfRangeException = Thrown when an array index is out of range
4. System.OutOfMemoryException = Thrown out of insufficient free memory
5. System.NullReferenceException = Thrown when a null object is referenced

6. System.IO.IOException = Thrown when I/O errors occur
7. System.OverflowException = Thrown when a number becomes too large to be represented in the bytes

C# provides a structured solution to the exception handling in the form of try and catch blocks
• The try..catch block is used to catch and handle exceptions
• The statements that may throw an exception are coded inside a try block
• Then immediately after the try block,The exception thrown will be caught by a catch block written to handle that exception.
• The finally block always executes whether the try block terminates normally or terminates due to an exception. The finally block is to release the system resources.

Syntax :

```
try{
#statements causing exception
}
catch (ExceptionName e1)
{
#error handling code1
}
catch (ExceptionName e2)
{
#error handling code2
}
finally
{
#statements to be executed even when the exception
is occurred or not
}
```

We can throw exception programmatically using throw statement
Eg :

```
using System;
namespace AppThrow
```

```
{
class Program
{
static void Main(string[] args)
{
int age;
try
{
Console.Write("Enter Age:");
age = Convert.ToInt32(Console.ReadLine());
if (age< 0)
throw new ArgumentOutOfRangeException("Age should
not be a negative value");
}
catch(ArgumentOutOfRangeException ex1)
{
Console.WriteLine("Age cannot be a negative value.."
+ ex1.Message);
}
finally
{
Console.WriteLine("Thanks for using this
application");
}}}
```

We will never need to throw system exceptions such as IndexOutOfRange or NullReferenceException. They are thrown automatically by the runtime. The .NET developer specification requires that we don't throw these exceptions programmatically.

Validating User input :-

When the user enters data, it needs to be checked for validity. We can display an error message if the data entered is not valid and let the user re-enter data.

- Data validation prevents exceptions from being thrown
- The data entered may be validated for different cases:

Var Keyword :

- The var keyword is used to declare *implicit type variables* in C#.
- Implicitly typed variables are those variables which are declared without specifying the .NET data type explicitly.
- We can store any type of values in an implicit type variable
- The compiler determines the type of an implicit variable at run time depending on the value stored in it.
- A var variable needs to be initialized in order to be declared.

Example:

```
var ageValue = Console.ReadLine();
```

// if we enter int type agevalue will become int datatype, if enter string type then agevalue is string data type. //

- ***To validate that the user has entered numeric data if it needs to be :***

Using TryParse Method:-

The TryParse method is used to convert a given string to another type

Eg :-

```
int age;
var ageValue = Console.ReadLine();
bool parseSucess = int.TryParse(ageValue, out age);
```

// The int.TryParse method tries to convert the string stored in 'ageValue' to integer type.
If the conversion is successful, it stores the converted value in the out variable 'age'
If the conversion is successful, it also returns the boolean result 'true'
If the conversion is un-successful, it returns the boolean result 'false'
//

TryParse Method - What it does?

int.TryParse(string,int) :

Converts the string representation of a number to its integer equivalent.
Returns a 'true' conversion succeeded, otherwise returns 'false'

decimal.TryParse(string, decimal) :

Converts the string representation of a number to its decimal equivalent.
Returns a 'true' conversion succeeded, otherwise returns 'false'
double.TryParse(string, double) :
Converts the string representation of a number to its double equivalent.
Returns a 'true' conversion succeeded, otherwise returns 'false'
float.TryParse(string,float) :
Converts the string representation of a number to its float equivalent.
Returns a 'true' conversion succeeded, otherwise returns 'false'
byte.TryParse(string,byte) :
Converts the string representation of a number to its byte equivalent.
Returns a 'true' conversion succeeded, otherwise returns 'false'

- ***To validate that the user has entered one or more characters in response to an input request :-***

☐ The **IsNullOrEmpty** method is used to check whether a specified string is null or an Empty string.
☐ A string will be null if it has not been assigned a value.
☐ A string will be empty if it is assigned with an empty string **“”**.
☐ This method is useful to validate that the user has entered some characters or not, in response to an input request.
☐ It returns 'true' if the string is empty or null, otherwise returns 'false'.
☐ Example:

```
string empName;
Console.Write("Enter the employee name: ");
empName = Console.ReadLine();
if (string.IsNullOrEmpty(empName)==true)
{
Console.Write("Name can't be empty!");
}
else
{
Console.WriteLine("You entered name: {0}",
empName);
}
```

- ***To validate that the numerical value entered is within a specific range :-***
  Use Logical operators or do it normal way.

OOPS :-

- Object Oriented Programming is a way of programming that involves breaking our requirements down into chunks that are more manageable than the whole
- Procedural Programming uses the concept of procedures or functions to break down our large code into more manageable units. It can also be called repeatedly and thereby saving code duplication.

Class and Objects:-

Class is blueprint for an entity. Think about a house blueprint. We can't live in it, but we can build a house from it; we build an instance of it. We implement an object of a class. This is called an instance of a class. *Object is an instance of a class*.

- Objects represent the state and behavior of class
- For example, when the house is built, we can say it has an area of 2000 sq. feet, 4 bedrooms, 3 baths etc. (state of the object). Behavior of an object is the things it can do, such as provide shelter
- Every Class has Unique Name.
- States of an Objects of a class = member variables
- Behavioral of an Objects of a class = member methods

The scope of these variables and methods determine if they can be accessed outside the class in which they are declared.

- Private : - Variables and methods are accessible only within the class in which they are declared If you don't specify any access modifier then 'private' is applied by default
- public :- Variables and methods are accessible from any other class in the same namespace

- Access is limited to only the current Assembly, that is any class or type declared as internal is accessible anywhere inside the same namespace. It is the ***default access modifier in C#***.

Accessing Member Variable and Member Method : -

- First declare an object of the Laptop class in the other class
- Syntax: *Classname objectname = new Classname();*
  Example : `Laptop objLaptop = new Laptop();`
- Now use the dot operator with the object of the class
  Syntax: *Objectname.membervariable=value*
  Objectname.memberMethod(); // calling member methods //
  Example : `objLaptop.color = "Silver";`
  `objLaptop.calculateTax();`

Constructor :-

- Constructor is a special member method of a class
- It has the same name as that of the class
- It is used to initialize an object. That is, It can be used to set initial values for member variables of an object.
- The constructor method is called and executed automatically when an object of the class is created. We do not have to call the constructor methods explicitly.
- It has no return type.
- Note:
  - ☐ All classes have constructors by default
  - ☐ If we do not create a class constructor ourselves, C# creates one for us However, then we are not able to set initial values for member variables.

Example :

```
class Employee {
public string empName;
public DateTime dateofBirth;
public string designation;
public decimal salary;
public Employee()//Constructer
{
empName = "Sam Peter";
```

```
dateofBirth = Convert.ToDateTime("11/08/2000");
designation = "Programmer";
salary = 5500;
}}
```

- Constructors can also accept parameters. These parameters can be used to initialize the object
- Constructor that accepts parameters are called as parameterized constructor
- We can pass parameters to a constructor at the time of creating an object Like this : - `Employee objEmp1 = new Employee("Sam Peter", "12/06/1998", "Programmer", 4700);`

Example : `public Employee (string name, string dob, string desig, decimal empSalary){`

```
empName = name;
dateofBirth Convert.ToDateTime(dob);
designation = desig;
salary empSalary;}
```

Array of Objects :-

- An Array of objects stores a collection of objects. Unlike a traditional array that store values like string, integer etc., an array of objects stores objects of a class
- Suppose we have a Student class and we want to process the details of 100 students, then we need 100 objects of the Student class. It is impractical to declare 100 individual objects of the Student class. In these situations we can declare an array to hold 100 objects of the class Student.
- Example :-

```
Consider the class Student
class Student
{ string name; DateTime dateofBirth; string
programName; decimal fees; public Student() {
}
public void displayDetails(){}}
```

We can declare an array of objects for the Student class as below:

```
Student[] objStudent = new Student [100];
```

Now we can initialize each individual object in the array of objects

```
objStudent[0] = new Student();
objStudent[1] = new Student(); and so on..
```

- The array objStudent will have 100 locations in the memory representing 100 objects
- Each location will be referenced by an index number starting from 0 to 99
- The first object will be identified as objStudent[0], the next one as objStudent[1] and so on up to objStudent[99].
- Each object can access the member variables and member methods using the dot operator

```
objStudent[0].name, objStudent[99].fees;
objStudent[0].displayDetails();
```

Static Classes :-

- A static class is like a non-static class, but the difference is that a static class cannot be instantiated.
- We cannot use the new operator to create an instance variable(object) of the static class type. Because there is no object for a static class, we can access the members of a static class by using the class name itself.
- Consider a static class Bank, with a public static method calculateInterest. We can call this static method from another class using the class name Bank:

```
Bank.calculateInterest();
```

- Static class is usually used as a convenient container for a collection of member methods that just need to operate on input parameters. Static class usually do not have any internal instance fields(member variables). They are normally used to implement utility classes

- There are some in-built static classes available in the.NET Class Library. *System.Math* is one such static class available in the .NET class library.
- This class contains static methods that perform mathematical operations on input parameters
- It does not require to store or retrieve data that is unique to an instance of the Math class.
- The static methods of this class can be accessed using the class name itself, since we cannot create an object of the static class
- **Method - Function - Example - Result:-**
- Abs() - Returns the absolute value of a specified number - Math.Abs(-6) - o/p : 6
- BigMul() - Returns the full product of two 32-bit numbers - Math.BigMul(12,76) - o/p : 912
- Sqrt() - Math.Sqrt(16) - 4
- Cbrt() - Returns the cube root of a specified number - Math.Cbrt(27) - 3
- Ceiling() - Math.Ceiling(6.765) - o/p : 7
- Floor() - Math.Floor(6.765) - o/p : 6
- Max() - Math.Max(5,12) - o/p : 12
- Returns the maximum of two specified numbers
- Min() - Math.Min(5,12) - o/p :5
- Pow() - Math.Pow(5,2) - o/p : 25
- Round() - Math.Round(4.255) - o/p : 4
  Math.Round(4.665) - o/p : 5
  Math.Round(4.565,2) - o/p : 4.66
- Truncate() - Returns the integral part of a number - Math.Truncate(4.665) - o/p : 4
  Math.Truncate(5.245) - o/p : 5
- Sign() - Math.Sign(16.25) - o/p : 1
  Math.Sign(-5) - o/p : -1
  Math.Sign(0) - o/p : 0
- Sin() - Math.Sin(45) - o/p: 0.85
- PI - Math.PI - 3.14159265358979
  Math.Round(Math.PI, 2) - 3.14

Value and Reference Type :-

- As our code executes, there are two locations in memory where .NET store items - Stack and the Heap. They reside in the operating memory (RAM) on our machine.
- Stack is responsible for keeping track of what's executing in our code
- Heap is responsible for keeping track of our objects

Stack :

- Stack can be visualized as a series of boxes stacked one on top of the next. Every time we call a method, we keep track of what is going on in our application by stacking another box on top.
- We can only use what is in the top box on the stack.
- When a method is completed execution, we are done with the top box of the stack. Then we throw it away and proceed to use the stuff in the previous box on the top of the stack

Heap :

- Heap is like stack, but its purpose is to hold information (it does not keep track of execution most of the time)
- So anything in our Heap can be accessed at any time

Value Type :

- A data type is a value type if it holds a data value within its own memory space. It means variables of these data types directly contain their values.
- All primitive data types like bool, byte, char, decimal, double, float, int etc.
    - ☐ struct, enum
- Value Types always go where they were declared, normally on the Stack. Consider the statement:
- `int i=10;` //The system stores 10 in the memory space allotted for the variable 'i' on the Stack 0x328211 //

Refrence - Type :

- Reference type does not store its value directly. Instead, it stores the address where the value is being stored. In other words, a reference type contains a pointer to another memory location that holds the actual data.(because they are from System.Object):
- Example : String, All arrays, Class, Delegates

- Consider the following statement,
  `string s = "C# Programming"` //Memory Allocation for Reference Type(Heap). Reference type variable contains address of the location where the value is stored. //
- The system selects a random location in the memory 0x701231 for the variable 's' The value of the variable 's' is 0x328211 which is the memory address of the actual data in the Heap.

STRUCTURES :

Structure is a value type that can encapsulate data and related functionality in a single unit.

- It helps us to make a single variable hold related data of various data types.
- Structures are used to represent a record.
- Structures are quite like classes but has many differences.
- Example :-

```
struct Book {
public string id;
public string title;
public string author;
public string subject;
public string publisher;
public decimal price;
public Book (parameters){.....} //parameterized
constructor //
public void DisplayBookInfo() {//method//}
```

- A structure can also have constructor like class. But structure cannot have default constructor like class. Structure can have only parameterized constructor.
- We can access a structure using its object. Object of a structure can be created in the same way as that of a class.
- The new key word is used to create an object of the structure
- The dot operator can be used to access the fields and methods of an object
- Creates an object book1 of the structure Book:

```
Book book1 = new Book();
```

- Accesses the field author of the structure Book:
  `book1.author;`
- Accesses the method DisplayBookInfo() of the structure Book:
  `book1.DisplayBookInfo();`

Difference Between Structures and Classes :

Structure : -

- Structure is a value type, so it is faster than a class object.
- Structure is used whenever we want to just store the data. Generally structures are good for game programming.
- Structure can have parameterized constructor only.
- Structure cannot be used as a base, and hence cannot derive child class (inheritance not supported).
- Structure members cannot be specified as abstract.

Class :-

- Class is a reference type.
- It is easier to transfer a class object than a struct. So do not use struct when we are passing data across the wire or to other classes.
- Class can have default constructor and parameterized constructor
- Class can be used a base and derive child class from the base class (inheritance).
- Class members can be specified as abstract.

Enum in C#

- enum (enumeration) is a value type data type. It is used to create a list of named integer constants. It can be defined using the enum keyword directly inside a namespace, class, or structure
- enum is used to give a name to each constant so that the constant integer can be referred using its name
- For example, the days of the week can be defined as an enumeration and used anywhere in the program
- Example:

```
enum WeekDays
{ Monday,
```

```
Tuesday,
Wednesday,
Thursday,
Friday,
Saturday,
Sunday }
```

- By default, the first list item of an enum has the value 0 and the value of each n successive enum member is increased by 1. For example, in the above enumeration, Monday is 0, Tuesday is 1, Wednesday is 2 and so on.
- We can access a list item from an enum using the enum name and the dot operator:

```
Console.WriteLine((int) WeekDays. Monday);
//Outputs 0
Console.WriteLine((int) WeekDays. Friday);
//Outputs 4
```

- Traverse through an enum using a foreach loop :-

```
foreach (string day in Enum.GetNames (typeof
(WeekDays)))
Console.WriteLine(day);
```

GetNames: Returns an array of the list of items in an enum

```
foreach (int dayNumber in Enum. GetValues
(typeof (WeekDays)))
Console.WriteLine (dayNumber);
```

GetValues: Returns an array of the list of values in an enum

- We can assign an integer constant to the list items of an enum :

```
enum WeekDays {
Monday=1,
Tuesday=2,
Wednesday=3,
Thursday=4,
Friday=5,
Saturday=6,
Sunday=7
}
Console.WriteLine((int) WeekDays. Monday);
//Outputs 1
```

```
Console.WriteLine((int) WeekDays. Friday);
//Outputs 5.
```

COLLECTIONS :-

Collection is a class that is useful to manage a group of objects in a flexible manner. We can perform various operations like insert, update, delete, get, etc. on a collection in a dynamic way based on our requirements.

In C# applications, we may come across with requirements like creating or managing a group of related objects. We can create group objects in two ways:

- by using the arrays
- by using collections

But arrays are useful only when we are working with a fixed number of strongly-typed objects. It cannot grow or shrink dynamically.

Collections are useful to work with a group of objects which can grow or shrink dynamically based on our requirements. That is collections are mutable.

We have different type of collection classes available:

- Non-Generic
- Generic

Non-Generic Collections: In non-generic collections, each element can represent a value of a different type. They are useful to store elements of different data types. The collection size is not fixed. Items from the collection can be added or removed at runtime. The Non-Generic collections are provided by the System.Collections namespace. The Non-Generic collections are now legacy types. Example: ArrayList, Hashtable

Generic Collections: Generic collections enforces type safety and so we can store only the elements the same data type. In case of Generic collections also the collection size is not fixed. Items from the collection can be added or removed at runtime. The Generic collections are provided by System.Collections.Generic namespace. Example: List<T>

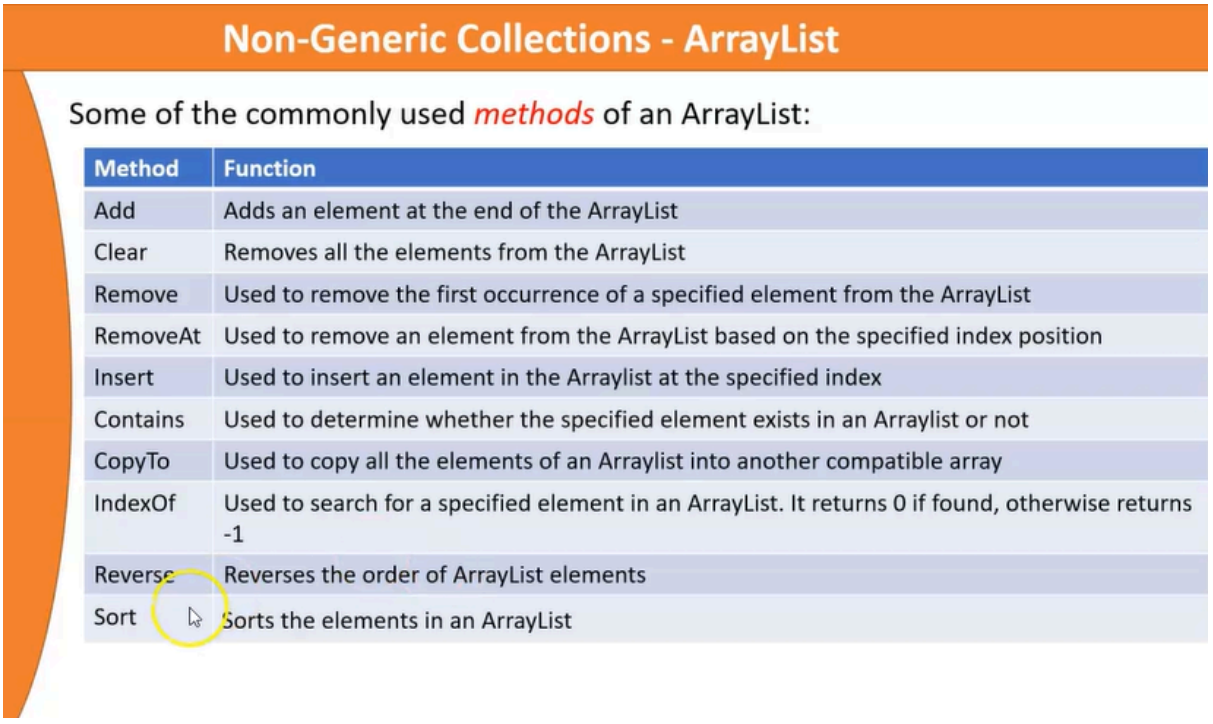
Non-Generic Collections - ArrayList :-

ArrayList is a Non-Generic collection. It is useful to store elements of different data types. The size of the ArrayList can grow or shrink dynamically based on the requirement of our application. We can manipulate an ArrayList at run time by adding or removing elements from the ArrayList.

- ArrayLists are like arrays but the difference is that arrays are used to store a fixed number of elements of the same data type.
- ArrayList is provided by the *System.Collections* namespace.
- To use ArrayList in an application, we need to import the System.Collections namespace into the application with the help of the using statement.

```
using System.Collections;
ArrayList arList = new ArrayList();
```

A new arraylist can be created with an instance of ArrayList class without specifying any size and data type.



Non-Generic Collections - ArrayList

Some of the commonly used *properties* of an ArrayList:

Property	Function
Count	Returns the number of elements in an ArrayList `arList.Count`
Item	Used to get or set an element at the specified index of an ArrayList It is an indexer property, so we don't have to use the property name, but just the index `arList[2] – Returns the item at index position 2`

See the following code segment to understand the working of an ArrayList:

```
using System;
using System.Collections; //Imports the namespace for ArrayList
namespace AppArrayList
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arListCountries = new ArrayList(); //declares an ArrayList, note that no
                                                          size specified
            arListCountries.Add("Canada"); //Adds a string type element to the ArrayList
            arListCountries.Add("USA");
            arListCountries.Add("Germany");
            arListCountries.Add(2300);  //Adds an integer type element to the ArrayList (Note
                                          that mixed type of elements can be added to an ArrayList
            arListCountries.Add("UK");
            Console.WriteLine("ArrayList Count: " + arListCountries.Count); //Returns the number of
                                                                              items in the
                                                                              ArrayList - 5
            Console.WriteLine("Item at index 2: " + arListCountries[2]); //Returns the item at
                                                                          index position 2 from
                                                                          the ArrayList - Germany
```

Non-Generic Collections - ArrayList

```
            Console.WriteLine("*********ArrayList Elements********");
            foreach (var item in arListCountries) //Traverses through the ArrayList to retrieve all
                                                    items
            {
                Console.WriteLine(item);
            }

            arListCountries.Reverse(); //Reverses the ArrayList
            Console.WriteLine("*********ArrayList Elements in the Reverse Order********");
            foreach (var item in arListCountries)
            {
                Console.WriteLine(item);
            }
            Console.ReadKey();
        }
    }
}
```

HashTable :-

Non-Generic Collections - Hashtable

In the case of *Array and ArrayList*, we can access the elements from the collection *using an index*. The index starts from 0 to the number of elements -1. It is *very difficult to remember the index position* of the element in order to access the values.

Hashtable is a collection that *stores the element* in the form of "*Key-Value*" *pairs*. The data in the Hashtable are organized based on the hash code of the key.

The key is defined by us and the key can be of any data type. We can access the elements of a Hashtable using the keys. It is easy to remember the keys rather than a numerical index.

Hashtable is provided by *System.Collections* namespace.

Since Hashtable is a collection class, we can use the new operator to declare a Hashtable

```
Hashtable hTable = new Hashtable();
```

A new Hashtable can be created with an instance of `Hashtable` class without specifying any size and data type

Some of the commonly used *methods* of a Hashtable:

Method	Function
Add	Used to add an element with a specified key and value in a Hashtable *hashtable.Add("key",value)*
Remove	Used to remove an element with a specified key from the Hashtable *hashtable.Remove("key")*
Clear	Removes all elements from the Hashtable *Hashtable.Clear()*
Contains	Finds whether the Hashtable contains a specific key or not. Returns 'true' if it is contained, otherwise 'false' *hashtable.Contains("key")*
ContainsKey	Finds whether the Hashtable contains a specific key or not. Returns 'true' if it is contained, otherwise 'false' *hashtable.ContainsKey("key")*
ContainsValue	Finds whether the Hashtable contains a specific value or not. Returns 'true' if it is contained, otherwise 'false' *hashtable.ContainsValue(value)*

Some of the commonly used *properties* of a Hashtable:

Property	Function
Count	Returns the number of key/value pair elements in the Hashtable *hashtable.Count*
Item	It is used to get or set the value associated with the specified key. We don't have to use the property name, but just the key *hashtable["key"]*

We can use *DictionaryEntry* to get key/value pairs from the Hashtable using a *foreach loop*

```
foreach (DictionaryEntry item in hTable)
  {
       Console.WriteLine($"Key = {item.Key}, Value = {item.Value}");
  }
```

See the following code segment to understand the working of an Hashtable:

```
using System;
using System.Collections; //Imports the namespace for Hashtable
namespace AppHashtable
{
    class Program
    {
        static void Main(string[] args)
        {
            Hashtable hTable = new Hashtable(); //declares a Hashtable, note that no size
                                                  specified
            hTable.Add("January", 31); //Adds a key-value pair to the Hashtable
            hTable.Add("February", 29);
            hTable.Add("March", 31);
            hTable.Add("April", 30);
            Console.WriteLine("Hashtable Count: " + hTable.Count); //Returns the number of items
                                                                     in the Hashtable
            Console.WriteLine("Value for the key February: " + hTable["January"]);//Returns the
                                                                  value at the key 'January'
            Console.WriteLine(hTable.ContainsKey("January"));//Returns 'true' since the key
                                                            'January' is contained in the Hashtable
            Console.WriteLine(hTable.ContainsValue(31));//Returns 'true' since the value 31 is
                                                          contained in the Hashtable

Console.WriteLine("*********Hashtable Elements********");
foreach (DictionaryEntry item in hTable) //Traverses through the Hashtable
 {
    Console.WriteLine($"Key = {item.Key}, Value = {item.Value}");
 }
    hTable.Remove("January");//Removes the element(kay-value pair) with the key 'January'
    hTable.Clear();//Removes all elements from the Hashtable
    Console.ReadKey();
        }
    }
}
```

Scope of Variables:-

Scope of a variable is related to its *accessibility in an application*. The part of the program where a specific variable is accessible is termed as the scope of that variable. A variable can be defined in a class, method, loop etc.

Scope of variables can be divided into the following categories:

- Global Scope
- Class Level Scope
- Method Level Scope
- Block Level Scope

Global Scope: Global variables have a global scope and it will be available to all the classes of the application. C# *does not support global variables* as such. But *static variables* declared *in the class that contains main()* will be available globally.

```
namespace AppBinaryFile
{
   class Program
    {
        static double interestRate = 13.5;
        static string fileName = "products.dat";
        static void Main(string[] args)
        {
```

Now the static variables *interestRate* and *fileName* will be accessible in all the classes of the application.

Class Level Scope: Declaring the *variables in a class but outside any method* can be directly *accessed anywhere in the class*. These variables are also termed as the fields or class members.

```
class Product
      {
          public string productId;
          public string name;
          public double price;
          public DateTime manufactDate;
          public DateTime expiryDate;
          public Product(string id, string name, double price, DateTime
                         manuDate, DateTime expDate)
          {
              this.productId = id;
              this.name = name;
              this.price = price;
              this.manufactDate = manuDate;
              this.expiryDate = expDate;
          }
      }
```

Class Level Variables

Method Level Scope: Variables that are *declared inside a method* have method level scope. These variables *can be accessed inside that method only*. These are *not accessible outside the method.*
These variables are termed as the *local variables.*

```
static void AddNumbers()
 {
   int n1,n2
   ---------
   ---------
 }

static void SubtractNumbers()
  {
    int x1,x2
    ---------
    ---------
  }
```

Variables n1, and n2 are local variables for the method AddNumbers, so they are accessible inside that method only. They are not accessible to the method SubtractNumbers.

Block Level Scope: These variables are generally declared inside the for, while statement etc. They have limited their scope up to the body of the statement in which it is declared. The variables declared inside a for loop is only accessible in that loop, and not accessible outside that loop.

```
for (int j = 0; j < 5; j++)
 {
   // accessing block level variable j
   Console.WriteLine(j);
 }
```

Variables 'j' is declared in the for loop. So it is block level variable and can be accessed only in that for loop. It is not accessible outside that loop.

Date Time in C# :-

Date Time in C# is a structure of value Type like int, decimal etc. It is available in the System namespace.

- DateTime.Now => will return Current date and Local Time.
- MinValue - It returns the smallest possible value of DateTime.(DateTime.MinValue => 0001-01-01 12:00:00 AM)
- MaxValue - It returns the largest possible value of DateTime.(DateTime.MaxValue => 9999-12-31 11:59:59 PM)

DateTime object has many different *properties* as listed below:

Property	Function
Day	Returns the day of the month of the date
Month	Returns the month component of the date as a value between 1 and 12
Year	Returns the year component of the date
DayOfYear	Returns the day of the year
TimeOfDay	Returns the time of the day component of a date
Hour	Returns the hour part of the time component
Minute	Returns the minute part of the time component
Second	Returns the second part of the time component
Millisecond	Returns the milli second part of the time component
DayOfWeek	Returns the day of the week like Sunday, Monday etc.
Kind	Returns a value that indicates whether the time represented is based on local time, or Coordinated Universal Time (UTC)

-
- We should use this property as
  DateTime now = DateTime.Now;
  Console.WriteLine(now.Day + "/"+now.Month + "/" + now.Year);
- We can add or subtract Minutes, Days, Seconds, hrs etc…

DateTime has a variety of *methods* to manipulate DateTime Object as below:

Method	Function
AddSeconds	Adds the specified number of seconds to a date time and returns new DateTime
AddMinutes	Adds the specified number of minutes to a date time and returns new DateTime
AddHours	Adds the specified number of hours to a date time and returns new DateTime
AddDays	Adds days to the DateTime. We can specify both positive or negative values
AddMonths	Adds months to the DateTime. We can specify both positive or negative values
AddYears	Adds years to the DateTime. We can specify both positive or negative values
Parse	Converts a DateTime string to a DateTime object

```
DateTime db = DateTime.Now();
DateTime date = db.AddSeconds(30); // for AddSeconds, AddMinutes, AddHours should be in +ve.
DateTime date2 = db.AddDays(-20); // for AddDays, AddMonths, AddYears can be in +ve and -ve.
```

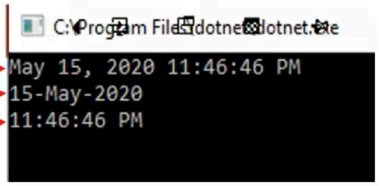
We may need to represent datetime in different formats according to the user requirements like dd/mm/yy or mm/dd/yy. There are different datetime format specifiers available as below:

Format Specifier	Description	Sample Output
d	Short Date	12/4/2020
D	Long Date	Friday, May 15, 2020
t	Short Time	6:20 PM
T	Long Time	6:20:10 PM
f	Full date and time	Friday, May 15, 2020 6:20 PM
F	Full date and time (long format)	Friday, May 15, 2020 6:20:10 PM
dd	Day	15
ddd	Short day name	Fri
dddd	Full day name	Friday
MM	Month	10
MMM	Month Name(short)	Oct

Format Specifier	Description	Sample Output
MMMM	Month name(long format)	October
yy	2 digit year	20
yyyy	4 digit year	2020
hh	2 digit hour(12 hour clock)	6
HH	2 digit hour(24 hour clock)	18
mm	2 digit minute	25
ss	seconds	13
tt	AM/PM	

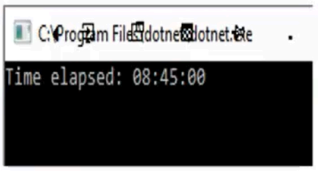
Example:

```
DateTime today = DateTime.Now;
Console.WriteLine(today.ToString("F"));
Console.WriteLine(today.ToString("dd-MMM-yyyy"));
Console.WriteLine(today.ToString("hh:mm:ss tt"));
```



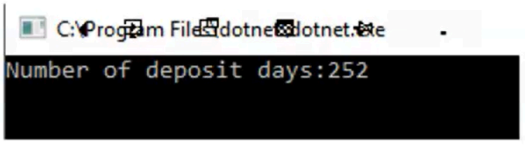
TimeSpan is a *structure type*. It represents *time intervals* and the intervals can be expressed in terms of days, hours, minutes , and seconds. It is also available in the *System namespace*.

```
string beginTime = "9:45 AM";
string endTime = "6:30 PM";
TimeSpan timeElapsed = DateTime.Parse(endTime).Subtract(DateTime.Parse(beginTime));
//TimeSpan timeElapsed = DateTime.Parse(endTime)-DateTime.Parse(beginTime));
Console.WriteLine($"Time elapsed: {timeElapsed}");
```



- A TimeSpan object can be declared using the *keyword TimeSpan*
- The *Parse method* converts the string representation of a time *to a DateTime object*
- The *Subtract method* subtracts two time values(We can also use the *minus* – operator instead of Subtract method

```
DateTime withdrawalDate = DateTime.Today;
DateTime depositDate = new DateTime(2019, 9, 7);
TimeSpan diff = withdrawalDate-depositDate;
Console.WriteLine($"Number of deposit days:{diff.TotalDays}");
```



- The *TotalDays* property of the TimeSpan object returns the *number of days* of the elapsed time
- TotalDays returns a double type value by default

```
DateTime withdrawalDate = DateTime.Today;
string depositDate = "9/7/2019";
TimeSpan diff = withdrawalDate - Convert.ToDateTime(depositDate);
Console.WriteLine($"Number of deposit days:{diff.TotalDays}");
```

- A time zone is a region of the globe that follows a uniform standard time for legal, commercial, and social purposes. There are many different time zones in the world. The TimeZoneInfo class is used to manipulate time zones in C#.
  - The GetSystemTimeZones() method returns the time zones.
  - TimeZoneInfo zone = TimeZoneInfo.Local;

```
Console.Writeline($"Current Timezone: {zone.StandardName}");
Console.WriteLine($"Daylight Name: {zone.DaylightName}");
Console.ReadKey();
```

StandardName property returns the current zone's standard name
DayLightName property returns the daylight-saving name

- DateTime.IsLeapYear(year) => true/false => this will say whether year is Leap Year or not.
-