My C# and .NET Notes

THIS IS MY JOURNEY WITH C# AND .NET ECO SYSTEM

C S Srikanth

Contents

C# and .NET

Why C#?

Flexible for many applications types with the help of .NET (Current version 8), especially business applications
Web (Asp.net)
Windows (WinForms, WPF and UPF)
Console Apps
Mobile Apps (Android, Ios and MacOS by Mono/Xamarin)
Graphics / Games by (Unity)
Cross Platform by .NET Core frame Work
Cloud (Azure)
IOT and Ai applications
Opensource (Lots of support on stack Overflow etc)
Current Version C# 12 (Refer wiki for Iterations details)
OOPS (Object Oriented Programming)

What is .NET Framework, .NET Core, .NET Standard, and .NET?

Evolution of .NET and Microsoft's Open-Source Journey

1. Early Days of Microsoft Development
    - Microsoft initially had Visual Basic (VB), which was easy to use but not powerful enough for serious applications.
    - Developers turned to C++ for building Windows applications, but it had many complexities and challenges.
2. Competition from Java
    - Java introduced a cross-platform Virtual Machine (JVM) and modern programming principles, making it a strong competitor.
3. Birth of .NET
    - Microsoft introduced .NET to provide a unified development framework.
    - At its core, .NET had the Common Language Runtime (CLR), which allowed different programming languages to work together.
    - Initially, .NET even supported Java, but the ecosystem remained closed-source and Windows-centric.
4. Mono and Open-Source Movement
    - The open-source community-built Mono, an alternative CLR for non-Windows platforms, which, in some cases, performed better than Microsoft's CLR.
    - Seeing this, Microsoft decided to open-source the CLR, C# compiler, and Base Class Library (BCL).
    - While doing so, they simplified and optimized .NET by removing unnecessary components, leading to .NET Core.
5. Integration of Mono and .NET Core
    - The Mono team joined Microsoft's .NET team.
    - With open-source collaboration, they gained direct access to BCL and other essential components, which they previously had to rewrite.

6. Why Did Microsoft Go Open Source?
    - Earlier, Microsoft relied on licensing revenue (e.g., selling SQL Server, Office, and .NET Framework licenses).
    - However, with the rise of cloud computing, licensing became less effective.
    - Cloud computing became the future, and Linux was the dominant OS for cloud servers.
    - To adapt, Microsoft:
        - Made .NET Core free and open-source.
        - Released VS Code for free.
        - Integrated Linux support in Windows (WSL, Docker, etc.).
        - Shifted revenue focus to cloud services (e.g., Azure).
7. .NET Today
    - .NET Framework is now considered legacy.
    - .NET Core evolved into just .NET, which is the future.
    - The .NET Foundation, a non-profit organization, now oversees the .NET ecosystem and C# development.
    - Microsoft shares .NET source code, team meetings, and development updates on public platforms.

🔗 You can explore the .NET source code here: https://source.dot.net/

.NET Framework

Used for building Web Applications (ASP.NET), Windows Applications (WinForms, WPF), Windows Server Apps, and Azure Apps. And Comes with Base Class Library (BCL) and Common Language Runtime (CLR).

Provides built-in services like:

- Memory Management
- Type and Memory Safety
- Security
- Networking
- Application Deployment
- Data Structures and APIs for lower-level operations
- Supports around 100 programming languages.

.NET Core

Includes all the essential features of .NET Framework but removes bad implementations and unnecessary components.

- Cross-platform and open-source.
- Officially supports C#, F#, and VB.

.NET Standard

A set of rules (minimum requirements) that every .NET platform must follow to ensure compatibility.

Why was it needed?

- Mono was used for cross-platform apps. And .NET Framework was for Windows apps.
- These two were separate and could not be merged easily.
- .NET Standard defined common features to make shared libraries work across different platforms.

When writing libraries, they should target .NET Standard for maximum compatibility.
Compatibility details: Check here.

.NET (Modern .NET – No More Core or Standard)

Everything merged into one unified framework—.NET Core, .NET Framework, Mono, and Xamarin are now just .NET.
.NET 5 was the first version of this unified approach (now out of support).
.NET 7 is the current standard support version.
.NET 8 is the latest Long-Term Support (LTS) version.
Where does .NET 7+ run?

- Smart TVs, iOS, Android, Web Browsers, Unity, Windows, Linux, macOS.

With .NET 7 and beyond, code runs anywhere .NET runs—making development easier.

So, after all This what should we do

1. For New Applications or Libraries
    - Use the latest .NET version.
    - Check which version has Long-Term Support (LTS) before choosing.
    - Download and learn more here: Download .NET.
2. For Backward-Compatible Libraries
    - If targeting .NET Core, use .NET Standard 2.1.
    - If targeting .NET Framework, use .NET Standard 2.0 or lower (since many older applications still rely on it).

This ensures your code remains future-proof while maintaining compatibility where needed.

.NET (in a nut shell)

.NET is an infrastructure for building applications using .NET-supported languages like C#, F#, and VB.

- .NET Core, .NET Framework, .NET Standard, and Mono/Xamarin are now merged into .NET.
- A cross-platform, open-source ecosystem that includes languages, runtime support, and libraries.
- Provides an SDK (Software Development Kit) to develop business applications.
- Uses NuGet for third-party packages and libraries.
- Key components:
    - .exe (Executable with Main Method)
    - .dll (Dynamic Link Library / Assembly)
    - .pdb (Portable Debug File)
- Commonly used features:
    - Lambda Expressions
    - Entity Framework
    - WinForms, ASP.NET, etc.

This makes .NET a powerful and flexible platform for modern application development.

Understanding .NET Components

Language Compiler

- Converts source code into MSIL (Microsoft Intermediate Language).
- Each language (C#, VB, F#) has its own compiler.
- Errors like syntax errors and logical errors are detected here (1st phase of compilation).
- After compilation, the MSIL is processed by the CLR.

CLR, CTS, CLS, and FCL

Common Language Runtime (CLR) – The Heart of .NET

- The execution engine of .NET, also called VES (Virtual Execution System).
- Functions of CLR:
    - Memory Management (Garbage Collection).
    - Converts MSIL to native machine code.
    - Ensures type safety.
    - Provides language, platform, and architecture independence.
    - Optimizes performance and security.
- Key Components of CLR:
    - JIT (Just-in-Time Compiler) → Converts MSIL to native code (2nd phase of compilation).
    - Class Loader → Loads required classes at runtime.
    - Thread Support → Manages multithreading.
    - Code Manager → Handles runtime code execution.
    - Exception Handling → Manages runtime errors.
- No pointers in .NET (improves safety but may slightly reduce speed).

Common Type System (CTS)

- Ensures consistent data types across .NET languages.
- Example: int in C# and Integer in VB.NET are both treated as Int32 in IL/CLR.

Common Language Specification (CLS)

- A set of rules that .NET languages must follow to work together.
- CLS is a subset of CTS, ensuring compatibility between languages.

Framework Class Library (FCL)

- A collection of predefined classes and methods.
- Example:
    - Namespaces like using System; allow access to built-in functionality.
    - Methods like Console.WriteLine() come from FCL.

The .NET Execution Flow

1. CTS, CLS, and FCL are used to write the source code.
2. Compiler converts source code to MSIL.
3. CLR picks up the MSIL and executes it with its built-in features.

This structure ensures that .NET applications are efficient, secure, and cross-platform.

.NET Development Essentials

Command-Line Interface (CLI)
The .NET CLI is useful for quick operations without opening an IDE. Here are some basic commands:

- dotnet help → Lists available commands.
- dotnet new → Shows available templates for different applications.
- dotnet new console → Creates a new console application.
- dotnet run → Builds and runs the application.

.csproj (C# Project File)

- Turns a folder into a C# project, similar to a package.json in JavaScript.
- Contains project configurations like:
    - Output Type (e.g., .exe, .dll).
    - Target Framework Moniker (TFM) (e.g., .NET 5, .NET Standard 2.1).
    - Root Namespace (Main namespace of the project).

Build and Output Folder (bin/)

- When you build an app, output files go into the bin/ folder.
- Based on Build Type:
    - Debug → Slower, includes extra information for debugging.
    - Release → Optimized for production, smaller and faster.
- Always publish your application properly instead of manually copying files.

Publishing Applications
When publishing, you can adjust:

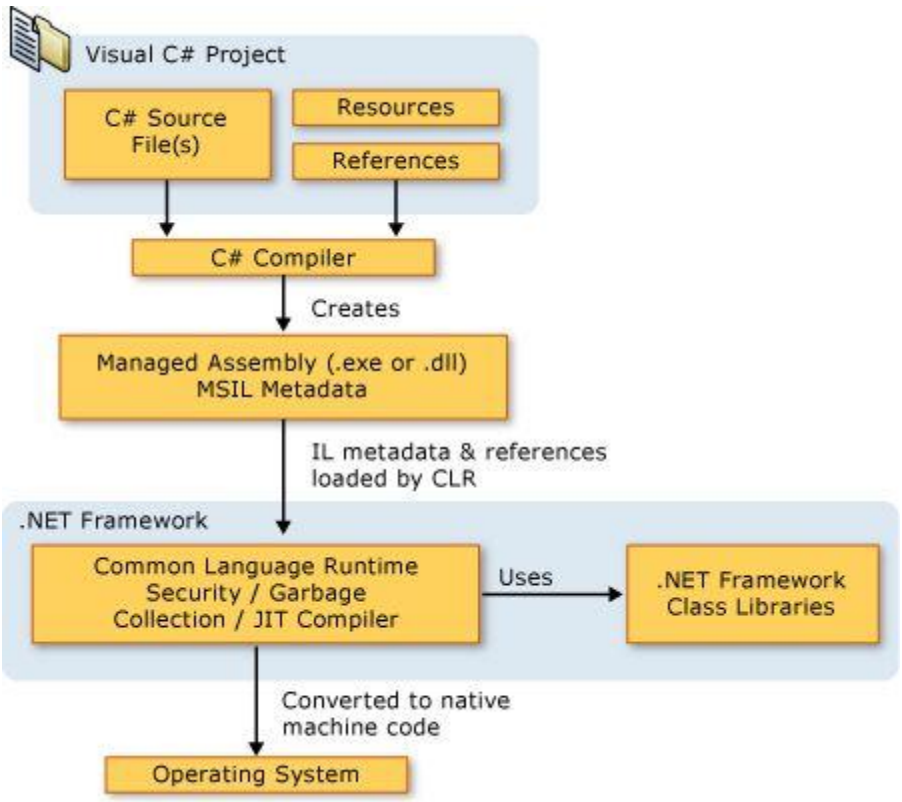
- Target Framework (e.g., .NET 5, .NET 8).
- Deployment Mode:
    - Self-contained → Packages all required DLLs, so .NET does not need to be installed on the client system.
        - Works on Windows, Linux, macOS, ARM.
    - Framework-dependent → Requires .NET to be installed on the client, making the package smaller.

Additional Publishing Features

- Produce Single File → Bundles all DLLs into one executable file.
    - During runtime, it unzips into memory and runs (minimal performance impact).
- Trim Unused Code → Removes unnecessary code to make the app smaller.
- Enable ReadyToRun Compilation → Precompiles code for faster startup.

These options help optimize performance, reduce size, and improve deployment flexibility

.NET Application Life Cycle



Reference from C# Corner

Follow the below link for Learning: By Rainer Stropek and HTL Leonding (Learning) (Git)

Understanding Memory Management in .NET Performance

.NET automatically manages memory using a Garbage Collector (GC), which removes objects that are no longer in use. This helps free up memory for new tasks. GC is part of CLR (Ref Link)

Stack and Heap Memory

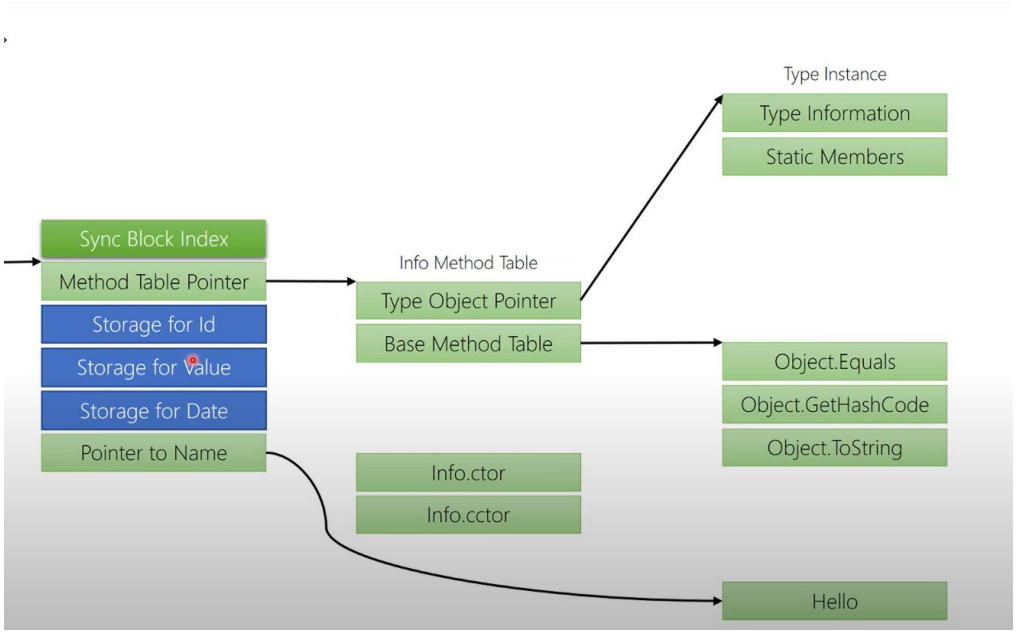
When a .NET application starts, the operating system assigns a **chunk of memory (virtual memory)**. This memory is divided into two parts:

1. **Stack** – Stores local variables and method calls.
2. **Heap** – Stores objects that need to persist beyond a single method call.
   Even though the stack and heap share the same memory space, they work differently:

- **The Stack is unmanaged** – It clears itself when a method finishes. This makes it **very fast**.
- **The Heap is managed by the Garbage Collector** – It checks which objects are still needed and removes unused ones. This makes it **slower than the stack**.
- Both Stack and Heap can grow in size

How .NET Stores Data

- Value types (like int, float, bool, struct) are stored in the stack.
- Reference types (like class, array, string) are stored in the heap.
- If a struct has a reference type property, that property is stored in the heap, even though the struct itself is in the stack.
- A class is stored in the heap, but a variable holding its reference is in the stack.
- Value types are stored close together in memory, making them faster to access. We can use special attributes to control how they are stored.
- Memory allocation (reserving space) in C# is very fast, but freeing up memory (deallocation) takes longer because the Garbage Collector has to clean up unused objects.
- Arrays are reference types, meaning they are stored in the heap. However, they can be optimized for better performance.
- Stack memory is cleared quickly – When a function ends, everything above the pointer is removed at once, making it very efficient.
- Each method call creates a "stack frame", which is added to the stack. This happens separately for each thread in the application.



Memory Layout of an Object – Reference Type

When an object is being created there are a lot of allocations happening

Object Memory Layout

When an object is created, it consists of:

- Sync Block Index – Used for threading (e.g., locks).
- Method Table Pointer – A 4-byte reference to methods of the class.
- Base Method Table – Links to parent class methods.
- Constructor (ctor) and Destructor (cctor) – Handle object creation and cleanup.
- Type Object Pointer – Used internally for managing types.
- Bare minimum size of a .Net object is 12 Bytes in 32 bit and 24 bytes in 64 bits

Why Are Loops and Method Calls Expensive?

- Every method call creates a stack frame, which takes time to manage.
- Loops and methods cause frequent memory allocations, increasing CPU workload.
- Procedural programming (sequential execution) is faster because of better memory layout.
- JIT Compiler (Just-In-Time Compiler) optimizes performance during execution.

What Are Inline Methods?

- Inlining is when the compiler replaces a method call with the actual method code to improve performance.
- It reduces the overhead of method calls, making execution faster. which simplifies memory management in .NET while keeping performance in mind.

Useful Classes and Tools in .NET

Important Classes

- Math Class → Provides mathematical functions like Math.Sqrt(), Math.Pow(), Math.Abs(), etc.
- Random Class → Generates random numbers (Random.Next(), Random.NextDouble()).
- StringBuilder Class → Efficiently modifies strings without creating multiple objects (useful for performance).
- String Methods and Properties → Methods like .Substring(), .Replace(), .ToUpper(), and .Trim() are useful for string manipulation.
- BenchMark .NET for Performance testing

Useful Tools

- .NET CLI (Command Line Interface) → Helps run, build, and manage .NET projects from the terminal.
    - Works with STDIN (input), STDOUT (output), and STDERR (error stream).
- dnSpy → A reverse engineering tool to decompile MSIL (Microsoft Intermediate Language) and convert it back to C#.
- PerfView → A performance analysis tool to observe Garbage Collection (GC) behavior in .NET applications.
- Mockaroo → A free tool to generate sample/mock data for testing databases and applications.
    - Website: https://mockaroo.com/

These tools and classes help in building, debugging, optimizing, and testing .NET applications efficiently.

Best IDE for C# Development

Visual Studio: The most comprehensive and popular IDE for C# development, offering advanced debugging, IntelliSense, and extensive .NET support.

Visual Studio Code: A lightweight, cross-platform editor with C# support through extensions, suitable for those who prefer a more minimal setup.

Fundamentals

Some coding jargons

Instantiation: Creating an object from a class.
Initialization: Assigning a value to a declared variable.
Declaration: Declaring a variable with a specific data type.
Realization / Implementation: Writing the logic to fulfill a specific requirement.
Debugging: Running the debugger to step through each line of code.
Statement: A single line of code or instruction.
Runtime: The period when the application is running.
Compile Time: The process of building the code.
Class Members: The fields, properties, methods, and variables inside a class.
Member Variable: A variable that is part of a class.
Local Variable: A variable that exists only within a method's scope.
Global Variable: A variable that is accessible across multiple classes (placed outside of any namespace to make it globally available).

Simple C# Code Structure

```
using System;
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("Hello, World!");
  }
}
```

Namespace

A collection of classes, interfaces, structs, enums, and delegates. Using statements at the top of the code page indicate that you are using those namespaces.

Class

A bundle of code that includes methods, variables, fields, properties, and constructors. It is a user-defined reference data type and serves as a blueprint for objects.

Object

An instance of a class. A class can be instantiated to create objects.

Method

A unit of code that can be reused and returns a result.

Explanation**:**

- **using System;** → Includes system libraries (**Namespace**).
- **class Program** → Defines a class named Program (**Class**).
- **static void Main()** → The main entry point of a C# program (**Method**).
- **Console.WriteLine("Hello, World!");** → Prints output to the console.

Comments

Comments are Ignorable Lines in Code usually for notes in code.
// Comments //- Single Line
/*
*Multi Line
*Comment
*/

String Interpolation

Use **$** before the string and wrap variables inside { }.
Why Use string interpolation
Easier to read – No need for + operators.
Better for long sentences – Avoids confusion.
Works with expressions – You can use calculations inside { }.

```
string name = "Hero";
// Standard way (Concatenation)
 Console.WriteLine("Hello " + name + "!");

 // Using String Interpolation (Cleaner way)
Console.WriteLine($"Hello {name}!");
```

Escape Sequences

(Check wiki for all escape sequences)
Output/ string literals are modified by the usage of Escape Sequences (/n new line , @ to ignore escape sequence).

Variables

A **variable** is like a container that holds data, just like in mathematics. In programming, variables store different types of values.
Common Data Types in C#:

1. **int** – Stores whole numbers (e.g., 10, -5).
2. **string** – Stores text (e.g., "Hello").
3. **float** – Stores decimal numbers (e.g., 3.14f).
4. **double** – Stores larger decimal numbers with more precision (e.g., 3.1415926535).
5. **bool** – Stores true or false values.
6. **char** – Stores a single character (e.g., 'A').

A constant is a variable whose value cannot change once assigned.

- Use the **const** keyword to define a constant.
- Constants help prevent accidental changes in the program.

Data Types

In C#, data types are classified into **Primitive** and **Non-Primitive** types.

1. Primitive Data Types (Basic types)
These are built-in types that store simple values.

Value Types (Store actual values, default = 0)

- int → Whole numbers (e.g., 10)
- float → Decimal numbers (e.g., 3.14f)
- double → More precise decimal numbers (e.g., 3.14159)
- bool → true or false
- char → Single character (e.g., 'A')

Reference Types (Store memory addresses, default = null)

- string → Text (e.g., "Hello")
- object → Can store any data type

2. Non-Primitive Data Types
These store complex data.
Arrays (int[] numbers = {1, 2, 3};)
Classes & Objects (class Car { })
Structs, Enums, Interfaces, Delegates

Key Differences: Value Type vs Reference Type

Feature	Value Type (int, float)	Reference Type (string, object)
Stores	Actual value	Memory address (reference)
Default Value	0	null
Nullable?	No (unless ? is used)	Yes

Type Casting

Type casting means converting a value from one data type to another. There are two types of type casting:

1. **Implicit Casting (Automatic Conversion)**
   - Happens automatically when converting a smaller data type to a larger one.
   - No data is lost.

   ```
   int i = 50;
   long j = i;  // Implicit casting from int to long
   ```

2. **Explicit Casting (Manual Conversion)**
   - Requires a cast operator or helper methods.
   - Used when converting a larger data type to a smaller one.
   - May result in data loss.

   ```
   float f = 50.5f;
    int i = (int)f; // Using cast operator (keyword)
    int j = Convert.ToInt32(f); // Using helper method
   ```

Boxing and Unboxing

Boxing and unboxing deal with converting between value types and reference types.

1. **Boxing (Value Type → Reference Type)**
   - Automatically converts a value type (like int) into an object.
     int i = 10;
     object o = i; // Boxing
2. **Unboxing (Reference Type → Value Type)**
   - Converts an object back into a value type.
   - Must be done explicitly.
     object o = 123;
     int i = (int)o; // Unboxing

Operators

Operators	**Category or name**
x.y, f(x), a[i], x?.y, x?[y], x++, x--, x!, new, typeof, checked, unchecked, default, nameof, delegate, sizeof, stackalloc, x->y	Primary
+x, -x, !x, ~x, ++x, --x, ^x, (T)x, await, &x, *x, true and false	Unary
x..y	Range
switch, with	switch and with expressions
x * y, x / y, x % y	Multiplicative
x + y, x – y	Additive
x << y, x >> y, x >>> y	Shift
x < y, x > y, x <= y, x >= y, is, as	Relational and type-testing
x == y, x != y	Equality
x & y	Booleanlogical AND or bitwise logical AND
x ^ y	Booleanlogical XOR or bitwise logical XOR
x | y	Booleanlogical OR or bitwise logical OR
x && y	Conditional AND
x || y	Conditional OR
x ?? y	Null-coalescing operator
c ? t : f	Conditional operator
x = y, x += y, x -= y, x *= y, x /= y, x %= y, x &= y, x |= y, x ^= y, x <<= y, x >>= y, x >>>= y, x ??= y, =>	Assignment and lambda declaration

Conditional Statements

if Statement

Runs a block of code only if a specific condition is true. If the condition is false, the block is skipped.

- **When to use**: Use if when you need to execute code based on a single condition.
- **Why use it**: It is the simplest way to make decisions in your code.

if-else Statement

Runs one block of code if the condition is true, and another block if the condition is false.

- **When to use**: Use if-else when you have two mutually exclusive outcomes.
- **Why use it**: It ensures that one of two blocks of code will always execute.

else if Statement

Adds additional conditions to check if the previous if or else if conditions are false.

- **When to use**: Use else if when you have multiple conditions to evaluate.
- **Why use it**: It allows you to handle multiple scenarios without nesting if statements.

Conditional (Ternary) Operator

A shorthand for if-else that assigns a value based on a condition.

- **When to use**: Use it for simple, single-line decisions.
- **Why use it**: It makes your code more concise and readable for straightforward conditions.

Switch Statements and Expressions

switch Statement

Compares a variable against multiple possible values and executes the block of code that matches.

- **When to use**: Use switch when you have many conditions to check against a single variable.
- **Why use it**: It is cleaner and more efficient than multiple else if statements.

Switch Expression

A more concise way to use switch for returning values based on conditions.

- **When to use**: Use it when you need to return a value based on a condition.
- **Why use it**: It is more compact and functional than a traditional switch statement.

Loops

for Loop

Repeats a block of code a specific number of times.

- **When to use**: Use for when you know exactly how many times you want to loop.
- **Why use it**: It is ideal for iterating over arrays, lists, or any fixed-range scenario.

while Loop

Repeats a block of code as long as a condition is true.

- **When to use**: Use while when the number of iterations is unknown.
- **Why use it**: It is useful for scenarios like reading input until a specific condition is met.

do-while Loop

Executes a block of code once, then repeats it as long as the condition is true.

- **When to use**: Use do-while when you want to ensure the loop runs at least once.
- **Why use it**: It is useful for menus or input validation.

foreach Loop

Iterates over each item in a collection (like an array or list).

- **When to use**: Use foreach when you need to process every item in a collection.
- **Why use it**: It is simpler and safer than manually managing indices.

Nested Loops

A loop inside another loop.

- **When to use**: Use nested loops for multi-dimensional data, like matrices or sorting algorithms.
- **Why use it**: It is essential for complex iterations, such as traversing 2D arrays.

When to Use What

- Use **if-else** for simple decisions.
- Use **switch** for multiple conditions against a single variable.
- Use **for** for fixed iterations.
- Use **while** or **do-while** for unknown iterations.
- Use **foreach** for collections.
- Use **nested loops** for multi-dimensional data or sorting.

Exception Handling

Exception handling helps us deal with errors in a program without crashing the application.

Structure of Exception Handling

1. **try block** – Code that might cause an error.
2. **catch block** – Handles the error if one occurs.
3. **finally block** – Runs no matter what (used for cleanup).

We can have infinite types of distinct catch blocks and the finally block always runs, whether there is an error or not.

Why Use Exception Handling?

Prevents the application from crashing.

Catches specific errors, like:

- DivideByZeroException (dividing by zero).
- FormatException (wrong data format).
- NullReferenceException (trying to use a null object).

Best Practices for Exception Handling

Use specific catch blocks to handle different errors.
Always inform the user about the error.
Use the finally block for cleanup, like closing files or database connections.
Don't overuse try-catch – Instead, **validate inputs** to prevent errors.

What is Throw

The **throw** keyword is used to manually trigger (raise) an exception.
Always check if the inner exception is null before passing it.
Custom exceptions help create **meaningful** error messages.
Use custom exceptions when built-in ones do not describe the issue well.

When to Use throw?

- To **rethrow an existing exception** (preserving details).
- To **create a new exception** when something goes wrong.
- To **handle inner exceptions** (track the original cause of an error).

Creating a Custom Exception

We can define our own exception by creating a class that **inherits** from Exception.

Steps to Create a Custom Exception

1. Create a class that extends Exception.
2. Call the base Exception constructor inside the class.
3. Optionally, include additional details in the exception.

Namespace

Collection of Classes , interfaces , Structs, Enums and Delegates
Using Statements on top of the code page indicate you are using those namespace's classes etc
We can create nested Namespaces
We can Also Alias the Namespace when needed

Class and object

A bundle of Code (Methods, variables, fields, properties, constructor etc) (Ref Link)
User Defined Reference Data Type
Is a blue print to objects
An instance of a class is called object
Class can be Static / Non-Static, if its Static then its methods should be static
When you declare the modifier static to method or field then the field will belong to the class rather than the instance of the class and instances will not have command over the static modified fields and methods.

Object syntax

Datatype variableName = new Constructor ();

Creates an instance of the class and we can access variableName.something()
Example
Dog Dog1 = new Dog ();
Dog Dog2 = new Dog ();
We have 2 different instances of Dog, we can do something like Dog1.Breed =x, Dog2.Breed = y;

Constructor

A special method in Class that has same name as class and has no return types, used to initialized data in the class.
Automatically called when the class is initiated.
Types of constructors (Default, Copy, parameterized, Static etc)?
Constructor Overloading is similar to that of Method overloading
Example
Dog Dog1 = new Dog (x);
Dog Dog2 = new Dog (x,y);

Method

Section of code to run a block of code whenever invoked / called. So that repetitive code can be eliminated
Syntax: Access Specifier RetunType MethodName (Parameters) {return variable of RetunType};
If Return type is Void, then no return value
Parament syntax = Datatype parameter name
All parameters combined are called Signature of the method Example: SomeMethod(int someInteger)
Variables declared inside the method are member variables only available in the scope of the method

Optional Parameter:

- Should be the last parameter in the signature
- Should have a default or custom value assigned to it
- We can have only 1 optional parameter
- So, if we do not pass value to that parameter then the assigned value is considered

Public void SomeMethod(int integer , double decimal , string = “someString”)
{// Does Something}

Params

Single Dimension Array of arguments / Parameters for a same method
Syntax

```
 Public double TotalCost (params double [] price)
{
Var Total = 0;
Foreach (var item in price)
{
Total +=price;
}
Return total;
}
```

i.e., is while calling this method we can pass as many parameters of similar data type to the method

Recursive Method

Advantage: Easier debugging, Easier to Code, Alternative to iteration
Disadvantages: Something slow and memory consumption is high,
Method calling itself over and over we will have break or return statement (to break cyclic loop)

OOPs

OOP(Object oriented Programming) helps in organizing code using real-world concepts like **Inheritance, Abstraction, Polymorphism, and Encapsulation**.

Inheritance

Inheritance allows code to be reused and organized efficiently.
For example, consider a **Vehicle** class as the parent. It has common properties like Speed, Year, Make, and Capacity. Child classes such as **Bicycle**, **Car**, **SUV**, and **Bus** inherit these properties while adding their own unique features.

Key Points:

- A **child class** inherits properties and methods from the **parent class**.
- The **parent class constructor** runs before the child class constructor.
- Use the **sealed** keyword to prevent a class from being inherited.
- The syntax for inheritance is:

Syntax: class ChildClass : ParentClass

Types of Inheritance:

1. **Single-Level** – One parent, one child.
2. **Multi-Level** – A child class acts as a parent to another class.
3. **Hierarchical** – One parent, multiple child classes.
4. **Multiple** – Not directly supported in C#, but achieved using interfaces.

Abstraction

Abstraction is used to define a blueprint for other classes. It helps hide implementation details while exposing only essential features.

Key Points:

- An **abstract class** is declared using the abstract keyword.
- It **cannot be instantiated** (objects cannot be created from it).
- It can contain both **regular methods (with implementation)** and **abstract methods (without a body)**.
- **Abstract methods** must be implemented in the child class using the override keyword.
- Every child class **must provide an implementation** for abstract methods.

Polymorphism

polymorphism allows a child class to provide a new version of a method that was defined in the parent class. This is a key feature of **polymorphism**, where the child class method overrides the parent class method.

Key Points:

- The parent class method must be marked with **virtual** or **abstract** to allow overriding.
- **virtual** means the method can have an implementation, but the child class can optionally override it.
- **abstract** means the method has no implementation in the parent class, and the child class **must** override it.
- In the child class, the **override** keyword is used to provide a new implementation of the method.
- If a method is overridden in the child class, the child class version **dominates** the parent class version.
- If no **override** is provided in the child class, the parent class version is called instead.

Encapsulation

Encapsulation is the concept of bundling related data and methods that operate on that data into a single unit, typically a class. This helps control how the data is accessed and modified.

Key Points:

- It involves hiding the internal details (fields and methods) of a class and providing public interfaces (properties and methods) to interact with the data.
- Access modifiers like **private**, **public**, and **protected** are used to control the visibility and accessibility of class members.

Benefits:

- Data protection: You can restrict direct access to fields and only expose them through properties or methods, ensuring the integrity of the data.
- Code maintainability: By grouping related data and operations together, it's easier to maintain and update the class.

Access Modifiers

Key / Accessibility	**Containing Class**		**Derived Class**		**Containing Assembly**		**Outside Assembly**	
	With Inheritance	**With object**	**With Inheritance**	**With object**	**With Inheritance**	**With object**	**With Inheritance**	**With object**
Private	Yes	Yes	No	No	No	No	No	No
Protected	Yes	Yes	Yes	No	Yes	No	Yes	No
Protected Internal	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Internal	Yes	Yes	Yes	Yes	Yes	Yes	No	No
Public	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Interface

An **interface** defines a contract or a set of requirements that a class must follow. It acts as a mediator between classes, promoting **loose coupling** and enabling better modularity, security, and multiple inheritance (through interfaces).

Key Points:

- **Interfaces define method signatures**, properties, delegates, events, but **not fields**. The implementation of these methods is provided by the class that implements the interface.
- An **interface** cannot be instantiated directly. Instead, objects of the **implementing class** are assigned to the interface reference.
- **Interfaces enable multiple inheritance**. A class can implement multiple interfaces.
- **Access modifiers**: By default, everything in an interface is **public**. You cannot specify access modifiers like private or protected.
- An **interface can inherit from other interfaces**, creating a chain of interfaces.

Explicit Interface Implementation:

- When a class implements multiple interfaces that have methods with the same signature, you can use **explicit interface implementation** to avoid conflicts. This way, methods are only accessible through the interface reference.
- **No access modifiers** (like public) should be used when explicitly implementing interface methods.

Properties

Properties are a way to provide controlled access to the fields of a class. They are a blend of **fields** (for storing data) and **methods** (for controlling access to data). Properties are part of **encapsulation**, as they allow you to control how data is accessed and modified.

Key Points:

- **Getters** (get) and **Setters** (set) are used to retrieve and assign values to the property, respectively.
    - The **getter** retrieves the value of the property.
    - The **setter** assigns a value to the property, and often includes logic for validation or manipulation before setting the value.
- **Auto-implemented properties** are simpler and do not require explicit backing fields. C# automatically provides the backing field behind the scenes.
- **Full properties** (also called **propfull**) provide a backing field, allowing you to implement custom logic for both the getter and setter.

When to Use:

- Use **auto-implemented properties** for simple properties where no validation or custom logic is required.
- Use **full properties** when you need to validate or manipulate the value before setting it, or when you want to implement custom logic for getting or setting the value.

Generics

Generics allow you to write code that works with any data type, without being tied to a specific one. You can define classes, methods, lists, and fields that accept any data type as a parameter. The type is specified when you call or instantiate the class or method.

Why Use Generics?

- Generics make your code more flexible and reusable. You do not need to write the same logic for different data types.
- They react based on the type you provide, making it easier to work with different kinds of data.

Collections

Collections in C# are **data structures** that hold multiple items. They provide a way to store, retrieve, and manage data in a more organized manner. In C#, collections are part of the **System.Collections** and **System.Collections.Generic** namespaces.
There are several types of collections in C#, each with its own features and use cases. (Ref Link)
Data Structures – is a named location that can be used to store data and organise data
To improve efficiency and memory management of code.
All the collections are ADT, that is they are all abstracted. So, we do not know the background implementation of it. Unless we read the documentation.

Commonly used Collections

Array

An **array** is a collection of elements of the same type, stored in contiguous memory locations. The key feature of an array is that it has a **fixed size** once created.
Advantages of Arrays:

- **Fast access**: You can quickly access elements using the index (e.g., array[0]).
- **Efficient storage**: Arrays are simple and use less memory.

Disadvantages of Arrays:

- **Fixed size**: Once created, the size of the array cannot be changed.
- **Difficult insertion/removal**: Adding or removing elements requires creating a new array and copying the elements.
-

Multi-Dimensional Array

Like a Grid, array of array or a rubix cube
Grid =2D and Rubix = 3D
Example of 2D
String [,] GridArray = {{honda, GTR}, {ford, Mustang}, {Audi, A4}}
Example for 3D
String [,,] Rubix Array ={{{Red,1}{Red,2}},{{Blue,1}{Blue,2}},{{green,1}{green,2}}}
Write Code to get further understanding

List

A **List** is a dynamic array that can grow and shrink in size as needed. It is part of the **System.Collections.Generic** namespace. Unlike arrays, lists allow you to add or remove elements during runtime.
Advantages of Lists:

- **Flexible size**: The size of a List can change dynamically as elements are added or removed.
- **Built-in methods**: Lists come with many useful methods, such as Add(), Insert(), Remove(), Sort(), and Clear().
- **Indexed access**: Like arrays, you can access elements by their index.

Disadvantages of Lists:

- **Performance**: Lists are not as fast as arrays when accessing elements, especially for large collections.

Common List Methods:

- Add(): Adds an element to the end of the list.
- Insert(): Inserts an element at a specified index.
- Remove(): Removes the first occurrence of an element.
- Sort(): Sorts the elements in the list.
- Reverse(): Reverses the order of elements.

Stacks

A **Stack** is a collection that follows the **Last-In, First-Out (LIFO)** principle. This means that the last element added to the stack will be the first one to be removed.

Advantages of Stacks:

- **Efficient for LIFO operations**: Stacks are perfect for situations where you need to access the most recently added elements first (e.g., undo operations).

Disadvantages of Stacks:

- **Limited access**: You can only access the top element directly.

Common Stack Methods:

- Push(): Adds an element to the top of the stack.
- Pop(): Removes and returns the top element from the stack.
- Peek(): Returns the top element without removing it.

Queues

A **Queue** is a collection that follows the **First-In, First-Out (FIFO)** principle. This means that the first element added to the queue will be the first one to be removed.

Advantages of Queues:

- **Efficient for FIFO operations**: Queues are ideal for managing tasks, such as print jobs or customer service lines.

Disadvantages of Queues:

- **Limited access**: You can only access the front and rear elements.

Common Queue Methods:

- Enqueue(): Adds an element to the end of the queue.
- Dequeue(): Removes and returns the first element from the queue.
- Peek(): Returns the first element without removing it.

Dictionaries

A **Dictionary** is a collection of **key-value pairs**. You can quickly retrieve values using keys, similar to looking up words in a dictionary.

Advantages of Dictionaries:

- **Fast lookups**: You can retrieve values in constant time using the key.
- **No duplicates**: Each key must be unique.

Disadvantages of Dictionaries:

- **Memory usage**: Dictionaries use more memory than lists because of the key-value structure.

Common Dictionary Methods:

- Add(): Adds a new key-value pair to the dictionary.
- Remove(): Removes a key-value pair from the dictionary.
- ContainsKey(): Checks if a specific key exists in the dictionary.
- TryGetValue(): Retrieves the value associated with a key without throwing an error if the key does not exist.

Linked Lists

A **Linked List** is a collection of nodes where each node contains **data** and a reference (or **link**) to the next node. Linked lists can grow and shrink dynamically, unlike arrays.

Advantages of Linked Lists:

- **Dynamic size**: Linked lists can grow or shrink in size without wasting memory.
- **Efficient insertion/deletion**: Inserting or removing elements is faster compared to arrays.

Disadvantages of Linked Lists:

- **Slower access**: Accessing elements is slower since you must traverse the list from the head to find the element.

Tuples

A **Tuple** is a simple data structure that can hold multiple values of different types. You can use it to group related data.

Advantages of Tuples:

- **Compact**: Useful for returning multiple values from a method.

Disadvantages of Tuples:

- **Readability**: Tuples may be less readable than using a custom class.

IEnumerable

IEnumerable is an interface in C# that allows iteration over a collection of items one at a time without storing all of them in memory. This makes it **efficient for handling large data sets**.

How **IEnumerable** Works

IEnumerable contains only **one method**:

- GetEnumerator() → Returns an **IEnumerator**

IEnumerator has **three important methods**:

1. **Current** → Returns the current item.
2. **MoveNext**() → Moves to the next item and returns true if there is one, false if there isn't.
3. **Reset**() → Moves back to the start (not commonly used).

Why we use **IEnumerable**

Lazy Evaluation: Items are **processed one at a time** when needed, instead of loading everything into memory. **Better Performance:** Suitable for **large datasets,** where storing all values at once is inefficient. **Forward-Only Access:** You can **only move forward**, not backward.

When to Use **IEnumerable**

- When **only reading** data (e.g., looping through a list).
- When **dealing with large collections** to avoid memory issues.
- When **forward-only iteration** is needed.

Enums

An **enum** (short for enumeration) is a special **value type** in C# that allows us to define a set of named constants. We use enums when values are **fixed** and **do not change** (e.g., days of the week, status codes, directions, etc.)
Example
enum daysofweek {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}
To access we use daysofweek.Moday etc

1. **Key Points**:
    - **Enum** is a value type that represents fixed constant values.
    - Default values start at 0 unless explicitly defined.
    - Casting is required when converting between enums and other types.
    - Enums are strongly typed and increase code readability.
    - [Flags] attribute allows enums to act as bit fields (useful for permissions).
    - Enums help make your code more readable, maintainable, and error-free

Coding Standards

Following consistent coding standards makes code **readable, maintainable, and efficient**. Here are some best practices

Variable name - camelCase
Class- PascalCase, use Noun
Constant variables – Pascal Case
Private variable use _camelCase
Interface prefix it with I, I Interface
Methods – PascalCase and verb
Properties - PascalCase
Proper namespace, close name
Vertically align the braces
Use **Ctrl +K** and **Ctrl +D** always for documentation
Follow SOLID , KISS , DRY and YAGNI

SOLID

S- Single Responsibility Principle
O- Open Close principle, open for extension, closed for modification
L -Liskov Substitution Principle, Child class should be substitutable for their parent class
I- Interface Segregation Do Not Force interfaces to force classes to implement them
D-Dependency inversion, components should depend on abstraction not on concretion

KISS ,YAGNI,DRY

KISS (Keep It Simple, Stupid**)** → Write clean and simple code.
DRY (Do not Repeat Yourself**)** → Avoid code duplication.
YAGNI (You Ain't Gonna Need It**)** → Do not add unnecessary features.

idea is to keep the code modular, simple. transparent such that readability, traceability, and handling of code is simple,

Unit Testing **(**More on it Later**)**

Moq and Nunit

Delegates

A **delegate** is a type-safe pointer to a method that allows you to call a method indirectly at runtime. You can think of it as a **container for functions**.

Key Points

- Used for **callbacks** (communication between two components).
- A **reference type** that holds the address of a method.
- Helps **decouple code** (avoids hardcoded logic).
- **Lambda expressions** are built on delegates.
- Signature of the delegate **must match** the method it points to.
- Acts like a **class**—you create an instance of a delegate and pass it around.
- Has similar Syntax as method but has as delegate keyword as modifier

Delegates and Event Handling

Delegates are **heavily used** in event handling to create loosely coupled applications.

Event handling works like this:

- A **publisher** (event sender) **raises an event**.
- A **subscriber** (event receiver) **listens to the event** and reacts.
- The **publisher does not need to know about subscribers**, making the system flexible.

Idea is to hook receivers to respective publisher, and publishers have no information on subscribers

Look at Delegate example they might be a bit complicated, but mastering them unlocks whole new things.
If Delegates do not make much sense focus on **Actions**, **Func** and **Predicate**.

Anonymous Methods

Anonymous methods allow you to create **inline** methods without explicitly defining a new function.

Multicast Delegate

A **multicast delegate** can hold multiple methods and execute them in order.
Key Points:

- Use += to **add** methods.
- Use -= to **remove** methods.
- If the delegate has a **return type**, it returns the result of the last method in the list.

The functions are invoked as per invocation List Order (that order is defined by in which order you register them)

Func, Action, Predicate

Predefined delegates to avoid creating custom delegate types and can have 0 to 16 parameters.
Action<T> (No Return Type)

- Always returns void.

Func<T, TResult> (Returns a Value)

- The last type is the return type.

Predicate<T> (Returns bool)

- Used when you need to check a condition and return **true** or **false**

LINQ is entirely based on Actions, Func and Predicates

Summary of Delegates

Concept	Description
Delegate	Type-safe pointer to a method.
Anonymous Methods	inline methods using delegate
Multicast Delegate	Delegate that calls multiple methods (+=, -=).
Events	Used for communication between objects (Publisher/Subscriber).
Func<T>	Predefined delegate with a return value.
Action<T>	Predefined delegate without a return value.
Predicate<T>	Predefined delegate with a bool return value.

Extension Methods

Create a Wrapper Class or Helper Class so that we can implement our own method and use it.

- The Class and Method should be Static
- The Method to which we are extending, that method's first parameter should begin with **this** keyword followed by **source data type**, then the receding parameters should be of desired type.
- Public static string Rightsubstring (this string ,int index)

Few Questions and Answers

Struct vs Class

Class	Struct
Class is a reference type	Struct is a value type
Class supports inheritance	Struct will not support inheritance (supports interface without implementation)
Class variables are stored on Heap	Struct variables are stored on Stack
Class can have destructor	Struct will not have destructor
Boxing a class object creates reference to same object	Boxing a strut object will create a copy in diff type
All the members of Class are private by default	All the members of struct are public by default
Class is well suited for Data hiding	Struts are not suited for Data hiding
When you copy a class into another class , we get copy of the reference variable , so making changes in either of the classes , effects the other	Copying a struct to another struct , creates a new copy of that struct , modifying either of the struct does not modify the other
Only the reference variable is destroyed , later GC Destroys the actual object	Immediately destroyed after scope of struct is lost
Can have parameter less constructor	Cannot have explicit parameter less constructor
Classes are not sealed by default	Structs are sealed types by default

In Short (Classes are Reference Type, have Inheritance, in Heap, can have destructor, can hide data)

Method Hiding vs Method Overriding vs Overloading

Overload = Same Method name but different Signature
Override = Overrides the parent's Method (virtual modifier in Parent and Override modifier in Child method)
Hiding = Child Class has same Name and Signature as Parent Method without any modifiers

Base Keyword and This Keyword

Base Keyword is used to call Base Class Methods or Constructors etc (base.parentMethod())
To call base class members we can we have 2 other ways

- While instantiation child class set data type to parent class
- Or Type cast the child to Parent class to access parent properties

This is used to refer the current reference of the object / instance

Static Vs instance

We can create
instance members (which belong to an object)
static members (which belong to a class and are shared across all objects).

What is an Instance?
When you **create an instance of a class**, a **new copy** of that class's members (fields, properties, and methods) is stored in memory.

What is Static?
When you **mark a class, method, field, or property as static**, it means **only one copy exists for the entire program**, and you **cannot create instances** of it.

Other Important Static Rules

1. Static Members Cannot Use **this**
    - Since this refers to a specific instance, and static members belong to the class (not an instance), this cannot be used inside a static method.
2. Static Classes Cannot Be Instantiated
    - If a class is **fully static**, you **cannot** create an object of it.
3. Static Constructors Are Called Before Instance Constructors
    - A static constructor is used to initialize static members and runs only once when the class is first used.

When to Use Static?

- When you need a **single, shared value** (e.g., Math.PI).
- When writing **utility/helper functions** that do not need object data (Math.Sqrt()).
- When creating **constants that do not change** (e.g., a database connection string

So Based on Functionality we make items as static or not
Access Modifiers cannot be applied on Constructor

Stack vs Heap

Physical Memory is divided into 2 parts stack and heap
Reference types are stored in Heap and Value types are stored in Stack
Reference variables(holds address of the actual object) are stored in stack and the actual object is inside heap
Destruction of variables

- When out of scope the stack data and reference variable are automatically destroyed
- The actual object is destroyed by Garbage collector (when GC looks at heap and sees that there is no pointer to the allocated memory , then it destroys it)

When does GC work

When memory is critically low , or when we explicitly call it or when GC thinks it right time.

Type vs Type Members

Type = Classes , Interfaces ,Delegates , Structs and Enums (internal or Public Access Modifiers only)
Type Members = Properties , Fields , Methods , Constructors etc (all 5 Access Modifiers applicable)

Attributes / Decorators

Allows you to add declarative information to your programs , and this information can be queried at runtime using reflection (Can be applied to types(classes , delegates etc) and type members(methods ,properties, fields etc)).
We can create out custom attributes or use pre-defined attributes.
Attributes are also classes
Examples :
[Obsolete]- Indicates that its outdated
[Serializable]- Indicates that the class can be serialized
[Web Method] – exposes a method as XML web service

There are many more attributes(pre-defined) , base for them is in System.Attributes namespace

Records

(Basic Idea **)**

- New Type of Class (Instead of creating class , then creating constructor and passing parameters to that constructor then setting properties values of that based on parameters we can use records)
- Open the curly braces and it behaves like class
- can use it like a simple data structure

is , as Keyword (Both used for type casting **)**

- is keyword during run time checks if the data types are same and returns bool
- as keyword is used for casting if casting fails returns null

POCO

Plain old CLR objects , just a class with bunch of properties

Yield Keyword

instead of creating a new List , and add items to list ,we can use Yield return (collective return)

- This works on any collection that returns IEnumerable

Syntax :

```
//List<HeroRecord> listofHerosThatCanFly = new List<HeroRecord>();
foreach (HeroRecord item in listofHeros)
{
  if (heroListDelegate(item))
  {
    yield return item;
     //listofHerosThatCanFly.add(item)
  }
}
// return listofHerosThatCanFly;
```

Closures

- Occurs when we are writing higher order functions
    - Functions that return another function are call higher order function
- While returning a function that function might have a variable (that is short lived because it is on stack), so during run time we run into an exception , cause that variable is destroyed as its out of its scope
- So, to tackle this C# binds the lifetime of the variable to the to the function by capturing the variable (so the variable is promoted from stack to heap) – in background it's like creating a class that has a field with stack variable and a method which returns the function, hence promoted to heap.

C# version History

(All new stuff that have been getting added in each iteration **)**

Top Level Statements in C# 9 and above

- Writing Program without Main Statements

What is Dynamic

(To complicated and out of scope for now **)**

- Use Dynamic to change the data type , at runtime you can do this
- Is a bridge between Statically typed language (C#) and Dynamically Typed Language (and only use this if you have to do this)
- EdgeJs Library is one way to day it
- COM (Component Object Model) and OLE (Object linking and embedding) (Reason that this exists)

Why do we Override ToString() Method

- When we call ToString() Method we actually base class to string (ie System.Object). This generates namespace.Class as output but in most cases, we would like to get a meaning full string so , we override it as per our requirement

Why do we Override Equals() Method

- For value type it works as intended
- For reference type (While checking equality we have 2 things
    - 1) value equality
    - 2) reference equality i.e., object reference variables pointing to same object(same references))
- == is reference equality
- .Equals() is value equality comparator
- Good Practice to override the Hashcode when overriding Equals()

Convert.ToString() vs .ToString()

- ToString gives string representation and comes from system.Object base class
- Convert.ToString() ,converts null to String.Empty
- So .ToString throws an exception when object is null
- Both will return same value , and performance wise no difference

String Builder vs System.String

String Builders are mutable and System.String are immutable , for string manipulation it better to use string builder

- Immutable means unchangeable once created , and mutable is vice versa.

So, on System.String every time you make an operation like concat etc a new object is created and the reference variable is pointing to that new object and previous objects are still present in heap as orphaned objects as they are not pointing to any reference variable (Memory constraint).
To use string builder, we create and object of string builder and use that object (Example for concat we use .Append() method).

System.String is in System namespace
StringBuilder is in System.Text namespace

Class vs Record vs Struct vs Record Struct

	Class	**Record**	**Struct**	**Record Struct**
Inheritance	Interface, Class	Interface ,Record	Interface	Interface
Memory Allocation	Heap	Heap	Stack	Stack
Methods	All of them can implement Methods			
Mutability	Mutable	Immutable	Immutable	Immutable
Equality Operator	Compares using reference equality	Value equality by default		
Use Case	Complex Objects are better handled here	Ideal for data structures when value equality is important	Small light weight objects residing on stack	Ideal for data structures when they benefit from staying on stack
Constructor	Multiple constructors			

Squiggly Lines while coding

Red squiggly line is Error
Green squiggly is warning

Partial Classes

Use Partial modifier to make the class as partial **(**same properties can be applied to structs ,interfaces**)**
What does that mean :

- These two or more partial classes combine to form a normal class and standard rules apply (if either is marked abstract or sealed entire combined class has that modifier applied).

Why use partial class :

- if we want to split a class(into multiple physical files) functionality we can achieve this by partial class.

Syntax : partial ClassA and partial class A (in different file).(Both should have same access specifier)

Example when we have lost of auto generated code mark that as partial so that autogenerated code is in one physical file and our manual code in separate code

Partial Methods

Similar to that of partial class

- In one partial class we have definition and in another we have implementation
    - So, if we do not have an implementation , the compiler removes all the calls to that method.
- Are private by default and cannot have virtual ,abstract , override ,new , sealed and extern modifiers
- Return Type must be only be void
- If we implement both declaration and implementation at same time then u get compilation error

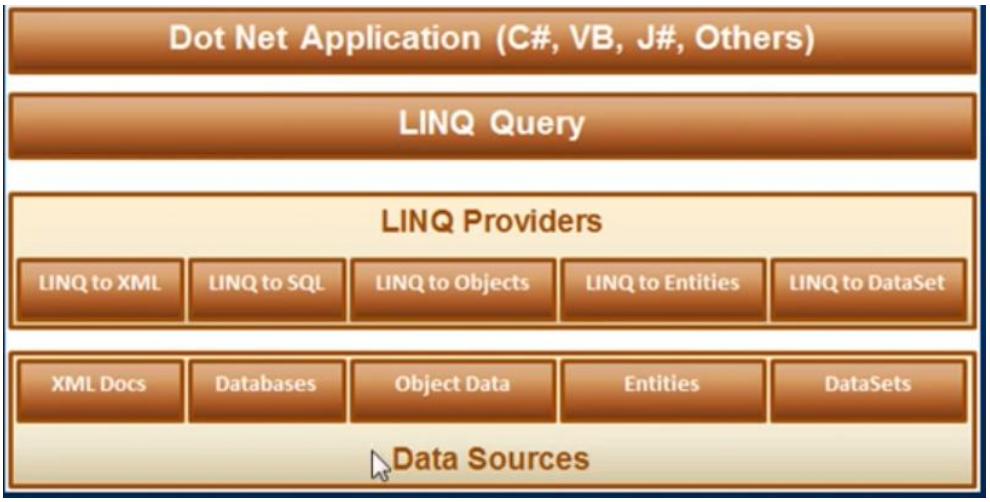
LINQ

LINQ Stands for Language integrated Query, it enables us to run queries on any type of Data Store (SQL Server, Xml Docs, Objects in Memory etc) (Ref Link)

Why Use LINQ?

- **Concise and Readable:** LINQ expressions are more readable compared to traditional loops and conditions.
- **Type Safety:** LINQ queries are checked at compile-time.
- **IntelliSense Support:** Visual Studio provides IntelliSense for LINQ queries.
- **Uniform Data Access:** LINQ provides a common query interface for different data sources like objects, databases (LINQ to SQL), XML (LINQ to XML), and more. We need know have Syntax Specific to the knowledge Domain
- **Deferred Execution:** Queries are executed only when they are iterated over, optimizing performance.

Architecture



LINQ Provider translates the LINQ Query w.r.t to Data Source (Such that its native to Data Source)

We can Write LINQ queries in 2 Ways (no difference in performance)

- Lambda Expressions
- SQL like Query

Should work on all IEnumerable <T> interface

Operators in LINQ (also known as LINQ Extended Methods) (Ref Link)

LINQ Operators

Aggregate Functions

```
int [] array = {1,2,3,4,5,6,7,8,9,10}
result = array.Min();
result = array.Max();
result = array.where(x=>x%2 == 0).Min();
result = array.Count();
result = array.Sum();
```

Aggregate Function itself

```
Result = array.Aggregate((a,b)=> a*b);
Result = stringarray.Aggregate((a,b)=> a + “,” =b);
Result = array.Aggregrate(10,(a,b)=> a*b); // Seeded Method for Multiplication
```

Restriction Operator

Where (valid Lambda Expression) is an extension Method used for filtering the data
Result = array.Where(x=x%2==0);

Projection Operator

Select () and SelectMany () are extension methods for transforming the result data and perform operations on them

Select ()

- We create New Anonymous data type by using the below methods
- Select ((x) => new {a = x. Something b=,x.someValue, c=x.someothervalue}
- //new{ **x** } =*new Anonymous Type*

SelectMany ()

- Let us assume we have a List of string property for few objects, using SelectMany we can flatten the list property of all objects into a single sequence.
- SelectMany(x=>x.SomeList);
- Use. Distinct () to get distinct values

Ordering / Sorting Operators

- OrderBy, OrderbyDesc = For Primary Sorting
- ThenBy, ThenByDesc = For Secondary Sorting
- Reverse () = Simple Reverse

Partition Operator

- Take, TakeWhile Pass a value in parameter and only the specified values range is retrieved
- Skip, SkipWhile Pass a value in parameter skips those values and get remaining
  In While we pass a condition

Conversion Operator

- ToList()- To List, ToArray()-To an Array , ToDictionary()-Unique Key Value Pairs, ToLookup() – Similar to that of Dictionary but can have similar values
- **Cast** will convert all elements in the list to specified data type , in case of inconsistencies will throw a Cast exception
- **ofType** will only get all elements of specified data type
- **AsEnumberable** runs get all data from SQL server and run the remaining operations on client side.
- **AsQueryable** runs on server side with all filters and returns data

Grouping Operator

- Takes a flat sequence data or flattens the data and organise them into groups and returns a dictionary
- Result = array.Groupby(valid lambda function);
- Then cw the result. Key and result. Count()
- And result can be filtered by adding another expression in count method or using other aggerate functions
- If you want to Group by Multiple Keys then create an anonymous type

Element Operators

- Retrieve a single element from sequence using element index or a condition , all these methods have a corresponding overloaded methods that accepts a predicate (returns bool takes DS as parameter)
- First / FirstorDefault – Get First Value
- Last / LastorDefault – Get Last Value
- ElementAt/ElementAtorDefault – At value at that index
- Single/SingleOrDefault – Use only when u expect only 1 element
- DefaultifEmpty – Returns a sequence , useful to check if empty or not , overloaded method takes a default value
- i.e., the first , Last ,ElementAt and Single will return respective values ,but if the dataset is empty then it will throw an invalid operation exception. and in the overloaded methods ,the methods accept a condition/ predicate like (x=>x %2 ==0) etc, if the condition is not satisfied then it throws invalid operation exception (with a message like “sequence doesn’t contain any matching elements ”).
- To Overcome this exception, we use methods with Default (default returns default value of that datatype on which we are performing operations)

Join Operators

- Group Join – useful for getting hierarchical data
    - .GroupJoin(on what you want to group , inner key selector , outer key selector , result)
- Inner Join – To obtain common / Matching elements from result and exclude unwanted results
    - .Join(on what you want to group , inner key selector , outer key selector , result)
- Left join and Right Join -attained by GroupJoin and SelectMany
- Cross join – Produces cartesian product inner key and outer key using SelectMany
- Outer join ?

General Operator

- Range – Enumerable.Range (start , count)
- Repeat – Enumerable.Repeat (Value , count)
- Empty – Enumerable.Empty<dataType>() – To return an empty sequence

Set Operator

- Distinct – Get Unique value, useful for handling single sequence , has overload for complex sequences implement IEquality comparator (Equality Method and get Hashcode method)
- Union – Combine to sequences , excludes duplicates , for complex sequences use anonymous type or implement IEquality comparator , anonymous type is simpler
- Intersect – Return common elements between
- Except – Get elements present in first sequence and exclude common elements in both sequences

Concatenation Operator

- Concat used to club multiple sequences to a single sequence , similar to that of union but doesn't exclude common elements

Sequence Operator

- SequenceEquall To Check if 2 sequences are equal by Pair wise and returns bool ,has over load
    - should have same number of elements in both sequences
    - same values are to be present (can use order by to sort them)

Quantifier Operator

- All – pass a condition to check if all elements sequence is satisfied
- Any – pass a condition to check any element in sequence is satisfied , has overload
- Contains - Returns bool ,has overload

To Master LINQ , use **LINQ Pad** to Test and work with LINQ

Lambda Expressions

- => is called as **go to** operator ,
- Most useful in writing LINQ query
- Automatically infers parameter data type
- Supersedes the Anonymous method all the time
    - A case where Anonymous method can replace lambda is
    - Item.Click += delegate { // does some action};
    - Item.Click +=(eventSender , eventArgs) = > { //some action };
- Actions ,Functions and Predicates are easy to read using lambda functions

Classification of Operators (Query Execution)	
Deferred or Lazy Operators – Uses differed Execution	**Immediate or Greedy Operators -**Uses Lazy Execution
Differed Execution = Not Executed at place where its declared , to make it as Immediate use Immediate Operators	Lazy Execution = Executed at the place where its declared
Select , Were , Skip ,	Count ,Average , Min, Max ,ToList

Reflection Framework

Using reflection, we get the ability to inspect an assembly's metadata at runtime.
Use Cases

- We use reflection to get the properties, Method etc through System.Type helpful for knowing objects in depth, and these objects may not be available during compile time.
- Useful for implementing generic methods, From System.Type we can get the properties, methods etc of the object at runtime.
- **Late Binding** is achieved through Reflection* (Binding at Runtime , Binding at compile time is Early Binding)
- Late Binding is complicated , slow , risk of runtime exceptions , always use Early Binding
- While using WinForms or asp pages the properties window uses reflection to show all properties of the UI Element

Every Type Directly or Indirectly inherits from System.Object , so while using Reflection and getting methos we might see following methods additionally

- GetType
- ToString
- GetHashcode
- Equalls

Syntax

- Type T = Type.GetType("FullyQuallifiedName")
- Type T = typeof(ClassName);
- Type T = Object.GetType();

Understanding Async Programming and Parallel Programming

To Get started with this we need to understand some fundamentals and definitions

Threads and Processes

When you build an application, it is stored as an executable file on disk. When you run the application, the operating system loads it into RAM and manages its execution. The application runs as a **process**, which includes allocated memory, system resources, and threads necessary for execution. Multiple processes can run simultaneously on a system, and the operating system handles their scheduling and resource allocation. (Ref Link)

What is a Process

A **process** is an instance of a program that the operating system manages and executes. It consists of allocated resources such as memory, security context, and CPU time.

- The **operating system** facilitates process execution by providing necessary resources (memory, security context, file handles, etc.).
- The **OS scheduler** distributes processes across available CPU cores for efficient execution.
- A process has at least **one thread**, known as the **main (or primary) thread**. In GUI applications, this is typically the **UI thread**.
- Each **process has a unique Process ID (PID)** and runs in an **isolated memory space**, ensuring that processes do not interfere with each other.

What is a Thread

A **thread** is the smallest unit of execution within a process. A single process can have multiple threads that share the same memory and resources.

- **Threads exist within a process**, meaning a process must have at least one thread.
- **Threads are lightweight in terms of execution overhead** but can be memory-intensive when created in large numbers.
- **Creating too many threads** can lead to memory bottlenecks and degrade performance.
- By default, the **number of threads available** is equal to the **number of CPU cores**.
- **Threads are managed in .NET via** System.Threading.

Advantages

Responsive Applications: Keeps the UI responsive by offloading work to background threads.
Efficient Resource Use: Utilizes CPU efficiently during I/O-bound operations (e.g., waiting for file or network operations).
Parallel Processing: Splits large tasks into smaller units for concurrent execution.

Disadvantage

Performance Overhead: On single-core machines, threading can hurt performance due to context switching (time taken to switch between threads).
Complexity: Requires more code to achieve the same task compared to single-threaded execution.
Debugging and maintaining threaded code is harder due to potential race conditions and deadlocks.

Use threading when the **advantages (responsiveness, efficiency, parallelism)** outweigh the **disadvantages (complexity, overhead)**.

ThreadStartDelegate

A delegate or lambda function that defines the **entry point** for the thread.
Background vs Foreground Threads:

- **Background Threads**: Automatically terminate when the main thread ends.
- **Foreground Threads**: Keep the application alive until they complete.

ParameterizedThreadStart Delegate

Allows passing a parameter to the thread's entry point method. And useful when the thread's method requires input parameters.

- Use a **helper class** or **global variable** to pass strongly-typed parameters.
- Avoid ParameterizedThreadStart if possible and use ThreadStart with a lambda or delegate.

Thread Pool

A **thread pool** is a collection of pre-created threads that are managed by the system.
Instead of creating and destroying threads manually, you can assign tasks (work items) to threads from the pool.
The system handles the lifecycle of these threads, including scheduling and monitoring.

How Does It Work

Pre-created Threads: The thread pool creates a set of threads in advance but does not start them immediately.
Task Assignment: You add tasks (work items) to the thread pool, and it assigns them to available threads.
Thread Reuse: Once a thread completes its task, it returns to the pool and becomes available for new tasks.
Thread Limits: You can configure the **minimum** and **maximum** number of threads in the pool to control resource usage.

Advantages of Using a Thread Pool

Simplified Thread Management:

- You do not need to worry about creating, scheduling, or monitoring threads.
- The system handles the lifecycle of threads automatically.

Resource Efficiency:

- Reusing threads reduces the overhead of creating and destroying threads repeatedly.
- Prevents overloading the processor by limiting the number of concurrent threads.

Scalability:

- The thread pool dynamically adjusts the number of threads based on the workload.

Disadvantages of Using a Thread Pool

Loss of Control:
You cannot directly control individual threads (e.g., setting priority or aborting a thread).

Limited Customization

Thread pools are designed for general-purpose tasks and may not be suitable for highly specialized scenarios.

Overhead for Short Tasks

For very short tasks, the overhead of managing the thread pool might outweigh the benefits.

When to Use a Thread Pool

Use a thread pool for **simple applications** or tasks that are:

- Short-lived.
- Independent of each other.
- Not highly resource-intensive.

Avoid using a thread pool for tasks that require fine-grained control over threads.

Thread Pool Configuration

Default Thread Count

The default number of threads in the thread pool is typically equal to the **number of CPU cores**.

Maximum Thread Limit

- The maximum number of threads in the thread pool depends on the system and framework.
- In .NET, for example, the maximum number of threads is quite high (e.g., thousands), but it's limited by available memory and system resources.
- You can configure the maximum limit programmatically, but exceeding the system's capacity can lead to performance degradation or out-of-memory errors.

Best Practices

Avoid Indefinite Thread Creation

- Creating too many threads can exhaust memory and degrade performance.
- Use the thread pool to limit the number of concurrent threads.

Monitor Thread Usage

- Keep an eye on the number of active threads to ensure optimal performance.

Use for Appropriate Tasks

- Thread pools are ideal for tasks like handling small I/O operations, background tasks, or parallel processing.

Key Points to Remember

- A thread pool is a **collection of pre-created threads** managed by the system.
- It simplifies thread management by handling **scheduling, monitoring, and reuse**.
- You can control the **minimum and maximum number of threads** to optimize resource usage.
- The default number of threads is equal to the **number of CPU cores**.
- The maximum number of threads is limited by **system memory and resources**.
- Use thread pools for **simple, short-lived tasks** and avoid them for tasks requiring fine-grained thread control.

Retrieving Data from Thread

Delegates can be used to retrieve data from a thread by implementing a callback function.

How it works:

- Define a delegate that matches the signature of the callback method.
- Pass the callback method to the thread.
- The thread invokes the callback method to return data once it completes its task.

Thread.IsAlive

Checks if the thread is still running or has terminated
Returns: true if the thread is alive; false if it has completed or terminated.

Thread.Priority

Sets or gets the scheduling priority of the thread.

- Priority Levels:
  - ThreadPriority.Low
  - ThreadPriority.BelowNormal
  - ThreadPriority.Normal (default)
  - ThreadPriority.AboveNormal
  - ThreadPriority.High

Thread.Join

Blocks the calling thread until the thread with the Join method completes.

- Useful when the output of one thread depends on the completion of another.

Overloads:

- Without timeout: Waits indefinitely for the thread to complete.

With timeout: Waits for a specified time (in milliseconds) and returns true if the thread completes within the timeout; otherwise, false.

Key Points

- Use delegates to retrieve data from threads via callback functions.
- Use Thread.IsAlive to check if a thread is still running.
- Adjust Thread.Priority to control the scheduling priority of threads.
- Use Thread.Join to synchronize threads when their outputs are dependent on each other.

Protecting Shared Resources

When multiple threads access shared resources, **inconsistency** can occur if the resources are not protected. To ensure **thread safety**, use synchronization mechanisms.

Synchronization Mechanisms

Interlocked Class

Provides atomic operations for simple variables (e.g., incrementing a counter).

- **Advantages**: Better performance than locks for simple operations.
- **Limitations**: Limited to basic operations like increment, decrement, and exchange.

Lock Statement

Ensures only one thread can access a code block at a time.

- Simple and effective for most synchronization needs. Use Case: Simple and effective for most synchronization needs.

Monitor Class

Provides more control over locking than the lock statement.

- **Features**:
    - **Enter and Exit**: Acquire and release locks.
    - **TryEnter**: Attempts to acquire a lock with a timeout.
    - **Wait**: Releases the lock and waits for a signal.
    - **Pulse and PulseAll**: Signal waiting threads.
    - Use Case: When you need advanced synchronization features.

Mutex

A synchronization primitive that allows only one thread to access a resource at a time.

- For cross-process synchronization.

Semaphore

What it does: Limits the number of threads that can access a resource simultaneously.

- When you need to control access to a resource for a limited number of threads.

ManualResetEvent and AutoResetEvent

ManualResetEvent

Acts like a gate that stays open once signalled.

- **Methods**:
  - **Set**: Opens the gate, allowing all waiting threads to proceed.
  - **Reset**: Closes the gate.
  - **WaitOne**: Blocks the thread until the gate is opened.
- **Use Case**: When you want to signal multiple threads at once.

AutoResetEvent

Acts like a gate that opens for one thread at a time.

- **Methods**:
  - **Set**: Opens the gate for one waiting thread.
  - **WaitOne**: Blocks the thread until the gate is opened.
- **Use Case**: When you want to signal one thread at a time.

Example Scenario

File Reader and Writer:

- Use ManualResetEvent or AutoResetEvent to ensure only one thread (reader or writer) accesses the file at a time.

Deadlock Resolution

Deadlocks occur when threads wait indefinitely for resources held by each other.

- **Prevention**:
  - **Lock Ordering**: Always acquire locks in a consistent order.
  - **Timeout**: Use Monitor.TryEnter or Mutex.WaitOne with a timeout to avoid indefinite waiting.
  - **Avoid Nested Locks**: Minimize locking multiple resources simultaneously.

Performance Considerations

- **Processor Count**: Use Environment.ProcessorCount to determine the number of CPU cores.
- **Parallelization**:
  - On a **single-core** machine, threads share the CPU, leading to context switching overhead.
  - On a **multi-core** machine, threads can run in parallel, improving performance.

Summary

- Use **Interlocked** for simple atomic operations.
- Use **Lock** for basic synchronization.
- Use **Monitor** for advanced synchronization features.
- Use **Mutex** for cross-process synchronization.
- Use **Semaphore** to limit concurrent access to a resource.
- Use **ManualResetEvent** and **AutoResetEvent** for thread signaling.
- Prevent **deadlocks** by following consistent locking practices.

Tread vs Task

Thread: A low-level way to run code in parallel. It directly represents an operating system thread.
Task: A higher-level abstraction for running code asynchronously or in parallel. It uses threads from the thread pool but simplifies the process.

Feature	Thread	Task
Level of Abstraction	Low-level (directly represents an OS thread).You have to manage the thread lifecycle (starting, stopping, etc.).	High-level (abstracts thread management).The system manages the underlying threads for you.
Resource Usage	Consumes significant resources (memory, CPU).Creating too many threads can overload the system.	Uses threads from the thread pool, making it more efficient. This makes tasks more efficient for short-lived operations.
Control	Full control over thread lifecycle (start, stop, priority, etc.).	Limited control over the underlying thread, but easier to use.
Use Cases	Best for long-running operations requiring fine-grained control.	Best for short-lived, asynchronous operations or parallel tasks.
Return Values	Does not return a value.	Can return a result and provide status (e.g., completed, cancelled).
Cancellation	Cancelling a thread is complex and not recommended.	Supports easy cancellation using Cancellation Token.
Multiple Operations	A thread can only run one task at a time.	A task can represent multiple operations or be part of a larger workflow.
Thread Pool Integration	Not tied to the thread pool. Threads are created and managed manually.	Automatically uses threads from the thread pool. Threads are reused.
Example Scenarios	- Long-running background processes. - Complex operations requiring fine-tuning.	- Fetching data asynchronously. - Running multiple small tasks in parallel.

When to Use What

Use Thread when

- You need full control over the thread.
- You are dealing with long-running operations.

Use Task when

- You want to run short-lived, asynchronous operations.
- You need to return a result or check the status of the operation.
- You want to cancel operations easily.
- You are working with the thread pool for better resource management.

Async Programming and Sync Programming

Sync Programming

Is working on single thread so every task is performed one after another (so at a time only 1 task is executed) , Example Blocked UI

- Here the thread is locked for quite some duration (based on task)
- So, if new requests are added on top, new threads would be created as CPU is still busy with the current thread, new threads consume loads of memory

Async Programming

(To overcome problems in Sync Programming * on Non-CPU Bound operations**)**

- We use / reuse threads efficiently by the usage of (async and await stuff)
    - Threads are not locked as in Sync Programming (so after execution the threads are moved back to the thread pool) and new threads are not created , saves a lot of memory and CPU load.
- Predefined Methods / Custom Async Methods end with Async suffix
- Tasks are foundations for Async programming
- **Task Based Pattern ***
- **Event Based Pattern ***

Async and Await

- Using Async Keyword turns a method into Asynchronous method
- Await keyword can be used only inside method with Async modifier
- Await suspends the calling method until the task is completed then , returns the control to caller
- Async wraps the return into a task implicitly and await unwraps the task implicitly
- Return Type of Async Method is either void or Task

- Use **Async** keyword on methods to mark the method as asynchronous method
- Use **Await** key word on the operation which is a task (when code reached the point, we wait for the task to be finished , once finished then we move ahead) – an async method cannot move past await until task is done
- Also, when calling this method use await Keyword
- Benefit of this is that , this way we do not block the thread.
- Also, while using Async the return is wrapped automatically around task , to unwarp it use await

Works like magic with Entity Framework

Parallel Programming

Parallel programming is about dividing a task into smaller units and executing them simultaneously across multiple CPU cores.

It is not the same as **asynchronous programming**:

- Parallel Programming: Requires multiple CPU cores to execute tasks simultaneously.
- Asynchronous Programming: Can run on a single CPU core by freeing up threads during waiting periods (e.g., I/O operations).

When to Use Parallel Programming

Use parallel programming for CPU-intensive tasks such as:

- Video processing.
- Complex algorithms.
- Data processing or transformations.
- It is not typically needed for web servers because the scheduler automatically distributes requests across threads and CPU cores.

Key Concepts in Parallel Programming

1. Data Parallelism

- What it is: Splitting a large dataset into smaller chunks and processing them in parallel.
- How it works: Each chunk is processed by a separate thread running on a different CPU core.
- Tools in .NET:
    - Parallel.For: Executes a loop in parallel.
    - Parallel.ForEach: Iterates over a collection in parallel.

2. Task Parallelism

- What it is: Executing multiple tasks (methods or functions) in parallel.
- How it works: Each task runs on a separate thread.
- Tools in .NET:
    - Parallel.Invoke: Executes multiple actions in parallel.

Parallel.For and Parallel.ForEach

Parallel.For: Replaces a standard for loop with a parallel version.
Parallel.ForEach: Replaces a standard foreach loop with a parallel version.

Key Features

Non-Sequential Execution: Unlike regular loops, these methods run iterations on multiple threads, so the order of execution is not guaranteed.

Use the **ParallelOptions** class to set the **MaxDegreeOfParallelism** (maximum number of threads to use).To have control over Parallelism

Example:

```
var options = new ParallelOptions { MaxDegreeOfParallelism = 4 };
Parallel.For(0, 100, options, i => { /* Your code */ });
```

Breaking Out of Loops

Use ParallelLoopState.Break() to stop further iterations after the current one.
Use ParallelLoopState.Stop() to terminate the loop immediately.

Example

```
Parallel.For(0, 10, i =>
{
   Console.WriteLine($"Iteration {i} on thread {Thread.CurrentThread.ManagedThreadId}");
});
```

Task Parallel Library (TPL)

A set of APIs in the System.Threading and System.Threading.Tasks namespaces that simplifies parallelism and concurrency by abstracting low-level thread management.

Key Features

- Encapsulates Multi-Core Execution: Automatically distributes tasks across CPU cores.
- High-Level Abstractions: Provides easy-to-use methods like Parallel.For, Parallel.ForEach, and Parallel.Invoke.

Parallel.Invoke

Executes multiple actions (methods or delegates) in parallel and each action runs on a separate thread.

Syntax :

```
Parallel.Invoke(
   () => Method1(),
   () => Method2(),
   () => Method3()
);
```

Control Over Parallelism

- You can pass ParallelOptions to limit the number of threads used:

```
var options = new ParallelOptions { MaxDegreeOfParallelism = 2 };
Parallel.Invoke(options, Method1, Method2, Method3);
```

Parallel LINQ (PLINQ)

An extension of LINQ (Language Integrated Query) that enables parallel execution of queries.

Add .AsParallel() to your LINQ query to enable parallelism.
Use .WithDegreeOfParallelism() to limit the number of threads used.
Example

```
var result = dataSource
   .AsParallel()
   .WithDegreeOfParallelism(4)
   .Where(item => item > 10)
   .Select(item => item * 2);
```

Best Practices for Parallel Programming

Avoid Over-Parallelization

- Creating too many threads can lead to thread contention and performance degradation.
- Use MaxDegreeOfParallelism to limit the number of threads.

Use for CPU-Intensive Tasks

- Parallel programming is ideal for tasks that require heavy computation.
- Avoid using it for I/O-bound tasks (use asynchronous programming instead).

Ensure Thread Safety

- Use synchronization mechanisms like locks or concurrent collections to avoid race conditions.

Test and Measure

- Parallel programming can introduce complexity and overhead. Always test and measure performance to ensure it provides a benefit.

Summary

- Parallel Programming is about executing tasks simultaneously across multiple CPU cores.
- Use Parallel.For, Parallel.ForEach, and Parallel.Invoke for data and task parallelism.
- The Task Parallel Library (TPL) simplifies parallelism by abstracting low-level thread management.
- PLINQ enables parallel execution of LINQ queries.
- Always control the degree of parallelism and ensure thread safety for optimal performance.

Data Structures with C#

Big O notation

Describes the performance of Algorithm

O (1) Constant Time, no matter the Data Size, Example: Random access of element
O(N) – Linear Time, as Data Increases the time also increases: Binary Search
O (Log N)- Logorrheic Time, increasingly less time: Searching / Looping through Array / List
O (N Log N)- Quasilinear Time, Linear up to a point: Sorting (Heap Sort, Merge Sort, Quicksort)
O(N^2)- Quadratic Time, Highly Time Consuming: Insertion Sort, Bubble Sort and Selection Sort
O (N!)- Factorial Time – Unsolvable, Travelling Sales man problem

Harvard CS 50 (Broader overview of CS , DS and Algorithms) (Ref Link)

Searching Algorithm

For in depth Search algorithms, Basics below (Ref Link)

Linear Search

Iterates through 1 element at a time of a collection
Complexity: O(N)
Disadvantage: Slow for larger Data Sets
Advantages: Fast for Small and Medium Data Sets, does not need Sorting, use full for Structures that do not have random access
Example: Scanning an array for a value in For Loop

Binary Search

Complexity: O (Log N)
Search Algorithm that finds the position of the target value with in a *sorted Array
Half of Array is eliminated in each step
Pretty Good with large Data Sets
Example: High = Array.Length-1
Low = 0;
Middle = Low +(High-low)/2;
In a for Loop ()
Value = array[middle];
If (value > Target Value)
{
low = middle+1;
Middle = Low +(High-low)/2
}
Else if (value <target Value)
{
high = middle-1;
Middle = Low +(High-low)/2
}
If (value = =target)
{
Cw(output);
Break;
}

Interpolation Search

Best Case: O (Log (Log N))
Worst Case: O(N)
Improvement / Extension of Binary Search
We calculate a guess where the Target value might be and search if the calculated guess is false were calculate the guess.
Best for Uniformly Distributed values
Instead of discarding the elements of array in Half like Binary Search we discard the array based on guess value non uniformly
Int probe = Low + ((High -Low) *(value – array[low])/array[high]-array[low]) Replace middle with probe in above algorithm

Sorting Algorithm

Bubble Sort

Complexity O (N^2)
Logic: Pick the starting element if the next element is smaller than, swap their respective places (Water example)
Algorithm

```
for (int i = 0; i < tempArray.Length; i++)
        {
           for (int j = 0; j < tempArray.Length - i - 1; j++)
           {
              if (tempArray[j] > tempArray[j + 1])
              {
                 var temp = tempArray[j];
                 tempArray[j] = tempArray[j + 1];
                 tempArray[j + 1] = temp;
              }
              for (int k = 0; k <= array.Length - 1; k++)
              {
                 Console.Write(tempArray[k] + " ");
              }
              Console.WriteLine();
           }
        }
```

Selection Sort

Complexity O (N^2)
Logic: Select the first element and compare it with entire array and find the min then replace the first element with min, do this for every iteration
Difference between Bubble and Selection sort is that bubble compares adjacent elements, selection compares with entire array
Algorithm

```
for (int i = 0; i < tempSortedArray.Length - 1; i++)
        {
          int min = i;
          for (int j = i + 1; j < tempSortedArray.Length; j++)
          {
            if (tempSortedArray[min] > tempSortedArray[j])
            {
               min = j;
            }
          }
          var temp = tempSortedArray[i];
          tempSortedArray[i] = tempSortedArray[min];
          tempSortedArray[min] = temp;
          }
```

Insertion Sort

Best Case :O(N)
Worst Case: O(N^2)
Logic: We Pick the element at index 1 and compare it with previous element if previous value is larger than we swap it with larger value's place, this happens for all the previous elements in the array
I.e., shifting elements to right and move the value to the left (Example Connection)
Algorithm

```
        for (int i = 1; i < tempSortedArray.Length; i++)
        {
          var temp = tempSortedArray[i];
          int j = i - 1;
          while (j>=0 && tempSortedArray[j]>temp)
          {
            tempSortedArray[j + 1] = tempSortedArray[j];
            j--;
            Steps++;
          }
          tempSortedArray[j + 1] = temp;
          PrintArray(tempSortedArray, "");
          Steps++;
        }
```

Merge Sort

Logic: Divide and conquer the array until last unit – By Recursion Method
Complexity: O (N Log N)
Problem with space complexity
Created by Recursion
Work on Algorithm and implement it

Quick Sort

Logic: We select a Pivot and have 2 counters i.e. I and j where I = j-1, if element at index j is less then pivot then increment i by 1, swap the values of index I and j and when j finally reaches the pivot, we increment I by 1 and swap the pivot to position of i.
If headed in right direction then elements in array to left of Pivot will be less then pivot value, and right will be greater than pivot value, then break the Array into multiple units excluding the pivot
This is done in recursive fashion
Best case and Average case: O (N Log N)
Worst Case: O(N^2) If Already Sorted
Space Complexity: O (N Log N)

Entity Framework (an ORM Framework)

- To be Good at Entity framework you need to have solid foundation on LINQ and Async Programming
- What this is Entity Framework
    - Used to handle RDBMS (SQL server , PostgreSQL , MYSQL , in Memory for testing etc) which are built around table (Refer SQL server Notes)
    - is ORM Framework (object relational Mapping) it is a technique that allows us to query and manipulate data from a Database in Object oriented paradigm
        - So ORM Framework automatically creates Classes for Tables and vice versa (just by adding database schema we can get the above features)
        - Objective of EF is to is to supersede ADO.NET
    - Advantages
        - Is that it abstracts us from the database
        - We can write queries using C# and LINQ
        - Clean code
        - Quick Development
    - Disadvantages
        - Performance wise querying in SQL is much faster and optimal as you have more control
        - Difficult learning curve
        - SQL Stored procedures and views are a bit complex to handle
    - Supports Code First (Default convention) and DB first approach (has limited support as of now)
        - We also have model first approach , but its closer to DB First approach (we create models and then we get auto generated scripts for creation of tables , views etc)

- Dapper is a light weight ORM (which we might consider using), better performance and easier to learn.
- Latest Version is EF 6 (Built for .NET Framework with Support for .NET core , this is still in support but no longer has future development)
- EF Core 7 * is latest and is future (as of date EF Core 8 is in preview stage).
- So, Use EF Core 7 for new developments
- comparison between EF 6 and EF Core 7 (Ref Link)

Getting Started with Entity Framework

- Install SQL Server Management Studio 19 (Latest as of now)
- Run the following commands
    - install --global dotnet-ef
    - dotnet tool update --global dotnet-ef
    - Install through Nuget Packages (More in case you need them)
        - Microsoft.EntityFrameworkCore.SqlServer
        - Microsoft.EntityFrameworkCore.Design
        - Microsoft.Extensions.Configuration.Json
        - Microsoft.Extensions.Logging.Console (use full for seeing background queries in EF)

(Reference Sheet)

To Bind the data to Front End we can use

- Entity Source control (in case of DB first approach)
- Object Data Source control (in case of Code first approach)

Working with Entity Framework with Code first approach

(The primary focus is Code First Approach **)**

- Create classes (called Model Classes) and fill them with properties (Classes = Tables and Properties = Column names)
- Create another class (Context Class / DerivedDBContextClass) and inherit DBContext class (refer Dbcontext , context options and DBset below)
- ***The Below step is for understanding and knowing the background , in ASP .NET we do not this part* (not sure if its same with WPF and WinForms)***
- Create another Class (The Factory Class) and it will implement the interface IDesignTimeDbContextFactory<DerivedDBContextClass> (or we can directly use DbcontextOptions)
    - In this implementation Create an instance of DbContextBuilder<DerivedDbContext>
        - And .UseSqlserver /.UseOracleDB etc and pass the connection strings (Connection string should always be in config files *)
        - Why store connection string config
            - It is safer as they contain sensitive information.
    - Create an instance of ConfigurationBuilder that reads setting from config and pass those settings.
- Then we run add migration command , then check the SQL script and update the database, this allows us to create data Database and tables etc automatically.
- Decorate the properties using Attributes to customize the schema that is being created (useful when setting names for PK , FK etc)
    - Example : Create an additional property and pass this property as string to attribute.
- Data Annotations are used to put Attributes on the properties (like max length , primary key etc). Present in System.ComponentModel.DataAnnotations (similar to that of putting constraints on columns on in SQL server)
- *By Default, Navigation properties are not loaded , for them to load we need to use include Method and pass the navigation property as string to method (while reading)*

DbContext ,DbContextOption and DbSet Class

DbContext

- Class that is responsible for interacting with underlying Database
- Manages the Database connections
- Used for Saving and retrieval of data from database
- We create a class and inherit DbContext class
- Dbcontext exists in Microsoft.EntityFrameworkCore Namespace

DbContextOptions

- For DbContext to perform operations we need an instance of DbContextOptions carries information such as
    - Connection String
    - Database Provider etc.

Dbset

Properties (are declared in DbContext Class)

- Create Properties for each available Entity type / Required Entity Type and we write LINQ queries against thing property and these queries are translated to SQL Queries
- Implements IQueryable and IEnumerable

All Put together

(Should looks something like this in the end)

- Syntax :

```
Public class DerivedDbConextClass : DBContext
{
Public DerivedDBContext Class(DbcontextOptions< DerivedDbConextClass abc> () : Base(abc){}
Public DBset ModelClass {get; set;}
}
```

Handling model changes in EF

- We handle changes using Migrations. So, what is this Migration anyway ?
  - The auto generated code responsible for maintaining our database schema (Tables , columns in tables , keys etc). Taking the C Sharp code and converting it into the SQL like queries in the background.
  - Whenever we make modifications to our model, we need to add new migration(creates a modified migration code) and then run database update command (for the changes to be reflected)

- So how do we use the migrations
  - Once all your models are set up appropriately , through the terminal we can run the below commands
  - dotnet ef migrations add <MigrationName>
    - Creates a new folder (named Migrations) in our Solution where we can see a class with LINQ Style auto generated code which is conversion of our Model to DB
  - dotnet ef migrations script
    - Auto generated Script of Schema creation in SQL, used for inspecting if the inconsistencies or mistakes in query before we run the update operations
  - dotnet ef database update
    - Updates / Runs the migration command in SQL domain (latest Migration is picked by default)
  - dotnet ef migrations remove
    - Removes the migration

- After a Db is generated and later model is changed and exception will be thrown as Model and DB are not in sync anymore.
- To check if the model has changed or not , check _MigrationHistory (auto generated table)
  - Contains GUID (Migration ID) ,ContextKey (Our context Class) ,HashID of the Context Class and EF Version
- Global asax (Global Application Class) (this is how we do it in .NET Framework , need check how we do this in .NET 7)
  - On Start check if there is a DB to our model and then choose to Drop , recreate the DB

Basic CRUD operations on DB using EF

Create

- After the schema is created in DB, we simply create new instances of our model classes and populate them with relevant information
- Then create an instance of DerivedDbContext , use the relevant DBset Property and then save the changes (always use async methods = SaveChangesAsync)

Update - Just Modify the object values as intended and call SaveChangesAsync
Delete - Use Remove Method and pass the object which we want to remove
Read - Use Where Clause to fetch Data

Change Tracker in EF

(is on by default , can be disabled to improve performance **)**

- Mechanism which tracks changes to objects which it already knows.
- dbContext.Entry(model class object) , to get entries of change tracker and each entry has a state(in other words Entries go through the below states)
    - Detached State then ef has no idea about the object / The object is not being tracked.
    - Added State then ef knows a new object is added
        - so, if SavechangesAsync is called then items with this status will be added to DB
    - Unchanged State that means that there are no changes
        - Objects are written to Database
    - Modified State , it means some changes have been made ,but not yet written to database
    - Deleted State , it means object has been removed
- We can not only read states but also manually assign states based on requirement

OriginalValue Property in Change Tracker(an important property)

- Entry.OriginalValue[nameof(modelClass.Property)] , to know what the original value
    - The value that is saved to Db is treated as original value
    - And then compares the current value of that objects are identifies the changes and marks the states in Change tracker
- These change trackers are specific for each specific data context (That means change trackers are isolated by context)

Attaching Entities Update() in Change Tracker

- Takes an object which is not part of change tracker and adds it to the change tracker (i.e., objects from sources like API or any external source which are not ef objects , we can take these objects and hook them to our change tracker)
- Overwrites the entire object when we use Update().

Disabling Change Tracking

- Use the .AsNoTracking() method
    - This improves performance , so when it is read only ,consider using this method

Executing Raw SQL statements in EF

(when LINQ is not sufficient, we can use this method to write sql statements**)**

- Downside is it opens gate for SQL injection
- FromSqlRaw (your sql statement)
    - Consider using FromSqlInterpolated
- FromSqlInterpolated (your sql statement)
    - If you have parameters use FromSqlInterpolated
- ExecuteSqlRawAsync (your sql statement)
    - if you do not return any data

Transactions in SQL

- Transaction is either everything is committed else everything is rolled back.
- So how do we implement transaction in EF ?
    - Using var transcation = await Context.Database.BeginTransactionAsync()
      Try
      {
      // Do something
      await transaction.CommitAsync();
      }
      Catch
      {
      //Do something
      }

Expression Trees

(Complex topic but need to understand its applications**)**
C# code translated by compiler into IL-> IL is compiled by JIT -> to assembly language and executed in memory is the general case.
But magically, EF prevents the statement from being compiled to ML because it makes more sense for our code to be compiled /translated to SQL language rather than ML.

- This is done at Runtime (cannot interpret the ML generated method , done by LINQ.Expression)
    - Func<Dish,bool> func = x => x.Title.StartsWith("B"); ← Randomly Generated Method

- With a concept called Expression Trees
    - Expression< Func<Dish, bool>> exfunc = x => x.Title.StartsWith("B"); ← Generates Object Tree ,EF Read this at runtime and translates this to SQL

Relationships and Inheritance

By Default , Base class and Derived class are put into single table this is called table per hierarchy.

Discriminator is a special column of type text , contains the name of the class that is object is of , the other columns are null based on the value in discriminator

To force EF to Store derived Tables we need do the following

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  // Can be used to Set Data Annotations

  modelBuilder.Entity<DerivedClass>().HasBaseType<ParentClass>();
}
```

We can use this to set DataAnnotations for our properties (another option)(need to check on all other properties)

Include Method – enforcing inner join in simplest way (also takes column names)
Entry().Collection().LoadAsync() – This line with explicitly load the instances

Pending Questions

BenchMark.Net instead of Stopwatch for Performance
What is Hash code
Dictionary vs Hash Table
ICollection
Ref Keyword
Unsafe Keyword
Indexers
Generic Constraints
CoVariance and ContraVariance

EF Questions
Include Method – enforcing inner join in simplest way
Object Graphs ?
Expression Trees ?
Relations (Many to Many , One to Many and One to One) ?
Constraints and Keys in EF
Generates Foreign Keys , Data Tables all on the Fly

Skills to Learn soon
VModel, Water Flow, and other Flows
Dependency Injection

- Constructor Injection
- Method injection
- Property injection

Git (Version Control **)**
Scripting (Objective is to Automate **)**
What is Type Script
Usage of Terminal
Architectural Pattern: MVC*, Impactor Methods, Façade ,3 Layer etc, MVVM
Azure (Cloud Tech)
ASP.NET (Frontend Tech: Angular, React etc, Blazor and Razor pages)
ADO.NET (Not sure if it is still relevant today**)**
Micro Services
WCF
REST API
Apache Bench mark tool , for load Testing

Pending Topics

Hash Table
Similar to that of dictionary KeyValue Pairs, Check the coding and theory once again

Graphs
Directed and Undirected Graphs, each node/Vertex is Data, Edge is connection
For Modelling a Network

Adjacency Matrix
Time Complexity :O (1)
Space Complexity: O(V^2) V = number of vertices

Adjacency List
Time Complexity :O (V)
Space Complexity: O(V+E) V = number of vertices + number of Edges

DFS
BFS
Tree Data Structure
Binary Search Trees
Tree Traversal
(Ref Link)
for the above remaining topics , also write programs for better understanding and efficiency