# .NET ( small things to remember )

| | |
|---|---|
| Created by | Aayush Malakar |
| Created time | @September 6, 2024 9:23 AM |
| Tags | |

## Top Level statement

Top-level statements were introduced in C# 9 as a syntax feature that allows you to reduce code by removing the `Main` method and its associated class. While the compiler needs this information to build the program, the programmer doesn't need to see any of it in most cases. The programmer only needs to see the working code. This means you can place executable code at the top of the file starting right on line 1. Hence, "top-level" statements. Top-level statements do enable quick experimentation and beginner tutorials. They also provide a smooth path from experimentation to full programs. The code in top-level statements are not reusable since all the  code is compiled inside of the Main method.

A few caveats about top-level statements:

- You can only use top-level statements *in one file* in a project. Most likely, `Program.cs`, as most developers expect to see an entry point in a file with this name.
- Any code blocks such as type definitions or methods after the individual lines of code are not considered top-level.
- Top-level statements are excellent for simplifying smaller applications. They might not work well in large or complex apps that need the extra structure.

## Define Namespaces.

Namespace is a way to organize and group related classes, interfaces, structs, enums, and other types into a logical hierarchy. It helps avoid naming conflicts between types with the same name by providing a unique identifier for each type based on its namespace.

```
namespace LibraryA {
    class Logger {
        // Logger implementation from Library A
    }
}

////
// Logger class from Library B
namespace LibraryB {
    class Logger {
        // Logger implementation from Library B
    }
}


////
using LibraryA;
using LibraryB;

class Program {
    static void Main(string[] args) {
        // Using Logger from Library A
        var loggerA = new LibraryA.Logger();
        loggerA.Log("This is a message from Library A");

        // Using Logger from Library B
        var loggerB = new LibraryB.Logger();
        loggerB.Log("This is a message from Library B");

        // ...
    }
}
```

# Garbage collection in C#

The garbage collector (GC) in C# is a feature of the .NET runtime responsible for automatically reclaiming memory occupied by objects that are no longer in use, thereby preventing memory leaks and managing memory efficiently

The garbage collector works by periodically scanning the application's memory to determine which objects are still being used and which are no longer needed. Objects that are no longer being used are marked for garbage collection, and their memory is freed up automatically by the garbage collector.

## Advantage

Frees developers from having to manually release memory.

- Allocates objects on the managed heap efficiently.
- Free developers from having to manually release memory
- Reclaims objects that are no longer being used, clears their memory, and keeps the memory available for future allocations. Managed objects automatically get clean content to start with, so their constructors don't have to initialize every data field.
- Provides memory safety by making sure that an object can't use for itself the memory allocated for another object.

## Process of garbage collection

A garbage collection has the following phases:

- Marking phase - Marking phase finds and creates a list of all live objects.
- Relocating phase - A relocating phase updates the references to the objects that will be compacted.
- Compacting phase - A compacting phase reclaims the space occupied by the dead objects and compacts the surviving objects. The compacting phase moves objects that have survived a garbage collection towards the older end of the segment.

After the CLR initializes the garbage collector, it allocates a segment of memory to store and manage objects. This memory is called the managed heap, as opposed to a native heap in the operating system.

There's a managed heap for each managed process. All threads in the process allocate memory for objects on the same heap.

When a garbage collection is triggered, the garbage collector reclaims the memory that's occupied by dead objects. The reclaiming process compacts live objects so that they're moved together, and the dead space is removed, thereby making the heap smaller. This process ensures that objects that are allocated together stay together on the managed heap to preserve their locality.

### Generations

Garbage collection primarily occurs with the reclamation of short-lived objects. To optimize the performance of the garbage collector, the managed heap is divided into three generations, 0, 1, and 2, so it can handle long-lived and short-lived objects separately. The garbage collector stores new objects in generation 0. Objects created early in the application's lifetime that survive collections are promoted and stored in generations 1 and 2. Because it's faster to compact a portion of the managed heap than the entire heap, this scheme allows the garbage collector to release the memory in a specific generation rather than release the memory for the entire managed heap each time it performs a collection.

- **Generation 0:** All the short-lived objects such as temporary variables are contained in the generation 0 of the heap memory. All the newly allocated objects are also generation 0 objects implicitly unless they are large objects. In general, the frequency of garbage collection is the highest in generation 0.
- **Generation 1 :** If space occupied by some generation 0 objects that are not released in a garbage collection run, then these objects get moved to generation 1. The objects in this generation are a sort of buffer between the short-lived objects in generation 0 and the long-lived objects in generation 2.
- **Generation 2 :** If space occupied by some generation 1 objects that are not released in the next garbage collection run, then these objects get moved to

generation 2. The objects in generation 2 are long lived such as static objects as they remain in the heap memory for the whole process duration.

```
using System;

public class Demo
{
    // Main Method - The entry point for the program
    public static void Main(string[] args)
    {
        // Print the number of generations in the garbage collec
        // GC.MaxGeneration returns the maximum number of genera
        // This will output something like "The number of genera
        Console.WriteLine("The number of generations are: " + GC

        // Create a new instance of the Demo class
        // This instance is a candidate for garbage collection
        Demo obj = new Demo();

        // Print the total amount of memory currently allocated
        // GC.GetTotalMemory(false) returns the number of bytes
        Console.WriteLine("Total Memory:" + GC.GetTotalMemory(fa

        // Print the generation number of the object 'obj'
        // GC.GetGeneration(obj) returns the generation number o
        // At this point, 'obj' is likely to be in Generation 0
        Console.WriteLine("The generation number of object obj i

        // Print the total amount of memory allocated again
        // This helps to see if memory has changed after the obj
        Console.WriteLine("Total Memory:" + GC.GetTotalMemory(fa

        // Force garbage collection for Generation 0
        // GC.Collect(0) forces a garbage collection of objects
        // This can be useful for testing purposes or to clean u
        GC.Collect(0);

        // Print the number of times garbage collection has occu
        // GC.CollectionCount(0) returns the number of times GC
        Console.WriteLine("Garbage Collection in Generation 0 is
    }
}



// output
//The number of generations are: 2
// Total Memory:117032
// The generation number of object obj is: 0
// Total Memory:117032
// Garbage Collection in Generation 0 is: 1
```

## Memory leakage and how to handle it

A memory leak in C# happens when an application repeatedly allocates memory but neglects to release it even after the memory has served it purpose. As a result, the application gradually used more memory over time, which may result in the application crashing or ceasing to function.

The most common type of unmanaged resource is an object that wraps an operating system resource, such as a file handle, window handle, or network connection. Although the garbage collector can track the lifetime of a managed object that encapsulates an unmanaged resource, it doesn't have specific knowledge about how to clean up the resource.

You must also provide a way for your unmanaged resources to be released in case a consumer of your type forgets to call `Dispose`. You can either use a safe handle to wrap the unmanaged resource, or override the **Object.Finalize()** method.

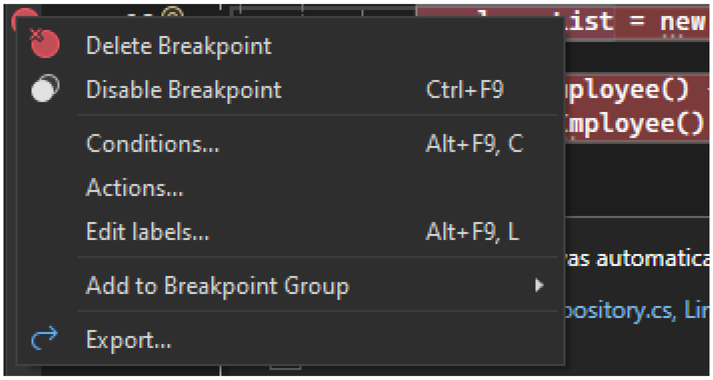
- Implement the dispose pattern. This requires that you provide an IDisposable.Dispose implementation to enable the deterministic release of unmanaged resources. A consumer of your type calls Dispose when the object (and the resources it uses) are no longer needed. The Dispose method immediately releases the unmanaged resources

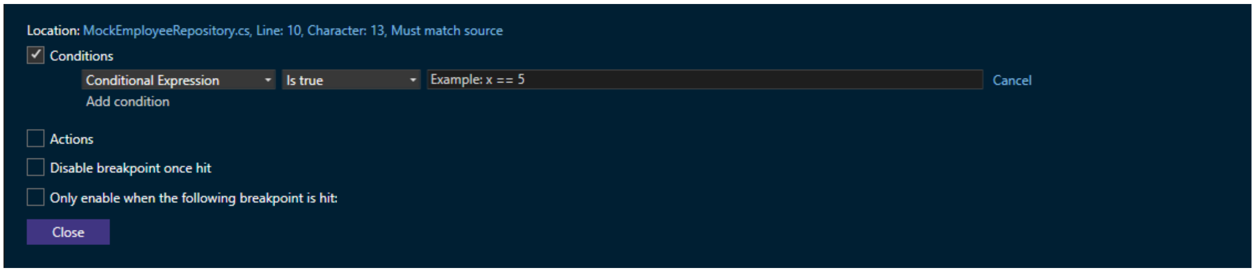
- Define a finalizer. Finalization enables the non-deterministic release of unmanaged resources when the consumer of a type fails to call IDisposable.Dispose to dispose of them deterministically.

## Conditional breakpoints in visual studio

Conditional breakpoints in C# are a powerful feature in debugging that allows you to pause execution of your program only when certain conditions are met. This can be incredibly useful for isolating and diagnosing specific issues without having to manually sift through a lot of data or code.

We can open conditional breakpoint by setting a breakpoint and the right clicking on the breakpoint.





Location: MockEmployeeRepository.cs, Line: 10, Character: 13, Must match source

☑ Conditions

Conditional Expression | Is true | Example: x == 5 | Cancel

Add condition

□ Actions

□ Disable breakpoint once hit

□ Only enable when the following breakpoint is hit:

Close

## Command line arguments ( String[] args )

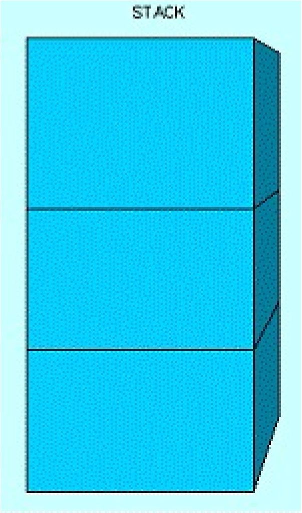
Command-line arguments are often used in C# applications to provide input parameters or configure the behavior of the application when it starts. They are particularly useful for command-line utilities, tools, or applications that need to be configurable without requiring user interaction or additional configuration files.

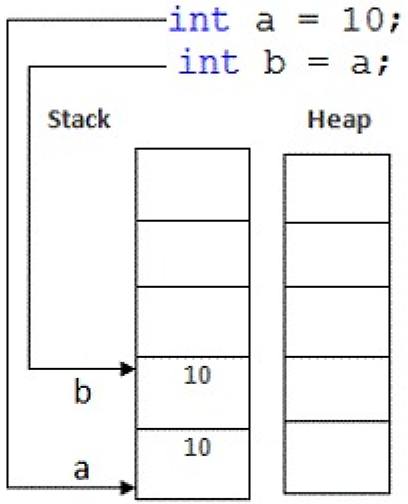
- Command line arguments are captured into the string array i.e. args parameter of the Main method.
- In General, the Command Line Arguments are used to specify configuration information while launching your application.
- Information is passed as strings.
- There is no restriction on the number of command line arguments. You can pass 0 or n number of command line arguments.

# Heap and Stack

## Stack

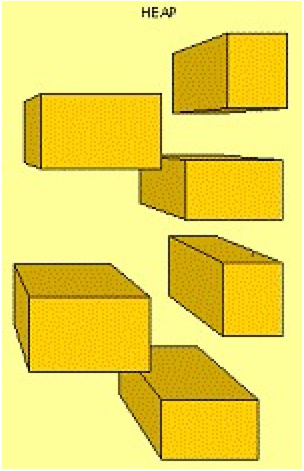
The stack is a region of memory that follows the Last In,  First out ( LIFO ) principle. It is used for storing local variables and managing function calls. When a method is called, a stack frame is created, containing information such as the method's parameters, local variable, and the return address. As the method completes execution, the stack frame is popped off the stack, releasing the allocated memory. In stack memory allocation is Static

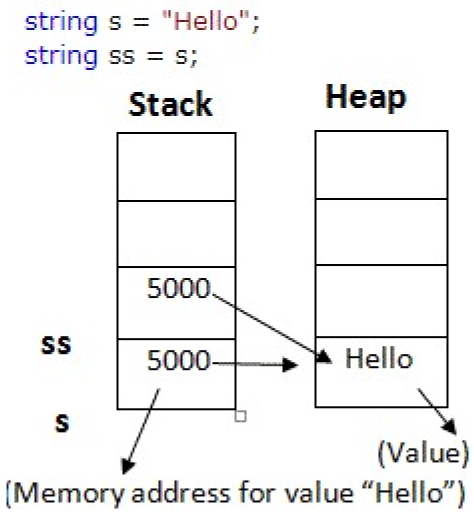




## Heap

Heap is a region of memory that follows no particular order, and memory allocation and deallocation are no straightforward. The heap is used for dynamic memory allocation, where the size and lifetime of variables are not known at compile time. In it data can be stored and removed in any order. Garbage collector is used to monitor allocation of heap space. It only collects heap memory since objects are only created in heap





```
string s = "Hello";
string ss = s;
```

# Scope of variable in C#

The part of the program where a particular variable is accessible is termed as the Scope of that variable. A variable can be defined in a class, method, loop etc. In C/C++, all identifiers are lexically (or statically) scoped, i.e. scope of a variable can be determined at compile time and independent of the function call stack. But the C# programs are organized in the form of classes.

So C# scope rules of variables can be divided into three categories as follows:

- **Class Level Scope**
- **Method Level Scope**
- **Block Level Scope**

### Class Level Scope

- Declaring the variables in a class but outside any method can be directly accessed anywhere in the class.
- These variables are also termed as the fields or class members.
- Class level scoped variable can be accessed by the non-static methods of the class in which it is declared.
- Access modifier of class level variables doesn’t affect their scope within a class.

- Member variables can also be accessed outside the class by using the access modifiers.

```
using System;

class MyClass
{
    // Class Level Scope
    private int classLevelVariable = 10;

    public void Display()
    {
        // Accessing class-level variable
        Console.WriteLine($"Class Level Variable: {classLevelVar
    }

    public void Modify()
    {
        // Modifying class-level variable
        classLevelVariable = 20;
    }
}

class Program
{
    static void Main()
    {
        MyClass obj = new MyClass();
        obj.Display(); // Outputs: Class Level Variable: 10

        obj.Modify();
        obj.Display(); // Outputs: Class Level Variable: 20
    }
}
```

### Method Level Scope

- Variables that are declared inside a method have method level scope. These are not accessible outside the method.
- However, these variables can be accessed by the nested code blocks inside a method.
- These variables are termed as the local variables.
- There will be a compile-time error if these variables are declared twice with the same name in the same scope.
- These variables don’t exist after method’s execution is over.

```
using System;

class MyClass
{
    public void MyMethod()
    {
        // Method Level Scope
        int methodLevelVariable = 5;
        Console.WriteLine($"Method Level Variable: {methodLevel
    }

    public void AnotherMethod()
    {
        // Trying to access methodLevelVariable here would cause
        // Console.WriteLine(methodLevelVariable); // Error
    }
}

class Program
{
    static void Main()
    {
        MyClass obj = new MyClass();
        obj.MyMethod(); // Outputs: Method Level Variable: 5

        // obj.AnotherMethod(); // No access to methodLevelVaria
    }
}
```

### Block Level Scope

- These variables are generally declared inside the for, while statement etc. and also in conditional statements like if.
- These variables are also termed as the loop variables or statements variable as they have limited their scope up to the body of the statement in which it declared.
- Generally, a loop inside a method has three level of nested code blocks(i.e. class level, method level, loop level).
- The variable which is declared outside the loop is also accessible within the nested loops. It means a class level variable will be accessible to the methods and all loops. Method level variable will be accessible to loop and method inside that method.
- A variable which is declared inside a loop body will not be visible to the outside of loop body.

```
using System;

class MyClass
{
    public void BlockScopeExample()
    {
        // Block Level Scope
        if (true)
        {
            int blockLevelVariable = 100;
            Console.WriteLine($"Block Level Variable: {blockLeve
        }
        // Trying to access blockLevelVariable here would cause
        // Console.WriteLine(blockLevelVariable); // Error
    }
}

class Program
{
    static void Main()
    {
        MyClass obj = new MyClass();
        obj.BlockScopeExample(); // Outputs: Block Level Variabl
    }
}
```

# Data Types along with declaring syntax

## Primitive type

- **Integer Type**
  - byte
  - short
  - int
  - long
  - DateTime - C# DateTime is a structure of value Type like int, double etc.
- **Floating Point Type**
  - float
  - double
- **Decimal Type**
  - decimal
- **Boolean Type**

    - bool

- **Character Type**
    - char
- **Predefined Reference Type**
    - string
    - object

```
using System;

class Program
{
    static void Main()
    {
        // Byte: 8-bit unsigned integer
        byte byteValue = 255;
        Console.WriteLine($"Byte Value: {byteValue}");

        // Short: 16-bit signed integer
        short shortValue = 32767;
        Console.WriteLine($"Short Value: {shortValue}");

        // Int: 32-bit signed integer
        int intValue = 2147483647;
        Console.WriteLine($"Int Value: {intValue}");

        // Float: 32-bit floating point
        float floatValue = 3.14f;
        Console.WriteLine($"Float Value: {floatValue}");

        // Double: 64-bit floating point
        double doubleValue = 3.141592653589793;
        Console.WriteLine($"Double Value: {doubleValue}");
        // Long: 64-bit signed integer
        long longValue = 9223372036854775807;
        Console.WriteLine($"Long Value: {longValue}");

        // Char: Single 16-bit Unicode character
        char charValue = 'A';
        Console.WriteLine($"Char Value: {charValue}");

        // Bool: Boolean value (true or false)
        bool boolValue = true;
        Console.WriteLine($"Bool Value: {boolValue}");

        // DateTime: Represents date and time
        DateTime dateTimeValue = new DateTime(2024, 9, 8);
        Console.WriteLine($"DateTime Value: {dateTimeValue.ToSh

        // String: Represents a sequence of characters
        string stringValue = "Hello, World!";
        Console.WriteLine($"String Value: {stringValue}");

        // Object: Base type of all other types
        object objectValue = "This is an object";
        Console.WriteLine($"Object Value: {objectValue}");

        // Demonstrating object can hold any data type
        objectValue = 12345; // Assigning an integer
        Console.WriteLine($"Object Value (now integer): {object
    }
}
```

## Non - Primitive type

- class - A class is a reference type that can contain data members (fields), methods, and other types such as properties and events.

- struct - A struct is a value type that is typically used for small, simple objects. Unlike classes, structs are usually used for data that has a small memory footprint.
- enum - An enum (short for "enumeration") defines a set of named integral constants.
- interface - An interface defines a contract that classes or structs can implement. It only contains method signatures, properties, events, or indexers, but no implementation.
- delegate - A delegate is a type that represents references to methods with a particular parameter list and return type. It is used to pass methods as arguments.
- array

```
// class
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public void Greet()
    {
        Console.WriteLine($"Hello, my name is {Name} and I am {
    }
}

//Struct
public struct Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public void Display()
    {
        Console.WriteLine($"Point coordinates: ({X}, {Y})");
    }
}

// Enum
public enum DaysOfWeek
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}

//Interface
public interface IPlayable
{
    void Play();
    void Pause();
}

// Delegate
public delegate void Notify(string message);

public class Notifier
{
    public Notify OnNotify;

    public void TriggerNotification(string message)
    {
        OnNotify?.Invoke(message);
    }
}

// Array
```

```
public class Program
{
    public static void Main()
    {
        // Initializing an array of integers
        int[] numbers = { 1, 2, 3, 4, 5 };

        // Accessing and printing array elements
        foreach (int number in numbers)
        {
            Console.WriteLine(number);
        }
    }
}
```

# String interpolation, String Literal and Verbatim string literal

## String literals

They are enclosed in double quotes and represent sequences of characters. They support escape sequences to represent special characters.

```
string standardString = "Hello, World!";

// Some of the Escape Sequences:
// \n - New line
// \t - Tab
// \" - Double quote
// \\ - Backslash

string escapedString = "This is a line.\nThis is another line."
string quotedString = "He said, \"Hello!\"";
```

## String Interpolation

**String interpolation** allows you to embed expressions inside string literals. This feature is useful for creating strings that include variable values or expressions in a readable way. It is introduced in C# 6.0.

```
string name = "Alice";
int age = 30;
string message = $"Hello, {name}! You are {age} years old.";
// message is "Hello, Alice! You are 30 years old."

//Expressions within Interpolation:
// You can use more complex expressions within the curly braces
int a = 5;
int b = 10;
string result = $"Sum of {a} and {b} is {a + b}";
// result is "Sum of 5 and 10 is 15"

//Formatting Dates:
//You can use format specifiers to control the format of dates i
DateTime now = DateTime.Now;
string formattedDate = $"Current date and time: {now:MMMM dd, yy
// Output example: "Current date and time: September 08, 2024 14

// Formatting Numbers:
// You can also format numbers using standard numeric format str
double pi = 3.14159;
int number = 1234567;
string formattedNumbers = $"Pi rounded to 2 decimal places: {pi
// Output:
// Pi rounded to 2 decimal places: 3.14
// Number with commas: 1,234,567

// Verbatim String Interpolation
// Verbatim string literals with interpolation are
// prefixed with @ and allow for multi-line strings with embedde
string userName = "Alice";
string filePath = @"C:\Users\";
string fileName = "document.txt";
string fileFullPath = $@"{filePath}{userName}\{fileName}";
// fileFullPath is "C:\Users\Alice\document.txt"
```

### Verbatim String

A verbatim string literal is a string that is prefixed with an `@` symbol and allows for multi-line strings and includes escape sequences like backslashes without needing additional escaping.

It uses ‘@’ before the opening double quotes. They ignore escape sequences, expect for ‘ “” ‘ which represent a single double quote.

```
string path = @"C:\Users\Public\Documents\file.txt";

// Mutli-line
string multiLineString = @"This is a string
that spans multiple lines.
Each new line is preserved as part of the string.";

//Including Double Quotes
string quotedString = @"He said, ""Hello, World!""";
```

# Value Type and reference type

## Value Type

- Value types are fixed in size.
- Value types are made in the system stack.
- Actual values of data are stored in the stack.
- If you assign a value of a variable to another it will create two copies.

All primitive data types except string and object are examples of value types.

The object is a supertype. It can store any type and any size of data. Object is called super type because it helps in inheritance.

struct and enum are value types.

**Note.** Stack is an operation entity (LIFO) i.e. it is fixed in size.

## Reference Type

- Reference types are not fixed in size.
- They are maintained in a system-managed heap but it also uses a stack to store the reference of the heap.
- Two primitive types (string and object) and non-primitive data types (class, interface & delegate) are examples of reference types.

CLR manages the heap (large memory area). The heap address is accessed from the stack. In reference type reference is used for processing using both managed heap and stack (operational entity).

## New Keyword

New keyword are used to created an instance of class and also can be used to invoke the constructor. There are also other purpose and it is listed below.

### Creating object

```
// Creating an instance of a class
MyClass myObject = new MyClass();

// Creating an instance of a struct
MyStruct myStruct = new MyStruct();
```

### Object initializer ( invokes constructor )

```
Person person = new Person
{
    FirstName = "John",
    LastName = "Doe",
    Age = 30
};
```

## Initializing Arrays or any other list

```
int[] numbers = new int[10]; // Creates an array of 10 integers

string[] names = new string[] { "Alice", "Bob", "Charlie" }; //
```

## Anonymous Types

`new` is used to create anonymous types, which are types created without a specific class definition. These types are often used for temporary data structures.

```
var person = new
{
    FirstName = "Jane",
    LastName = "Smith",
    Age = 25
};
```

## Overloading Constructors or hiding member

```
public class BaseClass
{
    public void Display()
    {
        Console.WriteLine("BaseClass Display");
    }
}

public class DerivedClass : BaseClass
{
    public new void Display() // Hides the BaseClass Display met
    {
        Console.WriteLine("DerivedClass Display");
    }
}

// you can hide the main methods without the new keyword but it
```

# Global, Static and Non-static variable difference

## Global variable

There is no concept of global variable in C#.

## Static variable

In C#, a `static` variable is a class-level variable that is shared among all instances of the class. Unlike instance variables, which have separate values for each object, static variables have a single shared value for the entire class. Here's a detailed overview of static variables in C#:

Static variables are initialized when the class is first accessed. They are initialized only once, making them suitable for storing data that is common to all instances of the class.

```
public class MyClass
{
    // Static variable
    public static int StaticVariable = 0;

    // Instance variable
    public int InstanceVariable = 0;

    // Static method
    public static void StaticMethod()
    {
        // Access static variable
        StaticVariable++;
        Console.WriteLine("StaticVariable: " + StaticVariable);
    }

    // Instance method
    public void InstanceMethod()
    {
        // Access static variable
        StaticVariable++;
        Console.WriteLine("StaticVariable from instance method:
    }
}

class Program
{
    static void Main()
    {
        // Access static variable via class name
        Console.WriteLine("Initial StaticVariable: " + MyClass.S

        // Call static method
        MyClass.StaticMethod();

        // Create instances of MyClass
        MyClass obj1 = new MyClass();
        MyClass obj2 = new MyClass();

        // Call instance method which also modifies the static v
        obj1.InstanceMethod();
        obj2.InstanceMethod();

        // Access static variable via class name
        Console.WriteLine("Final StaticVariable: " + MyClass.Sta
    }
}
```

## Use cases

- **Singleton Pattern**: : Static variables are often used in the Singleton pattern to ensure that only one instance of a class exists throughout the application.
- **Constants**:
  Use
  `static` in conjunction with `readonly` or `const` to define constant values that are shared across all instances.
- **Utility or Helper Classes**:
  Static classes and static methods are often used for utility functions that don’t require object state.

## Limitations

- **Cannot be Instantiated**: Static classes cannot be instantiated. They can only contain static members.
- **No `this` Reference**: Static methods and variables cannot access instance members (non-static members) or the `this` reference.
- **Concurrency**: Care must be taken when using static variables in a multi-threaded environment to avoid race conditions.

## Static variable

In C#, a non-static variable, often referred to as an instance variable, is a member variable of a class that is tied to a specific instance of that class. Unlike static variables, which are shared across all instances of a class, each instance of a class has its own copy of non-static variables.

Non-static variables are initialized each time a new instance of the class is created. You can set their initial values either in the class constructor or through instance-specific initializers.

The lifetime of a non-static variable is tied to the lifetime of the instance of the class. It exists as long as the instance is alive.

```
public class Person
{
    // Non-static (instance) variable
    public string Name;

    // Another non-static (instance) variable
    public int Age;

    // Constructor to initialize non-static variables
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    // Instance method to display person details
    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {Name}, Age: {Age}");
    }
}

class Program
{
    static void Main()
    {
        // Creating instances of the Person class
        Person person1 = new Person("Alice", 30);
        Person person2 = new Person("Bob", 25);

        // Accessing non-static variables through instances
        Console.WriteLine($"{person1.Name} is {person1.Age} year
        Console.WriteLine($"{person2.Name} is {person2.Age} year

        // Calling instance method
        person1.DisplayInfo();
        person2.DisplayInfo();
    }
}
```

### Examples and Use Cases

1. **Instance Data Storage**:
   - Non-static variables are typically used to store data specific to each instance of a class. For example, in a `Person` class, each person might have different `Name` and `Age`.
2. **Object Behavior**:
   - Non-static variables can influence the behavior of an object. For instance, methods that operate on instance data can use these variables to perform operations specific to that instance.
3. **Encapsulation**:
   - Non-static variables are often used with encapsulation principles, where private instance variables are accessed and modified through public methods or properties.
4. **Constructor Initialization**:
   - Non-static variables can be initialized using constructors. This allows setting up the initial state of an object when it is created.

## String builder and string

string is immutable so once created, its value cannot be change. Operation that appear to modify a string actually create a new string. Every time you use any of the methods of the System.String class, then you create a new string object in memory. For example, a string “Hello world” occupies memory in the heap, now, by changing the initial string “Hello world” to “HW” will create a new string object on the memory heap instead of modifying the initial string at the same memory location.

```
String result = "";
for (int i = 0; i < 1000; i++) {
    result += i; // Creates new String object each time
}
```

whereas string builder is a dynamic object. It doesn’t create a new object in the memory but dynamically expands the needed memory to accommodate the modified or new string. It allows modification to the string content without creating new instances. Its more efficient for scenarios involving many modifications or concatenations.

```
StringBuilder result = new StringBuilder();
for (int i = 0; i < 1000; i++) {
    result.append(i); // Modifies the existing StringBuilder obj
}
String finalResult = result.toString(); // Converts to String or
```

Using a `StringBuilder` has several advantages over using immutable strings, especially in scenarios involving frequent modifications. Here are some key benefits:

1. **Performance Efficiency**: `StringBuilder` is more efficient for scenarios where you need to perform multiple modifications on a string (e.g., concatenation, insertions, deletions). This is because `StringBuilder` is mutable and does not create a new object for every modification. Instead, it modifies the existing object, reducing the overhead associated with creating and garbage collecting multiple immutable `String` objects.
2. **Memory Usage**: With immutable `String` objects, each modification results in the creation of a new `String` object, leading to higher memory consumption. `StringBuilder`, on the other hand, operates on a single mutable object, thus using memory more efficiently.
3. **Performance Overhead**: String operations like concatenation using `String` can lead to excessive performance overhead, especially in loops. Each concatenation operation with `String` creates a new `String` object. `StringBuilder` avoids this issue by maintaining a single mutable buffer, making it more suitable for scenarios involving many changes.

4. **Avoiding Temporary Objects**: When concatenating strings in a loop using `String`, intermediate `String` objects are created and discarded. `StringBuilder` helps avoid this by accumulating changes in its internal buffer and only producing a final result when needed.
5. **Thread Safety**: `StringBuilder` is not thread-safe, which is an advantage when working in a single-threaded context where the overhead of thread synchronization is unnecessary. If thread safety is required, you might use `StringBuffer`, which is a synchronized version of `StringBuilder`.

## Index in for each loop

**Option 1: Make integer loop variable**

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        var items = new List<string> { "apple", "banana", "cher

        int index = 0;
        foreach (var item in items)
        {
            Console.WriteLine($"Index: {index}, Item: {item}");

            // Example usage of break
            if (item == "cherry")
            {
                Console.WriteLine("Breaking loop.");
                break; // Exits the loop
            }

            // Example usage of continue
            if (item == "banana")
            {
                Console.WriteLine("Skipping banana.");
                index++; // Move to the next index
                continue; // Skips to the next iteration
            }

            index++;
        }
    }
}
```

### Option 2: Tuple with value *and* index

```
using System;
using System.Collections.Generic;
using System.Linq;

class Kodify_Example
{
    static void Main(string[] args)
    {
        // Create a list of values
        List<int> myValues = new List<int>()
        {
            23, 34, 45, 56, 67
        };

        // Loop over tuples with the item and its index
        foreach (var (item, index) in myValues.Select((value, i
        {
            Console.WriteLine($"{index}: {item}");
        }
    }
}
```

## Does C# takes dd/MM/yy format in default

No by default, C# uses the  ‘MM/dd/yyyy’ format for dates in many locales, but it can handle ‘dd/MM/yyyy’ with custom formatting. You need to specify this format explicitly when converting dates to strings or parsing them.

```
DateTime date = new DateTime(2024, 9, 9);
string format = "dd/MM/yyyy";
string formattedDate = date.ToString(format);

DateTime date = DateTime.ParseExact(dateString, format, CultureI
// DateTime date = DateTime.ParseExact(dateString, format, null

Console.WriteLine("Formatted Date String: " + formattedDate);
```

# Does Params allows non-homogeneous values ?

In C#, the `params` keyword allows a method to accept a variable number of arguments of a specific type. However, these arguments must be of the same type or implicitly convertible to the type specified in the `params` array. Therefore, the `params` keyword does not support heterogeneous (i.e., non-homogeneous) types directly.

## Using Object Type for Non-Homogeneous Values

If you need to accept a mix of different types, you can use `object` as the type for the `params` parameter. Since `object` is the base type for all types in C#, this allows you to pass arguments of any type:

```
public class Example
{
    public void PrintValues(params object[] values)
    {
        foreach (var value in values)
        {
            Console.WriteLine(value);
        }
    }
}

class Program
{
    static void Main()
    {
        Example example = new Example();

        // Passing a mix of different types
        example.PrintValues(1, "Hello", 3.14, new DateTime(2024,
    }
}
```

## Tuple Return order

The order of elements in a tuple is significant. The first element is always the first, the second is always the second, and so on. The order determines how you interpret and access the elements.

```
public (int Age, string Name) GetPerson()
{
    return (25, "Alice");
}

// Decomposing the tuple with named elements
var (Age, Name) = GetPerson();
Console.WriteLine(Age);  // 25
Console.WriteLine(Name); // Alice
```

### use case

One of the most common use cases of tuples is as a method return type. That is, instead of defining `out` method parameters, you can group method results in a tuple return type, as the following example shows: