Csharp Version 14 New Features

1. The 'field' keyword

The token **field** enables you to write a property accessor body without declaring an explicit backing field.

The token **field** is replaced with a compiler synthesized backing field.

Legacy code:

```
private string _msg;
public string Message
{
  get => _msg;
  set => _msg = value ?? throw new ArgumentNullException(nameof(value));
}
```

New C# v14 code:

```
public string Message
{
  get;
  set => field = value ?? throw new ArgumentNullException(nameof(value));
}
```

Now see another more complex sample about this new feature in C# 14:

```
Console.WriteLine("--- 1. Field Keyword ---\n");
var sensor = new Sensor();
sensor.Temperature = 50.0;
Console.WriteLine($"Sensor Temperature: {sensor.Temperature}°C");

try
{
    sensor.Temperature = -500; // Invalid, triggers validation
}
catch (ArgumentOutOfRangeException ex)
{
    Console.WriteLine($"Validation Error: {ex.Message}");
}

Console.WriteLine();
```

And also we have to define the **Sensor** class

```
public class Sensor
{
    public double Temperature
    {
        get;
        set => field = (value < -273.15)
            ? throw new ArgumentOutOfRangeException(nameof(value), "Temperature
cannot be below absolute zero.")
            : value;
    }
}
```

2. Implicit Span Conversions

C# 14 introduces first-class support for **System.Span** and **System.ReadOnlySpan** in the language.

See this sample:

```
Console.WriteLine("--- 2. Implicit Span Conversions ---\n");
byte[] buffer = { 10, 20, 30, 40 };
ImplicitSpanConversions.ProcessSpan(buffer);
Console.WriteLine();
```

```
public class ImplicitSpanConversions
{
    public static void ProcessSpan(Span<byte> data)
    {
        Console.WriteLine("Processing Array Data:");
        foreach (var b in data)
            Console.WriteLine(b);
    }

    public static void ProcessSpan(ReadOnlySpan<byte> data)
    {
        Console.WriteLine("Processing Span Data:");
        foreach (var b in data)
            Console.WriteLine(b);
    }
}
```

3. Unbound generic types and 'nameof'

Beginning with C# 14, the argument to nameof can be an unbound generic type.

For example, **nameof(List<>)** evaluates to List.

In earlier versions of C#, only closed generic types, such as **List**, could be used to return the List name.

```
Console.WriteLine("--- 3. nameof with Unbound Generic Types ---\n");
Console.WriteLine($"Unbound generic type name:
{nameof(System.Collections.Generic.Dictionary<,>)}");
Console.WriteLine();
Console.WriteLine($"Unbound generic type name: {nameof(List<>)}");
Console.WriteLine();
```

4. Simple lambda parameters with modifiers

You can add parameter modifiers, such as **scoped**, **ref**, **in**, **out**, or **ref readonly** to lambda expression parameters without specifying the parameter type.

Legacy code:

```
TryParse<int> parse2 = (string text, out int result) => Int32.TryParse(text, out
result);
```

New C# 14 code:

```
TryParse<int> parse1 = (text, out result) => Int32.TryParse(text, out result);
```

4.1. Using 'ref' with Lambda Parameters

```
Console.WriteLine("--- 4.1. Lambda with 'ref' parameter ---");
Console.WriteLine("--- Sample1 ---");
var multiply = (ref int x, ref int y) => x * y;
int a = 5, b = 3;
Console.WriteLine($"Multiply using ref lambda: {multiply(ref a, ref b)}");
Console.WriteLine();
Console.WriteLine("--- Sample2 ---");
var increment = (ref int number) => number++;
int value = 10;
increment(ref value);
Console.WriteLine($"Incremented Value: {value}"); // Output: 11
Console.WriteLine();
```

4.2. Lambda with 'out' parameter

```
Console.WriteLine("--- 4.2. Lambda with 'out' parameter ---");
var initialize = (out int number) => number = 42;
initialize(out int initializedValue);
Console.WriteLine($"Initialized Value: {initializedValue}"); // Output: 42
Console.WriteLine();
```

4.3. Lambda with 'in' parameter (read-only)

```
Console.WriteLine("---  4.3. Lambda with 'in' parameter (read-only) ---");
var printReadOnly = (in double num) => Console.WriteLine($"Read-only value:
{num}");
double readOnlyValue = 100.5;
printReadOnly(in readOnlyValue); // Output: Read-only value: 100.5
Console.WriteLine();
```

4.4. Lambda with 'ref readonly' parameter

```
Console.WriteLine("---  4.4. Lambda with 'ref readonly' parameter ---");
var showValue = (ref readonly int num) => Console.WriteLine($"Readonly Ref Value:
{num}");
int readOnlyRefValue = 20;
showValue(ref readOnlyRefValue); // Output: Readonly Ref Value: 20
Console.WriteLine();
```

4.5. Lambda with 'scoped' modifier

```
Console.WriteLine("--- 4.5. Lambda with 'scoped' modifier ---");
Span<int> numbers = stackalloc int[] { 1, 2, 3, 4 };
var sumSpan = (scoped Span<int> span) =>
{
    int sum = 0;
    foreach (var num in span)
        sum += num;
    return sum;
};
int total = sumSpan(numbers);
Console.WriteLine($"Sum of Span: {total}"); // Output: Sum of Span: 10
Console.WriteLine();
```

5. Partial Constructor and Events

A partial member has one **declaring declaration** and often one **implementing declaration**.

The **declaring declaration** doesn't include a body.

The **implementing declaration** provides the body of the member.

See this sample:

```
var employee = new Employee("Ada", "Lovelace");
Console.WriteLine($"Employee created with name: {employee.FullName}");
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CsharpVersion14Samples
{
    public partial class Employee
    {
        // Defining declaration of the partial constructor
        public partial Employee(string firstName, string lastName);

        public string FullName { get; private set; }
    }

    public partial class Employee
    {
        // Implementing declaration of the partial constructor
        public partial Employee(string firstName, string lastName) : this() //
Constructor initializer goes here
        {
            FullName = $"{firstName} {lastName}";
            Console.WriteLine($"Employee partial constructor invoked:
{FullName}");
        }

        // Base parameterless constructor
        public Employee()
        {
            Console.WriteLine("Employee base constructor invoked.");
        }
    }
}
```