Table of Contents

[**C# 10.0 Features** 4](#_Toc193220249)

[Records Structs 4](#_Toc193220250)

[📝 Introduction 4](#_Toc193220251)

[🚀 Benefits & Use Cases 4](#_Toc193220252)

[📌 Syntax & Explanation 4](#_Toc193220253)

[📊 Performance Analysis 5](#_Toc193220254)

[🧐 Comparison with Classes & Record Classes 5](#_Toc193220255)

[⚠️ Potential Pitfalls & Considerations 6](#_Toc193220256)

[Global Using Directives 7](#_Toc193220257)

[📝 Introduction 7](#_Toc193220258)

[🚀 Benefits & Use Cases 7](#_Toc193220259)

[📌 Syntax & Explanation 7](#_Toc193220260)

[📊 Performance & Maintainability 8](#_Toc193220261)

[🧐 Best Practices & Considerations 8](#_Toc193220262)

[⚠️ Potential Pitfalls: 8](#_Toc193220263)

[File-Scoped Namespace Declaration 9](#_Toc193220264)

[📝 Introduction 9](#_Toc193220265)

[🚀 Benefits & Use Cases 9](#_Toc193220266)

[📌 Syntax & Explanation 9](#_Toc193220267)

[📊 Performance Analysis 10](#_Toc193220268)

[🧐 Comparison with Traditional Namespaces 10](#_Toc193220269)

[⚠️ Potential Pitfalls & Considerations 10](#_Toc193220270)

[Const Strings Initialized Using String Interpolation 11](#_Toc193220271)

[📝 Introduction 11](#_Toc193220272)

[🚀 Benefits & Use Cases 11](#_Toc193220273)

[📌 Syntax & Explanation 11](#_Toc193220274)

[📊 Performance Analysis 12](#_Toc193220275)

[🧐 Comparison with Traditional Constants 12](#_Toc193220276)

[⚠️ Potential Pitfalls & Considerations 12](#_Toc193220277)

[Lambda Improvements 13](#_Toc193220278)

[📝 Introduction 13](#_Toc193220279)

[🚀 Benefits & Use Cases 13](#_Toc193220280)

[📌 Syntax & Explanation 13](#_Toc193220281)

[📊 Performance Analysis 14](#_Toc193220282)

[🧐 Comparison with Pre-C# 10 Lambdas 14](#_Toc193220283)

[⚠️ Potential Pitfalls & Considerations 14](#_Toc193220284)

[Improvements of Structure Types 15](#_Toc193220285)

[📝 Introduction 15](#_Toc193220286)

[🚀 Benefits & Use Cases 15](#_Toc193220287)

[📌 Syntax & Explanation 15](#_Toc193220288)

[📊 Performance Analysis 17](#_Toc193220289)

[🧐 Comparison with Pre-C# 10 Structs 17](#_Toc193220290)

[⚠️ Potential Pitfalls & Considerations 17](#_Toc193220291)

[Extended Property Patterns 18](#_Toc193220292)

[📝 Introduction 18](#_Toc193220293)

[🚀 Benefits & Use Cases 18](#_Toc193220294)

[📌 Syntax & Explanation 18](#_Toc193220295)

[📊 Performance Analysis 19](#_Toc193220296)

[🧐 Comparison with Traditional Property Matching 19](#_Toc193220297)

[⚠️ Potential Pitfalls & Considerations 19](#_Toc193220298)

[Sealed Modifier When Overriding ToString In a Record Type 20](#_Toc193220299)

[📝 Introduction 20](#_Toc193220300)

[🚀 Benefits & Use Cases 20](#_Toc193220301)

[📌 Syntax & Explanation 20](#_Toc193220302)

[📊 Performance Analysis 21](#_Toc193220303)

[🧐 Comparison with Traditional Method Override 21](#_Toc193220304)

[⚠️ Potential Pitfalls & Considerations 21](#_Toc193220305)

[Allow both assignment and declaration in the same deconstruction. 22](#_Toc193220306)

[📝 Introduction 22](#_Toc193220307)

[🚀 Benefits & Use Cases 22](#_Toc193220308)

[📌 Syntax & Explanation 22](#_Toc193220309)

[📊 Performance Analysis 23](#_Toc193220310)

[🤔 Comparison with Traditional Assignment 23](#_Toc193220311)

[⚠️ Potential Pitfalls & Considerations 23](#_Toc193220312)

[CallerArgumentExpression Attribute 24](#_Toc193220313)

[📝 Introduction 24](#_Toc193220314)

[🚀 Benefits & Use Cases 24](#_Toc193220315)

[📌 Syntax & Explanation 24](#_Toc193220316)

[📊 Performance Analysis 25](#_Toc193220317)

[🤔 Comparison with Traditional Approach 25](#_Toc193220318)

[⚠️ Potential Pitfalls & Considerations 25](#_Toc193220319)

# **C# 10.0 Features**

## Records Structs

### 📝 Introduction

C# 10 introduced **record structs**, expanding on the **record** concept introduced in C# 9. While records in C# 9 were **reference types**, record structs are **value types**, providing a performance-efficient way to create immutable data models.

### 🚀 Benefits & Use Cases

* **Ensures immutability**: Prevents unintended modifications to data.
* **Value-based equality**: Compares content instead of references.
* **Concise syntax**: Reduces boilerplate code for data models.
* **Performance**: Stored on the stack (unlike reference types), reducing heap allocations.
* **Supports with expressions**: Enables easy modification without changing the original instance.

**Real-world use cases**

* Data Transfer Objects (DTOs)
* Configuration settings
* Immutable data models
* Lightweight performance-critical objects

### 📌 Syntax & Explanation

#### **Defining a Record Struct**

public record struct Point(int X, int Y);

* + record struct defines a **value-type** record.
  + Properties are **readonly by default**, making the struct immutable.

#### **Creating and Using Record Structs**

var p1 = new Point(5, 10);

var p2 = new Point(5, 10);

Console.WriteLine(p1 == p2); // True (Value-based equality)

* Two Point instances with the same values are considered **equal**.

#### **With-Expressions**

var updatedPoint = p1 with { X = 20 };

* Creates a **new** instance with modified values, preserving immutability.

#### **Mutable Record Structs**

If mutability is required, use mutable record struct:

public record struct MutablePoint

{

public int X { get; set; }

public int Y { get; set; }

}

* This allows modifications after creation:

var mp = new MutablePoint { X = 5, Y = 10 };

mp.X = 15; // Allowed

#### **Inheritance in Record Structs**

Unlike class-based records, **record structs do not support inheritance**. This is because **structs in C# do not support inheritance**.

### 📊 Performance Analysis

|  |  |  |
| --- | --- | --- |
| **Feature** | **Record Struct** | **Record (Class)** |
| **Type** | Value Type | Reference Type |
| **Memory Location** | Stack | Heap |
| **Equality** | Value-based | Value-based |
| **Immutability** | Immutable by default | Immutable by default |
| **Performance** | Better for small data models | More flexible but more overhead |

### 🧐 Comparison with Classes & Record Classes

**Advantages of Record Structs:**

* Value-based equality ensures predictable comparisons.
* Immutable by default, preventing unintended modifications.
* Stack allocation improves performance for small data models.
* Concise syntax reduces boilerplate code.
* Supports with expressions for easy modifications.

**Disadvantages of Record Structs:**

* No inheritance support (like all structs).
* Shallow copies by default (deep copying requires custom logic).
* Cannot have **parameterless constructors**.

**When to Use Record Structs?**

* When you need an **immutable, lightweight data structure**.
* When working with **performance-critical applications** that benefit from stack allocation.
* For DTOs and configuration settings where **value-based equality** is essential.

### ⚠️ Potential Pitfalls & Considerations

* **No Inheritance**: Unlike class-based records, record structs do **not** support inheritance.
* **Shallow Copies for Nested Objects**: If a record struct contains reference types, they are **not deeply copied**.
* **Performance Overhead for Large Objects**: Since record structs are copied by value, they may have performance overhead if used improperly.
* **No Default Constructor**: Record structs do **not** support parameterless constructors.

## ****Global Using Directives****

### ****📝 Introduction****

C# 10 introduced **Global Using Directives**, allowing using directives to be declared globally, reducing redundancy across multiple files. This feature simplifies project-wide namespace imports, improving maintainability and readability.

### ****🚀 Benefits & Use Cases****

**✅ Benefits:**

* **Reduces Boilerplate Code**: Eliminates repetitive using statements in every file.
* **Improves Code Maintainability**: Centralized management of imports.
* **Enhances Readability**: Declutters individual source files.
* **Better Organization**: Encourages consistent namespace usage across the project.

**💡 Use Cases:**

* Common **system namespaces** (System, System.Linq, System.Collections.Generic).
* **Framework-specific** namespaces (e.g., Microsoft.AspNetCore.Mvc).
* **Logging, Dependency Injection, or ORM Libraries** (Serilog, Entity Framework).
* **Unit Testing** namespaces (XUnit, MSTest).

### ****📌 Syntax & Explanation****

#### **Defining a Global Using Directive**

Global using directives must be declared in a .cs file:

// GlobalUsings.cs

global using System;

global using System.Collections.Generic;

global using System.Linq;

* global using applies the directive to **all files** in the project.
* Typically, these are placed in a **dedicated file** (e.g., GlobalUsings.cs).

#### **Using Global Usings in Other Files**

After defining global usings, they are automatically available in all files without needing explicit imports:

// SomeFile.cs (No need to declare `using System;` explicitly)

Console.WriteLine("Hello, C# 10!");

#### **Conditional Compilation for Different Environments**

You can include or exclude global usings based on build configurations:

#if DEBUG

global using MyApp.Debugging;

#endif

### ****📊 Performance & Maintainability****

|  |  |  |
| --- | --- | --- |
| **Feature** | **Traditional Using** | **Global Using** |
| Scope | File-specific | Project-wide |
| Readability | More cluttered | Cleaner |
| Maintainability | Requires changes in multiple files | Centralized management |
| Performance | No impact | No impact |

### ****🧐 Best Practices & Considerations****

* Use **a single file** (e.g., GlobalUsings.cs) to organize global usings.
* Group **commonly used namespaces** (e.g., system, frameworks, utilities).
* Apply **conditional directives** for environment-specific namespaces.

### ⚠️ ****Potential Pitfalls:****

* **Namespace Conflicts**: Ensure different libraries don't introduce conflicts.
* **Overuse**: Avoid excessive global usings, which may lead to unintended dependencies.
* **Implicit Dependencies**: Team members may not realize where imports come from.

## File-Scoped Namespace Declaration

### 📝 Introduction

C# 10 introduced **file-scoped namespace declarations** to simplify the syntax for defining namespaces. Previously, all namespace declarations required curly braces, leading to unnecessary indentation. This new feature reduces indentation and improves readability.

### 🚀 Benefits & Use Cases

* **Reduces Boilerplate Code**: Eliminates unnecessary indentation caused by the traditional namespace declaration.
* **Improves Readability**: Makes code cleaner and easier to maintain.
* **Enhances Consistency**: Encourages a uniform structure across files.
* **Best for Single-Namespace Files**: Works well when a file contains only one namespace.

### 📌 Syntax & Explanation

**• Traditional Namespace Declaration (Before C# 10)**

namespace MyNamespace

{

public class MyClass

{

public void MyMethod()

{

Console.WriteLine("Hello, World!");

}

}

}

**• File-Scoped Namespace Declaration (C# 10)**

namespace MyNamespace;

public class MyClass

{

public void MyMethod()

{

Console.WriteLine("Hello, World!");

}

}

* **Key Differences:**
* No curly braces {} after the namespace.
* Reduces indentation for all elements inside the namespace.
* Works only if the entire file belongs to a single namespace.

### 📊 Performance Analysis

* **Memory Efficiency**: No impact on performance.
* **Code Readability**: Improved due to reduced nesting.
* **Consistency**: Encourages a standard format for single-namespace files.

### 🧐 Comparison with Traditional Namespaces

|  |  |  |
| --- | --- | --- |
| **Feature** | **File-Scoped Namespace** | **Traditional Namespace** |
| Syntax Complexity | Simpler | More verbose |
| Indentation Level | Reduced | Increased |
| Readability | Improved | Moderate |
| Multiple Namespaces | Not Supported | Supported |
| Nesting Allowed? | No | Yes |

### ⚠️ Potential Pitfalls & Considerations

* **No Multiple Namespaces in a File**: Unlike traditional namespaces, you cannot declare multiple namespaces in a single file.
* **Backward Compatibility**: Codebases using older C# versions may require refactoring to adopt file-scoped namespaces.
* **Not Ideal for Complex Hierarchies**: If a file requires multiple namespace definitions, traditional namespaces are still necessary.

Const Strings Initialized Using String Interpolation

📝 Introduction  
In C# 10, an enhancement has been made that allows const strings to be initialized using string interpolation, **provided that all placeholders within the string interpolation are themselves constant values**. This feature simplifies the creation of constant strings while leveraging the power of string interpolation.

🚀 Benefits & Use Cases

**•** **Simplifies String Construction:** You can now build constant strings dynamically by using string interpolation, while keeping all the values constant.  
**• Improves Readability:** String interpolation is a cleaner and more readable way to concatenate strings compared to traditional concatenation methods.  
**• Reduces Boilerplate Code:** Instead of concatenating constants manually, you can directly interpolate them into a single const string.  
**• Best for Simple Concatenation:** Works best when all interpolated values are constants.

### 📌 Syntax & Explanation

**• Traditional Constant String Declaration (Before C# 10)**

const string firstName = "John";

const string lastName = "Doe";

const string fullName = firstName + " " + lastName; // Concatenation

**• Constant String with Interpolation (C# 10)**

const string firstName = "John";

const string lastName = "Doe";

const string fullName = $"{firstName} {lastName}"; // Interpolation with constants

**• Key Differences:**

* In C# 10, string interpolation can be used directly in the const string declaration if all interpolated values are constants.
* This makes code more concise and readable, especially when working with multiple constants.

📊 Performance Analysis

**• Memory Efficiency:** No impact on memory efficiency, as the string is still evaluated at compile time.  
**• Code Readability:** Improved due to cleaner syntax compared to concatenation.  
**• Consistency:** Helps standardize the approach of constructing constant strings, reducing the chance of errors.

### 🧐 Comparison with Traditional Constants

|  |  |  |
| --- | --- | --- |
| **Feature** | **Interpolated Const String** | **Traditional Const String** |
| **String Construction** | Uses interpolation | Uses concatenation |
| **Readability** | More readable | Slightly more verbose |
| **Dynamic Values** | All placeholders must be constant | Can concatenate non-constants |
| **Use Case** | Best for simple concatenation with constants | Works in all cases, including dynamic values |

⚠️ Potential Pitfalls & Considerations

* **All Placeholders Must Be Constants**: String interpolation with const works only if all placeholders are constants. Any non-constant placeholder will cause a compile-time error.
* **Backward Compatibility**: Codebases using older C# versions will not support this feature, requiring updates for compatibility.
* **Limited Use for Dynamic Values**: If any of the interpolated values are dynamic or evaluated at runtime, you cannot use string interpolation with const.

## Lambda Improvements

### 📝 Introduction

In C# 10, there have been several improvements made to the use of lambda expressions, enhancing their flexibility and simplifying common tasks in functional programming. These improvements focus on extending the power and expressiveness of lambda expressions, making them more efficient and easier to use in modern C# applications.

### 🚀 Benefits & Use Cases

* **Improved Readability**: Simplifies lambda expressions by allowing more concise syntax and reducing boilerplate code.
* **More Flexible**: The changes allow lambda expressions to be used in a wider range of scenarios, such as with generic types and pattern matching.
* **Enhanced Performance**: Optimizations in how lambdas are compiled can lead to improved performance, particularly in scenarios where lambda expressions are frequently invoked.
* **Easier Debugging**: New syntax and features help developers debug lambdas more easily, improving the development workflow.

### 📌 Syntax & Explanation

* **Lambda Expression Before C# 10**

Func<int, int, int> add = (x, y) => x + y;

**• Lambda Expression in C# 10 (New Features)**

* Improved Target-Typed Lambda Expressions

Func<int, int> square = x => x \* x; // Lambda type inferred from the target delegate

* Lambda with async keyword support

Func<Task<int>> fetchDataAsync = async () => await FetchData();

* Lambda with is Pattern Matching

Func<object, bool> checkString = x => x is string str && str.Length > 5;

**• Key Differences:**

* In C# 10, lambda expressions are more flexible and concise, allowing for advanced scenarios like asynchronous operations and pattern matching.
* The new syntax reduces the need for boilerplate code and allows for better type inference.

### 📊 Performance Analysis

* **Memory Efficiency**: The optimizations to lambda expressions in C# 10 do not have a significant impact on memory usage but improve runtime performance, especially in high-frequency lambda executions.
* **Execution Speed**: The performance improvements focus on reducing overhead when invoking lambda expressions, leading to faster execution.
* **Code Maintainability**: The cleaner syntax makes the codebase more maintainable and readable, reducing the chance of errors.

### 🧐 Comparison with Pre-C# 10 Lambdas

|  |  |  |
| --- | --- | --- |
| **Feature** | **Lambda Expression in C# 10** | **Lambda Expression Before C# 10** |
| Syntax | More concise and flexible | Requires more verbose code |
| Target Typing | Improved inference with fewer types specified | Manual type declarations needed |
| Asynchronous Support | Direct support for async lambdas | Workaround needed for async |
| Pattern Matching Support | Native support in lambdas | No support |

### ⚠️ Potential Pitfalls & Considerations

* **Backward Compatibility**: Older versions of C# will not support these new lambda features, so some codebases may need to be updated to take full advantage of these improvements.
* **Complexity with Pattern Matching**: While powerful, using pattern matching in lambdas can make expressions more complex and harder to debug for less experienced developers.
* **Performance in Large-Scale Applications**: While there are performance improvements, developers must still carefully optimize lambda expressions in performance-critical areas, especially when working with large datasets or highly concurrent applications.

## Improvements of Structure Types

📝 Introduction

C# 10 introduces several improvements to structure types, which enhance their usability and performance. These changes aim to make working with structs more efficient and flexible, while preserving the value-type characteristics that make structs ideal for high-performance scenarios.

### 🚀 Benefits & Use Cases

* **Memory Efficiency**: Structs in C# are value types, meaning they are stored on the stack, which can lead to better memory usage and faster access in scenarios with small, immutable data types.
* **Performance Improvements**: Structs are now more optimized for performance, particularly in terms of memory allocation and access speed.
* **Simplified Code**: New features allow for cleaner code when dealing with structs, reducing the need for manual initialization and other boilerplate code.
* **Better Interoperability**: The enhancements also improve interoperability with other .NET APIs that rely on structs, making them easier to use in various libraries and frameworks.

### 📌 Syntax & Explanation

* **Prior to C# 10**  
  Structs in earlier versions of C# required explicit initialization for members, and certain features like default struct constructors were not supported. A typical struct declaration would look like this:

public struct Point

{

public int X;

public int Y;

public Point(int x, int y)

{

X = x;

Y = y;

}

}

* **Improvements in C# 10**
* Global Usability of default Parameterless Constructor  
  C# 10 allows structs to have a parameterless constructor that initializes fields to their default values, making code cleaner and reducing the need for custom constructors.

public struct Point

{

public int X { get; set; }

public int Y { get; set; }

// The default constructor is automatically provided.

}

* **Improvements in Structs with readonly Modifier**  
  The readonly modifier has been improved to support greater immutability, meaning structs can be passed around without risk of modification while preserving performance benefits.

public readonly struct Point

{

public int X { get; }

public int Y { get; }

public Point(int x, int y)

{

X = x;

Y = y;

}

}

* **Support for static Members**  
  In C# 10, structs can now include static members, such as static fields or methods. This allows for utility functions or shared values across all instances of a struct.

public struct Point

{

public int X;

public int Y;

public static Point Origin = new Point(0, 0);

public static void PrintOrigin()

{

Console.WriteLine($"Origin: {Origin.X}, {Origin.Y}");

}

}

**• Key Differences:**

* C# 10 brings built-in parameterless constructors and enhanced readonly support to structs, making them easier to use and more powerful in functional and performance-driven scenarios.
* The addition of static members allows for the creation of utility-based structs that were previously cumbersome.

### 📊 Performance Analysis

* **Memory Efficiency**: Structs are still value types, meaning they’re allocated on the stack or inline in arrays, avoiding the overhead of heap allocation. The improvements reduce unnecessary memory allocations, especially in complex data structures.
* **Execution Speed**: The optimizations allow structs to be more performant when dealing with large collections, reducing the need for boxing/unboxing and making them more efficient for high-performance computing.
* **Code Simplification**: With the ability to have parameterless constructors and static members, structs become much easier to use in common scenarios, reducing boilerplate code and improving code quality.

### 🧐 Comparison with Pre-C# 10 Structs

|  |  |  |
| --- | --- | --- |
| **Feature** | **Struct in C# 10** | **Struct Before C# 10** |
| Parameterless Constructors | Automatically provided | Manually defined |
| Static Members | Supported | Not supported |
| Readonly Fields | Enhanced support for immutability | Basic support |
| Code Readability | Cleaner with default constructors | More verbose |

### ⚠️ Potential Pitfalls & Considerations

* **Boxing Issues**: While structs are efficient for small, immutable types, they can still suffer from performance penalties if boxed or used with non-value-type collections.
* **Struct Size Limitation**: Structs should remain small to avoid unnecessary memory overhead. Large structs can become inefficient due to the copy-by-value behavior of structs.
* **Immutability Assumption**: When marking structs as readonly, developers need to be aware that while the struct itself is immutable, mutable fields inside the struct are still allowed unless explicitly controlled.
* **Static Members**: The introduction of static members can make it easier to manage shared data within structs, but developers should be cautious to avoid misuse, especially with stateful static data.

## Extended Property Patterns

📝 Introduction  
In C# 10, **Extended Property Patterns** were introduced to simplify the pattern matching process for deeply nested objects. With this feature, you can match and access properties of nested objects directly within a pattern, reducing boilerplate code and improving readability. This feature is particularly useful when working with complex object structures where you need to check and extract values from nested properties.

### 🚀 Benefits & Use Cases

* **Simplified Pattern Matching**: Allows you to match properties of nested objects in a single statement, making your code cleaner and easier to understand.
* **Improved Readability**: Replaces multiple null checks and property accesses with a single, readable pattern matching expression.
* **Reduces Boilerplate Code**: Avoids manually checking each property in the object hierarchy, resulting in more concise and efficient code.
* **Use Case**: Best for scenarios where you need to match and access properties of deeply nested objects without excessive checks.

### 📌 Syntax & Explanation

* **Traditional Approach (Before C# 10)**  
  Before C# 10, if you wanted to match properties of a nested object, you would typically need multiple checks:

if (vehicle != null && vehicle.Engine != null)

{

var model = vehicle.Engine.Model;

var horsePower = vehicle.Engine.HorsePower;

// Further logic...

}

* **Extended Property Pattern (C# 10)**  
  With C# 10, you can directly access nested properties within a pattern:

if (vehicle is Vehicle { Engine: { Model: var model, HorsePower: var horsePower } })

{

// Access and use model and horsePower directly

}

* **Key Differences:**
  + In C# 10, nested properties can be matched directly within the pattern using extended property patterns.
  + This approach simplifies code and enhances readability by removing the need for multiple null checks.

### 📊 Performance Analysis

* **Memory Efficiency**: The memory consumption is not impacted, as the evaluation happens at compile-time, similar to regular pattern matching.
* **Code Readability**: Significant improvement in readability due to a more concise and expressive syntax.
* **Consistency**: Standardizes the way nested properties are accessed, making code more predictable and reducing the chance of errors.

### 🧐 Comparison with Traditional Property Matching

|  |  |  |
| --- | --- | --- |
| **Feature** | **Extended Property Patterns** | **Traditional Property Matching** |
| Matching Nested Properties | Directly matches nested properties in the pattern | Requires multiple checks (null checks, property accesses) |
| Readability | More readable and concise | Slightly more verbose and less readable |
| Use Case | Ideal for matching and accessing properties of deeply nested objects | Suitable for cases where the object hierarchy may vary or be dynamic |

### ⚠️ Potential Pitfalls & Considerations

* **All Nested Properties Must Be Available**: If any property in the pattern is null or not available, the pattern matching will fail.
* **Compatibility with Older Versions**: This feature is only available in C# 10, so older codebases using earlier versions of C# may require updates for compatibility.
* **Limited Use for Dynamic Properties**: The extended property pattern works best when the properties being matched are statically known at compile time. If any property is dynamic or evaluated at runtime, it may not be usable in a pattern.

## Sealed Modifier When Overriding ToString In a Record Type

### 📝 Introduction

In C# 10, you can now add the sealed modifier when overriding the ToString method in a record type. This allows you to prevent further overriding of the ToString method in derived record types, making the method final in the hierarchy. This feature ensures that the ToString implementation remains consistent and cannot be changed by any subclass.

### 🚀 Benefits & Use Cases

* **Prevents Further Overriding**: By marking the overridden ToString method as sealed, you prevent any derived classes from overriding it again, ensuring consistency in how ToString is implemented across your application.
* **Ensures Integrity**: This can be especially useful when you want to lock the behavior of the ToString method for a record type, ensuring that its string representation remains fixed and consistent.
* **Simplifies Code Maintenance**: When you are certain that the string representation of the object should not be altered in derived classes, using sealed reduces the risk of accidental changes or inconsistencies in derived classes.

### 📌 Syntax & Explanation

* **Before C# 10**  
  In previous versions of C#, the ToString method could be overridden in derived types without restriction. There was no way to prevent this override.
* **C# 10 with sealed modifier**  
  With C# 10, you can now use the sealed modifier when overriding the ToString method in a record type, like this:

public record Person(string Name, int Age)

{

// Sealed ToString method in the record type

public sealed override string ToString() => $"{Name}, Age: {Age}";

}

public record Employee(string Name, int Age, string Position) : Person(Name, Age)

{

// Error: Cannot override ToString because it is sealed in the base record

// public override string ToString() => $"{Name}, Age: {Age}, Position: {Position}";

}

* **Key Differences:**
  + The sealed modifier prevents any further overrides of ToString in derived types, unlike regular methods that can be overridden freely.
  + This is only applicable to the ToString method in record types, ensuring that the string representation remains fixed.

### 📊 Performance Analysis

* **Memory Efficiency**: There is no significant impact on memory efficiency as this change is purely syntactical and does not affect runtime performance.
* **Code Readability**: Helps maintain readability by ensuring that the string representation of an object is fixed and unchangeable, reducing the chances of confusion or inconsistency in subclasses.
* **Consistency**: Adds a layer of consistency, making sure that derived records will not accidentally modify the string representation of the base record.

### 🧐 Comparison with Traditional Method Override

|  |  |  |
| --- | --- | --- |
| **Feature** | **Sealed ToString in Record** | **Traditional Method Override** |
| Override Flexibility | Prevents further overrides | Can be freely overridden |
| Use Case | Ensures consistent string representation | Flexible for custom implementations |
| Code Maintenance | Simplifies maintenance by locking the ToString behavior | Requires extra caution in derived classes |

### ⚠️ Potential Pitfalls & Considerations

* **Limited to Record Types**: The sealed modifier on ToString is specific to record types in C# 10. This feature does not apply to other types (like classes or structs).
* **No Customization in Derived Types**: Once you mark the ToString method as sealed, derived types cannot change the string representation, which may limit flexibility in certain cases.
* **Backward Compatibility**: Codebases using earlier versions of C# will not support this feature and may require updates for compatibility.

## Allow both assignment and declaration in the same deconstruction.

### 📝 ****Introduction****

In C# 10, deconstruction has been improved to allow both **assignment and declaration** in the same deconstruction statement. This makes code more concise and reduces redundancy when working with tuples and record types. Previously, deconstruction required separate variable declarations or assignments, making the syntax slightly more verbose.

### 🚀 ****Benefits & Use Cases****

* **Simplifies Code:** Reduces the number of lines needed for deconstructing values.
* **Improves Readability:** Makes code cleaner and more maintainable.
* **Eliminates Redundancy:** Allows variable declaration and assignment in one step.
* **Useful for Pattern Matching:** Enhances usage with records and tuples, making them easier to work with in functional-style programming.

### 📌 ****Syntax & Explanation****

**Before C# 10**

Prior to C# 10, you had to **declare** variables first and then assign them using deconstruction:

var person = ("Alice", 30);

string name;

int age;

(name, age) = person; // Separate declaration and assignment

Console.WriteLine($"Name: {name}, Age: {age}");

**C# 10 with Combined Declaration & Assignment**

Now, you can **declare and assign** variables in a single deconstruction statement:

var person = ("Alice", 30);

(string name, int age) = person; // Declaration and assignment combined

Console.WriteLine($"Name: {name}, Age: {age}");

**Example with Record Types**

public record Person(string Name, int Age);

class Program

{

static void Main()

{

Person person = new("Bob", 25);

var (name, age) = person; // Deconstructing a record

Console.WriteLine($"Name: {name}, Age: {age}");

}

}

**Key Differences:**

* You **no longer** need to declare variables separately before assignment.
* The syntax is **cleaner** and **more readable**.
* Works with **tuples, records, and custom deconstruction methods**.

### 📊 ****Performance Analysis****

|  |  |  |
| --- | --- | --- |
| **Feature** | **C# 10 Deconstruction** | **Traditional Deconstruction** |
| Syntax Complexity | Simpler and more concise | Requires separate declarations |
| Readability | Improved | Less readable due to extra lines |
| Flexibility | Works seamlessly with tuples and records | Requires more code |

### 🤔 ****Comparison with Traditional Assignment****

* **Before C# 10: Requires explicit declaration before deconstruction.**
* **In C# 10: Allows declaring and assigning variables in a single step.**

### ⚠️ ****Potential Pitfalls & Considerations****

* **Scope Consideration:** When deconstructing inside methods or loops, ensure the declared variables do not conflict with existing ones.
* **Backward Compatibility:** Older C# versions do not support this feature, so upgrading to C# 10+ is necessary.
* **Custom Deconstruction Methods:** Ensure record types implement proper deconstruction methods if you plan to use this feature extensively.

## CallerArgumentExpression Attribute

### 📝 Introduction

C# 10 introduced the CallerArgumentExpression attribute, which allows developers to capture the string representation of an argument passed to a method. This feature enhances debugging and logging by enabling more informative error messages without requiring explicit argument names.

### 🚀 Benefits & Use Cases

* **Improved Debugging**: Helps capture argument expressions for better debugging and logging.
* **Better Error Messages**: Provides more meaningful error messages by including the exact argument expression.
* **Reduces Boilerplate Code**: Eliminates the need to manually pass argument names for logging or validation purposes.

### 📌 Syntax & Explanation

• **Before C# 10**, capturing an argument expression required manually passing the argument name as a string.

public static void ValidateNotNull(object obj, string paramName)

{

if (obj == null)

{

throw new ArgumentNullException(paramName, "Argument cannot be null.");

}

}

ValidateNotNull(myVar, nameof(myVar));

* **C# 10 with CallerArgumentExpression** With C# 10, the CallerArgumentExpression attribute automatically captures the argument expression without requiring a manual string parameter.

using System;

using System.Runtime.CompilerServices;

public static class Validator

{

public static void ValidateNotNull(

object obj,

[CallerArgumentExpression("obj")] string paramName = "")

{

if (obj == null)

{

throw new ArgumentNullException(paramName, $"Argument '{paramName}' cannot be null.");

}

}

}

class Program

{

static void Main()

{

string? myVar = null;

ValidateNotNull(myVar);

}

}

**• Key Differences:**

* The CallerArgumentExpression attribute automatically captures the argument expression, reducing redundancy.
* Developers no longer need to explicitly pass the argument name.

### 📊 Performance Analysis

* **Memory Efficiency**: No significant impact, as it primarily affects compile-time metadata.
* **Code Readability**: Improved, as developers can now omit manually passing argument names.
* **Consistency**: Helps ensure standardized logging and error messages across projects.

### 🤔 Comparison with Traditional Approach

|  |  |  |
| --- | --- | --- |
| **Feature** | **CallerArgumentExpression** | **Manual Argument Name** |
| Argument Name Capturing | Automatic | Requires explicit nameof() |
| Readability | Improved | Slightly verbose |
| Use Case | Debugging, Logging, Validation | Logging, Debugging |
| Code Simplicity | Less boilerplate | Requires additional argument |

### ⚠️ Potential Pitfalls & Considerations

* **Limited to Method Parameters**: The attribute only works for method arguments, not local variables or fields.
* **Backward Compatibility**: Requires C# 10 or later; older versions do not support this feature.
* **Compiler Support Needed**: Ensure your project is targeting .NET 6 or later to use this feature effectively.