Table of Contents

[**📄 C# 9.0 Features** 3](#_Toc193197727)

[Records 3](#_Toc193197728)

[📝 Introduction 3](#_Toc193197729)

[🚀 Benefits & Use Cases 3](#_Toc193197730)

[📌 Syntax & Explanation 3](#_Toc193197731)

[📊 Performance Analysis 4](#_Toc193197732)

[🧐 Comparison with Classes 4](#_Toc193197733)

[📌 Advantages & Disadvantages: Class vs. Record 4](#_Toc193197734)

[⚠️ Potential Pitfalls with Records in C# 9.0 5](#_Toc193197735)

[Init-only Setters 6](#_Toc193197736)

[📝 Introduction 6](#_Toc193197737)

[🔑 Key Changes of Init-only Setters 6](#_Toc193197738)

[🚀 Benefits & Use Cases 6](#_Toc193197739)

[📌 Syntax & Explanation 6](#_Toc193197740)

[📊 Performance Analysis 7](#_Toc193197741)

[🧐 Comparison with Older Approaches 7](#_Toc193197742)

[⚠️ Potential Pitfalls 8](#_Toc193197743)

[🚀 Use Case Scenario 8](#_Toc193197744)

[Top-Level Statements Documentation 9](#_Toc193197745)

[📝 Introduction 9](#_Toc193197746)

[🔑 Key Changes of Top-Level Statements 9](#_Toc193197747)

[📌 Syntax & Explanation 9](#_Toc193197748)

[📊 Comparison with Traditional Approach 10](#_Toc193197749)

[⚠️ Potential Pitfalls 10](#_Toc193197750)

[Pattern matching enhancements: relational patterns and logical patterns 11](#_Toc193197751)

[📝 Introduction 11](#_Toc193197752)

[🔑 Key Changes in Pattern Matching Enhancements 11](#_Toc193197753)

[📌 Syntax & Explanation 11](#_Toc193197754)

[📊 Comparison with Traditional Approach 12](#_Toc193197755)

[⚠️ Potential Pitfalls 12](#_Toc193197756)

[Performance and interop 13](#_Toc193197757)

[📝 Introduction 13](#_Toc193197758)

[🔑 Key Features of Performance and Interop Enhancements 13](#_Toc193197759)

[🚀 Benefits & Use Cases 13](#_Toc193197760)

[📊 Performance Analysis 15](#_Toc193197761)

[🧐 Comparison with Previous Approaches 16](#_Toc193197762)

[⚠️ Potential Pitfalls 16](#_Toc193197763)

[Fit and Finish Features in C# 9.0 17](#_Toc193197764)

[📝 Introduction 17](#_Toc193197765)

[🔑 Key Features of Fit and Finish 17](#_Toc193197766)

[🚀 Benefits & Use Cases 18](#_Toc193197767)

[📊 Feature Analysis 20](#_Toc193197768)

[🧐 Comparison with Previous Approaches 21](#_Toc193197769)

[⚠️ Potential Pitfalls 21](#_Toc193197770)

# **📄 C# 9.0 Features**

## Records

### 📝 Introduction

Records were introduced in C# 9.0 to provide an immutable, value-based data structure that simplifies object modeling. They are primarily used for defining data models and DTOs (Data Transfer Objects). Unlike classes, records are reference types but behave like value types when comparing instances.

### 🚀 Benefits & Use Cases

* Ensures immutability: Prevents unintended modifications to data.
* Concise syntax: Reduces boilerplate code for data models.
* Value-based equality: Compares content instead of references.
* Supports with-expressions: Allows easy modification of objects without changing the original instance.

**• Real-world use cases:**

Ideal for DTOs, configuration settings, and result objects.

### 📌 Syntax & Explanation

public record Person(string Name, int Age);

* + The record keyword defines a record type.
  + Person is immutable; its properties cannot be modified after creation.

#### Creating and Using Records

var person1 = new Person("John", 30);

var person2 = new Person("John", 30);

Console.WriteLine(person1 == person2); // True (value-based equality)

#### With-Expressions

var updatedPerson = person1 with { Age = 31 };

* + The with expression creates a new object with modified values, preserving immutability.

#### Inheritance in Records

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

* Records support inheritance, enabling efficient data modeling.

### 📊 Performance Analysis

* Memory Efficiency: Optimized for storing immutable data.
* Comparison Speed: Faster comparisons due to value-based equality.
* Time Complexity: O(1) for field access, O(n) for deep equality comparison.

### 🧐 Comparison with Classes

|  |  |  |
| --- | --- | --- |
| **Feature** | **Record (C# 9.0)** | **Class** |
| Equality | Value-based | Reference-based |
| Immutability | Immutable by default | Mutable |
| Syntax | Concise | More verbose |
| Use Case | DTOs, Configs | Business logic, State management |

### 📌 Advantages & Disadvantages: Class vs. Record

#### Advantages of Records:

* Value-based equality ensures predictable behavior when comparing objects.
* Immutable by default, reducing unintended side effects.
* Concise syntax eliminates boilerplate code.
* Ideal for immutable data models and DTOs.
* Supports with expressions for easy modifications.

#### Disadvantages of Records:

* Cannot modify properties after creation (unless explicitly designed as mutable).
* Not suitable for scenarios requiring mutable state or business logic.
* Deep copies require manual implementation for nested objects.

#### Advantages of Classes:

* Allows mutable state, making it more flexible for complex applications.
* Suitable for encapsulating business logic and state management.
* Supports reference-based equality, useful in scenarios where identity matters.
* More control over memory management and object lifecycle.

#### Disadvantages of Classes:

* Requires more boilerplate code for implementing value-based equality.
* Mutable by default, increasing the risk of unintended side effects.
* More verbose syntax compared to records.

### ⚠️ Potential Pitfalls with Records in C# 9.0

* **Limited Mutability:**  
  Records are immutable by default, which may not be suitable for scenarios requiring frequent modifications. You can use init accessors for some mutability but it goes against the immutability design.
* **Inheritance Complexity:**  
  Inheritance in records can cause issues with value-based equality, leading to unexpected results with deep hierarchies.
* **Shallow Copies for Nested Objects:**  
  Records do shallow copies by default, requiring custom logic for deep cloning of nested objects.
* **Performance Overhead for Complex Comparisons:**  
  Deep equality checks on complex records can introduce performance issues. Avoid frequent deep comparisons in performance-critical code.
* **with Expressions and Nested Records:**  
  With expressions may complicate nested record modifications, requiring careful handling for immutability.
* **Limited Support for Advanced Features:**  
  Records lack features like custom constructors and destructors, making them unsuitable for some use cases.
* **Serialization Issues:**  
  Some serialization libraries may not fully support records, especially with init properties, requiring configuration adjustments.

## Init-only Setters

### 📝 ****Introduction****

In C# 9.0, **init-only setters** were introduced, allowing properties to be initialized only during object creation (i.e., at the time of object instantiation). This feature enhances immutability while keeping the syntax clean and user-friendly.

### 🔑 ****Key Changes of Init-only Setters****

* **Init-only properties:** Properties that can only be set during object initialization, making them read-only afterward.
* **Object Initialization Syntax:** C# 9.0 allows for the use of the new init keyword, which provides the ability to set a property only once during object initialization.

### 🚀 ****Benefits & Use Cases****

* **Immutability:** It enforces immutability of properties after object creation, allowing modifications only when constructing the object.
* **Cleaner Syntax:** It eliminates the need for custom setters while maintaining flexibility.
* **Type Safety:** The property can still be part of an object but can be safely modified only during initialization.

📌 **Real-World Use Cases**

* **DTOs (Data Transfer Objects):** Init-only properties are perfect for DTOs, where values need to be set once and should remain constant throughout the object's lifecycle.
* **Immutable Object Patterns:** For libraries or APIs that enforce immutable patterns, init setters are an excellent fit for ensuring that properties can only be set at creation.

### 📌 ****Syntax & Explanation****

* **Traditional Property Setter (Before C# 9.0):**

public class Person

{

public string Name { get; set; }

public int Age { get; set; }

}

The set method can be called anytime after the object is created.

* **Init-only Setter (C# 9.0):**

public class Person

{

public string Name { get; init; }

public int Age { get; init; }

}

The init keyword ensures that the properties can only be set during object initialization, either by using an object initializer or a constructor.

* **Usage Example:**

var person = new Person { Name = "John", Age = 30 };

// person.Name = "Doe"; // Error: Cannot assign to 'Name' because it is an 'init' only property.

The properties Name and Age can only be set when the object is created and cannot be modified afterward.

### 📊 ****Performance Analysis****

* **Performance:** There is no noticeable performance cost when using init properties. The only difference is the restriction on setting the properties after object initialization.
* **Memory Efficiency:** The memory footprint is similar to that of a class with regular properties; however, since the properties cannot be changed after initialization, there is reduced risk of data inconsistency.

### 🧐 ****Comparison with Older Approaches****

|  |  |  |
| --- | --- | --- |
| **Feature** | **Before C# 9.0** | **After C# 9.0** |
| **Property Setter** | Can be modified anytime | Can only be set during object initialization |
| **Read-Only Behavior** | Read-only only with readonly keyword or via private setters | Read-only via init keyword |
| **Mutability** | Mutable at any point | Immutable after object creation |

### ⚠️ ****Potential Pitfalls****

* **Cannot Modify After Initialization:** Since init properties can only be set during object initialization, they cannot be modified afterward. This is a benefit but also a limitation in cases where you need to update the property after the object is created.
* **Compatibility with Other Features:**
* **Ref and out Parameters:** init-only properties cannot be passed as ref or out parameters, as they are read-only after initialization.
* **Deconstruction and Copying:** You cannot modify the properties during deconstruction or copying of an object.

### 🚀 ****Use Case Scenario****

Suppose you're building an immutable configuration object:

public class Configuration

{

public string AppName { get; init; }

public string Version { get; init; }

}

var config = new Configuration { AppName = "MyApp", Version = "1.0" };

// config.AppName = "NewApp"; // Error: Cannot modify after initialization.

This guarantees that the AppName and Version are immutable after initialization, ensuring the integrity of the Configuration object.

## ****Top-Level Statements Documentation****

### 📝 ****Introduction****

C# 9.0 introduced **top-level statements**, allowing developers to write programs without explicitly defining a Main method inside a class. This feature simplifies the syntax for small programs, scripts, and educational purposes by reducing boilerplate code.

### 🔑 ****Key Changes of Top-Level Statements****

* Eliminates the need for a Main method.
* Removes unnecessary boilerplate code (static void Main and class definition).
* Automatically generates a Main method behind the scenes.
* Supports asynchronous execution (await can be used without an async method).
* Allows multiple statements, but only in a single file.

### ****📌 Syntax & Explanation****

#### **Traditional** Main **Method (Before C# 9.0)**

using System;

class Program

{

static void Main()

{

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

}

}

✔ Requires explicit Main method and class definition.

#### **Top-Level Statement (C# 9.0)**

using System;

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

✔ No need for a Main method or class.  
✔ The compiler automatically generates an entry point.

#### **Top-Level Statement with Variables**

using System;

int number = 10;

Console.WriteLine($"The number is {number}");

#### **Top-Level Statement with Methods**

Methods can still be defined, but they must be placed after all top-level statements.

using System;

Console.WriteLine(SayHello("Samrat"));

string SayHello(string name) => $"Hello, {name}!";

✔ SayHello is defined after the top-level statement.

#### **Asynchronous Top-Level Statements**

Top-level statements support async without requiring an explicit Main method.

using System;

using System.Net.Http;

using System.Threading.Tasks;

HttpClient client = new HttpClient();

string data = await client.GetStringAsync("https://example.com");

Console.WriteLine(data);

✔ await can be used directly without wrapping it in an async method.

### ****📊 Comparison with Traditional Approach****

|  |  |  |
| --- | --- | --- |
| **Feature** | **Before C# 9.0** | **C# 9.0 (Top-Level Statements)** |
| Boilerplate Code | Requires Main method and class | No Main method needed |
| Readability | More lines of code | Cleaner and simpler syntax |
| Async Support | Explicit async method needed | await can be used directly |
| Execution Flow | Defined inside Main | Starts from the first statement |

### ****⚠️ Potential Pitfalls****

1. **Only One File Can Have Top-Level Statements**
   * A project can have **only one file** with top-level statements.
   * If multiple files contain top-level statements, a compilation error occurs.
2. **Not Ideal for Large Applications**
   * For large applications, using explicit classes and methods is preferred for maintainability.
3. **Implicit Main Method**
   * The compiler still generates a Main method behind the scenes, even though it's not explicitly written.

Pattern matching enhancements: relational patterns and logical patterns

📝 Introduction

C# 9.0 introduced significant enhancements to pattern matching, improving code readability and expressiveness. The major additions include **relational patterns** and **logical patterns**, which simplify condition checking and make switch expressions more powerful.

🔑 Key Changes in Pattern Matching Enhancements

• **Relational patterns** allow using comparison operators (<, >, <=, >=) inside patterns.  
• **Logical patterns** introduce and, or, and not operators for better pattern combinations.  
• **Improved expressiveness** in switch expressions, reducing the need for multiple if-else statements.  
• **Combining relational and logical patterns** for more concise and readable condition checks.

### 📌 Syntax & Explanation

#### *Relational Patterns (New in C# 9.0)*

Allows direct comparison within patterns using <, >, <=, and >=.

**Traditional Approach (Before C# 9.0):**

static string ClassifyAge(int age)

{

if (age < 13) return "Child";

else if (age < 20) return "Teen";

else if (age < 65) return "Adult";

else return "Senior";

}

✔ Uses multiple if-else conditions.

**Using Relational Patterns (C# 9.0):**

static string ClassifyAge(int age) => age switch

{

< 13 => "Child",

< 20 => "Teen",

< 65 => "Adult",

\_ => "Senior"

};

✔ Simplifies condition checking by eliminating if-else statements.

#### • Logical Patterns (New in C# 9.0)

Supports and, or, and not operators inside patterns.

**Using or in C# 9.0:**

static bool IsWeekend(DayOfWeek day) => day is DayOfWeek.Saturday or DayOfWeek.Sunday;

✔ Simplifies checking multiple values in a pattern.

**Using and and not in C# 9.0:**

static string DescribeNumber(int number) => number switch

{

< 0 => "Negative",

>= 0 and <= 10 => "Small",

> 10 and < 100 => "Medium",

>= 100 => "Large"

};

✔ Combines multiple conditions in a cleaner way.

### 📊 Comparison with Traditional Approach

|  |  |  |
| --- | --- | --- |
| **Feature** | **Before C# 9.0** | **C# 9.0 (Pattern Matching Enhancements)** |
| **Comparison Operators** | Not allowed in switch | <, >, <=, >= supported |
| **Logical Operators** | if-else required | and, or, not available in switch |
| **Readability** | Verbose with if-else | Cleaner and more concise expressions |
| **Flexibility** | Limited | Can combine relational and logical patterns |

### ⚠️ Potential Pitfalls

1. **Cannot Use Complex Expressions in Patterns**
   * Expressions like x + 1 > 10 are not supported inside switch patterns.
2. **Only Works in Pattern Matching Contexts**
   * or, and, and not are **only** valid in pattern matching, not as general boolean operators.
3. **May Not Be Ideal for All Scenarios**
   * For very complex conditions, traditional if-else may still be more readable.

## Performance and interop

### 📝 Introduction

C# 9.0 introduced several features aimed at enhancing performance and interoperability, particularly for scenarios requiring low-level programming and native interop.

### 🔑 Key Features of Performance and Interop Enhancements

* + **Native Sized Integers (`nint` and `nuint`):** Integer types that adapt to the native size of the processor (32-bit or 64-bit), useful for low-level memory or interop operations.
  + **Function Pointers (`delegate\*` syntax):** Allow for unmanaged function pointers in C#, enabling direct calls to native functions without the overhead of delegates.
  + S**uppress Emitting `localsinit` Flag:** Provides control over method initialization by allowing developers to suppress the default zeroing out of local variables, potentially improving performance.
  + **Module Initializers:** Methods that run once when an assembly is loaded, useful for setting up state or performing initialization tasks at the module level.
  + **Enhanced Partial Methods:** Removal of previous restrictions on partial methods, allowing them to have access modifiers, return values, and attributes, thus providing more flexibility.

### 🚀 Benefits & Use Cases

* + **Native Sized Integers:** Useful in scenarios involving low-level memory manipulation or interop operations where the size of the integer needs to match the platform's pointer size.
  + **Function Pointers:** Beneficial for performance-critical applications and interop scenarios requiring direct invocation of unmanaged code.
  + **Suppress Emitting `localsinit` Flag:** Offers performance improvements in high-performance scenarios by eliminating the overhead of zeroing out local variables.
  + **Module Initializers:** Ideal for performing initialization tasks at the module level without relying on static constructors.
  + **Enhanced Partial Methods:** Facilitates more flexible and expressive designs in partial classes.

**📌 Real-World Use Cases**

1. **Native Sized Integers:**

nint pointerSizeInteger = 42;

1. Function Pointers**:**

unsafe

{

delegate\* unmanaged<int, void> functionPointer = &UnmanagedFunction;

functionPointer(42);

}

Enables direct calls to native functions without the overhead of delegates.

1. **Suppress Emitting localsinit Flag:**

[System.Runtime.CompilerServices.SkipLocalsInit]

void PerformanceCriticalMethod()

{

// Method implementation

}

Provides control over method initialization, potentially improving performance in high-performance scenarios.

1. **Module Initializers**:

using System.Runtime.CompilerServices;

class Program

{

[ModuleInitializer]

public static void Initialize()

{

// Initialization code

}

}

Useful for setting up state or performing initialization tasks at the module level without relying on static constructors.

1. **Enhanced Partial Methods**:

public partial class MyClass

{

public partial string GetDetails();

}

public partial class MyClass

{

public partial string GetDetails()

{

return "Details";

}

}

Allows partial methods to have access modifiers, return values, and attributes, providing more flexibility in partial classes.

### 📊 Performance Analysis

|  |  |
| --- | --- |
| **Feature** | **Benefit** |
| Native Sized Integers | Ensures integer sizes match the platform's pointer size, optimizing memory and performance. |
| Function Pointers | Reduces overhead by enabling direct calls to unmanaged code, improving execution speed. |
| Suppress Emitting localsinit | Eliminates unnecessary initialization of local variables, enhancing method execution performance. |
| Module Initializers | Allows for efficient module-level initialization without the need for static constructors. |
| Enhanced Partial Methods | Provides more flexibility and expressiveness in partial classes, improving code maintainability. |

### 🧐 Comparison with Previous Approaches

|  |  |  |
| --- | --- | --- |
| **Feature** | **Previous Approach** | **C# 9.0 Enhancement** |
| Native Sized Integers | Manual handling of integer sizes for different platforms. | Automatic adaptation to the native size of the processor with nint and nuint. |
| Function Pointers | Use of delegates with associated overhead. | Direct use of unmanaged function pointers with delegate\* syntax, reducing overhead. |
| Suppress Emitting localsinit | Automatic zeroing out of local variables, potentially adding overhead. | Option to suppress this behavior using [SkipLocalsInit], giving more control to developers. |
| Module Initializers | Initialization through static constructors or other mechanisms. | Direct module-level initialization with [ModuleInitializer] attribute. |
| Enhanced Partial Methods | Partial methods were limited in functionality, lacking access modifiers and return types. | Removal of restrictions, allowing access modifiers, return values, and attributes for partial methods. |

### ⚠️ Potential Pitfalls

* **Native Sized Integers**: Ensure that the use of nint and nuint aligns with the platform's architecture to avoid compatibility issues.
* **Function Pointers**: Using unmanaged function pointers requires careful handling to prevent memory corruption and ensure security.
* **Suppress Emitting localsinit Flag**: Suppressing the localsinit flag can lead to uninitialized variables, which may cause unpredictable behavior if not handled properly.
* **Module Initializers**: Overuse of module initializers can lead to complex initialization sequences that are hard to debug.
* **Enhanced Partial Methods**: While more flexible, enhanced partial methods require careful design to maintain code readability and maintainability.

## ****Fit and Finish Features in C# 9.0****

### ****📝 Introduction****

C# 9.0 introduces several "fit and finish" features, which aim to refine and streamline the language. These features enhance both the syntactic flexibility and expressiveness of C#, making it easier to write cleaner and more efficient code.

### ****🔑 Key Features of Fit and Finish****

* **Target-Typed New Expressions**  
  Simplifies object creation by inferring the type of an object based on the context, reducing redundancy in code.
* **Static Anonymous Functions**  
  Allows anonymous functions (lambdas) to be declared static, which eliminates the need for them to capture outer variables, improving performance.
* **Target-Typed Conditional Expressions**  
  Allows the type of the result in a conditional expression (?:) to be inferred from the context, reducing explicit type declarations.
* **Covariant Return Types**  
  Enables overriding a method with a return type that is more specific than the base method, allowing for more flexible and type-safe overrides.
* **Extension GetEnumerator Support for foreach Loops**  
  Introduces the ability to use a custom GetEnumerator method in extensions for foreach loops, providing more flexibility in iterating over collections.
* **Lambda Discard Parameters**  
  Allows unused parameters in lambda expressions to be marked as discards (\_), improving code clarity and reducing warnings for unused parameters.
* **Attributes on Local Functions**  
  Local functions in C# can now have attributes, which was previously not allowed. This feature increases the flexibility of method declarations.

### ****🚀 Benefits & Use Cases****

* **Target-Typed New Expressions**:  
  Reduces the need for repeating type information during object instantiation, making the code more concise and readable.
* **Static Anonymous Functions**:  
  Improves performance by preventing anonymous methods from capturing unnecessary outer variables, resulting in fewer allocations.
* **Target-Typed Conditional Expressions**:  
  Eliminates the need for specifying the type of the conditional expression when the type can be inferred, making the code more compact and intuitive.
* **Covariant Return Types**:  
  Improves the type safety of overridden methods, allowing derived classes to return more specific types without breaking base class contracts.
* **Extension GetEnumerator Support for foreach Loops**:  
  Extends the foreach loop functionality by allowing GetEnumerator methods in extension classes, enabling customized iteration behavior for non-standard collections.
* **Lambda Discard Parameters**:  
  Reduces clutter and warnings when lambda expressions have unused parameters, promoting cleaner code.

**📌 Real-World Use Cases**

1. **Target-Typed New Expressions:**

var myList = new List<int>(); // Type is inferred from the context

Simplifies object creation and enhances readability.

1. **Static Anonymous Functions:**

Func<int, int> square = static (x) => x \* x;

Helps to avoid unnecessary captures of outer variables, improving performance.

1. **Target-Typed Conditional Expressions:**

var result = condition ? new MyClass() : new MyOtherClass();

Simplifies conditional expressions by letting the type be inferred from the context.

1. **Covariant Return Types:**

public class Animal { }

public class Dog : Animal { }

public class AnimalShelter

{

public virtual Animal GetAnimal() => new Animal();

}

public class DogShelter : AnimalShelter

{

public override Dog GetAnimal() => new Dog(); // Covariant return type

}

Allows overriding methods with more specific return types, ensuring better type safety.

1. **Extension GetEnumerator Support for foreach Loops:**

public static class MyEnumerableExtensions

{

public static IEnumerator<int> GetEnumerator(this MyEnumerable source)

{

yield return 1;

yield return 2;

}

}

foreach (var item in myEnumerable)

{

Console.WriteLine(item);

}

Customizes the behavior of foreach by adding custom GetEnumerator functionality.

1. **Lambda Discard Parameters:**

Action<int, int> add = (\_, y) => Console.WriteLine(y);

Discards the first parameter in a lambda function, eliminating unnecessary warnings.

**Attributes on Local Functions:**

void Main()

{

[Obsolete]

void LocalFunction() { }

}

Adds attributes like [Obsolete] to local functions for greater flexibility.

### ****📊 Feature Analysis****

|  |  |
| --- | --- |
| **Feature** | **Benefit** |
| Target-Typed New Expressions | Reduces verbosity and improves readability by inferring types. |
| Static Anonymous Functions | Improves performance by preventing unnecessary variable captures. |
| Target-Typed Conditional Expressions | Simplifies expressions and improves code conciseness. |
| Covariant Return Types | Ensures safer and more flexible method overrides with specific return types. |
| Extension GetEnumerator Support | Enhances foreach loop support by allowing custom enumeration behavior. |
| Lambda Discard Parameters | Reduces warnings and clutter when parameters are unused in lambdas. |

### ****🧐 Comparison with Previous Approaches****

|  |  |  |
| --- | --- | --- |
| **Feature** | **Previous Approach** | **C# 9.0 Enhancement** |
| Target-Typed New Expressions | Explicitly defining the type when creating new objects. | Inferred type, reducing redundancy in object creation. |
| Static Anonymous Functions | Anonymous functions captured outer variables by default. | Static lambdas prevent capturing of outer variables, improving performance. |
| Target-Typed Conditional Expressions | Explicit type definitions in conditional expressions. | Type inferred in conditional expressions, simplifying the code. |
| Covariant Return Types | Return types had to exactly match the base method's return type. | More specific return types are allowed in overridden methods. |
| Extension GetEnumerator Support | Custom iteration required manually implementing GetEnumerator in the collection. | Extension methods can now define custom GetEnumerator, simplifying iteration logic. |
| Lambda Discard Parameters | Unused parameters in lambdas would generate warnings. | Discard unused parameters to avoid warnings and improve code clarity. |

### ****⚠️ Potential Pitfalls****

* **Target-Typed New Expressions**:  
  Ensure that the inferred type is always clear and understandable to avoid confusion in complex codebases.
* **Static Anonymous Functions**:  
  Static lambdas should be used carefully as they do not capture outer variables, which may sometimes be required for certain functionality.
* **Target-Typed Conditional Expressions**:  
  While target-typing simplifies the code, be cautious when working with complex types, as inference might not always behave as expected.
* **Covariant Return Types**:  
  Covariant return types can increase flexibility but should be used with care, ensuring the override does not break the intended contract of the base class method.
* **Extension GetEnumerator Support**:  
  Extending GetEnumerator may lead to potential issues in iterating over collections if not implemented correctly, leading to performance issues or runtime errors.
* **Lambda Discard Parameters**:  
  While discard parameters improve readability, overuse of this feature can lead to the loss of important parameters that might be useful in the future.