

C# 11

C# 11 – New features

- Raw string literals
- List Pattern
- Slice Pattern
- File Local Types / File-Spaced Types
- Required Members
- Auto Default Structs
- Ref Fields

Harsha
Web University

Raw String Literals

C# 11

Raw String Literals

Raw string literals are a new way to represent strings in C# that span multiple lines and preserve special characters (like quotes, backslashes, and newlines) without the need for explicitly disabling the escape sequences.

Syntax:

- Enclosed in at least three double-quote characters ("""").
- Can span multiple lines.
- "\" can be included without needing to escape the string without explicitly disabling the escape sequences.

Example:

```
string sql = """  
SELECT *  
FROM Users  
WHERE Age > 30  
""";
```

Harsha
Web University

Purpose of Raw String Literals

Simplify the representation of strings containing special characters or spanning multiple lines.

Improve the readability of code by removing the need for backslash escape sequences.

Connections between routing keys (events or actions) and message queues within a specific exchange.

Make it easier to work with strings containing paths, JSON, XML, regular expressions, etc.



Use Cases of Raw String Literals

Multi-line SQL queries.

JSON or XML literals.

Regular expressions with special characters.

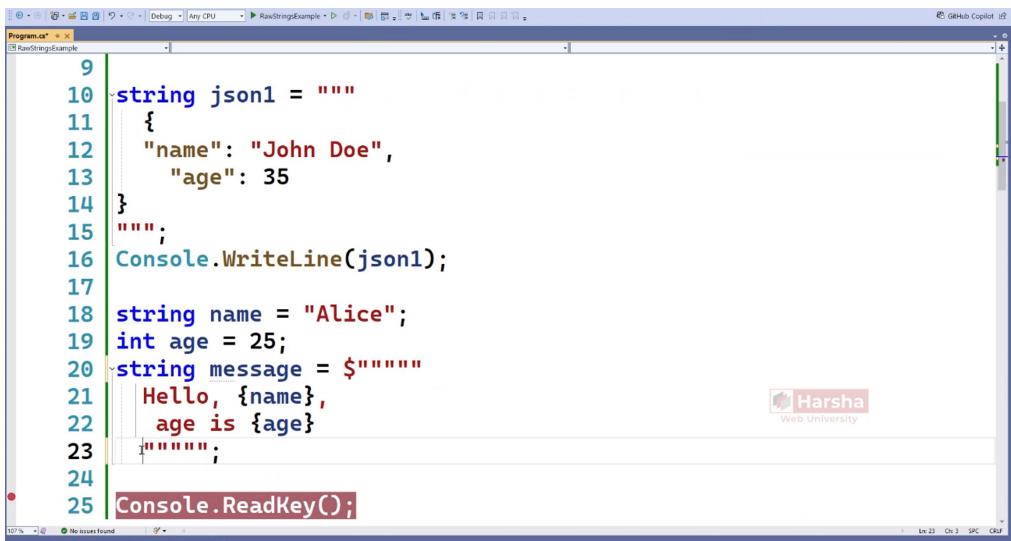
File paths with backslashes.



Additional Notes of Raw String Literals

Leading and trailing whitespace on each line is preserved.





A screenshot of the Visual Studio IDE showing a C# file named Program.cs. The code demonstrates raw string literals and string interpolation. The code is as follows:

```
9  string json1 = """;
10 {
11     "name": "John Doe",
12     "age": 35
13 }
14 """
15 ;
16 Console.WriteLine(json1);
17
18 string name = "Alice";
19 int age = 25;
20 string message = $"""
21     Hello, {name},
22     age is {age}
23 """
24
25 Console.ReadKey();
```



C# 11

Harsha

Additional Notes of Raw String Literals

Leading and trailing whitespace on each line is preserved.

You can combine raw string literals with string interpolation (\$).

Can significantly improve code quality and maintainability in certain scenarios.

The delimiters (""""") can have any number of double quotes.

The opening and closing delimiters must have the same number of double quotes.

In case of multi-line strings, both the opening delimiter (""""") and the closing delimiter (""""") need to be on their own lines.

Any whitespace to the left of the closing quotes should be present in all lines and will be removed from all lines of the multi-line raw string literal.



Odemy

List Pattern

List Pattern

List patterns extend pattern matching to match sequences of elements in a list or an array.

Syntax:

- Enclosed in square brackets [].
- Elements separated by commas.

Example:

```
{ ... } is [1, 2, 3]
```



Purpose of List Pattern

Simplify pattern matching on lists and arrays.

Express complex conditions concisely.

Extract elements while matching.



Use Cases of List Pattern

Validating input lists or arrays.

Simplifying conditional logic on lists and arrays.



Additional Notes of List Pattern

Works with lists (`List<T>`) and arrays (`T[]`).

Nested list patterns allow for matching complex structures.

Can contain other patterns such as type patterns, property patterns, relational patterns etc.



Slice Pattern

Slice Pattern

The slice pattern (..) in C# 11 is used within list patterns to match zero or more consecutive elements in a list or array.

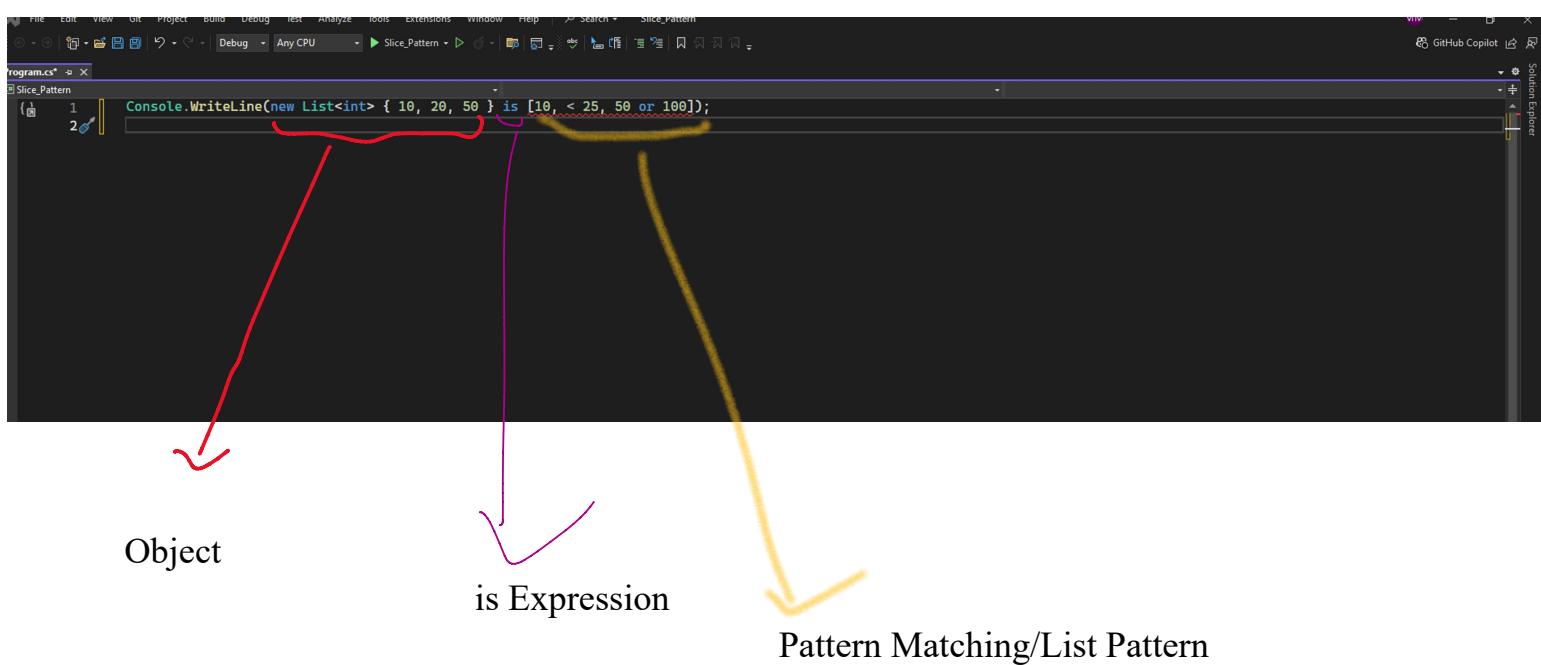
Syntax:

- Represented by two dots (..).
 - Can appear either at the beginning, middle, or end of a list pattern:
- [.., pattern]** (zero or more elements followed by a specific pattern)
 - [pattern1, .., pattern2]** (elements matching pattern1, then zero or more, then elements matching pattern2)
 - [pattern, ..]** (elements matching pattern followed by zero or more)

{ ... } is [..., 2, 3]

{ ... } is [1, ..., 3]

{ ... } is [1, 2, ...]



Additional Notes of Slice Pattern

Exclusive to list patterns.

Can be used only at beginning, middle or ending or list pattern.

Can be used only once within a list pattern.

Can be used in combination with other patterns.



Var Pattern

Var Pattern

The 'var' pattern within a list pattern allows you to capture and assign elements of a list or array to new variables during pattern matching.

Syntax:

- `var` variableName
- Can be used in list pattern:

1. `[var variable, pattern1, pattern2]`
(Variable captures element at index 0)
2. `[pattern1, var variable, pattern2]`
(Variable captures element at index 1)
3. `[pattern1, pattern2, var variable]`
(Variable captures element at index ^1)

`{ ... } is [var x, 1, 2]`

`{ ... } is [1, var x, 2]`

`{ ... } is [1, 2, var x]`



File Local Types

Until C 11 you have only three access modifiers for the types right for example you have types like classes interfaces structs enumerations or delegate types and you can use only three access modifiers for them internal public and private you cannot use other access modifiers on types isn't it but now with C 11 you can use additional access modifier on the types that is file access modifier so what happens if you give file access modifier for a particular type such as class or struct that particular type is now accessible within the same source code file only but not in other source code files for example here we have created class one within the file one with file access modifier it is not internal or public but is file so this particular class one is now accessible within the same file only but not in other files so if you try to access this class one from other files you will get compile time error and in the same way class two but you can access that type inside the same file.

C# 11 | Harsha

File Local Types

File local types are types (classes, structs, interfaces, etc.) are declared with "file" access modifier, whose visibility is restricted to the single source file where they are declared.

File1.cs:

```
file class Class1
{}
```

File2.cs:

```
file class Class2
{}
```

C# 11 | Harsha

File Local Types

File local types are types (classes, structs, interfaces, etc.) are declared with "file" access modifier, whose visibility is restricted to the single source file where they are declared.

File1.cs:

```
file class Class1
{}
```

File2.cs:

```
file class Class2
{}
```

//Class2 is not accessible

//Class1 is not accessible

Use Cases of File Local Types

Source Generators: Prevent generated code from polluting the global namespace.

Private Implementation Details: Encapsulate helper types within a file.



Additional Notes of File Local Types

File-local types cannot be used as base classes or implemented interfaces in other files.

They cannot be used as generic type arguments or public method return types in other files.

They cannot be marked with other access modifiers such as public or internal.



Required Members

Required Members

Required members are class or struct properties/fields that MUST be initialized either within the constructor or in the object initializer.

Class:

```
class ClassName
{
    public required string Name { get; set; }
}
```

Object:

```
ClassName c1 = new ClassName();
//Error: You must initialize the required property
```



udemy

While working with model classes or other classes or strings, you may have some fields or properties that should be assigned a value compulsorily. Otherwise, we may not be able to process calculations.

For example, consider a class called Circle. To calculate the area of a circle, the object must contain the value of radius. Without it, you cannot perform the calculation.

Traditionally, to handle such required members (fields or properties), you would have to manually write code for null checks. For example, you might check if radius is not null or zero before calculating the area of the circle. If the value is missing, you would manually throw exceptions like ArgumentNullException. You would then need to catch these exceptions somewhere in your code.

However, with C# 11, this is no longer necessary, thanks to the new **required members** feature.

In structs or classes, you can mark an important field or property with the required modifier. This ensures that the compiler makes sure the field or property is initialized to some value when creating the object.

The value of a required field can be initialized:

- Within the object initializer
- Within a constructor (parameterless or parameterized)

If the required field is not initialized while creating the object, it will throw a **compile-time error**.

Benefits of Required Members:

1. Prevents unexpected runtime errors by ensuring required values are initialized.
2. Reduces the need for manual null checks or validation code.

Purpose of Required Members

Guarantee that essential properties are not left uninitialized.

Eliminate the need for manual null checks in many cases.



Use Cases of Required Members

Domain Models: Ensure entities have mandatory attributes.

DTO Models: Clearly indicate required input parameters.



Additional Notes of Required Members

Required members must have a setter (init or set).

If the field / property is being initialized in the constructor, that constructor should be marked with [SetsRequiredMembers] attribute; otherwise you will get compile-time error.



Auto Default Structs

Until C 11, struct fields should be initialized explicitly within the constructor, but it's no longer a requirement from C 11 onwards.

But from C 11, all the struct fields are automatically initialized to their default values based on their types as soon as you invoke the default constructor of the struct. Even if you have a user-defined struct constructor and you don't initialize all the fields, I mean you just initialize some of the fields, all the uninitialized fields get automatically initialized.

So this is called auto-default structs. This update is meant for simplifying the code for initializing the struct fields explicitly.

Auto Default Structs

In C# 11, struct fields are automatically initialized to their default values (based on their types), if not explicitly assigned in a constructor.

Struct:

```
struct StructName  
{  
    public type field;  
}
```

Struct Instance:

```
StructName variableName = new StructName();  
//Struct fields are automatically initialized
```



Ref Fields

Ref Fields in Ref Structs

Ref structs are value types that cannot be boxed and must be allocated on the stack. They can contain ref fields, which hold references to other variables.

Struct:

```
ref struct StructName  
{  
    public ref type fieldName;  
    //Ref field can store reference of another variable  
}
```

Harsha
Web University

When working with complex and large data structures in business applications, you may want to copy the data from one variable to another variable, but it might be an expensive operation. To overcome this problem, reference fields are introduced, which allow you to create references to your variable in your field of a struct.

That field is called a reference field, and its containing struct is called a reference struct. Since the struct field and the original variable are connected via a reference, manual copying of the data is not required, thereby improving performance.

Purpose of Ref Fields and Ref Structs

Enable the creation of custom types that directly reference other values or objects without incurring the overhead of copying or boxing.

Allow for efficient manipulation of large data structures or unmanaged memory.



Additional Notes of Ref Fields

Ref Fields

- Hold references, not values
- Assignment creates references, not copies
- Scope limited to ref struct lifetime



Additional Notes of Ref Structs

A ref struct can't be the element type of an array.

A ref struct can't be a type of a field of a class or a non-ref struct.

A ref struct can't implement interfaces.

A ref struct can't be boxed to System.ValueType or System.Object.

A ref struct can't be a type argument for generic types or generic methods.

A ref struct variable can't be captured in a lambda expression or a local function.

Before C# 13, ref struct variables can't be used in an async method.

Beginning with C# 13, a ref struct variable can be used until "await" keyword appears.

