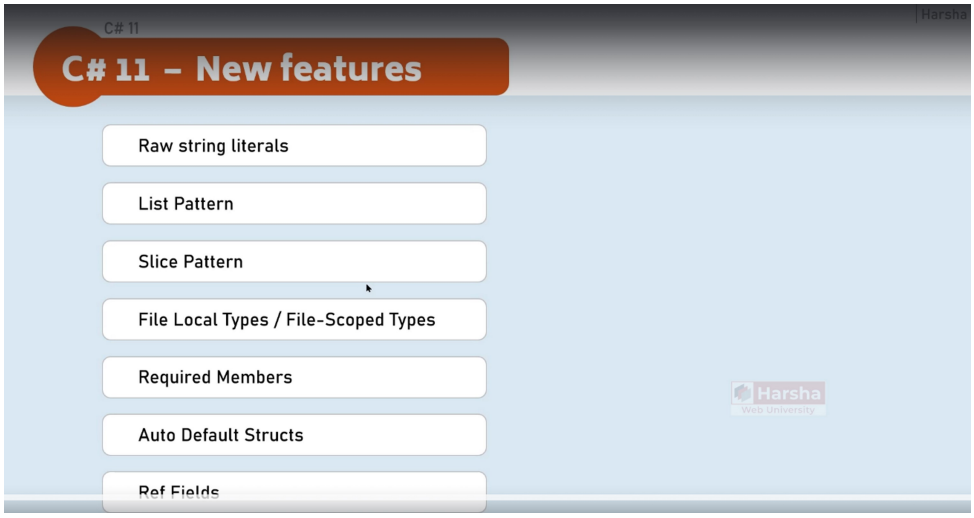
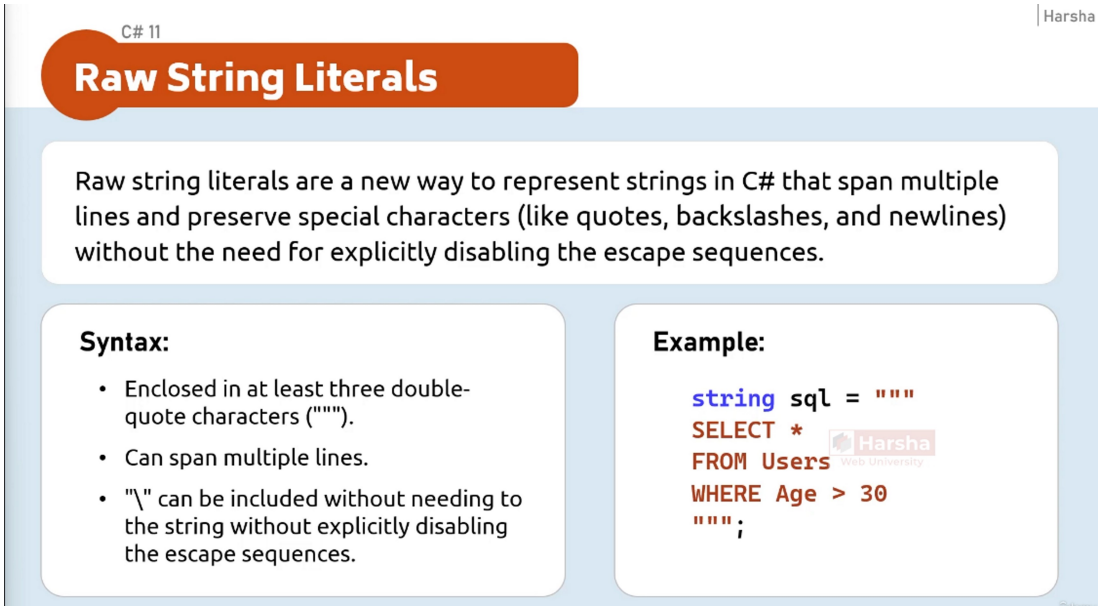
Section 34: C# 11 (.NET7) - New Features

25 December 2024 11:20



Raw String Literals



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.

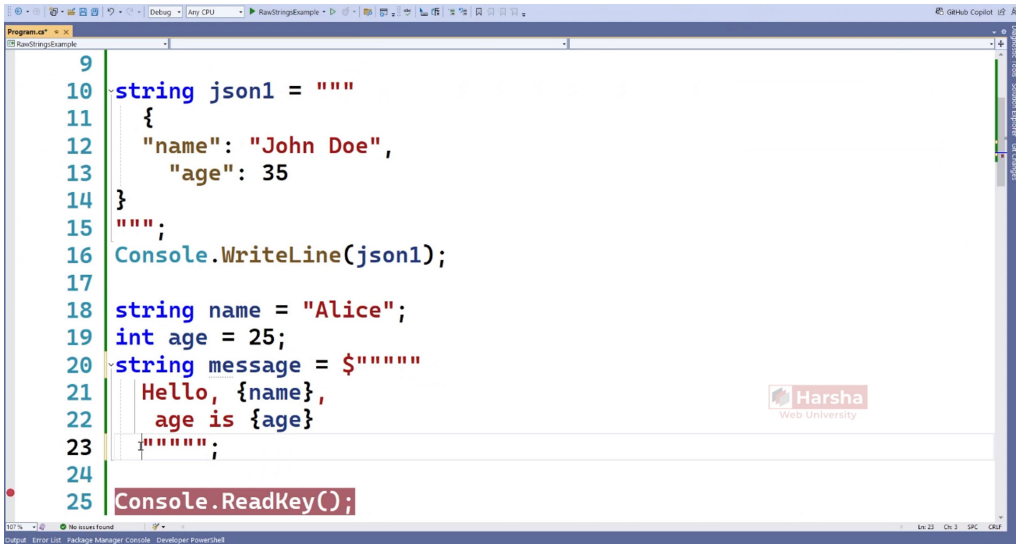
JSON or XML literals.

Regular expressions with special characters.

File paths with backslashes.

Additional Notes of Raw String Literals

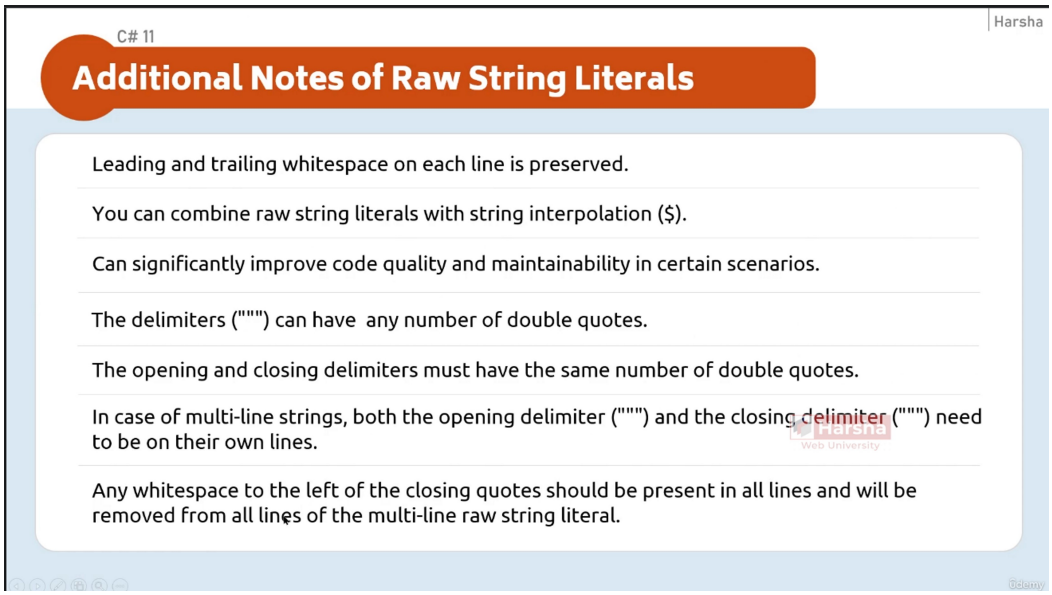
Leading and trailing whitespace on each line is preserved.



```
string json1 = """
  {
  "name": "John Doe",
    "age": 35
}
""";
Console.WriteLine(json1);

string name = "Alice";
int age = 25;
string message = $"""""
  Hello, {name},
   age is {age}
  """""";

Console.ReadKey();
```



List Pattern

C# 11

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]
```

C# 11

Purpose of List Pattern

Simplify pattern matching on lists and arrays.

Express complex conditions concisely.

Extract elements while matching.

C# 11

Use Cases of List Pattern

Validating input lists or arrays.

Simplifying conditional logic on lists and arrays.

C# 11

Harsha

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

C# 11

Harsha

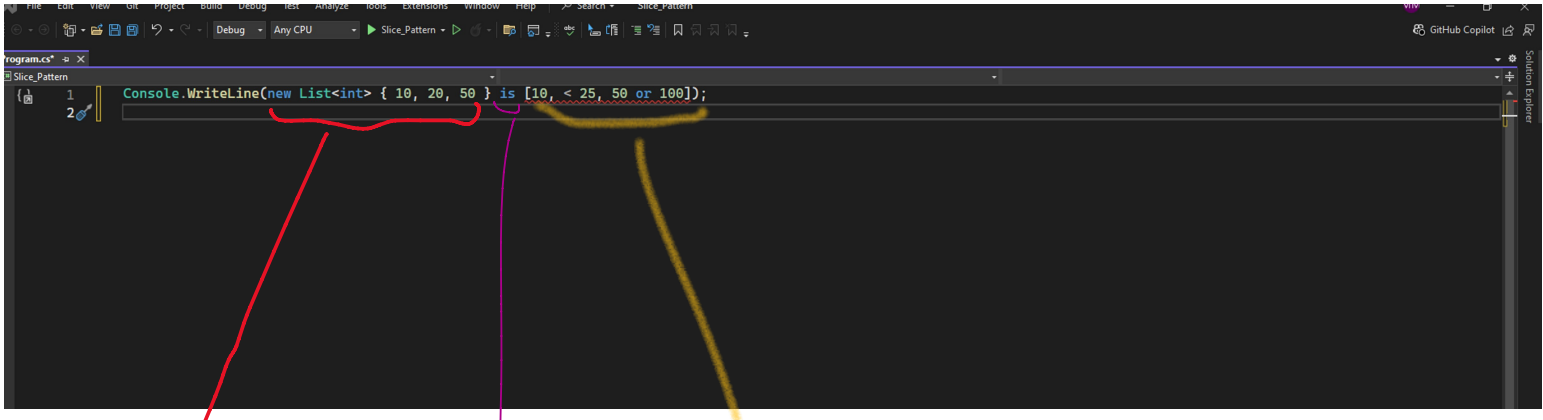
Slice Pattern

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:

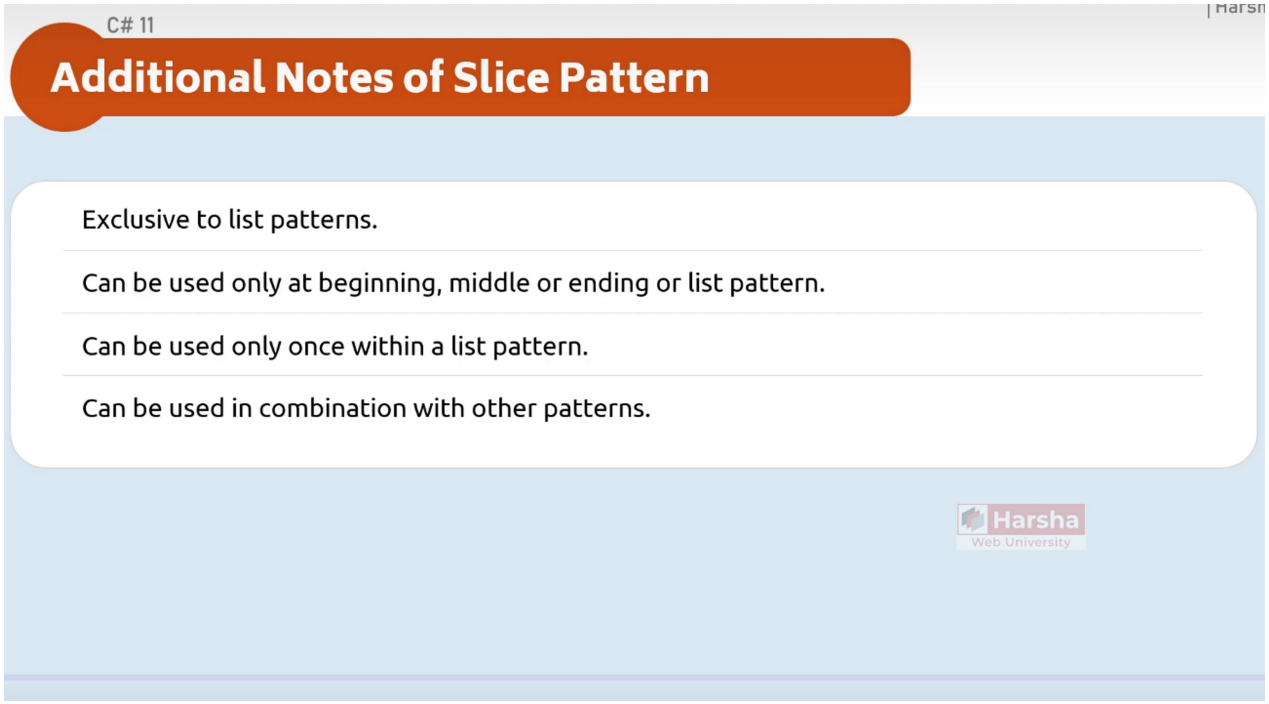
1. [.., pattern] (zero or more elements followed by a specific pattern)	`{ ... } is [.., 2, 3]`
2. [pattern1, .., pattern2] (elements matching pattern1, then zero or more, then elements matching pattern2)	`{ ... } is [1, .., 3]`
3. [pattern, ..] (elements matching pattern followed by zero or more)	`{ ... } is [1, 2, ..]`



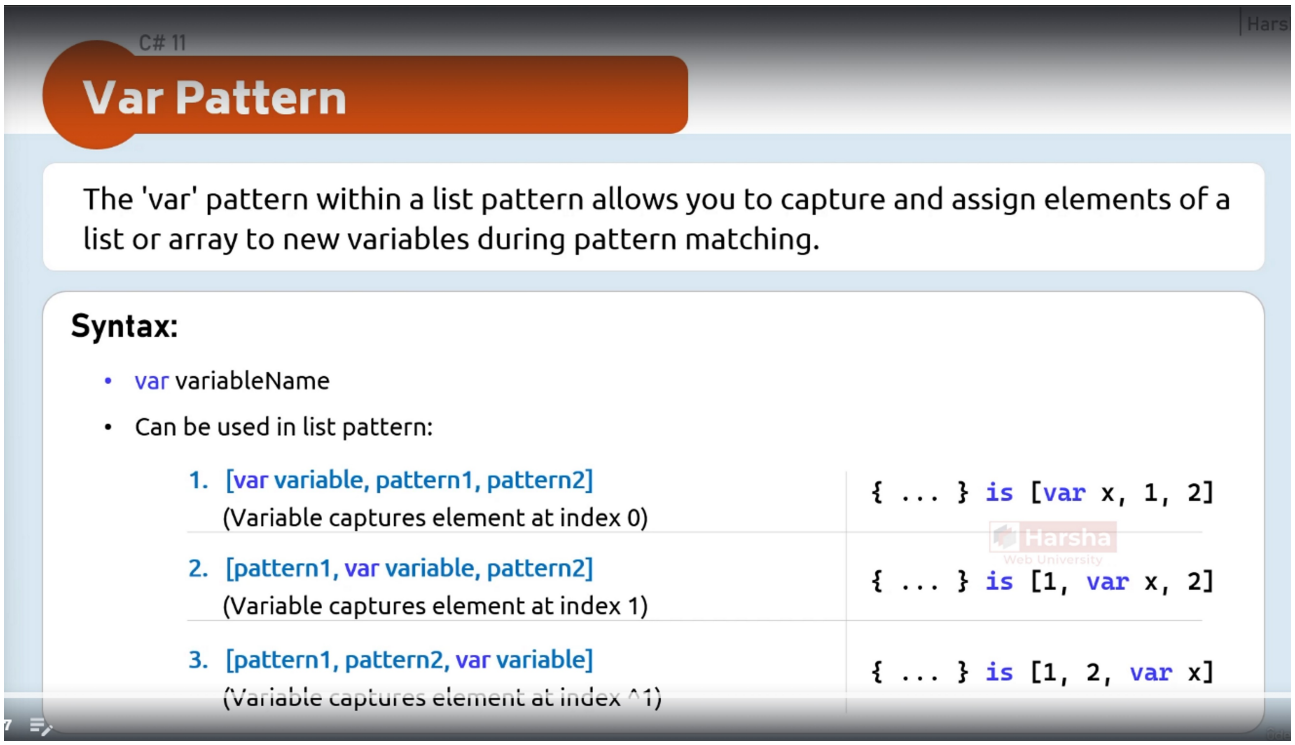
Object

is Expression

Pattern Matching/List Pattern

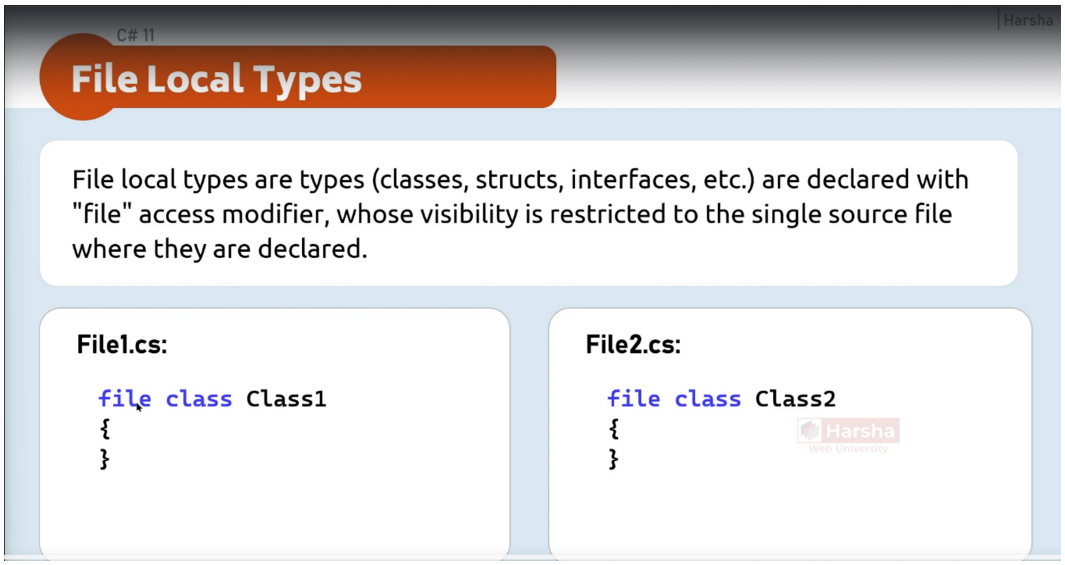

C# 11
Additional Notes of Slice Pattern
Exclusive to list patterns.
Can be used only at beginning, middle or ending or list pattern.
Can be used only once within a list pattern.
Can be used in combination with other patterns.
Harsha
Web University

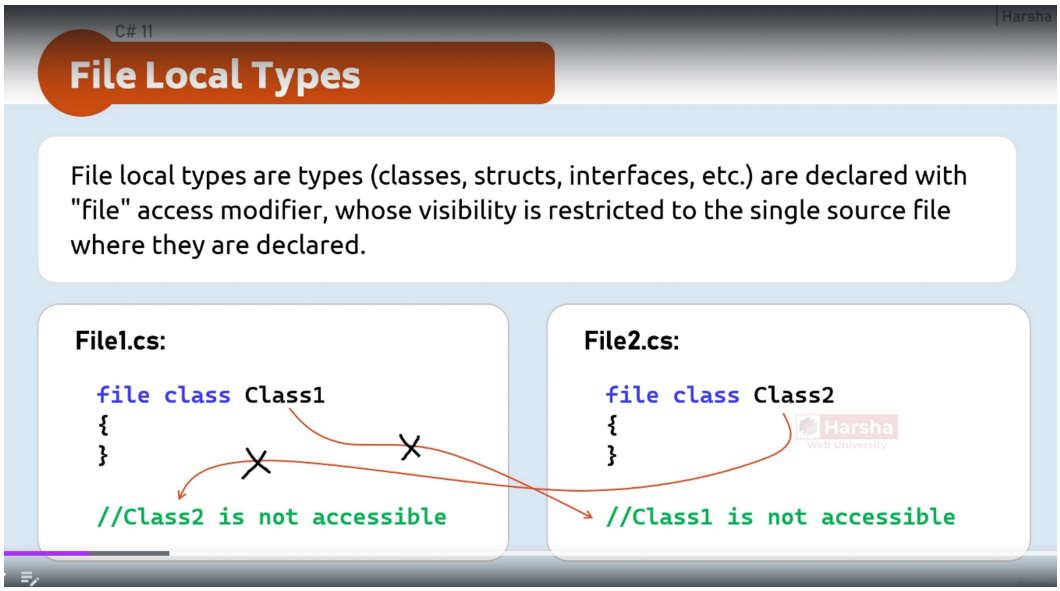
Var Pattern

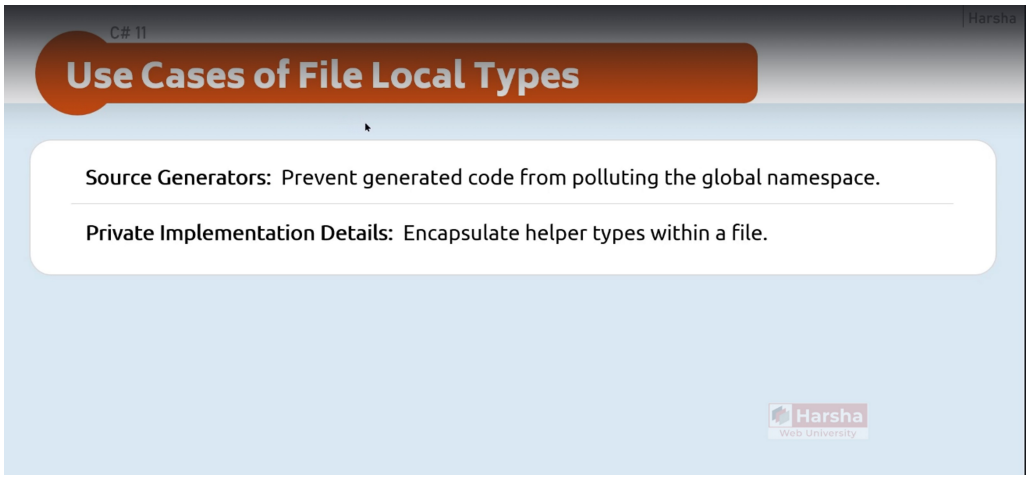

C# 11
Var Pattern
The 'var' pattern within a list pattern allows you to capture and assign elements of a list or array to new variables during pattern matching.
Syntax:
var variableName
Can be used in list pattern:
1. [var variable, pattern1, pattern2]
(Variable captures element at index 0)
{ ... } is [var x, 1, 2]
2. [pattern1, var variable, pattern2]
(Variable captures element at index 1)
{ ... } is [1, var x, 2]
3. [pattern1, pattern2, var variable]
(Variable captures element at index ^1)
{ ... } is [1, 2, var x]
Harsha
Web University

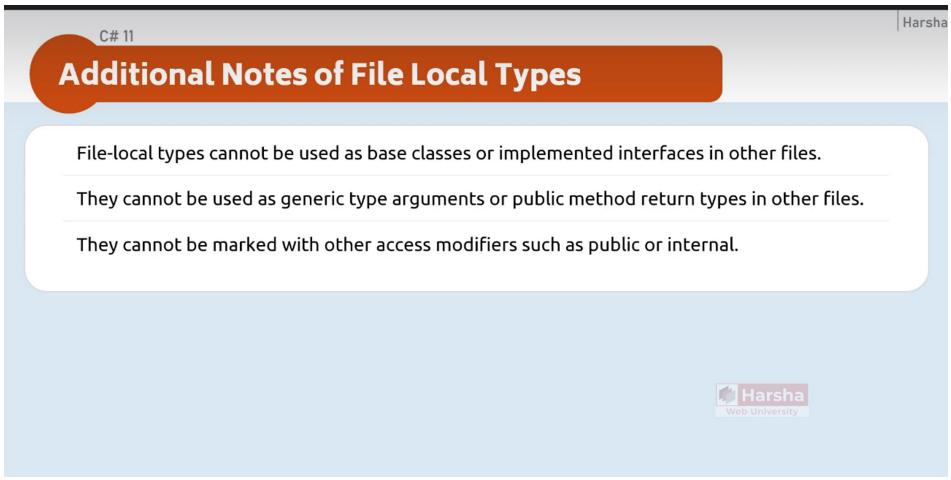
File Local Types

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 modific 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.

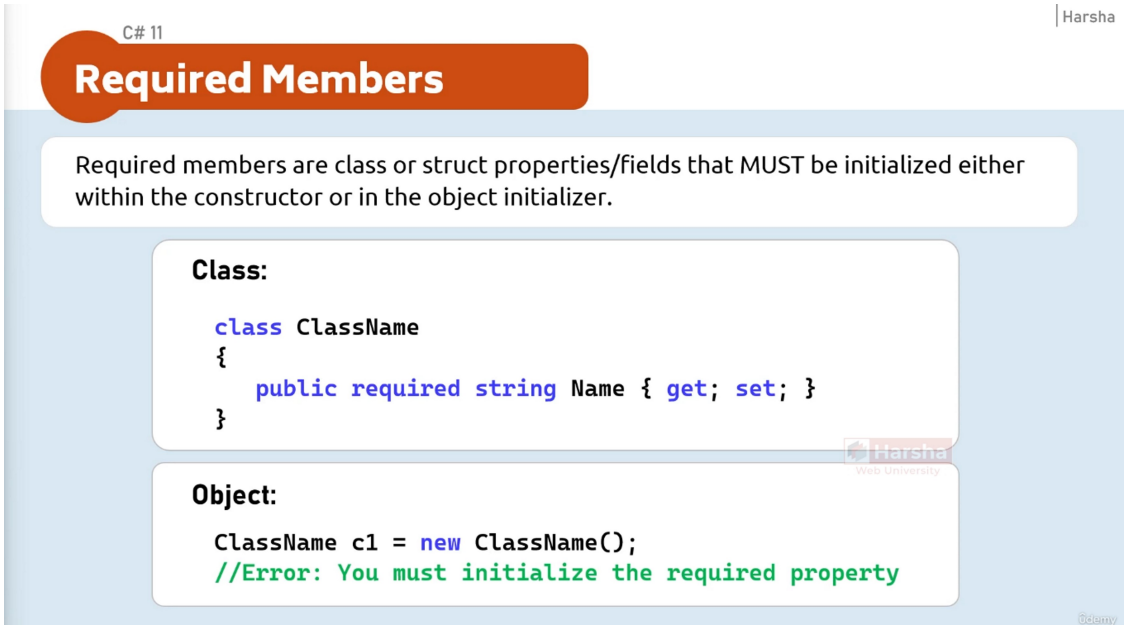








Required Members

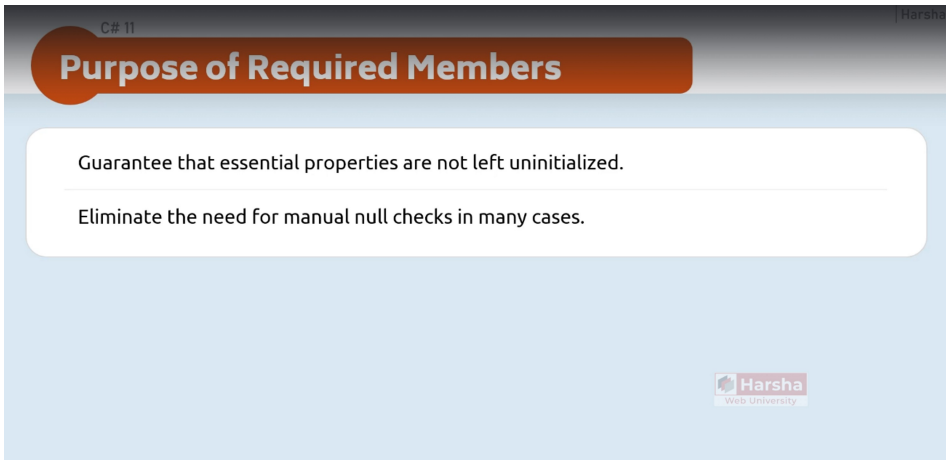


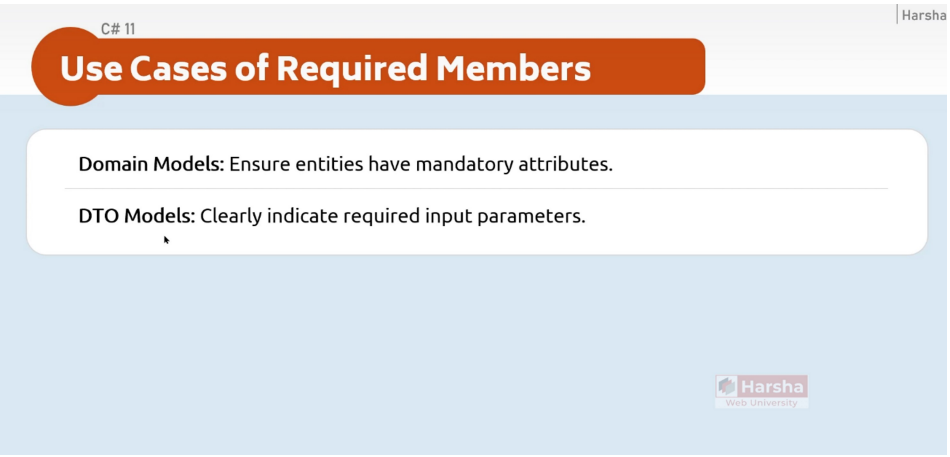
While working with model classes or other classes or strings, you may have some fields or properties that should be assigned a value compulsorily. Otherwise, we may not be able to process calculations.
For example, consider a class called Circle. To calculate the area of a circle, the object must contain 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:

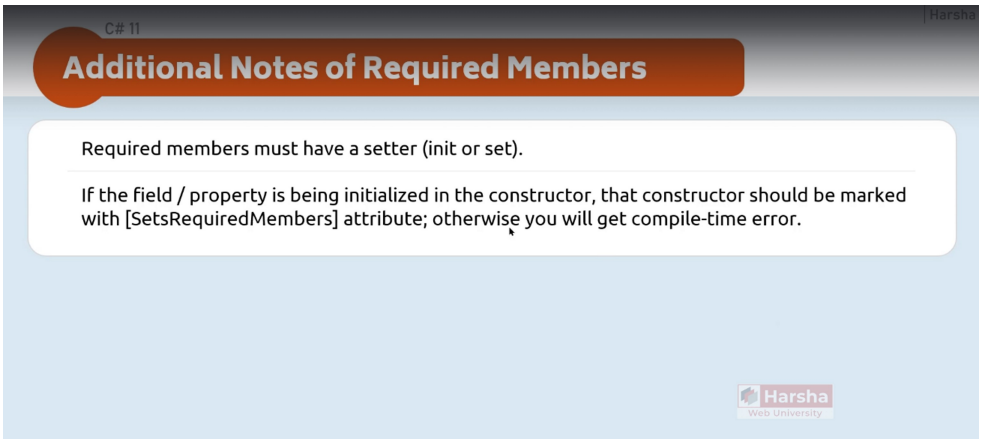
- Within the object initializer
- Within a constructor (parameterless or parameterized)

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.



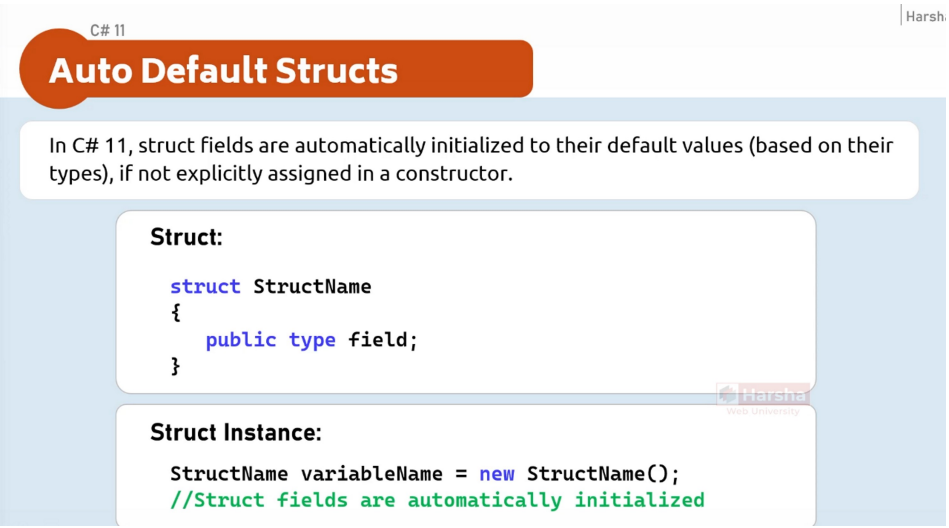




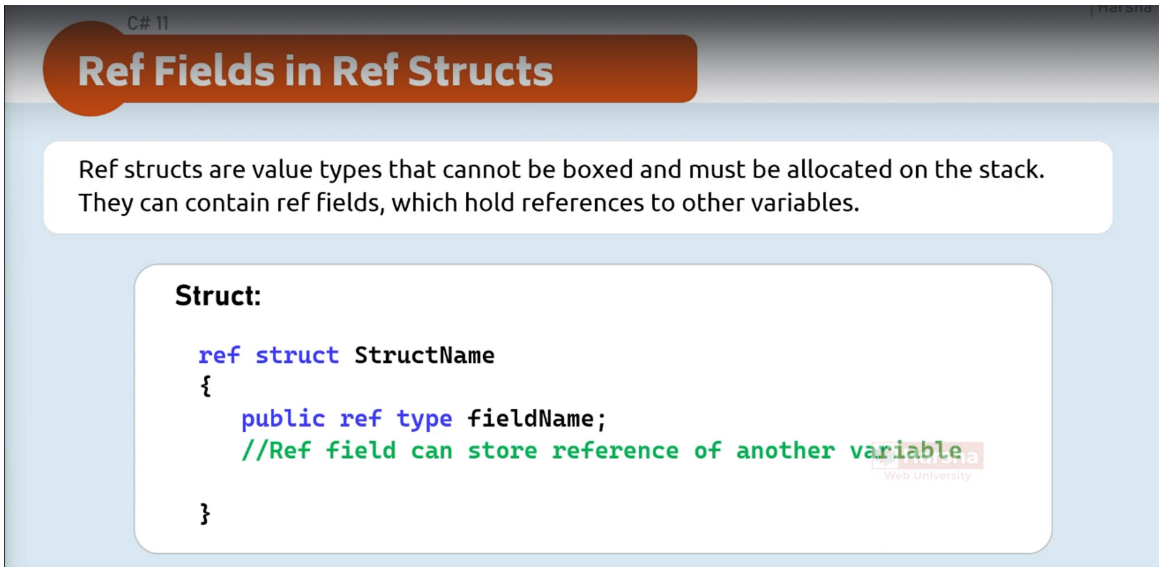
Auto Default Structs

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.



Ref Fields



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.

C# 11

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.

C# 11

Additional Notes of Ref Fields

Ref Fields

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

C# 11

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.