Section 30: C# 9 and 10 - New Features

10 November 2024 11:03

Top-level Statements

Top Level Statements

```
statement1;
statement2;
```

Top level statements

```
namespace namespace1
{
  //types
}
```

Harsha Web University

```
using System;
using System.IO;

Console.WriteLine("Hello, World!");
string name = "Harsha";
Console.WriteLine($"Hello to {name}");
Console.ReadKey();

namespace namespace1
{
  0 references
  class Sample
  {
    0 references
    public void Method1()
    {
      string str = name;
    }
  }
```

```
class Program
{
  static void Main()
  {
  }
}
```

Harsha Web University

```
using System;
using System.IO;

Console.WriteLine("Hello, World!");
string name = "Harsha";
Console.WriteLine($"Hello to {name}");
Console.ReadKey();

namespace namespace1
{
  0 references
  class Sample
  {
    0 references
    public void Method1()
    {
      string str = name;
    }
  }
```

```
class Program
{
  static void Main()
  {
    name = "Harsha";
  }
}
```

Harsha Web University

C# 9

Top Level Statements

File1.cs

statements...

namespaces / types...

Allows a sequence of statements to occur right before the namespaces / type definitions in a single file in the C# project.

Would compile as

```
static class Program
{
    static async Task Main(string[] args)
    {
        //statements...
    }
}
```


C# 9

Top Level Statements

Advantage: Make C# learning curve easy for C# learners (newbies).

Only one compilation unit (C# file) can have top level statements in a C# project.

The compiler-generated class and Main method are NOT accessible through code of any other areas of the project.

The local variables / local functions declared in the top-level statements are NOT accessible elsewhere (in other types / files).

C# 9

Top Level Statements

Advantage: Make C# learning curve easy for C# learners (newbies).

Only one compilation unit (C# file) can have top level statements in a C# project.

The compiler-generated class and Main method are NOT accessible through code of any other areas of the project.

The local variables / local functions declared in the top-level statements are NOT accessible elsewhere (in other types / files).

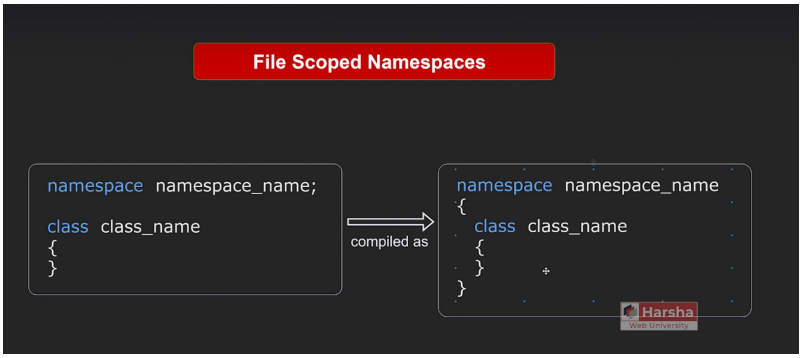
The compiled Main method would be 'async', by default.

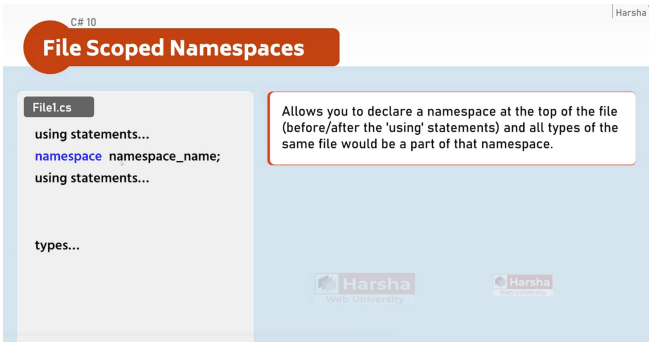
So it allows 'await' statements in top-level statements.

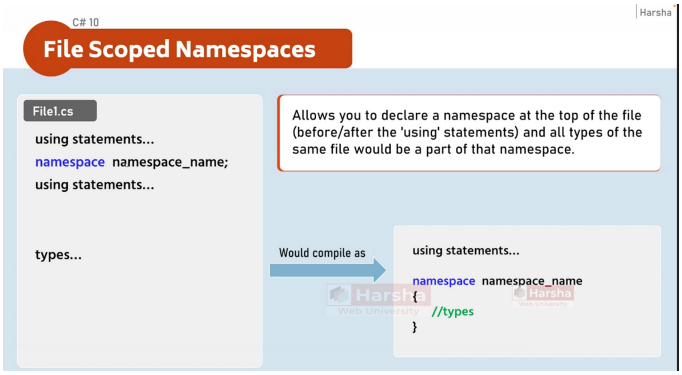
Top level statements can access command-line arguments using 'args'.

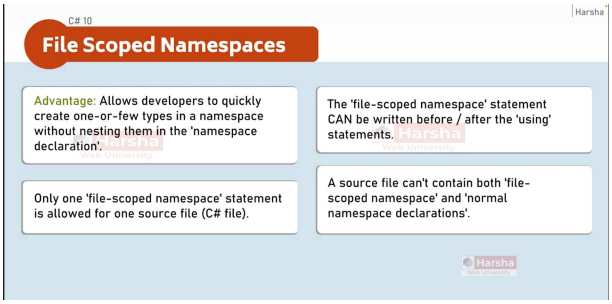
A string[] args parameter would be generated by the compiler automatically.

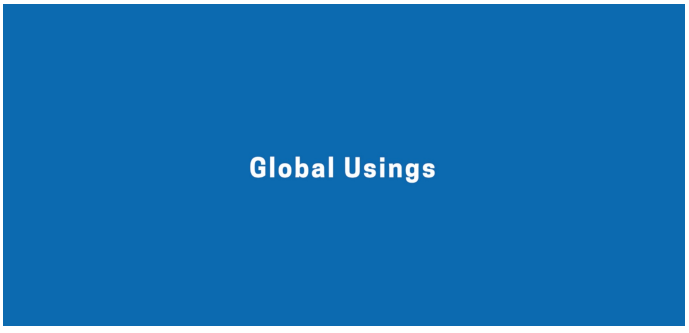
File-Scoped Namespaces


File Scoped Namespaces
namespace namespace_name;
class class_name
{
}
compiled as
namespace namespace_name
{
class class_name
{
}
}
Harsha
Web University

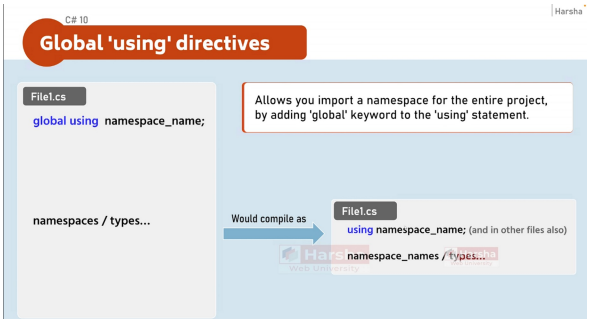

C# 10
File Scoped Namespaces
Harsha
File1.cs
using statements...
namespace namespace_name;
using statements...
types...
Allows you to declare a namespace at the top of the file (before/after the 'using' statements) and all types of the same file would be a part of that namespace.

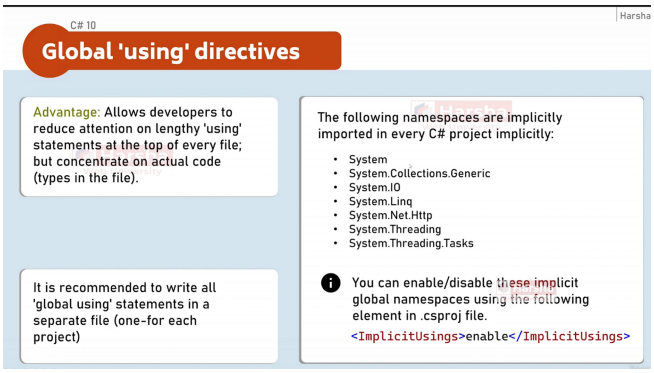

C# 10
File Scoped Namespaces
Harsha
File1.cs
using statements...
namespace namespace_name;
using statements...
types...
Allows you to declare a namespace at the top of the file (before/after the 'using' statements) and all types of the same file would be a part of that namespace.
Would compile as
using statements...
namespace namespace_name
{
//types
}

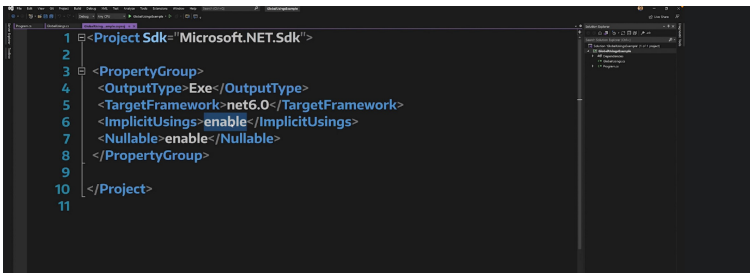

C# 10
File Scoped Namespaces
Harsha
Advantage: Allows developers to quickly create one-or-few types in a namespace without nesting them in the 'namespace declaration'.
The 'file-scoped namespace' statement CAN be written before / after the 'using' statements.
Only one 'file-scoped namespace' statement is allowed for one source file (C# file).
A source file can't contain both 'file-scoped namespace' and 'normal namespace declarations'.


Global Usings

Instead of importing common namespace at all the file, we can define at a single place.

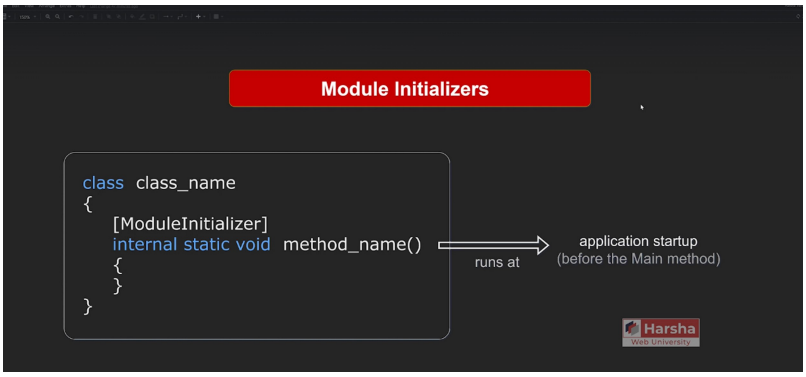


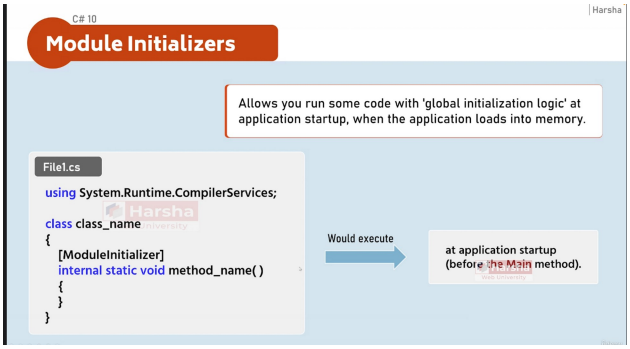


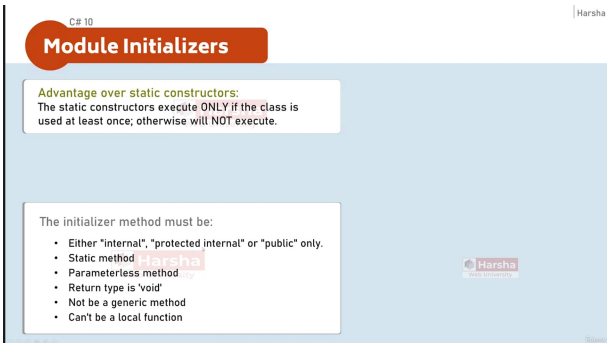


Module Initializers means, application startup logic that should be executed when the applications starts. Means when the application loads into memory.

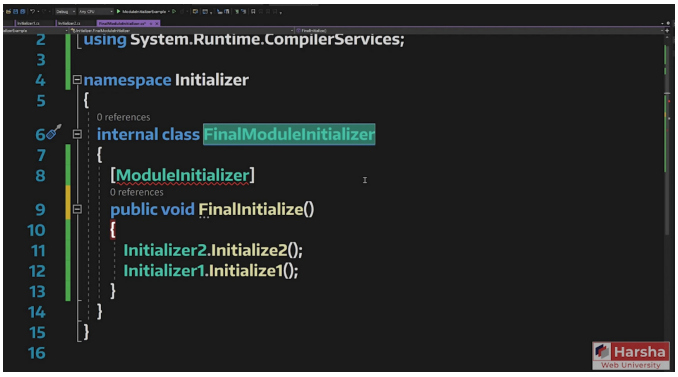
For example, the code for initializing the database connection strings or the code for loading the environment variables etc. Earlier we solved this problem using static constructor.

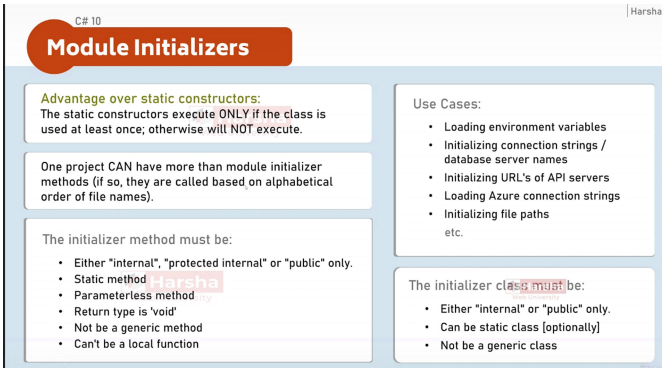


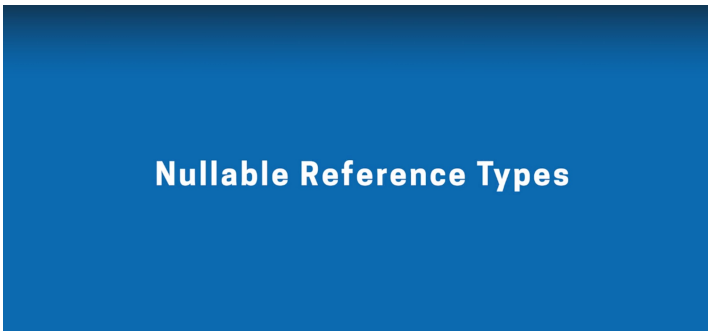


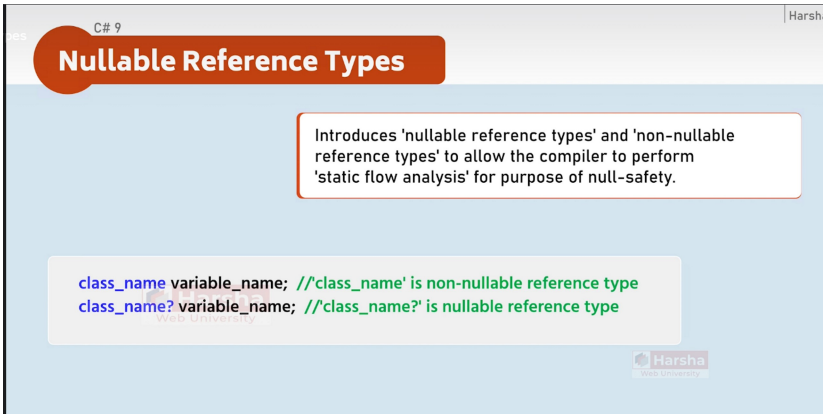


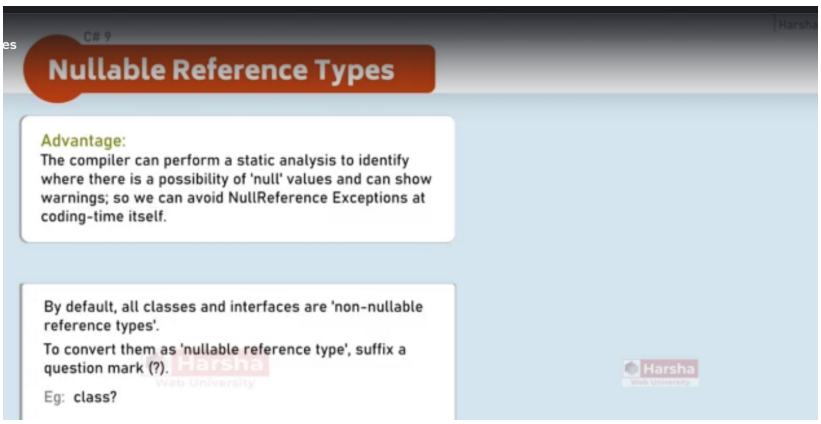
It should be static method so that CLR does not have to create object of the class.



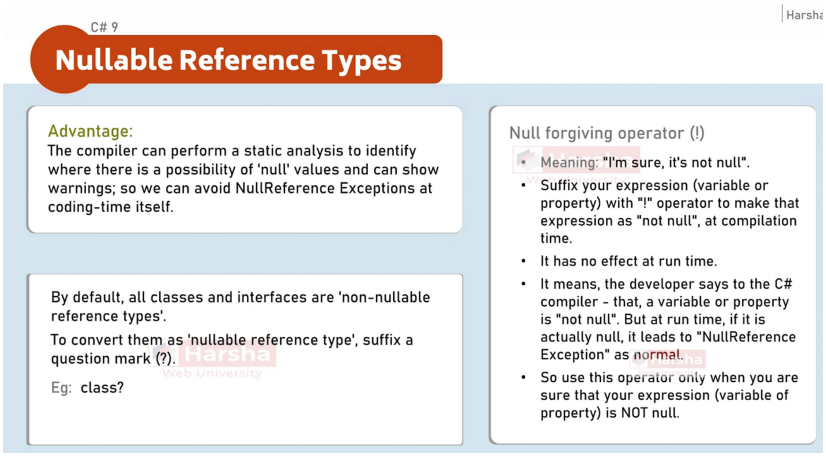

C# 10
Harsha
Module Initializers
Advantage over static constructors:
The static constructors execute ONLY if the class is used at least once; otherwise will NOT execute.
One project CAN have more than module initializer methods (if so, they are called based on alphabetical order of file names).
The initializer method must be:
Either "internal", "protected internal" or "public" only.
Static method
Parameterless method
Return type is 'void'
Not be a generic method
Can't be a local function
Use Cases:
Loading environment variables
Initializing connection strings / database server names
Initializing URL's of API servers
Loading Azure connection strings
Initializing file paths
etc.
The initializer class must be:
Either "internal" or "public" only.
Can be static class [optionally]
Not be a generic class


Nullable Reference Types

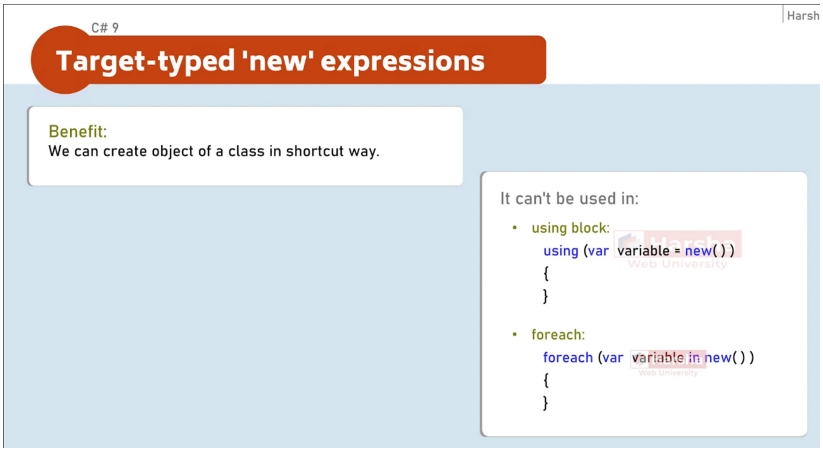

C# 9
Harsh
Nullable Reference Types
Introduces 'nullable reference types' and 'non-nullable reference types' to allow the compiler to perform 'static flow analysis' for purpose of null-safety.
class_name variable_name; //'class_name' is non-nullable reference type
class_name? variable_name; //'class_name?' is nullable reference type
Harsha


C# 9
Nullable Reference Types
Advantage:
The compiler can perform a static analysis to identify where there is a possibility of 'null' values and can show warnings; so we can avoid NullReference Exceptions at coding-time itself.
By default, all classes and interfaces are 'non-nullable reference types'.
To convert them as 'nullable reference type', suffix a question mark (?).
Eg: class?
Harsha

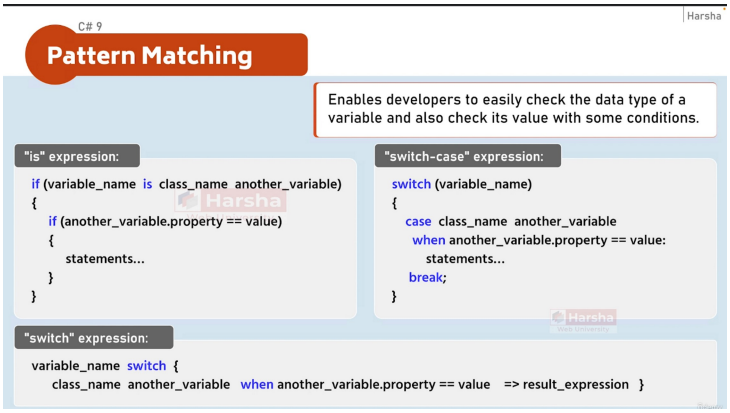
Null Forgiving (!) Operator



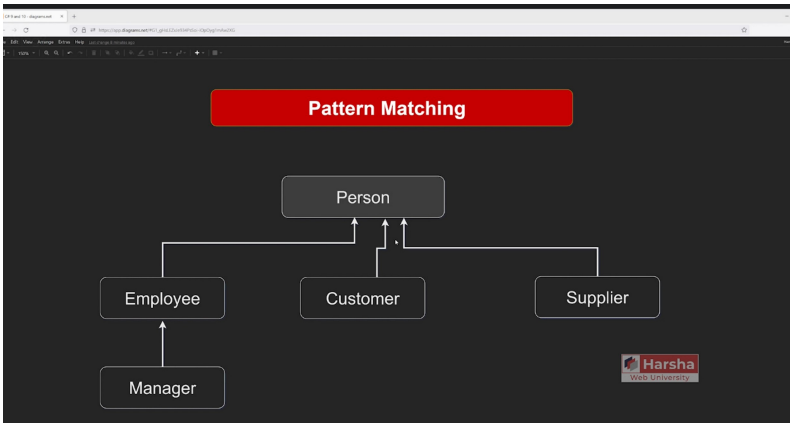
C# 9
Target-typed 'new' expressions



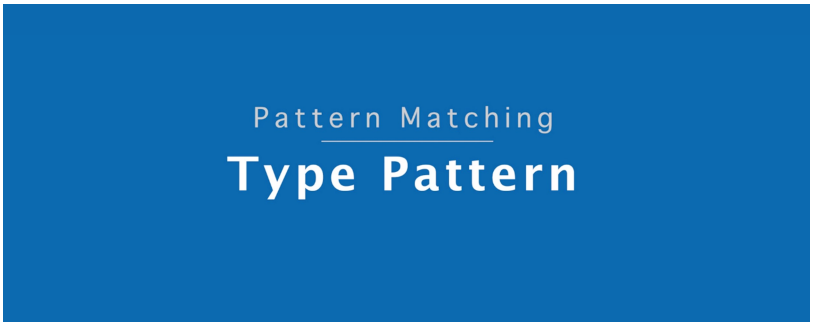
Why to use Pattern Matching?

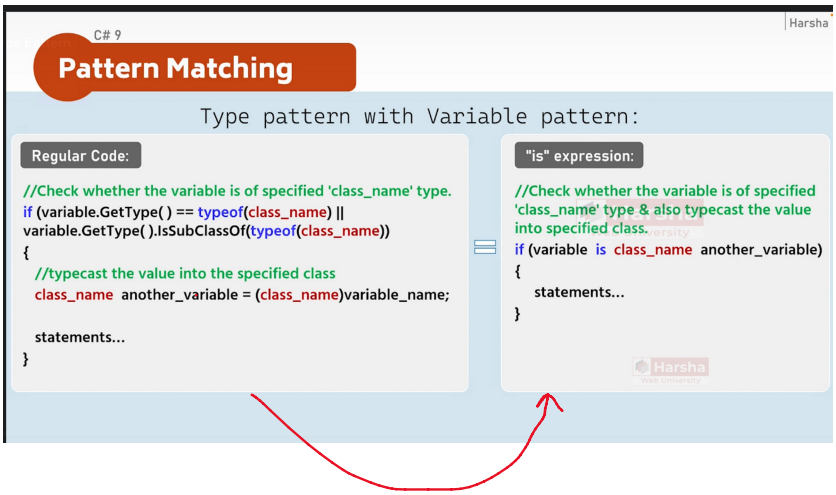


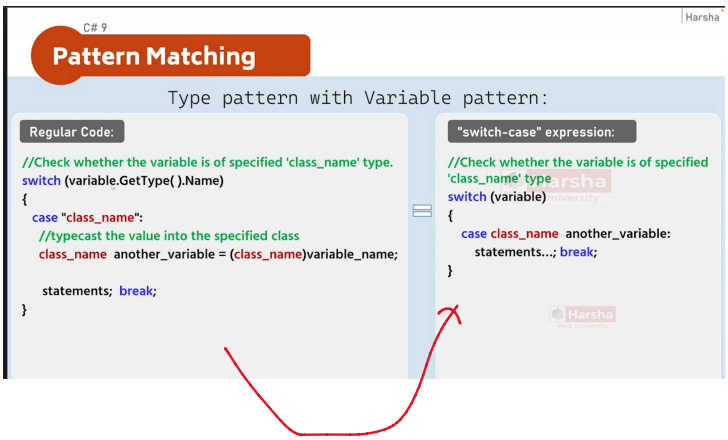
In this lecture, I will show how to write code without pattern matching concept?



Pattern Matching - Type Pattern







Pattern Matching

When Pattern

C# 9

Pattern Matching

Type pattern with When pattern:

Regular Code:

```
//Check whether the variable is of specified 'class_name' type.
switch (variable.GetType( ).Name)
{
  case "class_name":
    //typecast the value into the specified class
    class_name  another_variable = (class_name)variable_name;

    if (another_variable.property == value)
    {
       statements;
    }
    break;
}
```

"switch-case" expression:

```
//Check whether the variable is of specified
'class_name' type
switch (variable)
{
   case class_name  another_variable
      when  another_variable.property == value:
      statements...; break;
}
```

Pattern Matching

Switch-Expression Pattern

C# 9

Pattern Matching

Type pattern with When pattern:

Regular Code:

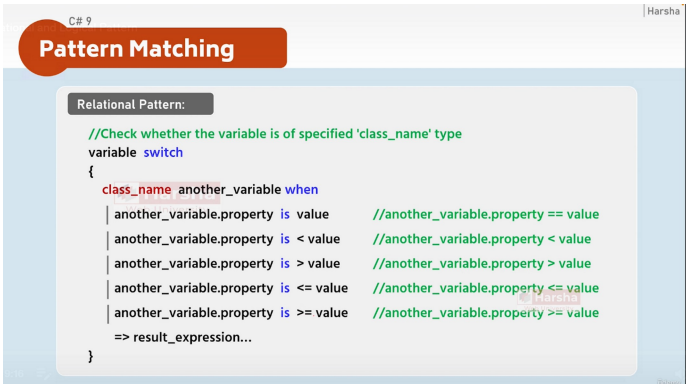
```
//Check whether the variable is of specified 'class_name' type.
switch (variable.GetType( ).Name)
{
  case "class_name":
    //typecast the value into the specified class
    class_name  another_variable = (class_name)variable_name;

    if (another_variable.property == value)
    {
       statements;
    }
    break;
}
```

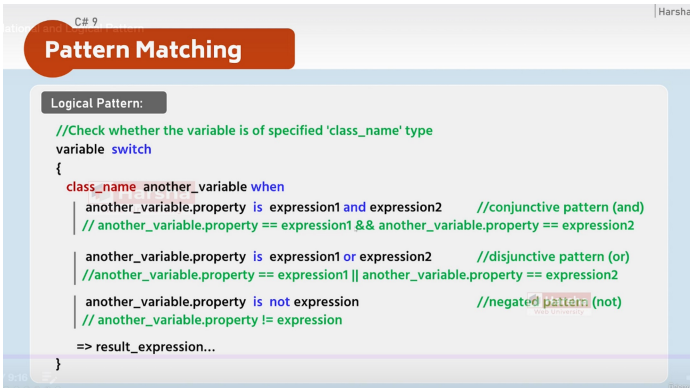
"switch" expression:

```
//Check whether the variable is of specified
'class_name' type
variable  switch
{
   class_name  another_variable
      when  another_variable.property == value
      => statements...
}
```

Pattern Matching

Relational and Logical Pattern





Pattern Matching

Property Pattern

C# 9

Pattern Matching

Property Pattern:

```
//Check whether the variable is of specified 'class_name' type
variable  switch
{
   class_name  another_variable when
     | { property:  value }        //another_variable.property == expression
     | { property:  < value }      //another_variable.property < value
     | { property:  > value }      //another_variable.property > value
     | { property:  <= value }     //another_variable.property <= value
     | { property:  >= value }     //another_variable.property >= value
       => result_expression...
}
```


Pattern Matching

Tuple Pattern

C# 9

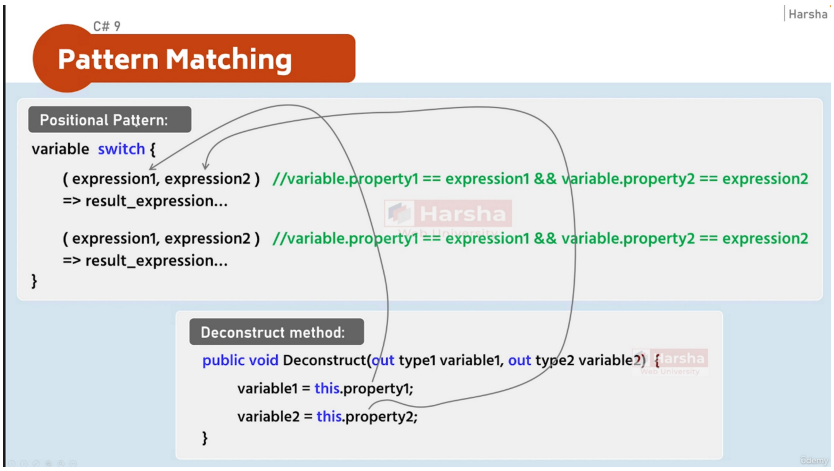
Pattern Matching

Tuple Pattern:

```
(variable.property1, variable.property2)  switch
{
     ( expression1, expression2 )  //variable.property1 == expression1 && variable.property2 == expression2
     => result_expression...

     ( expression1, expression2 )  //variable.property1 == expression1 && variable.property2 == expression2
     => result_expression...
}
```


Pattern Matching

Positional Pattern

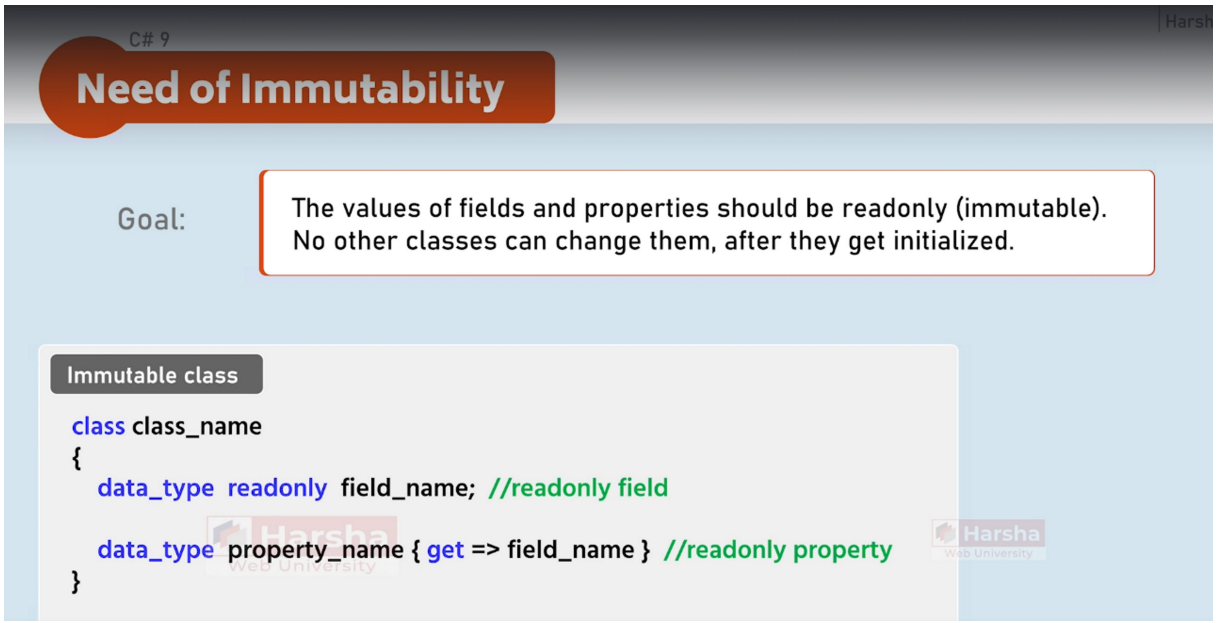

C# 9
Harsha
Pattern Matching
Positional Pattern:
variable switch {
(expression1, expression2) //variable.property1 == expression1 && variable.property2 == expression2
=> result_expression...
(expression1, expression2) //variable.property1 == expression1 && variable.property2 == expression2
=> result_expression...
}
Deconstruct method:
public void Deconstruct(out type1 variable1, out type2 variable2) {
variable1 = this.property1;
variable2 = this.property2;
}

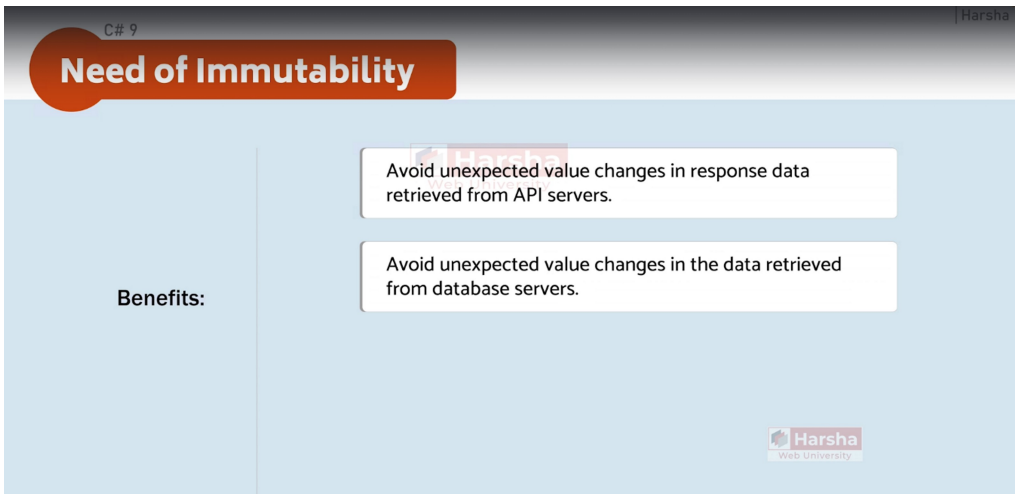

Pattern Matching
Extended Property Pattern

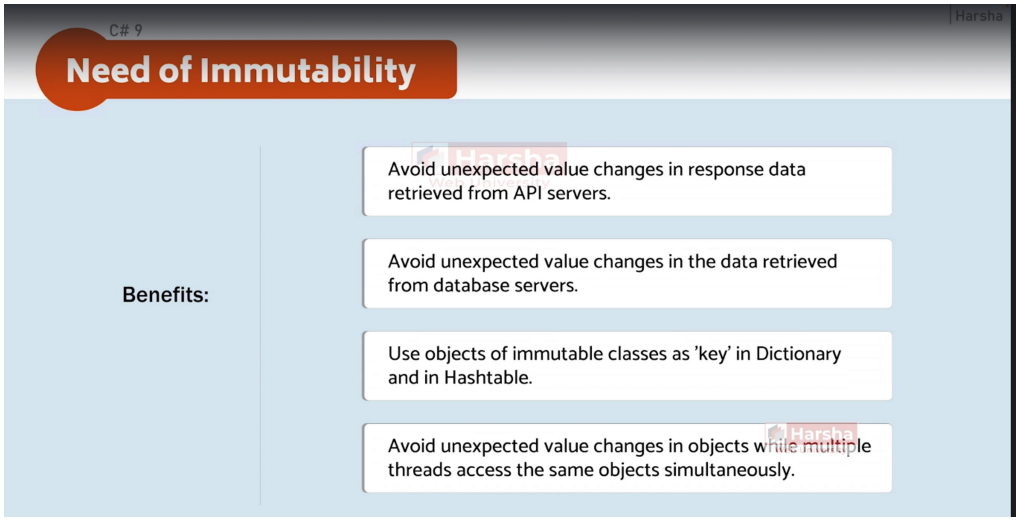

ended Property Pattern
C# 9
Harsha
Pattern Matching
Nested Property Pattern:
variable switch
{
{ outer_property: { nested_property: value } } //variable.outer_property.nested_property == expression
{ outer_property: { nested_property: < value } } //variable.outer_property.nested_property < value
{ outer_property: { nested_property: > value } } //variable.outer_property.nested_property > value
{ outer_property: { nested_property: <= value } } //variable.outer_property.nested_property <= value
{ outer_property: { nested_property: >= value } } //variable.outer_property.nested_property >= value
=> result_expression...
}

C#

Need of Immutability

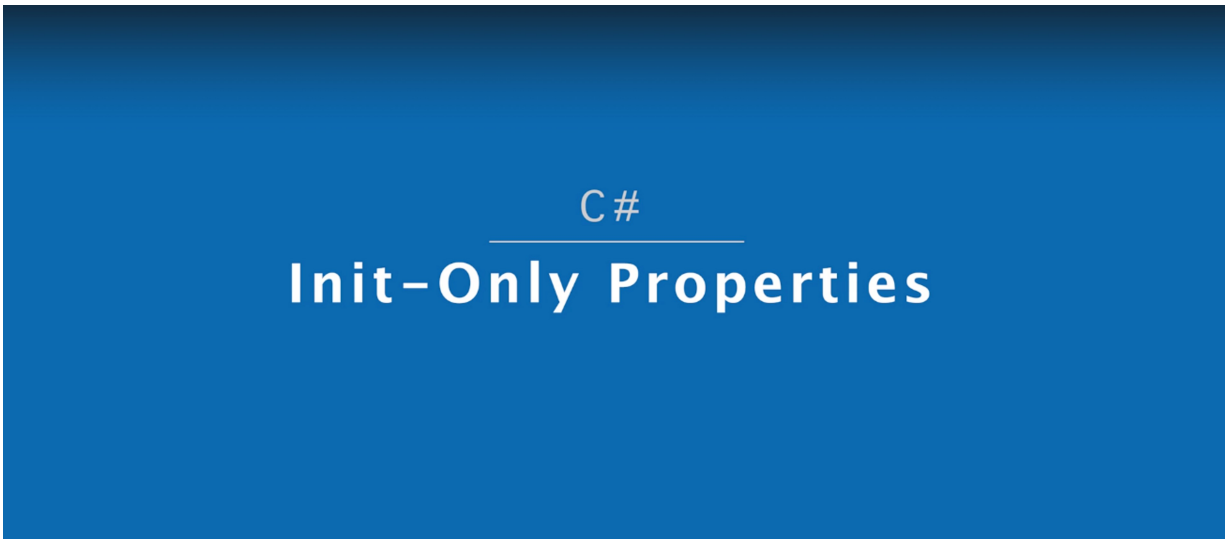




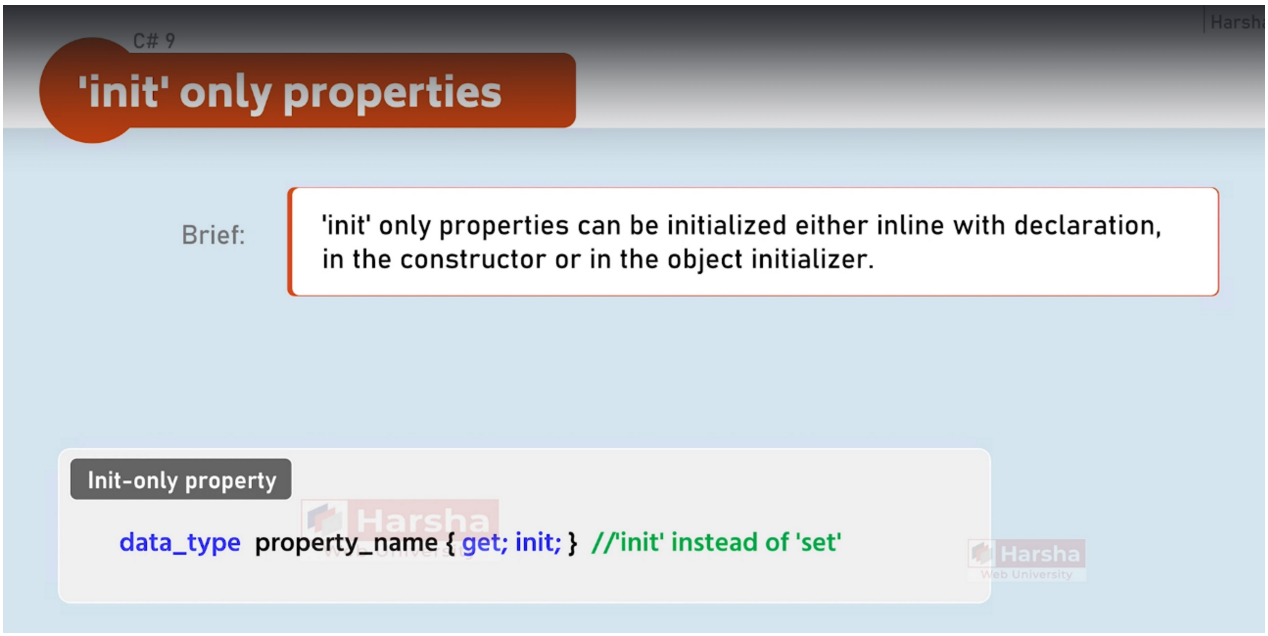

C# 9
Harsha
Need of Immutability
Benefits:
Avoid unexpected value changes in response data retrieved from API servers.
Avoid unexpected value changes in the data retrieved from database servers.
Use objects of immutable classes as 'key' in Dictionary and in Hashtable.
Avoid unexpected value changes in objects while multiple threads access the same objects simultaneously.

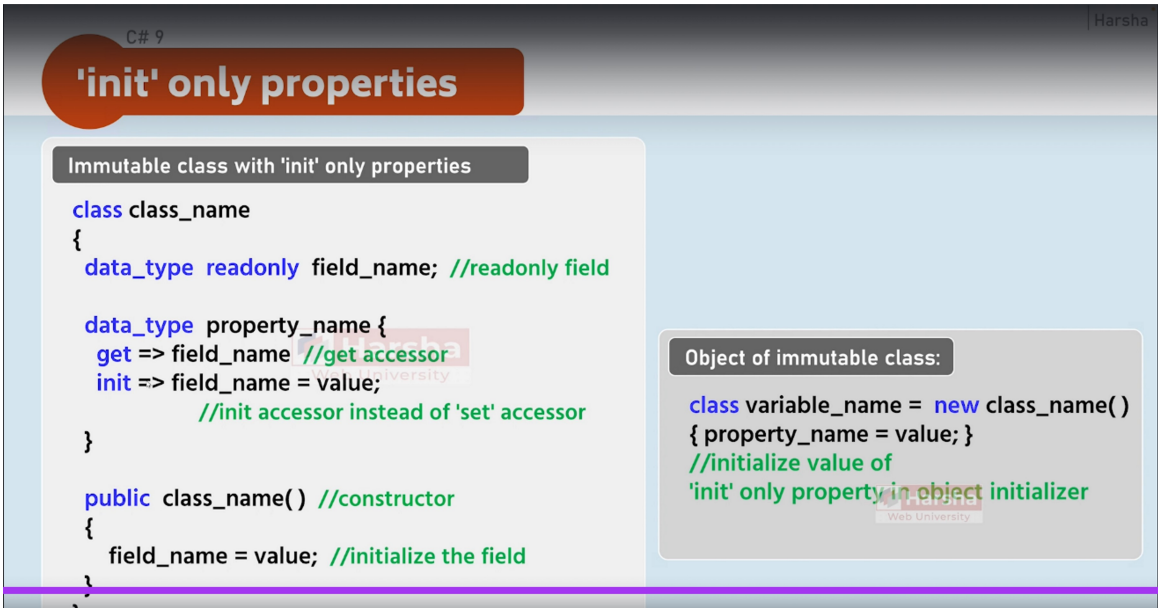

C#
Immutable Classes

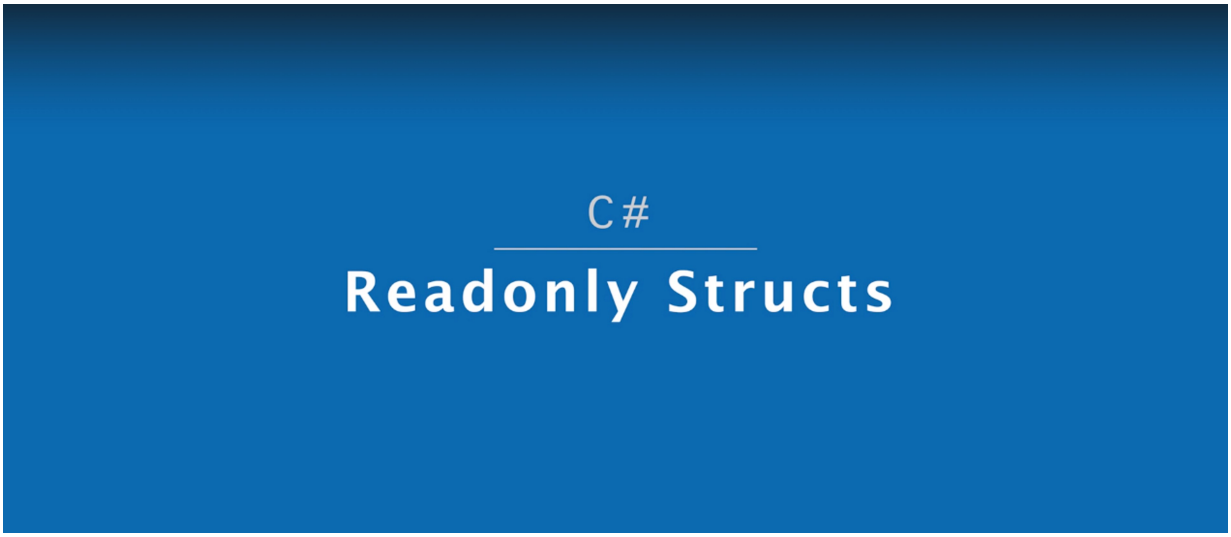

C# 9
Harsha
Immutable classes
Brief:
A class with readonly fields and readonly properties.
Immutable class
class class_name
{
data_type readonly field_name; //readonly field
data_type property_name { get => field_name } //readonly property
public class_name( ) //constructor
{
field_name = value; //initialize the field
}
}

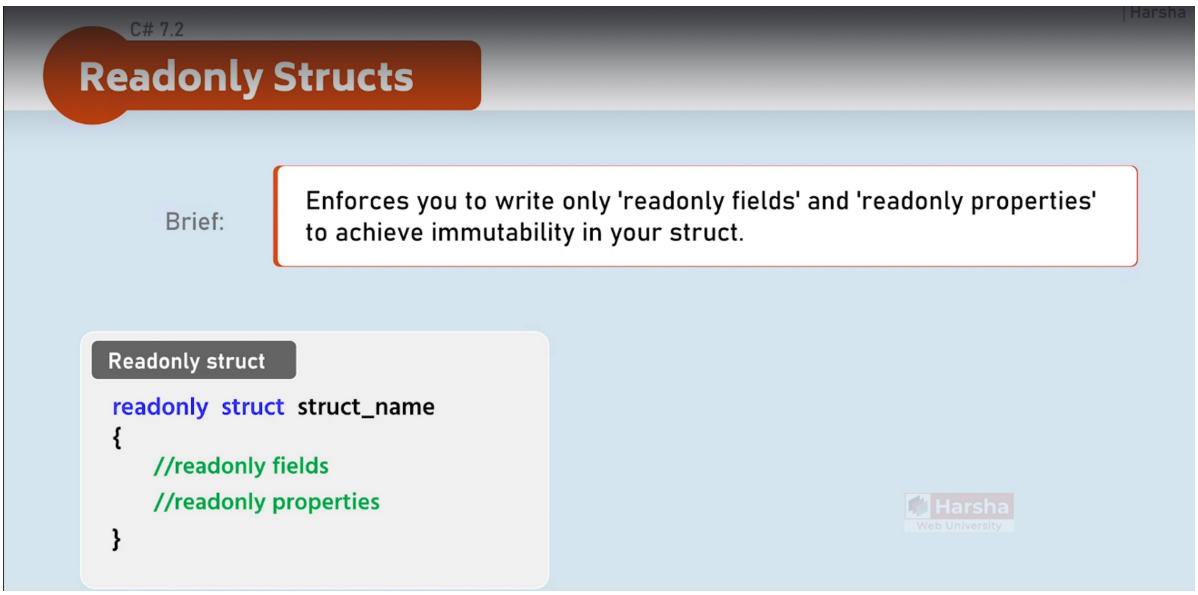


There is a problem with Read only properties. You can not initialize the value of the read-only property in the object initializer. To overcome this problem, C# 9 introduces 'init' only properties.



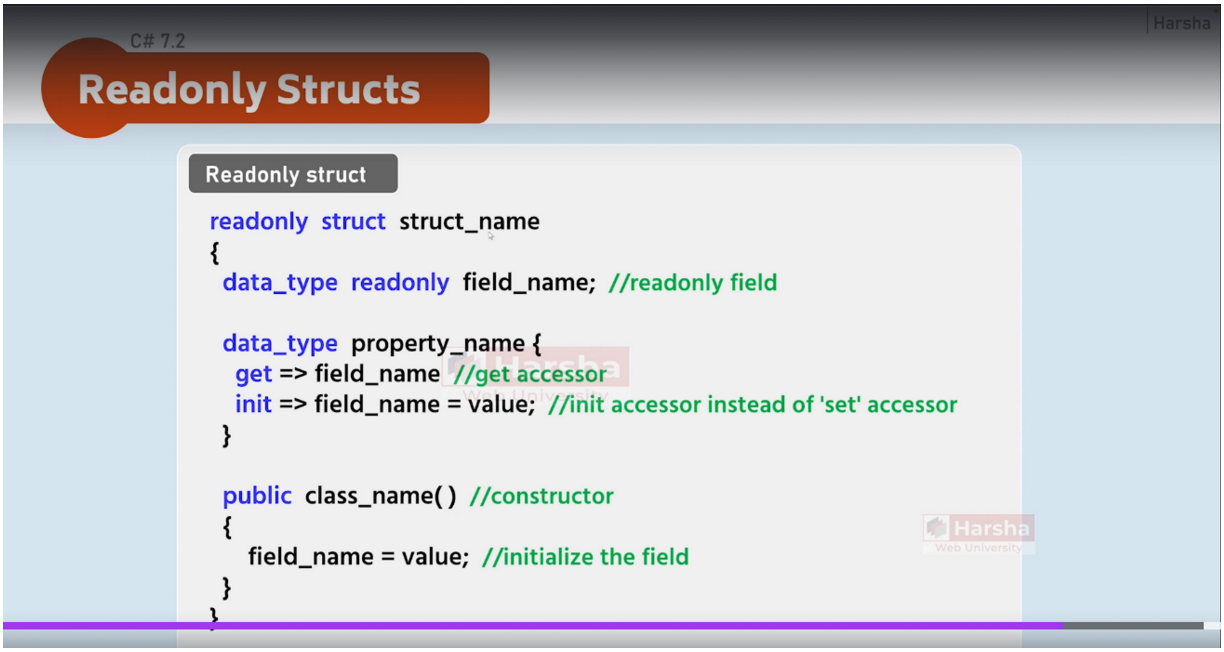


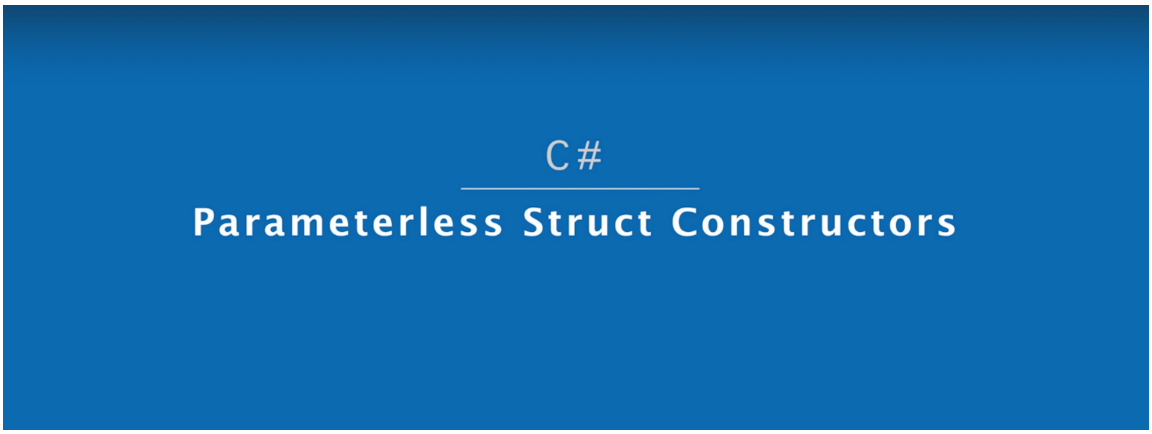




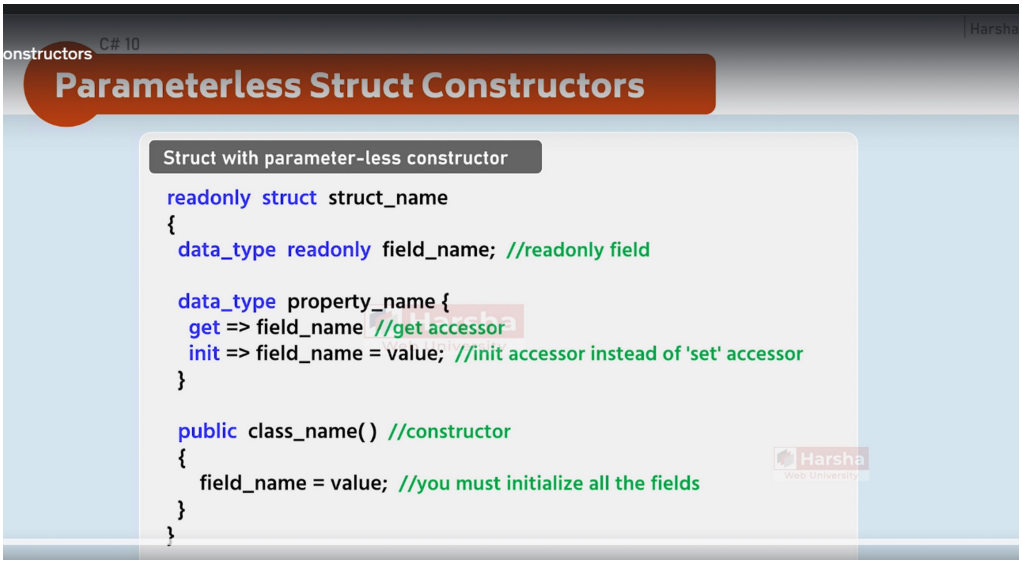
In general, structures are better suited for representing immutable objects because they are not reference-based. Structure instances contain

data directly, and it's best to make them read-only.
This doesn't mean structures are inherently read-only, but they are ideal for representing read-only data or immutable objects.
In C# 7.2, a new concept called **readonly structures** was introduced. This feature enforces all fields and properties in the structure to be read-only. If you attempt to write to a normal field or property in a readonly structure, a compile-time error is generated, ensuring that the code's intention is clearly expressed.





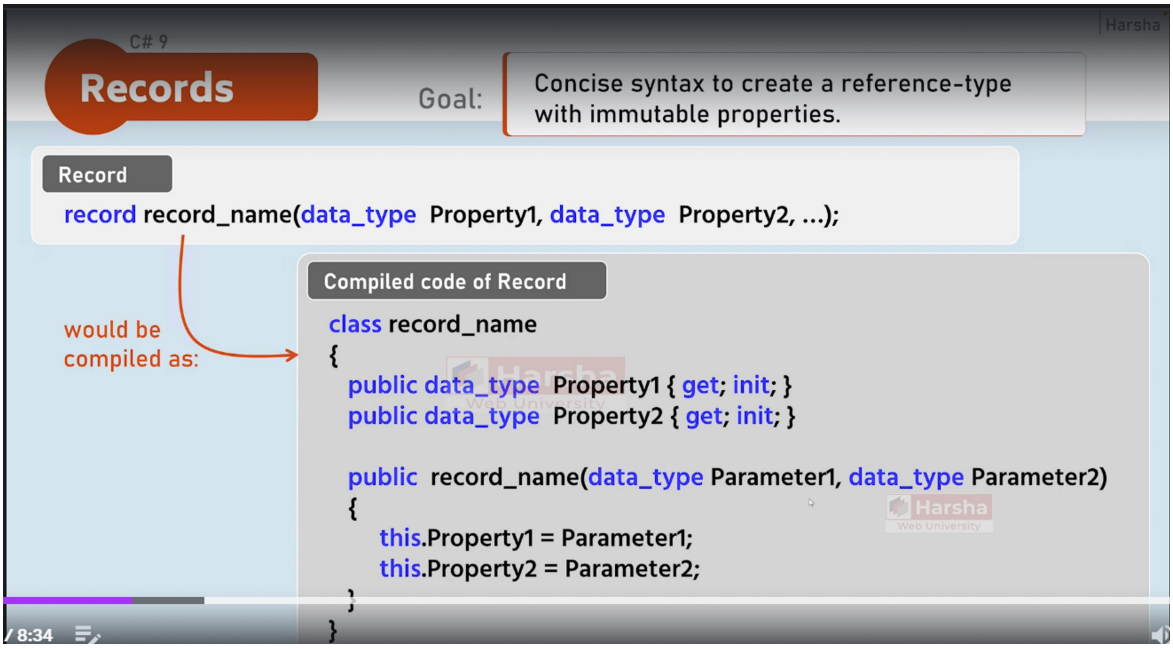
Before C# 10, it was not possible to create a parameterless constructor in a structure.
With C# 10, it is possible, but with a condition: you must initialize all fields in the parameterless constructor.
For example, if your structure has 10 fields, you must initialize all of them in the constructor.
If you follow this rule, you can create a parameterless constructor in the structure.
The same rule applies to parameterized constructors as well.

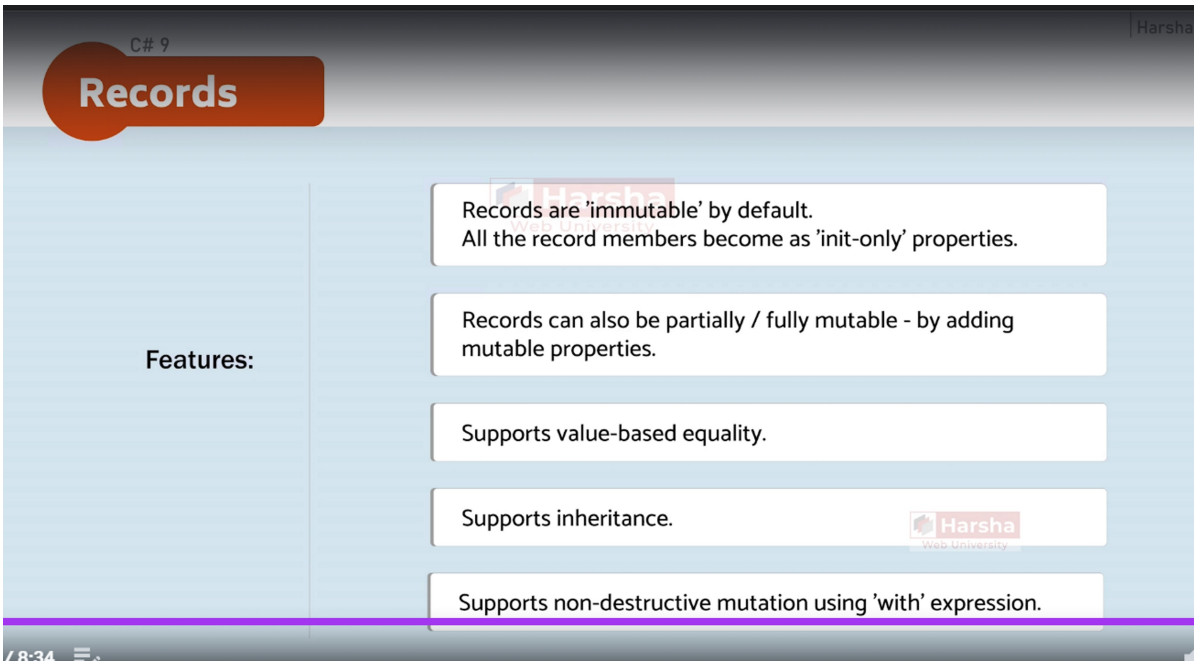


Records are a new feature introduced in C# 9.
The goal of records is to allow developers to create immutable reference types using a concise syntax.
For example, instead of writing a lengthy class with properties and constructors, you can achieve the same functionality in a single line with a record.
To create a record, use the record keyword followed by the record name, similar to a class declaration. Then, specify the data types and property names for all the properties you want.
Example:

- A Person record might have properties like Name and DateOfBirth.
- An Employee record might include EmployeeId, Name, and Address.

These properties are declared in a single line, and the compiler automatically converts the record into a class. Each property, defined as comma-separated, becomes an independent property.
Special Features of Records:
1. By default, all properties in a record are **init-only**, meaning they have get and init accessors but no set.
2. The compiler automatically generates a constructor that initializes all properties.
In summary, records provide a shortcut syntax for defining immutable types, replacing lengthy class definitions with a single line of code.

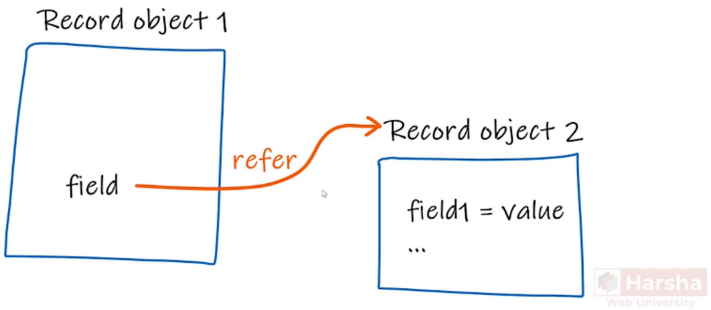

C# 9
Records
Goal:
Concise syntax to create a reference-type with immutable properties.
Record
record record_name(data_type Property1, data_type Property2, ...);
would be compiled as:
Compiled code of Record
class record_name
{
public data_type Property1 { get; init; }
public data_type Property2 { get; init; }
public record_name(data_type Parameter1, data_type Parameter2)
{
this.Property1 = Parameter1;
this.Property2 = Parameter2;
}
}
8:34


C# 9
Records
Features:
Records are 'immutable' by default.
All the record members become as 'init-only' properties.
Records can also be partially / fully mutable - by adding mutable properties.
Supports value-based equality.
Supports inheritance.
Supports non-destructive mutation using 'with' expression.
8:34

C# - Records

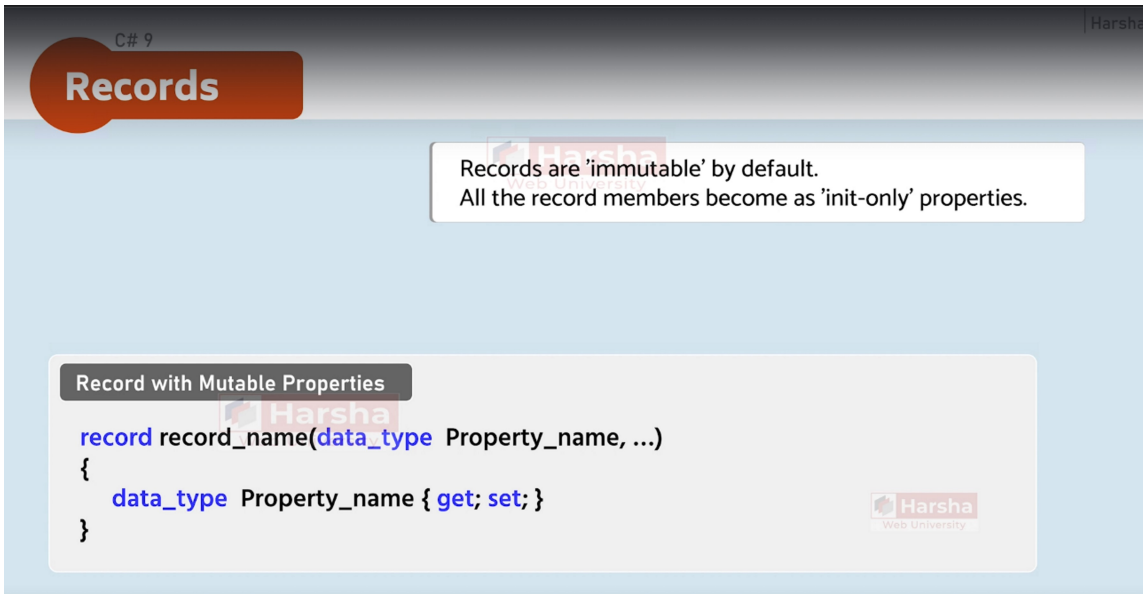
Nested Records

Nested Records



C# - Records

Immutability



Here's the trimmed and formatted version of your text:

By default, records in C# support **value-based equality**.

What does this mean?

When you create a record, the compiler automatically:
1. Overloads the == (equal to) and != (not equal to) operators.
2. Provides an Equals method.
These allow you to compare the values of two record objects.

How does it work?

If the values of all properties in two record objects are the same, the comparison will return true. Otherwise, it will return false.
This makes records particularly useful for scenarios where you need to compare objects based on their content rather than their reference.

C# 9

Records

Harsha

Supports value-based equality.

Record - 'Equals' method

```
record_name  variable1 = new record_name(value1, value2);
record_name  variable2 = new record_name(value1, value2);
variable1 == variable2;  //true
variable1.Equals(variable2);  //true
```

C# 9

Records

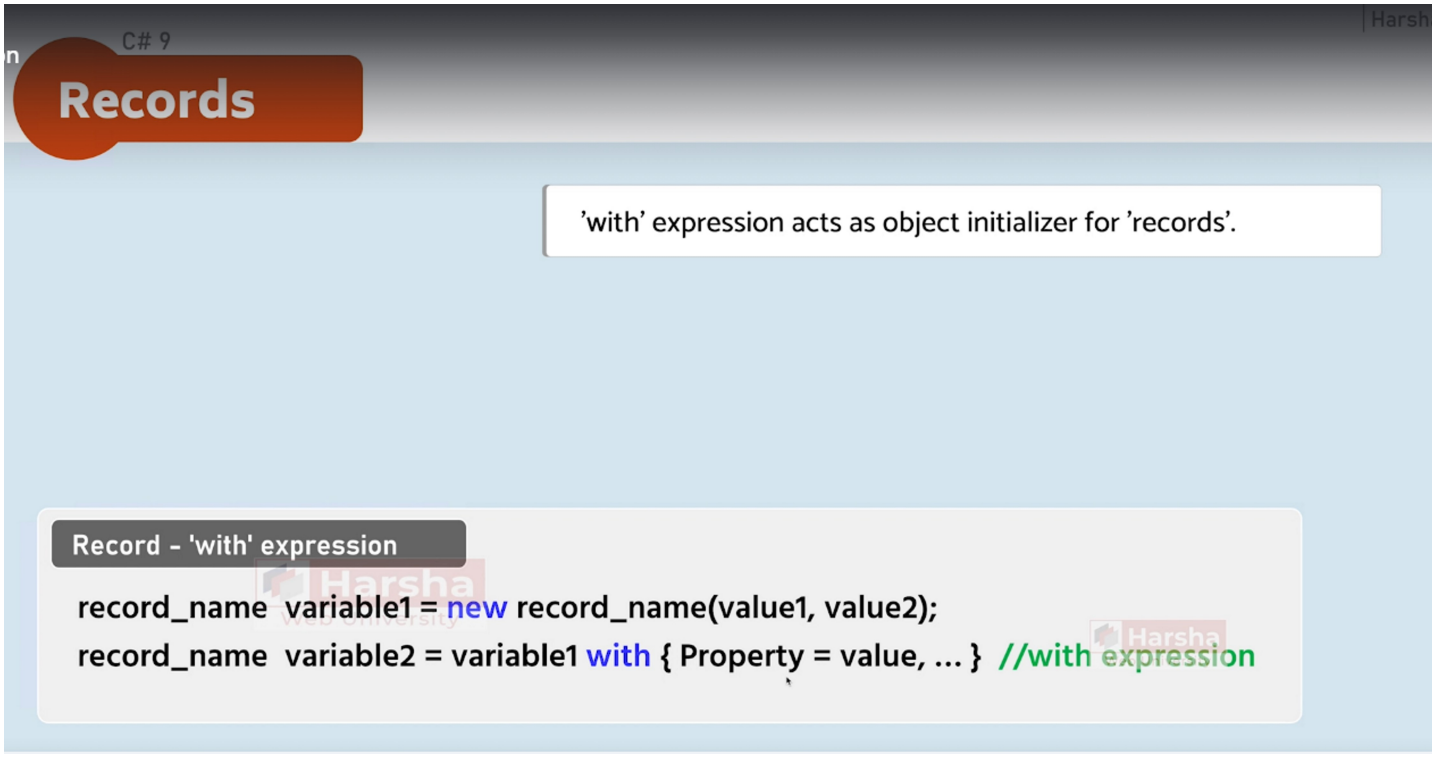
Harsha

Supports value-based equality.

Records provide a compiler-generated Equals() method and overloads == and != operators that compares two instances of records that compare the **values** of fields (but doesn't compare references) .

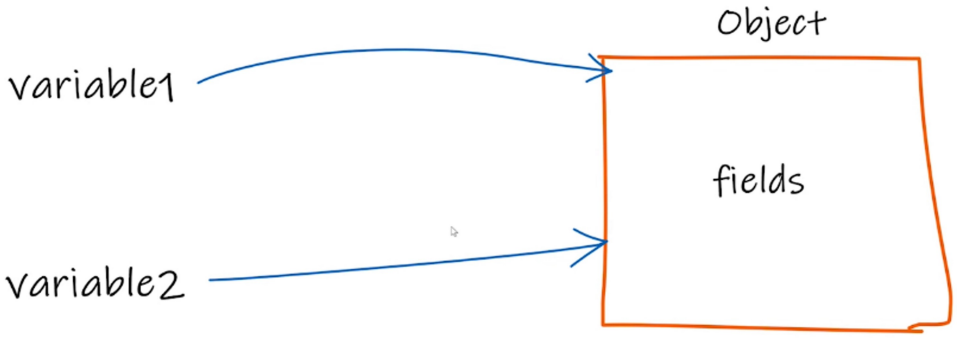
Record - 'Equals' method

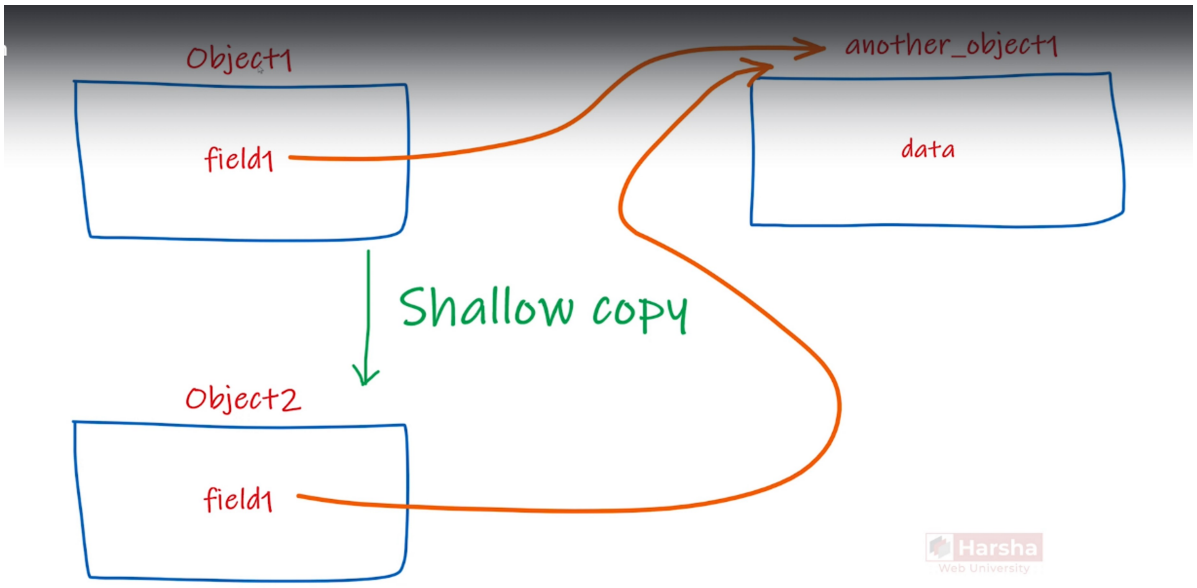
```
record_name  variable1 = new record_name(value1, value2);
record_name  variable2 = new record_name(value1, value2);
variable1 == variable2;  //true
variable1.Equals(variable2);  //true
```

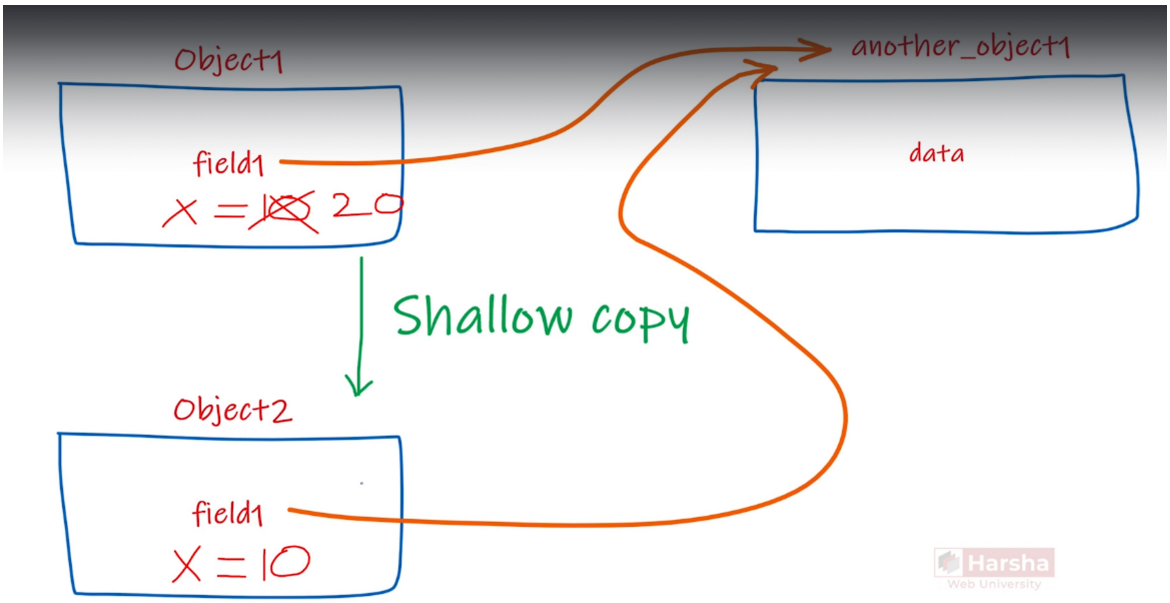


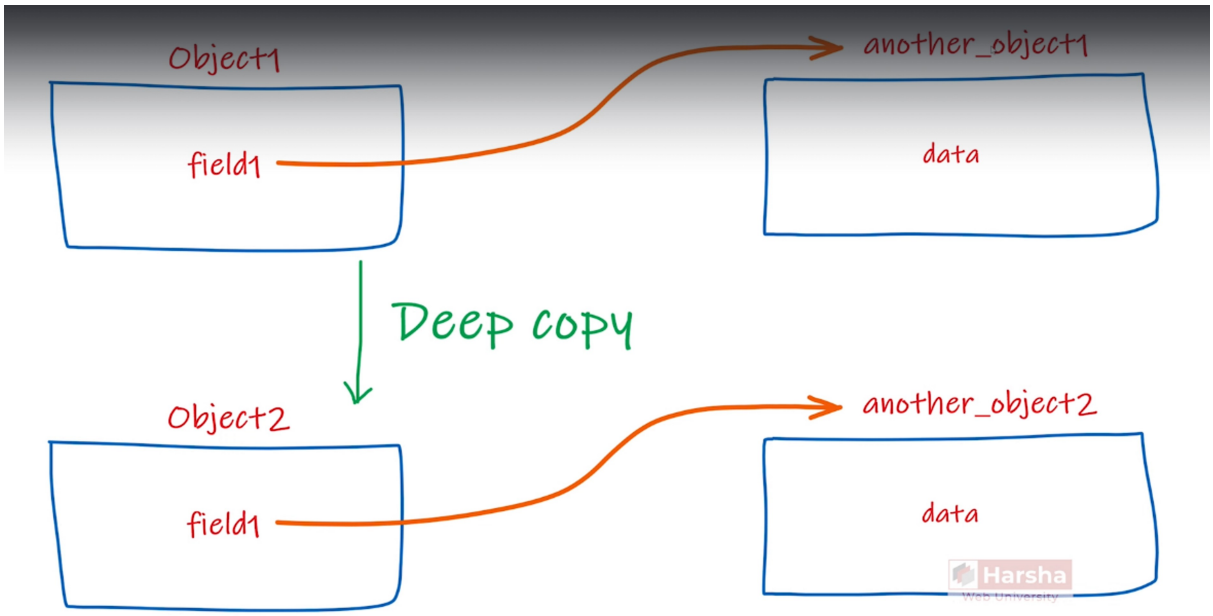
Reference Copy

variable2 = variable1

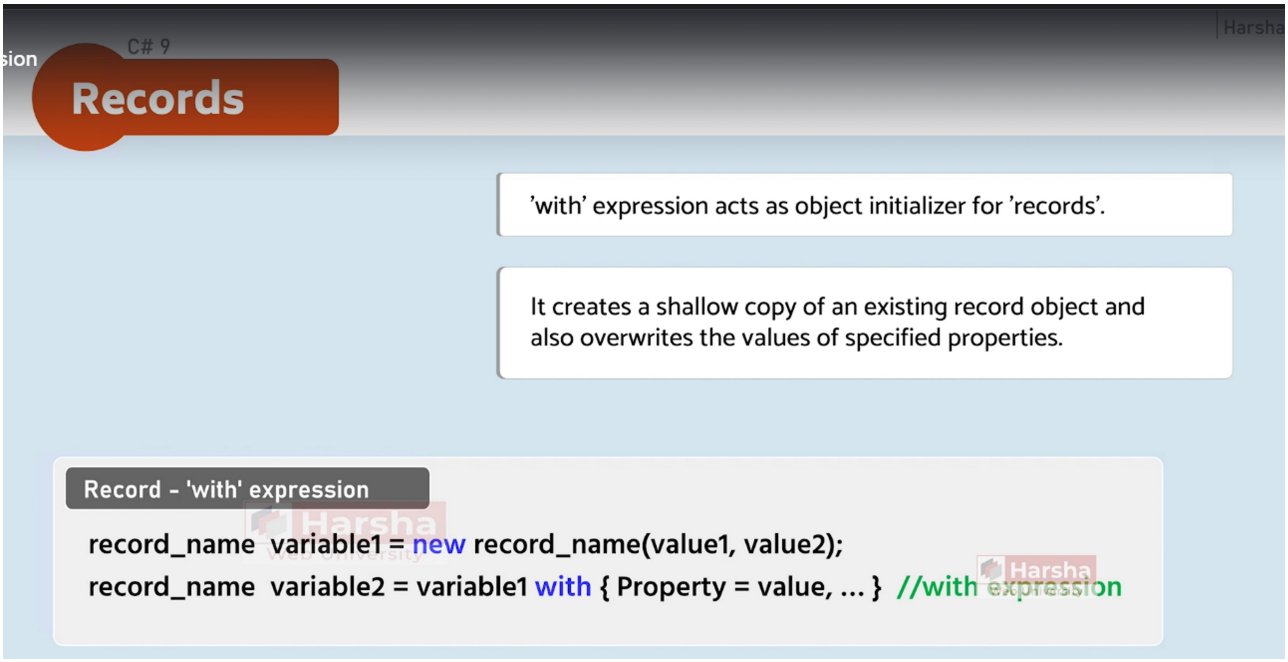








There is no straight forward built-in way to create 'Deep Copy'



What is Deconstruction?

Deconstruction refers to copying the values of an object's fields into independent variables.

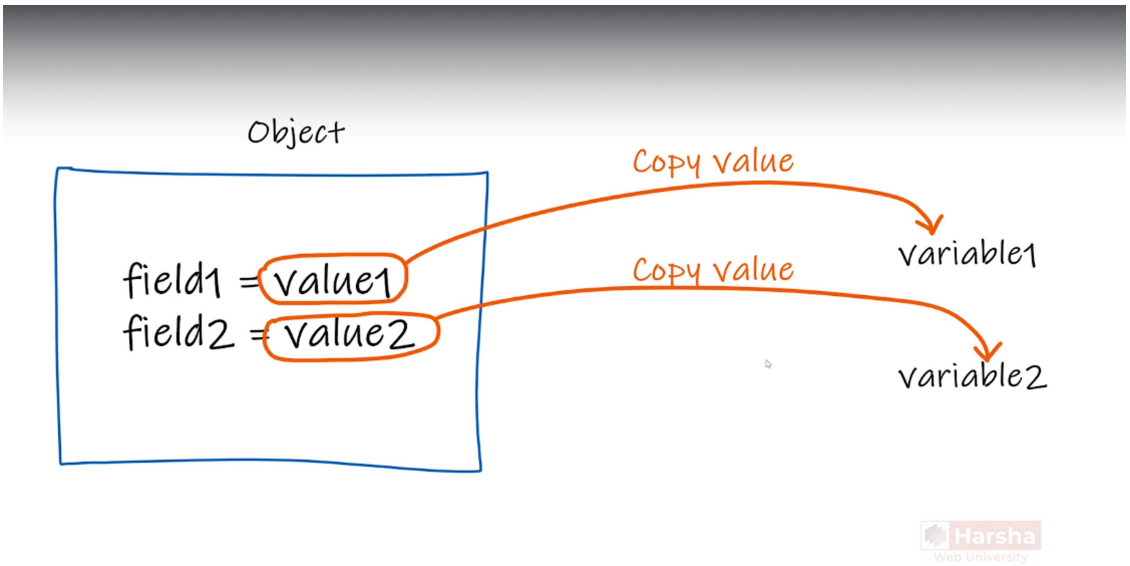
Example.

An object has multiple fields, each with a value. Using deconstruction, you can assign:
- Field 1's value to Variable 1.
- Field 2's value to Variable 2, and so on.

This process of extracting field values into separate variables is called **deconstruction**.

How to Implement Deconstruction?

To enable deconstruction, a method called Deconstruct is required. In C#, records provide the Deconstruct method by default, allowing you to seamlessly extract values from record objects into individual variables.



C# 9

Records

A compiler-generated 'Deconstruct' method is provided for all records that returns all property values as a tuple.

It is useful while reading few specific set of properties from a record object.

Record - deconstruct

```
record_name reference_variable = new record_name(value1, value2);
var (variable1, variable2, ...) = reference_variable;
```

C# - Records

ToString()

Harsha

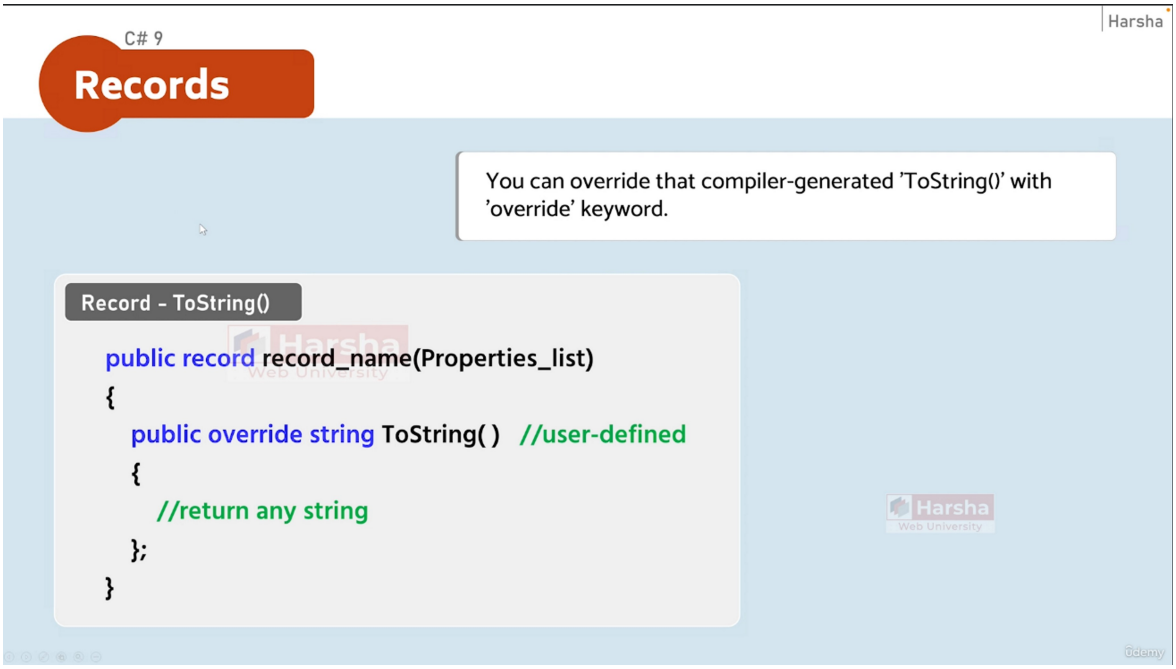
C# 9

Records

A compiler-generated 'ToString()' is provided for all records that returns a string with all properties and values.

Record - ToString()

```
public record record_name(Properties_list)
{
   public override string ToString( )   //compiler-generated
   {
      //returns a string:   Record_Name { Property1 = value1, Property2 = value2, ... }
   };
}
```


Harsha
C# 9
Records
You can override that compiler-generated 'ToString()' with 'override' keyword.
Record - ToString()
public record record_name(Properties_list)
{
public override string ToString( ) //user-defined
{
//return any string
};
}
Harsha
Web University
Udemy


C# - Records
Constructor

C# 9

Records

A compiler-generated 'parameterized constructor' is provided for all records that initializes all property values.

Record - User-Defined Constructor

```
public record_name(parameters): this(parameters)  //invokes compiler-generated constructor
{
   Property = value;
}
```

Harsha

C# 9

Records

A compiler-generated 'parameterized constructor' is provided for all records that initializes all property values.

You must invoke the compiler-generated constructor of the record with 'this' keyword, in case if you create your own constructor.

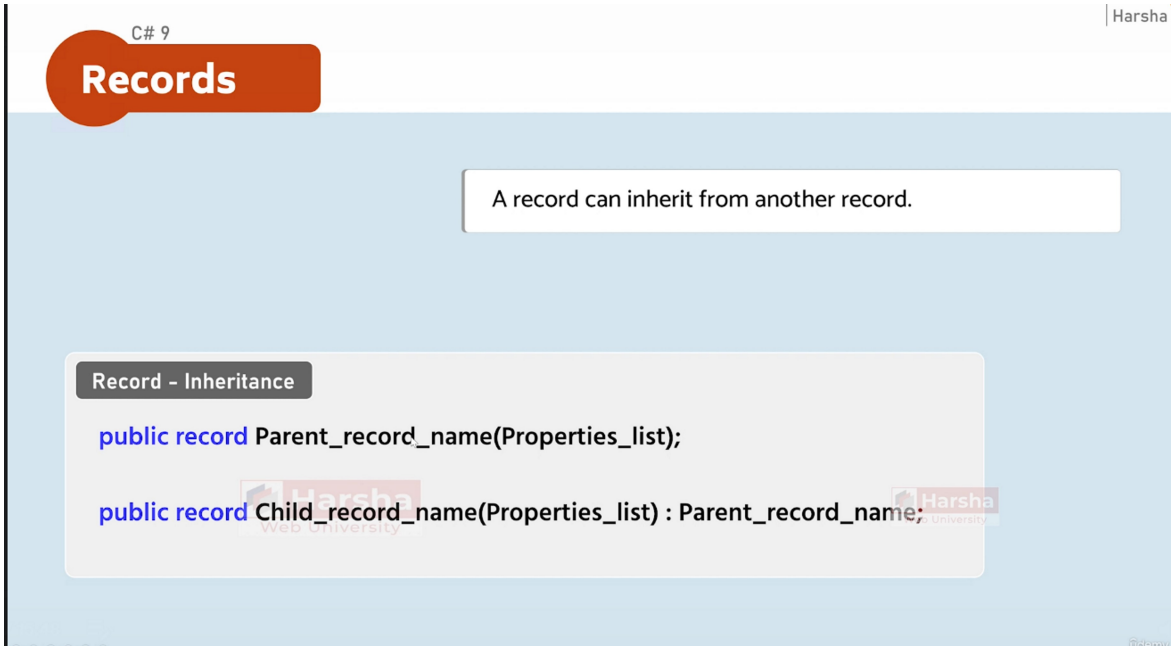
Record - User-Defined Constructor

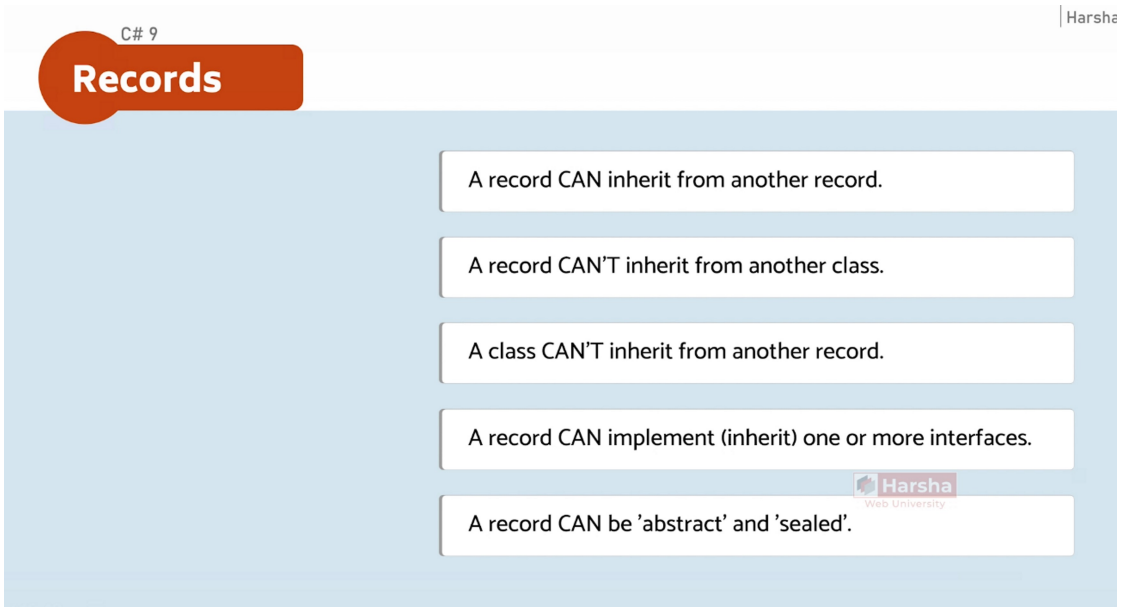
```
public record_name(parameters): this(parameters)  //invokes compiler-generated constructor
{
   Property = value;
}
```

C# - Records

Inheritance

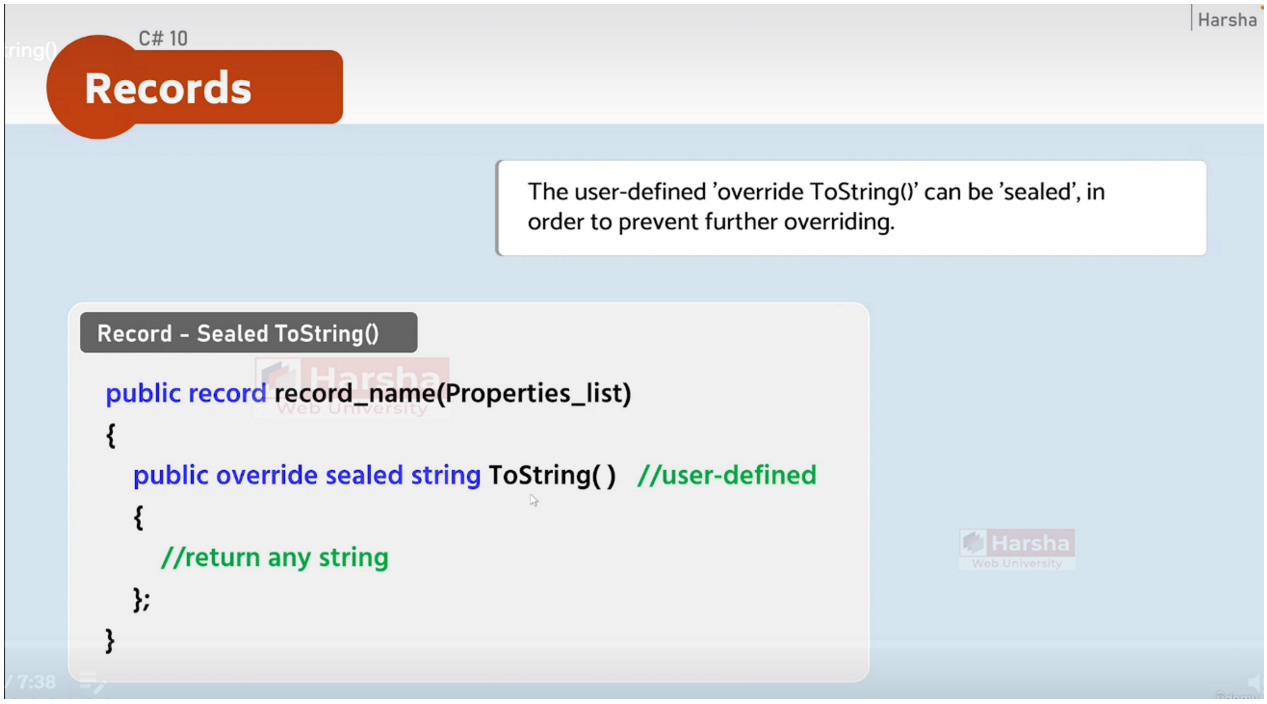
It is possible that a record can inherit from another record.





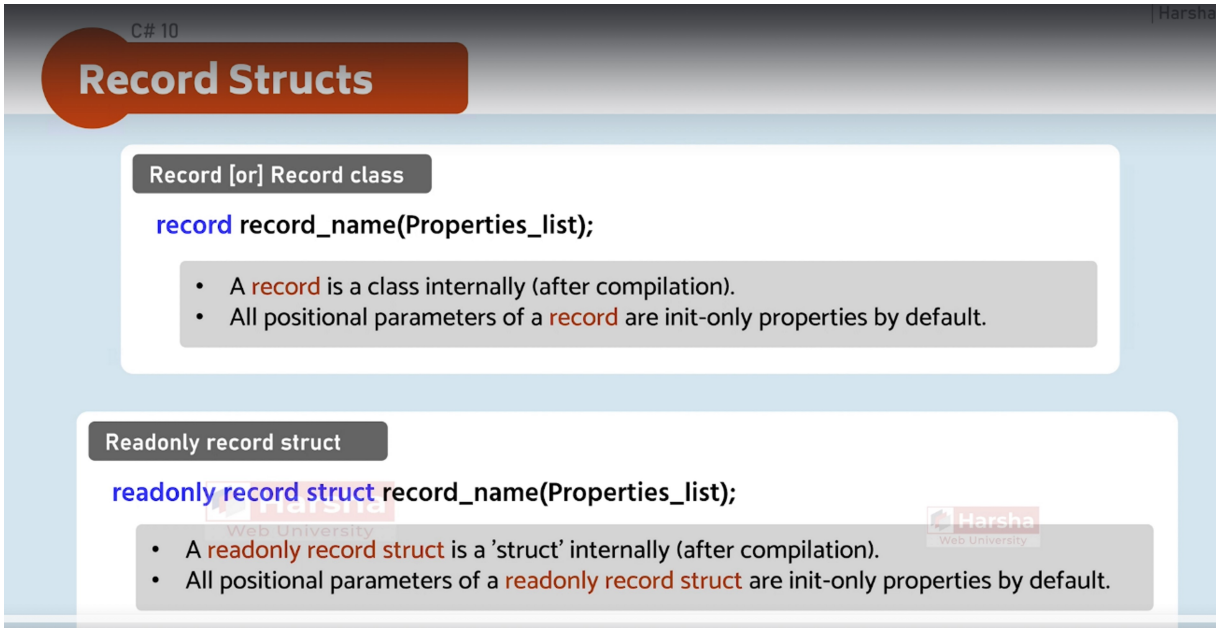
C# - Records

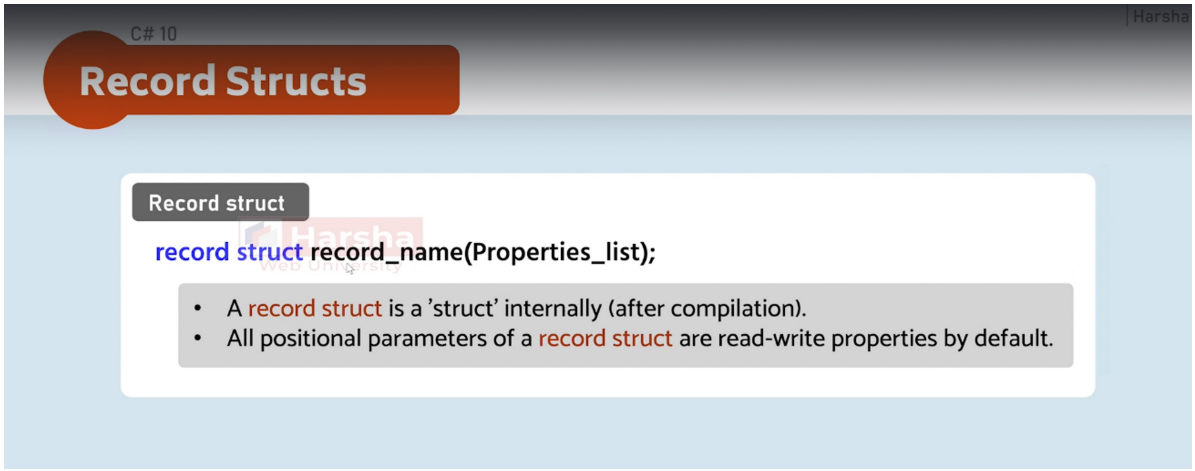
sealed ToString()



C# - Records

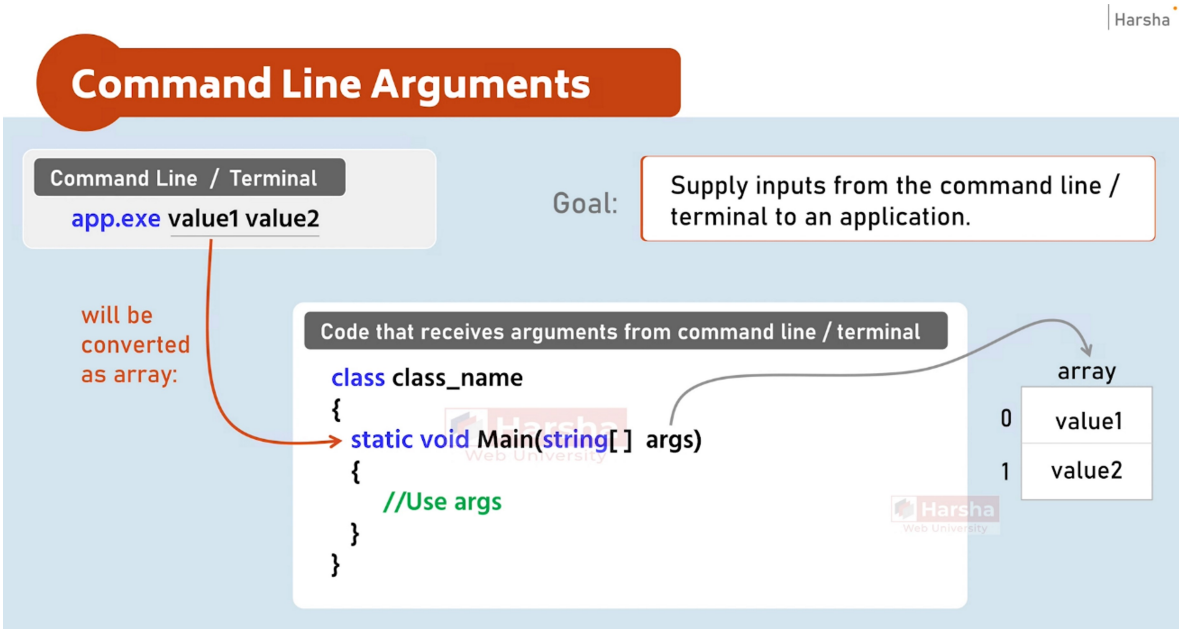
Record Structs



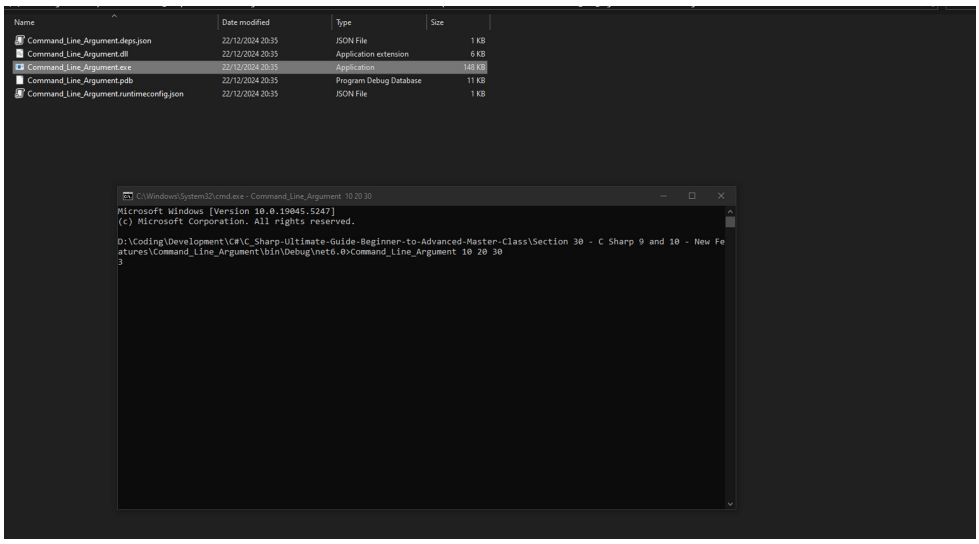


Part 1

Command Line Arguments



Generally, for any application, you can supply one or more input values while invoking that application. For example, you're trying to invoke (i.e., execute) an application, and before that application execution starts, you can supply one or more input values. The application's main method can receive the same. By default, the C# main method receives the arguments in the form of a string array. This means that when you supply one or more values to a C# application, the CLR will convert those values into a string array, which can be received into a variable of string array type. You can then write code to consume or utilize those values from the string array args. Note that the name args is not fixed—you can give it any other name. If you don't supply the values, by default, it contains an array with zero values.

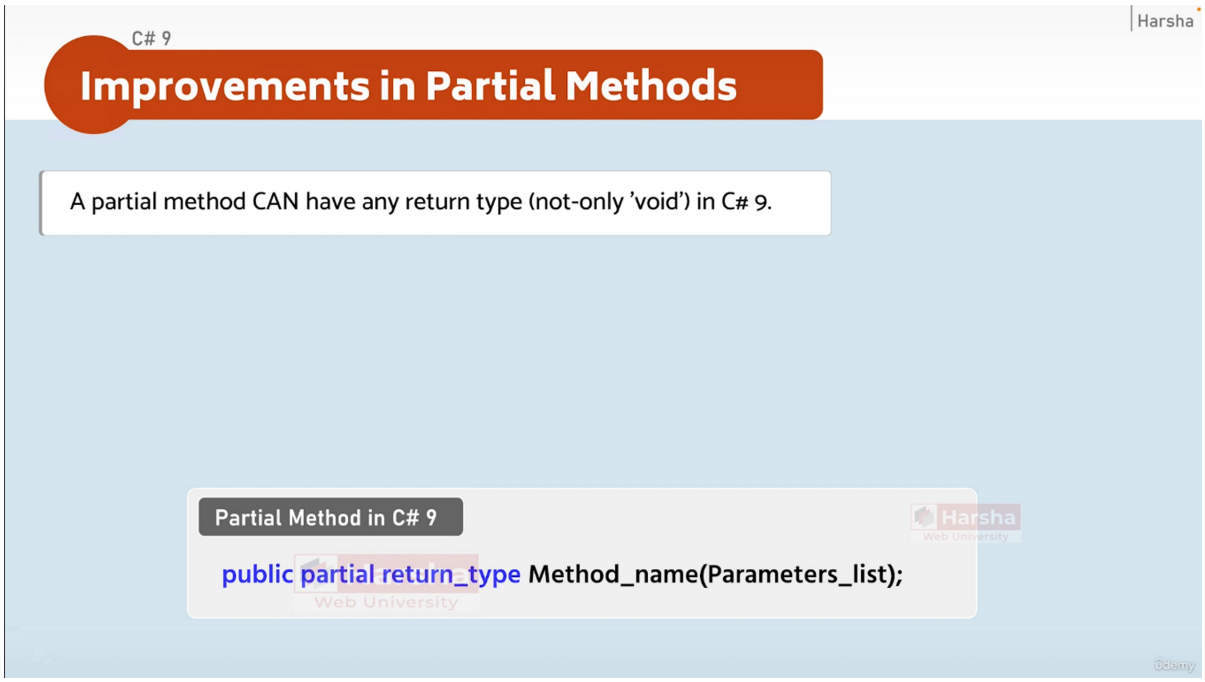


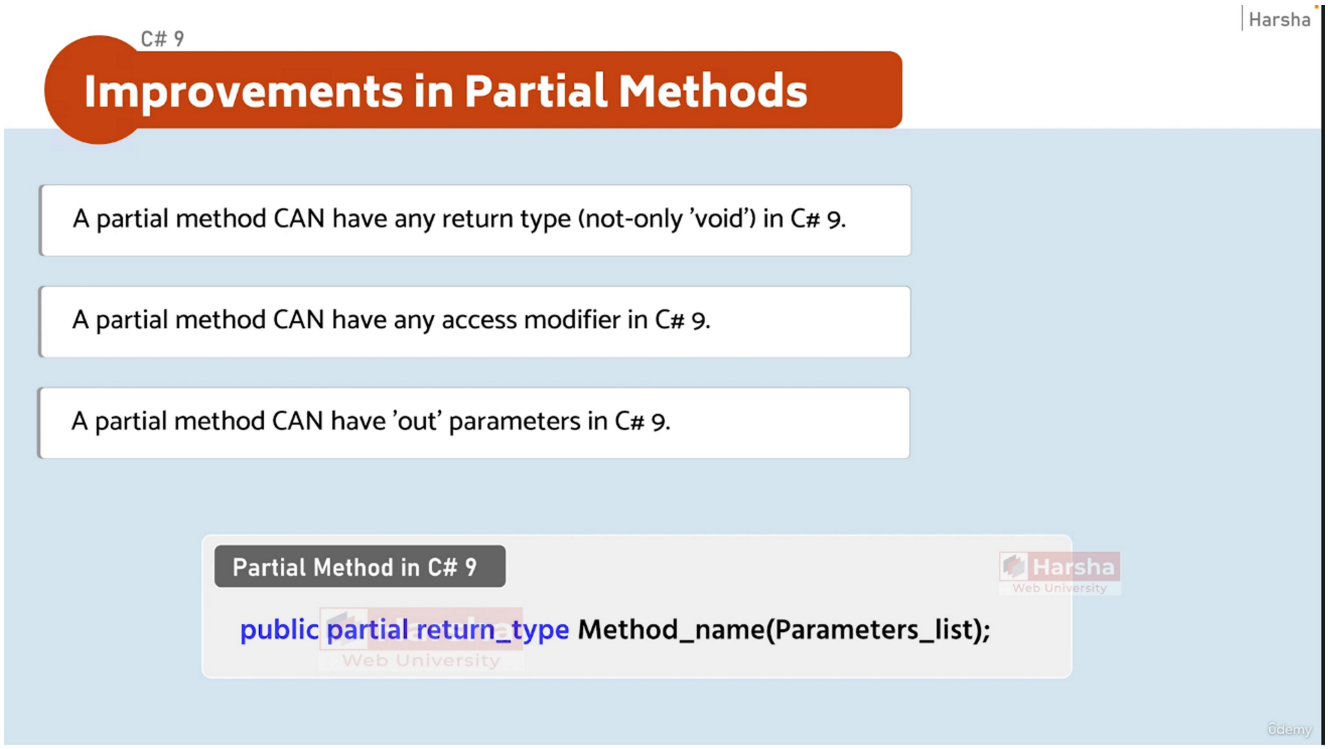
Part 2

Command Line Arguments

C#

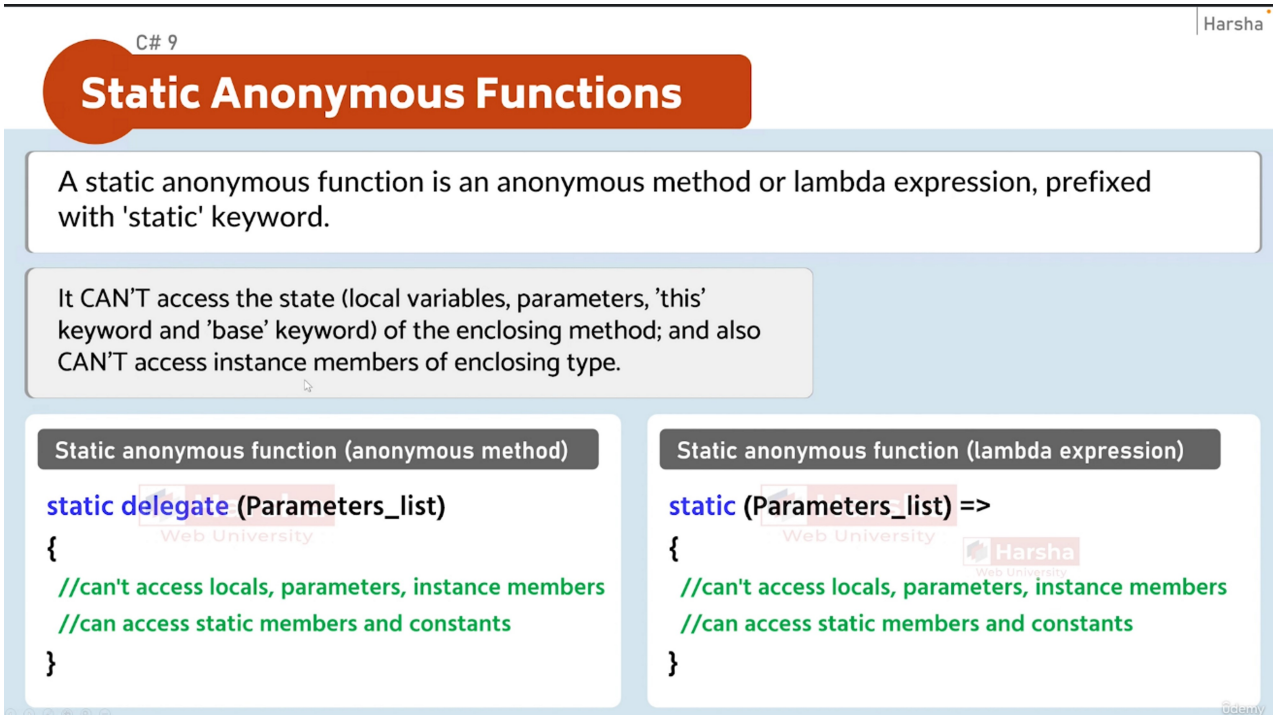
Partial Methods – Return Type


Harsha
C# 9
Improvements in Partial Methods
A partial method CAN have any return type (not-only 'void') in C# 9.
Partial Method in C# 9
public partial return_type Method_name(Parameters_list);
Harsha
Web University
Udemy


Harsha
C# 9
Improvements in Partial Methods
A partial method CAN have any return type (not-only 'void') in C# 9.
A partial method CAN have any access modifier in C# 9.
A partial method CAN have 'out' parameters in C# 9.
Partial Method in C# 9
public partial return_type Method_name(Parameters_list);
Harsha
Web University
Udemy

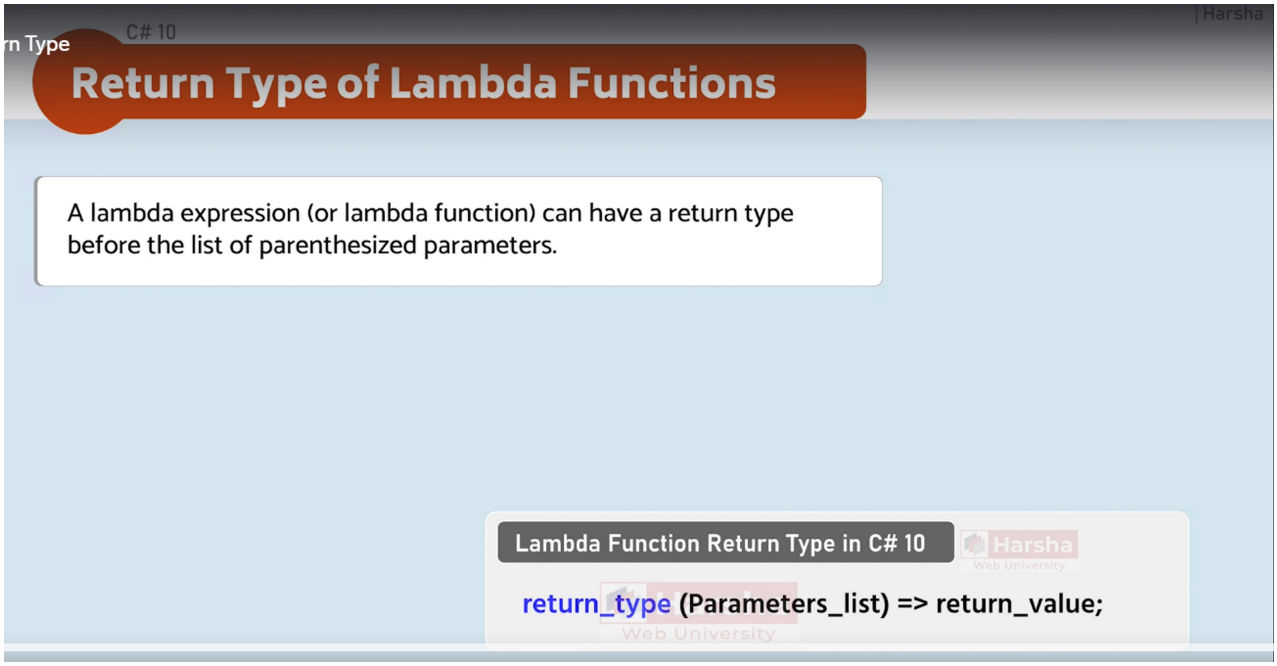
C#

Static Anonymous Functions



C#

Lambda Expression – Return Type



C#

Constant Interpolated Strings

Harsh

C# 10

Constant Interpolated Strings

Constant strings may be initialized using 'string interpolation' i.e. with ${}, if all the placeholders are constant strings.

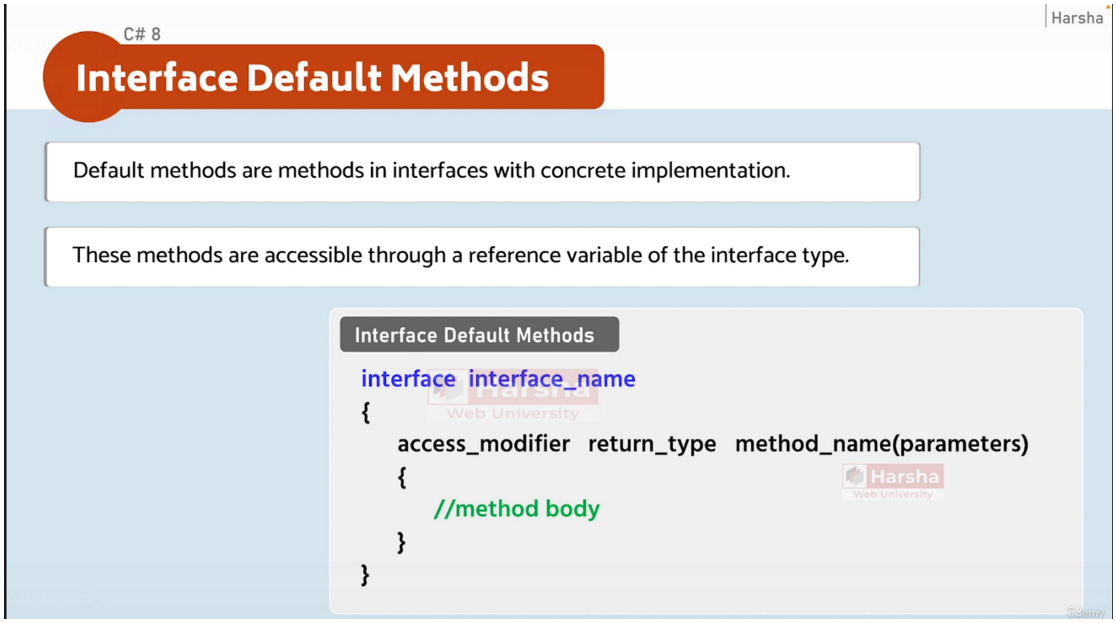
Eg: Useful when you are creating global API URLs.

Constant Interpolated Strings

```
const data_type variable_name = $"{constant_string}";
```

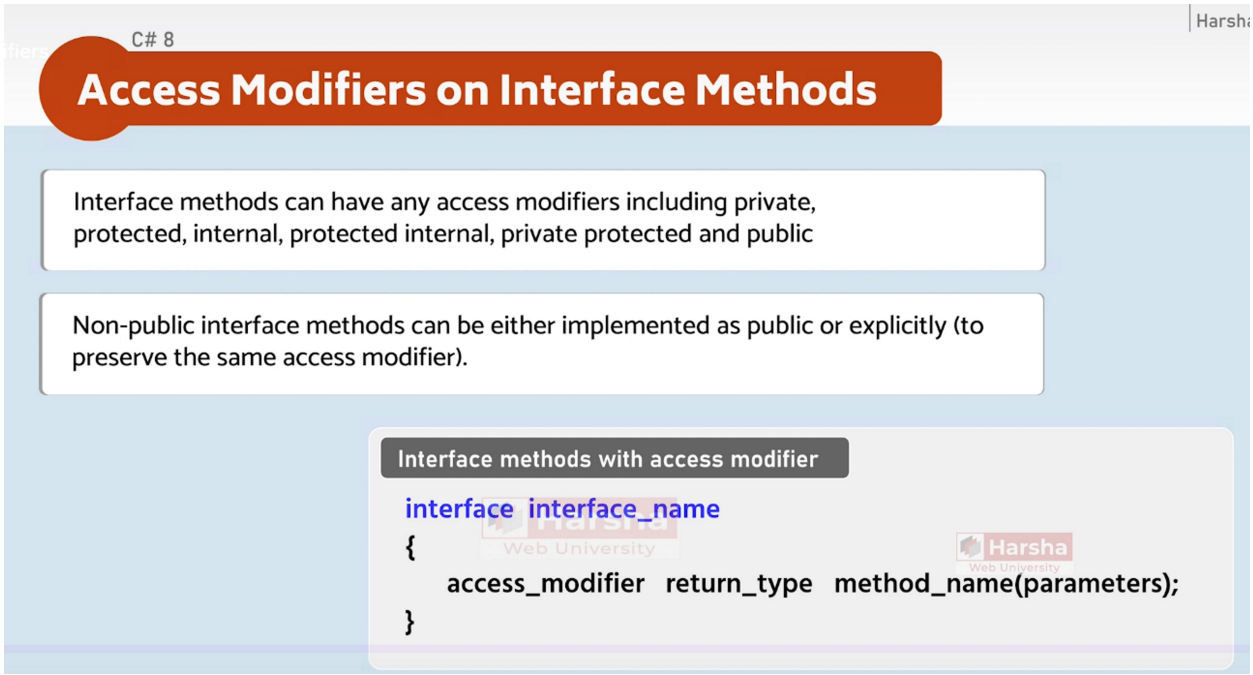
C#

Default Interface Methods



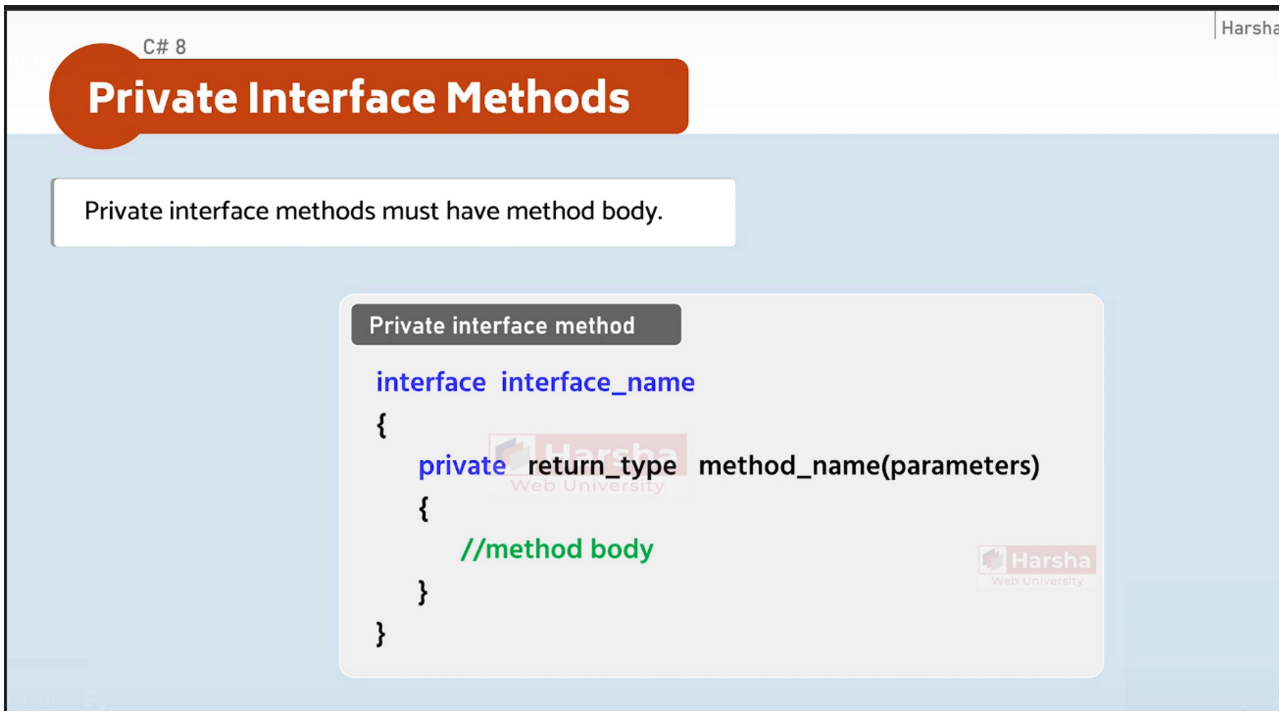
C#

Interface Method Modifiers



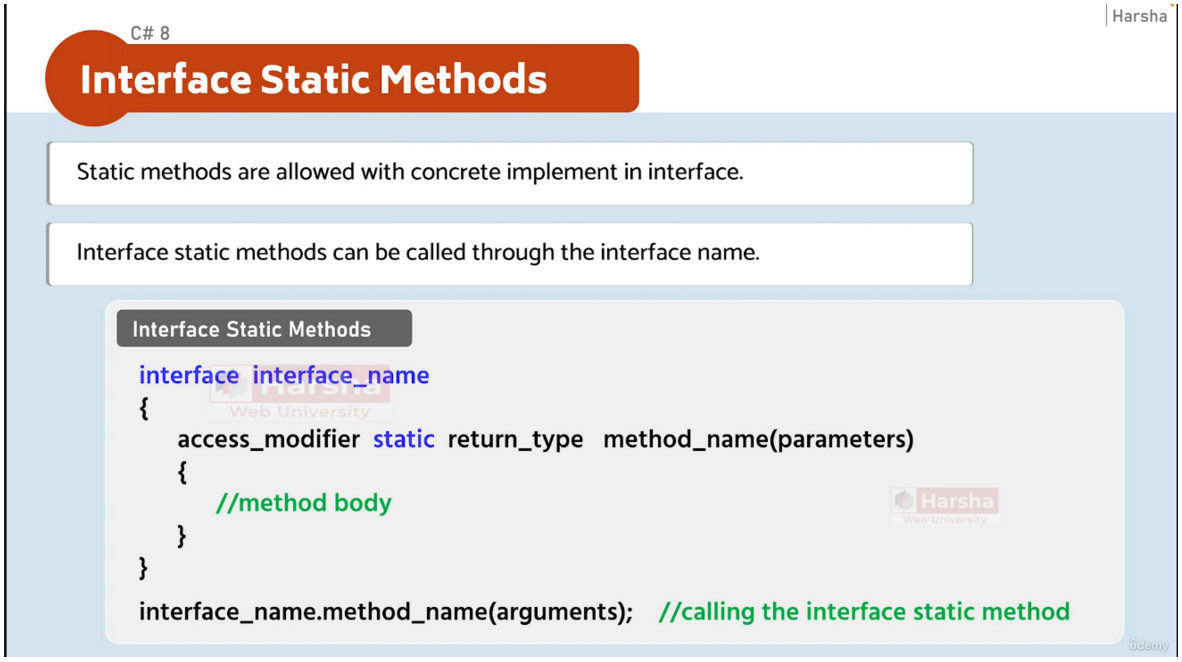
C#

Interface Private Methods



C#

Interface Static Methods

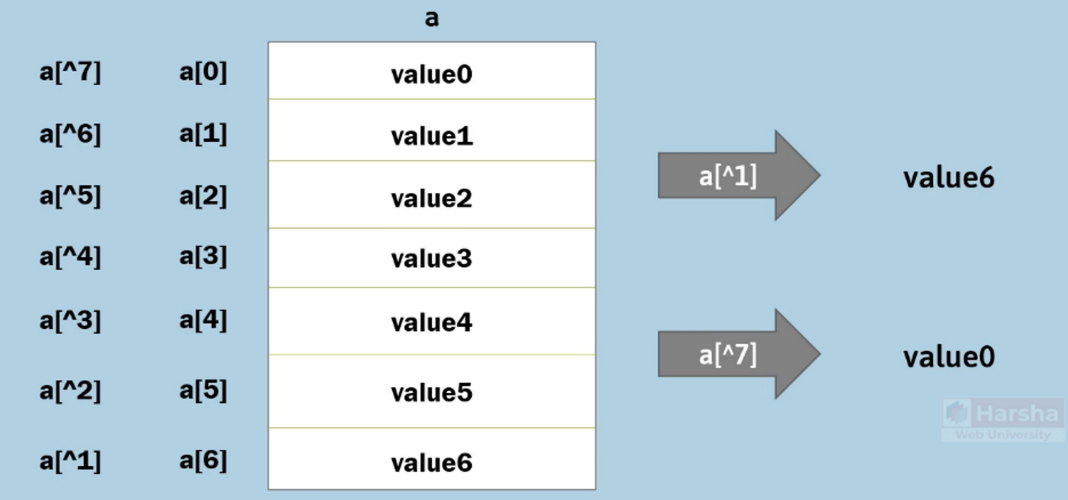


IndexFormEnd Operator

'IndexFromEnd' operator

^

› This operator returns the index from end of the array (index starts from '1').



Range_Struct

Range

'Range' struct

› The 'Range' struct represents "start index" and "end index" together.

› It can be used in conjunction with "indexer" is an array or collection.

› It is created using double dot (..) operator.

› It includes the "start index"; but excluding the "end index".