

Top-level Statements

Top Level Statements

```
statement1;  
statement2; | Top level statements
```

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



The screenshot shows a C# code editor in Visual Studio. The code includes top-level statements:

```
1 using System;  
2 using System.IO;  
3  
4 Console.WriteLine("Hello, World!");  
5 string name = "Harsha";  
6 Console.WriteLine($"Hello to {name}");  
7 Console.ReadKey();  
8  
9  
10 namespace namespace1  
11 {  
12     0 references  
13     class Sample  
14     {  
15         0 references  
16         public void Method1()  
17         {  
18             string str = name;  
19         }  
20     }  
21 }
```

A red oval highlights the first four lines of code, which are top-level statements. A red arrow points from this highlighted area to the 'Program' class definition in the code editor.

The screenshot shows a C# code editor in Visual Studio. The code includes top-level statements:

```
1 using System;  
2 using System.IO;  
3  
4 Console.WriteLine("Hello, World!");  
5 string name = "Harsha";  
6 Console.WriteLine($"Hello to {name}");  
7 Console.ReadKey();  
8  
9  
10 namespace namespace1  
11 {  
12     0 references  
13     class Sample  
14     {  
15         0 references  
16         public void Method1()  
17         {  
18             string str = name;  
19         }  
20     }  
21 }
```

A red oval highlights the first four lines of code, which are top-level statements. A red arrow points from this highlighted area to the 'Program' class definition in the code editor. In this version, the variable 'name' is explicitly assigned the value "Harsha" within the 'Main' method.

Top Level Statements

File1.cs
statements...

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

namespaces / types...

Would compile as

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



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).



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.

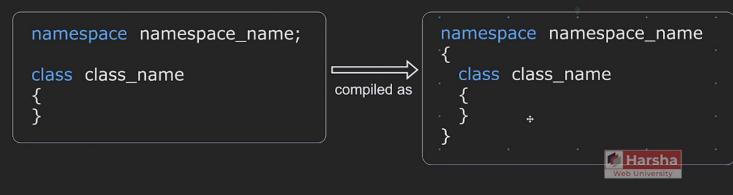
The local variables / local functions declared in the top-level statements are NOT accessible elsewhere (in other types / files).

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-Spaced Namespaces

File Scoped Namespaces



C# 10

File Scoped Namespaces

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

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

Advantage: Allows developers to quickly create one-or-few types in a namespace without nesting them in the 'namespace declaration.'

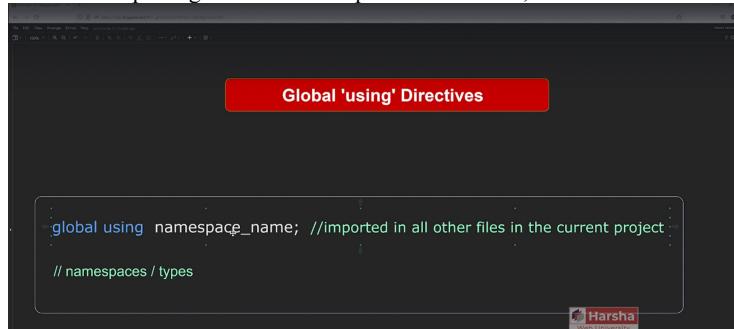
The 'file-scoped namespace' statement CAN be written before / after the 'using' statements.

Only one 'file-scoped namespace' statement is allowed for one source file (C# file).

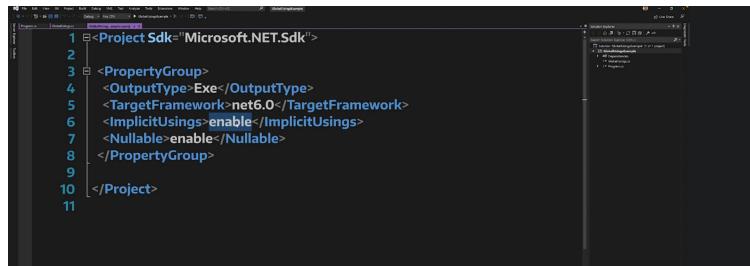
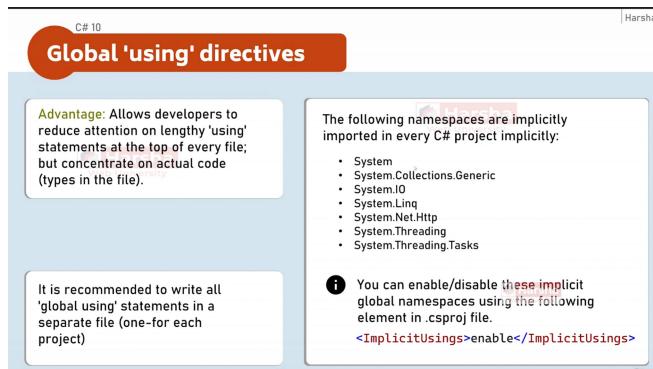
A source file can't contain both 'file-scoped namespace' and 'normal namespace declarations'.

Global Usings

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



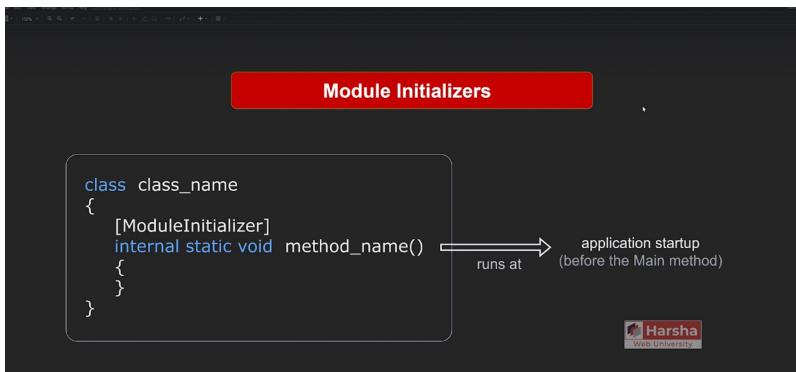
A screenshot of a code editor showing a C# file. A red box highlights the line `global using namespace_name; //imported in all other files in the current project`. Below it, another line `// namespaces / types` is visible. The status bar at the bottom left says "C# 10".



Module Initializers

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.



C# 10

Module Initializers

Allows you run some code with 'global initialization logic' at application startup, when the application loads into memory.

```
File.cs
using System.Runtime.CompilerServices;
class class_name
{
    [ModuleInitializer]
    internal static void method_name()
    {
    }
}
```

Would execute
at application startup
(before the Main method).

C# 10

Module Initializers

Advantage over static constructors:
The static constructors execute ONLY if the class is used at least once; otherwise will NOT execute.

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

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

```
2 using System.Runtime.CompilerServices;
3
4 namespace Initializer
5 {
6     internal class FinalModuleInitializer
7     {
8         [ModuleInitializer]
9         public void FinalInitialize()
10        {
11            Initializer2.Initialize2();
12            Initializer1.Initialize1();
13        }
14    }
15 }
16
```

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

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
```

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?`

Null Forgiving (!) Operator

C# 9

| Harsha

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 (?).

Harsha
Web University

Eg: class?

Null forgiving operator (!)

- Meaning: "I'm sure, it's not null".
- Suffix your expression (variable or property) with "!" operator to make that expression as "not null", at compilation time.
- It has no effect at run time.
- It means, the developer says to the C# compiler - that, a variable or property is "not null". But at run time, if it is actually null, it leads to "NullReference Exception" as normal.
- So use this operator only when you are sure that your expression (variable or property) is NOT null.

C# 9

Target-typed 'new' expressions

C# 9

| Harsha

Target-typed 'new' expressions

Allows the developer "not-to-mention" the class name; but allows to create an object in the 'new' expression.

class_name variable_name = new(); //equivalent to "new class_name()"

Harsha
Web University

Harsha
Web University

Target-typed 'new' expressions

Benefit:

We can create object of a class in shortcut way.

It can't be used in:

- **using block:**

```
using (var variable = new())
{
}
```
- **foreach:**

```
foreach (var variable in new())
{
}
```

Motivation to Pattern Matching

Why to use Pattern Matching?

Pattern Matching

Enables developers to easily check the data type of a variable and also check its value with some conditions.

"is" expression:

```
if (variable_name is class_name another_variable)
{
    if (another_variable.property == value)
    {
        statements...
    }
}
```

"switch-case" expression:

```
switch (variable_name)
{
    case class_name another_variable
        when another_variable.property == value:
            statements...
            break;
}
```

"switch" expression:

```
variable_name switch {
    class_name another_variable when another_variable.property == value => result_expression }
```

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

Pattern Matching

Person

Employee

Customer

Supplier

Manager

Pattern Matching - Type Pattern

Pattern Matching Type Pattern

C# 9 | Harsha

Pattern Matching

Type pattern with Variable pattern:

Regular Code:

```
//Check whether the variable is of specified 'class_name' type.  
if (variable.GetType() == typeof(class_name) ||  
variable.GetType().IsSubClassOf(typeof(class_name)))  
{  
    //typecast the value into the specified class  
    class_name another_variable = (class_name)variable_name;  
  
    statements...  
}
```

"is" expression:

```
//Check whether the variable is of specified  
'class_name' type & also typecast the value  
into specified class.  
if (variable is class_name another_variable)  
{  
    statements...  
}
```



Pattern Matching Switch-Case Pattern

C# 9 | Harsha

Pattern Matching

Type pattern with Variable 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;  
  
        statements; break;  
}
```

"switch-case" expression:

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



Pattern Matching

When Pattern

C# 9

Harsha

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

Harsha

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

C# 9 | Harsha

Pattern Matching

Relational Pattern:

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

C# 9 | Harsha

Pattern Matching

Logical Pattern:

```
//Check whether the variable is of specified 'class_name' type
variable switch
{
    class_name another_variable when
        | another_variable.property is expression1 and expression2      //conjunctive pattern (and)
        | // another_variable.property == expression1 && another_variable.property == expression2

        | another_variable.property is expression1 or expression2       //disjunctive pattern (or)
        | //another_variable.property == expression1 || another_variable.property == expression2

        | another_variable.property is not expression                  //negated pattern (not)
        | // another_variable.property != expression

    => result_expression...
}
```

Pattern Matching

Property Pattern

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

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

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

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

Goal:

The values of fields and properties should be readonly (immutable).
No other classes can change them, after they get initialized.

Immutable class

```
class class_name
{
    data_type readonly field_name; //readonly field

    data_type property_name { get => field_name } //readonly property
}
```

C# 9

| Harsha
Web University

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.

| Harsha
Web University

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

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
    }
}
```

C#

Init-Only Properties

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.

C# 9 | Harsha

'init' only properties

Brief: 'init' only properties can be initialized either inline with declaration, in the constructor or in the object initializer.

Init-only property

```
data_type property_name { get; init; } //init instead of 'set'
```

Harsha Web University

'init' only properties

Immutable class with 'init' only properties

```
class class_name
{
    data_type readonly field_name; //readonly field

    data_type property_name {
        get => field_name //get accessor
        init => field_name = value;
            //init accessor instead of 'set' accessor
    }

    public class_name() //constructor
    {
        field_name = value; //initialize the field
    }
}
```

Object of immutable class:

```
class variable_name = new class_name()
{ property_name = value; }
//initialize value of
'init' only property in object initializer
```

C# Readonly Structs

Readonly Structs

Brief:

Enforces you to write only 'readonly fields' and 'readonly properties' to achieve immutability in your struct.

Readonly struct

```
readonly struct struct_name
{
    //readonly fields
    //readonly properties
}
```

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.

C# 7.2

Readonly Structs

Readonly struct

```
readonly struct struct_name
{
    data_type readonly field_name; //readonly field

    data_type property_name {
        get => field_name //get accessor
        init => field_name = value; //init accessor instead of 'set' accessor
    }

    public class_name() //constructor
    {
        field_name = value; //initialize the field
    }
}
```

Harsha
Web University

C#

Parameterless Struct Constructors

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.

Parameterless Struct Constructors

Struct with parameter-less constructor

```
readonly struct struct_name
{
    data_type readonly field_name; //readonly field

    data_type property_name {
        get => field_name; //get accessor
        init => field_name = value; //init accessor instead of 'set' accessor
    }

    public class_name() //constructor
    {
        field_name = value; //you must initialize all the fields
    }
}
```

C# Records

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

Harsha

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.

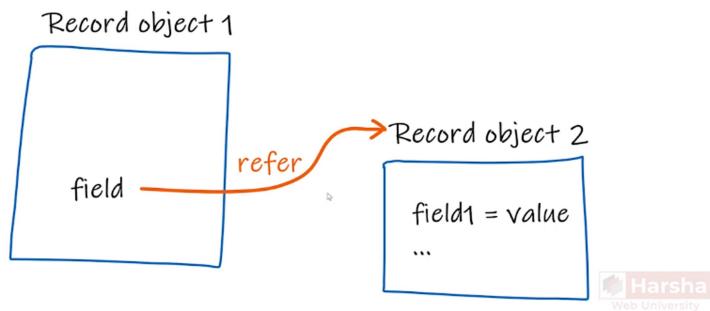
/ 8:34

Harsha
Web University

C# - Records

Nested Records

Nested Records



C# - Records

Immutability

Records

Records are 'immutable' by default.
All the record members become as 'init-only' properties.

Record with Mutable Properties

```
record record_name(data_type Property_name, ...)  
{  
    data_type Property_name { get; set; }  
}
```



C# - Records

Equality

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.

Records

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
```



Records

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
```



C# - Records

'with' expression

n C# 9 | Harsh

Records

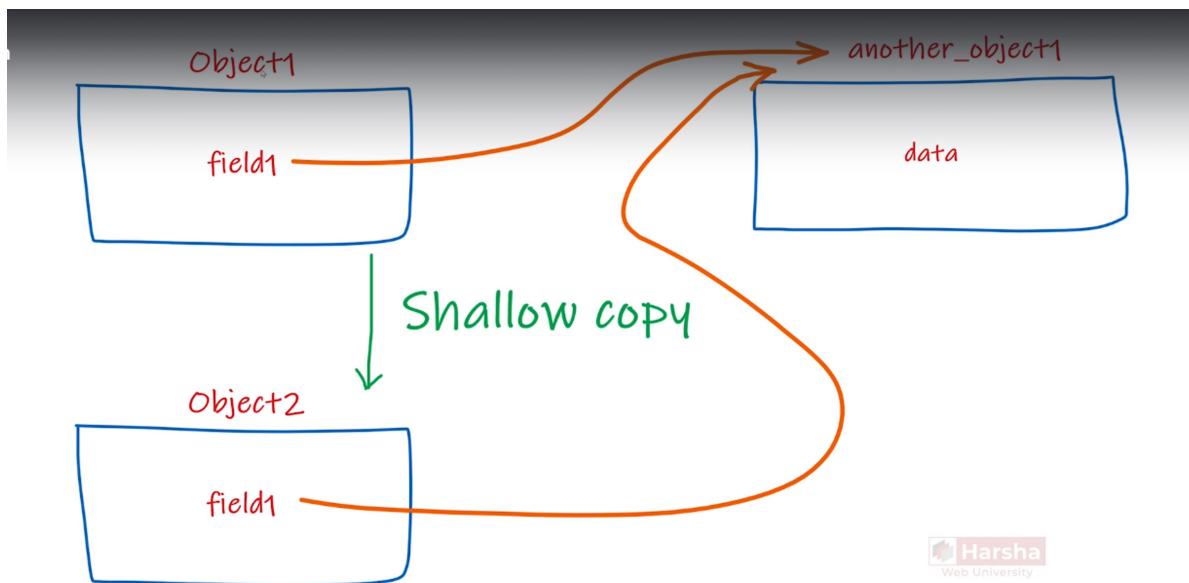
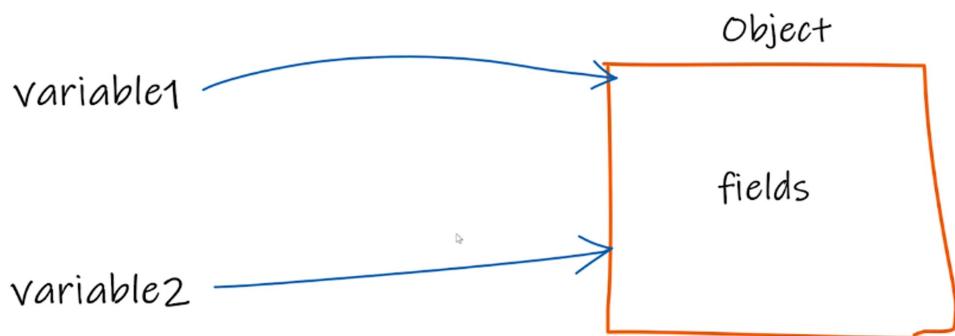
'with' expression acts as object initializer for 'records'.

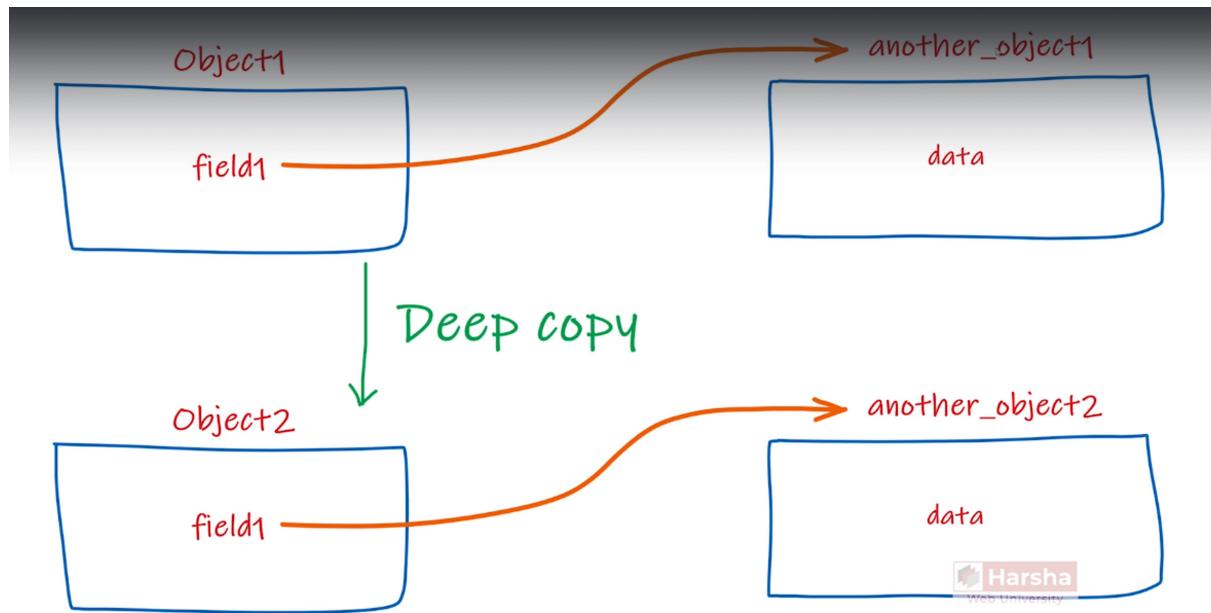
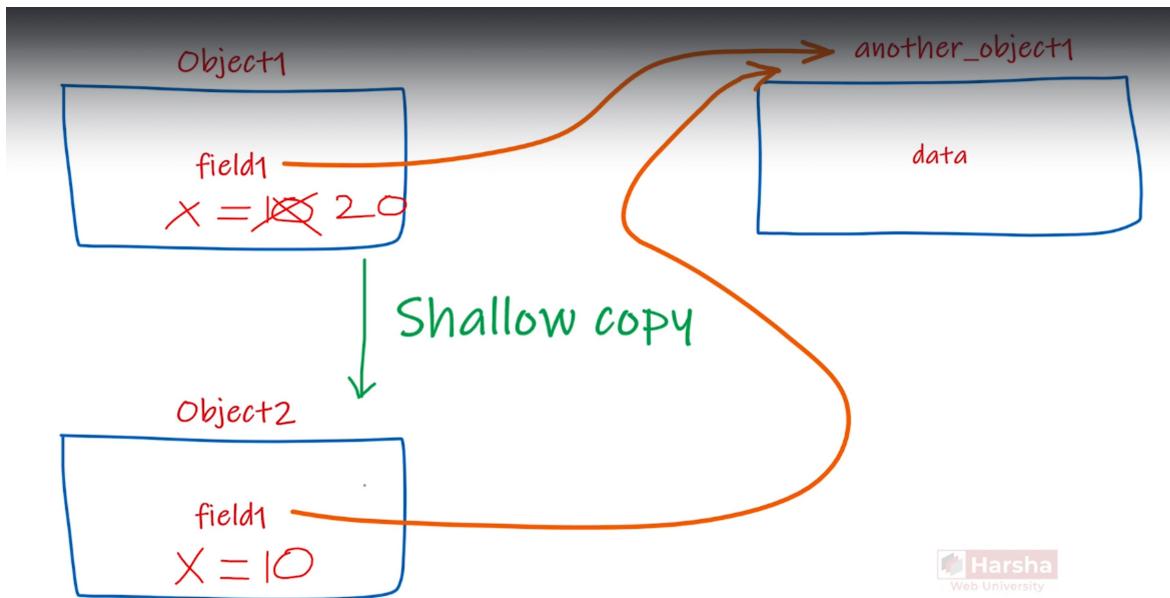
Record - 'with' expression

```
record_name variable1 = new record_name(value1, value2);  
record_name variable2 = variable1 with { Property = value, ... } //with expression
```

Reference Copy

variable2 = variable1





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

Records

'with' expression acts as object initializer for 'records'.

It creates a shallow copy of an existing record object and also overwrites the values of specified properties.

Record - 'with' expression

```
record_name variable1 = new record_name(value1, value2);  
record_name variable2 = variable1 with { Property = value, ... } //with expression
```

C# - Records

Deconstruct()

What is Deconstruction?

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

Example:

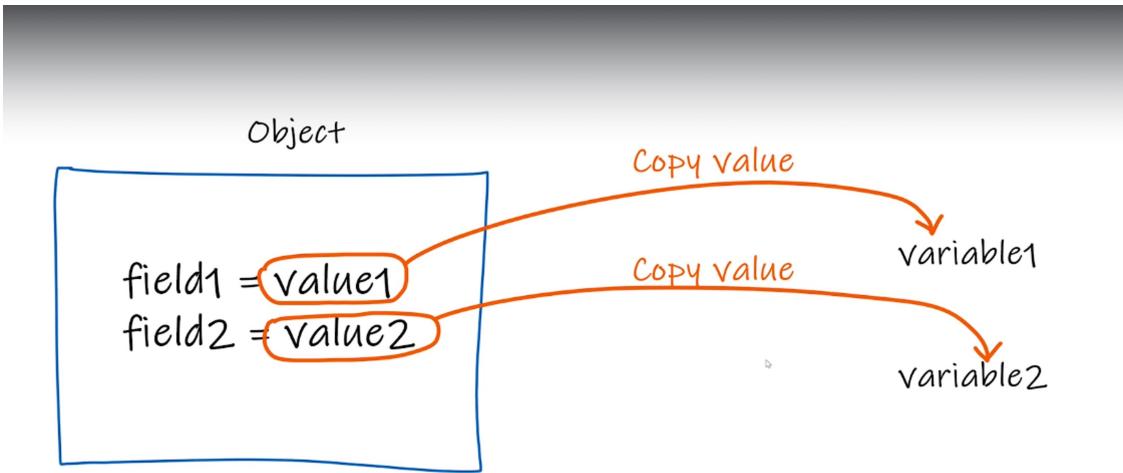
An object has multiple fields, each with a value. Using deconstruction, you can assign:

- 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()

C# 9

Harsha

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, ... }
    };
}
```

Odemiy

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
    }
}
```



Odemy

C# - Records Constructor

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;
}
```

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# 9

| Harsha

Records

A record can inherit from another record.

Record - Inheritance

```
public record Parent_record_name(Properties_list);
```

```
public record Child_record_name(Properties_list) : Parent_record_name;
```

C# 9

| Harsha

Records

A record CAN inherit from another record.

A record CAN'T inherit from another class.

A class CAN'T inherit from another record.

A record CAN implement (inherit) one or more interfaces.

A record CAN be 'abstract' and 'sealed'.



C# - Records

sealed ToString()

ring() C# 10 Harsha

Records

The user-defined 'override ToString()' can be 'sealed', in order to prevent further overriding.

Record - Sealed ToString()

```
public record record_name(Properties_list)
{
    public override sealed string ToString() //user-defined
    {
        //return any string
    };
}
```

7:38

C# - Records

Record Structs

Record Structs

Record [or] Record class

```
record record_name(Properties_list);
```

- A **record** is a class internally (after compilation).
- All positional parameters of a **record** are init-only properties by default.

Readonly record struct

```
readonly record struct record_name(Properties_list);
```

- A **readonly record struct** is a 'struct' internally (after compilation).
- All positional parameters of a **readonly record struct** are init-only properties by default.

Record Structs

Record struct

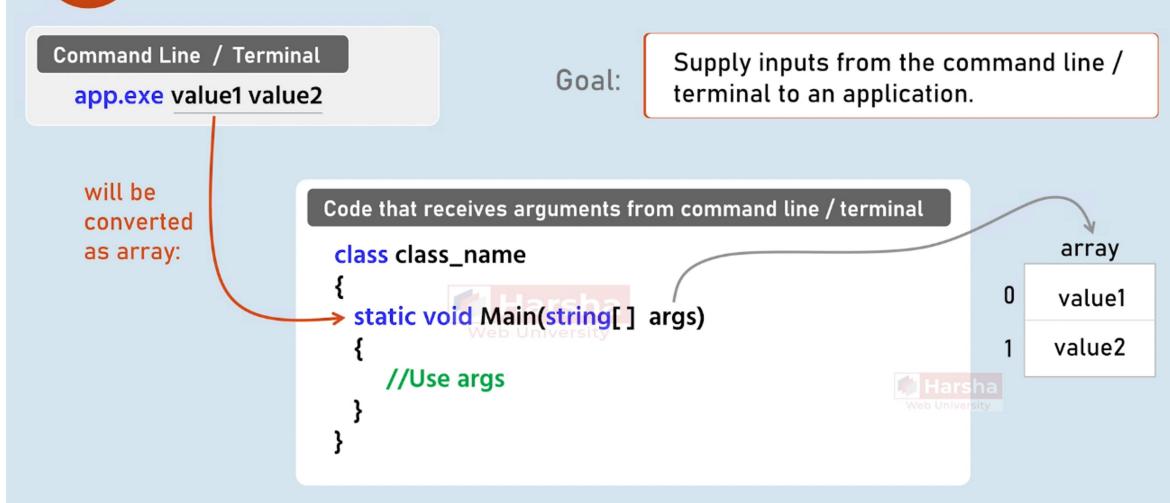
```
record struct record_name(Properties_list);
```

- A **record struct** is a 'struct' internally (after compilation).
- All positional parameters of a **record struct** are read-write properties by default.

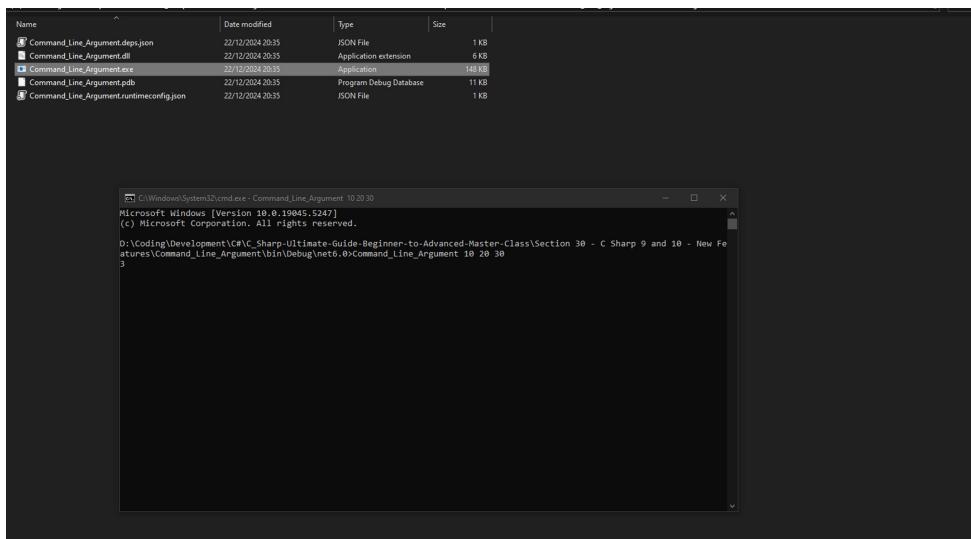
Part 1

Command Line Arguments

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

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);
```



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);
```



C#

Static Anonymous Functions

C# 9

Harsha

Static Anonymous Functions

A static anonymous function is an anonymous method or lambda expression, prefixed with 'static' keyword.

It CAN'T access the state (local variables, parameters, 'this' keyword and 'base' keyword) of the enclosing method; and also CAN'T access instance members of enclosing type.

Static anonymous function (anonymous method)

```
static delegate (Parameters_list)
{
    //can't access locals, parameters, instance members
    //can access static members and constants
}
```

Static anonymous function (lambda expression)

```
static (Parameters_list) =>
{
    //can't access locals, parameters, instance members
    //can access static members and constants
}
```

C#

Lambda Expression – Return Type

Return Type | C# 10 | Harsha

Return Type of Lambda Functions

A lambda expression (or lambda function) can have a return type before the list of parenthesized parameters.

Lambda Function Return Type in C# 10

`return_type (Parameters_list) => return_value;`

C#

Constant Interpolated Strings

C# 10

| Harsh

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# 8

Harsha

Interface Default Methods

Default methods are methods in interfaces with concrete implementation.

These methods are accessible through a reference variable of the interface type.

Interface Default Methods

```
interface interface_name
{
    access_modifier return_type method_name(parameters)
    {
        //method body
    }
}
```

Harsha
Web University

Harsha
Web University

Odemys

C#

Interface Method Modifiers

fiers | C# 8 | Harsha

Access Modifiers on Interface Methods

Interface methods can have any access modifiers including private, protected, internal, protected internal, private protected and public

Non-public interface methods can be either implemented as public or explicitly (to preserve the same access modifier).

Interface methods with access modifier

```
interface interface_name
{
    access_modifier return_type method_name(parameters);
}
```

C#

Interface Private Methods

C# 8

Harsha

Private Interface Methods

Private interface methods must have method body.

Private interface method

```
interface interface_name
{
    private return_type method_name(parameters)
    {
        //method body
    }
}
```

C#

Interface Static Methods

C# 8

| Harsha

Interface Static Methods

Static methods are allowed with concrete implement in interface.

Interface static methods can be called through the interface name.

Interface Static Methods

```
interface interface_name
{
    access_modifier static return_type method_name(parameters)
    {
        //method body
    }
}
interface_name.method_name(arguments); //calling the interface static method
```

Harsha

Harsha
Web University

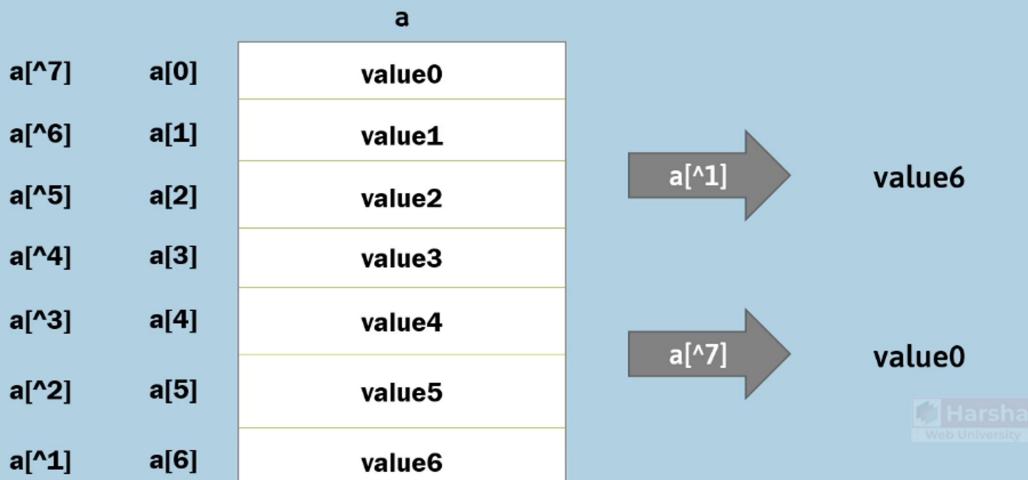
Odemy

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".

