



Harsha Vardhan

- › UI Expert
- › .NET Expert
- › Module Lead
- › Corporate Instructor

 Harsha
Web University

Structures

Agenda

- 1 **Value Types (vs) Reference Types**
Difference between value-types and reference-types.
- 2 **Understanding Structures**
What is Structure and its features.
- 3 **Classes (vs) Structures**
Differences between classes and structures.
- 4 **Constructors in Structures**
Working with parameter-less & parameterized constructors in structures.

Agenda

- 5 **Readonly structs**
New feature in C# 8.0
- 6 **Primitive Types as Structures**
Understanding how all primitive types are structures.

Value-Types (vs) Reference-Types

Next: Structures

Value Types (Structures, Enumerations)	Reference Types (string, Classes, Interfaces, Delegates)
--	--

In C# you can classify the types as two categories i.e. value types and reference types.

Value-Types (vs) Reference-Types

Next: Structures

Value Types (Structures, Enumerations)	Reference Types (string, Classes, Interfaces, Delegates)
--	--

- Mainly meant for storing simple values.
- Instances (examples) are called as "structure instances" or "enumeration instances".
- Instances are stored in "Stack". Every time when a method is called, a new stack will be created.

- Mainly meant for storing complex / large amount of values.
- Instances (examples) are called as "Objects" (Class Instances / Interface Instances / Delegate Instances).
- Instances (objects) are stored in "heap". Heap is only one for entire application.

Introducing Structures

Next: Class (vs) Structure

What

- Structure is a "type", similar to "class", which can contain fields, methods, parameterized constructors, properties and events.

Structure - Example <pre>struct Student { public int studentId; public string studentName; public string GetStudentName() { return studentName; } }</pre>	Structure - Syntax <pre>struct StructureName { fields methods parameterized constructors properties events }</pre>
---	---

Structures are meant for limited amount of fields, may be 1 or 2. Where as classes are meant for storing complex amount of data.

When to use Class and Structure?

=> If you really want to store one or two fields, then you can go for structure. because structures are little bit faster than classes. This is because; structure instances get stored in the stack. The data that is present in the stack can be accessible faster than heap.

On the other hand, if you want to really store more than two fields, then it is recommended to use classes.

Additionally, structures doesn't support inheritance but classes support inheritance.

The screenshot shows the Microsoft Visual Studio IDE. On the left, the code editor displays `Program.cs` with the following content:

```
using System;

class Program
{
    static void Main()
    {
        //create structure instance
        Category category = new Category();

        //initialize fields through properties
        category.CategoryID = 20;
        category.CategoryName = "General";

        //access methods
        Console.WriteLine(category.CategoryID);
        Console.WriteLine(category.CategoryName);
        Console.WriteLine(category.GetCategoryNameLength());

        Console.ReadKey();
    }
}
```

A red box highlights the variable declaration `Category category = new Category();`. Above the code editor, the text "Stack of Main()" is displayed. To the right of the code editor is the Solution Explorer window, which lists the project structure and references.

Structure does not 'user-defined parameter less constructor' and destructor. However, we can write parametrized constructor and constructor overloading.

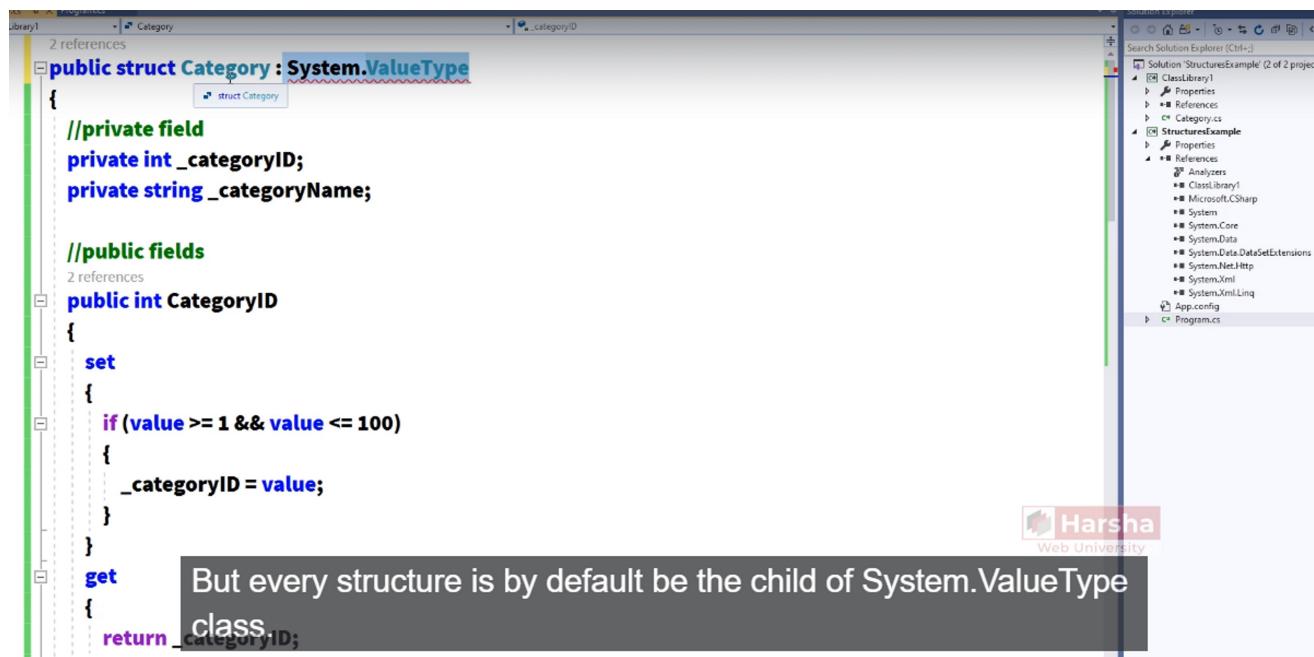
Understanding Structures

Next: Class (vs) Structure

- The instance of structure is called as "structure instance" or "structure variable"; but not called as 'object'.
- We can't create object for structure.
- Objects can be created only based on 'class'.
- Structure instances are stored in 'stack'.
- Structure doesn't support 'user-defined parameter-less constructor and also destructor'.
- Structure can't inherit from other classes or structures.
- Structure can implement one or more interfaces.
- Structure doesn't support virtual and abstract methods.
- Structures are mainly meant for storing small amount of data (one or very few values).
- **Structures are faster than classes, as its instances are stored in 'stack'.**

Structure does not support inheritance because it was meant for storing small amount of value.

Structures	Classes
<ul style="list-style-type: none"> • Structures "value-types". • Structure instances (includes fields) are stored in stack. Structures doesn't require Heap. • Suitable to store small data (only one or two values). • Memory allocation and de-allocation is faster, in case of one or two values. • Structures doesn't support Parameter-less Constructor. • Structures doesn't support inheritance (can't be parent or child). • The "new" keyword just initializes all fields of the "structure instance". 	<ul style="list-style-type: none"> • Classes are "reference-types". • Class instances (objects) are stored in Heap; Class reference variables are stored in stack. • Suitable to store large data (any no. of values) • Memory allocation and de-allocation is a bit slower. • Classes support Parameter-less Constructor. • Classes support Inheritance • The "new" keyword creates a new object.



The screenshot shows the Visual Studio IDE. On the left, the code editor displays the `Category.cs` file:

```

public struct Category : System.ValueType
{
    //private field
    private int _categoryID;
    private string _ categoryName;

    //public fields
    public int CategoryID
    {
        set
        {
            if (value >= 1 && value <= 100)
            {
                _categoryID = value;
            }
        }
        get
        {
            return _categoryID;
        }
    }
}

```

A tooltip box is overlaid on the code editor, containing the text: "But every structure is by default be the child of System.ValueType class".

On the right, the Solution Explorer shows a solution named "StructureExample" with two projects: "ClassLibrary1" and "StructuresExample".

ClassLibrary1 - Category

CategoryID

2 references

```
public struct Category
{
    //private field
    private int _categoryID = 101;
    private string _name;

    //public fields
}
```

(field) int Category._categoryID

'Category': cannot have instance property or field initializers in structs

This value cannot be initialized; because in the structures you cannot initialize fields or properties.

Harsha
Web University

Class (vs) Structure (contd...)

Next: Comparison Table

Structure	Classes
<ul style="list-style-type: none"> Structures doesn't support abstract methods and virtual methods. Structures doesn't support destructors. Structures are internally derived from "System.ValueType". <p>System.Object → System.ValueType → Structures</p> <ul style="list-style-type: none"> Structures doesn't support to initialize "non-static fields", in declaration. Structures doesn't support "protected" and "protected internal" access modifiers. Structure instances doesn't support to assign "null". 	<ul style="list-style-type: none"> Classes support abstract methods and virtual methods. Classes support destructors. Classes are internally and directly derived from "System.Object". <p>System.Object → Classes</p> <ul style="list-style-type: none"> Classes supports to initialize "non-static fields", in declaration. Classes support "protected" and "protected internal" access modifiers. Class's reference variables support to assign "null".

Comparison Table: Structure (vs) Class

Based on inheritance and object

Class Type	Can Inherit from Other Classes	Can Inherit from Other Interfaces	Can be Inherited	Can be Instantiated
Normal Class	Yes	Yes	Yes	Yes
Abstract Class	Yes	Yes	Yes	No
Interface	No	Yes	Yes	No
Sealed Class	Yes	Yes	No	Yes
Static Class	No	No	No	No
Structure	No	Yes	No	Yes

Comparison Table: Structure (vs) Class

Type	1. Non-Static Fields	2. Non-Static Methods	3. Non-Static Constructors	4. Non-Static Properties	5. Non-Static Events	6. Non-Static Destructors	7. Constants
Normal Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Abstract Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface	No	No	No	No	Yes	No	No
Sealed Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Static Class	No	No	No	No	No	No	Yes
Structure	Yes	Yes	Yes	Yes	Yes	No	Yes

Comparison Table: Structure (vs) Class

Type	8. Static Fields	9. Static Methods	10. Static Constructors	11. Static Properties	12. Static Events	13. Virtual Methods	14. Abstract Methods	15. Non-Static Auto-Impl Properties	16. Non-Static Indexers
Normal Class	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Abstract Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface	No	No	No	No	No	No	Yes	Yes	No
Sealed Class	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Static Class	Yes	Yes	Yes	Yes	Yes	No	No	No	No
Structure	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes

1

Value Types (vs) Reference Types

Difference between value-types and reference-types.

2

Understanding Structures

What is Structure and its features.

3

Classes (vs) Structures

Differences between classes and structures.

4

Constructors in Structures

Working with parameter-less & parameterized constructors in structures.



Constructors in Structures

Next: Readonly Structures

| Harsha

Rules

- › C# provides a parameter-less constructor for every structure by default, which initializes all fields.
- › You can also create one or more user-defined parameterized constructors in structure.
- › Each parameterized constructor must initialize all fields; otherwise it will be compile-time error.
- › The "new" keyword used with structure, doesn't create any object / allocate any memory in heap; It is a just a syntax to call constructor of structure.



```
public StructureName( datatype parameter)
{
    field = parameter;
}
```

Rules

- › C# provides a parameter-less constructor for every structure by default, which initializes all fields.
- › You can also create one or more user-defined parameterized constructors in structure.
- › Each parameterized constructor must initialize all fields; otherwise it will be compile-time error.
- › The "new" keyword used with structure, doesn't create any object / allocate any memory in heap; It is a just a syntax to call constructor of structure.



```
public StructureName(datatype parameter)
{
    field = parameter;
}
```

(vs) Class Practically

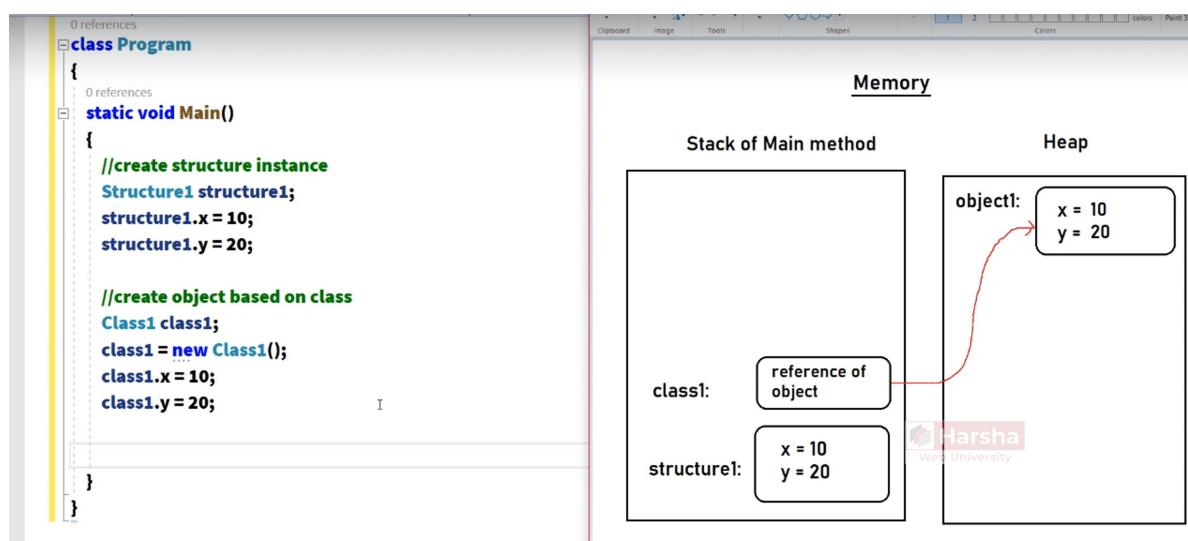
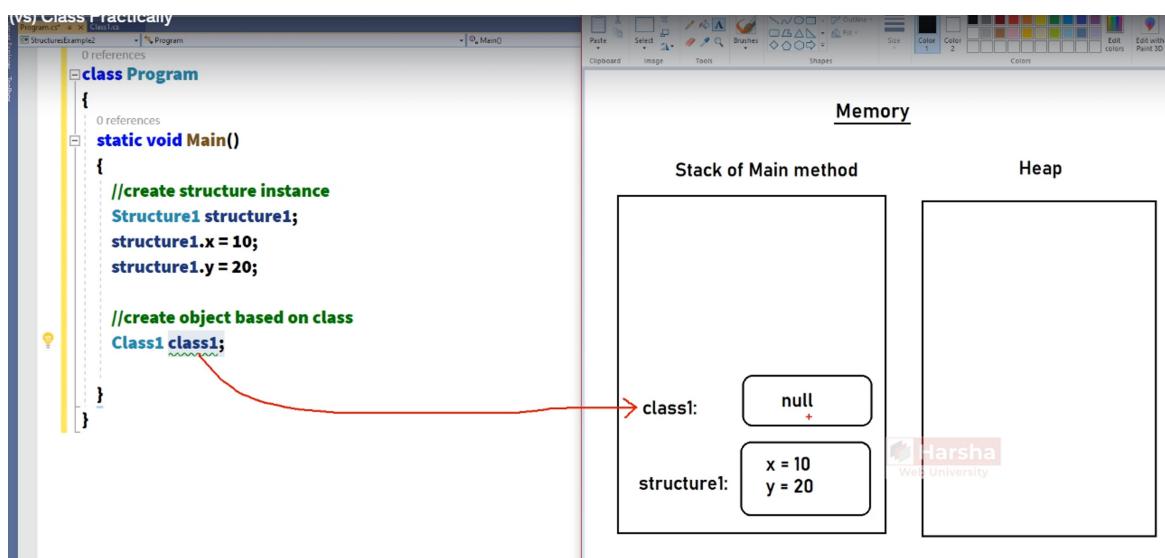
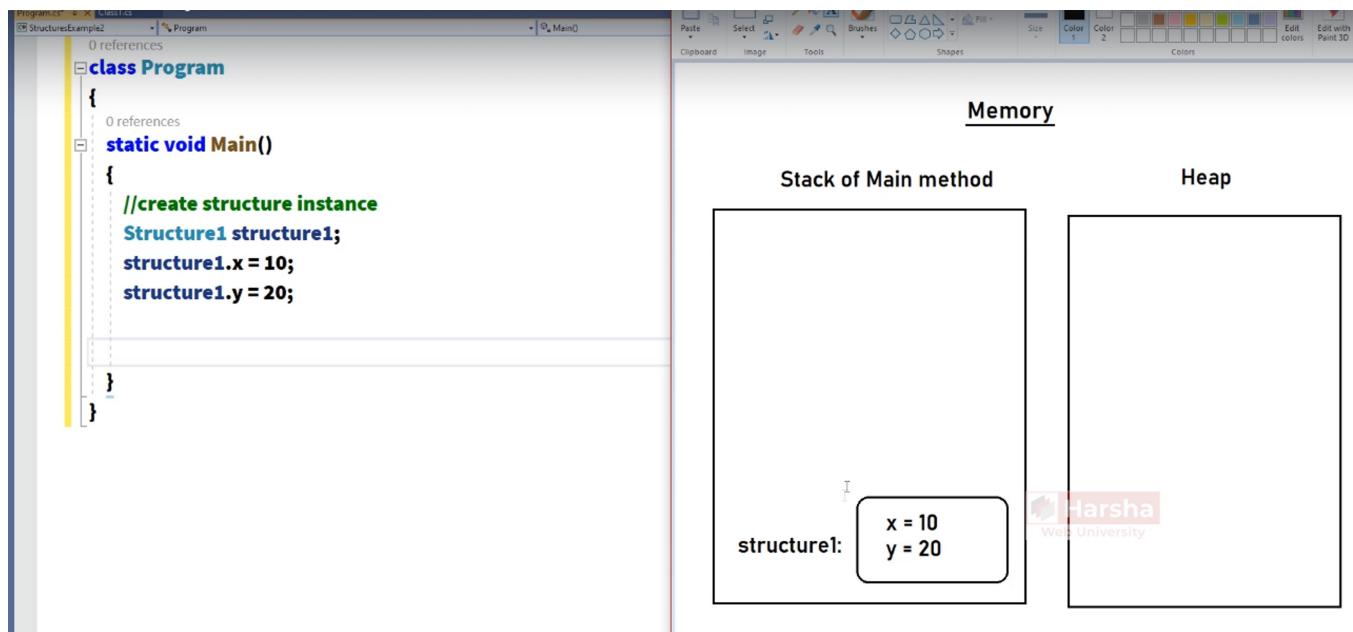


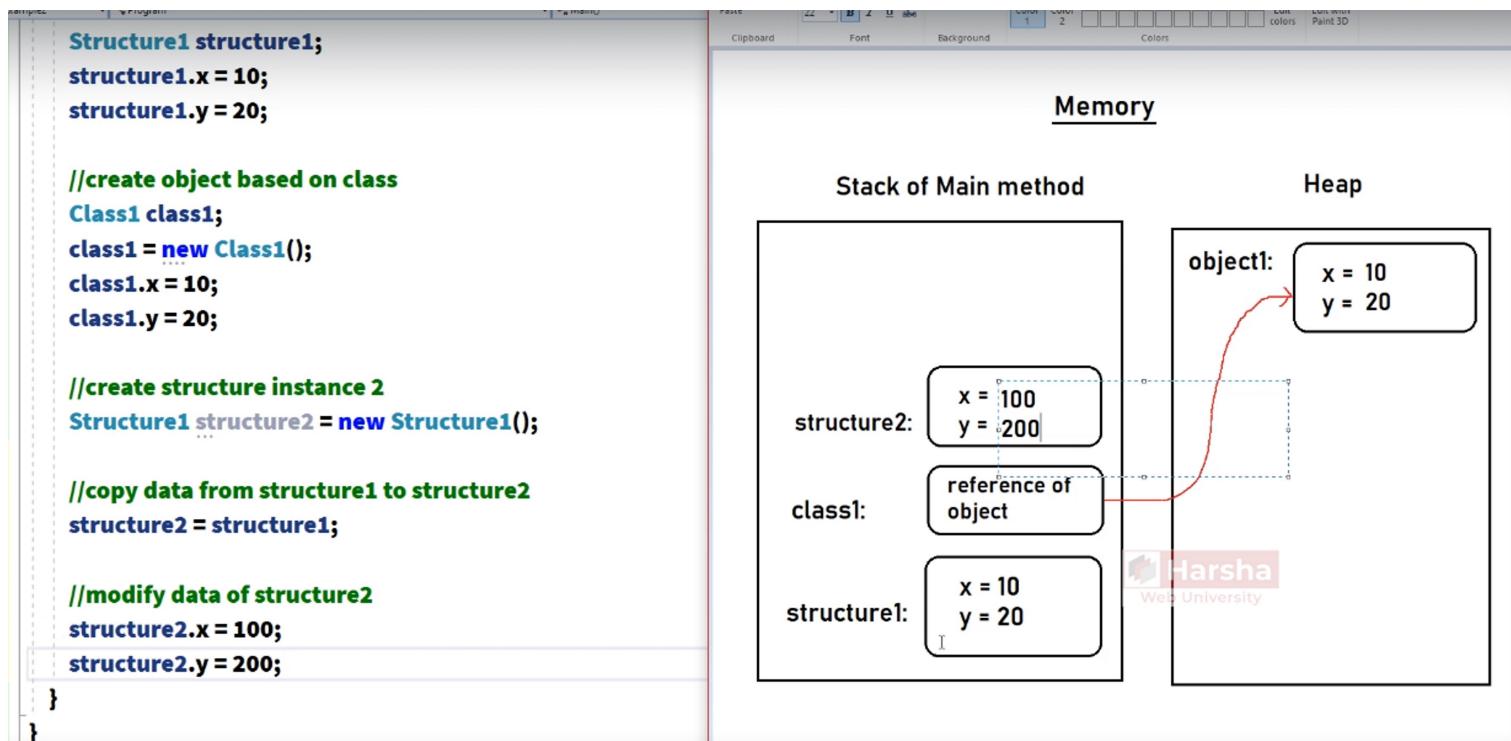
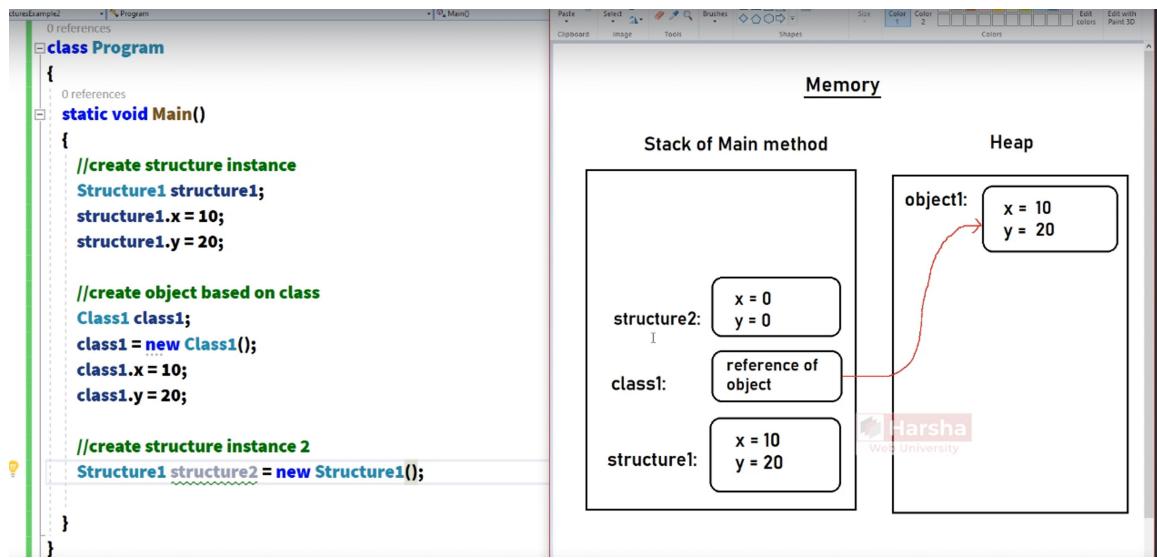
Harsha Vardhan

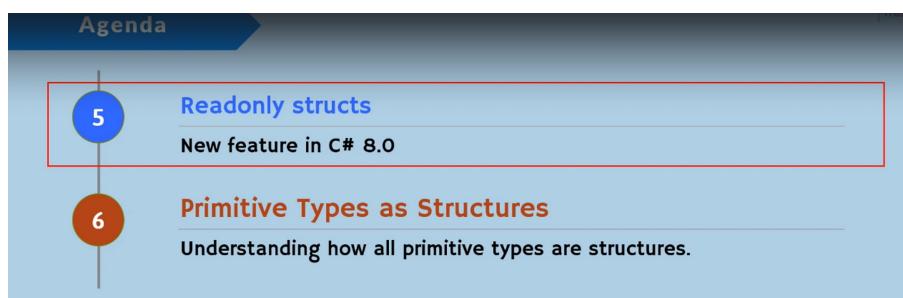
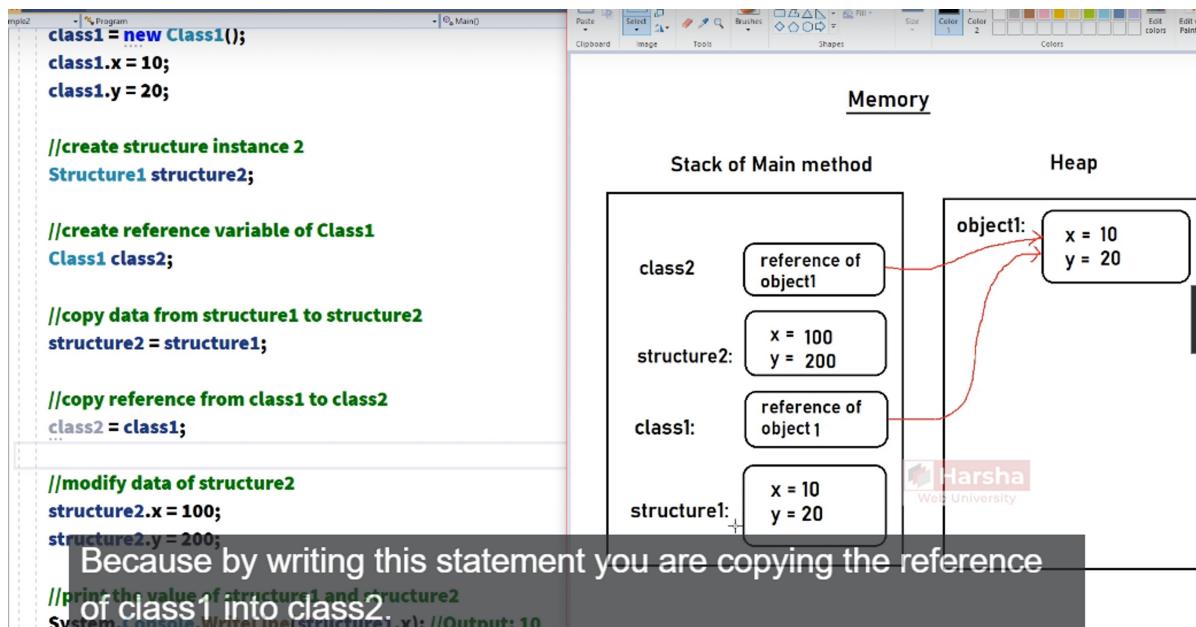
- › UI Expert
- › .NET Expert
- › Module Lead
- › Corporate Instructor



Structures







Readonly Structures Next: Primitive Types as Structures

When

- Use readonly structures in case of all of these below:
 - All fields are readonly.
 - All properties have only 'get' accessors (readonly properties).
 - There is a parameterized constructor that initializes all the fields.
 - You don't want to allow to change any field or property of the structure.
 - Methods can read fields; but can't modify.

💡 'Readonly structures' is a new feature in C# 8.0.
This feature improves the performance of structures.

Readonly Structure - Example

```

readonly struct Student
{
    public readonly int studentId;
    public string studentName { get; }
    public Student()
    {
        studentId = 1;
        studentName = "Scott";
    }
}

```

In real time projects, whenever you want to restrict yourself to create all members as readonly members. In that case better to make it as readonly structures.

Primitive Types as Structures

Understanding how all primitive types are structures.

All the primitive types except string are structures. That includes sbyte, byte, short, ushort, int, uint, long, ulong, float, double, decimal, char and bool. All these are internally structures.

Primitive Types as Structures



- › All primitive types are structures.
- › For example, "sbyte" is a primitive type, which is equivalent to "System.SByte" (can also be written as 'SByte') structure.
- › In C#, it is recommended to always use primitive types, instead of structure names.

Primitive Types as Structures

Data Type	Is it Structure / Class?	Name of Structure / Class	Full Path (with namespace)
sbyte	Structure	SByte	System.SByte
byte	Structure	Byte	System.Byte
short	Structure	Int16	System.Int16
ushort	Structure	UInt16	System.UInt16
int	Structure	Int32	System.Int32
uint	Structure	UInt32	System.UInt32
long	Structure	Int64	System.Int64
ulong	Structure	UInt64	System.UInt64
float	Structure	Single	System.Single
double	Structure	Double	System.Double
decimal	Structure	Decimal	System.Decimal
char	Structure	Char	System.Char
bool	Structure	Boolean	System.Boolean
string	Class	String	System.String

```
nativeTypesAndStructures - 1% Program          0 references  0% Main()
0 references
class Program
{
    static void Main()
    {
        //create a structure variable
        sbyte a = 10;

        //create a reference variable and object of string
        string b = "Hello";
    }
}
```

```
static void Main()
{
    //create a structure variable
    sbyte a = 10;

    //create a reference variable and object of string
    System.String b = "Hello";
}
```

class System.String
Represents text as a sequence of UTF-16 code units.