C# - Ultimate Guide - Beginner to Advanced | Master class

Section 14 - Structures

Value-Types vs Reference-Types

Value Types (Structures, Enumerations)

• 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.

Reference Types (string, Classes, Interfaces, Delegates)

• Mainly meant for storing complex / large amount of values.•Instances (examples) are called as "Objects" (Class Instances / Interface Instances / Delegate Instances).

• 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.

Structures

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

Structure - Example

```
struct Student
{
  public int studentId;
  public string studentName;

  public string GetStudentName( )
  {
    return studentName;
  }
}
```

Structure - Syntax

```
struct StructureName
{
  fields
  methods
  parameterized constructors
  properties
  events
}
```

- 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'.

Class (vs) Structure

Structures

1. Structures "value-types".
2. Structure instances (includes fields) are stored in stack. Structures doesn't require Heap.Structure instances (includes fields) are stored in stack. Structures doesn't require Heap.
3. Suitable to store small data (only one or two values).
4. Memory allocation and de-allocation is faster, in case of one or two values.
5. Structures doesn't support Parameter-less Constructor.
6. Structures doesn't support inheritance (can't be parent or child).
7. The "new" keyword just initializes all fields of the "structure instance".
8. Structures doesn't support abstract methods and virtual methods.
9. Structures doesn't support destructors.
10. Structures are internally derived from "System.ValueType". System.Object -> System.ValueType -> Structures
11. Structures doesn't support to initialize "non-static fields“, in declaration.
12. Structures doesn't support "protected" and "protected internal" access modifiers.
13. Structure instances doesn't support to assign "null".

Classes

1. Classes are "reference-types".
2. Class instances (objects) are stored in Heap; Class reference variables are stored in stack.
3. Suitable to store large data (any no. of values)
4. Memory allocation and de-allocation is a bit slower.
5. Classes support Parameter-less Constructor.
6. Classes support Inheritance.
7. The "new" keyword creates a new object.
8. Classes support abstract methods and virtual methods.
9. Classes support destructors.Classes are internally and directly derived from "System.Object". System.Object -> Classes

10. Classes supports to initialize "non-static fields“, in declaration.
11. Classes support "protected" and "protected internal" access modifiers.
12. Class's reference variables support to assign "null".

Comparison Table - Class (vs) Structure

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

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

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

Constructors in Structures

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;
}
```

Read-only Structures

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 Structure - Example

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

'Readonly structures' is a new feature in C# 8.0.

This feature improves the performance of 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.

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