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**

1. struct Student
2. {
3. public int studentId;
4. public string studentName;
6. public string GetStudentName( )
7. {
8. return studentName;
9. }
10. }

**Structure - Syntax**

1. struct StructureName
2. {
3. fields
4. methods
5. parameterized constructors
6. properties
7. events
8. }

* 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

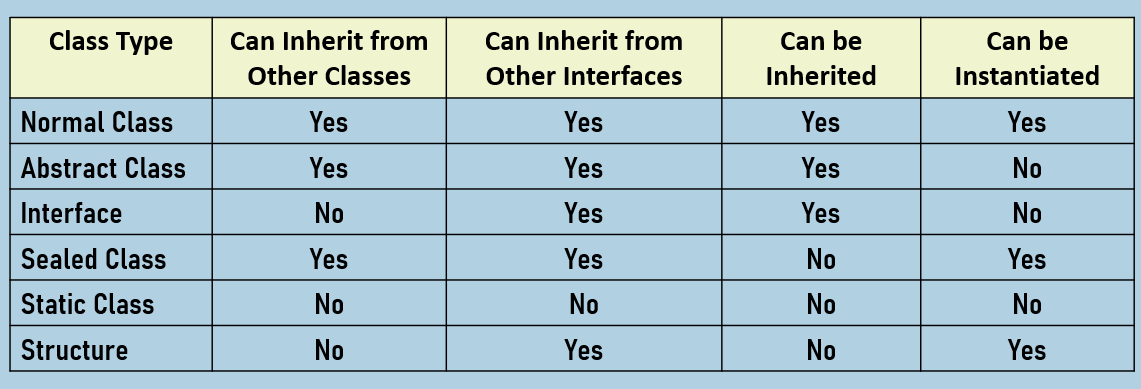
**Structures**

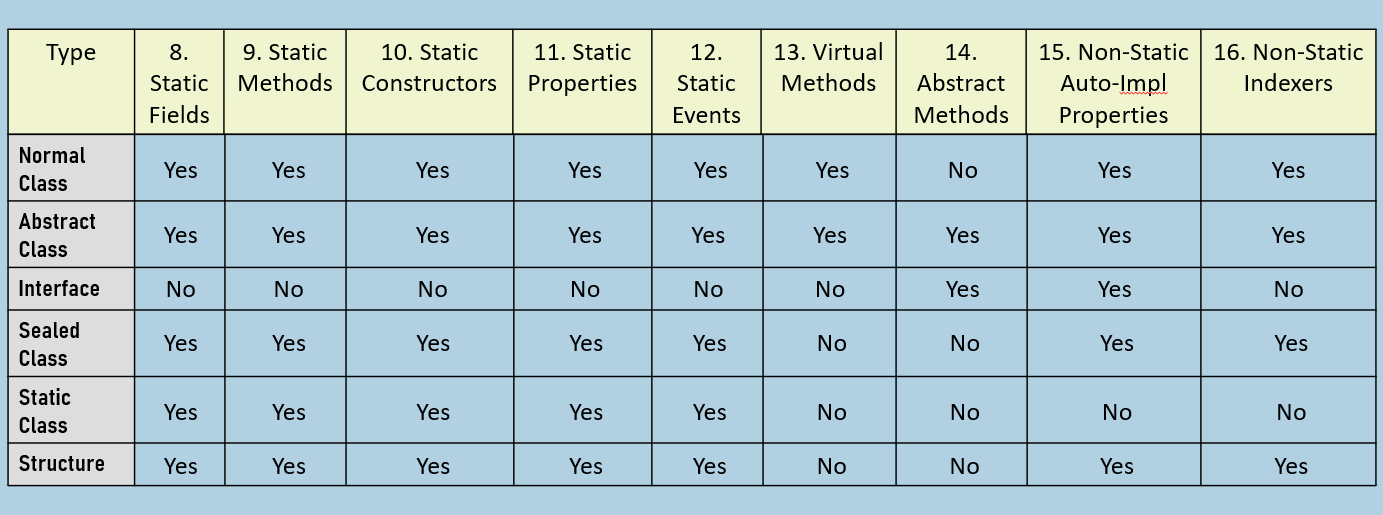
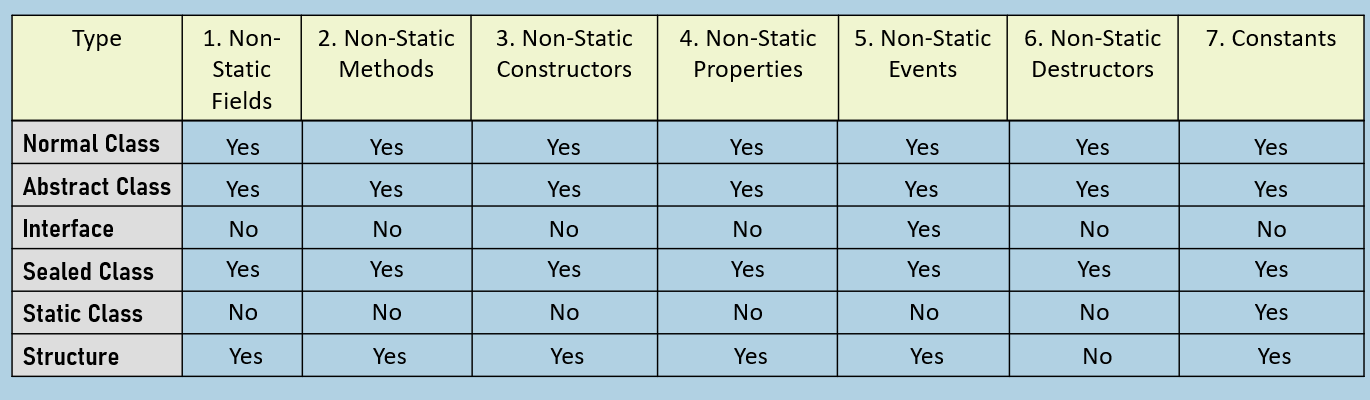
1. Structures "value-types".
2. Structure instances (includes fields) are stored in stack. Structures doesn't require Heap.Structure instances (includes fields) are stored in stack. Structures doesn't require Heap.
3. Suitable to store small data (only one or two values).
4. Memory allocation and de-allocation 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





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.

1. public StructureName( datatype parameter)
2. {
3. field = parameter;
4. }

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**

1. readonly struct Student
2. {
3. public readonly int studentId;
4. public string studentName { get; }
5. public Student( )
6. {
7. studentId = 1;
8. studentName = "Scott";
9. }
10. }

'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.

