What's the difference between struct and class in .NET?

Asked 12 years, 11 months ago Active 8 months ago Viewed 413k times

779

287

What's the difference between struct and class in .NET?

.net class struct value-type reference-type

Share Follow

edited Dec 16 '17 at 22:41
Mark Amery
113k 59 359 411

asked Aug 16 '08 at 8:21
Keith
135k 68 275 393

19 Answers

Active Oldest Votes

1144

✓

In .NET, there are two categories of types, *reference types* and *value types*.

Structs are *value types* and classes are *reference types*.

The general difference is that a *reference type* lives on the heap, and a *value type* lives inline, that is, wherever it is your variable or field is defined.

A variable containing a *value type* contains the entire *value type* value. For a struct, that means that the variable contains the entire struct, with all its fields.

A variable containing a *reference type* contains a pointer, or a *reference* to somewhere else in memory where the actual value resides.

Internally, *reference types* are implemented as pointers, and knowing that, and knowing how variable assignment works, there are other behavioral patterns:

- copying the contents of a *value type* variable into another variable, copies the entire contents into the new variable, making the two distinct. In other words, after the copy, changes to one won't affect the other
- copying the contents of a *reference type* variable into another variable, copies the reference, which means you now have two references to the same *somewhere else* storage of the actual data. In other words, after the copy, changing the data in one reference will appear to affect the other as well, but only because you're really just looking at the same data both places

When you declare variables or fields, here's how the two types differ:

- variable: *value type* lives on the stack, *reference type* lives on the stack as a pointer to somewhere in heap memory where the actual memory lives (though note Eric Lipperts article series: The Stack Is An Implementation Detail.)
- class/struct-field: *value type* lives completely inside the type, *reference type* lives inside the type as a pointer to somewhere in heap memory where the actual memory lives.

Share Follow

edited Dec 20 '17 at 11:47

answered Aug 16 '08 at 18:41

Lasse V. Karlsen
353k 94 586 782

51 In the interest of full completeness, I should mention that Eric Lippert has said that the stack is an implementation detail, whenever I mention stack above, have Eric's post(s) in mind. – Lasse V. Karlsen Jun 16 '11 at 15:59

3 Is this all valid for C++ as well? – Koray Tugay Nov 24 '12 at 21:10

11 another crucial difference is usage. From MSDN: "structs are typically used to encapsulate small group of related variables, such as coordinates of rectangle. Structs can also contain constructors, constants, fields, methods, properties, indexers, operators, events, and nested types, although if several such members are required, you should consider making your type a class instead." – thewpfguy Feb 27 '13 at 5:08

1 Here is a code sample that illustrates the fact that class instances are assigned by reference and struct instances are assigned by value: ideone.com/5yqg2u – wip Nov 15 '13 at 7:51

232

A short summary of each:

Classes Only:

- Can support inheritance
- Are reference (pointer) types
- The reference can be null
- Have memory overhead per new instance

Structs Only:

- Cannot support inheritance
- Are value types
- Are passed by value (like integers)
- Cannot have a null reference (unless Nullable is used)
- Do not have a memory overhead per new instance - unless 'boxed'

Both Classes and Structs:

- Are compound data types typically used to contain a few variables that have some logical relationship
- Can contain methods and events
- Can support interfaces

Share Follow

edited Jun 7 '12 at 15:49

answered Oct 4 '08 at 22:07
Thomas Bratt
41.2k 34 113 134

1 struct do not support inheritance from other stucts/classes, but you CAN implement an interface on a struct. – thewpfguy Feb 27 '13 at 5:03

2 You might want to clarify what you mean when you claim that structs "*Do not have a memory overhead per new instance*". My first interpretation was that you were claiming - obviously absurdly - that structs use zero memory. Then I thought that maybe you're trying to say that a struct, unlike a class, requires exactly as much memory as the sum of its member fields, and no more. But then I Googled for `c# struct memory overhead` and found this answer by Hans Passant that says that no, that's not the case either. So what *do* you mean? – Mark Amery May 5 '17 at 23:51

4 @MarkAmery I had the same initial reaction as you id to the expression "no memory overhead", but I think that the OP is referring to the fact that instances of `class` are managed memory (handled by the garbage collector), whereas instances of `struct` are not. – Hutch Nov 6 '17 at 22:40

2 "Struct Are passed by value (like integers)" is false: all variable are passed by value, also the reference type. If you want to pass a variable by reference you have to use the "ref" keyword. jonskeet.uk/csharp/parameters.html#ref – Marco Staffoli Nov 8 '17 at 5:58

50

In .NET the struct and class declarations differentiate between reference types and value types.

When you pass round a reference type there is only one actually stored. All the code that accesses the instance is accessing the same one.

When you pass round a value type each one is a copy. All the code is working on its own copy.

This can be shown with an example:

```
struct MyStruct
{
    string MyProperty { get; set; }
}

void ChangeMyStruct(MyStruct input)
{
   input.MyProperty = "new value";
}

...

// Create value type
```

For a class this would be different

```
class MyClass
{
    string MyProperty { get; set; }
}

void ChangeMyClass(MyClass input)
{
   input.MyProperty = "new value";
}

...

// Create reference type
MyClass testClass = new MyClass { MyProperty = "initial value" };

ChangeMyClass(testClass);

// Value of testClass.MyProperty is now "new value"
// - the method changed the instance passed.
```

Classes can be nothing - the reference can point to a null.

Structs are the actual value - they can be empty but never null. For this reason structs always have a default constructor with no parameters - they need a 'starting value'.

Share Follow

edited Apr 6 '15 at 11:39
Peter Mortensen
28.5k 21 95 123

answered Aug 16 '08 at 8:30
Keith
135k 68 275 393

@T.Todua yeah, there are better answers above, that I voted up and picked as the answer after providing this one - this is from the early beta of SO when we were still figuring out the rules. – Keith Jul 15 '19 at 15:08

41

Difference between Structs and Classes:

- **Structs are value type** whereas **Classes are reference type**.
- **Structs are stored on the stack** whereas **Classes are stored on the heap**.
- Value types hold their value in memory where they are declared, but reference type holds a reference to an object memory.
- **Value types destroyed immediately** after the scope is lost whereas reference type only the variable destroy after the scope is lost. The object is later destroyed by the garbage collector.
- When you copy struct into another struct, a new copy of that struct gets created modified of one struct won't affect the value of the other struct.
- When you copy a class into another class, it only copies the reference variable.
- Both the reference variable point to the same object on the heap. Change to one variable will affect the other reference variable.
- **Structs can not have destructors**, but classes can have destructors.
- **Structs can not have explicit parameterless constructors** whereas classes can. Structs don't support inheritance, but classes do. Both support inheritance from an interface.
- **Structs are sealed type**.

Share Follow

edited Oct 30 '20 at 18:27
Community ♦
1 1

answered Jan 5 '18 at 1:58
shana
1,797 1 20 17

I find this an exhaustive and concise answer. Want to know more about each bullet? Go learn. – mireazma Sep 22 '20 at 9:37

1 It's not necessarily true that "structs are stored on the stack whereas classes are stored on the heap." See stackoverflow.com/questions/3542083/... – w128 Mar 12 at 10:42

As a rule of thumb, the majority of types in a framework should be classes. There are, however, some situations in which the characteristics of a value type make it more appropriate to use structs.

✓ **CONSIDER a struct** instead of a class:

- If instances of the type are small and commonly short-lived or are commonly embedded in other objects.

X AVOID a struct unless the type has **all** of the following characteristics:

- It logically represents a single value, similar to primitive types (int, double, etc.).
- It has an instance size under 16 bytes.
- It is immutable. *(cannot be changed)*
- It will not have to be boxed frequently.

Share Follow

answered May 14 '17 at 19:58
SunsetQuest
5,789 2 34 31

22

In addition to all differences described in the other answers:

1. Structs cannot have an explicit parameterless constructor whereas a class can
2. Structs cannot have destructors, whereas a class can
3. Structs can't inherit from another struct or class whereas a class can inherit from another class. (Both structs and classes can implement from an interface.)

If you are after a video explaining all the differences, you can check out *Part 29 - C# Tutorial - Difference between classes and structs in C#*.

4 Much more significant than the fact that .net languages will generally not allow a struct to define a parameterless constructor (the decision of whether or not to allow it is made by the language compiler) is the fact that structures can come into existence and be exposed to the outside world without any sort of constructor having been run (even when a parameterless constructor is defined). The reason .net languages generally forbid parameterless constructors for structs is to avoid the confusion that would result from having such constructors be sometimes run and sometimes not. – supercat Jun 15 '12 at 16:57

17

Instances of classes are stored on the managed heap. All variables 'containing' an instance are simply a reference to the instance on the heap. Passing an object to a method results in a copy of the reference being passed, not the object itself.

Structures (technically, value types) are stored wherever they are used, much like a primitive type. The contents may be copied by the runtime at any time and without invoking a customised copy-constructor. Passing a value type to a method involves copying the entire value, again without invoking any customisable code.

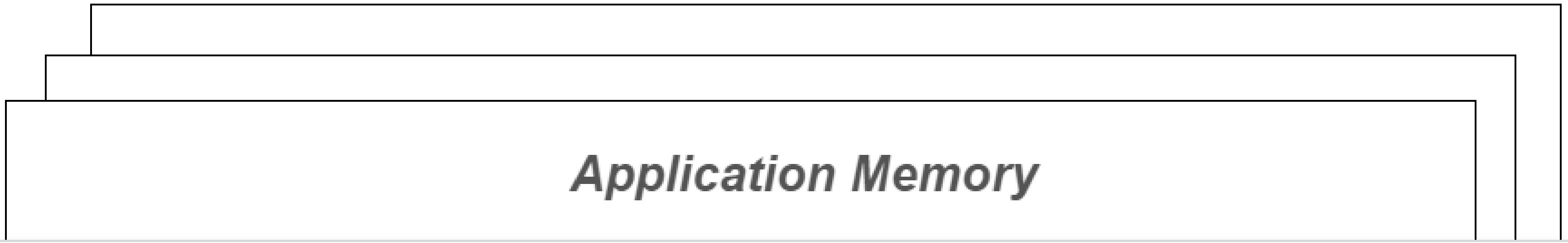
The distinction is made better by the C++/CLI names: "ref class" is a class as described first, "value class" is a class as described second. The keywords "class" and "struct" as used by C# are simply something that must be learned.

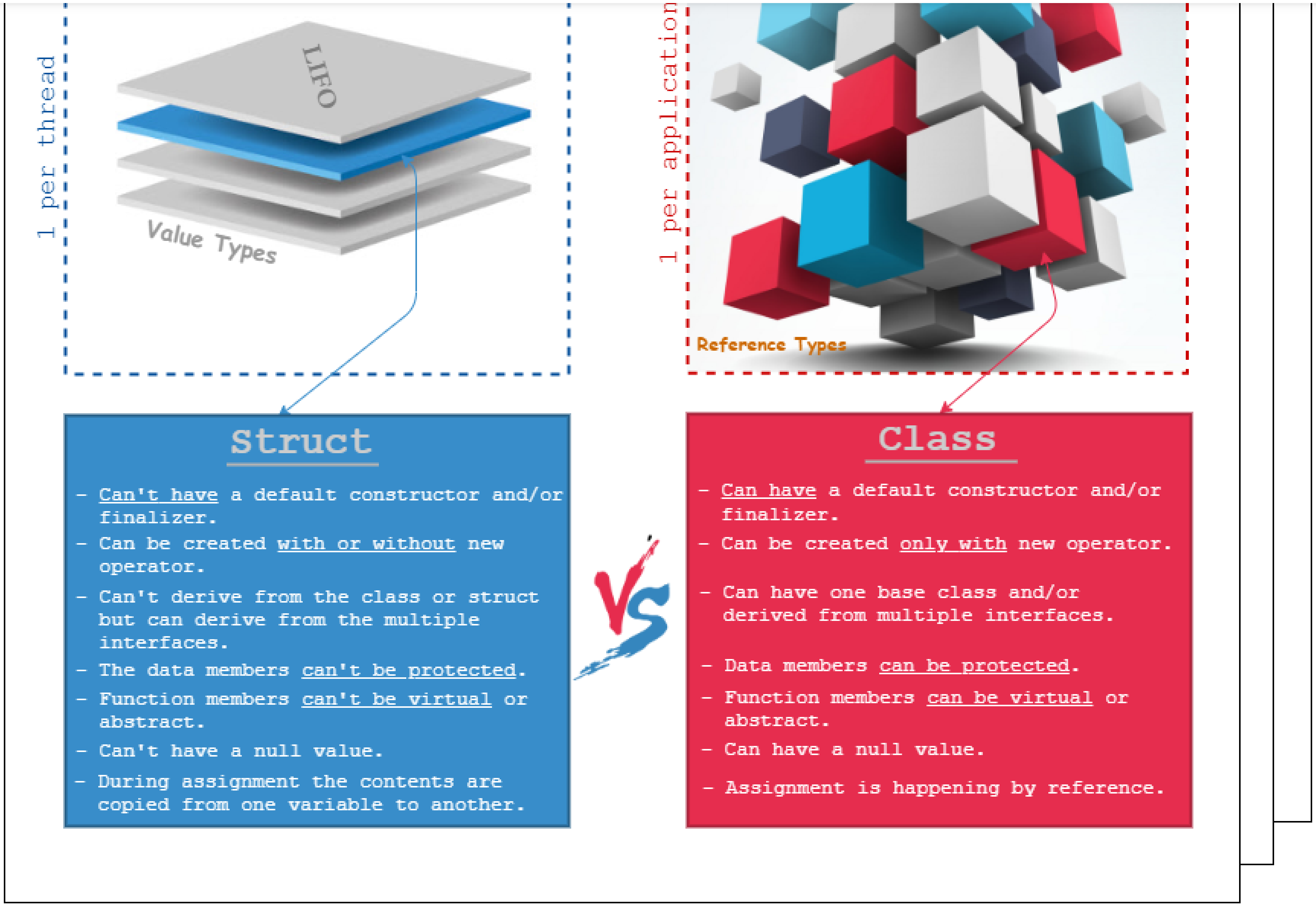
Share Follow

answered Aug 16 '08 at 12:16
Zooba
10.6k 2 33 39

15

I ♥ visualizations, and here I've created a one to show the basic differences between structs and classes.





Struct

- Can't have a default constructor and/or finalizer.
- Can be created with or without new operator.
- Can't derive from the class or struct but can derive from the multiple interfaces.
- The data members can't be protected.
- Function members can't be virtual or abstract.
- Can't have a null value.
- During assignment the contents are copied from one variable to another.

Class

- Can have a default constructor and/or finalizer.
- Can be created only with new operator.
- Can have one base class and/or derived from multiple interfaces.
- Data members can be protected.
- Function members can be virtual or abstract.
- Can have a null value.
- Assignment is happening by reference.

- Choosing Between Class and Struct (official documentation).

Share Follow

edited Jun 18 '20 at 16:49

answered Jun 18 '20 at 16:38
Arsen Khachaturyan
6,572 4 34 36

13

```
+------------------------+--------------------------------------------------------------
-----------------------------------------+----------------------------------------------
------------------------------------------------------+
|                        |                                                Struct
|                                               Class
|
+------------------------+--------------------------------------------------------------
-----------------------------------------+----------------------------------------------
------------------------------------------------------+
| Type                   | Value-type
| Reference-type
|
| Where                  | On stack / Inline in containing type
| On Heap
|
| Deallocation           | Stack unwinds / containing type gets deallocated
| Garbage Collected
|
| Arrays                 | Inline, elements are the actual instances of the value type
| Out of line, elements are just references to instances of the reference type residing
on the heap |
| Aldel Cost             | Cheap allocation-deallocation
| Expensive allocation-deallocation
|
| Memory usage           | Boxed when cast to a reference type or one of the interfaces
they implement,                         | No boxing-unboxing
|
|                        | Unboxed when cast back to value type
|
|
|                        | (Negative impact because boxes are objects that are
| Affect all references pointing to the instance
|
| Mutability              | Should be immutable
| Mutable
|
| Population              | In some situations
| Majority of types in a framework should be classes
|
| Lifetime                | Short-lived
| Long-lived
|
| Destructor              | Cannot have
| Can have
|
| Inheritance             | Only from an interface
| Full support
|
| Polymorphism            | No
| Yes
|
| Sealed                  | Yes
| When have sealed keyword
|
| Constructor             | Can not have explicit parameterless constructors
| Any constructor
|
| Null-assignments        | When marked with nullable question mark
| Yes (+ When marked with nullable question mark in C# 8+)
|
| Abstract                | No
| When have abstract keyword
|
| Member Access Modifiers| public, private, internal
| public, protected, internal, protected internal, private protected
|
+-------------------------+-----------------------------------------------------------
----------------------------------------+-----------------------------------------------
------------------------------------------------------+
```

Share Follow   edited Mar 27 '20 at 13:21   answered Jul 26 '18 at 10:43

explanations in some rows, also there are some typos. – Robert Synoradzki Nov 5 '18 at 11:28

1 @ensisNoctis Sorry for those mistakes and thanks for the edit. I should re-read my answers 😅 – Avestura Nov 6 '18 at 18:10

13

To add to the other answers, there is one fundamental difference that is worth noting, and that is how the data is stored within arrays as this can have a major effect on performance.

- With a struct, the array contains the instance of the struct
- With a class, the array contains a pointer to an instance of the class elsewhere in memory

So an array of structs looks like this in memory

```
[struct][struct][struct][struct][struct][struct][struct][struct]
```

Whereas an array of classes looks like this

```
[pointer][pointer][pointer][pointer][pointer][pointer][pointer][pointer]
```

With an array of classes, the values you're interested in are not stored within the array, but elsewhere in memory.

For a vast majority of applications this difference doesn't really matter, however, in high performance code this will affect locality of data within memory and have a large impact on the performance of the CPU cache. Using classes when you could/should have used structs will massively increase the number of cache misses on the CPU.

The slowest thing a modern CPU does is not crunching numbers, it's fetching data from memory, and an L1 cache hit is many times faster than reading data from RAM.

Here's some code you can test. On my machine, iterating through the class array takes ~3x longer than the struct array.

```
    private struct PerformanceStruct
    private class PerformanceClass
    {
        public int i1;
        public int i2;
    }

    private static void DoTest()
    {
        var structArray = new PerformanceStruct[100000000];
        var classArray = new PerformanceClass[structArray.Length];

        for (var i = 0; i < structArray.Length; i++)
        {
            structArray[i] = new PerformanceStruct();
            classArray[i] = new PerformanceClass();
        }

        long total = 0;
        var sw = new Stopwatch();
        sw.Start();
        for (var loops = 0; loops < 100; loops++)
        for (var i = 0; i < structArray.Length; i++)
        {
            total += structArray[i].i1 + structArray[i].i2;
        }

        sw.Stop();
        Console.WriteLine($"Struct Time: {sw.ElapsedMilliseconds}");
        sw = new Stopwatch();
        sw.Start();
        for (var loops = 0; loops < 100; loops++)
        for (var i = 0; i < classArray.Length; i++)
        {
            total += classArray[i].i1 + classArray[i].i2;
        }

        Console.WriteLine($"Class Time: {sw.ElapsedMilliseconds}");
    }
```

Share Follow   edited Apr 24 '20 at 10:20   answered Aug 25 '15 at 13:00

already understood it from the other answers here, and *"With a class the containing class will just contain a pointer to the new class in a different area of memory."* confuses classes with class instances. – Mark Amery Dec 16 '17 at 23:59

1 @MarkAmery I've tried to clarify slightly. The point I was really trying to make was the difference in the ways that arrays work with value and reference types and the effect this has on performance. I wasn't trying to re-explain what value and reference types are as this is done in plenty of other answers. – Will Calderwood Feb 23 '18 at 9:29

11

Structure vs Class

A structure is a value type so it is stored on the stack, but a class is a reference type and is stored on the heap.

A structure doesn't support inheritance, and polymorphism, but a class supports both.

By default, all the struct members are public but class members are by default private in nature.

As a structure is a value type, we can't assign null to a struct object, but it is not the case for a class.

Share Follow

edited Apr 6 '15 at 11:43
Peter Mortensen
28.5k 21 95 123

answered Nov 6 '09 at 8:02
Swagatika dhal
127 1 2

6 Regarding "all the struct members are public": If I'm not mistaken, that is incorrect. "The access level for class members and struct members, including nested classes and structs, is private by default." msdn.microsoft.com/en-us/library/ms173121.aspx – Nate Cook Sep 1 '15 at 16:14

" we can't assign null to a struct object" is not true as you can use the Nullable type with structs. – Denis Jul 20 '20 at 14:17

8

Well, for starters, a struct is passed by value rather than by reference. Structs are good for relatively simple data structures, while classes have a lot more flexibility from an architectural point of view via polymorphism and inheritance.

116k 20 165 244

Class objects are also passed by value by default – Imad Feb 18 at 16:18

@Imad: value of the reference, yes – Ed S. Mar 9 at 17:02

8

Just to make it complete, there is another difference when using the `Equals` method, which is inherited by all classes and structures.

Lets's say we have a class and a structure:

```
class A{
  public int a, b;
}
struct B{
  public int a, b;
}
```

and in the Main method, we have 4 objects.

```
static void Main{
  A c1 = new A(), c2 = new A();
  c1.a = c1.b = c2.a = c2.b = 1;
  B s1 = new B(), s2 = new B();
  s1.a = s1.b = s2.a = s2.b = 1;
}
```

Then:

```
s1.Equals(s2) // true
s1.Equals(c1) // false
c1.Equals(c2) // false
```

Share Follow

edited Dec 16 '17 at 23:50
Mark Amery
113k 59 359 411

answered Aug 2 '17 at 13:34
Ning
434 5 5

5

1. Events declared in a class have their += and -= access automatically locked via a lock(this) to make them thread safe (static events are locked on the typeof the class). Events declared in a struct do not have their += and -= access automatically locked. A lock(this) for a struct would not work since you can only lock on a reference type expression.
2. Creating a struct instance cannot cause a garbage collection (unless the constructor directly or indirectly creates a reference type instance) whereas creating a reference type instance can cause garbage collection.
3. A struct always has a built-in public default constructor.

```
class DefaultConstructor
{
    static void Eg()
    {
        Direct     yes = new   Direct(); // Always compiles OK
        InDirect maybe = new InDirect(); // Compiles if constructor exists and is
accessible
        //...
    }
}
```

This means that a struct is always instantiable whereas a class might not be since all its constructors could be private.

```
class NonInstantiable
{
    private NonInstantiable() // OK
    {
    }
}

struct Direct
{
```

4. A struct cannot have a destructor. A destructor is just an override of object.Finalize in disguise, and structs, being value types, are not subject to garbage collection.

```
struct Direct
{
    ~Direct() {} // Compile-time error
}
class InDirect
{
    ~InDirect() {} // Compiles OK
}

And the CIL for ~Indirect() looks like this:

.method family hidebysig virtual instance void
        Finalize() cil managed
{
  // ...
} // end of method Indirect::Finalize
```

5. A struct is implicitly sealed, a class isn't.
   A struct can't be abstract, a class can.
   A struct can't call : base() in its constructor whereas a class with no explicit base class can.
   A struct can't extend another class, a class can.
   A struct can't declare protected members (for example, fields, nested types) a class can.
   A struct can't declare abstract function members, an abstract class can.
   A struct can't declare virtual function members, a class can.
   A struct can't declare sealed function members, a class can.
   A struct can't declare override function members, a class can.
   The one exception to this rule is that a struct can override the virtual methods of System.Object, viz, Equals(), and GetHashCode(), and ToString().

Share Follow

edited Apr 6 '15 at 11:46

answered Jul 25 '11 at 5:24

@supercat Yeah, a non-static event in a struct would be very strange, and it will be useful only for mutable structs, and the event itself (if it is a "field-like" event) turns the struct into the "mutable" category and also introduces a field of reference type in the struct. Non-static events in structs must be evil. – Jeppe Stig Nielsen Mar 15 '14 at 22:05

@JeppeStigNielsen: The only pattern I could see where it would make sense for a struct to have an event would be if the purpose of the struct was to hold an immutable reference to a class object for which it behaved as a proxy. Auto-events would be totally useless in such a scenario, though; instead, subscribe and unsubscribe events would have to be relayed to the class behind the structure. I wish .NET had (or would make it possible to define) a 'cache-box" structure type with an initially-null hidden field of type `Object`, which would hold a reference to a boxed copy of the struct. – supercat Mar 15 '14 at 22:18

1 @JeppeStigNielsen: Structs outperform classes in many proxy usage scenarios; the biggest problem with using structs is that in cases where boxing ends up being necessary, it often ends up getting deferred to an inner loop. If there were a way to avoid having structures get boxed *repeatedly*, they would be better than classes in many more usage scenarios. – supercat Mar 15 '14 at 22:23

5

As previously mentioned: Classes are reference type while Structs are value types with all the consequences.

As a thumb of rule Framework Design Guidelines recommends using Structs instead of classes if:

- It has an instance size under 16 bytes
- It logically represents a single value, similar to primitive types (int, double, etc.)
- It is immutable
- It will not have to be boxed frequently

Share Follow

answered Sep 23 '15 at 11:32

kb9
339 2 7

Besides the basic difference of access specifier, and few mentioned above I would like to add some of the major differences including few of the mentioned above with a code sample with output, which will give a more clear idea of the reference and value

- Memory allocation is different and is stored in stack
- Useful for small data structures
- Affect performance, when we pass value to method, we pass the entire data structure and all is passed to the stack.
- Constructor simply returns the struct value itself (typically in a temporary location on the stack), and this value is then copied as necessary
- The variables each have their own copy of the data, and it is not possible for operations on one to affect the other.
- Do not support user-specified inheritance, and they implicitly inherit from type object

Class:

- Reference Type value
- Stored in Heap
- Store a reference to a dynamically allocated object
- Constructors are invoked with the new operator, but that does not allocate memory on the heap
- Multiple variables may have a reference to the same object
- It is possible for operations on one variable to affect the object referenced by the other variable

Code Sample

```
static void Main(string[] args)
{
    //Struct
    myStruct objStruct = new myStruct();
    objStruct.x = 10;
    Console.WriteLine("Initial value of Struct Object is: " + objStruct.x);
    Console.WriteLine();
    methodStruct(objStruct);
    Console.WriteLine();
    Console.WriteLine("After Method call value of Struct Object is: " +
    Console.WriteLine();
    methodClass(objClass);
    Console.WriteLine();
    Console.WriteLine("After Method call value of Class Object is: " + objClass.x);
    Console.Read();
}
static void methodStruct(myStruct newStruct)
{
    newStruct.x = 20;
    Console.WriteLine("Inside Struct Method");
    Console.WriteLine("Inside Method value of Struct Object is: " + newStruct.x);
}
static void methodClass(myClass newClass)
{
    newClass.x = 20;
    Console.WriteLine("Inside Class Method");
    Console.WriteLine("Inside Method value of Class Object is: " + newClass.x);
}
public struct myStruct
{
    public int x;
    public myStruct(int xCons)
    {
        this.x = xCons;
    }
}
public class myClass
{
    public int x;
    public myClass(int xCons)
    {
        this.x = xCons;
    }
}
```

Output

Initial value of Struct Object is: 10

Inside Class Method Inside Method value of Class Object is: 20

After Method call value of Class Object is: 20

Here you can clearly see the difference between call by value and call by reference.

Share Follow

edited May 30 '14 at 22:27
Brad Larson ♦
169k 45 388 563

answered May 29 '14 at 9:45
Arijit Mukherjee
3,459 2 29 48

3

There is one interesting case of "class vs struct" puzzle - situation when you need to return several results from the method: choose which to use. If you know the ValueTuple story - you know that ValueTuple (struct) was added because it should be more effective then Tuple (class). But what does it mean in numbers? Two tests: one is struct/class that have 2 fields, other with struct/class that have 8 fields (with dimension more then 4 - class should become more effective then struct in terms processor ticks, but of course GC load also should be considered).

P.S. Another benchmark for specific case 'sturct or class with collections' is there: https://stackoverflow.com/a/45276657/506147

```
BenchmarkDotNet=v0.10.10, OS=Windows 10 Redstone 2 [1703, Creators Update]
(10.0.15063.726)
Processor=Intel Core i5-2500K CPU 3.30GHz (Sandy Bridge), ProcessorCount=4
Frequency=3233540 Hz, Resolution=309.2586 ns, Timer=TSC
.NET Core SDK=2.0.3
  [Host] : .NET Core 2.0.3 (Framework 4.6.25815.02), 64bit RyuJIT
  Clr    : .NET Framework 4.7 (CLR 4.0.30319.42000), 64bit RyuJIT-v4.7.2115.0
  Core   : .NET Core 2.0.3 (Framework 4.6.25815.02), 64bit RyuJIT


            Method |  Job | Runtime |      Mean |      Error |     StdDev |       Min |
Max |    Median | Rank |  Gen 0 | Allocated |
------------------ |----- |-------- |---------:|----------:|----------:|---------:|----
-----:|---------:|-----:|-------:|----------:|
  TestStructReturn |  Clr |     Clr | 17.57 ns | 0.1960 ns | 0.1834 ns | 17.25 ns |
17.89 ns | 17.55 ns |    4 | 0.0127 |      40 B |
```

Join Stack Overflow to learn, share knowledge, and build your career.

Sign up with email | Sign up with Google | Sign up with GitHub | Sign up with Facebook

```
   TestStructReturn | Core |     Core | 12.28 ns | 0.1882 ns | 0.1760 ns | 11.92 ns |
12.57 ns | 12.30 ns |     1 | 0.0127 |       40 B |
    TestClassReturn | Core |     Core | 15.33 ns | 0.4343 ns | 0.4063 ns | 14.83 ns |
16.44 ns | 15.31 ns |     2 | 0.0229 |       72 B |
  TestStructReturn8 | Core |     Core | 34.11 ns | 0.7089 ns | 1.4954 ns | 31.52 ns |
36.81 ns | 34.03 ns |     7 | 0.0127 |       40 B |
   TestClassReturn8 | Core |     Core | 17.04 ns | 0.2299 ns | 0.2150 ns | 16.68 ns |
17.41 ns | 16.98 ns |     3 | 0.0305 |       96 B |
```

Code test:

```
using System;
using System.Text;
using System.Collections.Generic;
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Attributes.Columns;
using BenchmarkDotNet.Attributes.Exporters;
using BenchmarkDotNet.Attributes.Jobs;
using DashboardCode.Routines.Json;

namespace Benchmark
{
    //[Config(typeof(MyManualConfig))]
    [RankColumn, MinColumn, MaxColumn, StdDevColumn, MedianColumn]
    [ClrJob, CoreJob]
    [HtmlExporter, MarkdownExporter]
    [MemoryDiagnoser]
    public class BenchmarkStructOrClass
    {
        static TestStruct testStruct = new TestStruct();
        static TestClass testClass = new TestClass();
        static TestStruct8 testStruct8 = new TestStruct8();
        static TestClass8 testClass8 = new TestClass8();
        [Benchmark]
        public void TestStructReturn()
        {
            testStruct.TestMethod();
        }
```

```
[Benchmark]
public void TestStructReturn8()
{
    testStruct8.TestMethod();
}

[Benchmark]
public void TestClassReturn8()
{
    testClass8.TestMethod();
}

public class TestStruct
{
    public int Number = 5;
    public struct StructType<T>
    {
        public T Instance;
        public List<string> List;
    }

    public int TestMethod()
    {
        var s = Method1(1);
        return s.Instance;
    }

    private StructType<int> Method1(int i)
    {
        return Method2(++i);
    }

    private StructType<int> Method2(int i)
    {
        return Method3(++i);
    }

    private StructType<int> Method3(int i)
    {
        return Method4(++i);
        x.Instance = ++i;
        return x;
    }
}

public class TestClass
{
    public int Number = 5;
    public class ClassType<T>
    {
        public T Instance;
        public List<string> List;
    }

    public int TestMethod()
    {
        var s = Method1(1);
        return s.Instance;
    }

    private ClassType<int> Method1(int i)
    {
        return Method2(++i);
    }

    private ClassType<int> Method2(int i)
    {
        return Method3(++i);
    }

    private ClassType<int> Method3(int i)
    {
        return Method4(++i);
    }

    private ClassType<int> Method4(int i)
    {
        var x = new ClassType<int>();
        x.List = new List<string>();
        x.Instance = ++i;
        return x;
    public struct StructType<T>
    {
        public T Instance1;
        public T Instance2;
        public T Instance3;
        public T Instance4;
        public T Instance5;
        public T Instance6;
        public T Instance7;
        public List<string> List;
    }

    public int TestMethod()
    {
        var s = Method1(1);
        return s.Instance1;
    }

    private StructType<int> Method1(int i)
    {
        return Method2(++i);
    }

    private StructType<int> Method2(int i)
    {
        return Method3(++i);
    }

    private StructType<int> Method3(int i)
    {
        return Method4(++i);
    }

    private StructType<int> Method4(int i)
    {
        var x = new StructType<int>();
        x.List = new List<string>();
        x.Instance1 = ++i;
        return x;
    }
}
```

```
                public T Instance1;
                public T Instance2;
                public T Instance3;
                public T Instance4;
                public T Instance5;
                public T Instance6;
                public T Instance7;
                public List<string> List;
            }

            public int TestMethod()
            {
                var s = Method1(1);
                return s.Instance1;
            }

            private ClassType<int> Method1(int i)
            {
                return Method2(++i);
            }

            private ClassType<int> Method2(int i)
            {
                return Method3(++i);
            }

            private ClassType<int> Method3(int i)
            {
                return Method4(++i);
            }

            private ClassType<int> Method4(int i)
            {
                var x = new ClassType<int>();
                x.List = new List<string>();
                x.Instance1 = ++i;
                return x;
            }
        }
    }
}
```

2

> Structs are the actual value - they can be empty but never null

This is true, however also note that as of .NET 2 structs support a Nullable version and C# supplies some syntactic sugar to make it easier to use.

```
int? value = null;
value  = 1;
```

Share Follow

answered Aug 16 '08 at 14:03
denis phillips
12k 5 30 47

1 Be aware that this is only syntactic sugar which reads 'Nullable<int> value = null;' – Erik van Brakel Oct 4 '08 at 22:32

@ErikvanBrakel That's not just syntactic sugar. The different boxing rules mean `(object)(default(int?)) == null` which you can't do with any other value type, because there's more than just sugar going on here. The only sugar is `int?` for `Nullable<int>`. – Jon Hanna Dec 16 '13 at 21:11

-1; this doesn't address the question of what the difference between structs and classes is, and as such, it should've been a comment on the answer you're replying to, not a separate answer. (Although perhaps the site norms were different back in August 2008!) – Mark Amery Dec 17 '17 at 0:03

1

Every variable or field of a primitive value type or structure type holds a unique instance of that type, including all its fields (public and private). By contrast, variables or fields of reference types may hold null, or may refer to an object, stored elsewhere, to which any number of other references may also exist. The fields of a struct will be stored in the same place as the variable or field of that structure type, which may be either on the stack or may be *part of* another heap object.

Creating a variable or field of a primitive value type will create it with a default value; creating a variable or field of a structure type will create a new instance, creating all fields therein in the default manner. Creating a new *instance* of a reference type will start by creating all fields therein cause the latter to refer to the same instance as the former (if any).

It's important to note that in some languages like C++, the semantic behavior of a type is independent of how it is stored, but that isn't true of .NET. If a type implements mutable value semantics, copying one variable of that type to another copies the properties of the first to another instance, referred to by the second, and using a member of the second to mutate it will cause that second instance to be changed, but not the first. If a type implements mutable reference semantics, copying one variable to another and using a member of the second to mutate the object will affect the object referred to by the first variable; types with immutable semantics do not allow mutation, so it doesn't matter semantically whether copying creates a new instance or creates another reference to the first.

In .NET, it is possible for value types to implement any of the above semantics, provided that all of their fields can do likewise. A reference type, however, can only implement mutable reference semantics or immutable semantics; value types with fields of mutable reference types are limited to either implementing mutable reference semantics or weird hybrid semantics.

Share Follow

edited Apr 6 '15 at 11:54
Peter Mortensen
28.5k 21 95 123

answered Dec 2 '11 at 0:10
supercat
70.1k 7 144 186