# Почему нельзя наследоваться от value type

Проблема вот в чём. Допустим, разрешено было бы наследоваться от структур.

Рассмотрим такой код:

class C1

{

int x;

}

class C2 : C1

{

int y;

public override string ToString() { return y.ToString(); }

}

Вы можете написать так: C1 c = new C2(); и вызвать c.ToString();. Поскольку c является лишь ссылкой на данные, проблем с вызовом не возникает. Это стандартная фича, базовое полиморфное поведение ссылочных типов.

Представьте себе теперь аналогичную ситуацию:

struct S1

{

int x;

}

struct S2 : S1

{

int y;

public override string ToString() { return y.ToString(); }

}

S1 s = new S2();

Что будет в s? Из соображений эффективности структуры хранятся в памяти *как есть*, то есть, просто набор полей, а не ссылка на них. Это значит, что под s отводится столько байт, сколько занимает S1, и для поля y там просто нет места! Что в этом случае должен вернуть вызов s.ToString()?

Аналогичная проблема возникает при передаче параметров производного типа в функцию, требующую аргумент базового типа.

Эта проблема называется ***slicing***, и она актуальна для дизайна многих языков программирования. Например, она присутствует в C++. Решение в C# — отказаться от наследования структур: допустить неработающий полиморфизм при наследовании — плохая идея. При таком решении slicing в C# исключён.

Другие языки решают проблему по-другому. Например, в C++ настоящим типом объекта s будет просто S1, а не S2, и y просто потеряется. Создатели языка C# считают такое поведение неинтуитивным, ведущим к проблемам и ошибкам, поэтому они пошли по другому пути. Если вам нужно наследование, в C# нужно воспользоваться классами, или перейти к композиции.

Проблему *можно* было бы решить, если бы структуры хранились точно так же, как и классы: не значение, а ссылка на него. При этом операции наподобие присвоения должны были бы для сохранения семантики копировать не ссылку, а *значение* всё равно. Но такой подход, хоть и разрешил бы наследование структур, сильно ухудшил бы их *эффективность*: каждая операция со структурой стала бы медленнее из-за лишнего косвенного доступа, плюс аллокация структур в общем случае была бы вытеснена из стека в кучу.

Архитекторы языка разменяли полиморфизм структур на их эффективность.

# Почему быстрее создать лист классов, чем лист интов?

the reasons are probably mostly due to the list needing to allocate at least two objects, possibly more depending on how it is optimized.

For high performance code a common guideline is to avoid high frequency allocations. While allocations in c# are fast, they still take some time to manage. This often means sticking with fixed size arrays, or at least set the capacity of any lists on creation.

Another important point is using structs, they are stored directly in the list/array, instead of storing a reference to a separate object. This avoids some [object overhead](https://stackoverflow.com/questions/10655829/what-is-the-memory-overhead-of-a-net-object), removes the memory need of a separate reference, and ensures all values are stored sequentially in memory. All of this help ensure caches are used efficiently. Using a smaller datatype like short/ushort may also help if that is possible. Note that structs should preferably be [immutable](https://stackoverflow.com/questions/441309/why-are-mutable-structs-evil), and there are some keywords like 'ref' and ['in'](https://devblogs.microsoft.com/premier-developer/the-in-modifier-and-the-readonly-structs-in-c/) that can help avoid the overhead of copying data.

In some specific cases it can be an idea to separate values into different arrays, i.e. one for all the FromSquare values, one for all the ToSquare values etc. This can be a benefit if an operation mostly uses only a single value, again benefiting from better cache usage. It might also make SIMD easier to apply, but that might not apply in this case.

Moreover, when when measuring performance, at least use a [stopwatch](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-6.0). That is much more accurate than dateTime, and no harder to use. Benchmark.Net would be even better since that helps compensate for various sources of noise, and can benchmark for multiple platforms. A good profiler can also be useful since it can give hints at what takes most time, how much you are allocating etc.

# Как уменьшить количество аллокаций массивов (ArrayPool)

Remember, using ArrayPool instead of allocating with new puts the responsibility of freeing the memory on you. Your application will not leak memory if you don't guarantee that the Return method is called, but the ArrayPool is prevented from reusing the memory, thus denying the benefits you gain from using ArrayPool.

In simple use cases where you create a buffer and release it in the same method, it makes sense to put it into a finally clause:

private static void LocalUseOfSharedPool(int i)

{

int[] arr = ArrayPool<int>.Shared.Rent(ARRAYSIZE);

try

{

ShowAddress($"simple array {i}", arr);

FillTheArray(arr);

UseTheArray(arr);

}

finally

{

ArrayPool<int>.Shared.Return(arr);

}

}

In more complex cases you must make sure to not leak the memory in other ways.

Also note, that your buffer now has a lifetime. If you pass your array to another object B , you need to make sure, that object B is not using the array after your call to ArrayPool<>.Return.

Since using ArrayPool is a performance issue, measure the gains of using it, especially if you want to change an existing system.

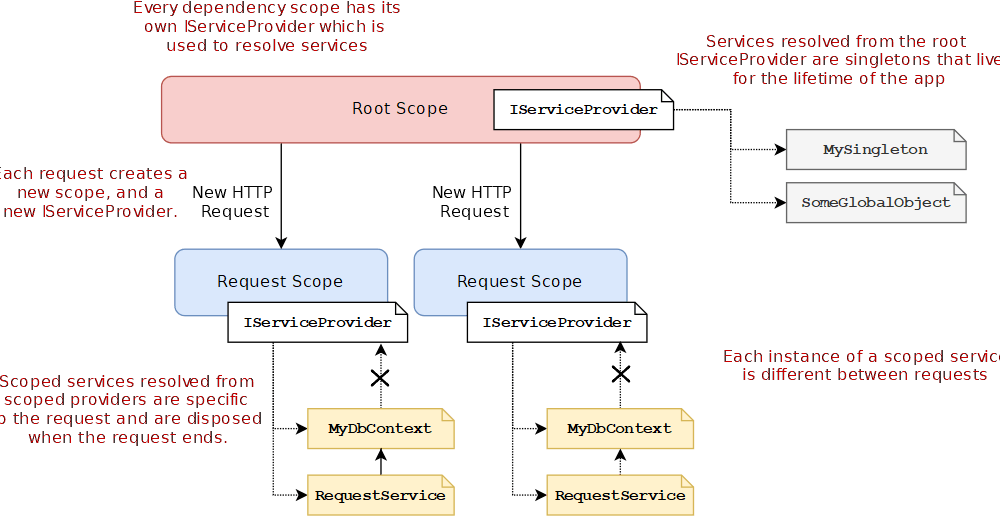
# Когда не сработает finally?

* **If the try block code goes into an infinite loop**, or the **thread is frozen and never unfrozen**, then the finally block code is never called.
* Environment.FailFast
* StackOverflowException
* ExecutingEngineException
* Power Failure

<https://stackoverflow.com/questions/3216046/does-the-c-sharp-finally-block-always-execute>

# Инджектнуть Scoped в Singletone

The [ASP.NET](http://asp.net/) Core DI container has a root IServiceProvider which is used to resolve singleton services. For scoped services, the container must first create a new scope, and each scope will have it's own IServiceProvider. Scoped services can only be accessed from the IServiceProvider within their own scope, and not from the root IServiceProvider.



## The Solution

To be able to use scoped services within a singleton, you must create a scope manually. A new scope can be created by injecting an IServiceScopeFactory into your singleton service (the IServiceScopeFactory is itself a singleton, which is why this works). The IServiceScopeFactory has a CreateScope method, which is used for creating new scope instances.

COPY

COPY

public class MySingletonService

{

private readonly IServiceScopeFactory \_serviceScopeFactory;

public MySingletonService(IServiceScopeFactory serviceScopeFactory)

{

\_serviceScopeFactory = serviceScopeFactory;

}

public void Execute()

{

using (var scope = \_serviceScopeFactory.CreateScope())

{

var myScopedService = scope.ServiceProvider.GetService<IMyScopedService>();

myScopedService.DoSomething();

}

}

}

The created scope has it's own IServiceProvider, which you can access to resolve your scoped services.

It is important to make sure that the scope only exists for as long as is necessary, and that it is properly disposed of once you have finished with it. This is to avoid any issues of captive dependencies (as discussed at the start of this article. Therefore, I would recommend:

* Only define the scope within the method that you intend to use it. It might be tempting to assign it to a field for reuse elsewhere in the singleton service, but again this will lead to captive dependencies.
* Wrap the scope in a using statement. This will ensure that the scope is properly disposed of once you have finished with it.

# Как заюзать Scoped сервис в бэкграунд джобе

<https://learn.microsoft.com/en-us/dotnet/core/extensions/scoped-service>