
# Sealed modifiers

Ivan Porta Follow
Mar 8, 2020 · 5 min read ★

In this article, I will discuss what is a sealed modifier, how to use it and what's its impact on your application's performance.

First of all, let's start with a definition; sealed is a modifier that, if it's applied to a class makes it **non-inheritable** and if applied to virtual methods or properties makes them **non-ovveridable**.

```
public sealed class A { ... }
public class B
{
    ...
    public sealed string Property { get; set; }
    public sealed void Method() { ... }
}
```

An example of its usage is **specialized class/method or property** in which potential alterations can make them stop working as expected (for example, the Pens class of the System.Drawing namespace).

```
...
namespace System.Drawing
{
    //
    // Summary:
    //     Pens for all the standard colors. This class
cannot be inherited.
    public sealed class Pens
    {
        public static Pen Transparent { get; }
        public static Pen Orchid { get; }
        public static Pen OrangeRed { get; }
        ...
    }
}
```

Because a sealed class cannot be inherited, it cannot be used as **base class** and by consequence, an **abstract class** cannot use the sealed modifier. It's also important to mention that **structs are implicitly sealed**.

## Performance

Using as reference for our test the following code, let's analyze the **Microsoft intermediate language (MSIL)** information generated by the compiler by using the **Ildasm.exe** (IL Disassembler) tool.

```
public sealed class Sealed
{
```

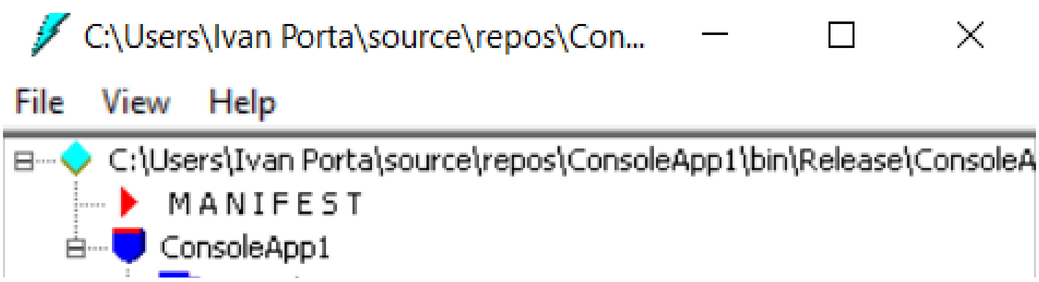
```
    public string Message { get; set; }
    public void DoStuff() { }
}
public class Derived : Base
{
    public sealed override void DoStuff() { }
}
public class Base
{
    public string Message { get; set; }
    public virtual void DoStuff() { }
}
static void Main()
{
    Sealed sealedClass = new Sealed();
    sealedClass.DoStuff();
    Derived derivedClass = new Derived();
    derivedClass.DoStuff();
    Base BaseClass = new Base();
    BaseClass.DoStuff();
}
```

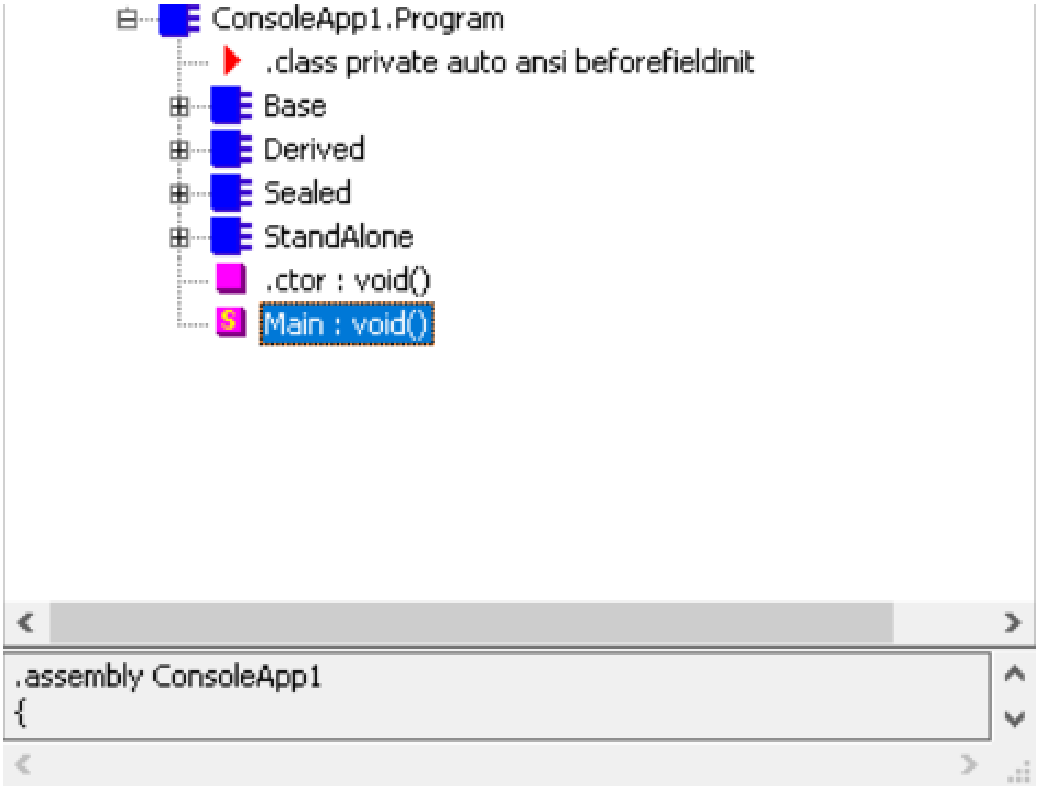
To run this tool, open the Developer Command Prompt for Visual Studio and execute the command **ildasm**.

```
*************************************************************
*********
** Visual Studio 2017 Developer Command Prompt v15.9.13
** Copyright (c) 2017 Microsoft Corporation
*************************************************************
*********

C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community>ildasm
```

Once the application is started, load the executable (or assembly) of the previous application






Double click on the Main method to view the Microsoft intermediate language (MSIL) information.

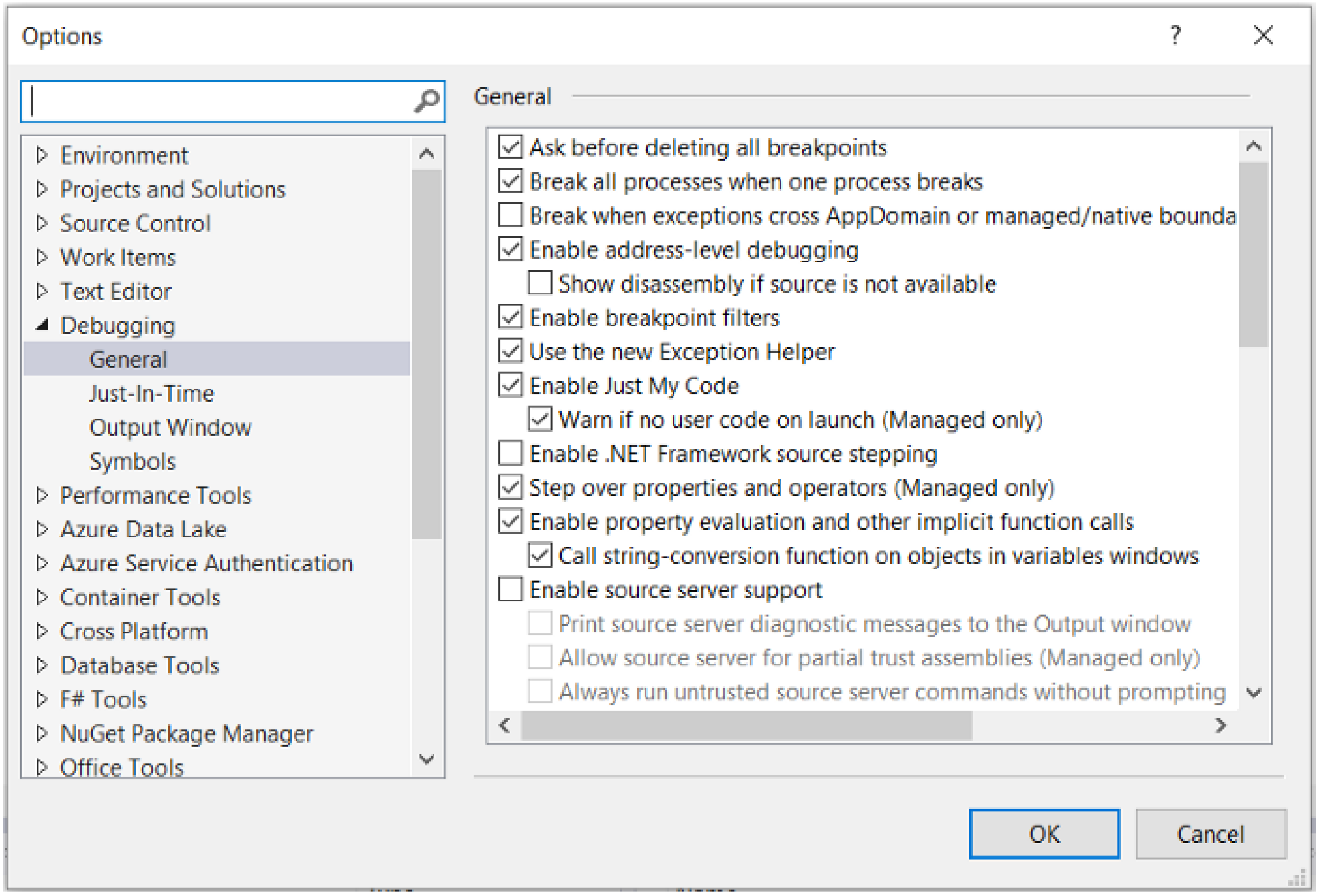
```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       41 (0x29)
  .maxstack  8
  IL_0000:  newobj     instance void
ConsoleApp1.Program/Sealed::.ctor()
  IL_0005:  callvirt   instance void
ConsoleApp1.Program/Sealed::DoStuff()
  IL_000a:  newobj     instance void
ConsoleApp1.Program/Derived::.ctor()
  IL_000f:  callvirt   instance void
ConsoleApp1.Program/Base::DoStuff()
  IL_0014:  newobj     instance void
ConsoleApp1.Program/Base::.ctor()
  IL_0019:  callvirt   instance void
ConsoleApp1.Program/Base::DoStuff()
  IL_0028:  ret
} // end of method Program::Main
```

As you can see each class use **newobj** to create a new instance by pushing an object reference onto the stack and **callvirt** to calls a late-bound of the

DoStuff() method of its respective object.

Base on this information seems that both sealed, derived and base classes are managed in the same way by the compiler. Just to be sure, let's get deeper by analyzing the **JIT-compiled code** with the **Disassembly** window in Visual Studio.

Enable the Disassembly by selecting Enable address-level debugging, under **Tools > Options > Debugging > General**.




Set a brake-point at the beginning of the application and start the debug. Once the application hits the brake-point open the Disassembly window by selecting **Debug > Windows > Disassembly**.

```
--- C:\Users\Ivan Porta\source\repos\ConsoleApp1\Program.cs -
-------------------
        {
0066084A  in          al,dx
0066084B  push        edi
0066084C  push        esi
```

```
0066084D  push        ebx
0066084E  sub         esp,4Ch
00660851  lea         edi,[ebp-58h]
00660854  mov         ecx,13h
00660859  xor         eax,eax
0066085B  rep stos    dword ptr es:[edi]
0066085D  cmp         dword ptr ds:[5842F0h],0
00660864  je          0066086B
00660866  call        744CFAD0
0066086B  xor         edx,edx
0066086D  mov         dword ptr [ebp-3Ch],edx
00660870  xor         edx,edx
00660872  mov         dword ptr [ebp-48h],edx
00660875  xor         edx,edx
00660877  mov         dword ptr [ebp-44h],edx
0066087A  xor         edx,edx
0066087C  mov         dword ptr [ebp-40h],edx
0066087F  nop
            Sealed sealedClass = new Sealed();
00660880  mov         ecx,584E1Ch
00660885  call        005730F4
0066088A  mov         dword ptr [ebp-4Ch],eax
0066088D  mov         ecx,dword ptr [ebp-4Ch]
00660890  call        00660468
00660895  mov         eax,dword ptr [ebp-4Ch]
00660898  mov         dword ptr [ebp-3Ch],eax
            sealedClass.DoStuff();
0066089B  mov         ecx,dword ptr [ebp-3Ch]
0066089E  cmp         dword ptr [ecx],ecx
006608A0  call        00660460
006608A5  nop
            Derived derivedClass = new Derived();
006608A6  mov         ecx,584F3Ch
006608AB  call        005730F4
006608B0  mov         dword ptr [ebp-50h],eax
006608B3  mov         ecx,dword ptr [ebp-50h]
006608B6  call        006604A8
006608BB  mov         eax,dword ptr [ebp-50h]
006608BE  mov         dword ptr [ebp-40h],eax
            derivedClass.DoStuff();
006608C1  mov         ecx,dword ptr [ebp-40h]
006608C4  mov         eax,dword ptr [ecx]
006608C6  mov         eax,dword ptr [eax+28h]
006608C9  call        dword ptr [eax+10h]
006608CC  nop
            Base BaseClass = new Base();
006608CD  mov         ecx,584EC0h
006608D2  call        005730F4
006608D7  mov         dword ptr [ebp-54h],eax
006608DA  mov         ecx,dword ptr [ebp-54h]
006608DD  call        00660490
006608E2  mov         eax,dword ptr [ebp-54h]
006608E5  mov         dword ptr [ebp-44h],eax
```

```
              BaseClass.DoStuff();
 006608E8  mov         ecx,dword ptr [ebp-44h]
 006608EB  mov         eax,dword ptr [ecx]
 006608ED  mov         eax,dword ptr [eax+28h]
 006608F0  call        dword ptr [eax+10h]
 006608F3  nop
         }
 0066091A  nop
 0066091B  lea         esp,[ebp-0Ch]
 0066091E  pop         ebx
 0066091F  pop         esi
 00660920  pop         edi
 00660921  pop         ebp

 00660922  ret
```

As we can see in the previous code, while the creation of the objects is the same, the instruction executed to invoke the methods of the sealed and derived/base class are slightly different. After moving data into registers of the RAM (**mov** instruction), the invoke of the sealed method, execute a comparison between dword ptr [ecx] and ecx (**cmp** instruction) before actually call the method.

According to the report written by Torbj¨orn Granlund, *Instruction latencies and throughput for AMD and Intel x86 processors*, the speed of the following instruction in an Intel Pentium 4 are:

- **mov**: has 1 cycle as latency and the processor can sustain 2.5 instructions per cycle of this type
- **cmp**: has 1 cycle as latency and the processor can sustain 2 instructions per cycle of this type

In conclusion, the optimization of the nowadays compilers and processors has made the performances between sealed and not-sealed classed basically so little that is irrelevant to the majority of the applications.

## References

- **NewObj**: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.newobj?