Is the "when" keyword in a try catch block the same as an if statement?

Asked 8 years, 3 months ago Modified 7 years, 1 month ago Viewed 7k times

60

In C# 6.0 the "when" keyword was introduced, now you're able to filter an exception in a catch block. But isn't this the same as a if statement inside a catch block? if so, isn't it just syntactic sugar or i'm missing something?

For example a try catch block with the "when" keyword:

```
try { … }
catch (WebException ex) when ex.Status == WebExceptionStatus.Timeout {
   //do something
}
catch (WebException ex) when ex.Status== WebExceptionStatus.SendFailure {
   //do something
}
catch (Exception caught) {…}
```

Or

```
try { … }
catch (WebException ex) {
   if(ex.Status == WebExceptionStatus.Timeout) {
      //do something
   }
}
catch (WebException ex) {
   if(ex.Status == WebExceptionStatus.SendFailure) {
      //do something
   }
}
catch (Exception caught) {…}
```

c# try-catch c#-6.0

Share
Improve this question
Follow

edited Sep 23, 2016 at 1:01
PC Luddite
6,078 ● 6 ● 25 ● 40

asked Sep 15, 2016 at 8:59
Undeadparade
1,542 ● 2 ● 14 ● 32

20 Exception filters don't unwind the stack – stuartd Sep 15, 2016 at 9:02

3 Simply put, if the when-clause returns false, the exception is not actually caught at all. With an if-statement, you need to rethrow the exception. (and can only be caught with another,

surrounding try-catch) – Dennis_E Sep 15, 2016 at 9:11

2 Your last examples with `if` still catch the `WebException` in all cases! The `if` means that the "do something" part is only executed if the criterion is met, but the `catch` has still occurred. It makes a huge difference if the exception is caught, or it propagates on. In your example, the last `catch` which works on type `Exception` will not be considered when an "earlier" `catch` has already been active. The `when` exception filter prevents the catch itself, not just the code execution inside the block. // If you added `else { throw; }` to your examples, stuartd's comment would apply. – Jeppe Stig Nielsen Sep 15, 2016 at 11:30

If it helps, think of `catch (WebException ex)` as syntactic sugar for `catch(object o) when (ex = o as WebException) != null` - the type check happens at the same time (well, immediately before) the filter, and IIRC is technically part of the filter from a MSIL point of view. – Random832 Sep 15, 2016 at 14:31

4 Answers

Sorted by: Highest score (default)

75

In addition to the several fine answers you already have here: there is a *very important difference* between an exception filter and an "if" in a catch block: **filters run before inner finally blocks**.

Consider the following:

```
void M1()
{
  try { N(); } catch (MyException) { if (F()) C(); }
}
void M2()
{
  try { N(); } catch (MyException) when F() { C(); }
}
void N()
{
  try { MakeAMess(); DoSomethingDangerous(); }
  finally { CleanItUp(); }
}
```

The order of calls differs between M1 and M2.

Suppose M1 is called. It calls N(), which calls MakeAMess(). A mess is made. Then DoSomethingDangerous() throws MyException. The runtime checks to see if there is any catch block that can handle that, and there is. The finally block runs CleanItUp(). The mess is cleaned up. Control passes to the catch block. And the catch block calls F() and then, maybe, C().

What about M2? It calls N(), which calls MakeAMess(). A mess is made. Then DoSomethingDangerous() throws MyException. The runtime checks to see if there is any catch block that can handle that, and there is -- maybe. The runtime calls F() to

see if the catch block can handle it, and it can. The finally block runs CleanItUp(), control passes to the catch, and C() is called.

Did you notice the difference? In the M1 case, F() is called *after the mess is cleaned up*, and in the M2 case, it is called *before* the mess is cleaned up. If F() depends on there being no mess for its correctness then you are in big trouble if you refactor M1 to look like M2!

There are more than just correctness problems here; there are security implications as well. Suppose the "mess" we are making is "impersonate the administrator", the dangerous operation requires admin access, and the cleanup un-impersonates administrator. In M2, the call to F *has administrator rights*. In M1 it does not. Suppose that the user has granted few privileges to the assembly containing M2 but N is in a full-trust assembly; potentially-hostile code in M2's assembly could gain administrator access through this luring attack.

As an exercise: how would you write N so that it defends against this attack?

(Of course the runtime is smart enough to know if there are *stack annotations* that grant or deny privileges between M2 and N, and it reverts those before calling F. That's a mess that the runtime made and it knows how to deal with it correctly. But the runtime doesn't know about any other mess that *you* made.)

The key takeaway here is that any time you are handling an exception, by definition something went horribly wrong, and the world is not as you think it should be. *Exception filters must not depend for their correctness on invariants that have been violated by the exceptional condition.*

UPDATE:

Ian Ringrose asks how we got into this mess.

This portion of the answer will be *somewhat* conjectural as some of the design decisions described here were undertaken after I left Microsoft in 2012. However I've chatted with the language designers about these issues many times and I think I can give a fair summary of the situation.

The design decision to make filters run before finally blocks was taken in the very early days of the CLR; the person to ask if you want the small details of that design decision would be Chris Brumme. (UPDATE: Sadly, Chris is no longer available for questions.) He used to have a blog with a detailed exegesis of the exception handling model, but I don't know if it is still on the internet.

It's a reasonable decision. For debugging purposes we want to know *before* the finally blocks run whether this exception is going to be handled, or if we're in the "undefined

behaviour" scenario of a completely unhandled exception destroying the process. Because if the program is running in a debugger, that undefined behaviour is going to include breaking at the point of the unhandled exception *before* the finally blocks run.

The fact that these semantics introduces security and correctness issues was very well understood by the CLR team; in fact I discussed it in my first book, which shipped a great many years ago now, and twelve years ago on my blog:

https://blogs.msdn.microsoft.com/ericlippert/2004/09/01/finally-does-not-mean-immediately/

And even if the CLR team wanted to, it would be a breaking change to "fix" the semantics now.

The feature has always existed in CIL and VB.NET, and the attacker controls the implementation language of the code with the filter, so introducing the feature to C# does not add any new attack surface.

And the fact that this feature that introduces a security issue has been "in the wild" for some decades now and to my knowledge has never been the cause of a serious security issue is evidence that it's not a very fruitful avenue for attackers.

Why then was the feature in the first version of VB.NET and took over a decade to make it into C#? Well, "why not" questions like that are hard to answer, but in this case I can sum it up easily enough: (1) we had a great many other things on our mind, and (2) Anders finds the feature unattractive. (And I'm not thrilled with it either.) That moved it to the bottom of the priority list for many years.

How then did it make it high enough on the priority list to be implemented in C# 6? Many people asked for this feature, which is always points in favour of doing it. VB already had it, and C# and VB teams like to have parity when possible at a reasonable cost, so that's points too. But the big tipping point was: there was a scenario *in the Roslyn project itself* where exception filters would have been really useful. (I do not recall what it was; go diving in the source code if you want to find it and report back!)

As both a language designer and compiler writer, you want to be careful to *not* prioritize the features that make *only* compiler writer's lives easier; most C# users are not compiler writers, and they're the customers! But ultimately, having a collection of real-world scenarios where the feature is useful, including some that were irritating the compiler team itself, tipped the balance.

edited Oct 25, 2017 at 16:24

answered Sep 15, 2016 at 15:32

Eric Lippert
659k ● 183 ● 1.3k ● 2.1k

1 Given this mess, when was it thought worthwhile to add "when" and make it run before any clearnup? – Ian Ringrose Sep 15, 2016 at 20:37

2 @IanRingrose: It's complicated. I'll add some notes to the answer. – Eric Lippert Sep 15, 2016 at 22:17

46

> But isn't this the same as a if statement inside a catch block?

No, because your second approach without `when` won't reach the second `Catch` if the `ex.Status== WebExceptionStatus.SendFailure`. With `when` the first `Catch` would have been skipped.

So the only way to handle the `Status` without `when` is to have the logic in one `catch`:

```
try { … }
catch (WebException ex) {
   if(ex.Status == WebExceptionStatus.Timeout) {
      //do something
   }
   else if(ex.Status == WebExceptionStatus.SendFailure) {
      //do something
   }
   else
      throw; // see Jeppe's comment
}
catch (Exception caught) {…}
```

The `else throw` will ensure that only `WebExceptions` with `status=Timeout` or `SendFailure` are handled here, similar to the `when` approach. All others will not be handled and the exception will be propagated. Note that it won't be caught by the last `Catch`, so there's still a difference to the `when`. This shows one of the advantages of `when`.

Share

Improve this answer

Follow

edited Sep 15, 2016 at 14:25

answered Sep 15, 2016 at 9:01

Tim Schmelter

460k ● 77 ● 709 ● 968

1 @RichaGarg: if we rethrow the exception in the if you won't catch it with the second `Catch`. That's what i meant with *"won't reach the second Catch"*. Once you have caught an exception the other `Catch`-blocks are unreachable. – Tim Schmelter Sep 15, 2016 at 9:27

1 @TimSchmelter: I think you meant to write `else if` in your code example (I can't edit it because it's less than 6 characters) – hoffmale Sep 15, 2016 at 10:05

4 If you want to make it similar to the `when` example of the question, you should append a final `else { throw; }` to the `if`-`else if` construct! It is not only a matter of whether the

code in the `catch` block runs or not, it is also a matter of whether the exception propagation is stopped (exception caught) or not. – Jeppe Stig Nielsen Sep 15, 2016 at 12:12

2 @JeppeStigNielsen: good point, added to the answer. However, isn't it still different to the `when` approach? Because that will handle all other exceptions in the last `Catch` whereas this workaround will propagate the exception without handling it here. That's a good example why `when` is useful. – Tim Schmelter Sep 15, 2016 at 12:20

1 Yes, you are absolutely right! The `else { throw; }` only makes it a bit more similar to the original case. You cannot transfer control to the "next" `catch` blocks, one would have to use ugly nested `try` statements to get even closer to the original behavior. Also note that the fact that the exception was re-thrown from the line `throw;` will be seen in the stack trace. So it is not "invisible" that the exception was "caught" for a period before it "flew" again. With `when`, the exception is not caught and rethrown, it just propagates through the filter. – Jeppe Stig Nielsen Sep 15, 2016 at 12:55

11

> isn't this the same as a if statement inside a catch block?

No. It acts more as a "discriminator" for the benefit of the Exception-throwing system.

Remember how Exceptions are thrown *twice*?

The first "throw" (those "first-chance" Exceptions that 'Studio goes on about) tells the Run-Time to locate the *nearest* Exception Handler that can deal with this *Type* of Exception and to collect up any "finally" blocks between "here" and "there".

The second "throw" unwinds the call stack, executing each of those "finally" blocks in turn and then delivers the execution engine to the entry point of the located Exception handling code.

Previously, we could only discriminate between different *Types* of Exception. This decorator gives us *finer-grain* control, only catching a particular Type of Exception that *happens to be in a **state** that we can do something about*.
For example (out of thin air) you might want to handle a "Database Exception" that indicates a broken connection and, when that happens, try to reconnect automatically. *Lots* of database operations throw a "Database Exception", but you're only *interested* in a particular "Sub-Type" of them, based on the *properties* of the Exception object, all of which are available to the exception-throwing system.

An "if" statement inside the catch block *will* achieve the same end result, but it will "cost" more at run-time. Because this block will catch *any and all* "Database Exceptions", it will be invoked for *all* of them, even if it can only do something useful for a [very] small fraction of them. It also means that you then have to re-throw [all] the Exceptions that you *can't* do anything useful with, which is just repeating the whole, two-pass, handler-finding, finally-harvesting, Exception-throwing farago all over again.

Analogy: A [very strange] Toll bridge.

By default, you have to "catch" every car in order for them to pay the toll. If, say, cars driven by city employees are *exempt* from the toll (I *did* say it was strange), then you only need to stop cars driven by anybody else.

You could stop *every* car and ask:

```
catch( Car car )
{
   if ( car.needsToPayToll() )
      takePayment( car );
}
```

Or, if you had some way of "interrogating" the car as it approached, then you could *ignore* those driven by city employees, as in:

```
catch( Car car ) when car.needsToPayToll()
{
   takePayment( car );
}
```

Share Improve this answer Follow

answered Sep 15, 2016 at 10:59

Phill W.

316 ● 1 ● 4

Nice car explanation. This made it really easy to understand!! Thanks – Mario Garcia Sep 16, 2016 at 11:55

0

Extending the answer by Tim.

C# 6.0 introduces a new feature exception filter and a new keyword **when**.

> Is the "when" keyword in a try catch block the same as a if statement?

The when keyword works like if. A when condition is a predicate expression, which can be appended to a catch block. If the predicate expression is evaluated to be true, the associated catch block is executed; otherwise, the catch block is ignored.

A wonderful explanation is given on C# 6.0 Exception Filter and when Keyword

Share Improve this answer Follow

answered Sep 15, 2016 at 9:10

Mohit S

14k ● 6 ● 35 ● 72

8 This answer must be incorrect for the simple fact that it is *inconsistent*. You first say that it is the same as a `catch` block which uses an `if` and then go-on describing a behaviour that does **not** match what you just said. – Bakuriu Sep 15, 2016 at 15:03