Catch multiple exceptions at once?

Ask Question

1868

304

It is discouraged to simply catch `System.Exception`. Instead, only the "known" exceptions should be caught.

Now, this sometimes leads to unneccessary repetitive code, for example:

```
try
{
    WebId = new Guid(queryString["web"]);
}
catch (FormatException)
{
    WebId = Guid.Empty;
}
catch (OverflowException)
{
    WebId = Guid.Empty;
}
```

I wonder: Is there a way to catch both exceptions and only call the `WebId = Guid.Empty` call once?

The given example is rather simple, as it's only a `GUID`. But imagine code where you modify an object multiple times, and if one of the manipulations fail in an expected way, you want to "reset" the `object`. However, if there is an unexpected exception, I still want to throw that higher.

c# .net exception exception-handling

edited Oct 18 '17 at 12:39

Mark Amery
65.1k 31 258 305

asked Sep 25 '08 at 20:56

Michael Stum ♦
118k 99 363 506

5 If you are using .net 4 and above i prefer to use aggregateexception msdn.microsoft.com/en-us/library/system.aggregateexception.aspx – Bepenfriends Oct 18 '13 at 3:21

1 Bepenfriends- Since **System.Guid** does not throw **AggregateException**, it would be great if you (or someone) could post an answer showing how you would wrap it into an AggregateException etc.. – weir Jan 30 '14 at 18:53

1 On using `AggregateException` : Throwing an AggregateException in my own code – DavidRR Aug 14 '14 at 17:25

7 "It is discouraged to simply catch System.Exception." -and if method can throw 32 types of exceptions, what one does? write catch for each of them separately? – giorgim May 23 '15 at 20:22

1 Keep it the way you have it. Move the code to an error handler if you wish so theres only one line per catch statement. – rolls Mar 4 '17 at 3:03

Home

PUBLIC

Stack Overflow

Tags

Users

Jobs

Teams
Q&A for work

27 Answers

1922

Catch `System.Exception` and switch on the types

```
catch (Exception ex)
{
    if (ex is FormatException || ex is OverflowException)
    {
        WebId = Guid.Empty;
        return;
```

```
        }

        throw;
    }
```

edited Nov 26 '16 at 1:53

Joe
4,798 1 23 32

answered Sep 25 '08 at 21:01

Joseph Daigle
37.2k 10 41 64

53 Unfortunately, FxCop (ie - Visual Studio Code Analysis) doesn't like it when you catch Exception. – Andrew Garrison Aug 27 '10 at 19:48

12 I agree with not catching Exception, but, in this case, the catch is a filter. You may have a layer higher that will handle other exception types. I would say this is correct, even though it includes a catch(Exception x). It doesn't modify the program flow, it just handles certain exceptions then lets the rest of the application deal with any other exception types. – lkg Jun 14 '11 at 19:13

27 The latest version of FxCop does not throw an exception when the code above is used. – Peter Jul 8 '11 at 5:12

24 Not sure what was wrong with the OP's code in the first place. The #1 accepted answer is almost twice as many lines and far less readable. – João Bragança Sep 4 '12 at 21:57

19 @JoãoBragança: While this answer in this example uses more lines, try to imagine if you are dealing with file IO for example, and all you want to do is catch those exceptions and do some log messaging, but only those you expect coming from your file IO methods. Then you often have to deal with a larger number (about 5 or more) different types of exceptions. In that situation, this approach can save you some lines. – Xilconic Nov 28 '13 at 14:47

0

It is worth mentioning here. You can respond to the multiple combinations (Exception error and exception.message).

I ran into a use case scenario when trying to cast control object in a datagrid, with either content as TextBox, TextBlock or CheckBox. In this case the returned Exception was the same, but the message varied.

```
try
{
 //do something
}
catch (Exception ex) when (ex.Message.Equals("the_error_message1_her
{
//do whatever you like
}
catch (Exception ex) when (ex.Message.Equals("the_error_message2_her
{
//do whatever you like
}
```

answered Dec 7 '18 at 15:14

George

94 1 10

47

Exception filters are now available in c# 6+. You can do

```
try
{
       WebId = new Guid(queryString["web"]);
}
catch (Exception ex) when(ex is FormatException || ex is OverflowExc
{
     WebId = Guid.Empty;
}
```

answered Jul 11 '18 at 13:12

Mat J
3,324 2 31 47

4

Maybe try to keep your code simple such as putting the common code in a method, as you would do in any other part of the code that is not inside a catch clause?

E.g.:

```
try
{
    // ...
}
catch (FormatException)
{
    DoSomething();
}
catch (OverflowException)
{
    DoSomething();
}

// ...

private void DoSomething()
{
    // ...
}
```

Just how I would do it, trying to find the *simple is beautiful* pattern

answered Jan 23 '18 at 14:02

Żubrówka
482 1 7 24

18

With C# 7 [the answer from Michael Stum] can be improved while keeping the readability of a switch statement:

```
catch (Exception ex)
{
    switch (ex)
    {
        case FormatException _:
        case OverflowException _:
            WebId = Guid.Empty;
            break;
        default:
            throw;
    }
}
```

answered Jan 5 '18 at 11:43

Fabian
229 2 9

2 This should be the accepted answer as of 2018 IMHO. – MemphiZ Aug 9 '18 at 15:38

1 Mat J's answer using `when` is far more elegant/appropriate than a switch. – rgoliveira Nov 6 '18 at 15:03

@rgoliveira: I agree that for the case asked in the question, the answer by Mat J is more elegant and appropriate. However, it gets hard to read if you have different code you want to execute depending on the exception type or if you want to actually use the instance of the exception. All these scenarios can be treated equally with this switch statement. – Fabian Nov 7 '18 at 13:47

1 @Fabian "if you have different code you want to execute depending on the exception type or if you want to actually use the instance of the exception", then you just make a different `catch` block, or you'd need to cast it anyway... In my experience, a `throw;` in your `catch` block is probably a code smell. – rgoliveira Nov 7 '18 at 16:05

@rgoliveira: Using a throw in a catch block is OK in several cases see link. Since the case statement actually uses pattern matching link you do not need to cast if you substitute the discard operator link (the underscore) by a variable name. Don't get me wrong, I agree with you that exception

filters are a cleaner way doing it, but multiple catch blocks add a lot of curly brackets. – Fabian Nov 8 '18 at 17:30

6

This is a classic problem every C# developer faces eventually.

Let me break your question into 2 questions. The first,

Can I catch multiple exceptions at once?

In short, no.

Which leads to the next question,

How do I avoid writing duplicate code given that I can't catch multiple exception types in the same catch() block?

Given your specific sample, where the fall-back value is cheap to construct, I like to follow these steps:

1. Initialize WebId to the fall-back value.
2. Construct a new Guid in a temporary variable.
3. Set WebId to the fully constructed temporary variable. Make this the final statement of the try{} block.

So the code looks like:

```
try
{
    WebId = Guid.Empty;
    Guid newGuid = new Guid(queryString["web"]);
    // More initialization code goes here like
    // newGuid.x = y;
    WebId = newGuid;
}
catch (FormatException) {}
catch (OverflowException) {}
```

If any exception is thrown, then WebId is never set to the half-constructed value, and remains Guid.Empty.

If constructing the fall-back value is expensive, and resetting a value is much cheaper, then I would move the reset code into its own function:

```
try
{
    WebId = new Guid(queryString["web"]);
    // More initialization code goes here.
}
catch (FormatException) {
    Reset(WebId);
}
catch (OverflowException) {
    Reset(WebId);
}
```

answered Nov 15 '17 at 20:27

Jeffrey Rennie
1,778 1 7 12

This is nice, "ecological coding" i.e. you're thinking ahead about your code & data footprint and making sure no leakage of half processed values. Nice going to follow this pattern thanks Jeffrey! – Tahir Khalid Mar 5 '18 at 0:44

4

Wanted to added my short answer to this already long thread. Something that hasn't been mentioned is the order of precedence of the catch statements, more specifically you need to be aware of the scope of each type of exception you are trying to catch.

For example if you use a "catch-all" exception as **Exception** it will preceed all other catch statements and you will obviously get compiler errors however if you reverse the order you can chain up

your catch statements (bit of an anti-pattern I think) you can put the catch-all **Exception** type at the bottom and this will be capture any exceptions that didn't cater for higher up in your try..catch block:

```
try
{
    // do some work here
}
catch (WebException ex)
{
    // catch a web excpetion
}
catch (ArgumentException ex)
{
    // do some stuff
}
catch (Exception ex)
{
    // you should really surface your errors but this is
    throw new Exception("An error occurred: " + ex.Messa
}
```

I highly recommend folks review this MSDN document:

Exception Hierarchy

answered Apr 16 '17 at 15:51

Tahir Khalid

801 9 15

128

For the sake of completeness, since **.NET 4.0** the code can rewritten as:

```
Guid.TryParse(queryString["web"], out WebId);
```

TryParse never throws exceptions and returns false if format is wrong, setting WebId to `Guid.Empty` .

Since **C# 7** you can avoid introducing a variable on a separate line:

```
Guid.TryParse(queryString["web"], out Guid webId);
```

You can also create methods for parsing returning tuples, which aren't available in .NET Framework yet as of version 4.6:

```
(bool success, Guid result) TryParseGuid(string input) =>
    (Guid.TryParse(input, out Guid result), result);
```

And use them like this:

```
WebId = TryParseGuid(queryString["web"]).result;
// or
var tuple = TryParseGuid(queryString["web"]);
WebId = tuple.success ? tuple.result : DefaultWebId;
```

Next useless update to this useless answer comes when deconstruction of out-parameters is implemented in C# 12. :)

edited Mar 18 '17 at 12:40

answered Apr 13 '13 at 12:18

Athari
27.8k 10 84 125

18 Precisely--concise, and you totally bypass the performance penalty of handling the exception, the bad form of intentionally using exceptions to control program flow, and the soft focus of having your conversion logic spread around, a little bit here and a little bit there. – Craig Apr 17 '13 at 3:56

9 I know what you meant, but of course `Guid.TryParse` never returns `Guid.Empty`. If the string is in an incorrect format, it sets the `result`

output parameter to `Guid.Empty`, but it *returns* `false`. I'm mentioning it because I've seen code that does things in the style of `Guid.TryParse(s, out guid); if (guid == Guid.Empty) { /* handle invalid s */ }`, which is usually wrong if `s` could be the string representation of `Guid.Empty`. – user743382 May 18 '13 at 11:39

14 wow you have answered the question, except that it is not in the spirit of the question. The larger problem is something else :(– nawfal May 18 '13 at 20:01

6 The proper pattern for using TryParse, of course, is more like `if( Guid.TryParse(s, out guid){ /* success! */ } else { /* handle invalid s */ }`, which leaves no ambiguity like the broken example where the input value might actually be the string representation of a Guid. – Craig Feb 22 '14 at 1:55

2 This answer may indeed be correct in regard to Guid.Parse, but it has missed the entire point of the original question. Which had nothing to do with Guid.Parse, but was in regard to catching Exception vs FormatException/OverflowException/etc. – Conor Gallagher Sep 14 '15 at 12:39

446

EDIT: I do concur with others who are saying that, as of C# 6.0, exception filters are now a perfectly fine way to go: `catch (Exception ex) when (ex is ... || ex is ... )`

Except that I still kind of hate the one-long-line layout and would personally lay the code out like the following. I think this is as functional as it is aesthetic, since I believe it improves comprehension. Some may disagree:

```
catch (Exception ex) when (
    ex is ...
    || ex is ...
    || ex is ...
)
```

ORIGINAL:

I know I'm a little late to the party here, but holy smoke...

Cutting straight to the chase, this kind of duplicates an earlier answer, but if you really want to perform a common action for several exception types and keep the whole thing neat and tidy within the scope of the one method, why not just use a lambda/closure/inline function to do something like the following? I mean, chances are pretty good that you'll end up realizing that you just want to make that closure a separate method that you can utilize all over the place. But then it will be super easy to do that without actually changing the rest of the code structurally. Right?

```
private void TestMethod ()
{
    Action<Exception> errorHandler = ( ex ) => {
        // write to a log, whatever...
    };

    try
    {
        // try some stuff
    }
    catch ( FormatException  ex ) { errorHandler ( ex ); }
    catch ( OverflowException ex ) { errorHandler ( ex ); }
    catch ( ArgumentNullException ex ) { errorHandler ( ex ); }
}
```

I can't help but wonder (**warning:** a little irony/sarcasm ahead) why on earth go to all this effort to basically just replace the following:

```
try
{
    // try some stuff
}
catch( FormatException ex ){}
catch( OverflowException ex ){}
catch( ArgumentNullException ex ){}
```

...with some crazy variation of this next code smell, I mean example, only to pretend that you're saving a few keystrokes.

```
// sorta sucks, let's be honest...
try
{
    // try some stuff
}
catch( Exception ex )
{
    if (ex is FormatException ||
        ex is OverflowException ||
        ex is ArgumentNullException)
    {
        // write to a log, whatever...
        return;
    }
    throw;
}
```

Because it certainly isn't automatically more readable.

Granted, I left the three identical instances of `/* write to a log, whatever... */ return;` out of the first example.

But that's sort of my point. Y'all have heard of functions/methods, right? Seriously. Write a common `ErrorHandler` function and, like, call it from each catch block.

If you ask me, the second example (with the `if` and `is` keywords) is both significantly less readable, and simultaneously significantly more error-prone during the maintenance phase of your project.

The maintenance phase, for anyone who might be relatively new to programming, is going to comprise 98.7% or more of the overall lifetime of your project, and the poor schmuck doing the maintenance is almost certainly going to be someone other than you. And there is a very good chance they will spend 50% of their time on the job cursing your name.

And of course FxCop barks at you and so you have to ***also*** add an attribute to your code that has precisely zip to do with the running program, and is only there to tell FxCop to ignore an issue that in 99.9% of cases it is totally correct in flagging. And, sorry, I might be

mistaken, but doesn't that "ignore" attribute end up actually compiled into your app?

Would putting the entire `if` test on one line make it more readable? I don't think so. I mean, I did have another programmer vehemently argue once long ago that putting more code on one line would make it "run faster." But of course he was stark raving nuts. Trying to explain to him (with a straight face--which was challenging) how the interpreter or compiler would break that long line apart into discrete one-instruction-per-line statements--essentially identical to the result if he had gone ahead and just made the code readable instead of trying to out-clever the compiler--had no effect on him whatsoever. But I digress.

How much *less* readable does this get when you add three more exception types, a month or two from now? (Answer: it gets a ***lot*** less readable).

One of the major points, really, is that most of the point of formatting the textual source code that we're all looking at every day is to make it really, really obvious to other human beings what is actually happening when the code runs. Because the compiler turns the source code into something totally different and couldn't care less about your code formatting style. So all-on-one-line totally sucks, too.

Just saying...

```
// super sucks...
catch( Exception ex )
{
    if ( ex is FormatException || ex is OverflowException || ex is A
)
    {
        // write to a log, whatever...
        return;
    }
    throw;
}
```

edited Feb 14 '17 at 20:38

answered Oct 12 '13 at 0:24

Craig
5,750 1 14 20

28 When I first stumbled across this question I was all over the accepted answer. Cool I can just catch all `Exception` s and the check the type. I thought it cleaned up the code, but something kept me coming back to the question and I actually read the other answers to the question. I chewed on it for a while, but I have to agree with you. It is *more* readable and maintainable to use a function to dry up your code than to catch everything, check the type comparing against a list, wrapping code, and throwing. Thanks for coming late and providing an alternative and sane (IMO) option. +1. – erroric Mar 6 '14 at 14:55

8 Using an error handling function wouldn't work if you wanted to include a `throw;` . You'd have to repeat that line of code in each catch block (obviously not the end of the world but worth mentioning as it is code that would need to be repeated). – kad81 Jun 17 '14 at 8:48

5 @kad81, that's true, but you would still get the benefit of writing the logging and clean-up code in one place, and changing it in one place if need be, without the goofy semantics of catching the base Exception type then branching based on the exception type. And that one extra `throw();` statement in each catch block is a small price to pay, IMO, and still leaves you in the position to do additional exception type-specific cleanup if necessary. – Craig Jun 18 '14 at 18:39

2 Hi @Reitffunk, just use `Func<Exception, MyEnumType>` instead of `Action<Exception>` . That's `Func<T, Result>` , with `Result` being the return type. – Craig Mar 30 '15 at 16:42

3 I'm in complete agreement here. I too read the first answer and thought seems logical. Moved to a generic 1 for all exception handler. Something inside me made me internally puke... so I reverted the code. Then came across this beauty! This **needs** to be the accepted answer – Conor Gallagher Sep 14 '15 at 12:33

17

Joseph Daigle's Answer is a good solution, but I found the following structure to be a bit tidier and less error prone.

```
catch(Exception ex)
{
    if (!(ex is SomeException || ex is OtherException)) throw;

    // Handle exception
}
```

There are a few advantages of inverting the expression:

- A return statement is not necessary
- The code isn't nested
- There's no risk of forgetting the 'throw' or 'return' statements that in Joseph's solution are separated from the expression.

It can even be compacted to a single line (though not very pretty)

```
catch(Exception ex) { if (!(ex is SomeException || ex is OtherExcept
    // Handle exception
}
```

Edit: The exception filtering in C# 6.0 will make the syntax a bit cleaner and comes with a number of other benefits over any current solution. (most notably leaving the stack unharmed)

Here is how the same problem would look using C# 6.0 syntax:

```
catch(Exception ex) when (ex is SomeException || ex is OtherExceptio
{
    // Handle exception
}
```

edited May 23 '17 at 10:31

Community ♦
1 1

answered Dec 8 '14 at 20:31

Stefan T
407 4 7

2 +1, this is the best answer. It's better than the top answer mostly because there's no `return` , although inverting the condition is also a little better. – DCShannon Mar 18 '15 at 20:22

I didn't even think of that. Good catch, I'll add it to the list. – Stefan T Mar 20 '15 at 0:35

6

So you´re repeating lots of code within every exception-switch? Sounds like extracting a method would be god idea, doesn´t it?

So your code comes down to this:

```
MyClass instance;
try { instance = ... }
catch(Exception1 e) { Reset(instance); }
catch(Exception2 e) { Reset(instance); }
catch(Exception) { throw; }

void Reset(MyClass instance) { /* reset the state of the instance */
```

I wonder why no-one noticed that code-duplication.

From C#6 you furthermore have the exception-filters as already mentioned by others. So you can modify the code above to this:

```
try { ... }
catch(Exception e) when(e is Exception1 || e is Exception2)
{
```

```
    Reset(instance);
}
```

edited May 23 '17 at 11:47

Community ♦
1 1

answered Sep 22 '16 at 11:40

HimBromBeere
24.4k 4 33 62

2 *"I wonder why no-one noticed that code-duplication."* - uh, what? The *entire point of the question* is to eliminate the code duplication. – Mark Amery Oct 23 '17 at 22:03

Just call the try and catch twice.

-13

```
try
{
    WebId = new Guid(queryString["web"]);
}
catch (FormatException)
{
    WebId = Guid.Empty;
}
try
{
    WebId = new Guid(queryString["web"]);
}
catch (OverflowException)
{
    WebId = Guid.Empty;
}
```

It is just that Simple!!

answered Jul 29 '16 at 4:42

scholar guy
1 7

1 um. this is defeating the purpose of the question. He asks this question to get rid of duplicate code. this answer adds more duplicate code. – James Esh Feb 23 '17 at 21:40

10

Update 2015-12-15: See https://stackoverflow.com/a/22864936/1718702 for C#6. It's a cleaner and now standard in the language.

Geared for people that want a more elegant solution to catch once and filter exceptions, I use an extension method as demonstrated below.

I already had this extension in my library, originally written for other purposes, but it worked just perfectly for `type` checking on exceptions. Plus, imho, it looks cleaner than a bunch of `||` statements. Also, unlike the accepted answer, I prefer explicit exception handling so `ex is ...` had undesireable behaviour as derrived classes are assignable to there parent types).

Usage

```
if (ex.GetType().IsAnyOf(
    typeof(FormatException),
    typeof(ArgumentException)))
{
    // Handle
}
else
    throw;
```

IsAnyOf.cs Extension (See Full Error Handling Example for Dependancies)

```
namespace Common.FluentValidation
{
    public static partial class Validate
    {
        /// <summary>
        /// Validates the passed in parameter matches at least one o
comparisons.
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="p_parameter">Parameter to validate.</param>
        /// <param name="p_comparisons">Values to compare against.</
        /// <returns>True if a match is found.</returns>
        /// <exception cref="ArgumentNullException"></exception>
        public static bool IsAnyOf<T>(this T p_parameter, params T[]
        {
            // Validate
            p_parameter
                .CannotBeNull("p_parameter");
            p_comparisons
                .CannotBeNullOrEmpty("p_comparisons");

            // Test for any match
            foreach (var item in p_comparisons)
                if (p_parameter.Equals(item))
                    return true;

            // Return no matches found
            return false;
        }
    }
}
```

Full Error Handling Example (Copy-Paste to new Console app)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Common.FluentValidation;

namespace IsAnyOfExceptionHandlerSample
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            // High Level Error Handler (Log and Crash App)
            try
            {
                Foo();
            }
            catch (OutOfMemoryException ex)
            {
                Console.WriteLine("FATAL ERROR! System Crashing. " +
                Console.ReadKey();
            }
        }

        static void Foo()
        {
            // Init
            List<Action<string>> TestActions = new List<Action<strin
            {
                (key) => { throw new FormatException(); },
                (key) => { throw new ArgumentException(); },
                (key) => { throw new KeyNotFoundException();},
                (key) => { throw new OutOfMemoryException(); },
            };

            // Run
            foreach (var FooAction in TestActions)
            {
                // Mid-Level Error Handler (Appends Data for Log)
                try
                {
                    // Init
                    var SomeKeyPassedToFoo = "FooParam";

                    // Low-Level Handler (Handle/Log and Keep going)
                    try
                    {
                        FooAction(SomeKeyPassedToFoo);
                    }
                    catch (Exception ex)
                    {
                        if (ex.GetType().IsAnyOf(
                            typeof(FormatException),
                            typeof(ArgumentException)))
```

```
                        {
                            // Handle
                            Console.WriteLine("ex was {0}", ex.GetTy
                            Console.ReadKey();
                        }
                        else
                        {
                            // Add some Debug info
                            ex.Data.Add("SomeKeyPassedToFoo",
SomeKeyPassedToFoo.ToString());
                            throw;
                        }
                    }
                }
                catch (KeyNotFoundException ex)
                {
                    // Handle differently
                    Console.WriteLine(ex.Message);

                    int Count = 0;
                    if (!Validate.IsAnyNull(ex, ex.Data, ex.Data.Key
                        foreach (var Key in ex.Data.Keys)
                            Console.WriteLine(
                                "[{0}][\"{1}\" = {2}]",
                                Count, Key, ex.Data[Key]);

                    Console.ReadKey();
                }
            }
        }
    }
}

namespace Common.FluentValidation
{
    public static partial class Validate
    {
        /// <summary>
        /// Validates the passed in parameter matches at least one o
comparisons.
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="p_parameter">Parameter to validate.</param>
        /// <param name="p_comparisons">Values to compare against.</
        /// <returns>True if a match is found.</returns>
        /// <exception cref="ArgumentNullException"></exception>
        public static bool IsAnyOf<T>(this T p_parameter, params T[]
```

```
        {
            // Validate
            p_parameter
                .CannotBeNull("p_parameter");
            p_comparisons
                .CannotBeNullOrEmpty("p_comparisons");

            // Test for any match
            foreach (var item in p_comparisons)
                if (p_parameter.Equals(item))
                    return true;

            // Return no matches found
            return false;
        }

        /// <summary>
        /// Validates if any passed in parameter is equal to null.
        /// </summary>
        /// <param name="p_parameters">Parameters to test for Null.</
        /// <returns>True if one or more parameters are null.</retur
        public static bool IsAnyNull(params object[] p_parameters)
        {
            p_parameters
                .CannotBeNullOrEmpty("p_parameters");

            foreach (var item in p_parameters)
                if (item == null)
                    return true;

            return false;
        }
    }
}

namespace Common.FluentValidation
{
    public static partial class Validate
    {
        /// <summary>
        /// Validates the passed in parameter is not null, throwing a
message if the test fails.
        /// </summary>
        /// <param name="p_parameter">Parameter to validate.</param>
        /// <param name="p_name">Name of tested parameter to assist t
</param>
        /// <exception cref="ArgumentNullException"></exception>
```

```
        public static void CannotBeNull(this object p_parameter, str
        {
            if (p_parameter == null)
                throw
                    new
                        ArgumentNullException(
                        string.Format("Parameter \"{0}\" cannot be n
                        p_name), default(Exception));
        }
    }
}

namespace Common.FluentValidation
{
    public static partial class Validate
    {
        /// <summary>
        /// Validates the passed in parameter is not null or an empt
throwing a detailed exception message if the test fails.
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="p_parameter">Parameter to validate.</param>
        /// <param name="p_name">Name of tested parameter to assist
</param>
        /// <exception cref="ArgumentNullException"></exception>
        /// <exception cref="ArgumentOutOfRangeException"></exceptio
        public static void CannotBeNullOrEmpty<T>(this ICollection<T
string p_name)
        {
            if (p_parameter == null)
                throw new ArgumentNullException("Collection cannot b
null.\r\nParameter_Name: " + p_name, default(Exception));

            if (p_parameter.Count <= 0)
                throw new ArgumentOutOfRangeException("Collection ca
empty.\r\nParameter_Name: " + p_name, default(Exception));
        }

        /// <summary>
        /// Validates the passed in parameter is not null or empty,
exception message if the test fails.
        /// </summary>
        /// <param name="p_parameter">Parameter to validate.</param>
        /// <param name="p_name">Name of tested parameter to assist
</param>
        /// <exception cref="ArgumentException"></exception>
        public static void CannotBeNullOrEmpty(this string p_paramet
```

```
        {
            if (string.IsNullOrEmpty(p_parameter))
                throw new ArgumentException("String cannot be null o
empty.\r\nParameter_Name: " + p_name, default(Exception));
        }
    }
}
```

Two Sample NUnit Unit Tests

Matching behaviour for `Exception` types is exact (ie. A child IS NOT a match for any of its parent types).

```
using System;
using System.Collections.Generic;
using Common.FluentValidation;
using NUnit.Framework;

namespace UnitTests.Common.Fluent_Validations
{
    [TestFixture]
    public class IsAnyOf_Tests
    {
        [Test, ExpectedException(typeof(ArgumentNullException))]
        public void
IsAnyOf_ArgumentNullException_ShouldNotMatch_ArgumentException_Test(
        {
            Action TestMethod = () => { throw new ArgumentNullExcept

            try
            {
                TestMethod();
            }
            catch (Exception ex)
            {
                if (ex.GetType().IsAnyOf(
                    typeof(ArgumentException), /*Note: ArgumentNullE
from ArgumentException*/
                    typeof(FormatException),
                    typeof(KeyNotFoundException)))
                {
                    // Handle expected Exceptions
                    return;
                }
```

```
                //else throw original
                throw;
            }
        }

        [Test, ExpectedException(typeof(OutOfMemoryException))]
        public void IsAnyOf_OutOfMemoryException_ShouldMatch_OutOfMe
        {
            Action TestMethod = () => { throw new OutOfMemoryExcepti

            try
            {
                TestMethod();
            }
            catch (Exception ex)
            {
                if (ex.GetType().IsAnyOf(
                    typeof(OutOfMemoryException),
                    typeof(StackOverflowException)))
                    throw;

                /*else... Handle other exception types, typically by
            }
        }
    }
}
```

edited May 23 '17 at 10:31

Community ♦

1 1

answered Nov 27 '13 at 21:53

HodlDwon

763 1 9 23

1 Enhancing the language is *not* "more elegent". In many places this actually created a maintenance hell. Years later, many programmers are not proud of what monster they created. It's not what you are used to read. It may cause a "huh?" effect, or even severe "WTFs". It's confusing, sometimes. The only thing it does is making the code much harder to grasp for those who need to deal with it later in maintenance - only because a single programmer tried to be "clever". Over the years, i

learned that those "clever" solutions are seldomly also the good ones. – Kaii Oct 9 '14 at 21:02

1 or in a few words: stick with the possibilites the language natively provides. don't try to override the semantics of a language, only because *you* don't like them. Your collegues (and possibly future-me) will thank you, honestly. – Kaii Oct 9 '14 at 21:07

Also note, your solution only approximates the semantics of the C# 6 `when` , as would *any* version of `catch (Exception ex) {if (...) {/*handle*/} throw;}` . The real value of `when` is that the filter runs *before the exception is caught*, thus avoiding the expense/stack corruption of a re-throw. It takes advantage of a CLR feature that was only previously accessible to VB and MSIL. – Marc L. Jun 27 '16 at 14:55

More elegant? This example is so large for such a simple problem and the code is so awful-looking that it wasn't even worth giving a look. Please don't make this code someone else's problem on an actual project. – KthProg Dec 28 '16 at 16:02

your entire `IsAnyOf` method can be rewritten as simply `p_comparisons.Contains(p_parameter)` – maksymiuk Jul 27 '17 at 16:04

30

If you don't want to use an `if` statement within the `catch` scopes, ***in `C# 6.0` you can use `Exception Filters` syntax*** which was already supported by the CLR in previews versions but existed only in `VB.NET` / `MSIL` :

```
try
{
    WebId = new Guid(queryString["web"]);
}
catch (Exception exception) when (exception is FormatException || ex
OverflowException)
{
    WebId = Guid.Empty;
}
```

This code will catch the `Exception` only when it's a `InvalidDataException` or `ArgumentNullException`.

Actually, you can put basically any condition inside that `when` clause:

```
static int a = 8;

...

catch (Exception exception) when (exception is InvalidDataException {
{
    Console.WriteLine("Catch");
}
```

Note that as opposed to an `if` statement inside the `catch`'s scope, `Exception Filters` cannot throw `Exceptions`, and when they do, or when the condition is not `true`, the next `catch` condition will be evaluated instead:

```
static int a = 7;

static int b = 0;

...

try
{
    throw new InvalidDataException();
}
catch (Exception exception) when (exception is InvalidDataException {
{
    Console.WriteLine("Catch");
}
catch (Exception exception) when (exception is InvalidDataException
ArgumentException)
{
    Console.WriteLine("General catch");
}
```

> Output: General catch.

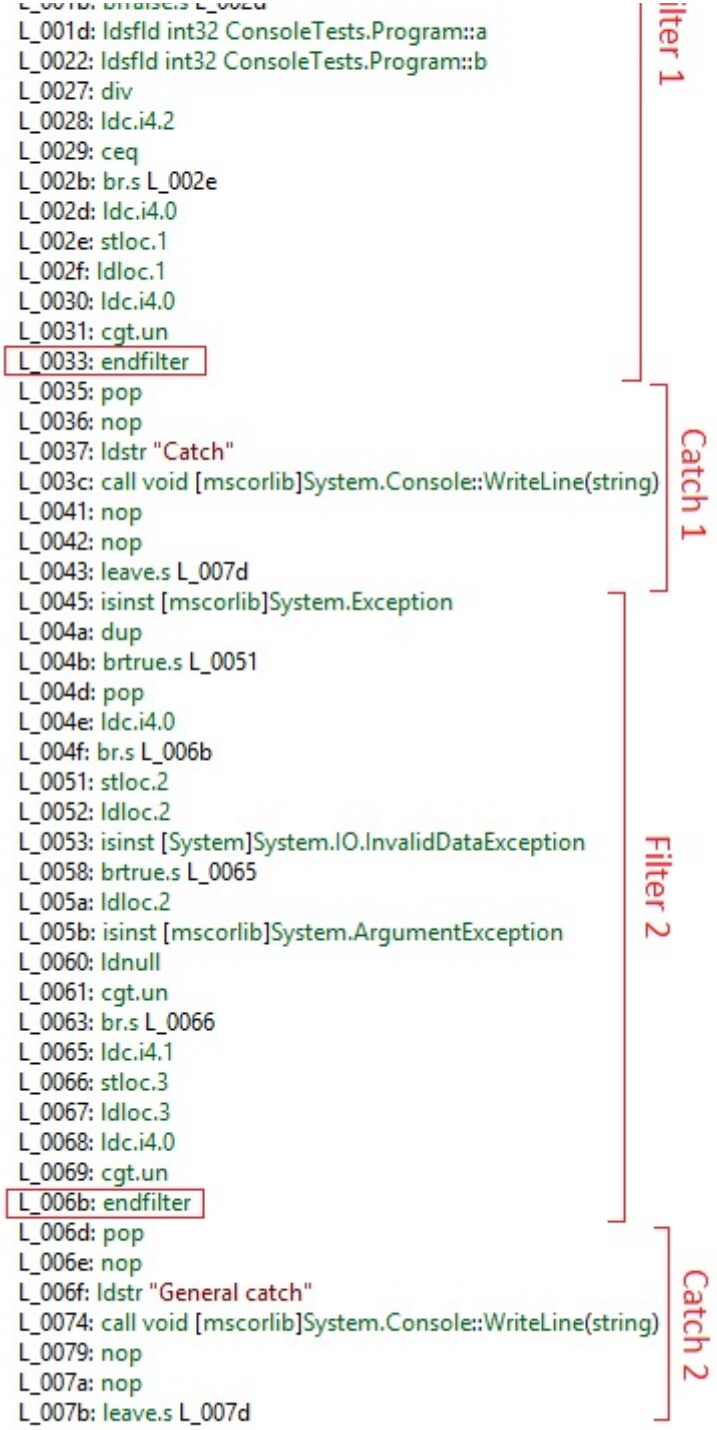
When there is more then one `true Exception Filter` - the first one will be accepted:

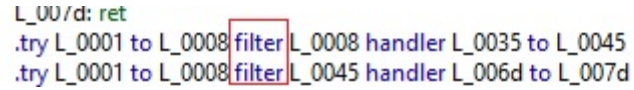
```
static int a = 8;

static int b = 4;

...

try
{
    throw new InvalidDataException();
}
catch (Exception exception) when (exception is InvalidDataException {
{
    Console.WriteLine("Catch");
}
catch (Exception exception) when (exception is InvalidDataException
ArgumentException)
{
    Console.WriteLine("General catch");
}
```

> Output: Catch.

And as you can see in the `MSIL` the code is not translated to `if` statements, but to `Filters` , and `Exceptions` cannot be throw from within the areas marked with `Filter 1` and `Filter 2` but the filter throwing the `Exception` will fail instead, also the last comparison value pushed to the stack before the `endfilter` command will determine the success/failure of the filter (`Catch 1` ***XOR*** `Catch 2` will execute accordingly):

```
L_0008: isinst [mscorlib]System.Exception
L_000d: dup
L_000e: brtrue.s L_0014
L_0010: pop
L_0011: ldc.i4.0
L_0012: br.s L_0033
L_0014: stloc.0
L_0015: ldloc.0
L_0016: isinst [System]System.IO.InvalidDataException
L_001b: brfalse.s L_002d
```


L_001d: ldsfld int32 ConsoleTests.Program::a
L_0022: ldsfld int32 ConsoleTests.Program::b
L_0027: div
L_0028: ldc.i4.2
L_0029: ceq
L_002b: br.s L_002e
L_002d: ldc.i4.0
L_002e: stloc.1
L_002f: ldloc.1
L_0030: ldc.i4.0
L_0031: cgt.un
L_0033: endfilter
L_0035: pop
L_0036: nop
L_0037: ldstr "Catch"
L_003c: call void [mscorlib]System.Console::WriteLine(string)
L_0041: nop
L_0042: nop
L_0043: leave.s L_007d
L_0045: isinst [mscorlib]System.Exception
L_004a: dup
L_004b: brtrue.s L_0051
L_004d: pop
L_004e: ldc.i4.0
L_004f: br.s L_006b
L_0051: stloc.2
L_0052: ldloc.2
L_0053: isinst [System]System.IO.InvalidDataException
L_0058: brtrue.s L_0065
L_005a: ldloc.2
L_005b: isinst [mscorlib]System.ArgumentException
L_0060: ldnull
L_0061: cgt.un
L_0063: br.s L_0066
L_0065: ldc.i4.1
L_0066: stloc.3
L_0067: ldloc.3
L_0068: ldc.i4.0
L_0069: cgt.un
L_006b: endfilter
L_006d: pop
L_006e: nop
L_006f: ldstr "General catch"
L_0074: call void [mscorlib]System.Console::WriteLine(string)
L_0079: nop
L_007a: nop
L_007b: leave.s L_007d
ilter 1
Catch 1
Filter 2
Catch 2



Also, specifically `Guid` *has the* [`Guid.TryParse`] *method.*

edited Oct 7 '15 at 18:37

answered Oct 7 '15 at 17:31

Tamir Vered
8,814 4 38 51

+1 for showing multiple when filters and providing an explanation of what happens when multiple filters are used. – steven87vt Oct 9 '18 at 13:59

18

in C# 6 the recommended approach is to use Exception Filters, here is an example:

```
try
{
     throw new OverflowException();
}
catch(Exception e ) when ((e is DivideByZeroException) || (e is Ove
{
      // this will execute iff e is DividedByZeroEx or OverflowEx
      Console.WriteLine("E");
}
```

edited Oct 7 '15 at 7:53

answered Oct 4 '15 at 7:51

SHM

1,071 12 29

294

+100

As others have pointed out, you can have an `if` statement inside your catch block to determine what is going on. C#6 supports Exception Filters, so the following will work:

```
try { … }
catch (Exception e) when (MyFilter(e))
{
    …
}
```

The `MyFilter` method could then look something like this:

```
private bool MyFilter(Exception e)
{
  return e is ArgumentNullException || e is FormatException;
}
```

Alternatively, this can be all done inline (the right hand side of the when statement just has to be a boolean expression).

```
try { … }
catch (Exception e) when (e is ArgumentNullException || e is FormatE
{
    …
}
```

This is different from using an `if` statement from within the `catch` block, using exception filters **will not** unwind the stack.

You can download Visual Studio 2015 to check this out.

If you want to continue using Visual Studio 2013, you can install the following nuget package:

> Install-Package Microsoft.Net.Compilers

[At time of writing, this will include support for C# 6.]

> Referencing this package will cause the project to be built using the specific version of the C# and Visual Basic compilers contained in the package, as opposed to any system installed version.

edited Oct 6 '15 at 18:11

answered Apr 4 '14 at 13:59

Joe
4,798 1 23 32

3 Patiently waiting for the official release of 6... I'd like to see this get the checky when that happens. – RubberDuck Feb 14 '15 at 13:41

@RubberDuck I am dying for the null propagation operator from C# 6. Trying to convince the rest of my team that the risk of an unstable language/compiler is worth it. Lots of minor improvements with huge impact. As for getting marked as answer, not important, as long as people realise this will/is possible, I'm happy. – Joe Feb 14 '15 at 19:52

Right?! I'll be taking a good look at my code base in the near future. =) I know the check isn't important, but given the accepted answer will soon be outdated, I'm hoping OP comes back to check this to give it the proper visibility. – RubberDuck Feb 14 '15 at 20:02

That's partially why I haven't awarded it yet @Joe. I want this to be visible. You may want to add an example of an inline filter for clarity though. – RubberDuck Oct 6 '15 at 16:15

-22

In c# 6.0,Exception Filters is improvements for exception handling

```
try
{
    DoSomeHttpRequest();
}
catch (System.Web.HttpException e)
{
    switch (e.GetHttpCode())
    {
        case 400:
            WriteLine("Bad Request");
        case 500:
            WriteLine("Internal Server Error");
        default:
            WriteLine("Generic Error");
    }
}
```

answered May 20 '15 at 7:48

Kashif
1,367 3 13 20

12 This example does not show any use of exception filters. – Stijn May 27 '15 at 13:37

This is standard way to filter exception in c#6.0 – Kashif May 27 '15 at 13:42

4 Take a look again at what exactly exception filters are. You are not using an exception filter in your example. There's a proper example in this answer posted a year before yours. – Stijn May 27 '15 at 13:44

5 An example of exception filtering would be `catch (HttpException e) when e.GetHttpCode() == 400 { WriteLine("Bad Request"; }` – saluce Nov 13 '15 at 16:29

2 This does not show a way to filter exceptions :/ – fergdev Mar 1 '16 at 2:51

18

The accepted answer seems acceptable, except that CodeAnalysis/FxCop will complain about the fact that it's catching a general exception type.

Also, it seems the "is" operator might degrade performance slightly.

CA1800: Do not cast unnecessarily says to "consider testing the result of the 'as' operator instead", but if you do that, you'll be writing more code than if you catch each exception separately.

Anyhow, here's what I would do:

```
bool exThrown = false;

try
{
    // Something
}
catch (FormatException) {
    exThrown = true;
}
catch (OverflowException) {
    exThrown = true;
}

if (exThrown)
{
    // Something else
}
```

edited Apr 9 '15 at 14:09

Peter Mortensen
13.9k 19 87 113

answered Jul 30 '10 at 17:09

Matt
874 8 8

19 But be aware that you can't rethrow the exception without losing the stack trace if you do it like this. (See Michael Stum's comment to the accepted

answer) – René Dec 9 '10 at 11:27

2 This pattern can be improved by storing the exception(please excuse the poor formatting -- I can't figure out how to put code in comments): Exception ex = null; try { // something } catch(FormatException e){ ex = e; } catch(OverflowException e){ ex = e; } if(ex != null) { // something else and deal with ex } – Jesse Weigert Mar 16 '12 at 20:03

3 @JesseWeigert: 1. You can use backticks to give a piece of text a mono-spaced font and light grey background. 2. You still won't be able to rethrow the original exception **including the stacktrace**. – Oliver Nov 22 '12 at 17:19

2 @CleverNeologism although it may be true that using the `is` operator may have a slight negative impact on performance, it's also true that an exception handler isn't the place to be overly concerned about optimizing performance. If your app is spending so much time in exception handlers that performance optimization there would make a real difference in app performance, then there are other code issues to take a hard look at. Having said that, I still don't like this solution because you lose the stack trace and because the cleanup is contextually removed from the catch statement. – Craig Sep 15 '15 at 7:22

2 The only time the `is` operator degrades performance is if you later perform an `as` operation (hence they qualify the rule with *unnecessarily*). If all you are doing is testing the cast without actually needing to perform the cast, then the `is` operator is exactly what you want to use. – saluce Nov 13 '15 at 16:25

2

Note that I did find one way to do it, but this looks more like material for The Daily WTF:

```
catch (Exception ex)
{
    switch (ex.GetType().Name)
    {
        case "System.FormatException":
        case "System.OverflowException":
            WebId = Guid.Empty;
            break;
```

```
            default:
                throw;
        }
    }
```

edited Apr 9 '15 at 14:07

community wiki
3 revs, 3 users 61%
Michael Stum

7 -1 vote, +5 WTF :-) This should not have been marked as an answer, but it is he-larious. – Aaron Sep 25 '08 at 21:23

1 Doesn't matter how simply we could do it. But he didn't sit idle and came up with his view to solve it. Really appreciate. – Maxymus Jan 21 '16 at 10:27

2 Don't actually do this though, use Exception Filters in C# 6 or any of the other answers - I put this here specifically as "This is one way, but it's bad and I want to do something better". – Michael Stum ♦ Jan 21 '16 at 19:16

WHY is this bad? I was puzzled you could not use the exception in a switch statement directly. – MKesper Jun 8 '16 at 14:02

2 @MKesper I see a few reasons it's bad. It requires writing the fully-qualified class names as string literals, which is vulnerable to typos that the compiler can't save you from. (This is significant since in many shops error cases are less well-tested and so trivial mistakes in them are more likely to be missed.) It will also fail to match an exception which is a *subclass* of one of the specified cases. And, due to being strings, the cases will be missed by tools like VS's "Find All References" - pertinent if you want to add a cleanup step everywhere a particular exception is caught. – Mark Amery Oct 23 '17 at 22:18

71

If you can upgrade your application to C# 6 you are lucky. The new C# version has implemented Exception filters. So you can write this:

```
catch (Exception ex) when (ex is FormatException || ex is OverflowEx
    WebId = Guid.Empty;
}
```

Some people think this code is the same as

```
catch (Exception ex) {
    if (ex is FormatException || ex is OverflowException) {
        WebId = Guid.Empty;
    }
    throw;
}
```

But it´s not. Actually this is the only new feature in C# 6 that is not possible to emulate in prior versions. First, a re-throw means more overhead than skipping the catch. Second, it is not semantically equivalent. The new feature preserves the stack intact when you are debugging your code. Without this feature the crash dump is less useful or even useless.

See a discussion about this on CodePlex. And an example showing the difference.

edited Apr 1 '15 at 16:04

answered Apr 1 '15 at 12:29

Maniero

4,159 6 32 64

4 Throw without exception preserves the stack, but "throw ex" will overwrite it. – Ivan Apr 12 '17 at 13:18

11

```
catch (Exception ex)
{
    if (!(
        ex is FormatException ||
        ex is OverflowException))
    {
        throw;
    }
    Console.WriteLine("Hello");
}
```

edited Oct 30 '14 at 11:15

mantale
696 15 32

answered Jan 7 '09 at 8:07

Konstantin Spirin
11.9k 9 55 83

7

Since I felt like these answers just touched the surface, I attempted to dig a bit deeper.

So what we would really want to do is something that doesn't compile, say:

```
// Won't compile... damn
public static void Main()
{
    try
    {
        throw new ArgumentOutOfRangeException();
    }
    catch (ArgumentOutOfRangeException)
    catch (IndexOutOfRangeException)
    {
```

```
        // ... handle
    }
```

The reason we want this is because we don't want the exception handler to catch things that we need later on in the process. Sure, we can catch an Exception and check with an 'if' what to do, but let's be honest, we don't really want that. (FxCop, debugger issues, uglyness)

So why won't this code compile - and how can we hack it in such a way that it will?

If we look at the code, what we really would like to do is forward the call. However, according to the MS Partition II, IL exception handler blocks won't work like this, which in this case makes sense because that would imply that the 'exception' object can have different types.

Or to write it in code, we ask the compiler to do something like this (well it's not entirely correct, but it's the closest possible thing I guess):

```
// Won't compile... damn
try
{
    throw new ArgumentOutOfRangeException();
}
catch (ArgumentOutOfRangeException e) {
    goto theOtherHandler;
}
catch (IndexOutOfRangeException e) {
theOtherHandler:
    Console.WriteLine("Handle!");
}
```

The reason that this won't compile is quite obvious: what type and value would the '$exception' object have (which are here stored in the variables 'e')? The way we want the compiler to handle this is to note that the common base type of both exceptions is 'Exception', use that for a variable to contain both exceptions, and then handle

only the two exceptions that are caught. The way this is implemented in IL is as 'filter', which is available in VB.Net.

To make it work in C#, we need a temporary variable with the correct 'Exception' base type. To control the flow of the code, we can add some branches. Here goes:

```
    Exception ex;
    try
    {
        throw new ArgumentException(); // for demo purposes; won't b
        goto noCatch;
    }
    catch (ArgumentOutOfRangeException e) {
        ex = e;
    }
    catch (IndexOutOfRangeException e) {
        ex = e;
    }

    Console.WriteLine("Handle the exception 'ex' here :-)");
    // throw ex ?

noCatch:
    Console.WriteLine("We're done with the exception handling.");
```

The obvious disadvantages for this are that we cannot re-throw properly, and -well let's be honest- that it's quite the ugly solution. The uglyness can be fixed a bit by performing branch elimination, which makes the solution slightly better:

```
Exception ex = null;
try
{
    throw new ArgumentException();
}
catch (ArgumentOutOfRangeException e)
{
    ex = e;
}
catch (IndexOutOfRangeException e)
{
    ex = e;
```

```
}
if (ex != null)
{
    Console.WriteLine("Handle the exception here :-)");
}
```

That leaves just the 're-throw'. For this to work, we need to be able to perform the handling inside the 'catch' block - and the only way to make this work is by an catching 'Exception' object.

At this point, we can add a separate function that handles the different types of Exceptions using overload resolution, or to handle the Exception. Both have disadvantages. To start, here's the way to do it with a helper function:

```
private static bool Handle(Exception e)
{
    Console.WriteLine("Handle the exception here :-)");
    return true; // false will re-throw;
}

public static void Main()
{
    try
    {
        throw new OutOfMemoryException();
    }
    catch (ArgumentException e)
    {
        if (!Handle(e)) { throw; }
    }
    catch (IndexOutOfRangeException e)
    {
        if (!Handle(e)) { throw; }
    }

    Console.WriteLine("We're done with the exception handling.");
```

And the other solution is to catch the Exception object and handle it accordingly. The most literal translation for this, based on the context above is this:

```
try
{
    throw new ArgumentException();
}
catch (Exception e)
{
    Exception ex = (Exception)(e as ArgumentException) ?? (e as
IndexOutOfRangeException);
    if (ex != null)
    {
        Console.WriteLine("Handle the exception here :-)");
        // throw ?
    }
    else
    {
        throw;
    }
}
```

So to conclude:

- If we don't want to re-throw, we might consider catching the right exceptions, and storing them in a temporary.
- If the handler is simple, and we want to re-use code, the best solution is probably to introduce a helper function.
- If we want to re-throw, we have no choice but to put the code in a 'Exception' catch handler, which will break FxCop and your debugger's uncaught exceptions.

answered Oct 21 '14 at 9:44

atlaste
23.4k 2 41 59

17

This is a variant of Matt's answer (I feel that this is a bit cleaner)...use a method:

```
public void TryCatch(...)
{
    try
    {
       // something
       return;
    }
    catch (FormatException) {}
    catch (OverflowException) {}

    WebId = Guid.Empty;
}
```

Any other exceptions will be thrown and the code `WebId = Guid.Empty;` won't be hit. If you don't want other exceptions to crash your program, just add this AFTER the other two catches:

```
...
catch (Exception)
{
     // something, if anything
     return; // only need this if you follow the example I gave and p
method
}
```

edited Oct 11 '13 at 20:52

answered Aug 31 '12 at 20:51

bsara
6,021 3 22 44

-1 This will execute `WebId = Guid.Emtpy` in the case where no exception was thrown. – Sepster Oct 23 '12 at 13:41

4 @sepster I think the return statement after "// something" is implied here. I do not really like the solution, but this is a constructive variant in the discussion. +1 to undo your downvote :-) – toong Oct 23 '12 at 21:12

@Sepster toong is right, I assumed that if you wanted a return there, then

you would put one...I was trying to make my answer general enough to apply to all situations in case others with similar but not exact questions would benefit as well. However, for good measure, I've add a `return` to my answer. Thanks for the input. – bsara Oct 24 '12 at 17:27

11

Cautioned and Warned: Yet another kind, functional style.

What is in the link doesn't answer your question directly, but it's trivial to extend it to look like:

```
static void Main()
{
    Action body = () => { ...your code... };

    body.Catch<InvalidOperationException>()
        .Catch<BadCodeException>()
        .Catch<AnotherException>(ex => { ...handler... })();
}
```

(Basically provide another empty `Catch` overload which returns itself)

The bigger question to this is *why*. I do not think the cost outweighs the gain here :)

answered May 18 '13 at 11:28

nawfal
43.9k 36 258 304

1 One possible advantage of this approach is that there's a semantic difference between catching and rethrowing an exception versus not catching it; in some cases, code should act upon an exception *without* catching it. Such a thing is possible in vb.net, but not in C# *unless* one uses a wrapper written in vb.net and called from C#. – supercat Jul 17 '13 at 2:43

1 How does on act on an exception without catching it? I do not fully understand you. – nawfal Jul 17 '13 at 7:20

@nawful ... using vb filter - function filt(ex as exception) :LogEx(ex):return false ... then in the catch line: catch ex when filt(ex) – FastAl Oct 7 '15 at 18:15

1 @FastAl Isn´t this what exception-filters allow in C#6? – HimBromBeere Sep 22 '16 at 11:52

@HimBromBeere yep they're direct analogues – FastAl Sep 22 '16 at 14:31

13

How about

```
try
{
    WebId = Guid.Empty;
    WebId = new Guid(queryString["web"]);
}
catch (FormatException)
{
}
catch (OverflowException)
{
}
```

edited Mar 22 '13 at 15:58

Sepster

4,359 13 36

answered Sep 25 '08 at 20:57

Maurice

25.7k 5 43 61

That works only if the Catch-Code can be fully moved into the Try-Block. But imaging code where you make multiple manipulations to an object,

and one in the middle fails, and you want to "reset" the object. – Michael Stum ♦ Sep 25 '08 at 20:59

4 In that case I would add a reset function and call that from multiple catch blocks. – Maurice Sep 25 '08 at 21:04

16

@Micheal

Slightly revised version of your code:

```
catch (Exception ex)
{
   Type exType = ex.GetType();
   if (exType == typeof(System.FormatException) ||
       exType == typeof(System.OverflowException)
   {
       WebId = Guid.Empty;
   } else {
      throw;
   }
}
```

String comparisons are ugly and slow.

edited Sep 25 '08 at 21:16

Michael Stum ♦
118k 99 363 506

answered Sep 25 '08 at 21:01

FlySwat
116k 63 231 300

20 Why not just use the "is" keyword? – Chris Pietschmann Sep 25 '08 at 21:02

28 @Michael - If Microsoft introduced, say, StringTooLongException derived from FormatException then it is still a format exception, just a specific

one. It depends whether you want the semantics of 'catch this exact exception type' or 'catch exceptions that mean the format of the string was wrong'. – Greg Beech Sep 25 '08 at 21:19

6 @Michael - Also, note that "catch (FormatException ex) has the latter semantics, it will catch anything derived from FormatException. – Greg Beech Sep 25 '08 at 21:21

14 @Alex No. "throw" without "ex" carries the original exception, including original stack trace, up. Adding "ex" makes the stack trace reset, so you really get a different exception than the original. I'm sure someone else can explain it better than me. :) – Samantha Branham Sep 22 '10 at 22:04

13 -1: This code is extremely fragile - a library developer could expect to replace `throw new FormatException();` with `throw new NewlyDerivedFromFormatException();` without breaking code using the library, and it will hold true for all exception handling cases except where someone used `==` instead of `is` (or simply `catch (FormatException)`). – Sam Harwell Jul 17 '13 at 3:47

185

Not in C# unfortunately, as you'd need an exception filter to do it and C# doesn't expose that feature of MSIL. VB.NET does have this capability though, e.g.

```
Catch ex As Exception When TypeOf ex Is FormatException OrElse TypeO
OverflowException
```

What you could do is use an anonymous function to encapsulate your on-error code, and then call it in those specific catch blocks:

```
Action onError = () => WebId = Guid.Empty;
try
{
    // something
}
catch (FormatException)
{
```

```
    onError();
}
catch (OverflowException)
{
    onError();
}
```

answered Sep 25 '08 at 21:03

Greg Beech
102k 36 187 236

26 Interesting idea and another example that VB.net has some interesting advantages over C# sometimes – Michael Stum ♦ Sep 25 '08 at 21:19

43 @MichaelStum with **that** kind of syntax I would hardly call it interesting at all... *shudder* – MarioDS Oct 9 '14 at 12:15

17 Exception filters are coming in c# 6! Note the difference of using filters in favor of rethrowing roslyn.codeplex.com/discussions/541301 – Arne Deruwe Nov 26 '14 at 13:26

@ArneDeruwe Thank you for that link! I just learned one more important reason not to re-throw: `throw e;` destroys stacktrace *and* callstack, `throw;` destroys "only" callstack (rendering crash-dumps useless!) A **very** good reason to use neither if it can be avoided! – AnorZaken Dec 11 '14 at 14:53

If I'm not mistake you can also drop through a catch in VB like you can with case or (switch in c#) statements like so `Try Catch ex As ArgumentException Catch ex As NullReferenceException End Try` But unfortunately C# does not so we're left with a helper method or to catch generically and determine type. – David Carrigan Sep 17 '15 at 16:13

protected by durron597 Sep 23 '15 at 18:16

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?