Why aren't variables declared in "try" in scope in "catch" or "finally"?

Asked 16 years, 3 months ago Modified 6 years, 1 month ago Viewed 99k times

In C# and in Java (and possibly other languages as well), variables declared in a "try" block are not in scope in the corresponding "catch" or "finally" blocks. For example, the following code does not compile:

```
try {
  String s = "test";
  // (more code...)
}
catch {
  Console.Out.WriteLine(s);  //Java fans: think "System.out.println" here
instead
}
```

In this code, a compile-time error occurs on the reference to s in the catch block, because s is only in scope in the try block. (In Java, the compile error is "s cannot be resolved"; in C#, it's "The name 's' does not exist in the current context".)

The general solution to this issue seems to be to instead declare variables just before the try block, instead of within the try block:

```
String s;
try {
  s = "test";
  // (more code...)
}
catch {
  Console.Out.WriteLine(s);  //Java fans: think "System.out.println" here
instead
}
```

However, at least to me, (1) this feels like a clunky solution, and (2) it results in the variables having a larger scope than the programmer intended (the entire remainder of the method, instead of only in the context of the try-catch-finally).

My question is, what were/are the rationale(s) behind this language design decision (in Java, in C#, and/or in any other applicable languages)?

c# java exception scope language-design

Share

Improve this question

Follow

edited Sep 26, 2008 at 1:51

asked Sep 18, 2008 at 17:56

Jon Schneider

27k ● 24 ● 151 ● 180

28 Answers

Sorted by: Highest score (default)

191

Two things:

1. Generally, Java has just 2 levels of scope: global and function. But, try/catch is an exception (no pun intended). When an exception is thrown and the exception object gets a variable assigned to it, that object variable is only available within the "catch" section and is destroyed as soon as the catch completes.
2. (and more importantly). You can't know where in the try block the exception was thrown. It may have been before your variable was declared. Therefore it is impossible to say what variables will be available for the catch/finally clause. Consider the following case, where scoping is as you suggested:

```
try
{
    throw new ArgumentException("some operation that throws an exception");
    string s = "blah";
}
catch (e as ArgumentException)
{
    Console.Out.WriteLine(s);
}
```

This clearly is a problem - when you reach the exception handler, s will not have been declared. Given that catches are meant to handle exceptional circumstances and finallys *must* execute, being safe and declaring this a problem at compile time is far better than at runtime.

Share

Improve this answer

Follow

edited Oct 15, 2017 at 1:32

Timo Tijhof

10.3k ● 6 ● 37 ● 53

answered Sep 18, 2008 at 18:01

John Christensen

5,030 ● 1 ● 29 ● 26

56

How could you be sure, that you reached the declaration part in your catch block? What if the instantiation throws the exception?

Share

Improve this answer

Follow

edited Jul 5, 2009 at 17:49

answered Sep 18, 2008 at 17:58

Burkhard

14.7k ● 22 ● 90 ● 113

22

Traditionally, in C-style languages, what happens inside the curly braces stays inside the curly braces. I think that having the lifetime of a variable stretch across scopes like that would be unintuitive to most programmers. You can achieve what you want by enclosing the try/catch/finally blocks inside another level of braces. e.g.

```
... code ...
{
    string s = "test";
    try
    {
        // more code
    }
    catch(...)
    {
        Console.Out.WriteLine(s);
    }
}
```

EDIT: I guess every rule *does* have an exception. The following is valid C++:

```
int f() { return 0; }

void main()
{
    int y = 0;

    if (int x = f())
    {
        cout << x;
    }
    else
    {
        cout << x;
    }
}
```

The scope of x is the conditional, the then clause and the else clause.

Share

Improve this answer

Follow

edited Dec 9, 2008 at 1:54

answered Sep 18, 2008 at 18:03

Ferruccio

101k ● 38 ● 229 ● 303

11

Everyone else has brought up the basics -- what happens in a block stays in a block. But in the case of .NET, it may be helpful to examine what the compiler thinks is happening. Take, for example, the following try/catch code (note that the StreamReader is declared, correctly, outside the blocks):

```
static void TryCatchFinally()
{
    StreamReader sr = null;
    try
    {
        sr = new StreamReader(path);
        Console.WriteLine(sr.ReadToEnd());
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
    finally
    {
        if (sr != null)
        {
            sr.Close();
        }
    }
}
```

This will compile out to something similar to the following in MSIL:

```
.method private hidebysig static void  TryCatchFinallyDispose() cil managed
{
  // Code size       53 (0x35)
  .maxstack  2
  .locals init ([0] class [mscorlib]System.IO.StreamReader sr,
           [1] class [mscorlib]System.Exception ex)
  IL_0000:  ldnull
  IL_0001:  stloc.0
  .try
  {
    .try
    {
      IL_0002:  ldsfld     string UsingTest.Class1::path
      IL_0007:  newobj     instance void
[mscorlib]System.IO.StreamReader::.ctor(string)
      IL_000c:  stloc.0
      IL_000d:  ldloc.0
      IL_000e:  callvirt   instance string
[mscorlib]System.IO.TextReader::ReadToEnd()
      IL_0013:  call       void [mscorlib]System.Console::WriteLine(string)
      IL_0018:  leave.s    IL_0028
    }  // end .try
    catch [mscorlib]System.Exception
    {
      IL_001a:  stloc.1
      IL_001b:  ldloc.1
      IL_001c:  callvirt   instance string
[mscorlib]System.Exception::ToString()
      IL_0021:  call       void [mscorlib]System.Console::WriteLine(string)
      IL_0026:  leave.s    IL_0028
    }  // end handler
    IL_0028:  leave.s    IL_0034
  }  // end .try
  finally
  {
    IL_002a:  ldloc.0
```

```
    IL_002b:  brfalse.s  IL_0033
    IL_002d:  ldloc.0
    IL_002e:  callvirt   instance void [mscorlib]System.IDisposable::Dispose()
    IL_0033:  endfinally
  }  // end handler
  IL_0034:  ret
} // end of method Class1::TryCatchFinallyDispose
```

What do we see? MSIL respects the blocks -- they're intrinsically part of the underlying code generated when you compile your C#. The scope isn't just hard-set in the C# spec, it's in the CLR and CLS spec as well.

The scope protects you, but you do occasionally have to work around it. Over time, you get used to it, and it begins to feel natural. Like everyone else said, what happens in a block stays in that block. You want to share something? You have to go outside the blocks ...

Share Improve this answer Follow

answered Sep 18, 2008 at 21:48

John Rudy
37.8k ● 14 ● 66 ● 101

9

In C++ at any rate, the scope of an automatic variable is limited by the curly braces that surround it. Why would anyone expect this to be different by plunking down a try keyword outside the curly braces?

Share Improve this answer Follow

answered Sep 18, 2008 at 17:59

ravenspoint
20.3k ● 7 ● 62 ● 111

1 Agreed; "}" means end-of-scope. However, try-catch-finally is unusual in that after a try block, you *must* have a catch and/or finally block; thus, an exception to the normal rule where the scope of a try block carried into the associated catch/finally might seem acceptable?
– Jon Schneider Sep 18, 2008 at 20:30

7

Like ravenspoint pointed out, everyone expects variables to be local to the block they are defined in. `try` introduces a block and so does `catch`.

If you want variables local to both `try` and `catch`, try enclosing both in a block:

```
// here is some code
{
    string s;
    try
    {
```

```
        throw new Exception(":(")
    }
    catch (Exception e)
    {
        Debug.WriteLine(s);
    }
}
```

Share Improve this answer Follow

answered Sep 18, 2008 at 18:03

Daren Thomas
70.2k ● 42 ● 155 ● 205

5

The simple answer is that C and most of the languages that have inherited its syntax are block scoped. That means that if a variable is defined in one block, i.e., inside { }, that is its scope.

The exception, by the way, is JavaScript, which has a similar syntax, but is function scoped. In JavaScript, a variable declared in a try block is in scope in the catch block, and everywhere else in its containing function.

Share Improve this answer Follow

answered Sep 18, 2008 at 18:06

dgvid
26.6k ● 5 ● 42 ● 57

5

According to the section titled "How to Throw and Catch Exceptions" in Lesson 2 of **MCTS Self-Paced Training Kit (Exam 70-536): Microsoft® .NET Framework 2.0—Application Development Foundation**, the reason is that the exception may have occurred before variable declarations in the try block (as others have noted already).

Quote from page 25:

"Notice that the StreamReader declaration was moved outside the Try block in the preceding example. This is necessary because the Finally block cannot access variables that are declared within the Try block. ***This makes sense because depending on where an exception occurred, variable declarations within the Try block might not yet have been executed***."

Share Improve this answer Follow

answered Sep 19, 2008 at 7:31

hurst
1,740 ● 12 ● 8

@burkhard has the question as to why answered properly, but as a note I wanted to add, while your recommended solution example is good 99.9999+% of time, it is not

4

good practice, it is far safer to either check for null before using something instantiate within the try block, or initialize the variable to something instead of just declaring it before the try block. For example:

```
string s = String.Empty;
try
{
    //do work
}
catch
{
   //safely access s
   Console.WriteLine(s);
}
```

Or:

```
string s;
try
{
    //do work
}
catch
{
   if (!String.IsNullOrEmpty(s))
   {
       //safely access s
       Console.WriteLine(s);
   }
}
```

This should provide scalability in the workaround, so that even when what you're doing in the try block is more complex than assigning a string, you should be able to safely access the data from your catch block.

Share Improve this answer Follow

answered Sep 18, 2008 at 18:12

Timothy Carter

15.8k ● 7 ● 46 ● 62

4

The answer, as everyone has pointed out, is pretty much "that's how blocks are defined".

There are some proposals to make the code prettier. See ARM

```
try (FileReader in = makeReader(), FileWriter out = makeWriter()) {
      // code using in and out
} catch(IOException e) {
      // ...
}
```

Closures are supposed to address this as well.

```
with(FileReader in : makeReader()) with(FileWriter out : makeWriter()) {
    // code using in and out
}
```

UPDATE: ARM is implemented in Java 7.
http://download.java.net/jdk7/docs/technotes/guides/language/try-with-resources.html

Share
Improve this answer
Follow

edited Jun 7, 2011 at 16:55

answered Sep 18, 2008 at 18:37

ykaganovich
14.9k ● 8 ● 62 ● 98

2

You solution is exactly what you should do. You can't be sure that your declaration was even reached in the try block, which would result in another exception in the catch block.

It simply must work as separate scopes.

```
try
    dim i as integer = 10 / 0 ''// Throw an exception
    dim s as string = "hi"
catch (e)
    console.writeln(s) ''// Would throw another exception, if this was allowed
to compile
end try
```

Share Improve this answer Follow

answered Sep 18, 2008 at 18:00

EndangeredMassa
17.5k ● 8 ● 56 ● 80

2

The variables are block level and restricted to that Try or Catch block. Similar to defining a variable in an if statement. Think of this situation.

```
try {
    fileOpen("no real file Name");
    String s = "GO TROJANS";
} catch (Exception) {
    print(s);
}
```

The String would never be declared, so it can't be depended upon.

Share Improve this answer Follow

answered Sep 18, 2008 at 18:02

jW.
9,300 ● 12 ● 48 ● 50

2

In the specific example you've given, initialising s can't throw an exception. So you'd think that maybe its scope could be extended.

But in general, initialiser expressions can throw exceptions. It wouldn't make sense for a variable whose initialiser threw an exception (or which was declared after another variable where that happened) to be in scope for catch/finally.

Also, code readability would suffer. The rule in C (and languages which follow it, including C++, Java and C#) is simple: variable scopes follow blocks.

If you want a variable to be in scope for try/catch/finally but nowhere else, then wrap the whole thing in another set of braces (a bare block) and declare the variable before the try.

Share Improve this answer Follow

answered Sep 18, 2008 at 18:02

Steve Jessop
279k ● 40 ● 469 ● 709

2

Because the try block and the catch block are 2 different blocks.

In the following code, would you expect s defined in block A be visible in block B?

```
{ // block A
  string s = "dude";
}

{ // block B
  Console.Out.WriteLine(s); // or printf or whatever
}
```

Share Improve this answer Follow

answered Sep 18, 2008 at 18:05

Francesca
21.6k ● 5 ● 52 ● 91

2

When you declare a local variable it is placed on the stack (for some types the entire value of the object will be on the stack, for other types only a reference will be on the stack). When there is an exception inside a try block, the local variables within the block are freed, which means the stack is "unwound" back to the state it was at at the beginning of the try block. This is by design. It's how the try / catch is able to back out of all of the function calls within the block and puts your system back into a functional

state. Without this mechanism you could never be sure of the state of anything when an exception occurs.

Having your error handling code rely on externally declared variables which have their values changed inside the try block seems like bad design to me. What you are doing is essentially leaking resources intentionally in order to gain information (in this particular case it's not so bad because you are only leaking information, but imagine if it were some other resource? you're just making life harder on yourself in the future). I would suggest breaking up your try blocks into smaller chunks if you require more granularity in error handling.

Share Improve this answer Follow

answered Sep 18, 2008 at 18:12

Wedge

19.8k ● 7 ● 50 ● 71

2

When you have a try catch, you should at the most part know that errors that it might throw. Theese Exception classes normaly tell everything you need about the exception. If not, you should make you're own exception classes and pass that information along. That way, you will never need to get the variables from inside the try block, because the Exception is self explainatory. So if you need to do this alot, think about you're design, and try to think if there is some other way, that you can either predict exceptions comming, or use the information comming from the exceptions, and then maybe rethrow your own exception with more information.

Share Improve this answer Follow

answered Sep 18, 2008 at 18:19

Jesper Blad Jensen

2,751 ● 17 ● 16

2

As has been pointed out by other users, the curly braces define scope in pretty much every C style language that I know of.

If it's a simple variable, then why do you care how long it will be in scope? It's not that big a deal.

in C#, if it is a complex variable, you will want to implement IDisposable. You can then either use try/catch/finally and call obj.Dispose() in the finally block. Or you can use the using keyword, which will automatically call the Dispose at the end of the code section.

Share Improve this answer Follow

answered Sep 18, 2008 at 18:22

Charles Graham

24.8k ● 14 ● 47 ● 56

In Python they are visible in the catch/finally blocks if the line declaring them didn't throw.

2

Share Improve this answer Follow

answered Sep 19, 2008 at 6:00

adal

What if the exception is thrown in some code which is above the declaration of the variable. Which means, the declaration itself was not happend in this case.

2

```
try {

       //doSomeWork // Exception is thrown in this line.
       String s;
       //doRestOfTheWork

} catch (Exception) {
        //Use s;//Problem here
} finally {
        //Use s;//Problem here
}
```

Share Improve this answer Follow

answered Dec 29, 2010 at 10:23

Ravi

301 ● 1 ● 2 ● 8

While in your example it is weird that it does not work, take this similar one:

2

```
    try
    {
         //Code 1
         String s = "1|2";
         //Code 2
    }
    catch
    {
         Console.WriteLine(s.Split('|')[1]);
    }
```

This would cause the catch to throw a null reference exception if Code 1 broke. Now while the semantics of try/catch are pretty well understood, this would be an annoying corner case, since s is defined with an initial value, so it should in theory never be null, but under shared semantics, it would be.

Again this could in theory be fixed by only allowing separated definitions (`String s; s = "1|2";`), or some other set of conditions, but it is generally easier to just say no.

Additionally, it allows the semantics of scope to be defined globally without exception, specifically, locals last as long as the `{}` they are defined in, in all cases. Minor point, but a point.

Finally, in order to do what you want, you can add a set of brackets around the try catch. Gives you the scope you want, although it does come at the cost of a little readability, but not too much.

```
{
     String s;
     try
     {
          s = "test";
          //More code
     }
     catch
     {
          Console.WriteLine(s);
     }
}
```

Share

Improve this answer

Follow

edited Jul 3, 2018 at 14:03

Lucas

17.4k ● 5 ● 47 ● 41

answered Sep 18, 2008 at 18:11

Guvante

19.2k ● 1 ● 37 ● 64

2

The C# Spec (15.2) states "The scope of a local variable or constant declared in a block ist the block."

(in your first example the try block is the block where "s" is declared)

Share

Improve this answer

Follow

edited Nov 2, 2018 at 6:09

Cristik

32.8k ● 25 ● 101 ● 139

answered Sep 18, 2008 at 18:14

tamberg

2,017 ● 16 ● 23

1

My thought would be that because something in the try block triggered the exception its namespace contents cannot be trusted - ie referencing the String 's' in the catch block could cause the throw of yet another exception.

Share Improve this answer Follow

answered Sep 18, 2008 at 17:58

jpbarto

39 ● 1

Well if it doesn't throw a compile error, and you could declare it for the rest of the method, then there would be no way to only declare it only within try scope. It's forcing you to be explicit as to where the variable is supposed to exists and doesn't make assumptions.

Share Improve this answer Follow

answered Sep 18, 2008 at 17:58

kemiller2002

115k • 28 • 199 • 253

Part of the reason they are not in the same scope is because at any point of the try block, you can have thrown the exception. If they were in the same scope, its a disaster in waiting, because depending on where the exception was thrown, it could be even more ambiguous.

At least when its declared outside of the try block, you know for sure what the variable at minimum could be when an exception is thrown; The value of the variable before the try block.

Share Improve this answer Follow

answered Sep 18, 2008 at 18:04

zxcv

7,621 • 8 • 36 • 30

If we ignore the scoping-block issue for a moment, the complier would have to work a lot harder in a situation that's not well defined. While this is not impossible, the scoping error also forces you, the author of the code, to realise the implication of the code you write (that the string s may be null in the catch block). If your code was legal, in the case of an OutOfMemory exception, s isn't even guaranteed to be allocated a memory slot:

```
// won't compile!
try
{
    VeryLargeArray v = new VeryLargeArray(TOO_BIG_CONSTANT); // throws
OutOfMemoryException
    string s = "Help";
}
catch
{
    Console.WriteLine(s); // whoops!
}
```

The CLR (and therefore compiler) also force you to initialize variables before they are used. In the catch block presented it can't guarantee this.

So we end up with the compiler having to do a lot of work, which in practice doesn't provide much benefit and would probably confuse people and lead them to ask why try/catch works differently.

In addition to consistency, by not allowing anything fancy and adhering to the already established scoping semantics used throughout the language, the compiler and CLR are able to provide a greater guarantee of the state of a variable inside a catch block. That it exists and has been initialized.

Note that the language designers have done a good job with other constructs like *using* and *lock* where the problem and scope is well defined, which allows you to write clearer code.

e.g. the *using* keyword with *IDisposable* objects in:

```
using(Writer writer = new Writer())
{
    writer.Write("Hello");
}
```

is equivalent to:

```
Writer writer = new Writer();
try
{
    writer.Write("Hello");
}
finally
{
    if( writer != null)
    {
        ((IDisposable)writer).Dispose();
    }
}
```

If your try/catch/finally is hard to understand, try refactoring or introducing another layer of indirection with an intermediate class that encapsulates the semantics of what you are trying to accomplish. Without seeing real code, it's hard to be more specific.

answered Sep 18, 2008 at 23:56

Robert Paulson

18k ● 6 ● 36 ● 53

Instead of a local variable, a public property could be declared; this also should avoid another potential error of an unassigned variable. public string S { get; set; }

1

Share Improve this answer Follow

answered Jul 23, 2013 at 17:40

usefulBee

9,670 ● 10 ● 52 ● 95

If the assignment operation fails your catch statement will have a null reference back to the unassigned variable.

-1

Share Improve this answer Follow

answered Sep 18, 2008 at 18:01

SaaS Developer

9,895 ● 8 ● 37 ● 45

2 It's unassigned. It's not even null (unlike instance and static variables).
– Tom Hawtin - tackline Sep 19, 2008 at 9:20

C# 3.0:

-1

```
string html = new Func<string>(() =>
{
    string webpage;

    try
    {
        using(WebClient downloader = new WebClient())
        {
            webpage = downloader.DownloadString(url);
        }
    }
    catch(WebException)
    {
        Console.WriteLine("Download failed.");
    }

    return webpage;
})();
```

edited Sep 19, 2008 at 0:08

answered Sep 18, 2008 at 23:58

core

33k ● 45 ● 140 ● 195

Share

Improve this answer

Follow

WTF? Why the down-vote? Encapsulation is integral to OOP. Looks pretty too. – core Jan 20, 2009 at 18:08

3 I wasn't the downvote, but what's wrong is returning an uninitialized string. – Ben Voigt Apr 27, 2010 at 5:33