Which, and why, do you prefer Exceptions or Return codes? [closed]

Asked 16 years, 3 months ago Modified 1 year, 3 months ago

Viewed 38k times

107

Closed. This question is opinion-based. It is not currently accepting answers.

Want to improve this question? Update the question so it can be answered with facts and citations by editing this post.

Closed last year.

The community reviewed whether to reopen this question last year and left it closed:

> Original close reason(s) were not resolved

Improve this question

My question is what do most developers prefer for error handling, Exceptions or Error Return Codes. Please be language(or language family) specific and why you prefer one over the other.

I'm asking this out of curiosity. Personally I prefer Error Return Codes since they are less explosive and don't force user code to pay the exception performance penalty if they don't want to.

update: thanks for all the answers! I must say that although I dislike the unpredictability of code flow with exceptions. The answer about return code (and their elder brother handles) do add lots of Noise to the code.

language-agnostic **exception**

Share

Improve this question

Follow

edited Sep 20, 2008 at 0:08

asked Sep 19, 2008 at 4:43

Robert Gould

69.7k ● 61 ● 191 ● 275

2 A chicken or the egg problem from the world of software... eternally debatable. :) – Gishu Sep 19, 2008 at 5:06

Sorry about that, but hopefully having a variety of opinions will help people (myself included) choose appropriately. – Robert Gould Sep 19, 2008 at 5:16

Generally, the use of exceptions for control flow is an anti-pattern softwareengineering.stackexchange.com/questions/189222/... – Michael Freidgeim Nov 7, 2021 at 10:54

26 Answers Sorted by: Highest score (default)

123

For some languages (i.e. C++) Resources leak should not be a reason

C++ is based on RAII.

If you have code that could fail, return or throw (that is, most normal code), then you should have your pointer wrapped inside a smart pointer (assuming you have a *very good* reason to not have your object created on stack).

Return codes are more verbose

They are verbose, and tend to develop into something like:

```
if(doSomething())
{
   if(doSomethingElse())
   {
      if(doSomethingElseAgain())
      {
          // etc.
      }
      else
      {
```

```
         // react to failure of
doSomethingElseAgain
      }
   }
   else
   {
      // react to failure of doSomethingElse
   }
}
else
{
   // react to failure of doSomething
}
```

In the end, you code is a collection of idented instructions (I saw this kind of code in production code).

This code could well be translated into:

```
try
{
   doSomething() ;
   doSomethingElse() ;
   doSomethingElseAgain() ;
}
catch(const SomethingException & e)
{
   // react to failure of doSomething
}
catch(const SomethingElseException & e)
{
   // react to failure of doSomethingElse
}
catch(const SomethingElseAgainException & e)
{
   // react to failure of doSomethingElseAgain
}
```

Which cleanly separate code and error processing, which **can** be a **good** thing.

Return codes are more brittle

If not some obscure warning from one compiler (see "phjr" 's comment), they can easily be ignored.

With the above examples, assume than someone forgets to handle its possible error (this happens...). The error is ignored when "returned", and will possibly explode later (i.e. a NULL pointer). The same problem won't happen with exception.

The error won't be ignored. Sometimes, you want it to not explode, though... So you must chose carefully.

Return Codes must sometimes be translated

Let's say we have the following functions:

- doSomething, which can return an int called NOT_FOUND_ERROR
- doSomethingElse, which can return a bool "false" (for failed)
- doSomethingElseAgain, which can return an Error object (with both the __LINE__, __FILE__ and half

the stack variables.

- doTryToDoSomethingWithAllThisMess which, well... Use the above functions, and return an error code of type...

What is the type of the return of doTryToDoSomethingWithAllThisMess if one of its called functions fail ?

Return Codes are not a universal solution

Operators cannot return an error code. C++ constructors can't, too.

Return Codes means you can't chain expressions

The corollary of the above point. What if I want to write:

```
CMyType o = add(a, multiply(b, c)) ;
```

I can't, because the return value is already used (and sometimes, it can't be changed). So the return value becomes the first parameter, sent as a reference... Or not.

Exception are typed

You can send different classes for each kind of exception. Ressources exceptions (i.e. out of memory) should be light, but anything else could be as heavy as necessary (I like the Java Exception giving me the whole stack).

Each catch can then be specialized.

Don't ever use catch(...) without re-throwing

Usually, you should not hide an error. If you do not re-throw, at the very least, log the error in a file, open a messagebox, whatever...

Exception are... NUKE

The problem with exception is that overusing them will produce code full of try/catches. But the problem is elsewhere: Who try/catch his/her code using STL container? Still, those containers can send an exception.

Of course, in C++, don't ever let an exception exit a destructor.

Exception are... synchronous

Be sure to catch them before they bring out your thread on its knees, or propagate inside your Windows message loop.

The solution could be mixing them?

So I guess the solution is to throw when something should **not** happen. And when something can happen, then use a return code or a parameter to enable to user to react to it.

So, the only question is "what is something that should not happen?"

It depends on the contract of your function. If the function accepts a pointer, but specifies the pointer must be non-NULL, then it is ok to throw an exception when the user sends a NULL pointer (the question being, in C++, when didn't the function author use references instead of pointers, but...)

Another solution would be to show the error

Sometimes, your problem is that you don't want errors. Using exceptions or error return codes are cool, but... You want to know about it.

In my job, we use a kind of "Assert". It will, depending on the values of a configuration file, no matter the debug/release compile options:

- log the error
- open a messagebox with a "Hey, you have a problem"
- open a messagebox with a "Hey, you have a problem, do you want to debug"

In both development and testing, this enable the user to pinpoint the problem exactly when it is detected, and not after (when some code cares about the return value, or inside a catch).

It is easy to add to legacy code. For example:

```
void doSomething(CMyObject * p, int iRandomData)
{
   // etc.
}
```

leads a kind of code similar to:

```
void doSomething(CMyObject * p, int iRandomData)
{
   if(iRandomData < 32)
   {
      MY_RAISE_ERROR("Hey, iRandomData " <<
iRandomData << " is lesser than 32. Aborting
processing") ;
      return ;
   }
```

```
   if(p == NULL)
   {
      MY_RAISE_ERROR("Hey, p is NULL
!\niRandomData is equal to " << iRandomData << ".
Will throw.") ;
      throw std::some_exception() ;
   }

   if(! p.is Ok())
   {
      MY_RAISE_ERROR("Hey, p is NOT Ok!\np is
equal to " << p->toString() << ". Will try to
continue anyway") ;
   }

   // etc.
}
```

(I have similar macros that are active only on debug).

Note that on production, the configuration file does not exist, so the client never sees the result of this macro... But it is easy to activate it when needed.

Conclusion

When you code using return codes, you're preparing yourself for failure, and hope your fortress of tests is secure enough.

When you code using exception, you know that your code can fail, and usually put counterfire catch at chosen strategic position in your code. But usually, your code is more about "what it must do" then "what I fear will happen".

But when you code at all, you must use the best tool at your disposal, and sometimes, it is "Never hide an error, and show it as soon as possible". The macro I spoke above follow this philosophy.

Share Improve this answer

Follow

edited Aug 2, 2020 at 17:57

Aykhan Hagverdili

29.8k ● 6 ● 48 ● 103

answered Sep 21, 2008 at 15:20

paercebal

83.2k ● 38 ● 134 ● 160

7 Yeah, BUT regarding your first example, that could be easily written as: `if( !doSomething() ) { puts( "ERROR - doSomething failed" ) ; return ; // or react to failure of doSomething } if( !doSomethingElse() ) { // react to failure of doSomethingElse() }` – bobobobo Jan 30, 2012 at 2:19

Why not... But then, I still find `doSomething(); doSomethingElse(); ...` better because if I need to add if/while/etc. statements for normal execution purposes, I don't want them mixed with if/while/etc. statements added for exceptional purposes... And as the real rule about using exceptions is **to throw**, not **to catch**, the try/catch statements usually are not invasive. – paercebal Jan 30, 2012 at 9:02

3 Your first point shows what the problem with exceptions is. Your control flow gets weird and is seperated from the actual problem. It is replacing some levels of identation by a cascade of catches. I would use both, return codes (or return objects with heavy information) for possible errors and exceptions for things which are not *expected*. – Peter Jan 3, 2013 at 13:06

3 @Peter Weber : It is not weird. It is separated from the actual problem because it is not part of the **normal** execution flow. It is an **exceptional** execution. And then, again, the point about exception is, in case of **exceptional** error, to **throw often**, and **catch rarely**, if ever. So the catch blocks rarely even appears in the code. – paercebal Jan 3, 2013 at 17:58

1 The example in this kind of debate are very simplistic. Usually, "doSomething()" or "doSomethingElse()" actually perform something, like changing a state of some object. The exception code does not guarantee returning the object to the previous state and even less when the catch is very far from the throw... As an example, imagine doSomething is called twice, and increment a counter before throwing. How do you know when catching the exception that you should decrement once or twice? In general, writing exception safe code for anything that's not a toy example is very hard (impossible ?). – xryl669 Sep 27, 2016 at 18:14

42

I use both actually.

I use return codes if it's a known, possible error. If it's a scenario that I know can, and will happen, then there's a code that gets sent back.

Exceptions are used solely for things that I'm NOT expecting.

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:46

Stephen Wrighton

37.8k ● 6 ● 70 ● 87

+1 for very simple way to go. Atleast Its short enough so that I can read it very quickly. :-) – Ashish Gupta May 11, 2010 at

14:08

2 "*Exceptions are used solely for things that I'm NOT expecting.*" If you are not expecting them, then why do or how can you use them? – anar khalilov Dec 4, 2013 at 13:13

6 Just because I'm not expecting it to happen, does not mean that I can't see how it could. I expect my SQL Server to be turned on and responding. But I still code my expectations so that I can fail gracefully if an unexpected downtime is occurring. – Stephen Wrighton Dec 4, 2013 at 14:15

2 Wouldn't an unresponsive SQL Server be easily categorized under "known, possible error"? – tkburbidge Oct 19, 2020 at 13:58

1 @StephenWrighton if you can see how something can happen, then how can you not expect it? it works both ways. you seem to be making a different distinction – symbiont Apr 3, 2022 at 18:09

27

According to Chapter 7 titled "Exceptions" in **Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries**, numerous rationales are given for why using exceptions over return values is necessary for OO frameworks such as C#.

Perhaps this is the most compelling reason (page 179):

"Exceptions integrate well with object-oriented languages. Object-oriented languages tend to impose constraints on member signatures that are not imposed by functions in non-OO languages. **For example, in the case of constructors, operator overloads, and properties, the developer has no choice in the return value.** For this

reason, it is not possible to standardize on return-value-based error reporting for object-oriented frameworks. **An error reporting method, such as exceptions, which is out of band of the method signature is the only option.**"

Share Improve this answer

Follow

answered Sep 19, 2008 at 8:06

hurst

1,740 ● 12 ● 8

1 Highly recommend giving this chapter a read. The chapter gives a very systematic guide to error/exception handling with lots of dos and don'ts that I cannot find online. – Zhe Feb 23, 2021 at 13:38

11

My preference (in C++ and Python) is to use exceptions. The language-provided facilities make it a well-defined process to both raise, catch and (if necessary) re-throw exceptions, making the model easy to see and use. Conceptually, it's cleaner than return codes, in that specific exceptions can be defined by their names, and have additional information accompanying them. With a return code, you're limited to just the error value (unless you want to define an ReturnStatus object or something).

Unless the code you're writing is time-critical, the overhead associated with unwinding the stack is not significant enough to worry about.

Share Improve this answer

answered Sep 19, 2008 at 4:50

Follow

Jason Etheridge
6,897 ● 5 ● 31 ● 33

3 Remember that using exceptions makes program analysis harder. – Paweł Hajdan Sep 19, 2008 at 9:17

1 Using Exceptions can make program analysis easier. It's not a rule. Error code returns can be a nightmare to analyse when the call stack gets deep. – NeilG Sep 23, 2023 at 7:05

10

Always use exceptions by default, BUT consider providing an additional tester-doer option or TryParse!

For me the answer is really clear. When the context dictates a TryParse or Tester-Doer pattern (ie cpu intensive or public api), I will ADDITIONALLY provide those methods to the exception throwing version. I think blanket rules of avoiding exceptions are misguided, unsupported, and likely cause far more expense in terms of bugs, than any performance issues they claim to prevent.

No, Microsoft does NOT say to not use exceptions (common misinterpretation).

It says if you're **designing an API** provide ways to help a user of that API to avoid THROWING exceptions if they need too (TryParse and Tester-Doer patterns)

> ❌ DO NOT use exceptions for the normal flow of control, if possible.
>
> Except for system failures and operations with potential race conditions, framework designers should design APIs so users can write code that does not throw exceptions. For example, you can provide a way to check preconditions before calling a member so users can write code that does not throw exceptions.

What is inferred here is that the non-tester-doer/non-TryParse implementation SHOULD throw an exception upon failure and then the user CAN change that to one of your tester-doer or TryParse methods for performance. Pit of success is maintained for safety and the user OPTS INTO the more dangerous but more performant method.

Microsoft DOES say to NOT use return codes TWICE, here:

> ❌ DO NOT return error codes.
>
> Exceptions are the primary means of reporting errors in frameworks.
>
> ✔️ DO report execution failures by throwing exceptions.

and here:

> ❌ DO NOT use error codes because of concerns that exceptions might affect performance negatively.
>
> To improve performance, it is possible to use either the Tester-Doer Pattern or the Try-Parse Pattern, described in the next two sections.

If you're not using exceptions you're probably breaking this other rule of returning return codes or booleans from a non-tester/non-try implementation. Again, TryParse does not replace Parse. It is provided in addition to Parse

MAIN REASON: Return codes fail the "Pit of Success" test for me almost every time.

- It is far too easy to forget to check a return code and then have a red-herring error later on.
    - var success = Save()? How much performance is worth someone forgetting an if check here?
    - var success = TrySave()? Better, but are we going to abuse everything with the TryParse pattern? Did you still provide a Save method? *Note: It is worth mentioning here that I now believe the TryParse pattern was not lazily named, but was named specifically on purpose to discourage a more generic use. A TryX*

method should not have multiple failure reasons that cannot be described by a simple boolean.

- Return codes don't have any of the great debugging information on them like call stack, inner exceptions.
- Return codes do not propagate which, along with the point above, tends to drive excessive and interwoven diagnostic logging instead of logging in one centralized place (application and thread level exception handlers).
- Return codes tend to drive messy code in the form of nested 'if' blocks
- Developer time spent debugging an unknown issue that would otherwise have been an obvious exception (pit of success) IS expensive.
- If the team behind C# didn't intend for exceptions to govern control flow, execeptions wouldn't be typed, there would be no "when" filter on catch statements, and there would be no need for the parameter-less 'throw' statement.

Regarding Performance:

- Exceptions may be computationally expensive RELATIVE to not throwing at all, but they're called EXCEPTIONS for a reason. Speed comparisons always manage to assume a 100% exception rate which should never be the case. Even if an exception is 100x slower, how much does that really matter if it only happens 1% of the time?

- Context is everything. For example, A Tester-Doer or Try option to avoid a unique key violation is likely to waste more time and resources on average (checking for existance when a collision is rare) than just assuming a successful entry and catching that rare violation.
- Unless we're talking floating point arithmetic for graphics applications or something similar, CPU cycles are cheap compared to developer time.
- Cost from a time perspective carries the same argument. Relative to database queries or web service calls or file loads, normal application time will dwarf exception time. Exceptions were nearly sub-MICROsecond in 2006
- I dare anybody that works in .net, to set your debugger to break on all exceptions and disable just my code and see how many exceptions are already happening that you don't even know about.
- Jon Skeet says "[Exceptions are] not slow enough to make it worth avoiding them in normal use". The linked response also contains two articles from Jon on the subject. His generalized theme is that exceptions are fine and if you're experiencing them as a performance problem, there's likely a larger design issue.

answered Oct 27, 2017 at 19:41

b_levitt

7,395 ● 3 ● 46 ● 60

1 I wouldn't take programming advice from Microsoft anyway. Some of the biggest budgets and time available with some of the worst software in the industry. – NeilG Sep 23, 2023 at 7:00

I understand that this is a language agnostic thread, but believe the references are worth citing for those programming on the MS stack. Further, I provided them more to dispute the idea that exceptions are to be avoided which I'm framing as a common misconception due to a misreading of that documentation. – b_levitt Sep 25, 2023 at 17:34

I'm not knocking your conclusions @b_levitt, but just reacting to the idea of taking design advice from Microsoft, but on reading your post more carefully I see that, probably, Microsoft is giving advice on how to use particular features of their software less so that you don't run into performance problems. So glad I'm not using that stack. – NeilG Sep 25, 2023 at 23:25

8

Exceptions should only be returned where something happens that you were not expecting.

The other point of exceptions, historically, is that return codes are inherently proprietary, sometimes a 0 could be returned from a C function to indicate success, sometimes -1, or either of them for a fail with 1 for a success. Even when they are enumerated, enumerations can be ambiguous.

Exceptions can also provide a lot more information, and specifically spell out well 'Something Went Wrong, here's what, a stack trace and some supporting information for the context'

That being said, a well enumerated return code can be useful for a known set of outcomes, a simple 'heres n outcomes of the function, and it just ran this way'

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:51

johnc

40.2k ● 37 ● 103 ● 140

1 This is not correct. It depends on the domain and the design. In Python using Exceptions for flow control is normal and baked into the language. – NeilG Sep 23, 2023 at 7:01

7

I dislike return codes because they cause the following pattern to mushroom throughout your code

```
CRetType obReturn = CODE_SUCCESS;
obReturn = CallMyFunctionWhichReturnsCodes();
if (obReturn == CODE_BLOW_UP)
{
  // bail out
  goto FunctionExit;
}
```

Soon a method call consisting of 4 function calls bloats up with 12 lines of error handling.. Some of which will never happen. If and switch cases abound.

Exceptions are cleaner if you use them well... to signal exceptional events .. after which the execution path cannot continue. They are often more descriptive and informational than error codes.

If you have multiple states after a method call that should be handled differently (and are not exceptional cases), use error codes or out params. Although Personaly I've found this to be rare..

I've hunted a bit about the 'performance penalty' counterargument.. more in the C++ / COM world but in the newer languages, I think the difference isn't that much. In any case, when something blows up, performance concerns are relegated to the backburner :)

Share Improve this answer

Follow

edited Nov 13, 2017 at 19:07

abatishchev

100k ● 88 ● 301 ● 442

answered Sep 19, 2008 at 5:05

Gishu

137k ● 47 ● 226 ● 311

7

In Java, I use (in the following order):

1. Design-by-contract (ensuring preconditions are met before trying *anything* that might fail). This catches most things and I return an error code for this.

2. Returning error codes whilst processing work (and performing rollback if needed).
3. Exceptions, but these are used *only* for unexpected things.

Share Improve this answer Follow

edited Nov 14, 2017 at 1:50

answered Sep 19, 2008 at 4:58

paxdiablo
880k ● 241 ● 1.6k ● 2k

1 Wouldn't it be slightly more correct to use assertions for contracts? If the contract is broken, there's nothing to save you. – Paweł Hajdan Sep 19, 2008 at 9:21

@PawełHajdan, assertions are disabled by default, I believe. This has the same problem as C's `assert` stuff in that it won't catch problems in production code unless you run with assertions on all the time. I tend to see assertions as a way to catch problems during development, but *only* for stuff that will assert or not assert consistently (such as stuff with constants, *not* stuff with variables or anything else that can change at runtime). – paxdiablo Mar 10, 2020 at 0:50

2 And twelve years to respond to your question. I should run a help desk :-) – paxdiablo Mar 10, 2020 at 0:50

4

I wrote a blog post about this a while ago.

The performance overhead of throwing an exception should not play any role in your decision. If you're doing it

right, after all, an exception is *exceptional*.

Share Improve this answer

Follow

answered Sep 19, 2008 at 5:00

Thomas

181k ● 55 ● 376 ● 501

The linked blog post is more about look before you leap (checks) vs. easier to ask forgiveness than permission (exceptions or return codes). I've replied there with my thoughts about that issue (hint: TOCTTOU). But this question is about a different issue, namely under what conditions to use a language's exception mechanism rather than returning a value with special properties. – Damian Yerrick Nov 29, 2016 at 18:31

I completely agree. It seems I've learned a thing or two in the past nine years ;) – Thomas Nov 30, 2016 at 10:41

1 Exceptions are not necessarily exceptional. (I thought it was well known that) in Python Exceptions are part of normal flow control. You may make a decision to use them that way if it suits your design but there are other models. It's a good idea to compare the **behaviour** differences between the two and decide on that basis. Refer to other answers and comments. – NeilG Sep 23, 2023 at 7:06

1 Also, in Python at least, exceptions are not costly at all, and may sometimes be quicker and lighter. They have that reputation from older languages like Java. – NeilG Sep 23, 2023 at 7:07

4

I have a simple set of rules:

1) Use return codes for things you expect your immediate caller to react to.

2) Use exceptions for errors that are broader in scope, and may reasonable be expected to be handled by something many levels above the caller so that awareness of the error does not have to percolate up through many layers, making code more complex.

In Java I only ever used unchecked exceptions, checked exceptions end up just being another form of return code and in my experience the duality of what might be "returned" by a method call was generally more of a hinderance than a help.

Share Improve this answer

Follow

answered Sep 19, 2008 at 5:02

Kendall Helmstetter Gelner

75k ● 26 ● 131 ● 151

1 This is definitely a major difference that Exceptions provide: to enable flow control to be reverted back up several levels. I used this in a service that needed to execute multiple levels that would invoke further levels depending on the input, searching for completion of the problem set. This execution therefore naturally had many dead end pathways and it was most efficient and simple to just raise an exception for any execution thread that proved fruitless, automatically dropping out of the whole execution tree at that point. It would be a nightmare to implement with return codes. – NeilG Sep 23, 2023 at 7:03

I use Exceptions in python in both Exceptional, and non-Exceptional circumstances.

4

It is often nice to be able to use an Exception to indicate the "request could not be performed", as opposed to returning an Error value. It means that you /always/ know that the return value is the right type, instead of arbitarily None or NotFoundSingleton or something. Here is a good example of where I prefer to use an exception handler instead of a conditional on the return value.

```
try:
    dataobj = datastore.fetch(obj_id)
except LookupError:
    # could not find object, create it.
    dataobj = datastore.create(....)
```

The side effect is that when a datastore.fetch(obj_id) is run, you never have to check if its return value is None, you get that error immediately for free. This is counter to the argument, "your program should be able to perform all its main functionality without using exceptions at all".

Here is another example of where exceptions are 'exceptionally' useful, in order to write code for dealing with the filesystem that isn't subject to race conditions.

```
# wrong way:
if os.path.exists(directory_to_remove):
    # race condition is here.
    os.path.rmdir(directory_to_remove)

# right way:
try:
    os.path.rmdir(directory_to_remove)
except OSError:
```

```
    # directory didn't exist, good.
    pass
```

One system call instead of two, no race condition. This is a poor example because obviously this will fail with an OSError in more circumstances than the directory doesn't exist, but it's a 'good enough' solution for many tightly controlled situations.

Share Improve this answer

Follow

answered Sep 19, 2008 at 5:12

Jerub

42.6k ● 15 ● 75 ● 90

1 Second example is misleading. Supposedly wrong way is wrong because os.path.rmdir code is designed to throw exception. Correct implementation in return code flow would be 'if rmdir(...)==FAILED: pass' – MaR Feb 12, 2013 at 12:14

1 This answer is well worth reading and understanding properly because it's one of the few that articulate the benefits of using exceptions as non-exceptional error returns or failures. There's this myth apparently established by earlier more rigid languages that "exceptions always have to be for exceptional circumstances" that may be generally good advice in Java but a good programmer should have understanding why he uses return codes or exceptions and choose wisely, not just some ancient myth from the past that it's taboo to do otherwise because confucius said so. – NeilG Sep 23, 2023 at 7:18

3

A great piece of advice I got from *The Pragmatic Programmer* was something along the lines of "your

program should be able to perform all its main functionality without using exceptions at all".

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:53

Smashery

59.6k ● 31 ● 100 ● 129

4 You're misinterpreting it. What they meant was "if your program throws exceptions in its normal flows, it's wrong". In other words "only use exceptions for exceptional things". – Paweł Hajdan Sep 19, 2008 at 9:18

1 And even that is not necessarily true, @PawełHajdan. In Python exceptions are used for normal flow control. The only real way to decide between error codes and exceptions is to compare behaviours and choose on that basis. It may be a helpful design decision to rule that exceptions are not used in normal flow (although that's a ridiculous proposition for Python) but in some problem sets that may exclude better designs. Either way this answer is not helpful. It provides no reasons. And it's wrong. Not sure why the advice is supposed to be "great". – NeilG Sep 23, 2023 at 7:13

3

I prefer to use exceptions for error handling and return values (or parameters) as the normal result of a function. This gives an easy and consistent error-handling scheme and if done correctly it makes for much cleaner looking code.

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:55

Trent

13.5k ● 4 ● 41 ● 36

1 I think this is probably the default approach that works best for most scenarios but I don't think it's necessarily the rule. I've worked with complex deeply nested calls where I've found it's better for design, readability and performance to just throw out of a deeply nested routine when it's determined it's not going to work. – NeilG Sep 23, 2023 at 7:15

3

One of the big differences is that exceptions force you to handle an error, whereas error return codes can go unchecked.

Error return codes, if used heavily, can also cause very ugly code with lots of if tests similar to this form:

```
if(function(call) != ERROR_CODE) {
    do_right_thing();
}
else {
    handle_error();
}
```

Personally I prefer to use exceptions for errors that SHOULD or MUST be acted upon by the calling code, and only use error codes for "expected failings" where returning something is actually valid and possible.

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:56

Daniel Bruce

11.5k ● 4 ● 32 ● 28

1 At least in C/C++ and gcc you can give a function an attribute that will generate a warning when its return value is ignored.

– Paweł Hajdan Sep 19, 2008 at 9:20

1 phjr: While I disagree with the "return error code" pattern, your comment should perhaps become a full answer. I find it interesting enough. At the very least, it did give me an useful information. – paercebal Sep 21, 2008 at 14:32

3

I believe the return codes adds to code noise. For example, I always hated the look of COM/ATL code due to return codes. There had to be an HRESULT check for every line of code. I consider the error return code is one of the bad decisions made by architects of COM. It makes it difficult to do logical grouping of the code, thus code review becomes difficult.

I am not sure about the performance comparison when there is an explicit check for the return code every line.

Share Improve this answer Follow

answered Sep 19, 2008 at 5:29

community wiki
rpattabi

2 COM wad designed to be usable by languages that don't support exceptions. – Kevin Sep 19, 2008 at 7:15

That is a good point. It makes sense to deal with error codes for scripting languages. At least VB6 hides error code details well with encapsulating them in the Err object, which somewhat helps in cleaner code. – rpattabi Sep 20, 2008 at 11:58

1 I disagree: VB6 only logs the last error. Combined with the infamous "on error resume next", you'll miss the source of your problem altogether, by the time you see it. Note that this is the basis of error handling in Win32 APII (see GetLastError function) – paercebal Sep 21, 2008 at 14:24

3

There is many reason to prefer Exceptions over return code:

- Usually, for readibility, people try to minimize the number of return statement in a method. Doing so, exceptions prevent to do some extra work while in a incoorect state, and thus prevent to potentially damage more data.
- Exception are generally more verbose arn more easilly extensible than return value. Assume that a method return natural number and that you use negative numbers as return code when an error occurs, if the scope of you method change and now return integers, you'll have to modify all the method calls instead of just tweaking a little bit the exception.
- Exceptions allows more easilly to separate error handling of normal behaviour. They allows to ensure that some operations performs somehow as an atomic operation.

Share Improve this answer

Follow

edited Oct 22, 2008 at 21:07

Adam Bellaire

110k ● 19 ● 152 ● 165

answered Sep 19, 2008 at 5:01

gizmo
11.9k ● 6 ● 46 ● 62

3

I generally prefer return codes because **they let the caller decide whether the failure is exceptional**.

This approach is typical in the Elixir language.

```
# I care whether this succeeds. If it doesn't
return :ok, raise an exception.
:ok = File.write(path, content)

# I don't care whether this succeeds. Don't check
the return value.
File.write(path, content)

# This had better not succeed - the path should be
read-only to me.
# If I get anything other than this error, raise
an exception.
{:error, :erofs} = File.write(path, content)

# I want this to succeed but I can handle its
failure
case File.write(path, content) do
  :ok => handle_success()
  error => handle_error(error)
end
```

People mentioned that return codes can cause you to have a lot of nested `if` statements, but that can be handled with better syntax. In Elixir, the `with` statement lets us easily separate a series of happy-path return value from any failures.

```
with {:ok, content} <- get_content(),
  :ok <- File.write(path, content) do
    IO.puts "everything worked, happy path code
goes here"
else
  # Here we can use a single catch-all failure
clause
  # or match every kind of failure individually
  # or match subsets of them however we like
  _some_error => IO.puts "one of those steps
failed"
  _other_error => IO.puts "one of those steps
failed"
end
```

Elixir still has functions that raise exceptions. Going back to my first example, I could do either of these to raise an exception if the file can't be written.

```
# Raises a generic MatchError because the return
value isn't :ok
:ok = File.write(path, content)

# Raises a File.Error with a descriptive error
message - eg, saying
# that the file is read-only
File.write!(path, content)
```

If I, as the caller, know that I want to raise an error if the write fails, I can choose to call `File.write!` instead of `File.write`. Or I can choose to call `File.write` and handle each of the possible reasons for failure differently.

Of course it's always possible to `rescue` an exception if we want to. But compared to handling an informative return value, it seems awkward to me. If I know that a

function call can fail or even should fail, its failure isn't an exceptional case.

Share Improve this answer

Follow

answered Aug 13, 2020 at 14:33

Nathan Long

126k ● 103 ● 350 ● 463

2

With any decent compiler or runtime environment exceptions do not incur a significant penalty. It's more or less like a GOTO statement that jumps to the exception handler. Also, having exceptions caught by a runtime environment (like the JVM) helps isolating and fixing a bug a lot easier. I'll take a NullPointerException in Java over a segfault in C any day.

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:47

Kyle Cronin

79k ● 45 ● 151 ● 167

3 Exceptions are extremely expensive. They have to walk the stack to find potential exception handlers. This stack walk is not cheap. If a stack trace is built, it's even more expensive, because then the entire stack *must* be parsed. – Derek Park Sep 19, 2008 at 4:54

1 Call stacks can get extremely complex at runtime, and compilers generally don't do that kind of analysis. Even if they did, you'd still have to walk the stack to get a trace. You'd also still have to unwind the stack to deal with `finally` blocks and destructors for stack-allocated objects. – Derek Park Sep 19, 2008 at 5:16

1 I do agree that the debugging benefits of exceptions often make up for the performance costs, though. – Derek Park Sep 19, 2008 at 5:17

1 Derak Park, exception are expensive when they happen. This is the reason they should not be overused. But when they don't happen, they cost virtually nothing. – paercebal Sep 21, 2008 at 15:26

1 This question is specifically tagged language-agnostic. Java may have problems with exceptions but not all languages do. Python uses exceptions copiously and as part of normal program flow. – NeilG Sep 23, 2023 at 7:31

2

Exceptions are not for error handling, IMO. Exceptions are just that; exceptional events that you did not expect. Use with caution I say.

Error codes can be OK, but returning 404 or 200 from a method is bad, IMO. Use enums (.Net) instead, that makes the code more readable and easier to use for other developers. Also you don't have to maintain a table over numbers and descriptions.

Also; the try-catch-finally pattern is an anti-pattern in my book. Try-finally can be good, try-catch can also be good but try-catch-finally is never good. try-finally can often times be replaced by a "using" statement (IDispose pattern), which is better IMO. And Try-catch where you actually catch an exception you're able to handle is good, or if you do this:

```
try{
    db.UpdateAll(somevalue);
}
catch (Exception ex) {
    logger.Exception(ex, "UpdateAll method
failed");
    throw;
}
```

So as long as you let the exception continue to bubble it's OK. Another example is this:

```
try{
    dbHasBeenUpdated = db.UpdateAll(somevalue); //
true/false
}
catch (ConnectionException ex) {
    logger.Exception(ex, "Connection failed");
    dbHasBeenUpdated = false;
}
```

Here I actually handle the exception; what I do outside of the try-catch when the update method fails is another story, but I think my point has been made. :)

Why is then try-catch-finally an anti-pattern? Here's why:

```
try{
    db.UpdateAll(somevalue);
}
catch (Exception ex) {
    logger.Exception(ex, "UpdateAll method
failed");
    throw;
}
finally {
```

```
    db.Close();
}
```

What happens if the db object has already been closed? A new exception is thrown and it has to be handled! This is better:

```
try{
    using(IDatabase db =
DatabaseFactory.CreateDatabase()) {
        db.UpdateAll(somevalue);
    }
}
catch (Exception ex) {
    logger.Exception(ex, "UpdateAll method
failed");
    throw;
}
```

Or, if the db object does not implement IDisposable do this:

```
try{
    try {
        IDatabase db =
DatabaseFactory.CreateDatabase();
        db.UpdateAll(somevalue);
    }
    finally{
        db.Close();
    }
}
catch (DatabaseAlreadyClosedException dbClosedEx)
{
    logger.Exception(dbClosedEx, "Database
connection was closed already.");
}
catch (Exception ex) {
```

```
    logger.Exception(ex, "UpdateAll method
failed");
    throw;
}
```

That's my 2 cents anyway! :)

Share Improve this answer

Follow

edited Jul 22, 2015 at 14:45

superjos

12.7k ● 6 ● 94 ● 141

answered Sep 19, 2008 at 7:41

noocyte

2,522 ● 5 ● 29 ● 44

It will be strange if an object has .Close() but has no .Dispose() – abatishchev Aug 17, 2009 at 8:12

I'm only using Close() as an example. Feel free to think of it as something else. As I state; the using pattern should be used (doh!) if available. This of course implies that the class implements IDisposable and as such you could call Dispose. – noocyte Aug 20, 2009 at 6:39

1 Any such problems as these with try / catch or try / except or whatever language you are using are due to mismanagement of the code and are not inherent to exceptions. You need to make sure you handle the cases where a file object or connection is no longer available properly. Languages often provide a finally clause and other methods for just this purpose. This answer seems to be very language specific and doesn't really contain any good reasons. Please refer to other answers and comments. – NeilG Sep 23, 2023 at 7:11

One thing I fear about exceptions is that throwing an exception will screw up code flow. For example if you do

```
void foo()
{
  MyPointer* p = NULL;
  try{
    p = new PointedStuff();
    //I'm a module user and  I'm doing stuff that
might throw or not

  }
  catch(...)
  {
    //should I delete the pointer?
  }
}
```

Or even worse what if I deleted something I shouldn't have, but got thrown to catch before I did the rest of the cleanup. Throwing put a lot of weight on the poor user IMHO.

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:55

Robert Gould

69.7k ● 61 ● 191 ● 275

This is what the <code>finally</code> statement is for. But, alas, it is not in the C++ standard... – Thomas Sep 19, 2008 at 4:59

In C++ you should stick to the rule of thumb "Acquire resources in construcotor and release them in destrucotor. For this particular case auto_ptr will do just prefectly. – Serge Sep 19, 2008 at 9:02

1 Thomas, you're wrong. C++ has not finally because it does not need it. It has RAII instead. The solution of Serge is one solution using RAII. – paercebal Sep 21, 2008 at 14:26

Robert, use Serge's solution, and you'll find your problem going away. Now, if you write more try/catches than throws, then (by judging your comment) perhaps you have a problem in your code. Of course, using catch(...) without a re-throw is usually bad, as it hides the error to better ignore it.
– paercebal Sep 21, 2008 at 14:28

1 Yes, well, @RobertGould, programming involves complexity proportional to the problem domain and you need to handle that. That's why we break the problem down into modules and solve each problem separately. Either way this is the complexity you need to handle but do that by breaking it into simpler problems, not by avoiding certain programming tools. Exceptions may sometimes be the behaviour that you want with cleaner and simpler code. – NeilG Sep 23, 2023 at 7:22

1

My general rule in the exception vs. return code argument:

- Use errorcodes when you need localization/internationalization -- in .NET, you could use these errorcodes to reference a resource file which will then display the error in the appropriate language. Otherwise, use exceptions
- Use exceptions only for errors that are *really* exceptional. If it's something that happens fairly often, either use a boolean or an enum errorcode.

Share Improve this answer answered Sep 19, 2008 at 4:58

Follow

Jon Limjap
95.3k • 15 • 103 • 153

1 There is no reason while you could not use an exception when you do l10n/i18n. Exceptions can contain localised information as well. – gizmo Sep 19, 2008 at 5:03

1 There are no reasons given here in this answer and these "rules" fly against good reasons given for using exceptions in other answers and comments. Please read other answers and comments for better advice. – NeilG Sep 23, 2023 at 7:20

1

I don't find return codes to be less ugly than exceptions. With the exception, you have the `try{} catch() {} finally {}` where as with return codes you have `if(){}`. I used to fear exceptions for the reasons given in the post; you don't know if the pointer needs to be cleared, what have you. But I think you have the same problems when it comes to the return codes. You don't know the state of the parameters unless you know some details about the function/method in question.

Regardless, you have to handle the error if possible. You can just as easily let an exception propagate to the top level as ignore a return code and let the program segfault.

I do like the idea of returning a value (enumeration?) for results and an exception for an exceptional case.

Share Improve this answer answered Sep 19, 2008 at 5:09

Jonathan Adelson
3,321 ● 5 ● 32 ● 39

There are some important aspects that remain unmentioned in this - very interesting - discussion so far.

First, it is important to note that exceptions don't apply to distributed computing, but error codes still do. Imagine communicating services distributed over multiple servers. Some communication might even be asynchronous. And the services might even use different technology stacks. Cleary, an error-handling concept is crucial here. And clearly, exceptions can't be used in this most general case, since errors have to be serialized things sent "through the cable", perhaps even in a language-neutral way. From that angle, error codes (really, error *messages*) are more universal than exceptions. One needs good error-message Kung Fu once one assumes a system-architect view and things need to scale.

The second point is a very different, it is about if or how a language represents discriminated unions. The question was strictly speaking about "error codes". And so were some the answers, mentioning that error codes cannot transport information as nicely as exceptions. This is true if an error code is a number. But for a fairer contrasting with exceptions, one should probably consider error *values* of discriminated union type. So, the return value of the callee would be of discriminated union type, and it would either be the desired happy-path value or *the payload the exception would otherwise have*. How often

this approach is elegant enough to be preferable depends on the programming language. For example, F# has super elegant discriminated unions and the according pattern matching. In such a language it would be more seductive to avoid exceptions than in, say, C++.

The third and final point is about functional programming and pure functions. Exceptions are (in a practical way and in a theoretical-computer-science way) "side effects". In other words, functions or methods that deal with exceptions are not *pure*. (One practical consequence is that with exceptions one must pay attention to evaluation order.) By contrast, error values are pure, because the are just ordinary return values, with no side effects involved. Therefore, functional programmers may more likely frown upon exceptions than object-oriented programmers. (In particular if the language also has an elegant representation of the aforementioned discriminated unions.)

Share Improve this answer Follow

answered Jul 11, 2022 at 11:03

Carsten Führmann

3,440 ● 4 ● 28 ● 25

0

I only use exceptions, no return codes. I'm talking about Java here.

The general rule I follow is if I have a method called `doFoo()` then it follows that if it doesn't "do foo", as it were, then something exceptional has happened and an Exception should be thrown.

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:52

SCdF

59.3k • 24 • 79 • 114

1 There's no reasons given here and no way to evaluate your proposition. I can see you're following the old trope of "exceptions should be exceptional" which others here have also trotted out without reasons too. There's a lot more languages and platforms out there these days and a variety of problem sets and designs which mean choosing between exceptions and error codes needs to be done with reasons and understanding about their behaviour. This answer adds no value and is better off deleted. Please read some other answers and comments for advice. – NeilG Sep 23, 2023 at 7:29

0

For a language like Java, I would go with Exception because the compiler gives compile time error if exceptions are not handled.This forces the calling function to handle/throw the exceptions.

For Python, I am more conflicted. There is no compiler so it's possible that caller does not handle the exception thrown by the function leading to runtime exceptions. If you use return codes you might have unexpected behavior if not handled properly and if you use exceptions you might get runtime exceptions.

Share Improve this answer

Follow

answered Feb 20, 2018 at 22:33

2ank3th

3,109 • 3 • 23 • 36

1 That's an interseting aspect of Java that I did not know about, but for Python you clearly need more experience. Python uses exceptions as part of its normal flow and you get "runtime exceptions" all the time. Although I'm not sure exactly what you mean by that. You should understand what exceptions can be thrown and make sure you catch them. If you get an unexpected exception at run time that's good news because you just discovered a bug you didn't know about. Code better next time. – NeilG Sep 23, 2023 at 7:24

I prefer exceptions over return codes.

Consider a scenario when I call a function `foo` and forget to handle potential errors (exceptions).

- If errors in `foo` are passed via return codes (error codes),
  - at compile time, I won't be alerted
  - if I run the code
    - if the error doesn't happen, I don't notice my mistake
    - if the error happens but doesn't affect the code after calling `foo`, I don't notice my mistake
    - if the error happens and affects the code after calling `foo`, I notice my mistake, but it may be hard to locate the problem
- If errors in `foo` are passed via exceptions that are thrown,

        - at compile time, I won't be alerted
        - if I run the code
            - if the error doesn't happen, I don't notice my mistake
            - if the error happens, I'm assured to notice my mistake

From the comparison above, my conclusion is that exceptions are better than error codes.

However, exceptions are not perfect. There are at least two critical problems:

1. As discussed above, if I forget to handle a potential exception, I won't be alerted until I run the code and the exception is actually thrown.
2. It's hard to determine all the exceptions `foo` may throw, especially when foo calls other functions (which may also throw exceptions).

A feature in Java, “checked exceptions”, solves both problems.

In Java, when defining a function `foo`, I'm required to explicitly specify what exceptions it may throw using the `throws` keyword. Example:

```
private static void foo() throws
FileNotFoundException {
    File file = new File("not_existing_file.txt");
    FileInputStream stream = new
```

```
FileInputStream(file);
}
```

If I call `foo` and forget to handle the potential `FileNotFoundException`, the compiler produces an error. I get alerted at compile time (problem 1 solved). And all possible exceptions are listed explicitly (problem 2 solved).

Share Improve this answer

Follow

answered Dec 11, 2022 at 0:35

Forrest Wei

47 • 6

1 What happens in Java if your code throws an exception that isn't listed as a "checked exception"? – NeilG Sep 23, 2023 at 7:26

If an exception isn't checked, then the compiler doesn't force the programmer to handle it. So it's more likely to forget to handle unchecked exceptions. That's why I think checked exceptions are the way to go. – Forrest Wei Sep 24, 2023 at 12:22

1 So if not checked it may appear unexpectedly at run time, @Forrest_Wei. I don't contradict your proposal; it's nice that the compiler will check that you caught the exceptions you knew your code might throw, that's fine. But the real issue is handling unexpected exceptions. Exceptions that arise because you didn't understand the full implications. Or because someone updated it wrong. Bugs, in other words. Checked exceptions is just a little bell for tired programmers. – NeilG Sep 25, 2023 at 0:03