Catching exceptions in Java

Asked 15 years, 10 months ago Modified 9 years, 11 months ago Viewed 4k times

4

There are certain predefined exceptions in Java, which, if thrown, report that something serious has happened and you'd better improve your code, than catching them in a catch block (if I have understood it correctly). But still I find many programs in which I have the following:

```
} catch (IOException e) {
     ...
} catch (FileNotFoundException e) {
     ....
}
```

and I thought that IOException and FileNotFoundException are exactly such kind of exceptions, which we shouldn't catch in a catch block. Why people do this? Is it better to catch them like this? Java compiler warns anyway about any problem of that kind.

Thank you.

java exception catch-block

Share Improve this question Follow

asked Feb 10, 2009 at 23:17
user42155
49.6k ● 27 ● 62 ● 60

7 Answers

Sorted by: Highest score (default)

14

No, there's nothing wrong with catching `IOException` and `FileNotFoundException` - if you can genuinely *handle* those exceptions. That's the important bit - can you really proceed in the face of that exception? Sometimes you can - very often at the top level of a server, for example, where just because one request fails doesn't mean the next can't proceed. Less often in client apps, although it very much depends on the situation. Can't read a file when you're trying to do a batch import? Okay, abort the operation but don't necessarily shut down the whole process...

You shouldn't have them that way round, admittedly - the `FileNotFoundException` would be masked by the `IOException` which it derives from. Fortunately the compiler flat-out prevents you from doing this.

Share Improve this answer Follow

answered Feb 10, 2009 at 23:19

Jon Skeet
1.5m ● 889 ● 9.3k ● 9.3k

5

The order that you show, with `IOException` caught before `FileNotFoundException` is wrong. Since `FileNotFoundException` extends `IOException`, when a `FileNotFoundException` is thrown, the first handler will be used, and the second handler is dead code.

I haven't tried it, but I'm a little surprised if this compiles. A static analysis tool like FindBugs would catch this error, I hope.

As far as whether you *should* catch a `FileNotFoundException`, it depends on the caller. However, I will say that a `FileNotFoundException` can often be recovered in a meaningful way—prompting for another file, trying a fallback location—rather than simply logging the error or aborting the process.

Share Improve this answer Follow

answered Feb 10, 2009 at 23:30

erickson
269k ● 59 ● 401 ● 497

3

The do this in order to handle different types of exceptions differently. Typically you are going to want to catch the most granular exceptions first, if you put the more broad exceptions at the beginning of your catch block, you will execute that code first, then hit the finally block.

Jon is right, the catch that catches the IOException will catch all IOExceptions and any sub-type of IOException, and since FileNotFoundException is a type of IOException, it will never hit the 2nd catch.

Share Improve this answer Follow

answered Feb 10, 2009 at 23:20

RKitson
2,013 ● 1 ● 18 ● 21

3

There are two types of exceptions in Java, checked exceptions and unchecked exceptions.

Checked exceptions must be handled in a catch block. Not doing this will cause a compiler error. IOException is an example of a checked exception, and must be handled. What you actually do here depends on the application in question, but the Exception must be handled to keep the compiler happy.

Unchecked exceptions do not need to be caught. All classes that extend from RuntimeException are unchecked. A good example of this is a NullPointerException or an ArrayIndexOutOfBoundsException. The compiler doesn't force you to catch these exceptions, but they may still occur when your program runs, causing it to crash.

For the record, IOException can be thrown for something as simple as trying to open a file that doesn't exist. It is a good idea to handle something like this and recover gracefully (dialog to the user saying the file doesn't exist and make an open file dialog reappear or something), rather than letting the program crash.

Share Improve this answer Follow

answered Feb 10, 2009 at 23:32

MichaelM
5,788 ● 2 ● 31 ● 23

3

As Jon says, catching these exceptions is fine in many cases. The kind of exceptions that you shouldn't be catching are things like NullPointerException and ArrayIndexOutOfBoundsException, these indicate bugs in your code.

Java has two types of exception: checked exceptions and unchecked exceptions (those that inherit from RuntimeException).

Checked exceptions, such as IOException, are usually used for unpredictable scenarios that can't be avoided by writing better code. The fact that they are checked means that the compiler forces you to write code that accounts for the possibility of the exceptional scenario. For example, you have to consider the possibility of a FileNotFoundException because you can't guarantee that the file will exist (somebody might move it while your program is running). An IOException might occur because a network connection gets dropped. The compiler forces you to provide a strategy for dealing with these cases, even if it's just to pass the buck by allowing the exception to propagate up the stack for calling code to handle.

Unchecked exceptions on the other hand are best used for things that can be avoided by changing the code. A NullPointerException can always be avoided if the code makes a check for the possibility of a null reference. Likewise, if you are careful with your indices, you will never get an ArrayIndexOutOfBoundsException. The compiler doesn't oblige you to handle these scenarios since they represent bugs that should be fixed.

Share
Improve this answer
Follow

edited May 23, 2017 at 12:11
Community Bot
1 ● 1

answered Feb 10, 2009 at 23:51
Dan Dyer
54.4k ● 19 ● 135 ● 166

3

> *I thought that IOException and FileNotFoundException are exactly such kind of exceptions*

Nope, those are actually the "other" type of exceptions, the kind that go beyond your programming skills. No matter how good do you program, the compiler and the libraries make you be "conscious" that something might happen.

Think about this scenario:

You create an application that saves data to a temp folder.

Everything is alright, you have checked the folder exists, and if not, you create it your self.

And then you're writing 2 mb to that temp folder.

Suddenly, other system process, deletes your temp folder, and you cannot write anymore.

There is nothing you can do to prevent this programmatically, in some systems that operation could happen (In unix the root user may perform rm -rf /tmp and there is nothing you can do about it. In windows I think the system won't let other process delete a file is is being used)

By forcing you to check this kind of exceptions in the code, the platform designers thought that at least you're aware of this.

Jon is correct sometimes there is nothing you can do about it, probably logging before the program dies, that is considered as "handle the exception" (poor handle yes, but handle at least)

```
try {
   ....
} catch( IOException ioe ) {
    logger.severe(
        String.format("Got ioe while writting file %s. Data was acquired using
id = %d, the message is: %s",
             fileName,
             idWhereDataCame,
             ioe.getMessage()) );
  throw ioe;
}
```

Another thing you can do is to "chain" the exception to fit the abstraction.

Probably your application, is has a GUI, showing a IOException to the user won't mean anything or could be a security vulnerability. A modified message could be sent.

```
try {
   ....
} catch( IOException ioe ) {
   throw new EndUserException("The operation you've requeste could not be
completed, please contact your administrator" , ioe );
}
```

And the EndUserException could be trapped somewhere in the gui and presented to the user in a Dialog message (instead of just disappearing the app in his eyes without further information). Of course there was nothing you can do to recover that IOException, but at least you die with style :P

Finally a client code, could use different implementations, and not all the exceptions would make sense.

For instance, think again on the fist scenario. That same "operation" could have three kinds of "plugins" services to perform the data saving.

```
a) Write the data  to a file.
b) Or, write to a db
c) Or write to a remote server.
```

The interface should not throw:

```
java.io.IOException
java.sql.SQLException
```

nor

```
java.net.UnknownHostException
```

But instead something like

```
my.application.DataNotSavedException
```

and the different implementations would handle the exception at the correct level, **and** transform it to the appropriate abstraction:

Client code:

```
 DataSaver saver = DataServer.getSaverFor("someKeyIdString");
```

```
try {
    saver.save( myData );
} catch( DataNotSavedException dnse ) {

    // Oh well... .
    ShowEndUserError("Data could not be saved due to : " dnse.getMessage() );
}
```

Implementation code:

```
class ServerSaver implements DataSaver {
....
    public void save( Data data ) throws DataNotSavedException {
        // Connect the remore server.
        try {
            Socket socket = new Socket( this.remoteServer, this.remotePort );
            OuputStream out = socket.getOut....

            ....
            ....
        } catch ( UnknownHostException uhe ) {
           // Oops....
           throw new DataNotSavedException( uhe );
        }
    }
}
```

FileSaver and DatabaseSaver would do something similar.

All of these are **Checked Exceptions** because the compiler make you check them.

When to use one or the other (checked / unchecked): here

There are other two kinds: here

And finally a much simpler explanation of the Runtime is: here

Share

Improve this answer

Follow

edited May 23, 2017 at 11:48

Community Bot
1 ● 1

answered Feb 11, 2009 at 0:29

OscarRyz
199k ● 119 ● 396 ● 573

0

To take this idea sideways a bit: maybe in another domain it is clearer

What do you do if the car in front of you stops suddenly.

Stop!

So we handle the exception.

So back to code:

What do you do if the file you need is not available ?

either

1. Have a backup. Compiled in as a resource because it's part of your program. *I'm not kidding.*
2. IFF It's a user supplied file :Tell the user; it's their file.
3. Abort the program with a message to the user because your software SYSTEM is broken.

It is my opinion that there is no fourth option.

Our C#/VC++ brethren choose unchecked exceptions. Many "experts" think checked exceptions are bad: my contention is that life is difficult, get over it. Checked exceptions represent known failure modes: and have to be addressed. Your fishbone diagram has a straight lie for normal operation and branches off the side for failures. Checked exceptions are the anticipated failures.

Now, once you start handling Runtime exceptions, then it gets interesting. A Java program can run mostly normally with functions that do not work. By this, I mean they throw null pointer exceptions, array bounds errors, invalid arguments, and run out of heap space. This makes incremental delivery quite feasible.

(If you ever do catch Runtime errors, log them. Otherwise you never know to fix things)

Share Improve this answer Follow

answered Feb 11, 2009 at 0:51

Tim Williscroft

3,735 ● 25 ● 37