Introduction to error handling

You don’t have to have tried computer programming for long to discover that things go wrong.

In so many different ways.

In this part we’ll look at the ways that software can fail. We’ll see how to write code that is able not only to cope with the failure, but also communicate about what failures it is coping with. In the worked example we’ll refactor Java, exception-based error handling, into a more functional Kotlin style.

Why should we care about error handling?

Early in our careers Nat and I tended to gloss-over errors. We often still do, at least early in a project when we don’t know how things might fail. Most systems default to raising exceptions when something goes wrong, and catching and logging those exceptions at some outer level. Most command-line utilities will just exit in this case, hopefully having given enough information for the user to correct the problem and try again. A server app, or a GUI with an event-loop, will usually abort the current interaction and get on with the next.

Often this is fine, but sometimes the error will have corrupted the persistent state of the system, so that correcting the initial problem and retrying does not work. This is of course the source of the hackneyed advice to turn it off and on again - our systems mainly start in a safe state - so that after a restart a retry should succeed. If not, well you’ve probably been in a situation where the only solution has been to reinstall the operating system - the ultimate way of removing corrupted persistent state.

> **Rebooting the Internet**
>
> Duncan had a problem where the integration between his Nest thermostat and IfThisThenThat was not working. IFTTT was receiving notifications when the Nest entered home mode, but not away mode. The great AWS outage of 28th February 2017 mysteriously fixed the problem - it turns out that all it required was a reboot of the Internet.

If a system becomes successful, diagnosing and fixing corruption due to errors can expand to fill all the time available to a team. This is not a great place for a software project to be - ask us how we know!

How do programs go wrong?

Programs can go wrong for so many reasons!

Note that when we say *program* we also mean functions, methods, procedures - any code that we invoke.

- Sometimes they need to talk to other systems and that fails in some way.
- Often we don't give them the correct input that they need to do their job.
- We have heard of programmers making errors. Apparently they read past the end of arrays, or try to get the first item of an empty list.
- Sometimes the environment that we are running in fails for some reason, we might run out of memory, or not be able to load a class

There are failures that don't fit into these categories, but these cover most eventualities.

Error handling is hard to get right

Empirical studies have found that

- Error signals are frequently lost in large systems between where they are raised and where they can be handled
- Where error handling code is invoked, it often has defects that have catastrophic effects, but could be caught by simple unit tests.

"Without correct error propagation, any comprehensive failure policy is useless … We find that error handling is occasionally correct. Specifically, we see that low-level errors are sometimes lost as they travel through […] many layers […]" EIO: Error handling is occasionally correct. H. S. Gunawi, et al. In Proc. of the 6th USENIX Conference on File and Storage Technologies, FAST'08, 2008.

"Almost all catastrophic failures (92%) are the result of incorrect handling of non-fatal errors explicitly signaled in software" Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. Ding Yuan, et al., University of Toronto. In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI14, 2014

Why is error handling hard?

- We often don't know if and how an operation could fail
- Knowledge of how to handle an error can be a long way from where the error is detected
- Error handling is hard to isolate from its surrounding code and therefore hard to test
- Errors can leave our system in states from which it can't recover

TODO - expand on these

If something is hard work and error prone, we like the computer to do the drudge work, so that we can focus on the creative work

TODO - sidebar on this

We'll look at error handling in the context of these problems.

Pre-exception error handling strategies

Whilst we mostly signal errors these days with exceptions or special error types, other techniques have been used, and are still applicable in some circumstances.

Ignoring errors

We can ignore errors - either the failing routine does nothing to bring it to the attention of the caller, or the caller doesn't bother to check.

Returning a special value

Where possible, returning a special value to signify an error can be convenient. For example a, returning -1 instead of an index when an item is not found in a list.

This technique can only be used when not all of the range of return values are valid results for a function. It can also be dangerous, because the caller has to know (and remember) the convention. If we try to calculate the distance between two items in a list by subtracting their indices, when one of them is not found and returns -1, our calculation will be incorrect unless we explicitly to handle the special case. We are not able to lean on the type checker to help us avoid errors.

A special case of returning a special value is returning null on error. This is quite dangerous in most languages, because if the caller doesn't explicitly check for null, then using the result will throw a NullPointerException, which be worse than the initial problem. In Kotlin though, callers are forced to deal with null - an example of the type checker helping us to avoid errors - so this is a good technique.

Global flags

One problem with returning special values is that they make it hard to signal which of several possible errors occurred. To solve this we can combine the special value with setting a global variable. When the special value is detected the caller can read eg `errno` to establish what the problem was.

This technique was popular in C, but was largely superseded by exception-based error handling.

Returning a status code

Another technique from the days before exceptions is returning a status code. This is possible when a function either returns no value (it is entirely side-effect), or returns a value in another way, often by mutating a parameter passed by reference.

Invoking a special function

Even when exceptions are available, invoking a special function when an error occurs is sometimes a good strategy. Usually the function is passed as a parameter to the invoked function - if a problem is detected the error function is invoked with the issue as a parameter. Sometimes the error function is able to signal by its return value if the failed operation should be retried or aborted.

This technique is an example of the strategy pattern applied to error handling. Even when exceptions are available it is a useful tool to have mastered.

Continuation Passing Style

TODO - should we mention this? It bears comparison with folding over an error type.

Error handling with exceptions

All the above techniques suffer from the drawback that the calling code is able, to a greater or lesser extent, to ignore that an error occurred.

Exceptions solve this problem - the operation is aborted on error. - and the caller explicitly handles the exception. If the caller does not handle it, the exception propagates further up the call stack until someone does. If no handler is found the application exits.

Java and checked exceptions

Exceptions were relatively new when Java was released, and the language designers decided to innovate in this area. They made the exceptions that a function could throw part of its signature. This way callers could know that, for example, a method might fail because the network resource that it was reading was no longer available. If a method declared that it could fail in this way, then every caller would either have to deal with the failure (by catching it) or declare that it, too, was liable to fail with the same exception. This ensures that the programmer takes account of these errors. They are called checked exceptions, because the compiler checks that they are handled or redeclared.

Problems that we anticipate are only some of the possible reasons for failure though. The language designers identified two other types.

Errors

Subclasses of Error are reserved for failures so severe that the JVM can no longer guarantee the semantics of the language - maybe a class cannot be loaded or the system runs out of memory. These conditions could happen at any point in the execution of a program, and so could cause any function to fail. If any function could fail in this way there is little point in including such errors in the function signature, so they are exempted.

RuntimeExceptions

Subclasses of RuntimeException represent other errors. The intention was that these would be reserved for problems caused by programmer mistakes, such as accessing a null reference, or

trying to read outside the bounds of a collection. In both these cases the programmer could have been more careful, but again this potentially an issue for practically every piece of code, and so these are also exempted from having to be declared and handled.

In this scheme developers are forced to deal with operations that can fail due to IO or other things that are out of their control (the checked exceptions). This allows defensive programming where it is economical.

If a (capital E) Error occurs the best default approach is exit the process as quickly as possible, before any more damage can be done to persistent state.

RuntimeExceptions are a middle ground. If programmer error then we should probably assume that we have just proved that we don't really know what is going on in our program and abort. Otherwise we might try to recover, especially if our system has been designed to limit the damage that can be done by a single unexpected error.

We both really liked checked exceptions, but it seems we were in the minority, as they fell out of favour in Java over the years. They were hampered from the start by the odd decision to make the unchecked RuntimeException a subclass of the checked exception, so that code that wanted to handle all checked exceptions found itself catching unchecked ones as well, hiding programming errors. They were also not helped by the fact that the Java APIs used them inconsistently, with operations like `Integer.parseInt(String)` throwing the unchecked NumberFormatException, while `URL(String)` throws the checked MalformedURLException.

How should `parseInt` fail?

This is an interesting case, and goes to the heart of why error handling is so hard.

Looking through our strategies, it can't return a special integer value, because all the ints are spoken for. It could return null as a boxed Integer, but having to box and unbox for this, a really fundemental low-level operation that will be used in performance critical code, is undesireable, especially on the JVMs of the mid-1990s.

Invoking an error function would similarly involve inefficient ceremony, so we are left with throwing an exception. Should that exception be checked or unchecked?

The language designers decided that

- `parseInt` should throw NumberFormatException
- NumberFormatException should be an IllegalArgumentException, which is a RuntimeException and so unchecked.

Those are both reasonable decisions, but combined lead to `parseInt` not forcing its callers to consider that it might fail by declaring a checked exception.

I suspect that the JVM programmers were very used to parsing ints from char* in C, where there were no exceptions, and `atoi` returned 0 if it didn't work. They would have considered not planning for this failure to be a programmer error, rather than a failure of the function itself.

We will have more to say on the characterisation of errors later. TODO - make sure that we do.

Confusion begat confusion, and it wasn't long before the default was that the only checked exceptions that most Java libraries declared were IOExceptions. Even then database libraries such as Hibernate, which were definitely talking over the network and definitely subject to IOExceptions, would throw only RuntimeExceptions.

What should Hibernate have thrown?

Where a programmer explicitly invoked a Hibernate method to load an object, that method should declare IOException. If there are other ways that function could fail - maybe failure to parse a query - the method might also declare a checked exception to cover these. We might expect most Hibernate methods to declare both IOException and a checked HibernateException, with the latter having different subclasses for different failure modes.

Hibernate is an interesting case though because of lazy loading. If you load an object that contains a collection, Hibernate could be configured to load the contents of that collection only when it was accessed. So calling `Person.getContacts().size()` might go to the database. But `Collection.size()` doesn't declare that it throws `IOException`, so what is Hibernate to do?

It must throw an unchecked exception, but should that be an Error or a RuntimeException? Given that the JVM is almost certainly still perfectly serviceable, we are left with RuntimeException. But in this case this is a RuntimeException that is not the result of programmer error, in as much as there is no defensive action that could have been take to avoid it.

So perhaps Hibernate should have declared a checked HibernateException and an unchecked HibernateRuntimeException.

Once a good proportion of the code that you call just uses unchecked exceptions the game is up. You can't rely on checked exceptions to warn you about how a function might fail. Instead you are reduced to some tactical defensive programming and the age-old technique of putting it into production, seeing what errors you log, and adding code to handle those you don't like the look of.

The final nail in the coffin of checked exceptions was the introduction of lambdas in Java 8. All the Function interfaces designed to work with streams did not declare an exception type, and so cannot propagate checked exceptions. To be fair I would probably have given up there too.

This has been a cathartic rant.

It might be removed by the time you read the book, but if it hasn't, it is because we feel that it is important to make the point that we can aspire to better error handling than became the default in Java.

Kotlin and exceptions

Kotlin has exceptions, because it runs on the JVM, and exceptions are built into the ecosystem. It does not have checked exceptions, because Java had already lost that fight.

> You might ask how Kotlin can just disregard checked exceptions. The answer is that they are not a feature of the JVM, but rather of the Java compiler. The compiler does record in the bytecode what checked exceptions are declared by a method (in order for the compiler to to be able to check them) but the JVM itself does not care.

The result is that Kotlin programs are by default no better or worse than most Java programs when it comes to error handling.

> An exception ;-) to this is that, as we observed above, Kotlin can use null to indicate an error, safe in the knowledge that callers will have to write code to handle it.

Functional Error Handling

Statically-typed functional programming languages often use another error handling technique.

A distinguishing feature of functional programming is Referential Transparency. When this applies we can replace an expression with the result of its evaluation. So if we write

```
val secondsIn24hours = 60 * 60 * 24
```

then we can replace 60 * 60 with 3600 or 60 * 24 with 1440 without affecting the results. In fact the compiler may decide to replace the whole expression with 86400 for us, and unless we examine the bytecode or use a debugger we will be none the wiser.

In contrast

```
val dayLengthInHours = secondsIn(today()) / 60.0 / 60 / 24
```

is not referentially transparent, because `today()` will yield a different result than it did yesterday, and any day may have had a leap second applied.

Why should we care? Because referential transparency makes it a lot easier to reason about the behaviour of a program, which in turn leads to fewer errors and more opportunities to refactor and optimise. If we want these things (and at the very least we don't want more errors and fewer opportunities) then we should strive for referential transparency.

What does this have to with error handling? Let's return to our `Integer.parseInt(String)` example and see. For a given input, this will always return the same value, so it could be referentially transparent. But what for the cases where the String doesn't represent an integer? We can't replace the function invocation with an exception, because the type of the expression is `int`. Exceptions break referential transparency.

If instead of using exceptions we returned to the old trick of using a special value to represent errors, then we would have referential transparency, because that error value can replace the

expression. In Kotlin, null would be great here, but what if we needed to say which was the first character that wasn't a digit? That was information we could convey in an exception, but not in an return type of `Int?`.

Can we find a way for our function to return either the `int`, or the way that it failed?

The answer, as they say, is in the question. We define a type Either, which can hold one of two types, but only one at a time.

```
sealed class Either<out L, out R>

data class Left<out L>(val l: L) : Either<L, Nothing>()

data class Right<out R>(val r: R) : Either<Nothing, R>()
```

When used for error handling, the convention is that Right is used for a result, Left for an error.

If we stick to this convention we could define

```
fun parseInt(s: String): Either<String, Int> = try {
    Right(Integer.parseInt(s))
} catch (exception: Exception) {
    Left(exception.message ?: "No message")
}
```

How would we use this? As it is a sealed class, `when` expressions and smart casting work really nicely to let us write things like

```
val result: Either<String, Int> = parseInt(readLine() ?: "")
when (result) {
    is Right -> println("Your number was ${result.r}")
    is Left -> println("I couldn't read your number because ${result.l}")
}
```

which admittedly is pathologically not functional, but gives the general idea. By returning an Either we force our clients to deal with the fact that we may have failed - in effect we have reproduced some of checked exceptions in a functional form. To embrace this style you make all functions that might ordinarily throw an exception return Either, and when they in turn invoke something that could fail, pass on any failure or unwrap the success and act on it.

```
fun doubleString(s: String): Either<String, Int> {
    val result: Either<String, Int> = parseInt(s)
    return when (result) {
        is Right -> Right(2 * result.r)
        is Left -> result
    }
}
```

Whilst using `when` to unwrap an Either is handy, it quickly gets old, so we write

```
inline fun <L, R1, R2> Either<L, R1>.map(f: (R1) -> R2): Either<L, R2> =
    when (this) {
        is Right -> Right(f(this.r))
        is Left -> this
    }
```

which allows us to write the previous function as

```
fun doubleString(s: String): Either<String, Int> = parseInt(s).map { 2 * it }
```

Why is that function called `map` and not `invokeUnlessLeft`? Well if you squint you may be able to see that it is kind of the same thing as `List.map`. Practice that squinting, because we are now going to define

```
inline fun <L, R1, R2> Either<L, R1>.flatMap(f: (R1) -> Either<L, R2>): Either<L, R2>
=
    when (this) {
        is Right -> f(this.r)
        is Left -> this
    }
```

This unpacks our value and uses it to invoke a function that in turn might fail (as it returns Either). What can we do with that? Well lets say we want to read from a Reader and print double the result.

```
fun BufferedReader.eitherReadLine(): Either<String, String> =
    try {
        val line = this.readLine()
        if (line == null)
            Left("No more lines")
        else
            Right(line)
    } catch (x: IOException) {
        Left(x.message ?: "No message")
    }


fun doubleNextLine(reader: BufferedReader): Either<String, Int> =
    reader.eitherReadLine().flatMap { doubleString(it) }
```

This code will return a Left with the failure if `eitherReadLine` fails, otherwise it will return the result of `doubleString`, which may itself be either a Left for failure, or a Right with the final `int` result. In this way a chain of map and/or flatMap calls acts like a series of expressions which might throw an exception - the first failure aborts the rest of the computation.

If you come from an OO background this style does take some getting used to. No amount of reading helps - you just have to knuckle down and start writing code this way until it becomes less painful.