Error Handling

Avoiding Errors

Like a photographer who becomes more interested in cameras than taking pictures, it is easy to become more engrossed in the techniques of error handling than the end result. Duncan in particular has spent way too long just not listening to what the pain of trying to cope with errors was telling him.

The truth is that handling errors is itself tedious and error-prone - and we could all do with less tedium and fewer errors. Exceptions as a strategy are less tiresome than other techniques, but as they allow us to sweep possible problems under the carpet, we either get a less reliable system, or we have to lift a lot of carpets. Result types can be tiresome in code, but they help the type checker tell us what can fail and how. Checked exceptions were a valiant effort to get the best of both worlds, but in the end must be considered a failed experiment that are in any case not available to Kotlin developers.

Errors are particularly pernicious in code because they are transitive. If a function relies on code that can fail then it either has to have a strategy for succeeding despite the failure of its subordinate, or it is itself subject to failure. Pretty soon any moderately complex functionality has a combinatorial explosion of ways that it can fail. We then end up adding code to reason with the possible errors just so that a function can fail with an error that makes sense to its callers. And that error handling code itself is subject to errors!

If we rely on exceptions for our error handling then the net effect is that most complex code paths can effectively throw any exception, and without checked exceptions we have easy no way to work out which. Even if we do work it out, then just swapping out one function for another can invalidate the analysis. We will look later at using a Result type with sealed classes to represent errors within a bounded context, which can restore some of the benefits of checked exceptions, but doesn’t change the transitive nature of failure.

We can’t mitigate this problem completely, but it would be good if it applied to less of our codebase. Our aim should be to reduce the number of functions that can fail. In particular, if we can do that for lower-level functions, then the higher level functions that call them will also become less subject to failure, which in time should significantly reduce the proportion of functions that have to consider error cases at all. The boundary of code that is subject to failure will move out through the layers. This seems like a far better outcome than Duncan’s previous experience, which was that learning better error handling techniques just allowed him to tolerate the pain all over, instead of encouraging him to expand the pain-free zones.

Avoiding Partial Functions

The most common reason that a function may not be able to return normally is that it is only able to give an output for some of the possible inputs.

We previously looked at

```
fun parseInt(s: String): Int = SOMECODE()
```

This is only able to return an Int for the subset of strings that represent valid integers (and not actually most of those).

In contrast

```
fun length(s: String): Int = SOMECODE()
```

is defined for all strings.

We say that `parseInt` is a partial function - it is defined only for a restricted range of its inputs. `length` is a total function - it can yield a result for every input.

Note that any Java method that throws a NullPointerException when passed a null argmument is by this definition a partial function. By introducing nullablity into the typesystem Kotlin drastically reduces the proportion of partial functions. Looked at through this lens, when we use a dynamic-typed language, we give up all hope that any function is total.

Is it a programmer error to call a partial function with inputs that would make it fail, in which case we should throw a RuntimeException? As seen from our previous discussion around `parseInt` this is nuanced, but exceptions of some kind are the accepted way of signalling that out-of-range inputs have been supplied, and so the function cannot return a result.

One way of making an otherwise partial function total is to widen its return type. If we define

```
fun parseInt(s: String): Result<Int, String> =
    try {
        Success(Integer.parseInt(s))
    } catch (x: NumberFormatException) {
        Failure("Could not parse %s as int")
    }
```

then `parseInt` can now return a Result for the strings that would previously have been out of its range. This achieves the desired effect of forcing the caller to consider the error case, but doesn't make our function not subject to errors. I suppose it can reduce the errors in our error handling, but it doesn't prevent the transitive error handling expansion that we noted earlier. So let's not count introducing returning a special value as avoiding errors, and for the purposes of this book consider a total function one that can return a non-error for all combinations of its parameters.

The other way to make an otherwise partial function total is to use the typesystem to ensure that it can only be passed valid arguments. Let's say that we define

```
data class ShortDigitString(
    val value: String
) {
    init {
        check(
            value.length < 10 &&
            value.all { it.isDigit() }
        )
    }
}
```

Now we could define `parseInt`

```
fun parseInt(s: ShortDigitString) =
    Integer.parseInt(s.value)
```

Here we have used logic to deduce that the base Integer.parseInt cannot fail it we only pass it up to 9 characters all of which are digits. If we are wrong, then the consequences are that we leak a RuntimeException, which is consistent with our policy that these signal when a programmer has made a mistake.

By introducing a new type we have narrowed the possible inputs to `parseInt` until it is able to succeed for all of them. You may see this as just validation. It *is* validation, but it isn't *just* validation, as we are propagating the guarantee that validation has been performed using the typesystem.

Does this help? Well if we just writing up writing

```
fun numberOfMonths(years: String) =
    12 * parseInt(ShortDigitString(years))
```

then no, as this is as just as susceptible to failing with a NumberFormatException as the raw `Integer.parseInt`.

But if we push the parsing out a level

```
fun numberOfMonths(years: ShortDigitString) =
    12 * parseInt(years)
```

then yes - now we have a function that isn't subject to failure, and doesn't taint its callers with the failure too. If all its callers still have to create the ShortDigitString we haven't bought much, but push the creation out through the layers and we can begin to turn the tide.

You probably don't do anything this gratuitous. Who would pass around strings that mean integers? But you might pass around a string that represent a URI and subject layers of invocations to the potential failure when it is finally parsed. You have probably passed a string rather than a File, or a

File rather than an ExistingFile, or an Int rather than a PositiveInt.

Collections of potential parameters are particularly pernicious. Take for example an HTTP request object passed through several layers of code in order that a customer id can be extracted from the path, and a date from the query string. Not only are all the intervening layers subject to failure, but we loose the opportunity to document the required shape of the data, and the tests for each layer have to construct complicated arguments rather than passing CustomerId and LocalDateTime.

How Kotlin Helps

You can avoid partial functions in any language - Java isn't particularly at a disadvantage here. Kotlin has features that are particularly helpful though.

We've already seen how nullability in Kotlin makes functions that would otherwise be partial total, because null cannot be passed as an argument. Another pertinent feature is data classes, because, as with the ShortDigitString above, they allow us to produce wrapper value classes, with validation, in very few lines of code, reducing the programmer overhead of this technique. Unfortunately the runtime overhead remains, but in practice you will have trouble detecting it in most code.

Inline Classes

Inline classes would seem to be ideal to represent validated versions of more general types, especially strings, as they don't require the creation of a wrapper object.

In practice though they cannot have private constructors, or init blocks, and so you cannot guarantee that an inline class has validated content.

Sealed classes are also helpful in avoiding partial functions. By restricting subtyping they allow the receiver of a sealed parameter type to account for all possibilities. So, using Result as an example

```
val Result<*, Exception>.errorMessage : String? get() =
    when (this) {
        is Success -> null
        is Failure -> this.reason.message ?: "no message"
    }
```

Here the receiver must be considered a parameter to the function. Because Result is a sealed class, the code knows that Success and Failure are the only possibilities. For non-sealed classes it could only invoke methods on the base Result type, or be partial because another subtype might be passed that it could not cope with.

Suggested Types

NonZeroInt

NonEmptyList

Map<K, V, PhantomType>