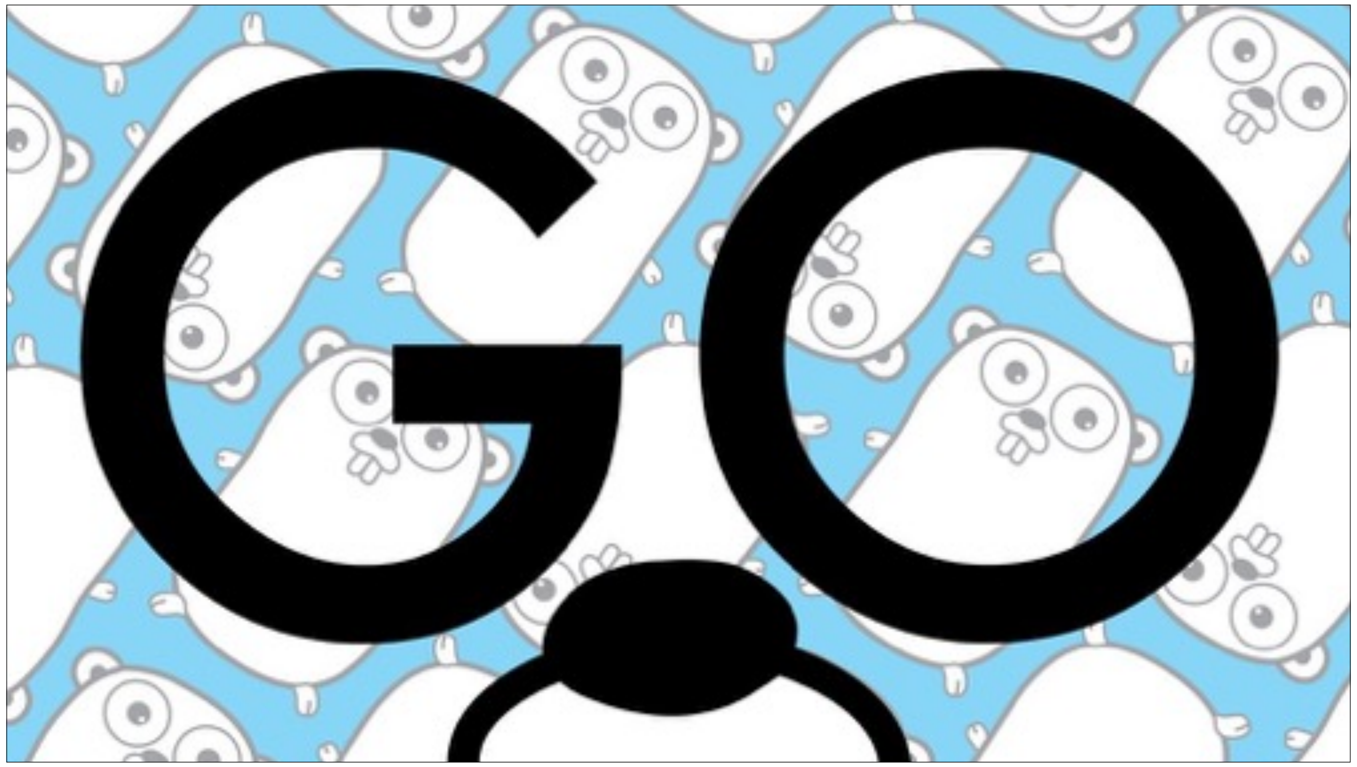
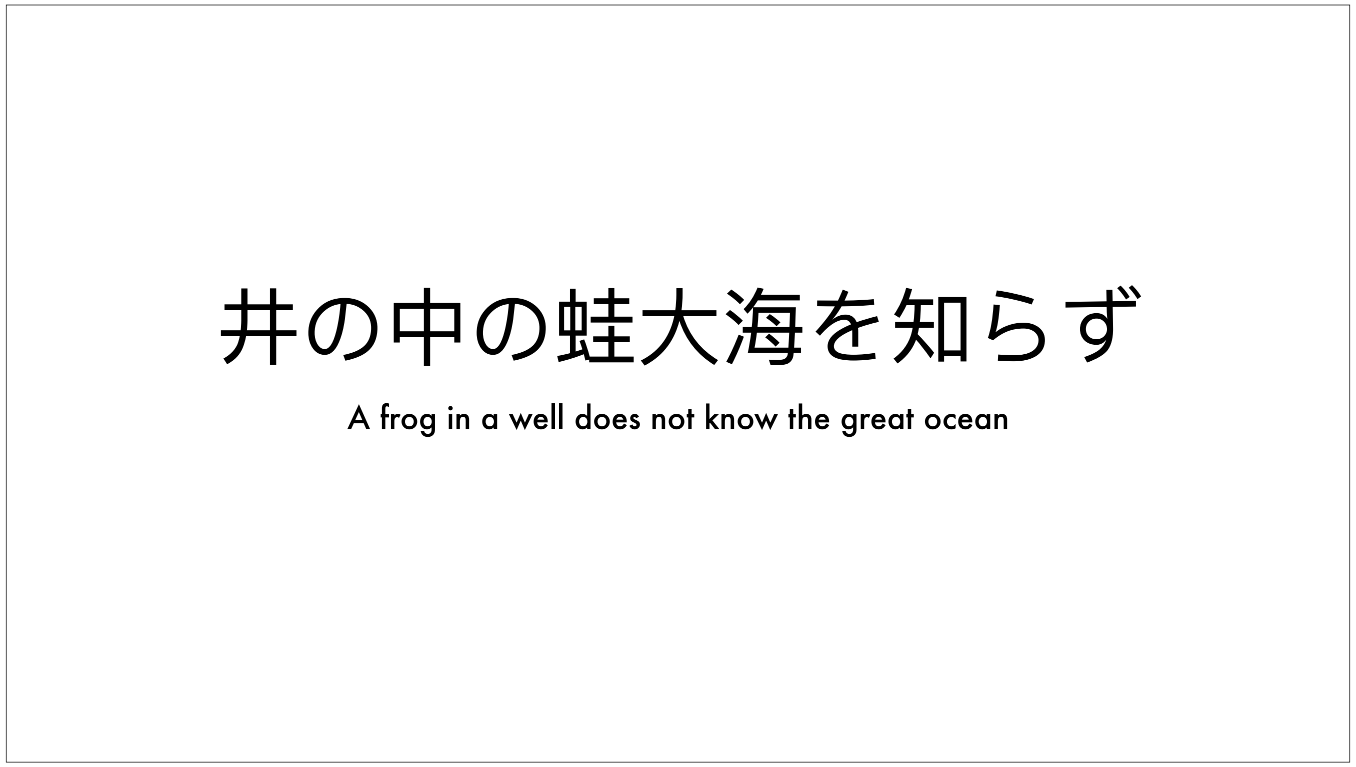


Good morning! Thank you for coming to my talk.



井の中の蛙大海を知らず

A frog in a well does not know the great ocean

There is a popular Japanese proverb, adapted from a Chinese fable, who’s english translation goes something like this.

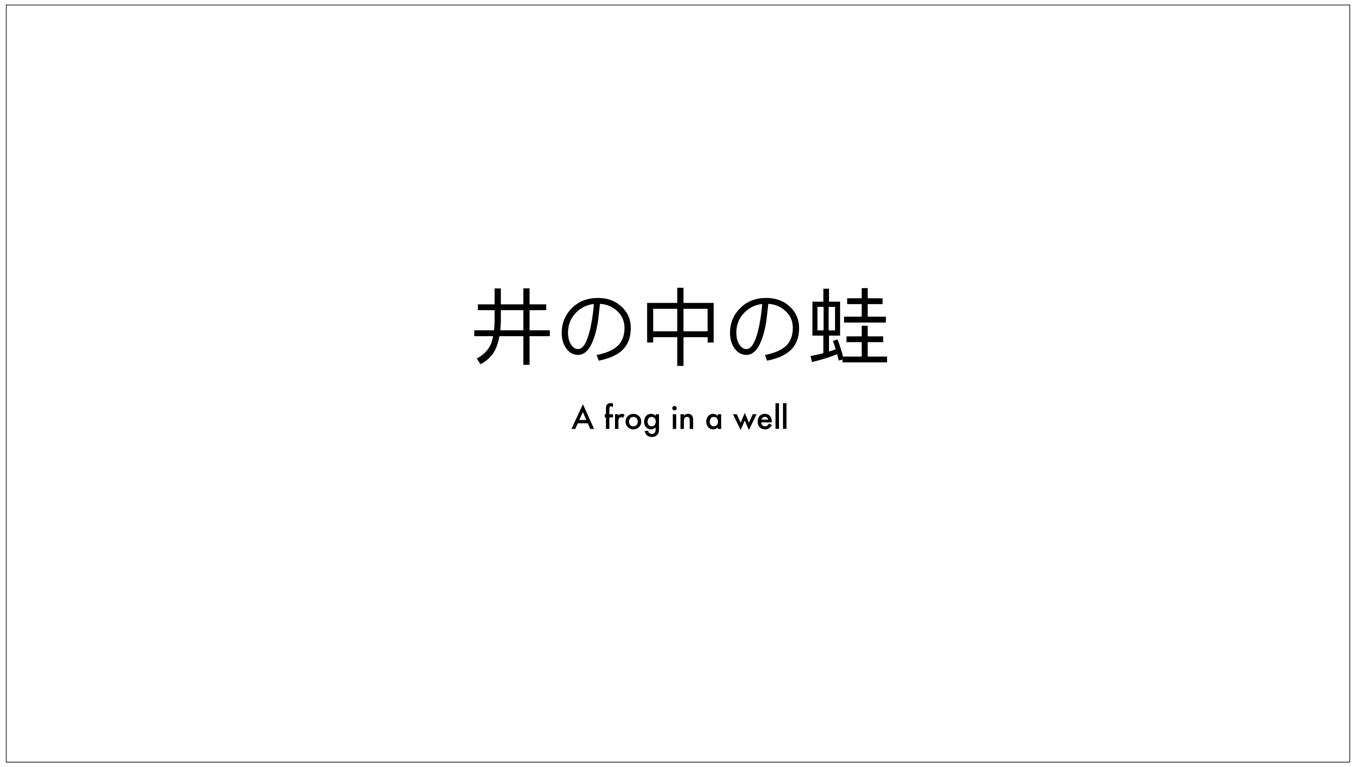
A frog in a well does not know the great ocean.

The frog lived down in a well where there was all he had to live. One day, a softshelled turtle came by and told him about the sea. 'The sea? Hah!’, scoffed the the Frog, 'It’s paradise in here. Nothing can be better than this well.'

What is this proverb trying to teach us ?

If you are the frog, maybe think you know all there is to know. You know your surroundings intimately, yet you are unaware of just how small your home is compared to the world outside, and you don’t know how limited your knowledge is.

The lesson is; question your environment, question your assumptions, don't assume you are right just because nobody has convinced you that you are wrong.



井の中の蛙

A frog in a well

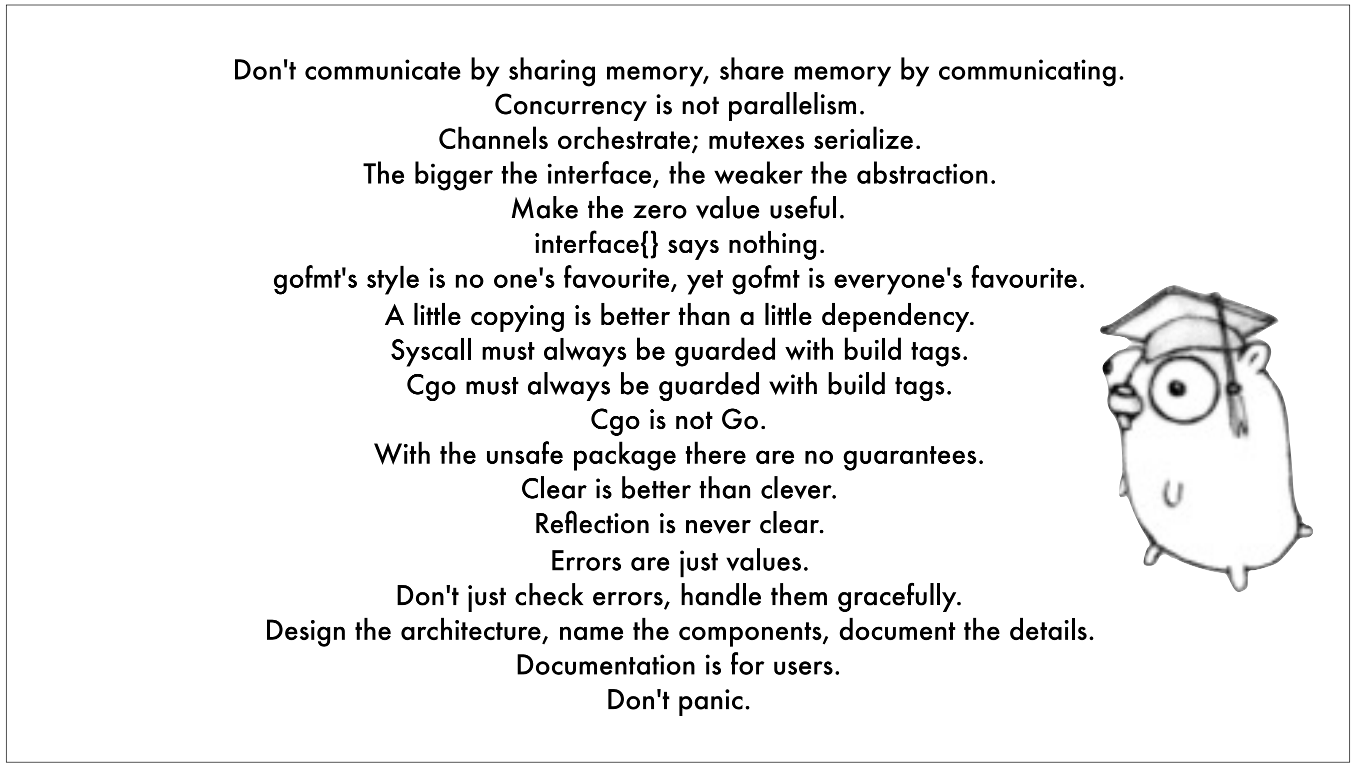
Proverbs such as these are very popular in Japan, so popular they often have shorter forms, So if someone said to you

A frog in a well

maybe you understand the advice, and if so, you understand it not from the words *a frog in a well*, but because you understand the meaning behind those words

Last year Rob Pike gave a talk at GopherFest entitled "Go Proverbs” inspired by the english translation of Se-go Ken-sa-ku’s famous book on the game of Go.

In his talk, Rob Pike asked, "Are there Go proverbs?", and answer is yes, there are indeed Go proverbs



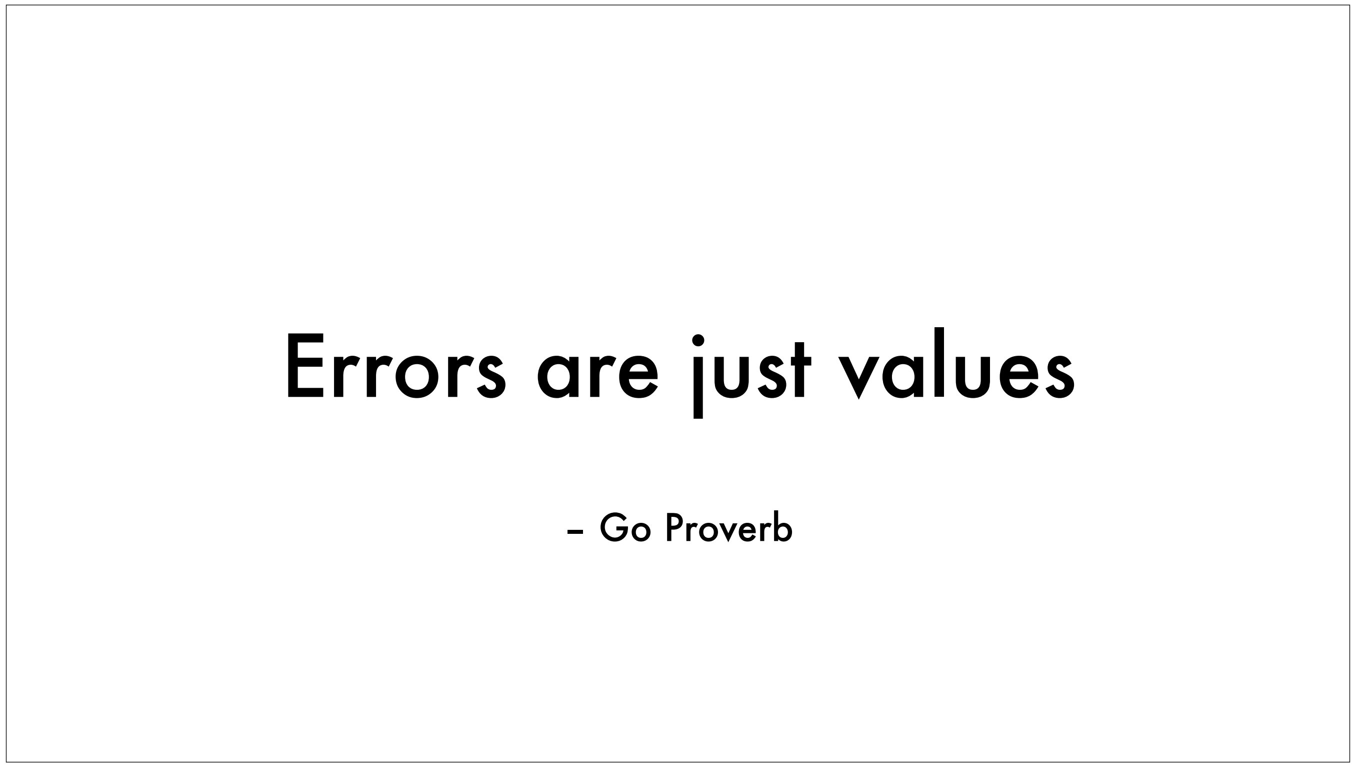
Don't communicate by sharing memory, share memory by communicating.
Concurrency is not parallelism.
Channels orchestrate; mutexes serialize.
The bigger the interface, the weaker the abstraction.
Make the zero value useful.
interface{} says nothing.
gofmt's style is no one's favourite, yet gofmt is everyone's favourite.
A little copying is better than a little dependency.
Syscall must always be guarded with build tags.
Cgo must always be guarded with build tags.
Cgo is not Go.
With the unsafe package there are no guarantees.
Clear is better than clever.
Reflection is never clear.
Errors are just values.
Don't just check errors, handle them gracefully.
Design the architecture, name the components, document the details.
Documentation is for users.
Don't panic.

… you probably recognise them.

And just like the story of the frog, to understand these Go proverbs, it is not sufficient to simply memorise the words, you must understand the lesson each proverb teaches.

I don’t time today to discuss every one of the proverbs.

Instead I want to talk about one aspect of Go's design, and relate these ideas to some of the Go proverbs as I understand them.



Errors are just values

– Go Proverb

What do Go programmers mean when they say to each other "errors are just values"?

When we say "errors are just values", what we actually mean is

"any value that implements the `error` interface is itself an error",

but saying "errors are just values" to someone who is still a student of Go is not useful.

To understand this proverb, the student must understand the underlying message it tries to teach.

This is a good place to start the discussion on what I think it means to understand Go's error handling philosophy.

Programming with errors

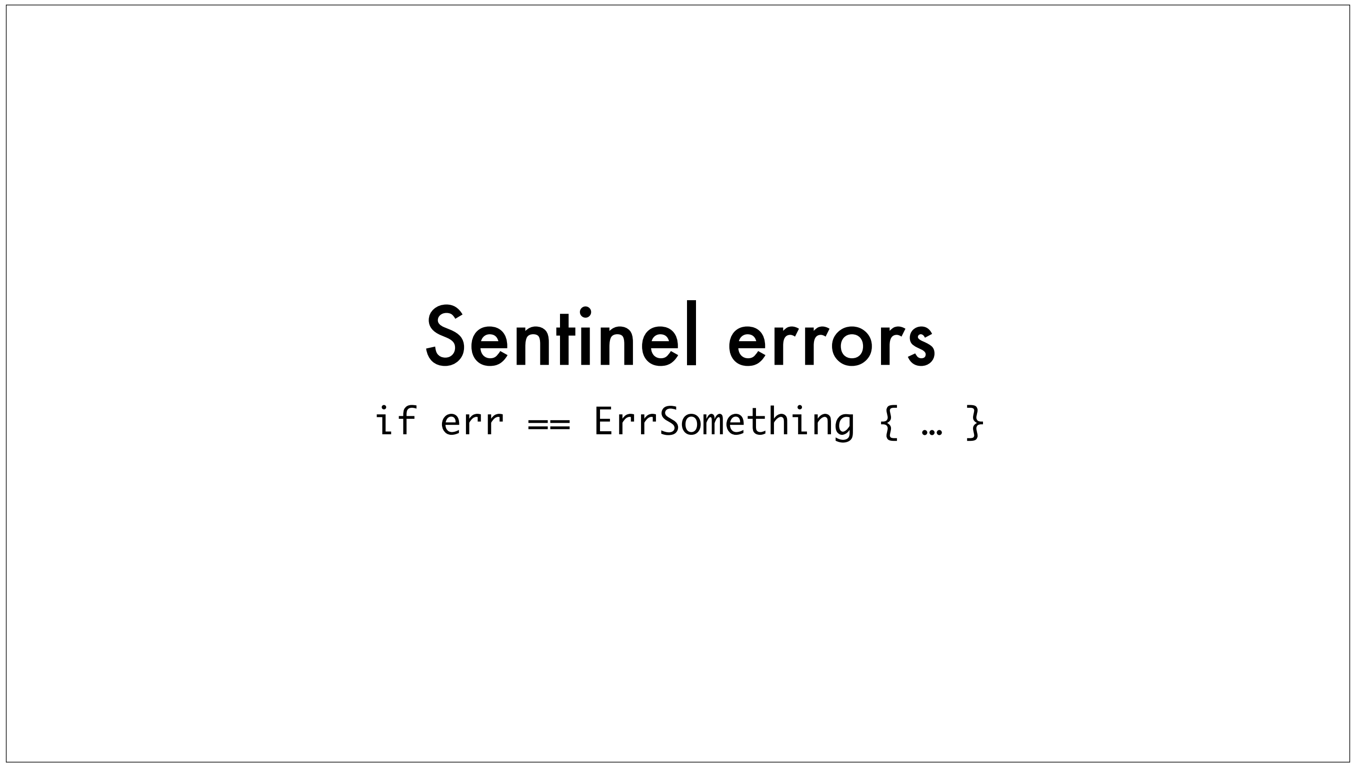
I've spent a long time thinking about the best way to handle errors in Go programs.

I really wanted there to be a single way to do error handling, something that we could teach all Go programmers by rote, just as we teach the mechanics of slices, or the rules of the languages' syntax.

Unfortunately, I'm pretty sure that there is no one single way to handle errors.

However, while there may be no single methodology for every error condition,

Instead, I believe Go's error handling can be classified into the three core strategies.



Sentinel errors

```
if err == ErrSomething { … }
```

The first category of error handling is what I call *sentinel errors*.

The name descends from the practice in computer programming of using a specific value to signify that no further processing is possible.

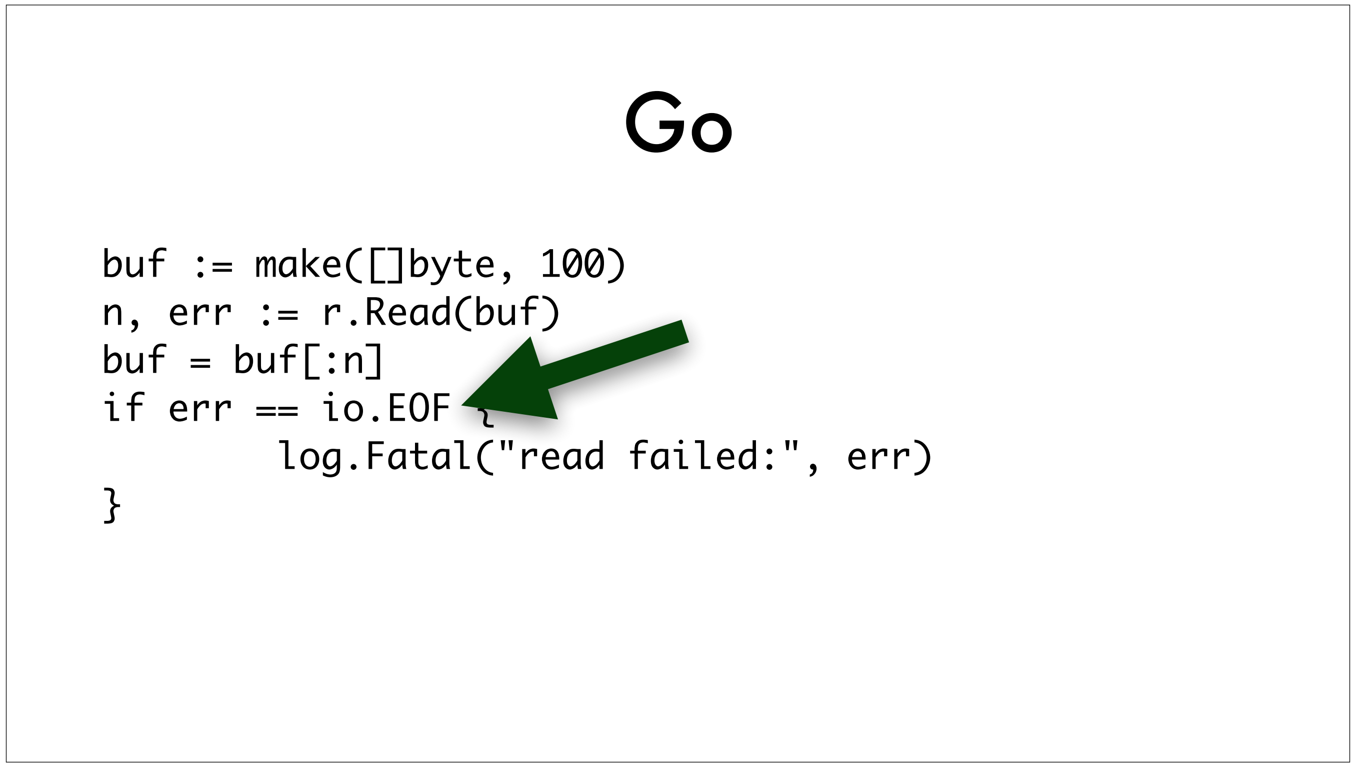
And so to with Go, we can use specific values to signify an error.

Sentinel error examples

```
io.EOF

syscall.ENOENT

go/build.NoGoError

path/filepath.SkipDir
```

Examples include values like `io.EOF`. or low level errors like the constants in the syscall package.

There are even sentinel errors that signify that an error *did not* occur, like `go/build.NoGoError` and `path/filepath.SkipDir` from filepath.Walk



Go

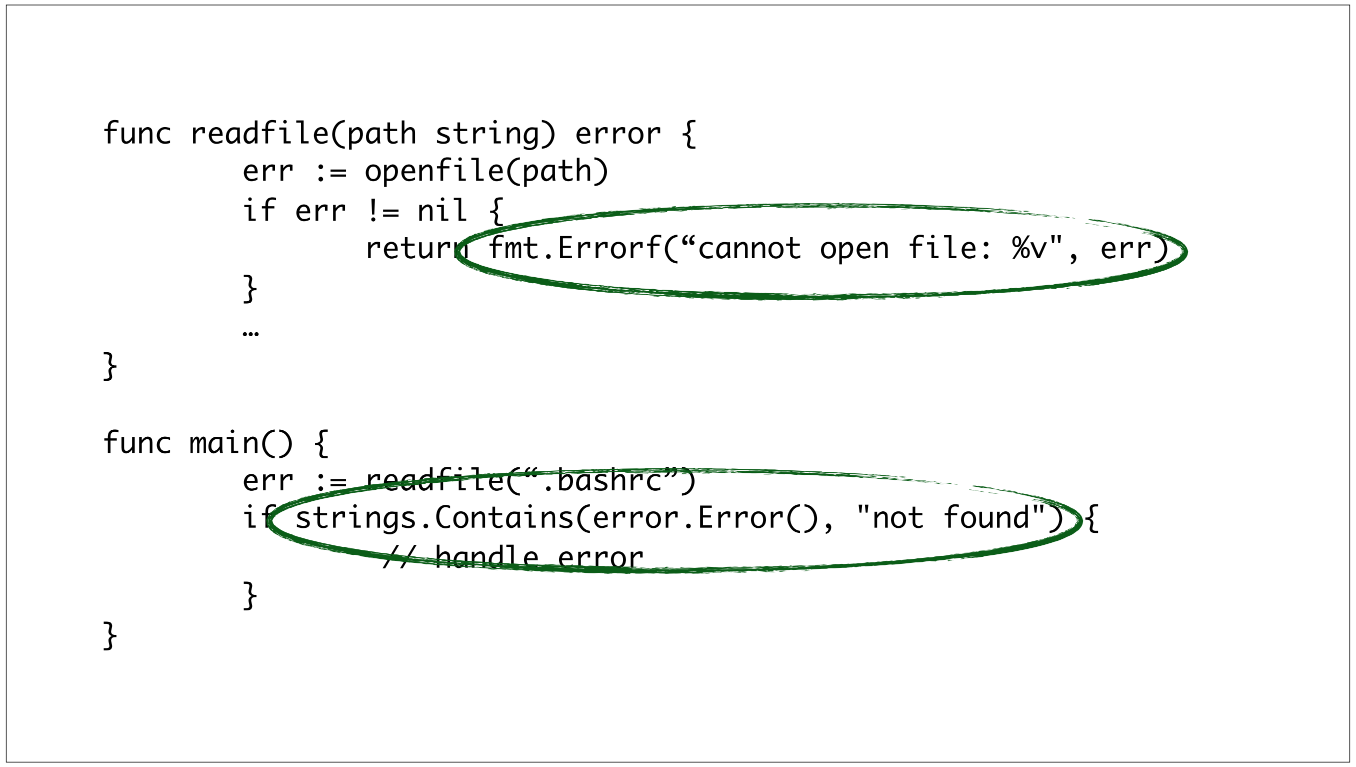
```
buf := make([]byte, 100)
n, err := r.Read(buf)
buf = buf[:n]
if err == io.EOF {
        log.Fatal("read failed:", err)
}
```

Using sentinel values is the least flexible error handling strategy.

The caller must compare the result to pre declared value using the equality operator.

[click]

This presents a problem when you want to provide more context, as returning a different error would will break the equality check.



```
func readfile(path string) error {
        err := openfile(path)
        if err != nil {
                return fmt.Errorf("cannot open file: %v", err)
        }
        …
}

func main() {
        err := readfile(".bashrc")
        if strings.Contains(error.Error(), "not found") {
                // handle error
        }
}
```

Even something as well meaning as using fmt.Errorf to add some context to the error will defeat the caller's equality test.

Instead the caller will be forced to look at the string output of the error's `Error` method to see if it matches a specific string.

Never inspect the output of error.Error()

💣

As an aside, i believe you should *never* inspect the output of the `error.Error` method because the `Error` method exists for humans, not code.

The contents of that string belong in a log file, or displayed on screen. You shouldn’t try to change the behaviour of your program by inspecting that error string

I know that sometimes this isn't possible, and this advice is less absolute when writing tests.

Never the less, comparing the string form of an error is, in my opinion, a code smell, and you should try to design your programs to avoid needing to do this.

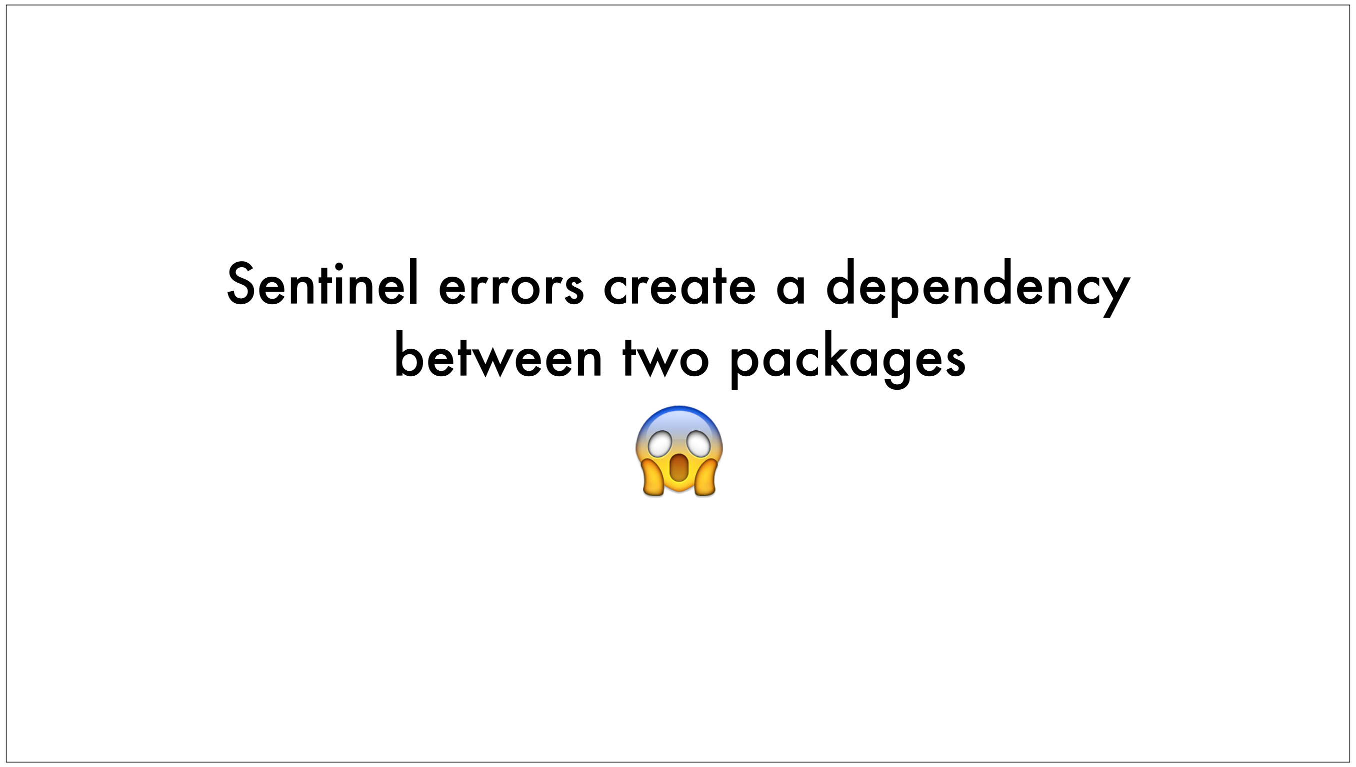
Sentinel errors become part of your public API

Sentinel errors have other drawbacks. For example

If your public function or method returns a particular sentinel value, then that value must be public, and of course documented. This adds to the surface area of your API.

Also, if your API defines an interface which returns a specific sentinel error, all implementations of that interface will then be restricted to returning only that error, even if they *could* provide a more descriptive error.

We see this with io.Reader. Functions like io.Copy *require* a reader implementation to return exactly io.EOF to signal to the caller “there is no error, there is just no more data”



Sentinel errors create a dependency between two packages

😱

But the worst problem with sentinel error values is they create a dependency between two packages.

To check if an error is equal to `io.EOF`, obviously the calling code must import the `io` package.

This specific example does not sound so bad, because it is so common.

But imagine the coupling that exists when many packages in your project export the own set of error values; which other packages in your project must import to check for specific error conditions.

Having worked in a large project that toyed with this pattern, i can tell you from experience that the spectre of bad design--in the form of an import loop--was never far from our minds.

Conclusion: avoid sentinel errors

👎

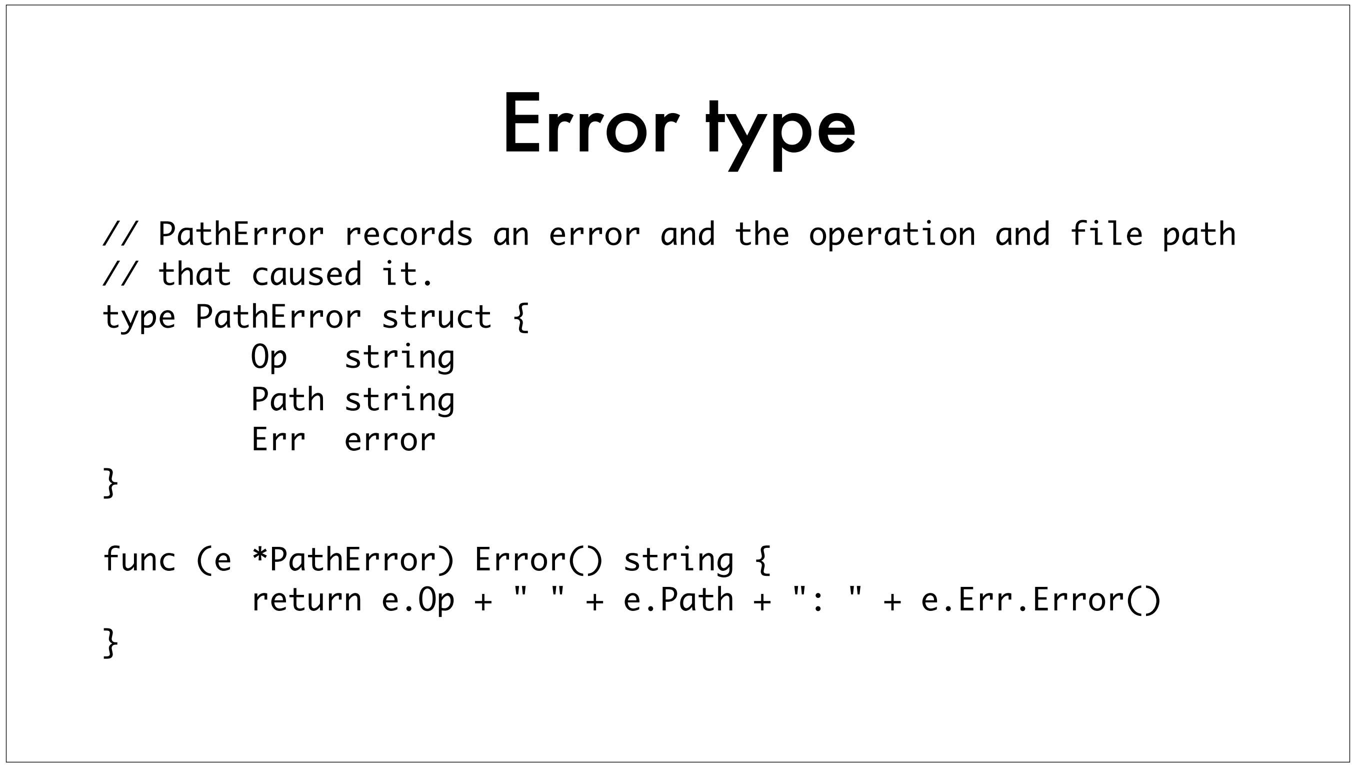
So, my advice to you is to avoid using sentinel error values in the code you write. There are a few cases where they are used in the standard library, but this is not a pattern i think you should emulate.

If someone asks you to export an error value from your package, i think you should politely decline, and instead suggest an alternative method, such as the ones I'm going to discuss next.

Error Types

```
if err, ok := err.(SomeType); ok { … }
```

Error types are the second form of Go error handling I want to discuss.

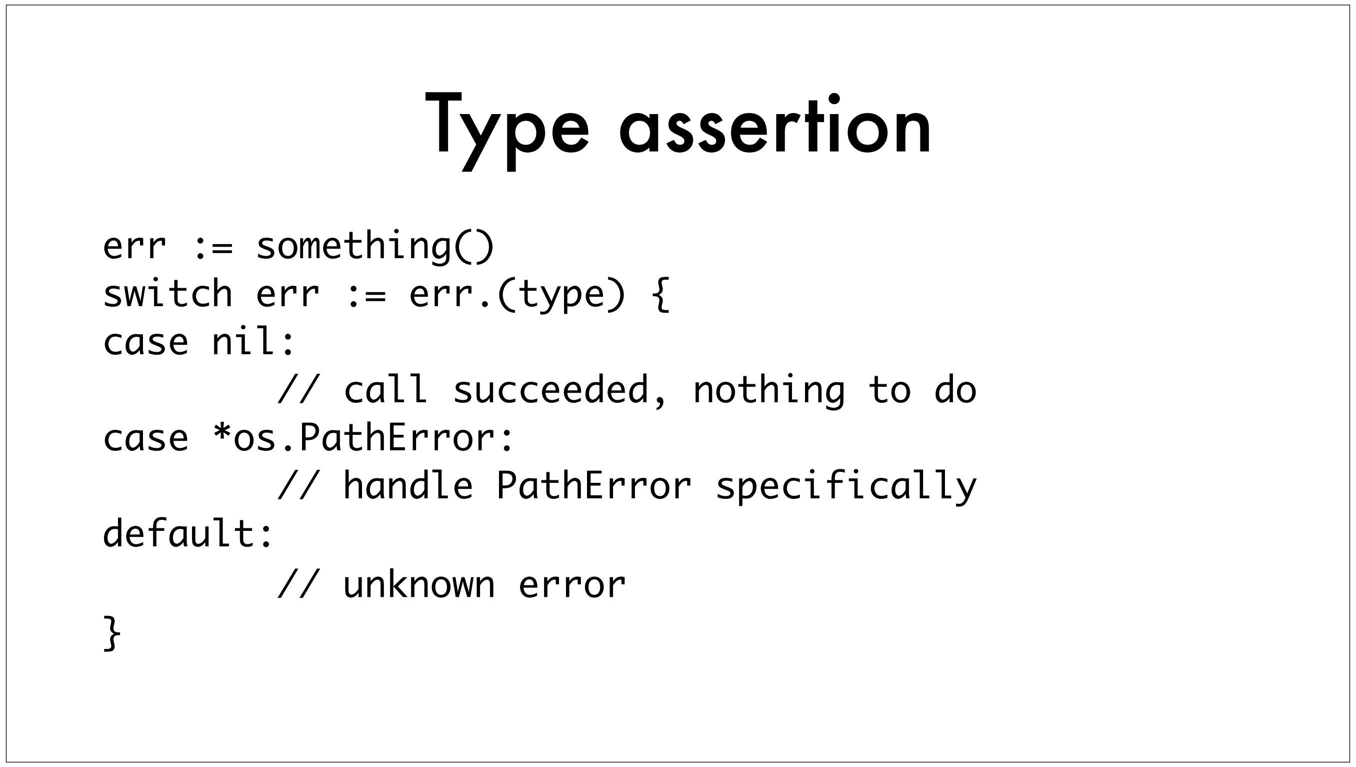


Error type

```go
// PathError records an error and the operation and file path
// that caused it.
type PathError struct {
        Op   string
        Path string
        Err  error
}

func (e *PathError) Error() string {
        return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

An error type is a type that you create that implements the error interface.

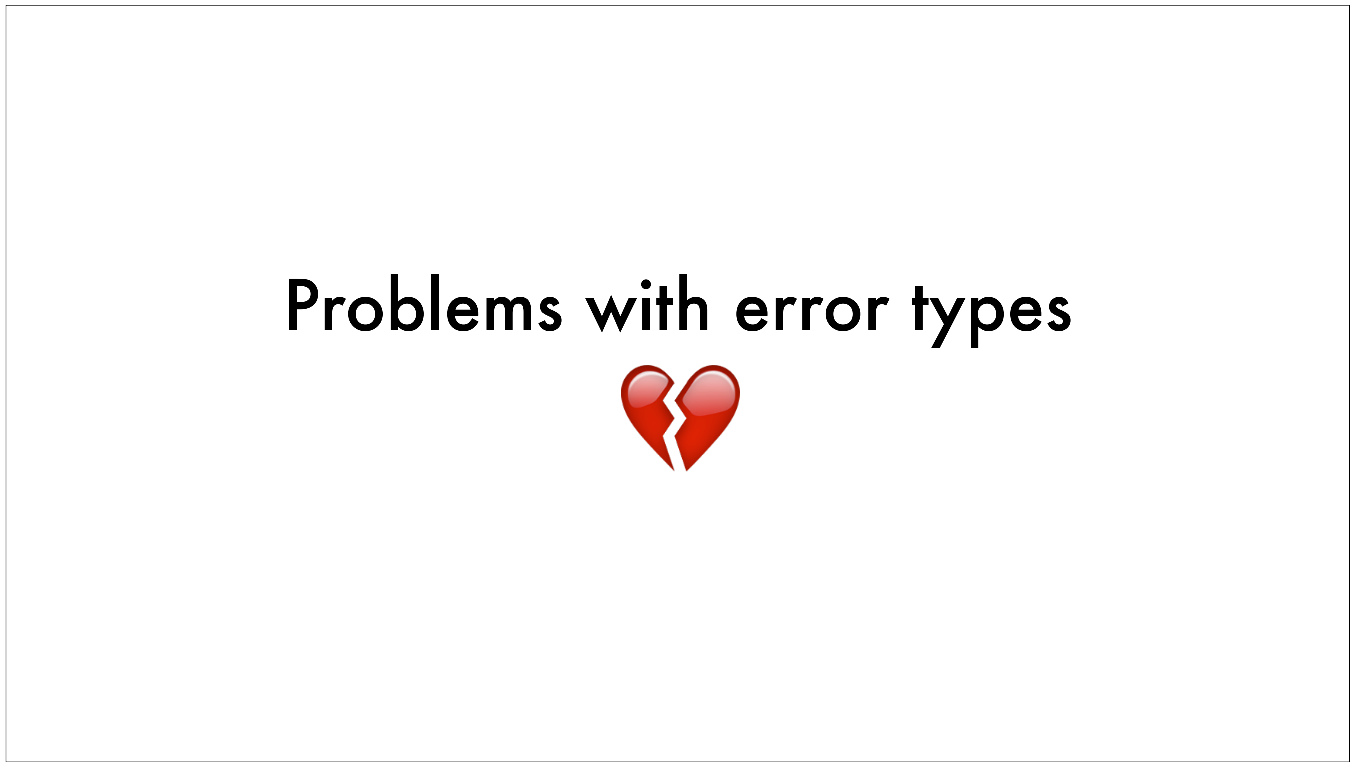
Here's an example of the os.PathError type which tracks the underlying error of a file operation as well as the path and the operation that was being attempted at the time of failure.



Type assertion

```
err := something()
switch err := err.(type) {
case nil:
        // call succeeded, nothing to do
case *os.PathError:
        // handle PathError specifically
default:
        // unknown error
}
```

Because os.PathError is a public type, callers can use type assertion to extract the extra context from the error.



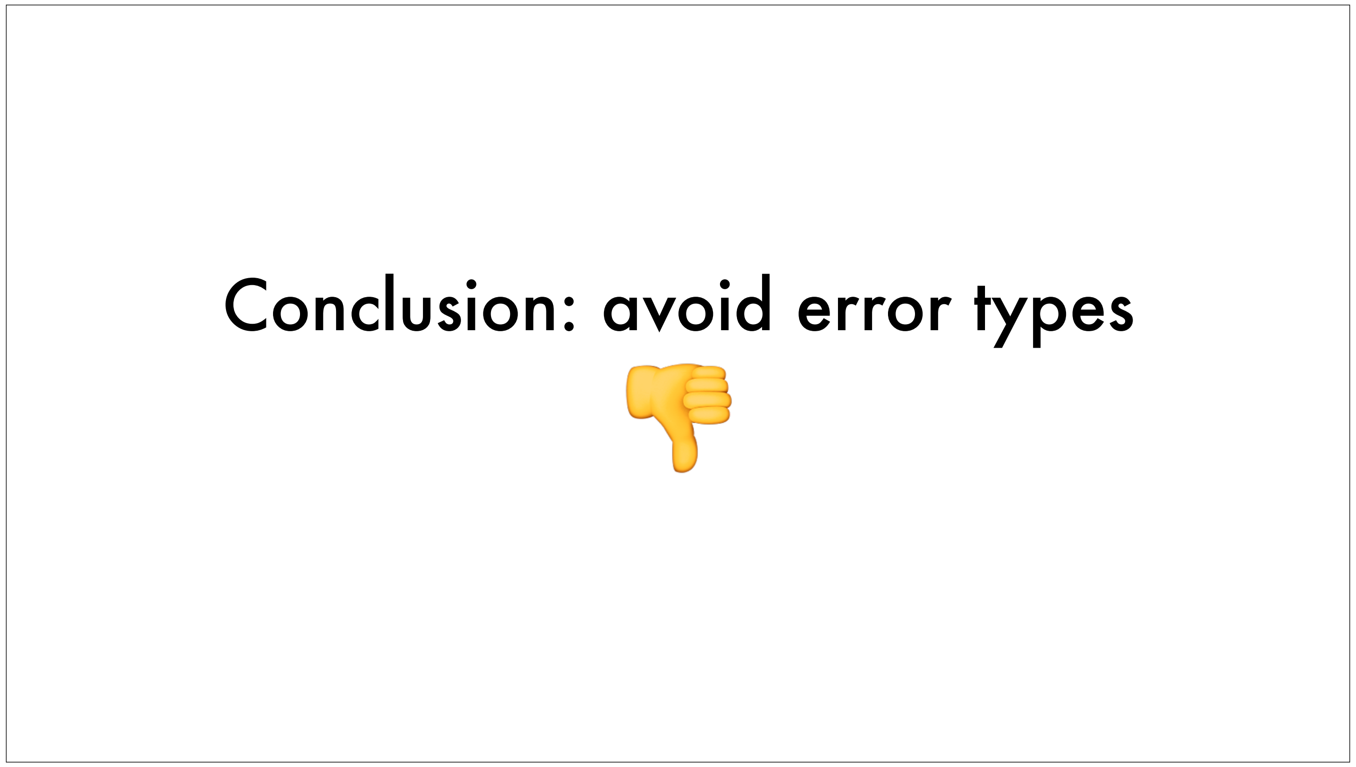
Problems with error types

💔

However, error types must be made public, so the caller can use a type assertion or type switch.

If your code implements an interface who's contract requires a specific error type, all implementors of that interface need to depend on the package that defines the error type.

This intimate knowledge of a package's types creates a strong coupling with the caller, making for a brittle API.



Conclusion: avoid error types

👎

So, while error types are better than error values, because they can capture more context about what went wrong, error types share many of the problems of error values.

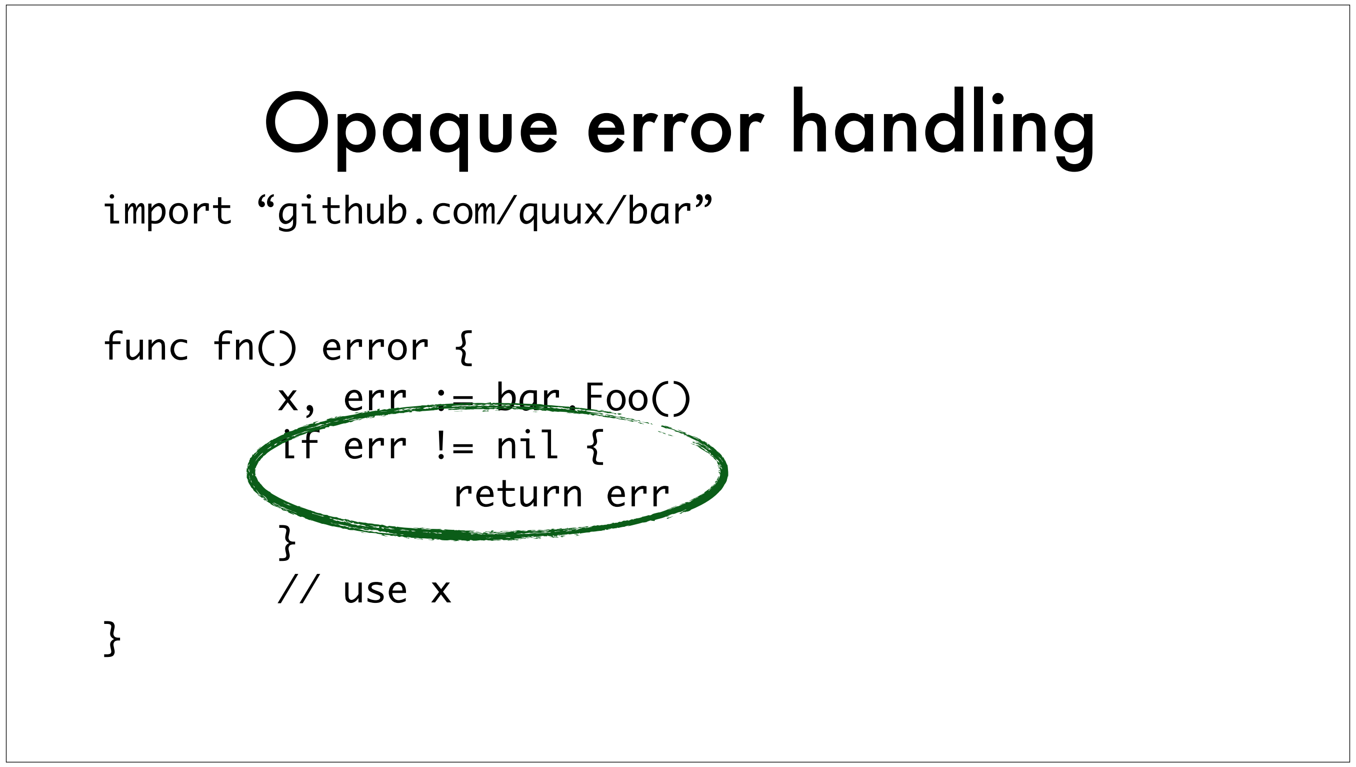
So again, my advice is to avoid error types, or at least, avoid making them part of your public API.

Opaque errors

Now we come to the third category of error handling. This is, in my opinion, the most flexible error handling strategy as it requires the least coupling between your code and caller.

I call this opaque error handling, because while you know an error occurred, you don’t have the ability to see inside it.

As the caller, all you know about the result of the operation is that it worked, or it didn’t.



Opaque error handling

```
import “github.com/quux/bar”


func fn() error {
        x, err := bar.Foo()
        if err != nil {
                return err
        }
        // use x
}
```

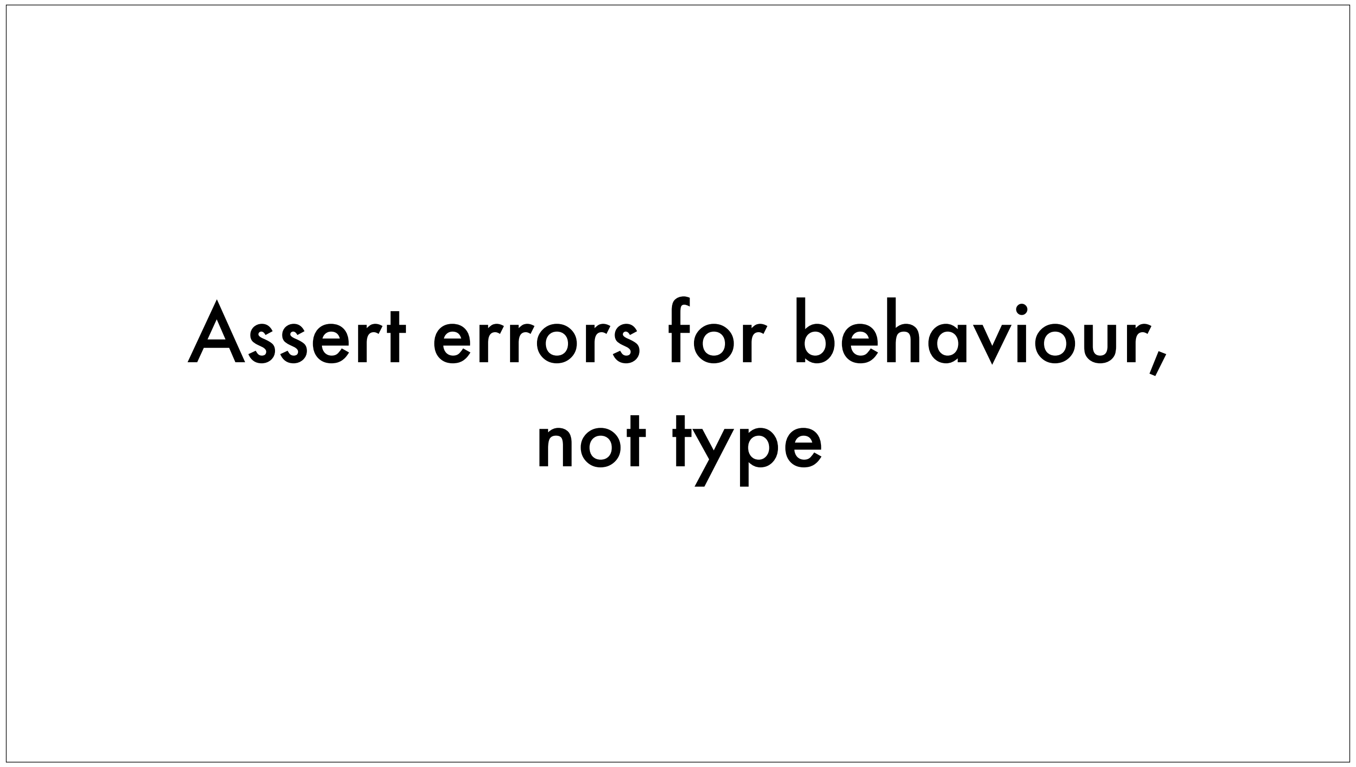
If you adopt this position, then error handling can become significantly more useful as a debugging tool.

For example, bar.Foo’s contract makes no guarantees about what it will return in the context of an error.

So the author of bar.Foo can now annotate errors that it produces with additional context without breaking its contract with the caller.

[click]

This is all there is to opaque error handling, just return the error without assuming anything about its contents.

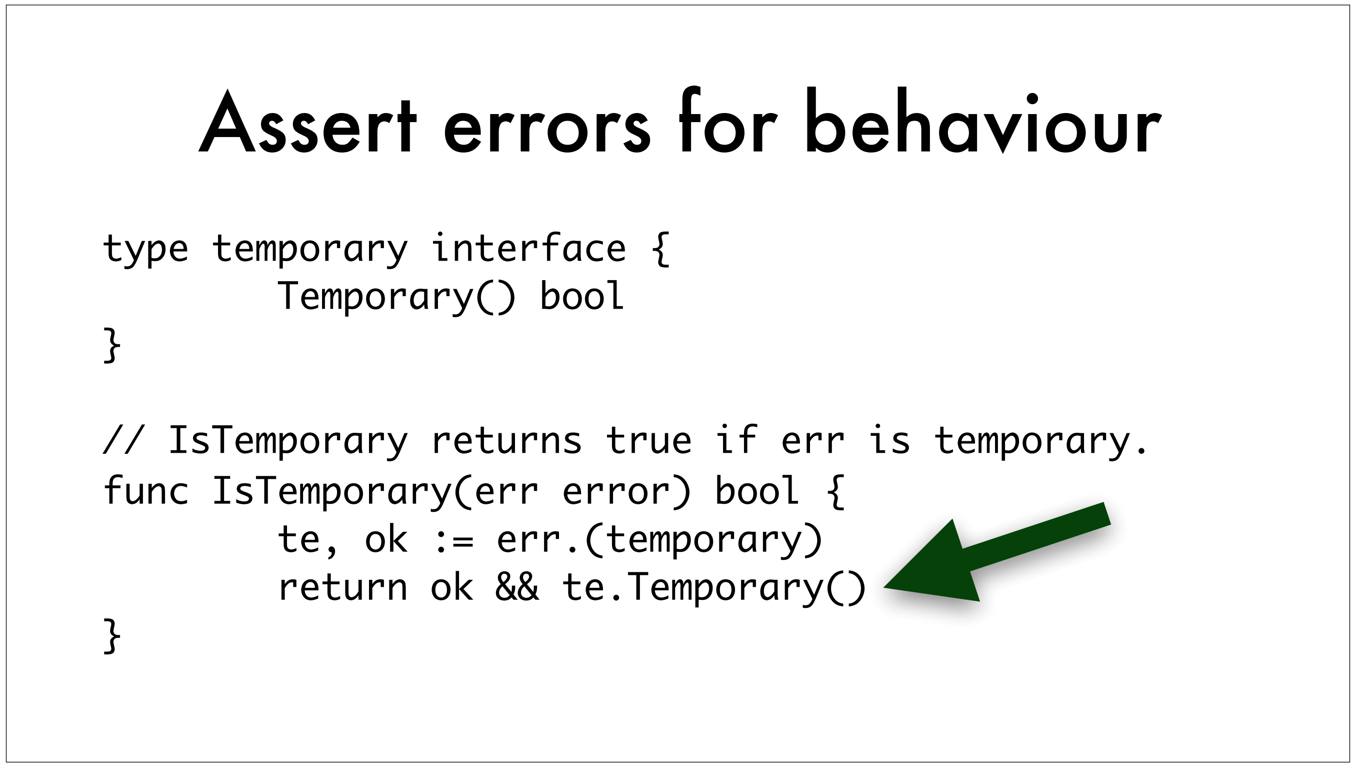


Assert errors for behaviour, not type

In a small number of cases, this binary approach to error handling is not sufficient.

For example, interactions with the world outside your process, like network activity, require that the caller investigate the nature of the error to decide if it is reasonable to retry the operation.

In this case rather than asserting the error is a specific type or value, we can assert that the error implements a particular behaviour.



Assert errors for behaviour

```go
type temporary interface {
        Temporary() bool
}

// IsTemporary returns true if err is temporary.
func IsTemporary(err error) bool {
        te, ok := err.(temporary)
        return ok && te.Temporary()
}
```

Consider this example:

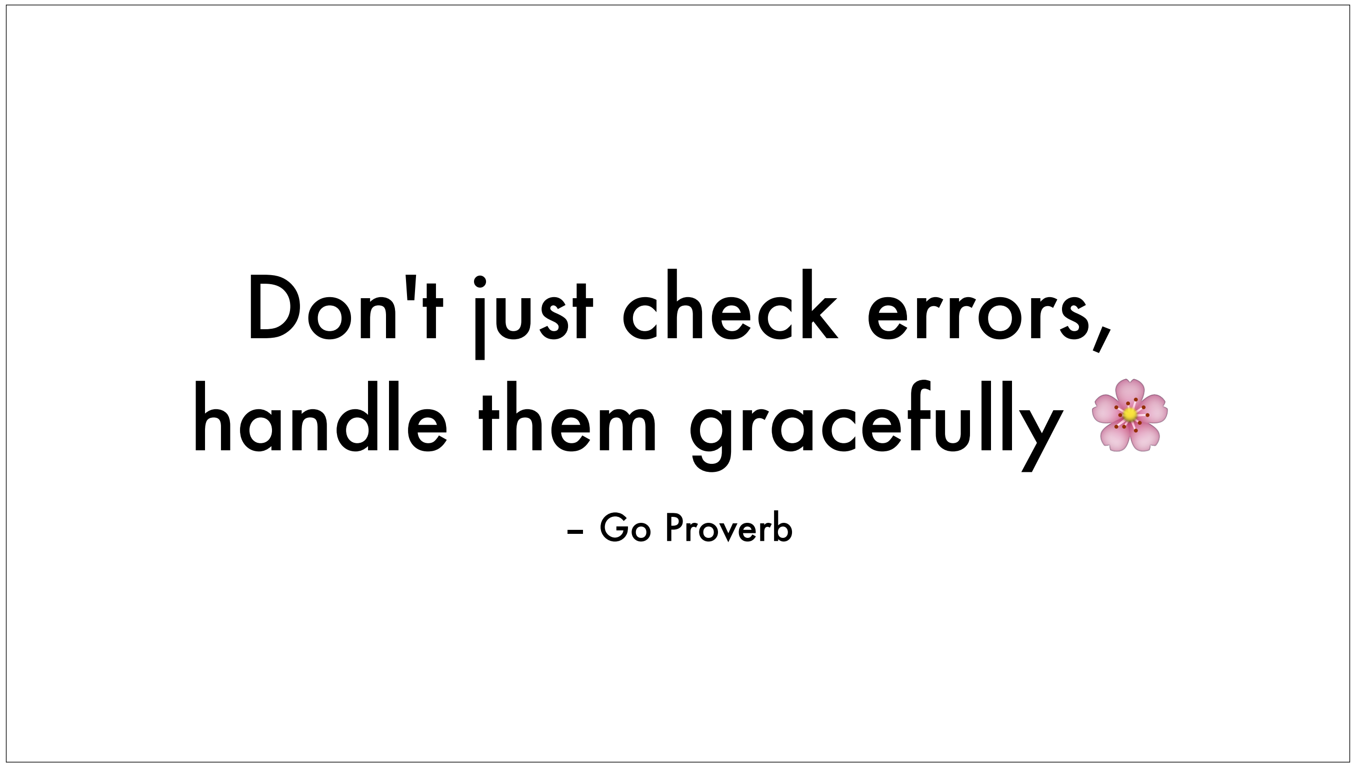
We can pass any error to our IsTemporary function to determine if the operation could be retried.

If the error does not implement the `temporary interface; it does not have a `Temporary` method; then then error is not temporary.

[click]

If the error does implement Temporary, we know a little bit more about the error and perhaps the caller can use the Temporary method to see if the operation can retried.

And the key here is all of this logic can be implemented without importing the package that defines the error or indeed knowing anything about err's underlying type.
We're simply interested in its behaviours.

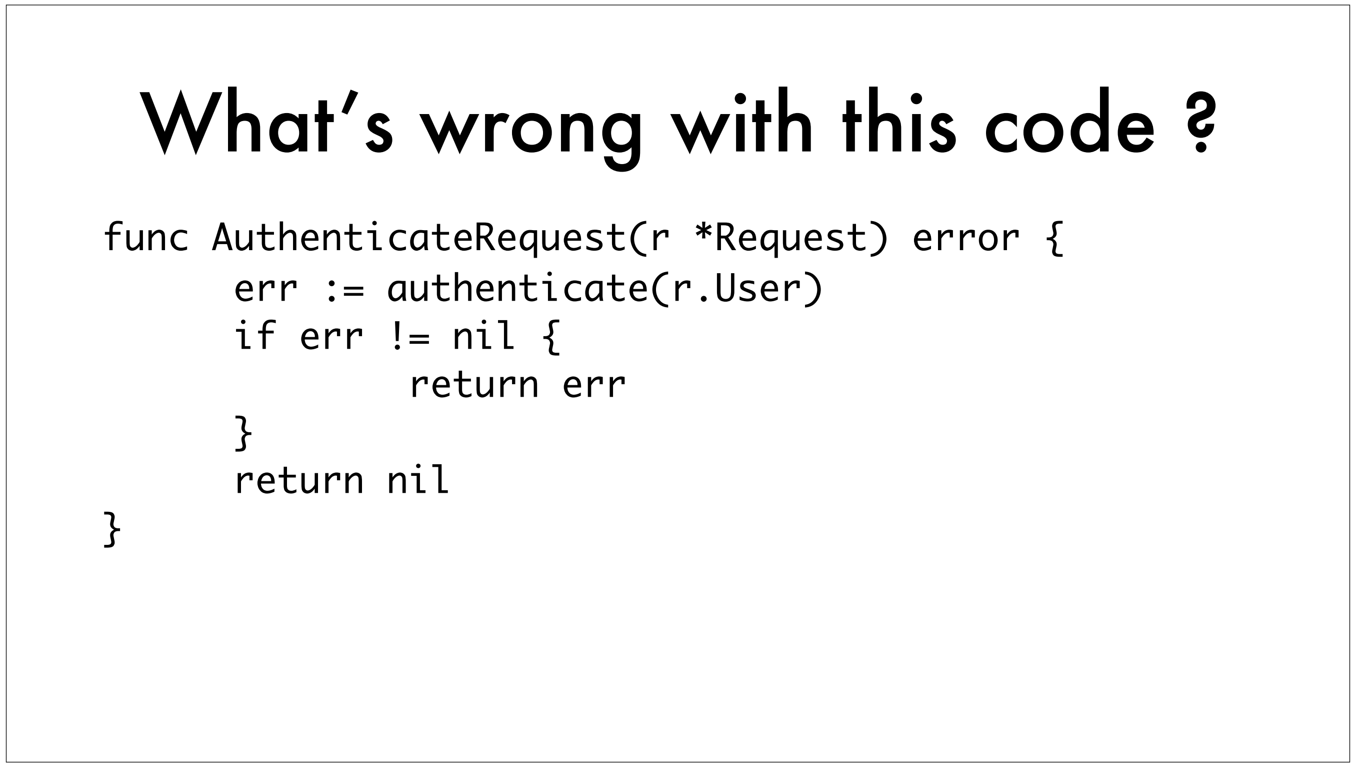


Don't just check errors,
handle them gracefully 🌸

- Go Proverb

This brings me to a second Go proverb that I want to talk about

Don’t just check errors, handle them gracefully.



What's wrong with this code ?

```
func AuthenticateRequest(r *Request) error {
      err := authenticate(r.User)
      if err != nil {
            return err
      }
      return nil
}
```

Can anyone tell me wrong with this piece of code?

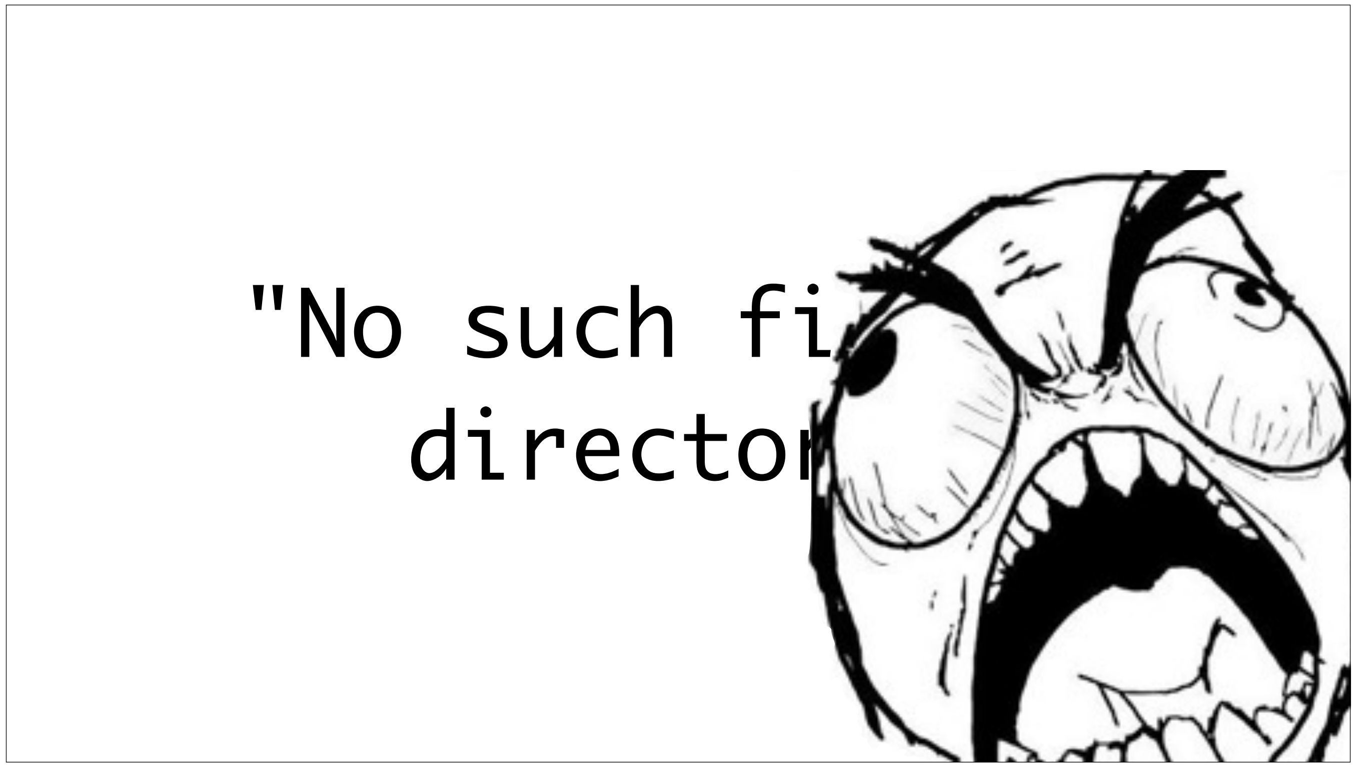
Obviously it's too verbose, it could be written as one line, return authenticate(r.User)

But there is a more serious issue with this piece of code.

To me at least, the problem with this code is if authenticate fails, I cannot tell where the failure happened.

If authenticate fails, then we pass the error up to the caller of AuthenticateRequest, and it'll pass the error up to it's caller, and so on.

At the top of my program we might print out the error to find the result is …

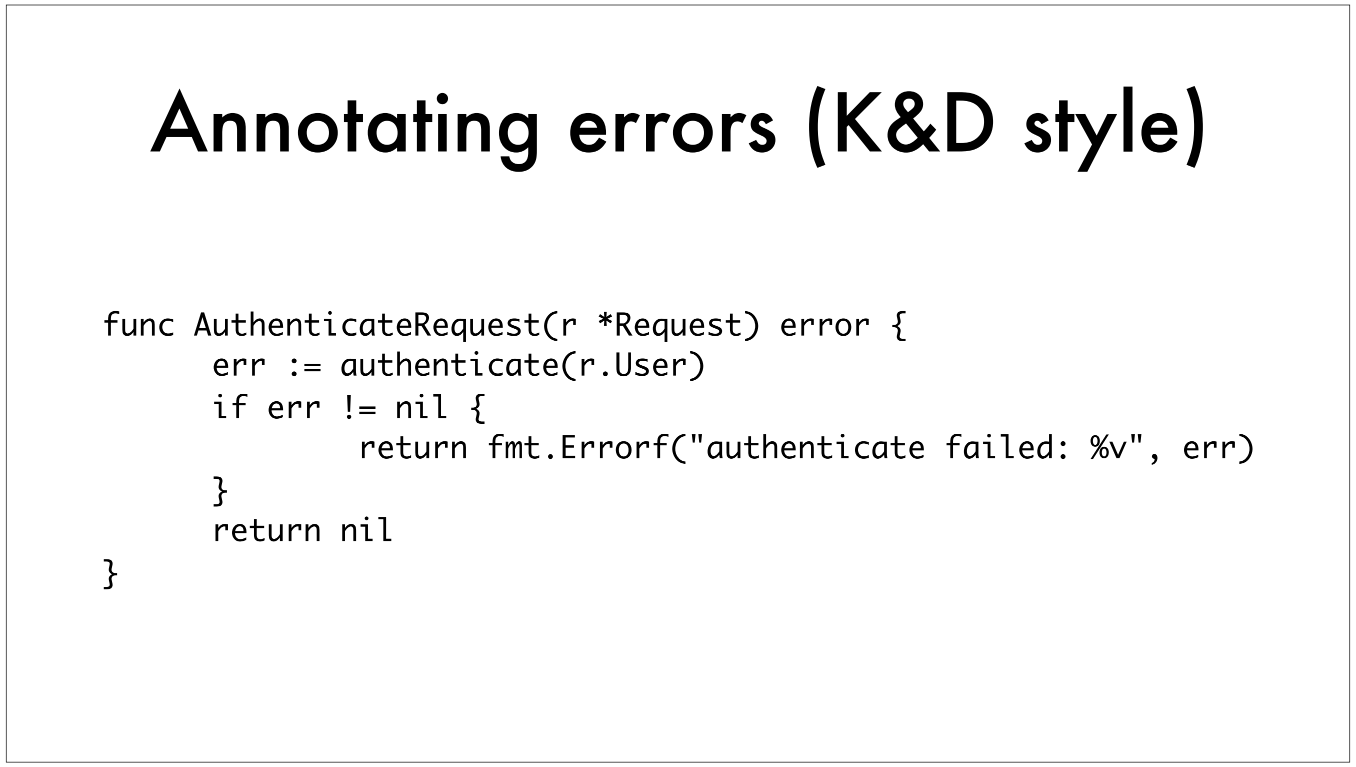


"No such fi
directo

… “No such file or directory".

There is no information of file and line where the error was generated.

There is no stack trace of the call stack leading up to the error.

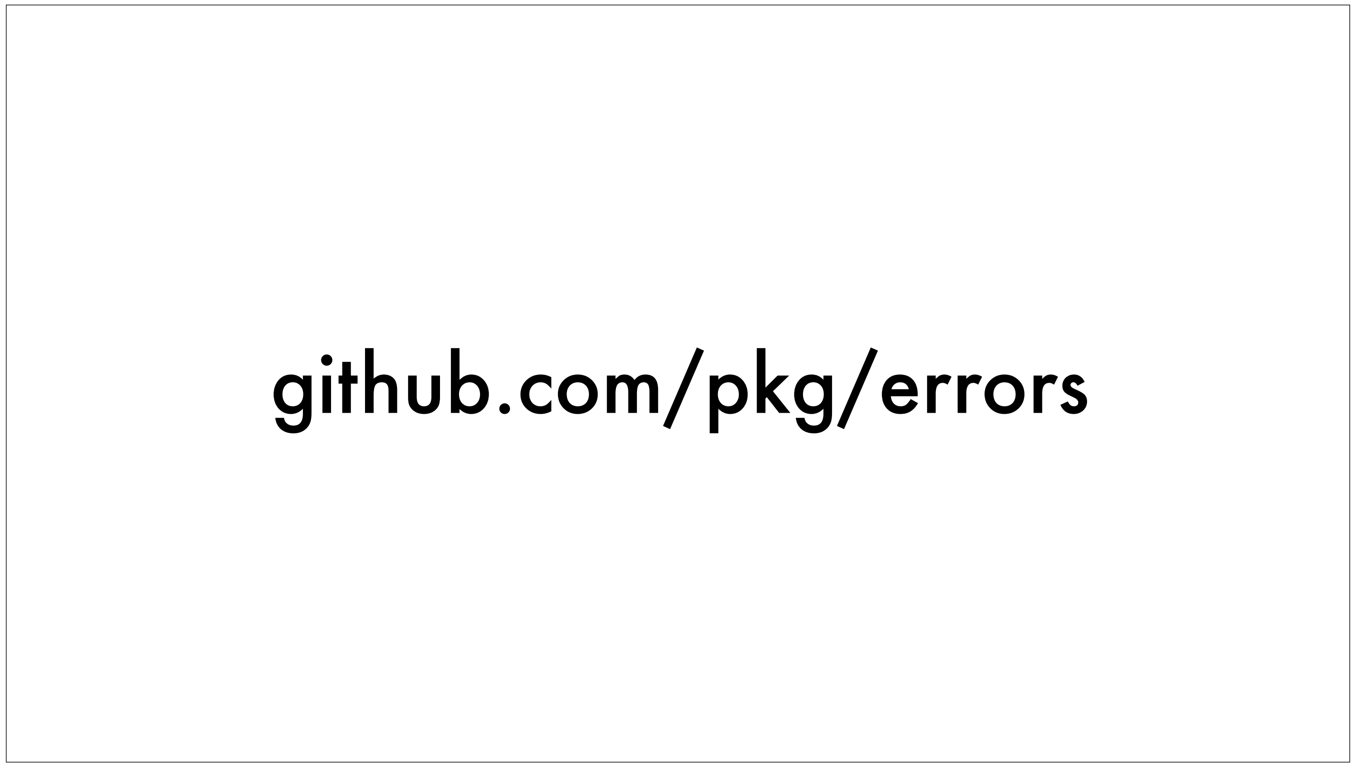


Annotating errors (K&D style)

```go
func AuthenticateRequest(r *Request) error {
      err := authenticate(r.User)
      if err != nil {
              return fmt.Errorf("authenticate failed: %v", err)
      }
      return nil
}
```

Donovan and Kernighan's *The Go Programming Language* recommends that you add context to the error path using fmt.Errorf

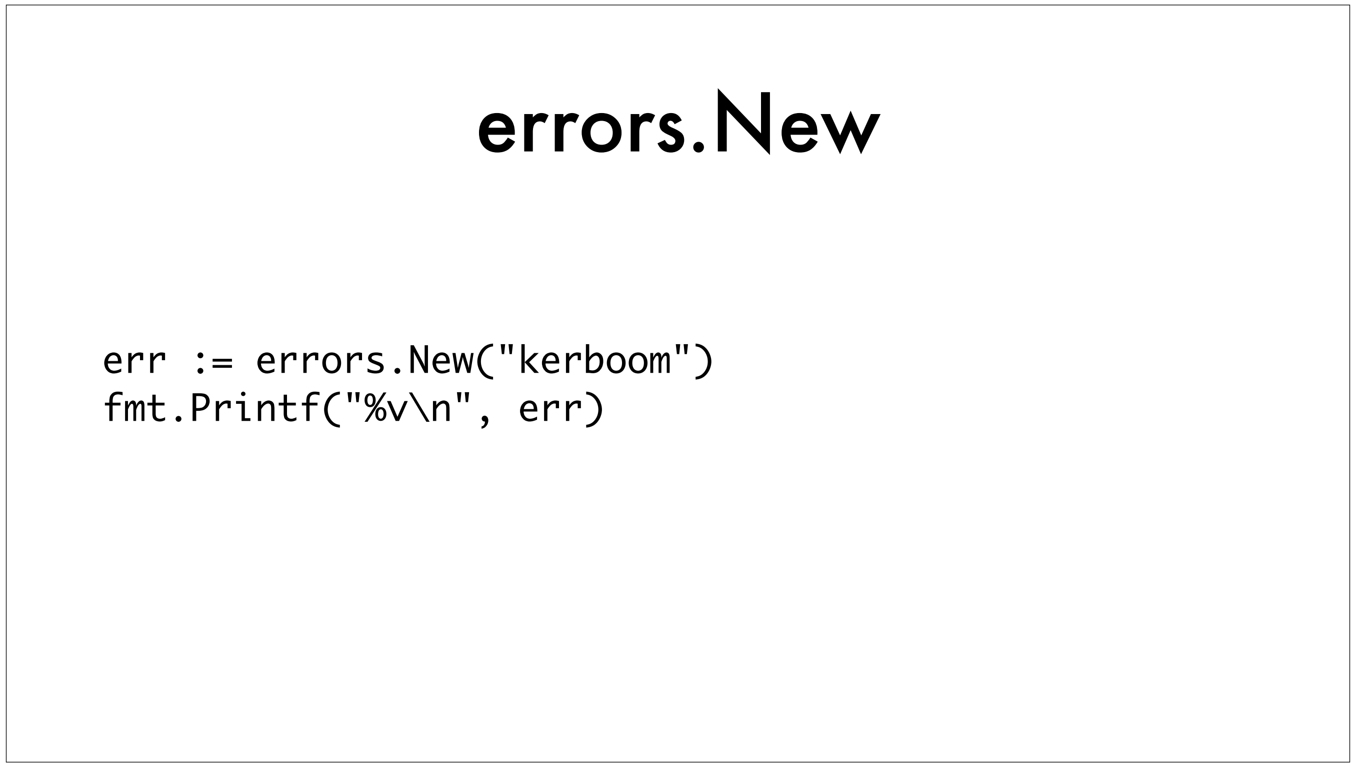
But as we saw earlier, this pattern is incompatible with the use of sentinel error values and type assertion because converting the error value to a string, merging it with another string, then converting it back to a new error with fmt.Errorf breaks equality and destroys the context in the original error.



github.com/pkg/errors

I'm going to talk a bit about how I add context to errors, and to do that I'm going to use a very simple errors package that I’ve been working on.

The errors package contains only five public functions and a few types, which exists mainly as a place to hang documentation.



errors.New

```
err := errors.New("kerboom")
fmt.Printf("%v\n", err)
```

The first function is New, which, just like the errors package in the standard library, returns an error with the message you provide.

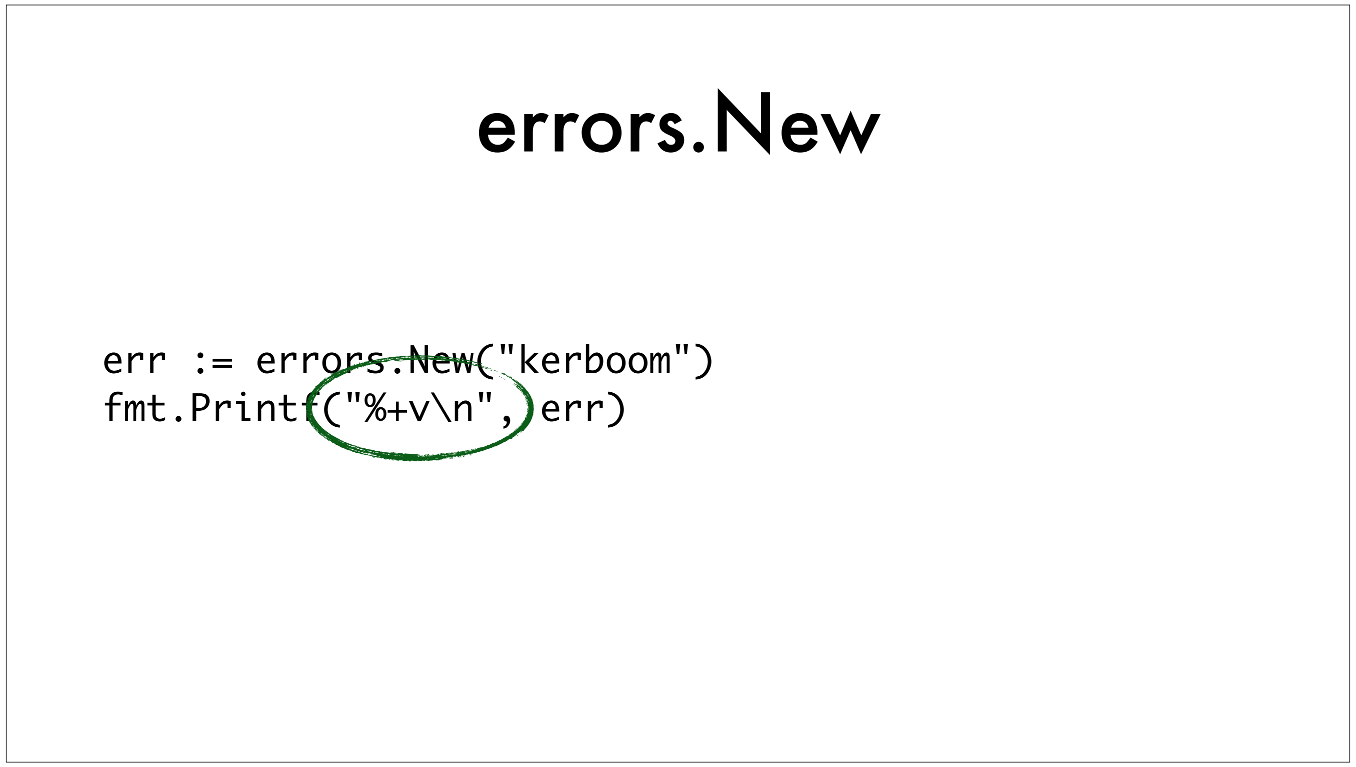
kerboom

errors.Wrap

```
err := errors.Wrap(
          syscall.EBADF, "couldn't write to stream")
fmt.Printf("%v\n", err)
```

The second function is Wrap, which returns a new error wrapping the error you provided with a message.

```
couldn't write to stream: bad file descriptor
```



errors.New

```
err := errors.New("kerboom")
fmt.Printf("%+v\n", err)
```

And you’re probably thinking at this point … what’s the big deal.

So let’s take a look at the first example again, the same call to errors.New, except this time when we print the error, we’ll use the %+v formatting verb

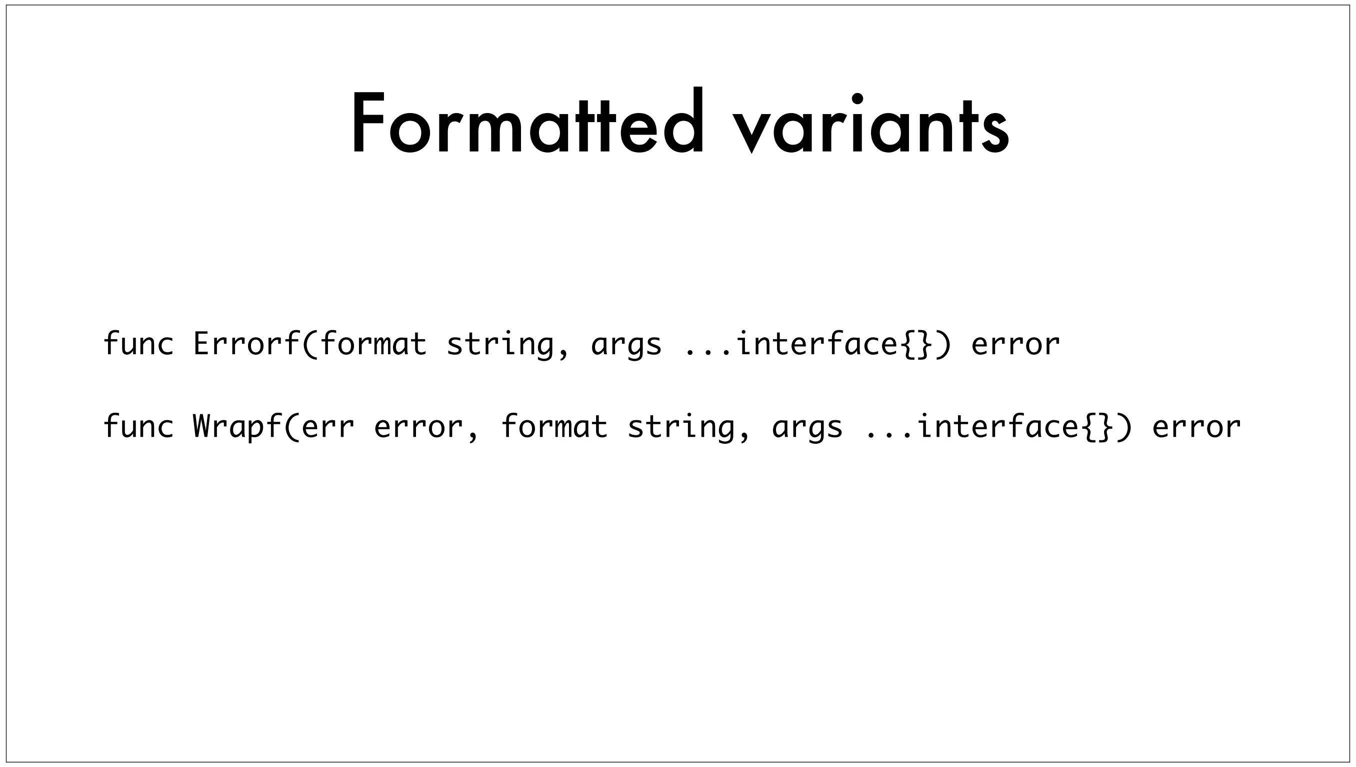
```
kerboom
github.com/pkg/errors/example.ExampleNew_extended
      /Users/dfc/src/github.com/pkg/errors/example/example_test.go:27
testing.runExample
      /Users/dfc/go/src/testing/example.go:114
testing.RunExamples
      /Users/dfc/go/src/testing/example.go:38
testing.(*M).Run
      /Users/dfc/go/src/testing/testing.go:744
main.main
      github.com/pkg/errors/example/_test/_testmain.go:60
runtime.main
      /Users/dfc/go/src/runtime/proc.go:183
runtime.goexit
      /Users/dfc/go/src/runtime/asm_amd64.s:2086
```

And we get something like this.

And this works because

Each time you call one of the functions in the errors package, the error returned, contains as well as the message you provided, the full stack trace at the point it was created.

These error values also implement fmt.Formatter interface and know how to print themselves with various levels of detail depending on the formatting verb you provide.

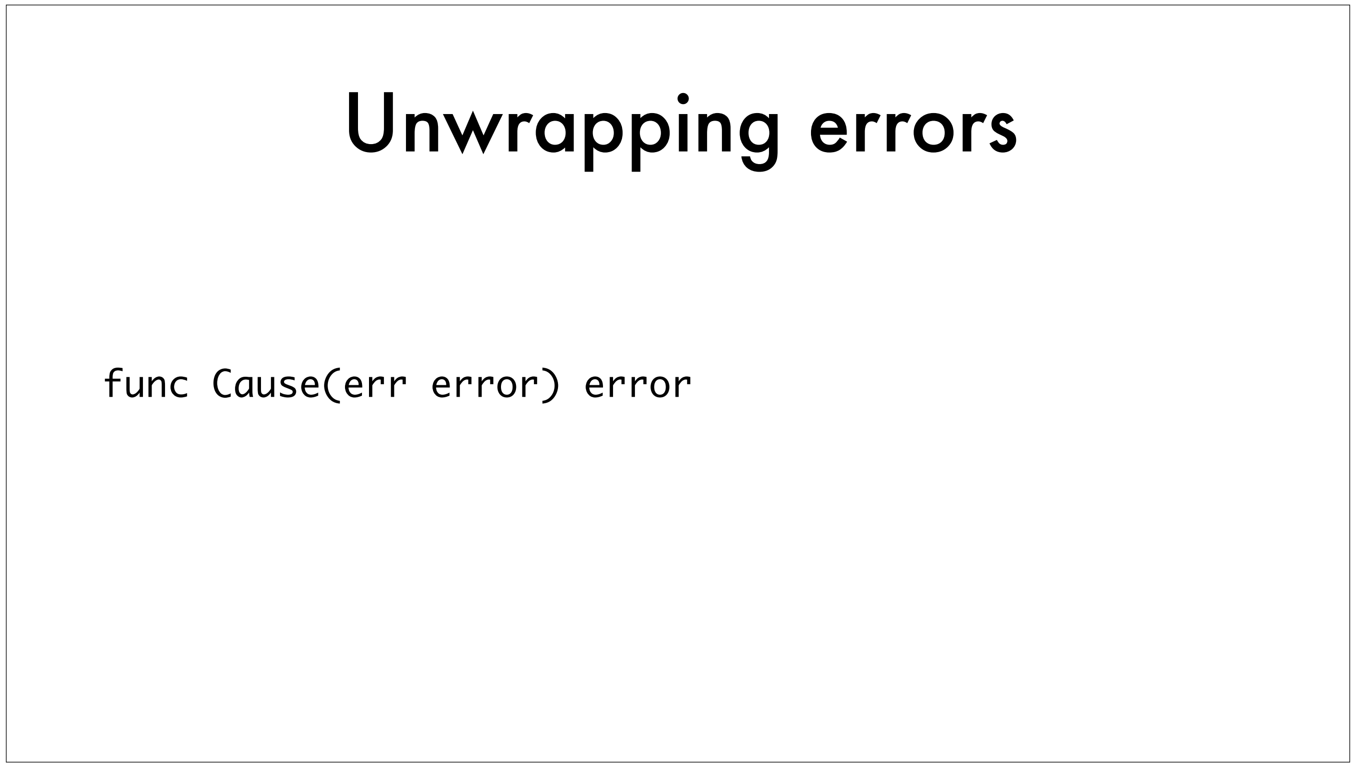


Formatted variants

```
func Errorf(format string, args ...interface{}) error

func Wrapf(err error, format string, args ...interface{}) error
```

For convenience, the errors package also offers version of this functions which support the usual printf style formatting verbs.

The function of Errorf is the same a New, Wrapf is the same as Wrap.



Unwrapping errors

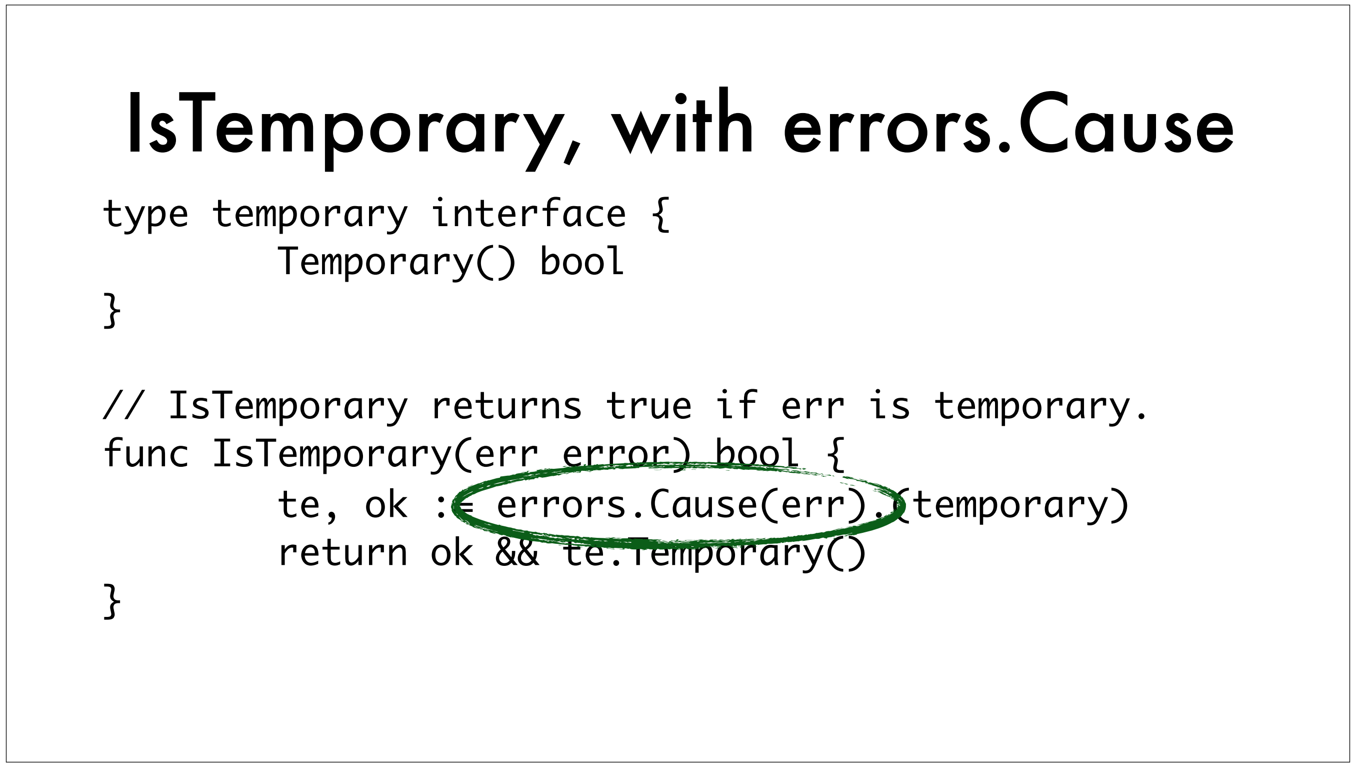
```
func Cause(err error) error
```

Because we've now introduced the concept of wrapping errors, we need to talk about the reverse, unwrapping them.

This is the domain of the `errors.Cause` function.

Cause takes an error, and undoes the wrapping process to return the original error.

Cause always unwraps all the way down to the original error.



IsTemporary, with errors.Cause

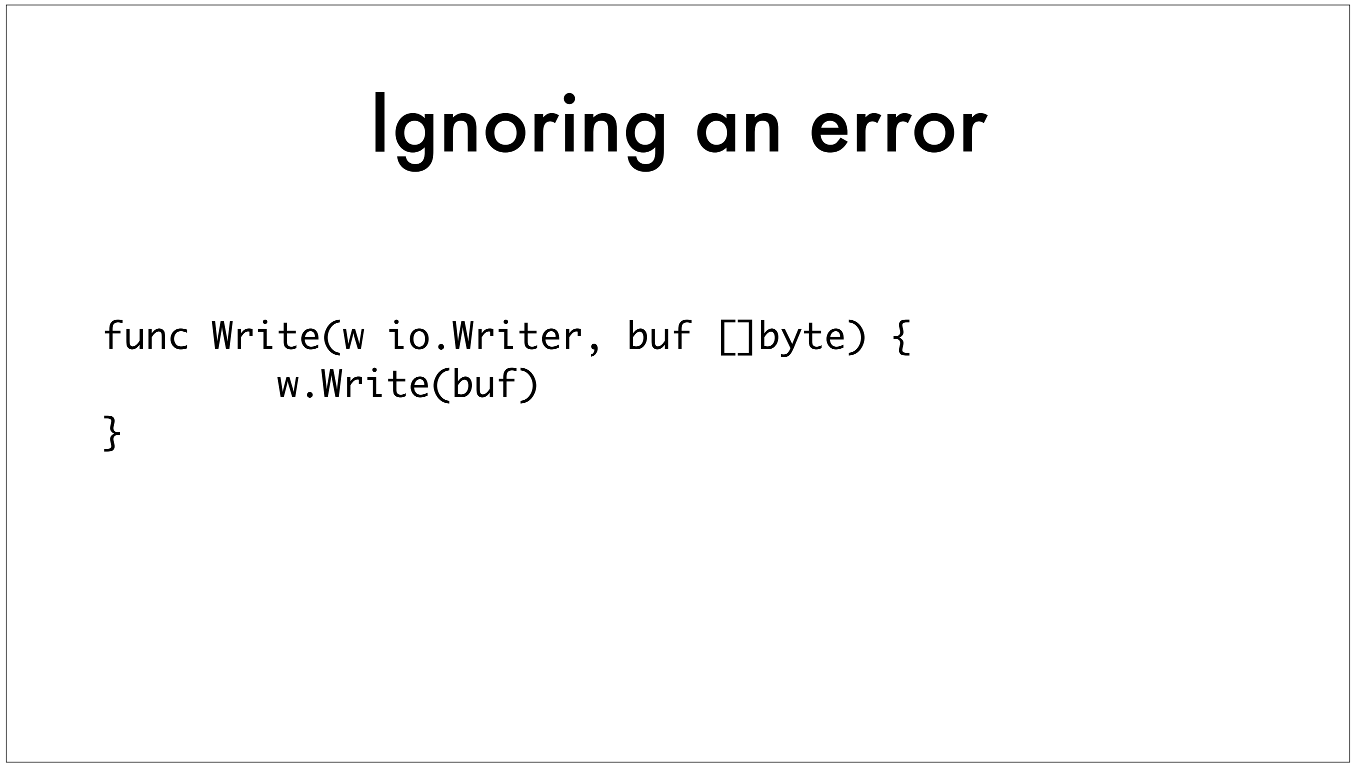
```
type temporary interface {
        Temporary() bool
}

// IsTemporary returns true if err is temporary.
func IsTemporary(err error) bool {
        te, ok := errors.Cause(err).(temporary)
        return ok && te.Temporary()
}
```

In operation, whenever you need to check an error matches a specific value or type, you should first unwrap it to recover the original error using the errors.Cause function.

Only handle errors once 👌

Lastly, I want to mention that you should only handle errors once.

Handling an error means inspecting the error, and making a single decision.



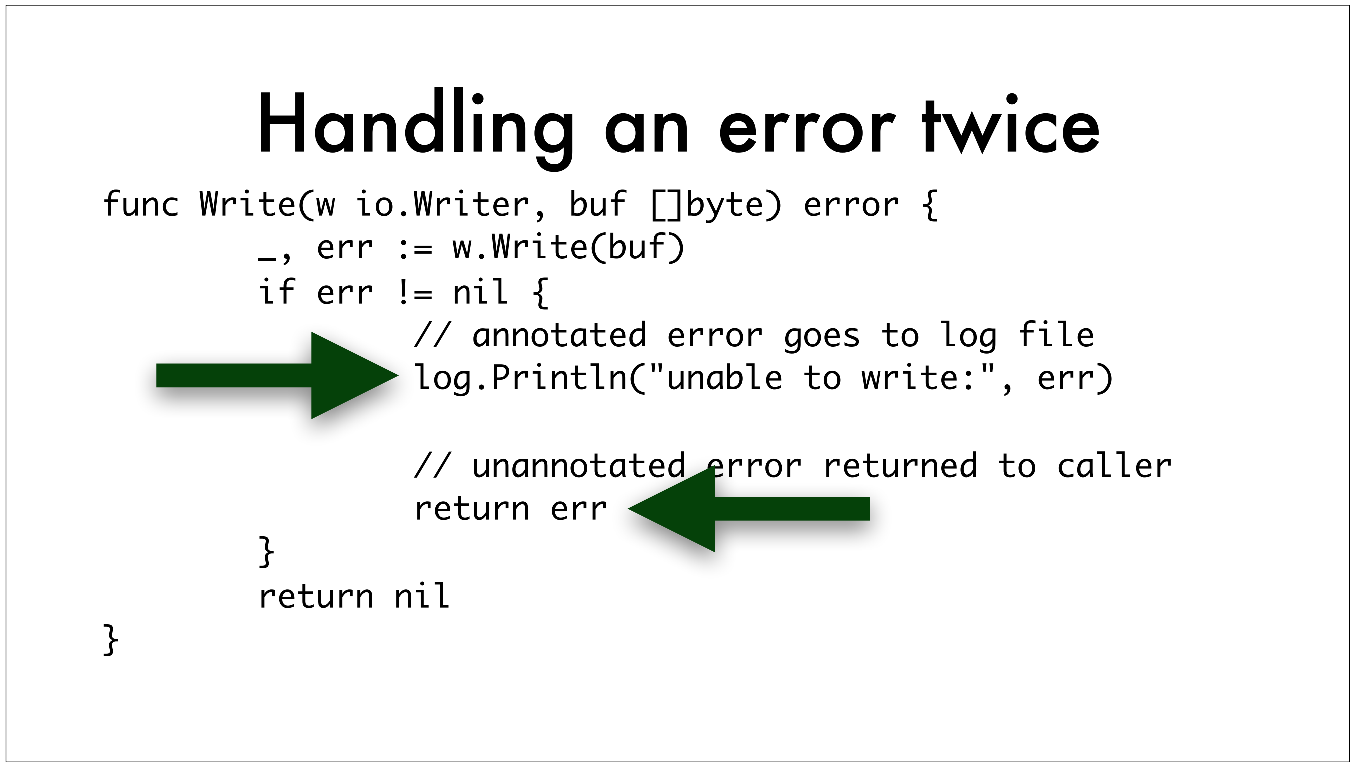
Ignoring an error

```
func Write(w io.Writer, buf []byte) {
        w.Write(buf)
}
```

If you make less than one decision, you're obviously ignoring the error

as we see here, the error from w.Write is simply being discarded.

This is pretty straight forward.



Handling an error twice

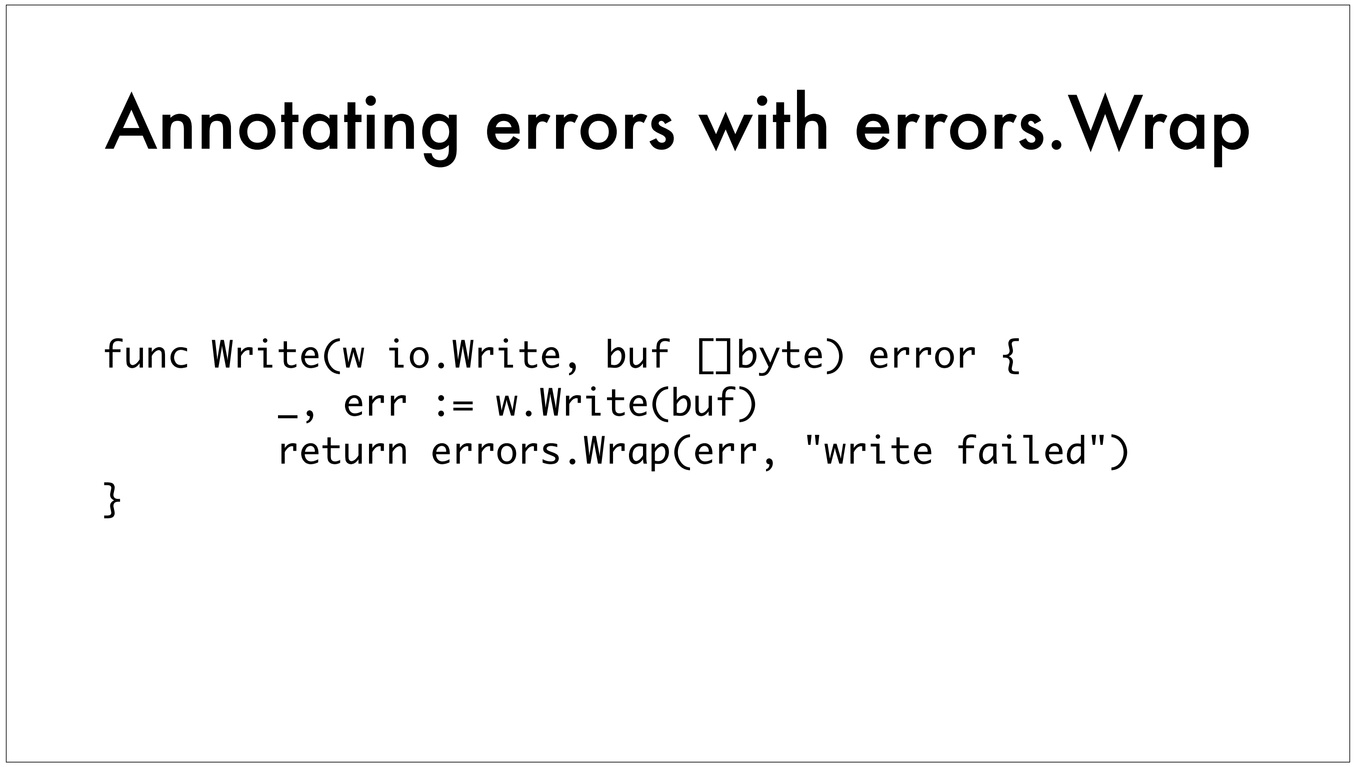
```
func Write(w io.Writer, buf []byte) error {
        _, err := w.Write(buf)
        if err != nil {
                // annotated error goes to log file
                log.Println("unable to write:", err)

                // unannotated error returned to caller
                return err
        }
        return nil
}
```

But making more than one decision in response to a single error is also bad.

In this example if an error occurs during write, a line will be written to a log file, noting the file and line close to where the error occurred

The error is also returned to the caller, who possibly will log it, and return it, all the way back up to the top of the program.

So the result of this, is you get a stack of duplicate lines in your log file — and at the top of the program you get the original error without any context.



Annotating errors with errors.Wrap

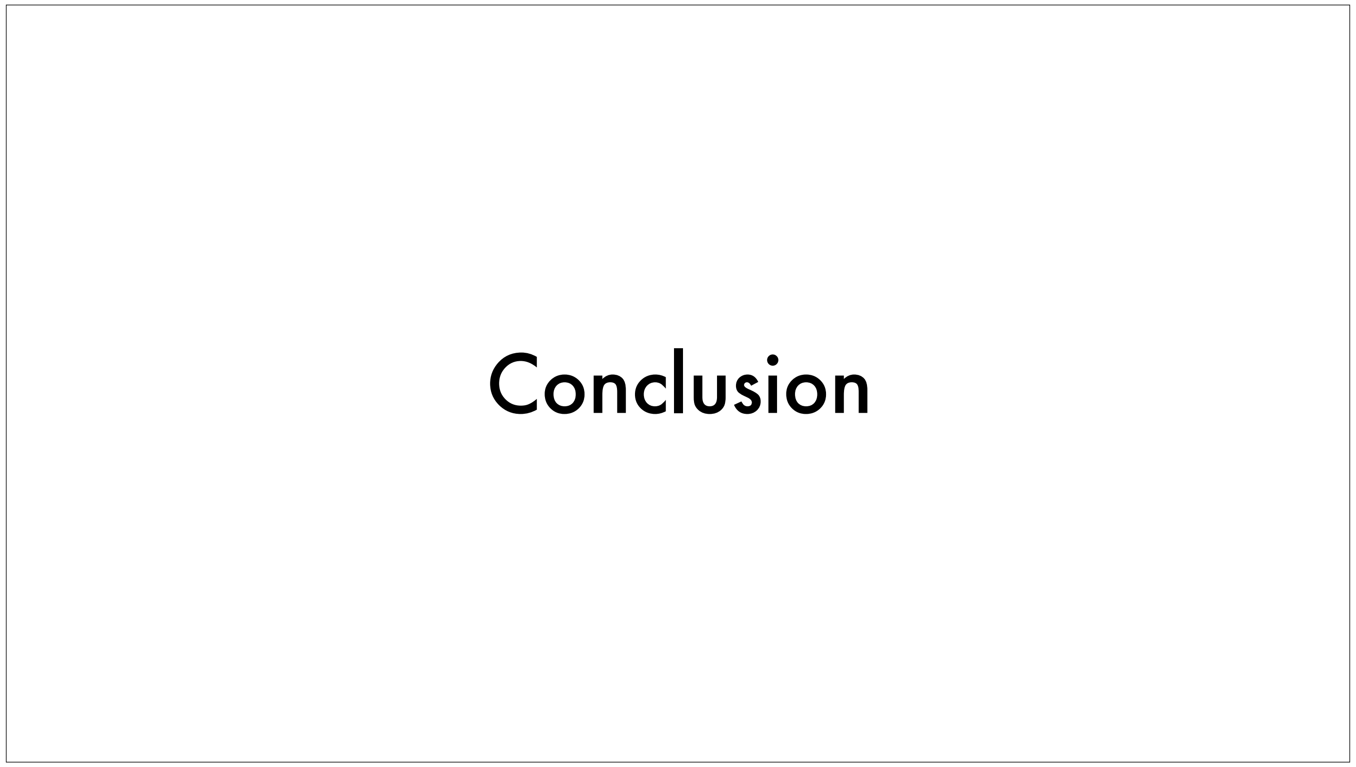
```
func Write(w io.Write, buf []byte) error {
        _, err := w.Write(buf)
        return errors.Wrap(err, "write failed")
}
```

The errors package gives you the ability to add context to error values, in a way that is inspectable by both a human, when you print them out, and a machine, if you use errors.Cause to recover the underlying error.

In this example, if there is an error in the call to w.Write, we're going to wrap it with the annotation "write failed"

Again the errors package makes this easy because if you pass nil to errors.Wrap it will return nil unconditionally.

This helps reduce the verbosity of having to check if the error was not nil just so you could annotate it.



Conclusion

To summarise

Errors are part of your package's public API🤔

Errors are part of your package's public API, treat them with as much care as you would any other part of your public API.

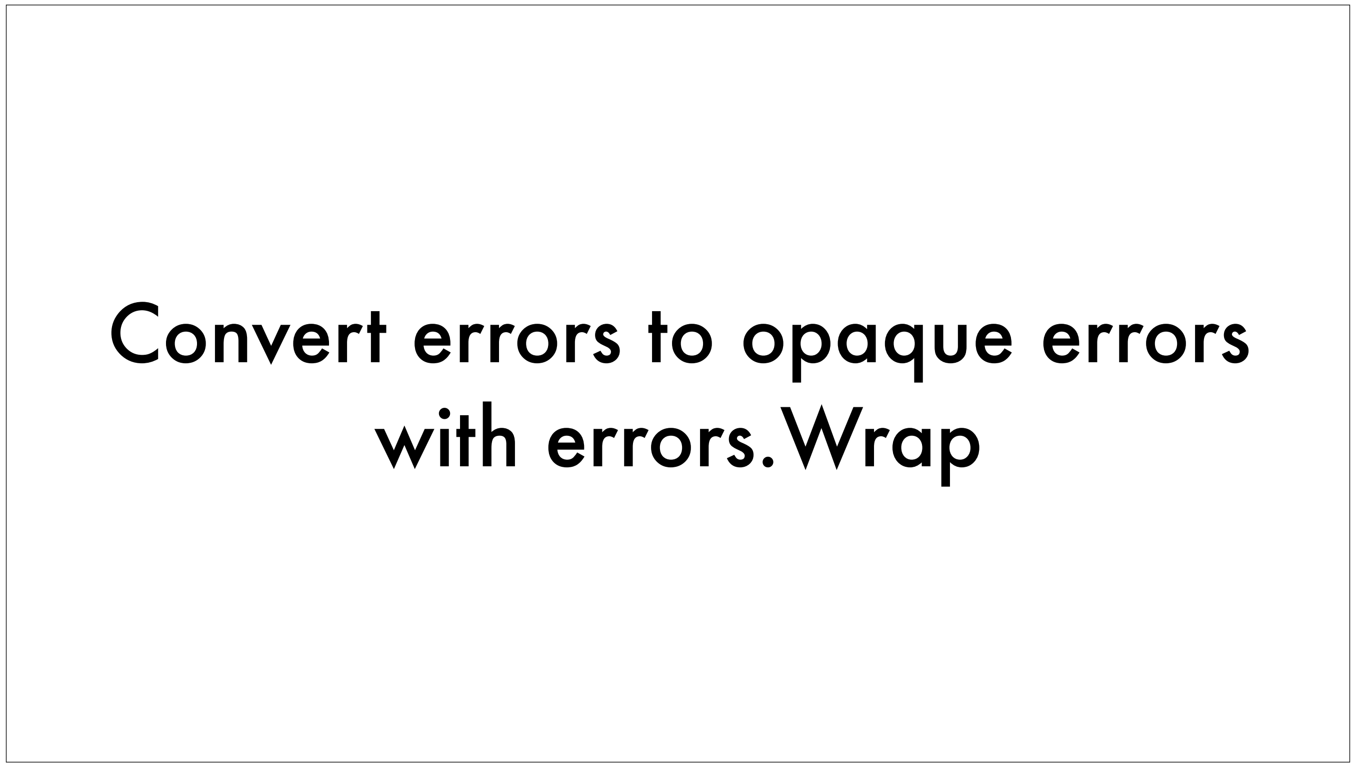
Treat errors as opaque; assert for behaviour, not type

For maximum flexibility i recommend that you try to treat all errors as opaque.

When in the situations where you cannot do that, assert errors for behaviour, not for their type.

Minimise the use of sentinel error values in your program

Minimise the number of sentinel error values in your program. And avoid overloading the error value to convey something which is not actually an error.



Convert errors to opaque errors with errors.Wrap

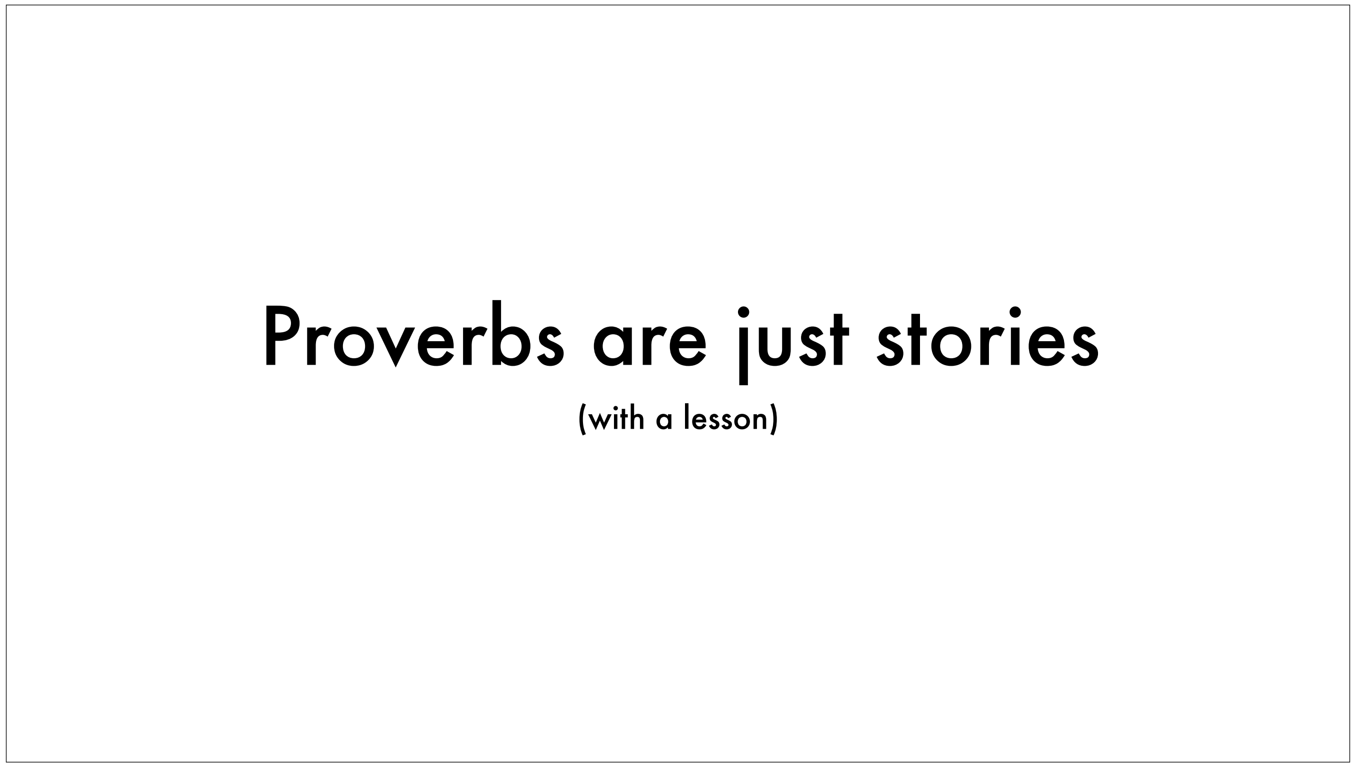
If you interact with a package from another repository, convert any errors you receive to opaque errors with errors.Wrap (or Wrapf).

This also establishes a stack trace that you can follow when debugging your program.

This advice also applies when interacting with the standard library.

Use errors.Cause to recover the underlying error

Lastly, use `errors.Cause` to recover the underlying error if you need to inspect it.



Proverbs are just stories

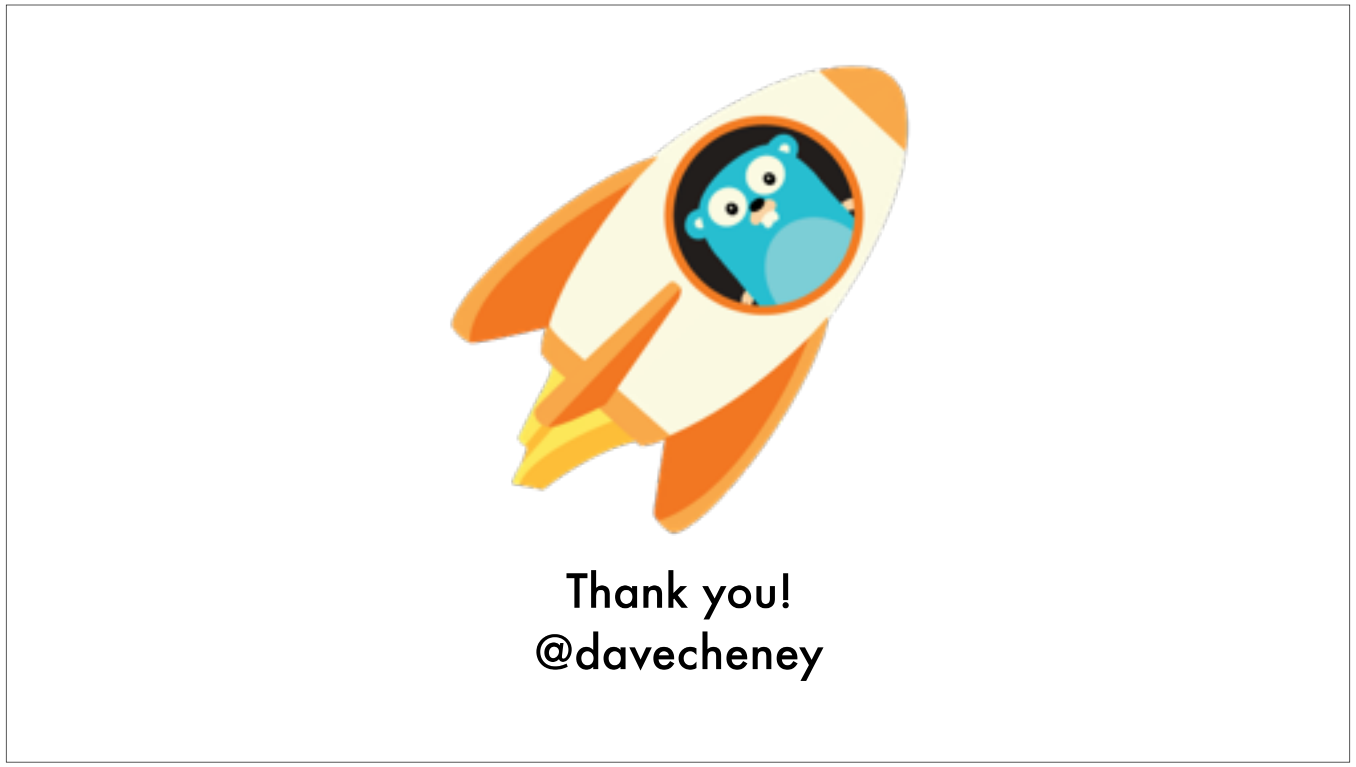
(with a lesson)

In conclusion, proverbs are not rules or laws, they're just stories.

Proverbs are a great way of encapsulating information; capturing the essence of a lesson or teaching a moral.

But they can equally be bewildering to newcomers who do not know the meaning behind the proverb.

And so, I will leave you to consider the meaning behind the other Go proverbs.



Thank you!
@davecheney

Thank you.