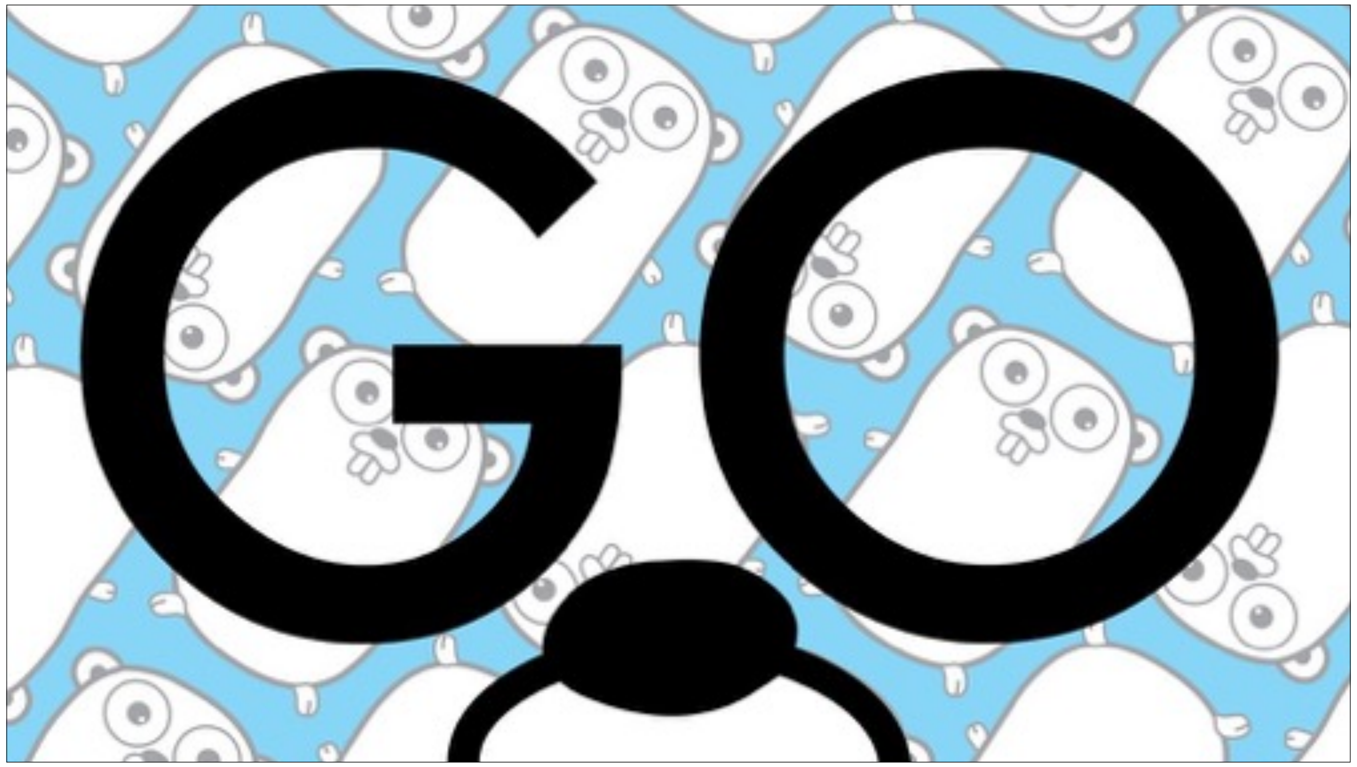
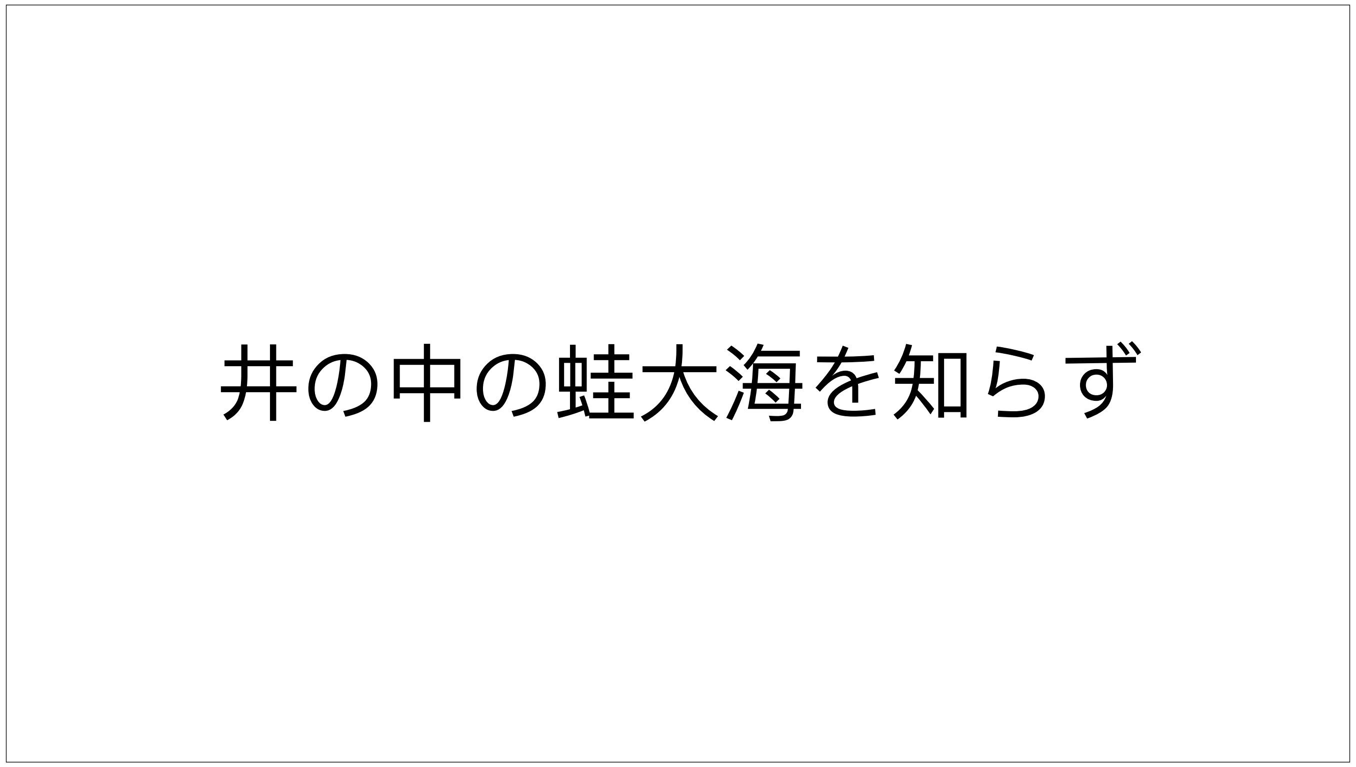


Good morning! Thank you for coming to my talk.

Before I begin, I want to express my gratitude to tenntenn and the organisers of GoCon for inviting me to speak today.

I also want to acknowledge the generous sponsorship of our hosts, Cyber Agent, for providing the venue for today.



井の中の蛙大海を知らず

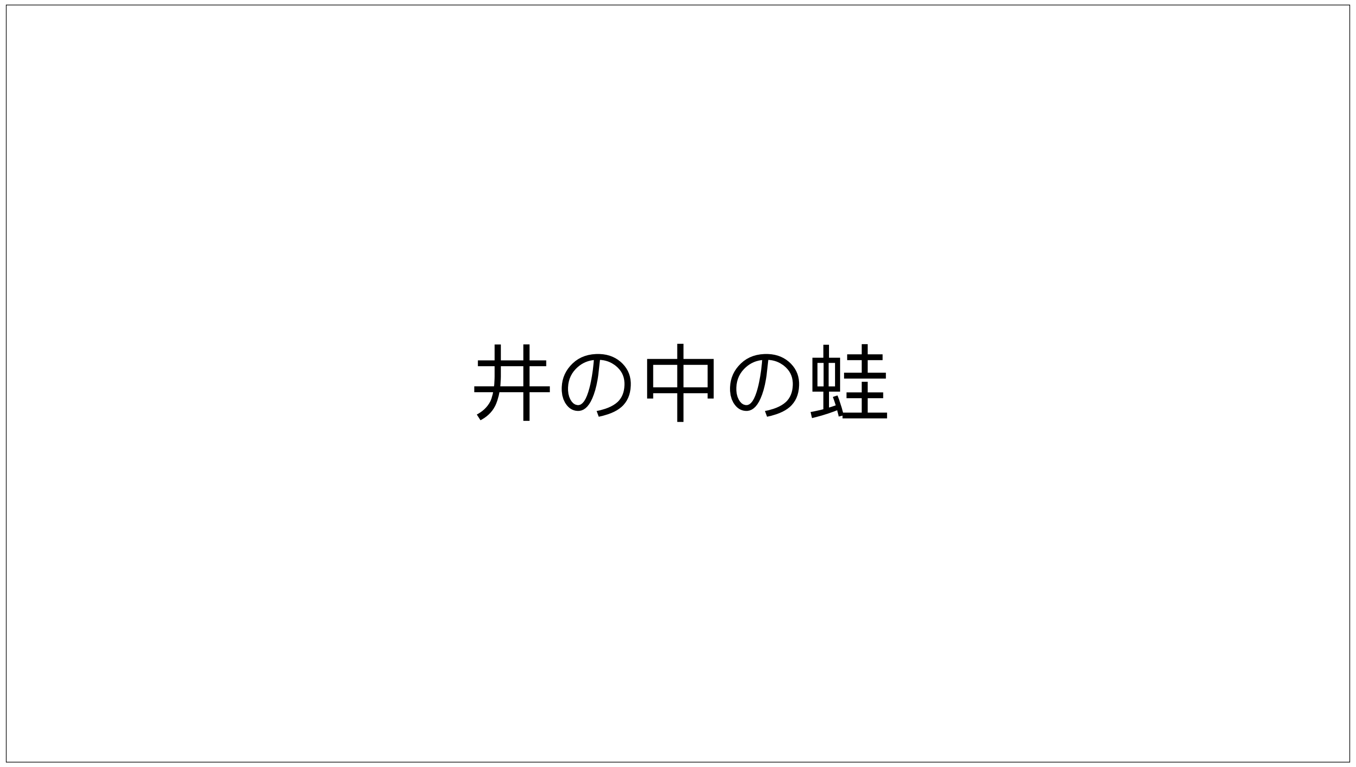
[to audience] would someone be kind enough to read this for me ? Do you know the english translation?

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

What is this proverb trying to teach us ?

If you are the frog, you may 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 yet convinced you that you are wrong.


井の中の蛙

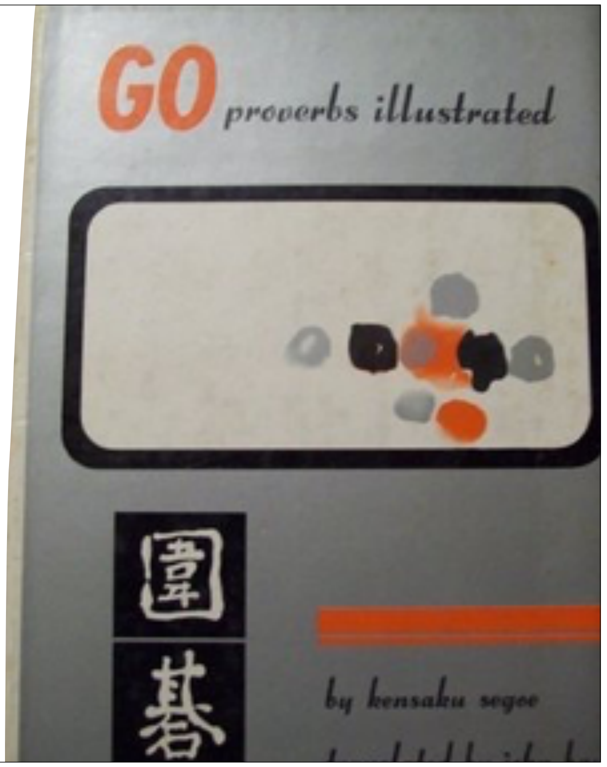
So if I say to you

I noh na-ka noh ka-wa-zuu,

maybe some of you understand the advice, and those who do, understand it not from the words *a frog in a well*, but because you understand their meaning.

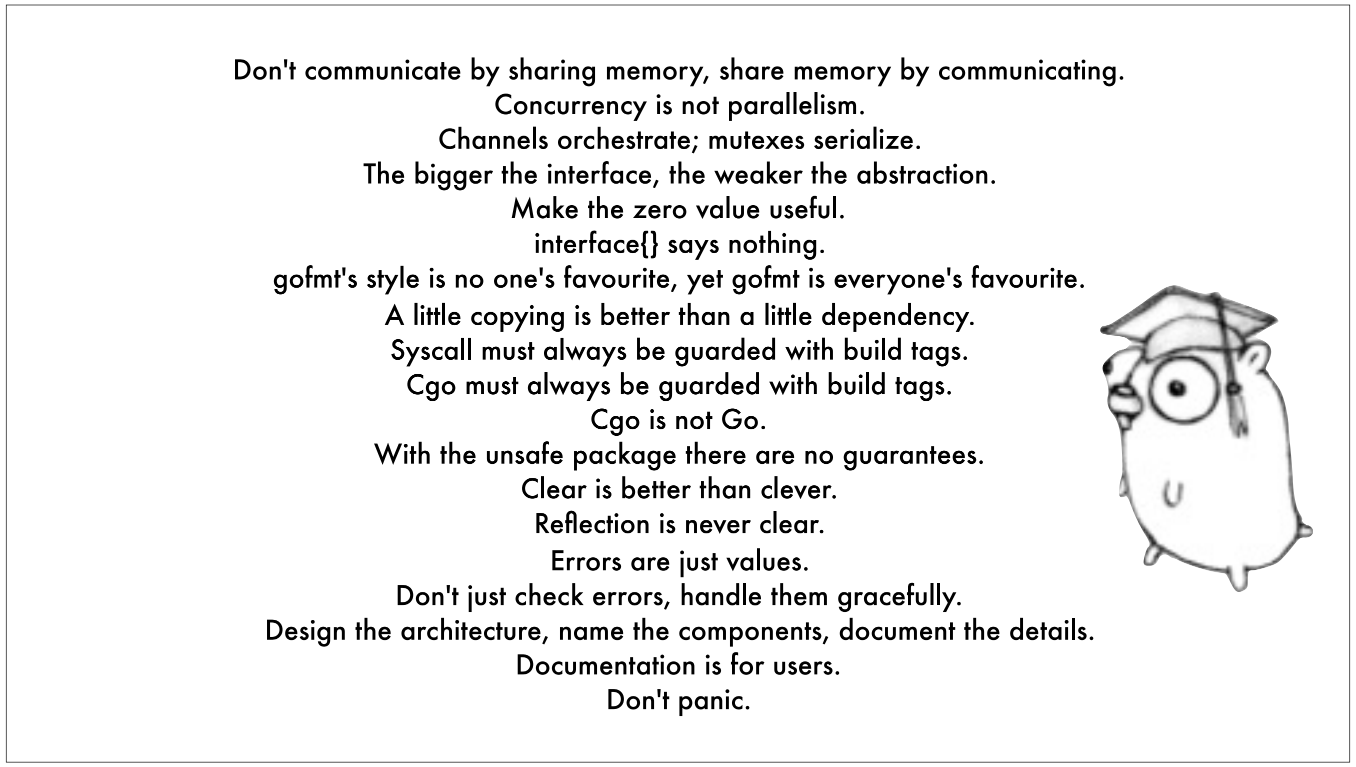
You understand the *lesson* that the story of the frog in his well teaches us.

Go proverbs



Last year Rob Pike gave a talked 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 may recognise some of them.

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 lessons they try to teach.

My goal today is not to repeat Sensei Pike's words, I cannot do them justice.

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

Errors are just values

– Go Proverb

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

When we say "errors are just values", we actually mean "any value that implements the `error` interface is itself an error", but saying "errors are just values" to someone who is just a student of Go is not useful.

The student must understand the underlying message of the proverb to appreciate its wisdom

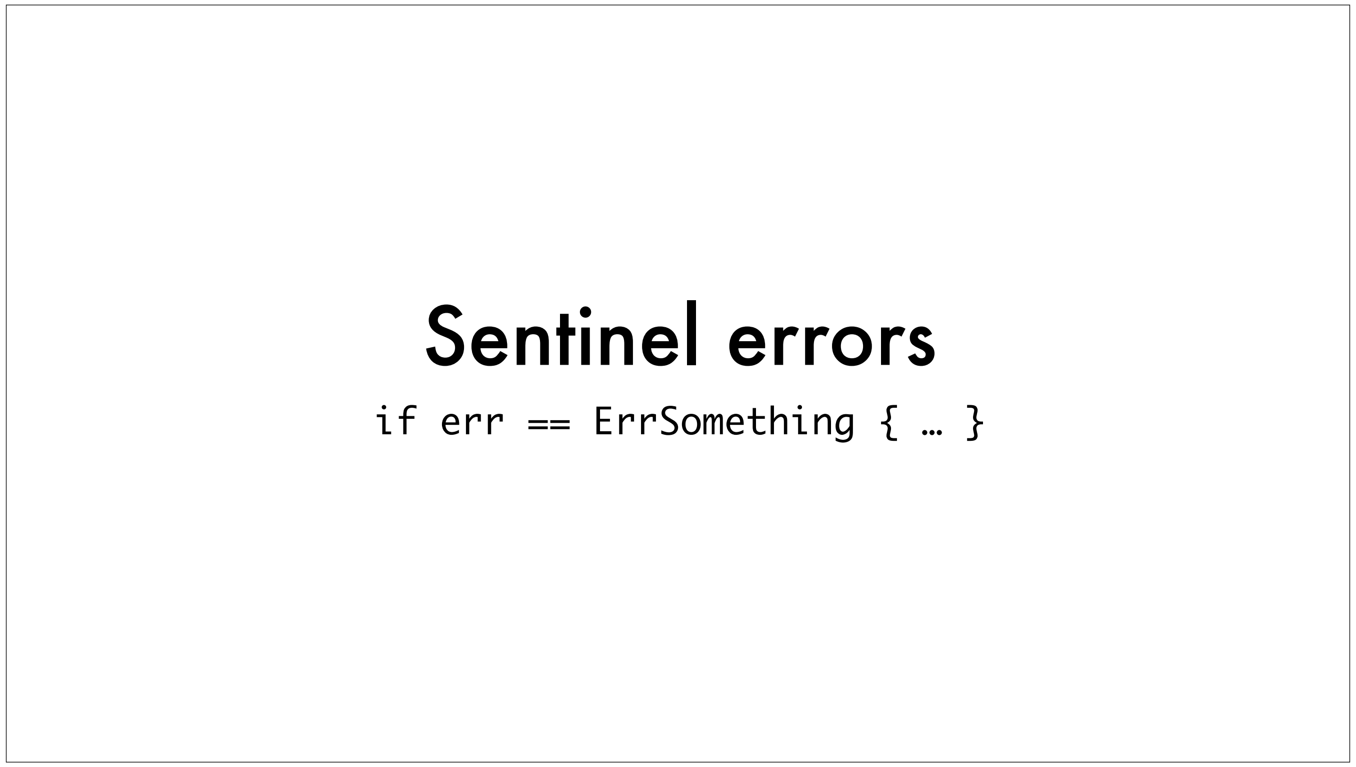
This is a good place to start the discussion on what I think it means to understand Go's error handling philosophy.

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 mathematics, or the alphabet.

However, I have concluded that there is no single way to handle errors.

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



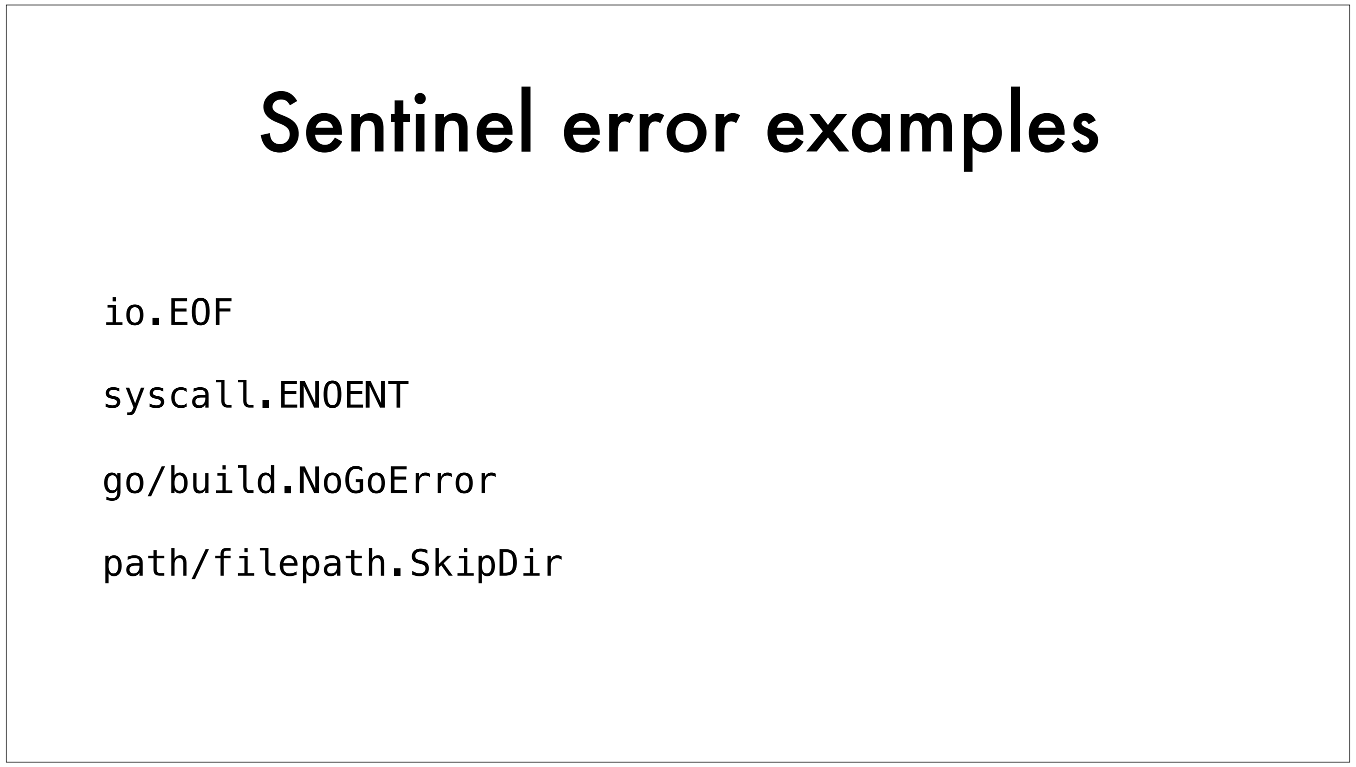
Sentinel errors

```
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.

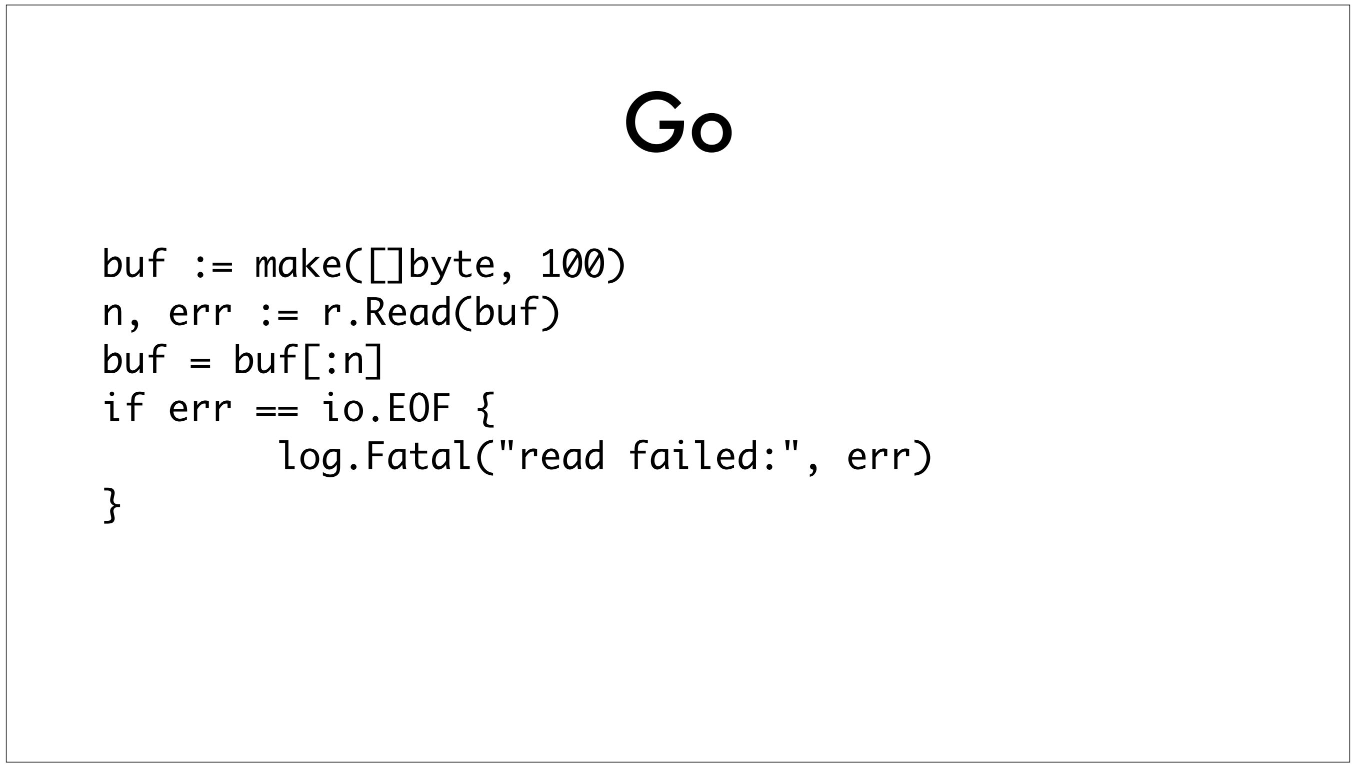


Sentinel error examples

```
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

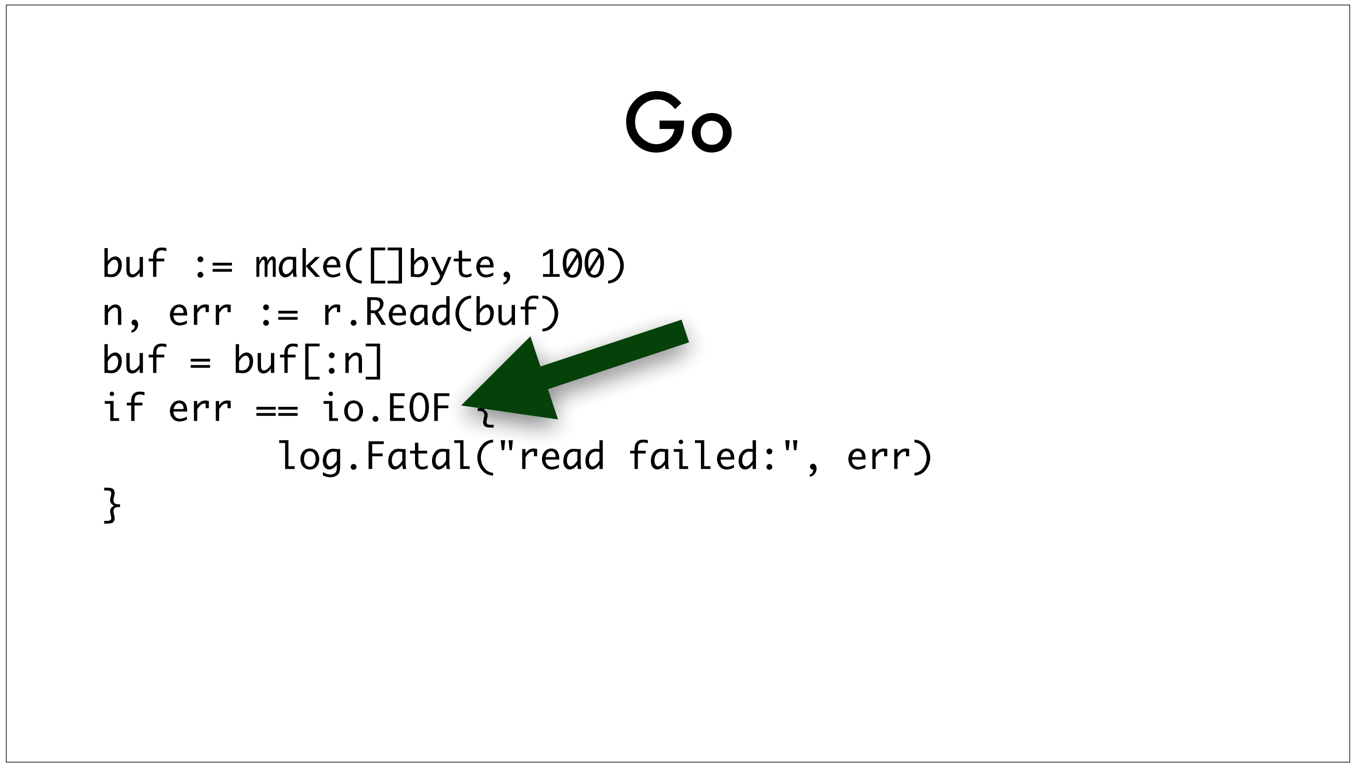


Go

```
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, as the caller must compare the result to pre declared value using the equality operator.

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



Go

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

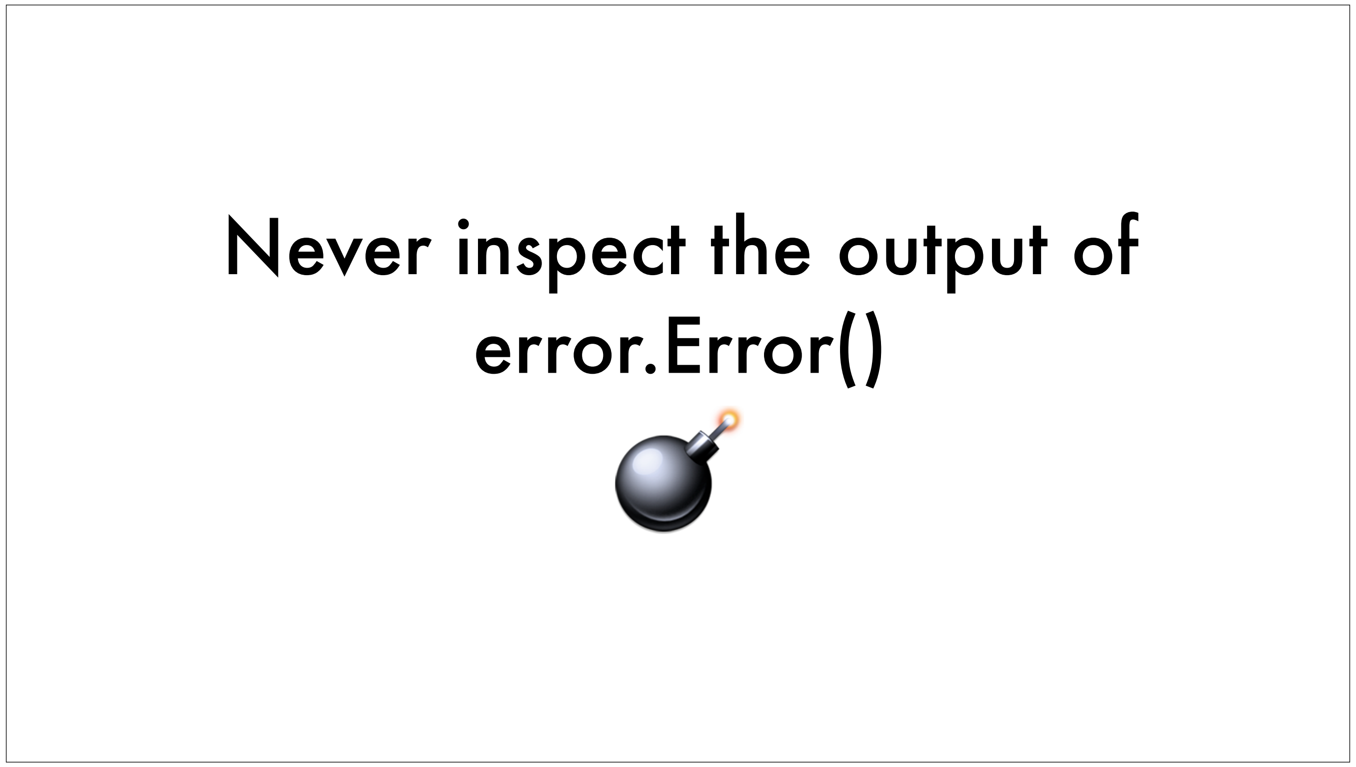
Using sentinel values is the least flexible error handling strategy, as the caller must compare the result to pre declared value using the equality operator.

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 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.

The `Error` method on the `error` interface 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 it.

I know that sometimes this isn't possible, and as someone pointed out on twitter, this advice doesn’t apply to writing tests.

Never the less, comparing the string form of an error is, in my opinion, a code smell, and you should try to avoid it.

Sentinel errors become part of your public API

If your public function or method returns an error of a particular 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 error, all implementations of that interface will 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 “no more data, but that isn’t an error”

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`, your code must import the `io` package.

This specific example does not sound so bad, because it is quite common, imagine the coupling that exists when many packages in your project export 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 that the spectre of bad design--in the form of an import loop--was never far from our minds.

Conclusion: avoid sentinel errors

👎

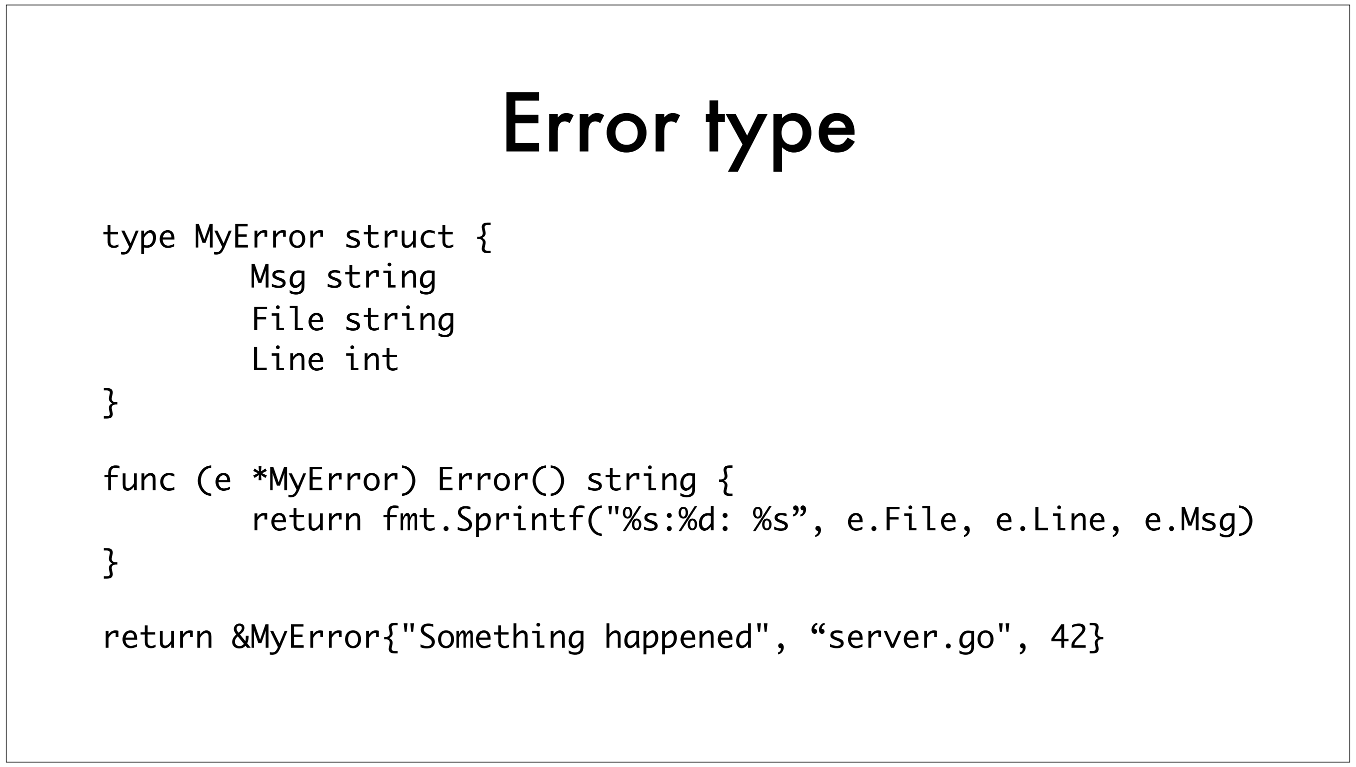
So, my advice to you is to avoid using sentinel error values in the code you write. There are a few cases where they are used in the standard library, but this is not a pattern i think you should emulate.

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.

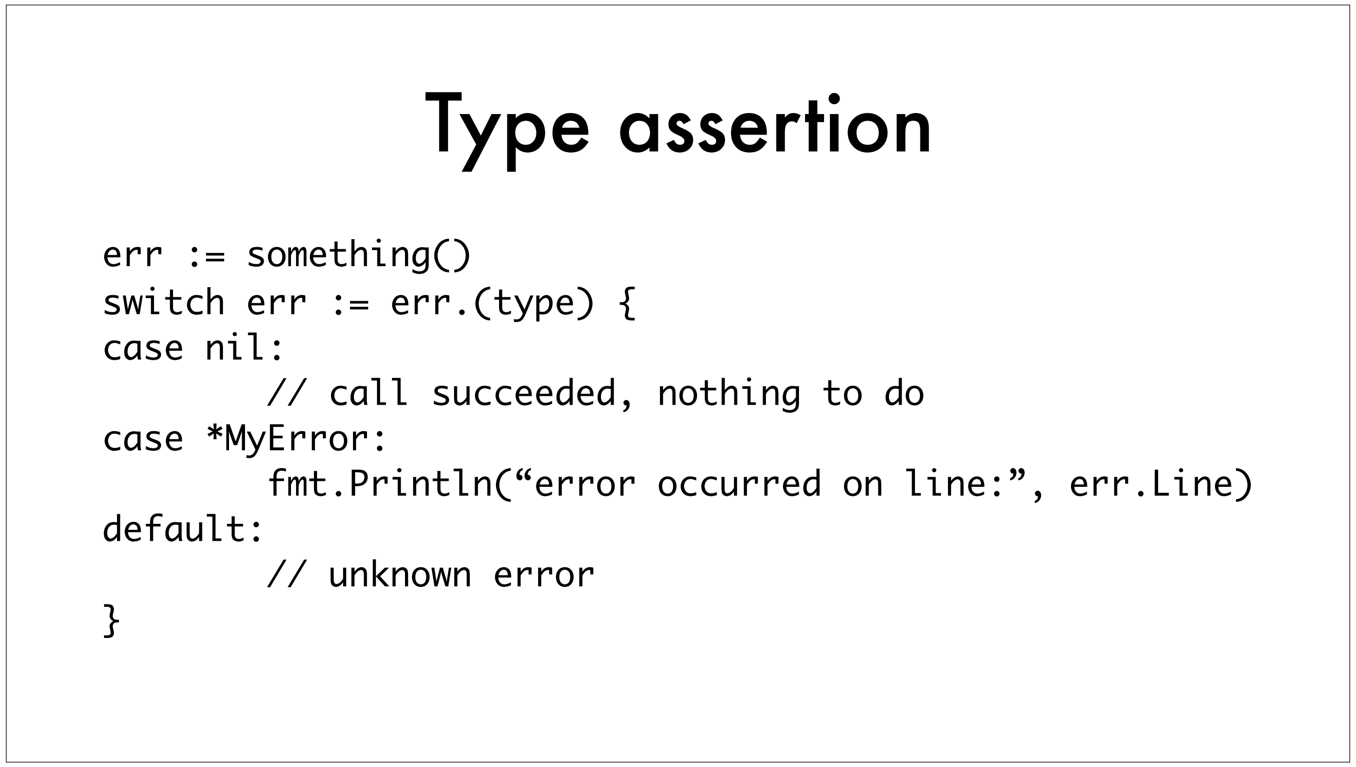


Error type

```
type MyError struct {
        Msg string
        File string
        Line int
}

func (e *MyError) Error() string {
        return fmt.Sprintf("%s:%d: %s”, e.File, e.Line, e.Msg)
}

return &MyError{"Something happened", “server.go", 42}
```

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

In this example, the MyError type tracks both the file and line, as well as a message explaining what happened.



Type assertion

```
err := something()
switch err := err.(type) {
case nil:
        // call succeeded, nothing to do
case *MyError:
        fmt.Println(“error occurred on line:”, err.Line)
default:
        // unknown error
}
```

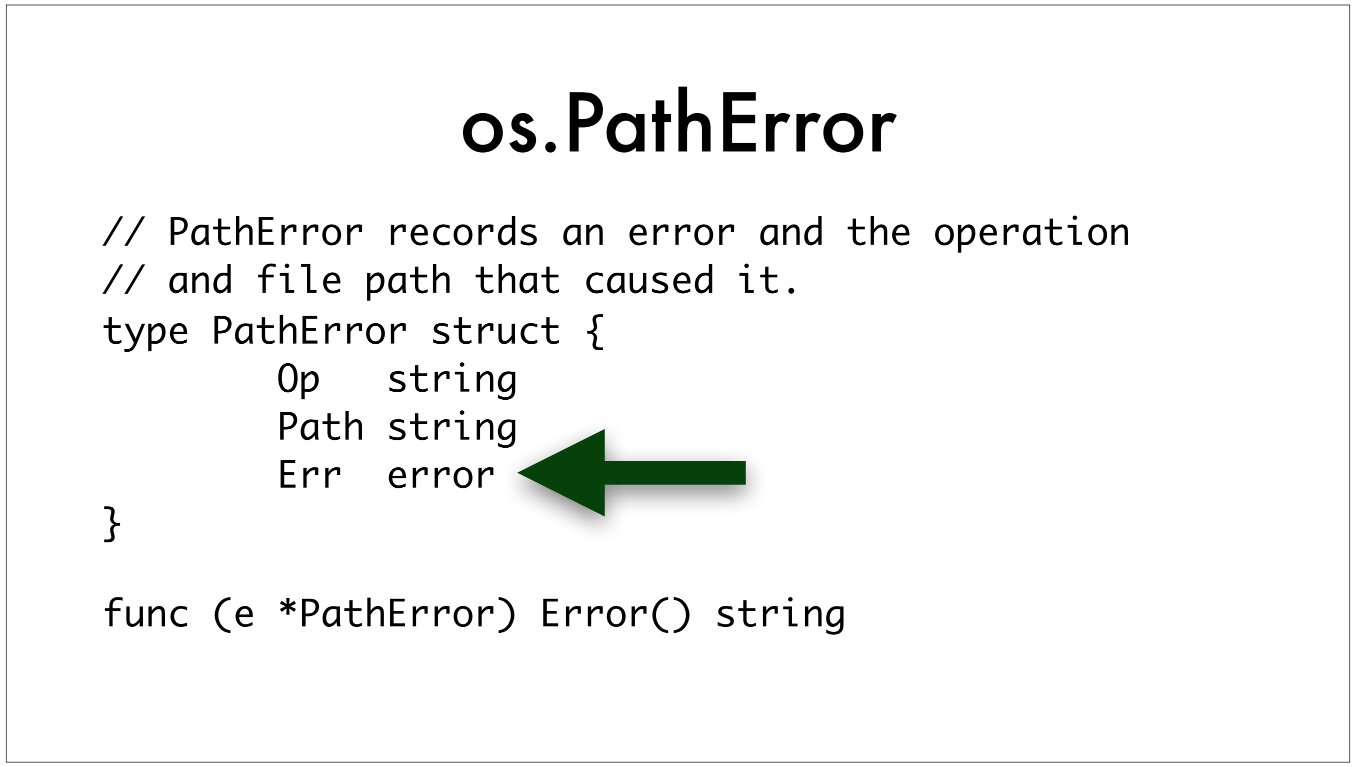
Because MyError error is a type, callers can use type assertion to extract the extra context from the error.

os.PathError

```
// 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
```

A big improvement of error types over error values is their ability to wrap an underlying error in a new type to provide more context.

An excellent example of this is the `os.PathError` type, which annotates the underlying error with the operation it was trying to perform, and the file it was trying to use.



os.PathError

```
// 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
```

A big improvement of error types over error values is their ability to wrap an underlying error in a new type to provide more context.

An excellent example of this is the `os.PathError` type, which annotates the underlying error with the operation it was trying to perform, and the file it was trying to use.

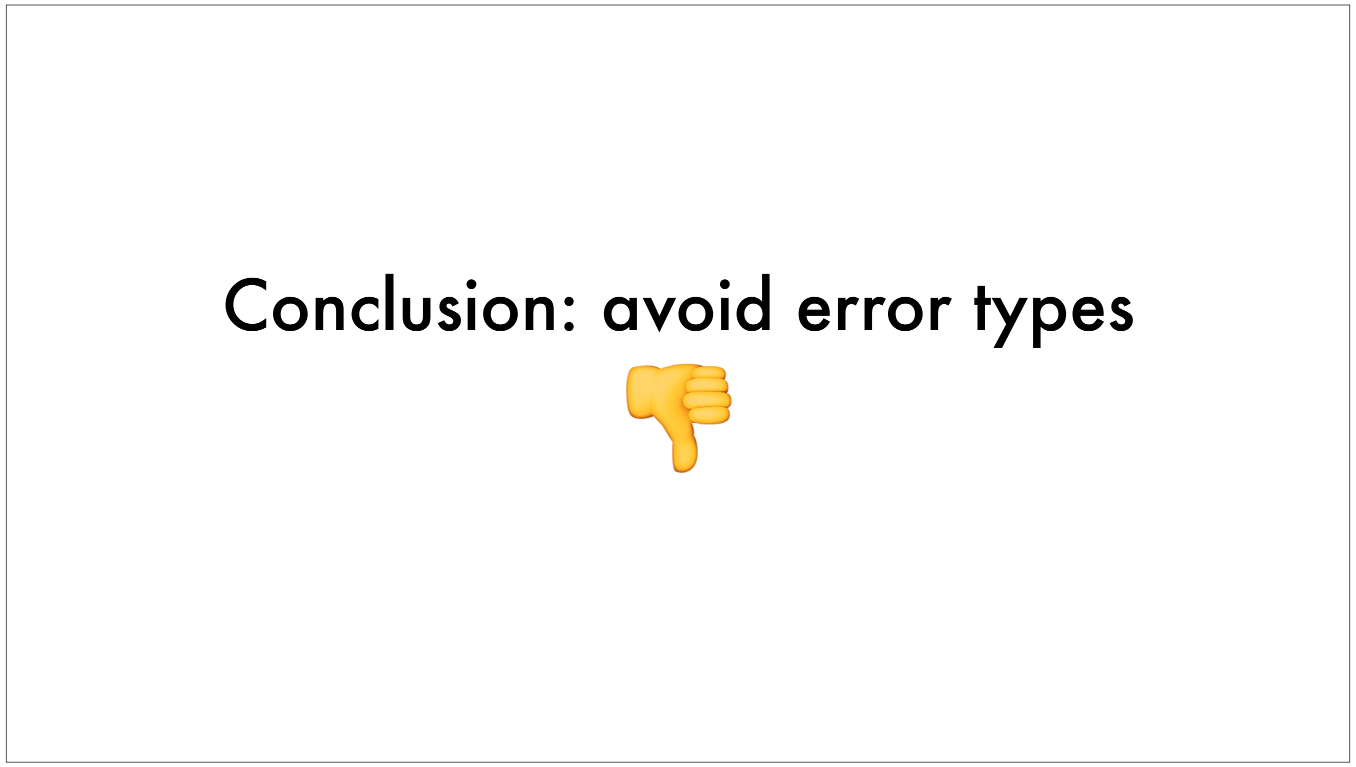
Problems with error types

💔

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.

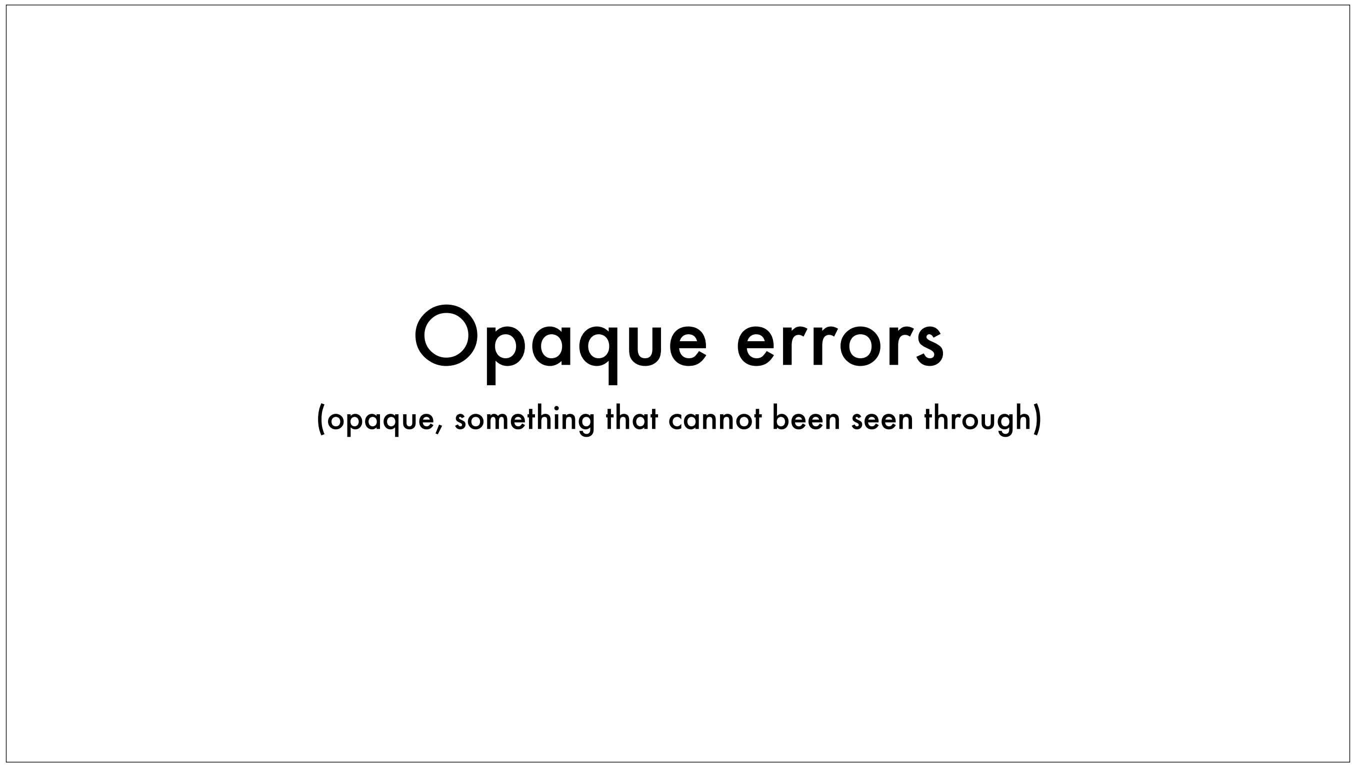


Conclusion: avoid error types

👎

So, while error types are better than sentinel 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.

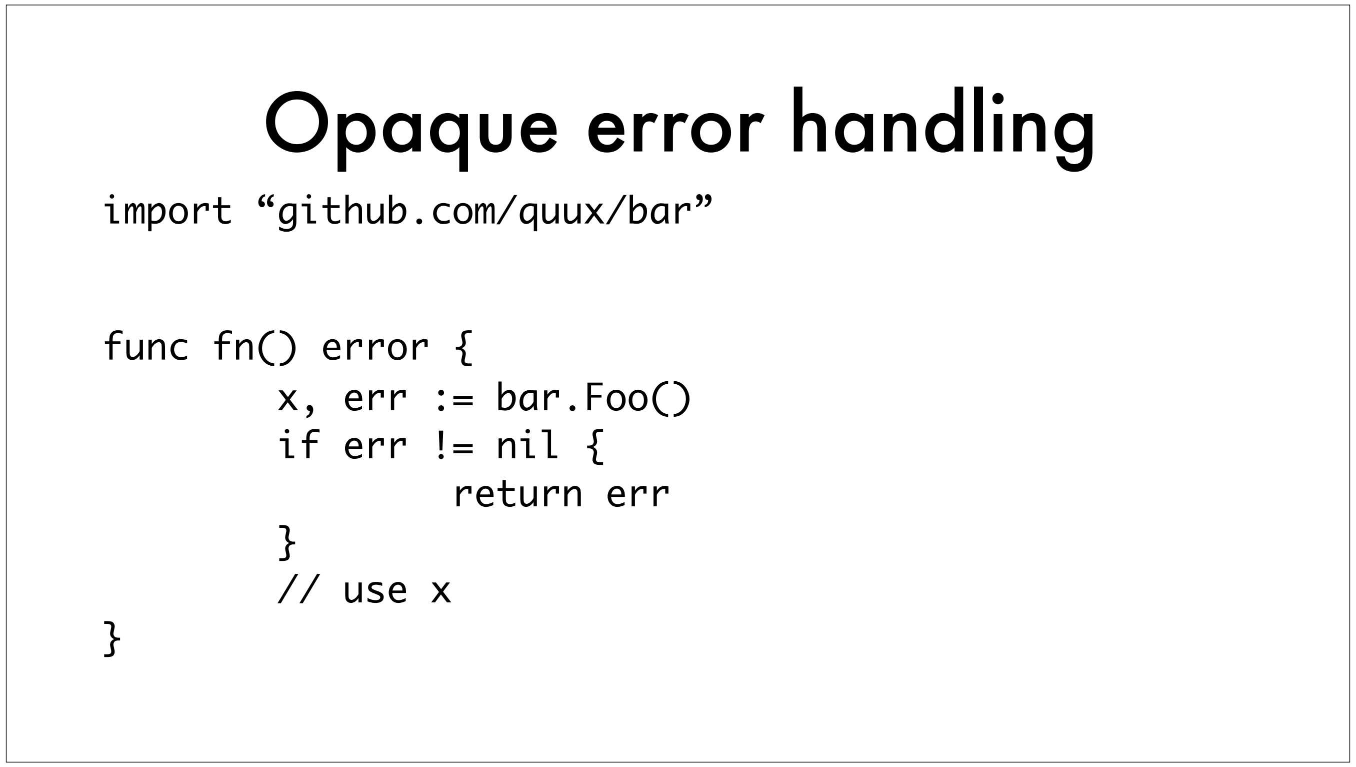


Opaque errors

(opaque, something that cannot been seen through)

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 the error.



Opaque error handling

```
import “github.com/quux/bar”


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

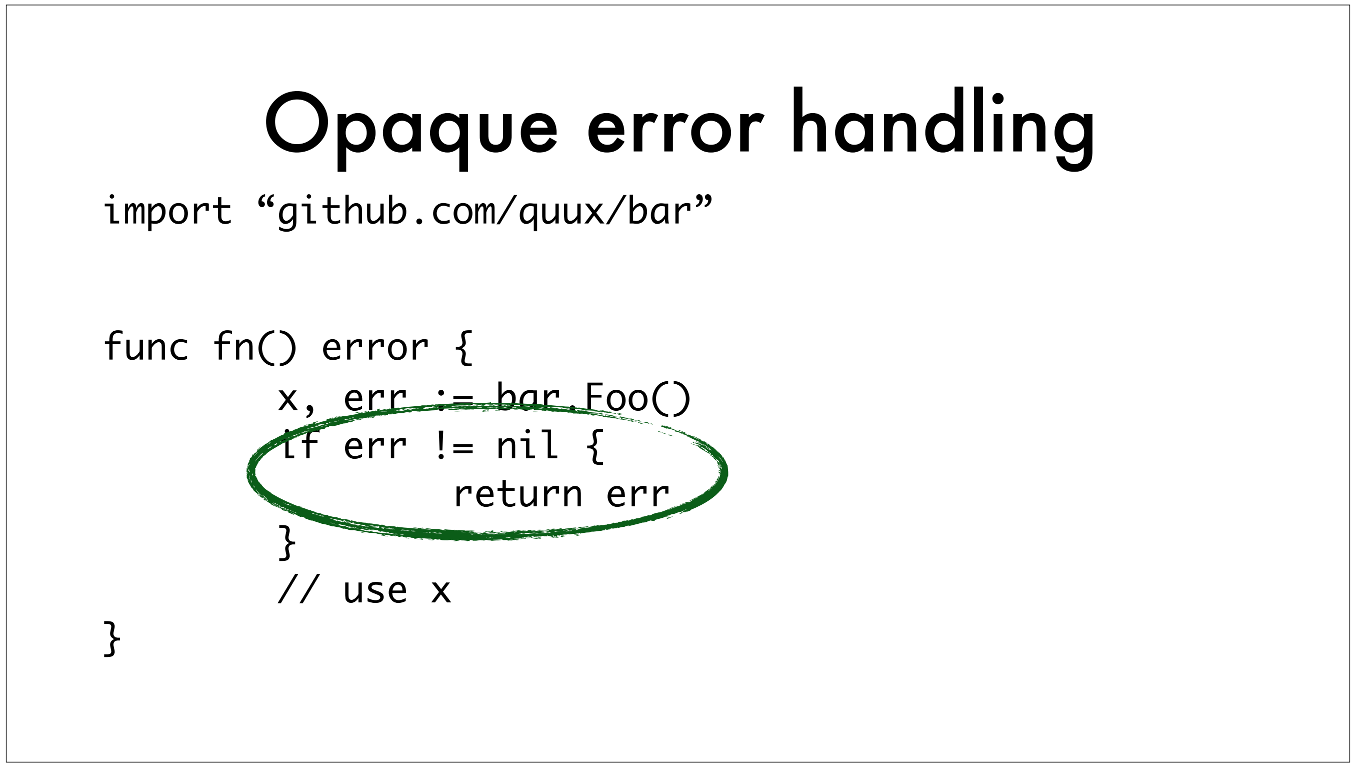
As the caller, all you know about the result of the operation is that it worked, or it didn’t.

If you adopt this position, then error handling becomes significantly more useful as a debugging aid.

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

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

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



Opaque error handling

```
import “github.com/quux/bar”


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

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

If you adopt this position, then error handling becomes significantly more useful as a debugging aid.

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

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

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 an error to IsTemporary to determine if the error 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.

If the error does implement Temporary, then perhaps the caller can retry if Temporary returns true.

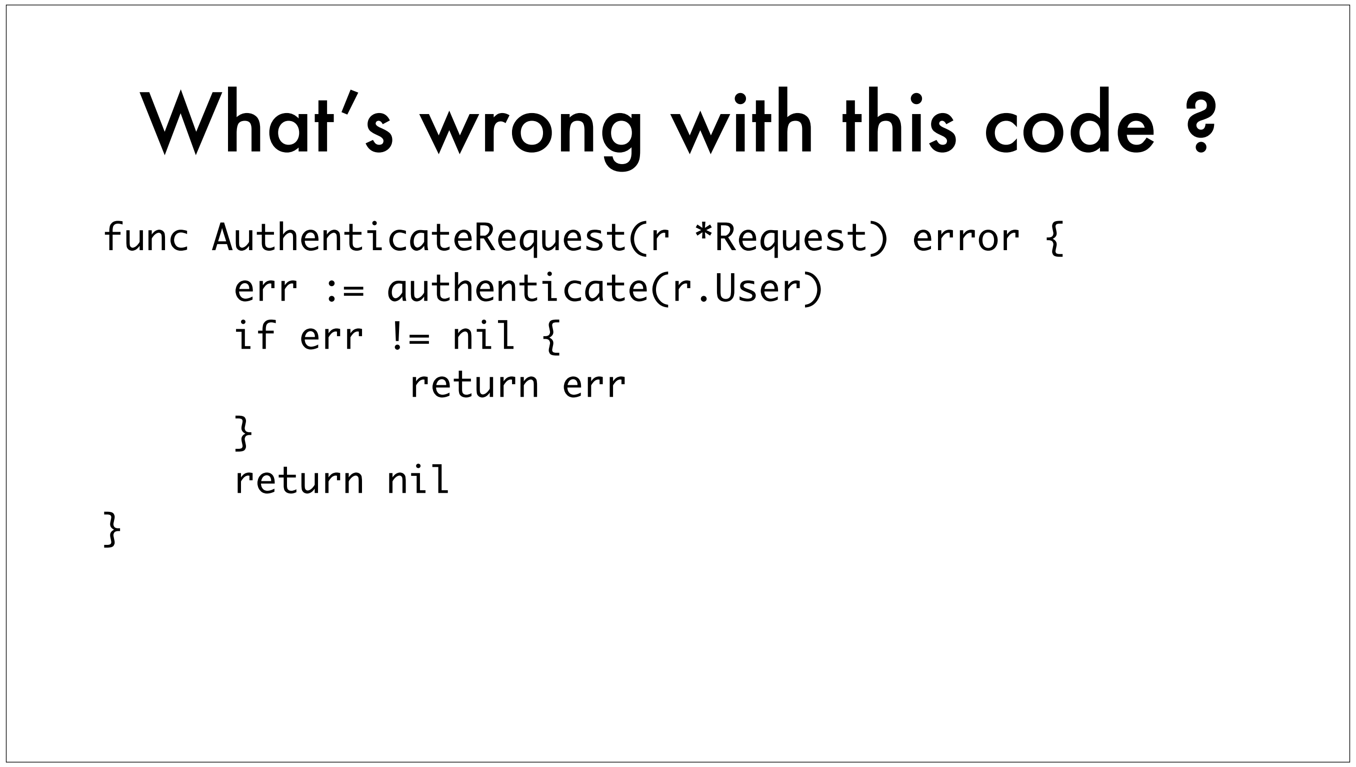
And the key here is all of this logic can be implemented without importing the package that defines the error or indeed knowing anything about err’s underlying type. We’re simply interested in its behaviours.

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.



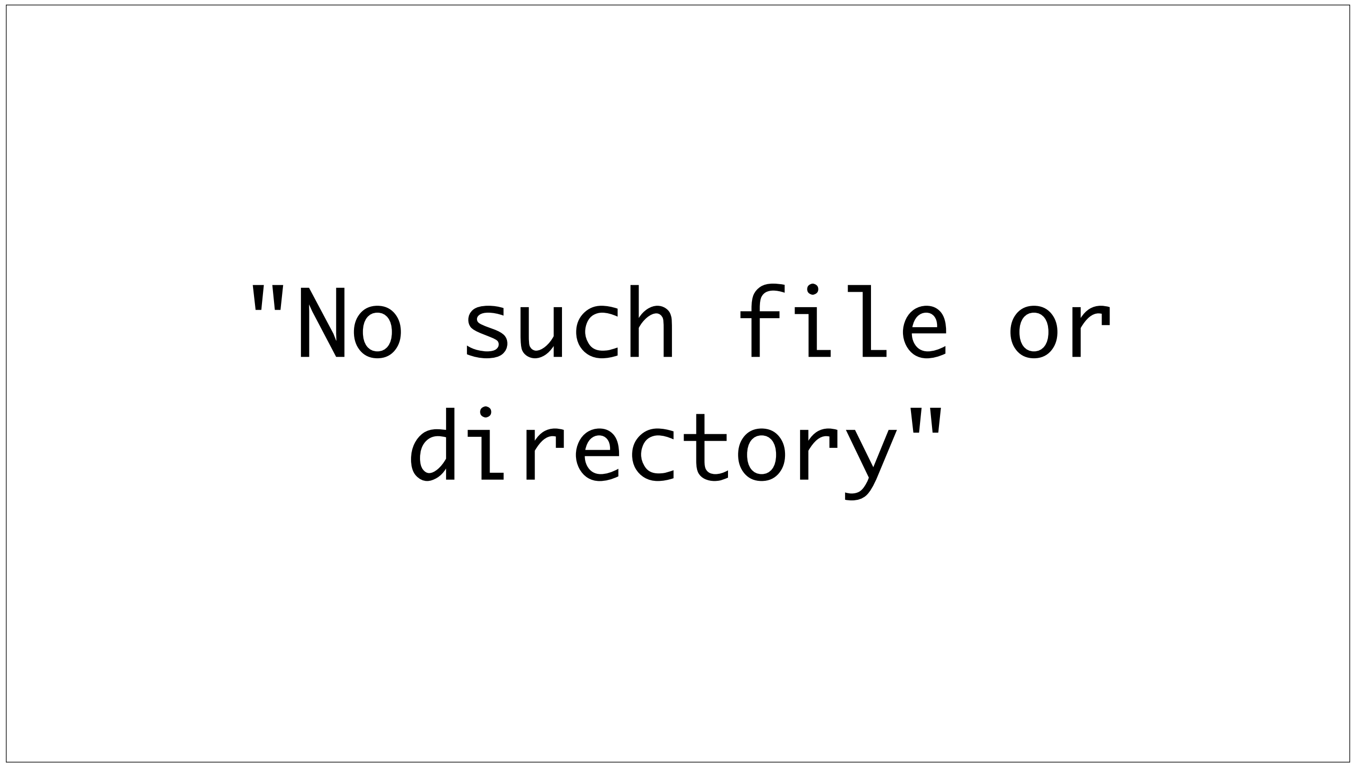
What’s wrong with this code ?

```
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?

To me at least, the problem with this code is I cannot tell where the original error came from.

At the top of my program I might print out the error to find the result is “No such file or directory".


```
"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.

"No such fi
directo

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)

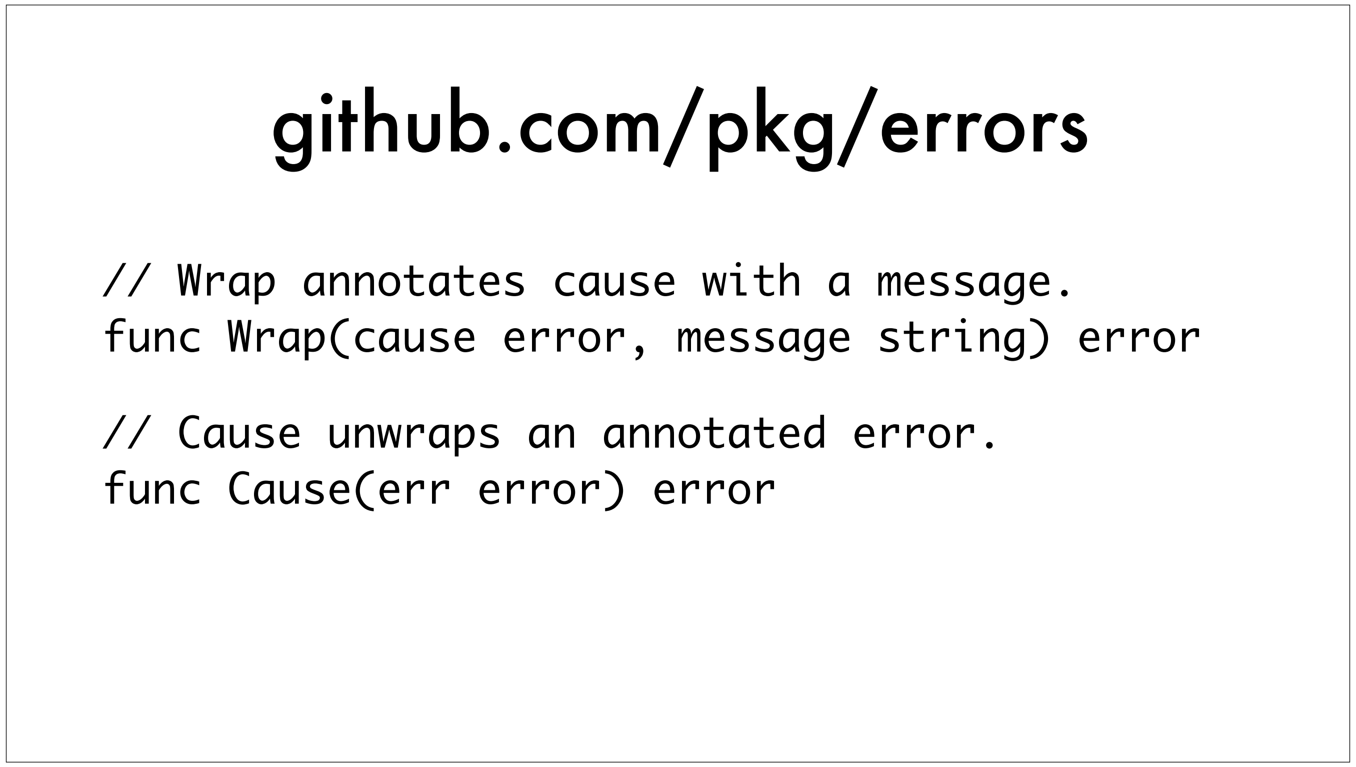
```
func AuthenticateRequest(r *Request) error {
	err := authenticate(r.User)
	if err != nil {
		return fmt.Errorf("authenticate failed: %v", err)
	}
	return nil
}
```

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 an error with fmt.Errorf breaks equality and destroys the context in the original error.

Annotating 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.



github.com/pkg/errors

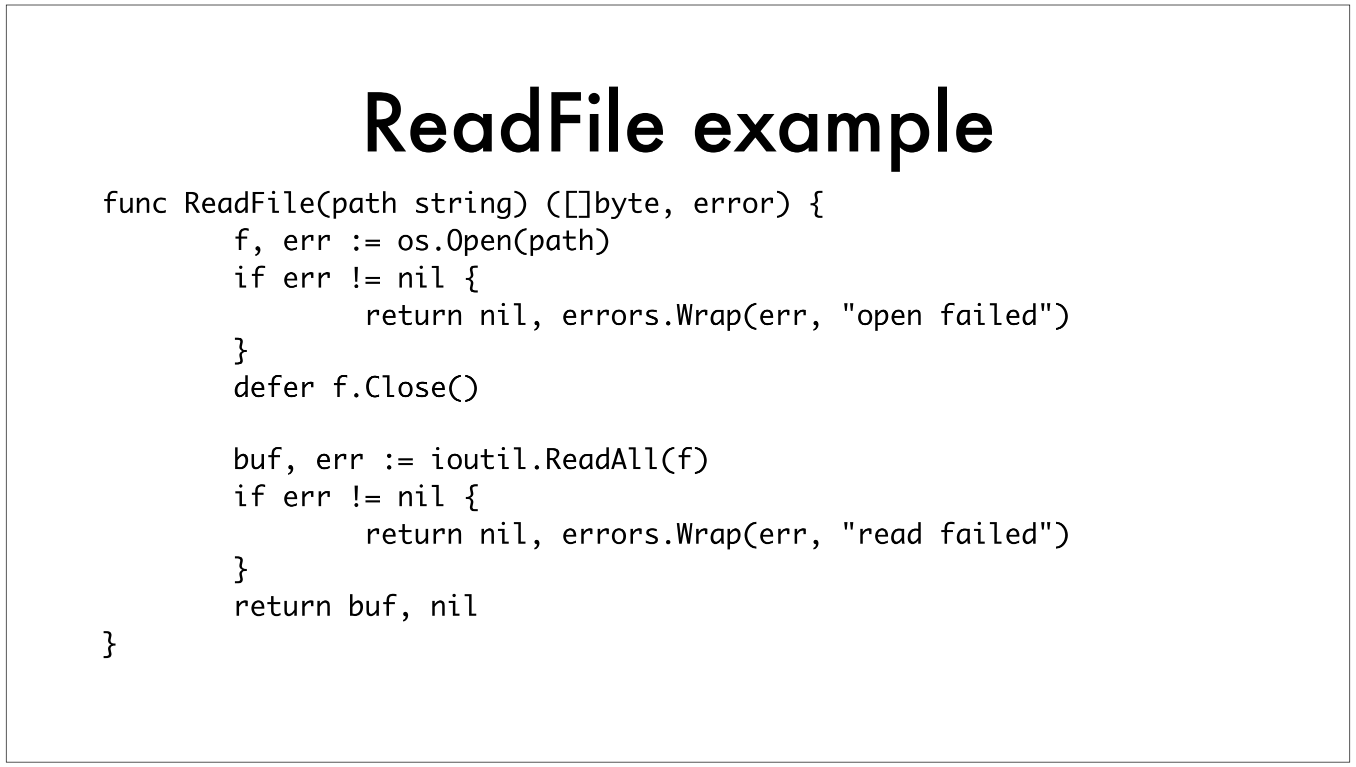
```
// Wrap annotates cause with a message.
func Wrap(cause error, message string) error

// Cause unwraps an annotated error.
func Cause(err error) error
```

The code is online at github.com/pkg/errors I'll just recap the API

The first function is wrap, which takes an error, and a message and produces a new error.

The second function is Cause, which takes an error that has possibly been wrapped, and unwraps it to recover the original error.

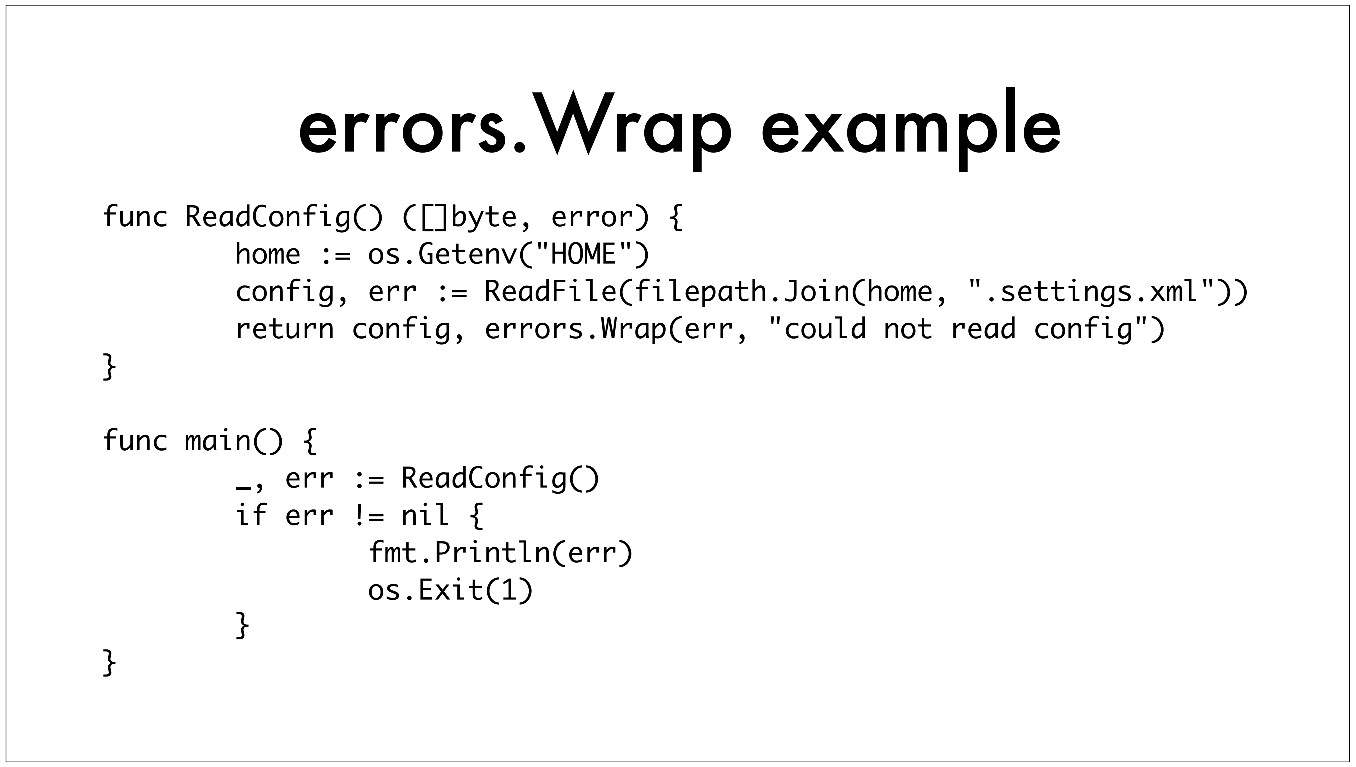
Using these two functions, we can now annotate any error, and recover the underlying error if we need to inspect it.



ReadFile example

```go
func ReadFile(path string) ([]byte, error) {
        f, err := os.Open(path)
        if err != nil {
                return nil, errors.Wrap(err, "open failed")
        }
        defer f.Close()

        buf, err := ioutil.ReadAll(f)
        if err != nil {
                return nil, errors.Wrap(err, "read failed")
        }
        return buf, nil
}
```

Consider this example of a function that reads the content of a file into memory.

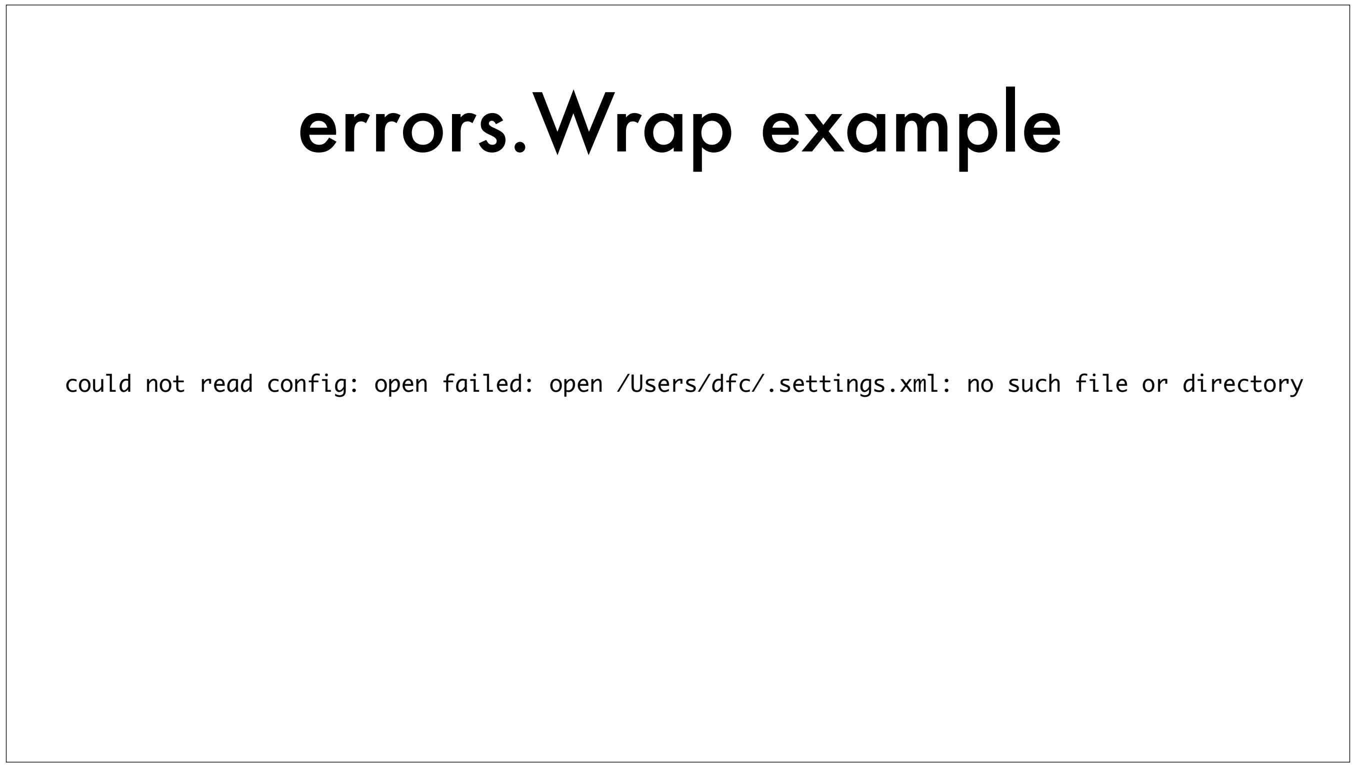


errors.Wrap example

```go
func ReadConfig() ([]byte, error) {
        home := os.Getenv("HOME")
        config, err := ReadFile(filepath.Join(home, ".settings.xml"))
        return config, errors.Wrap(err, "could not read config")
}

func main() {
        _, err := ReadConfig()
        if err != nil {
                fmt.Println(err)
                os.Exit(1)
        }
}
```

If we use this function to read a config file, now we get a nicely annotated error

If that code path fails, because we used errors.Wrap we get a nicely annotated error.

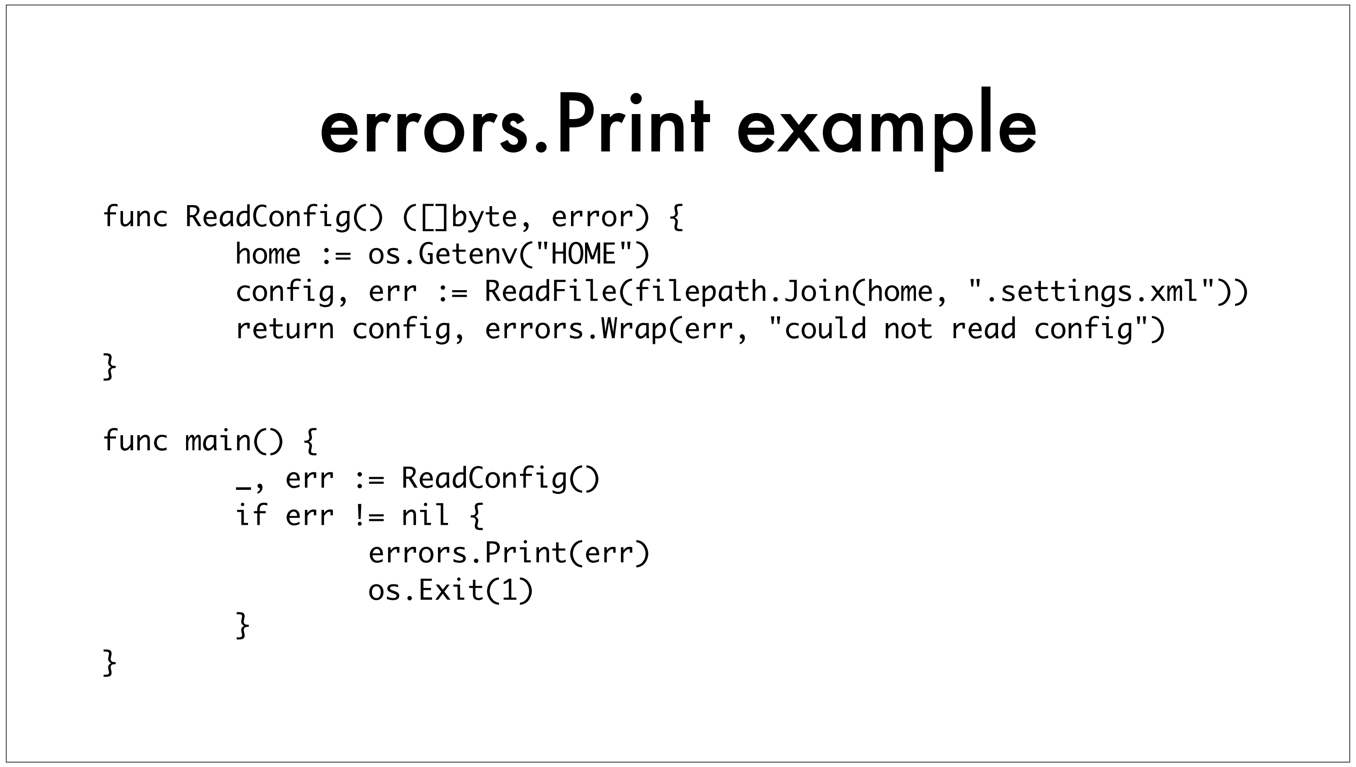


errors.Wrap example

```
could not read config: open failed: open /Users/dfc/.settings.xml: no such file or directory
```

If we use this function to read a config file, now we get a nicely annotated error

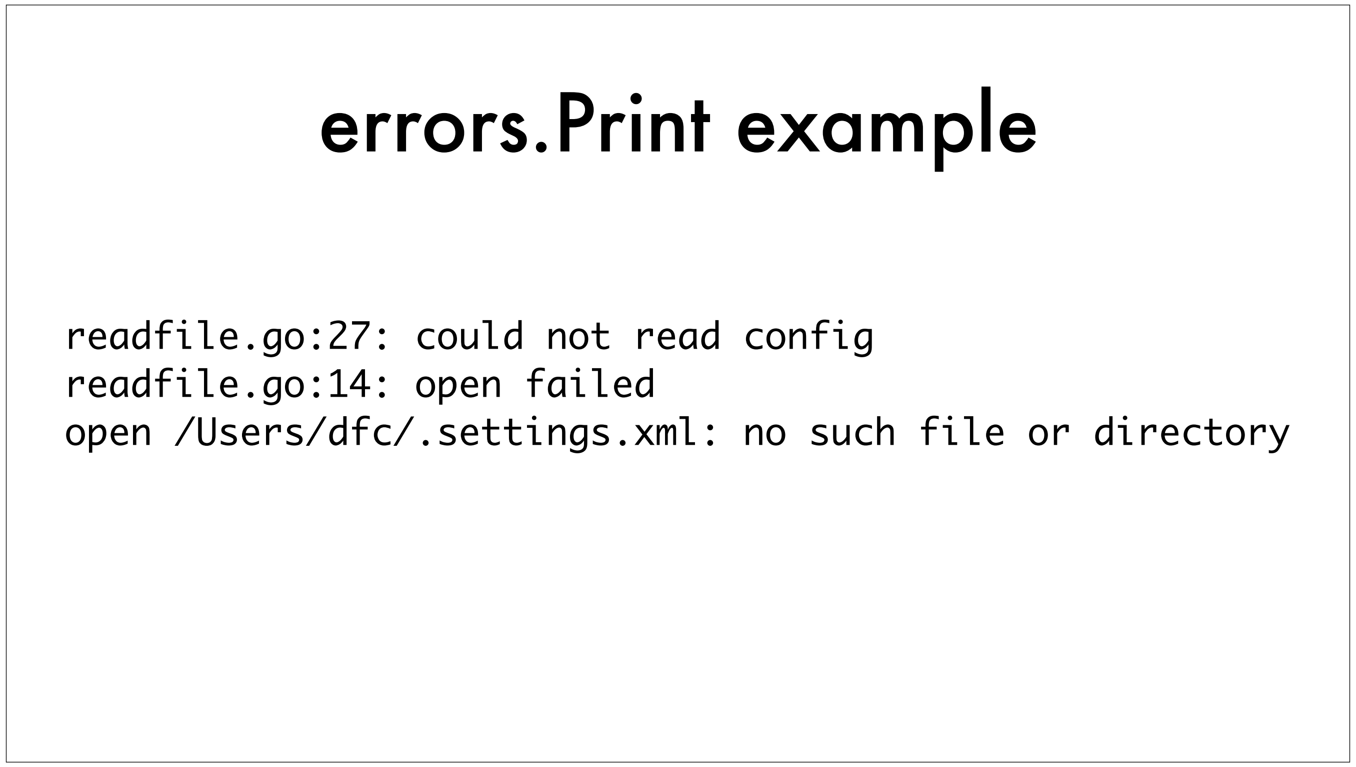
If that code path fails, because we used errors.Wrap we get a nicely annotated error.



errors.Print example

```
func ReadConfig() ([]byte, error) {
        home := os.Getenv("HOME")
        config, err := ReadFile(filepath.Join(home, ".settings.xml"))
        return config, errors.Wrap(err, "could not read config")
}

func main() {
        _, err := ReadConfig()
        if err != nil {
                errors.Print(err)
                os.Exit(1)
        }
}
```

This is the same example again, but this time we are using the Print function from the errors package.



errors.Print example

```
readfile.go:27: could not read config
readfile.go:14: open failed
open /Users/dfc/.settings.xml: no such file or directory
```

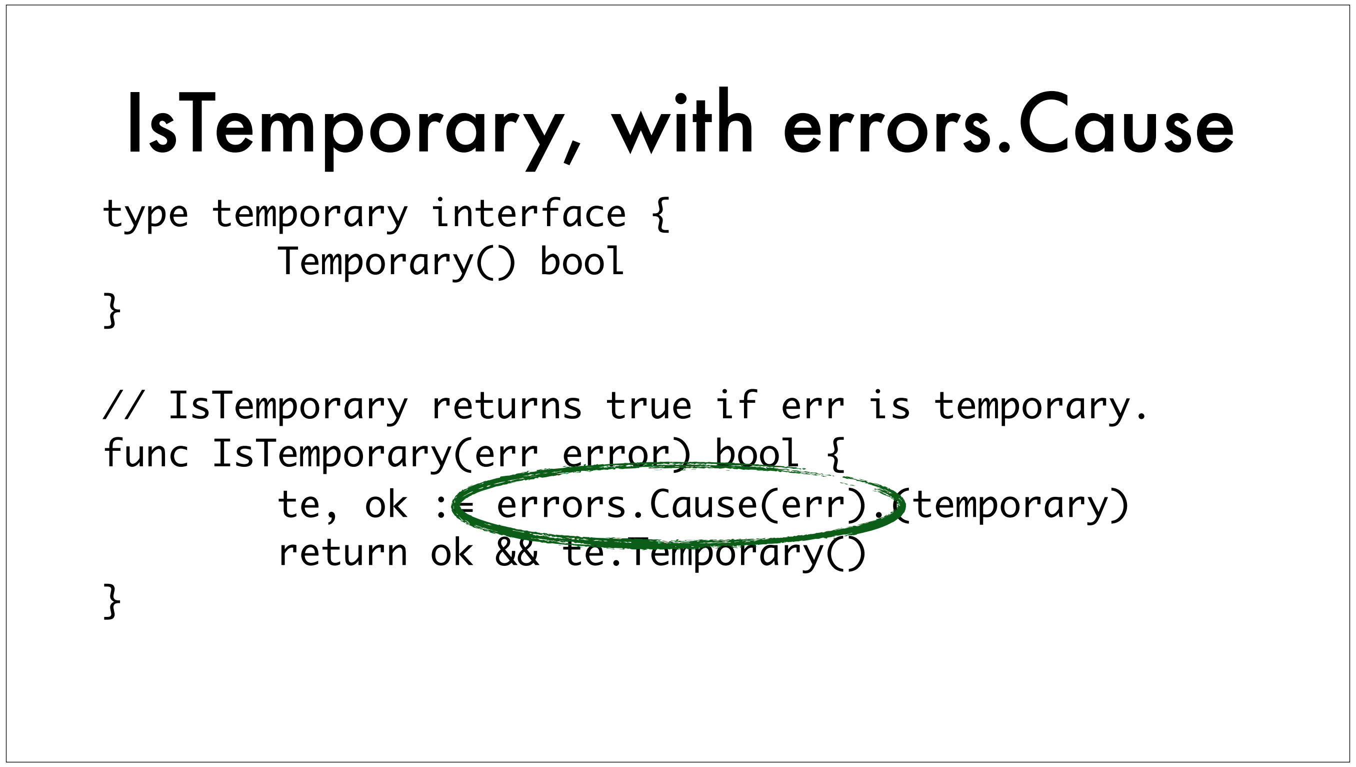
This is the same example again, but this time we are using the Print function from the errors package.

IsTemporary, with errors.Cause

```go
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()
}
```

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.

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



IsTemporary, with errors.Cause

```go
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()
}
```

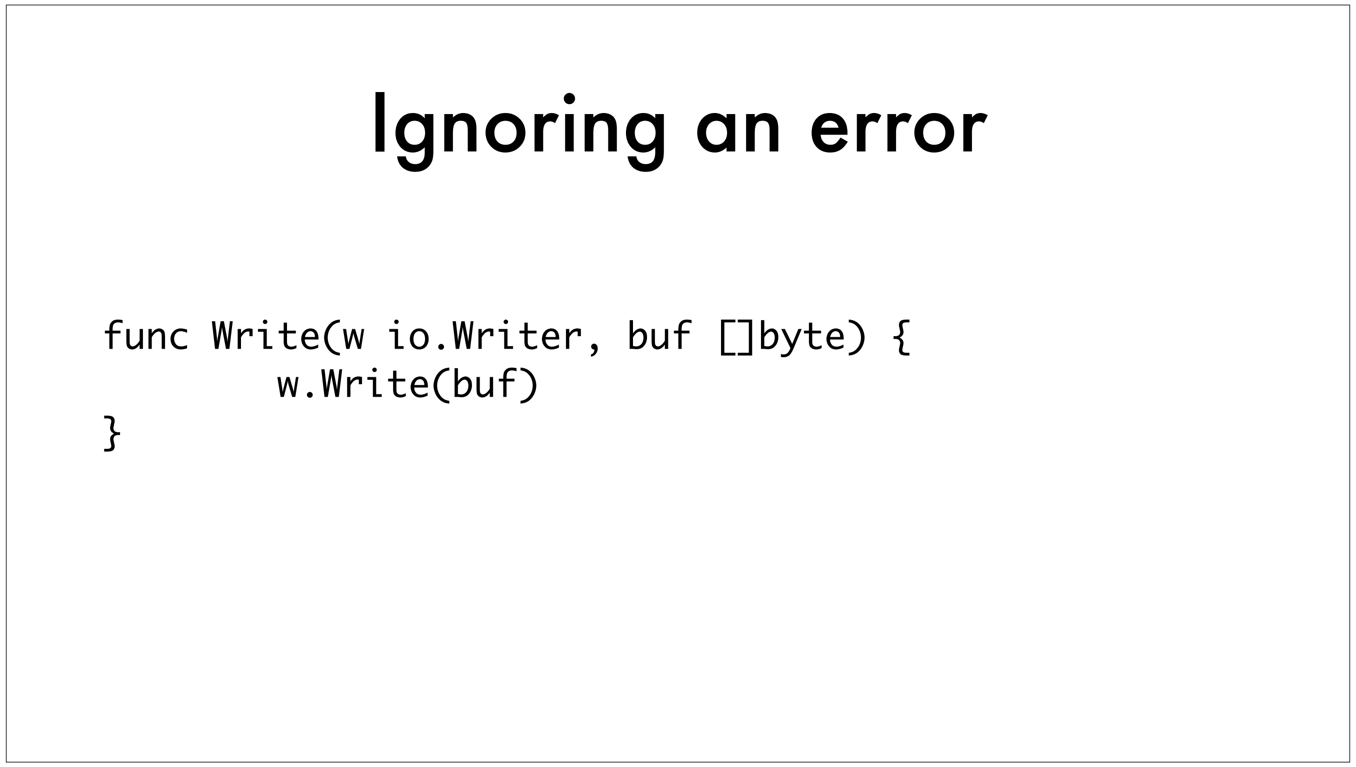
Because we've now introduced the concept of wrapping errors, we need to talk about the reverse, unwrapping them.
This is the domain of the `errors.Cause` function.

In operation, whenever you want to check an error matches a specific value or type, you should first 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.

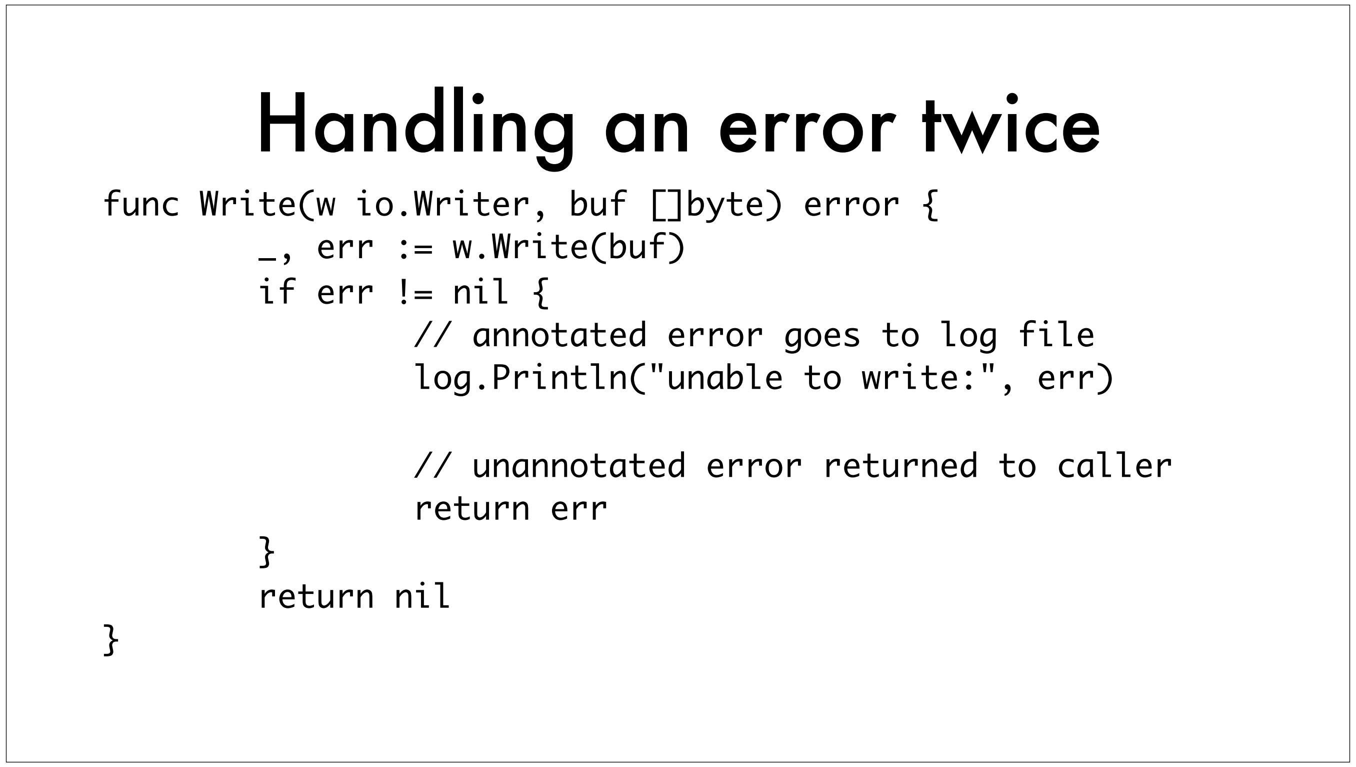
Handling an error means inspecting the error value, and making a decision.



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 being discarded.



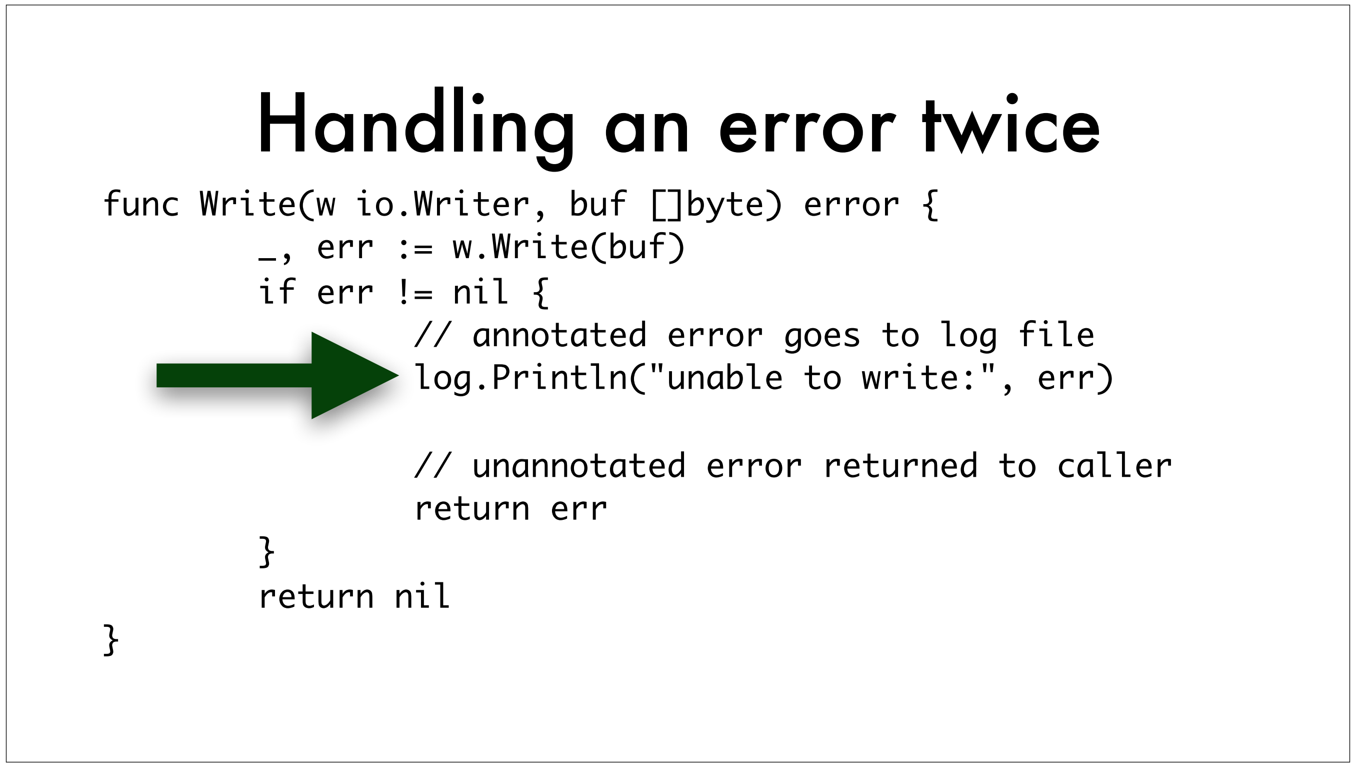
Handling an error twice

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

So 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.



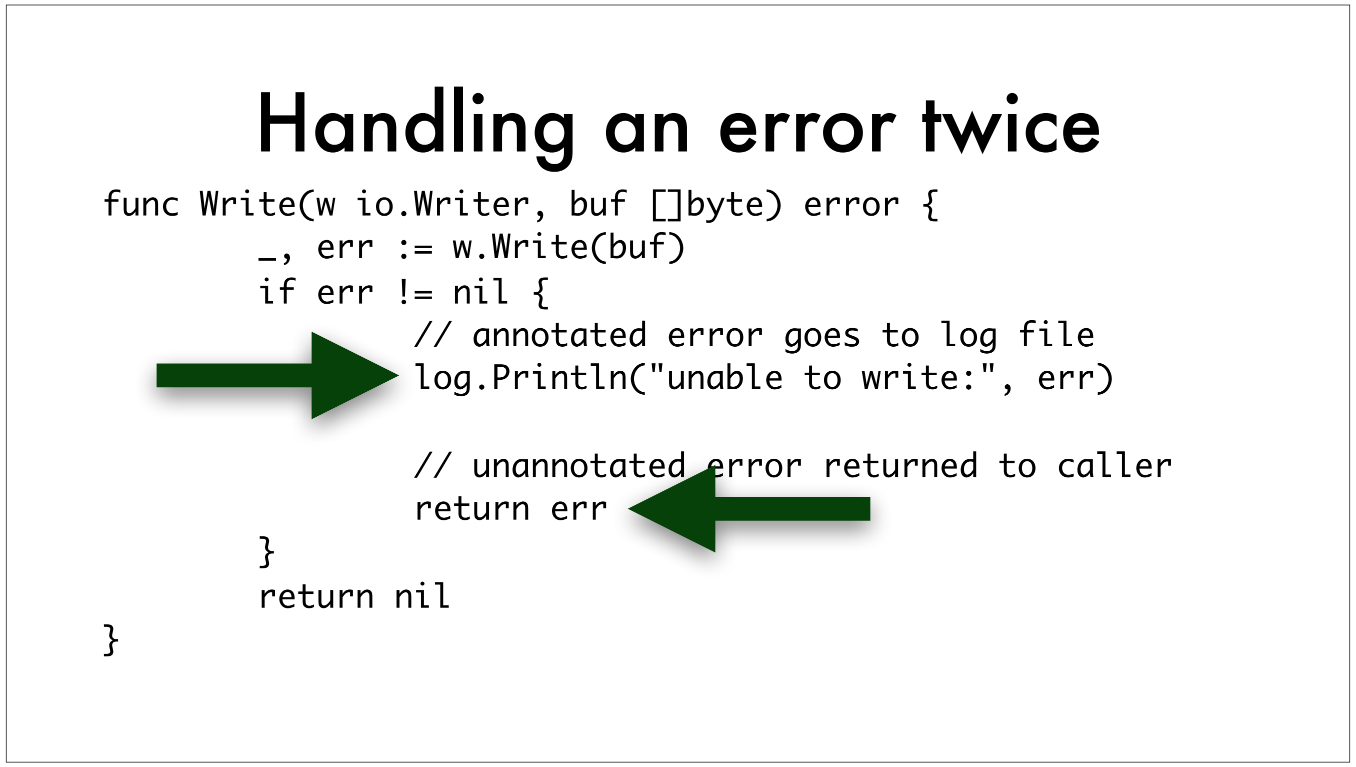
Handling an error twice

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

So 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.



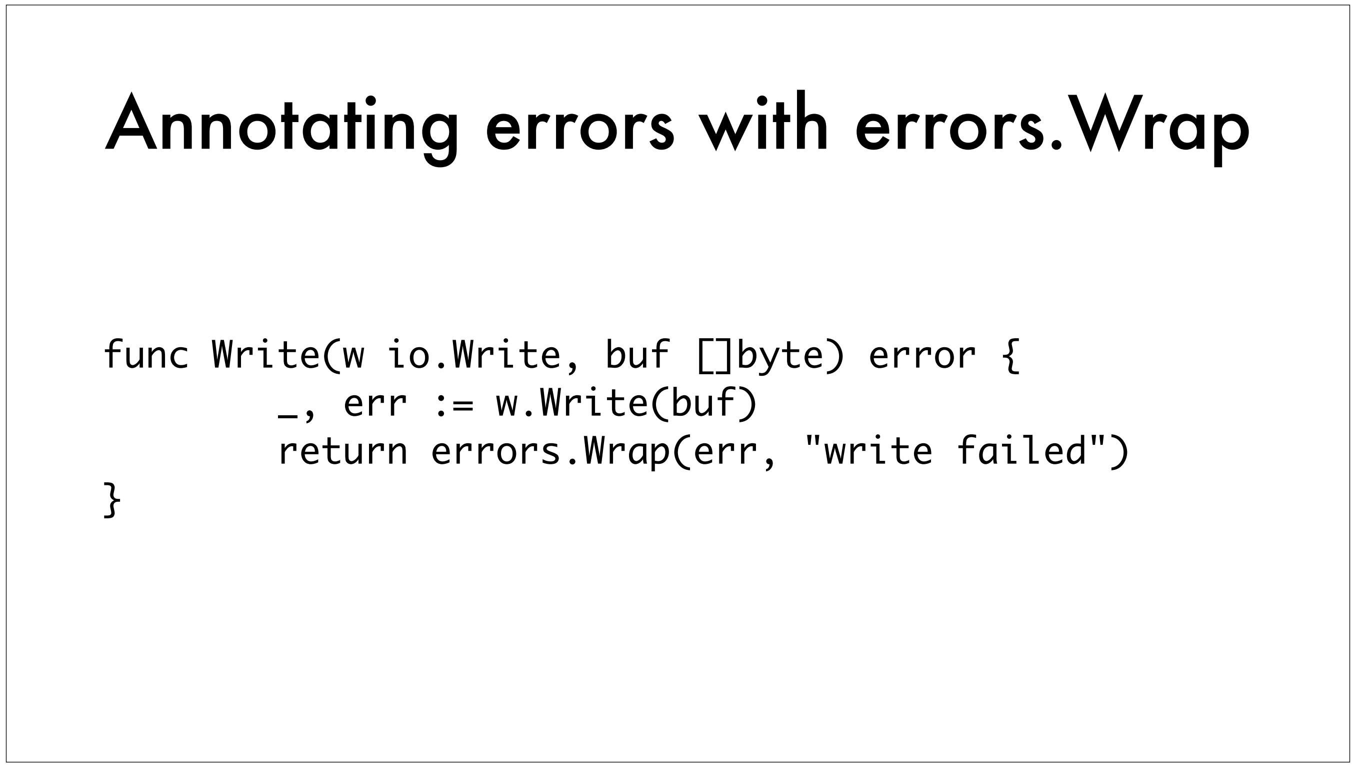
Handling an error twice

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

So 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.



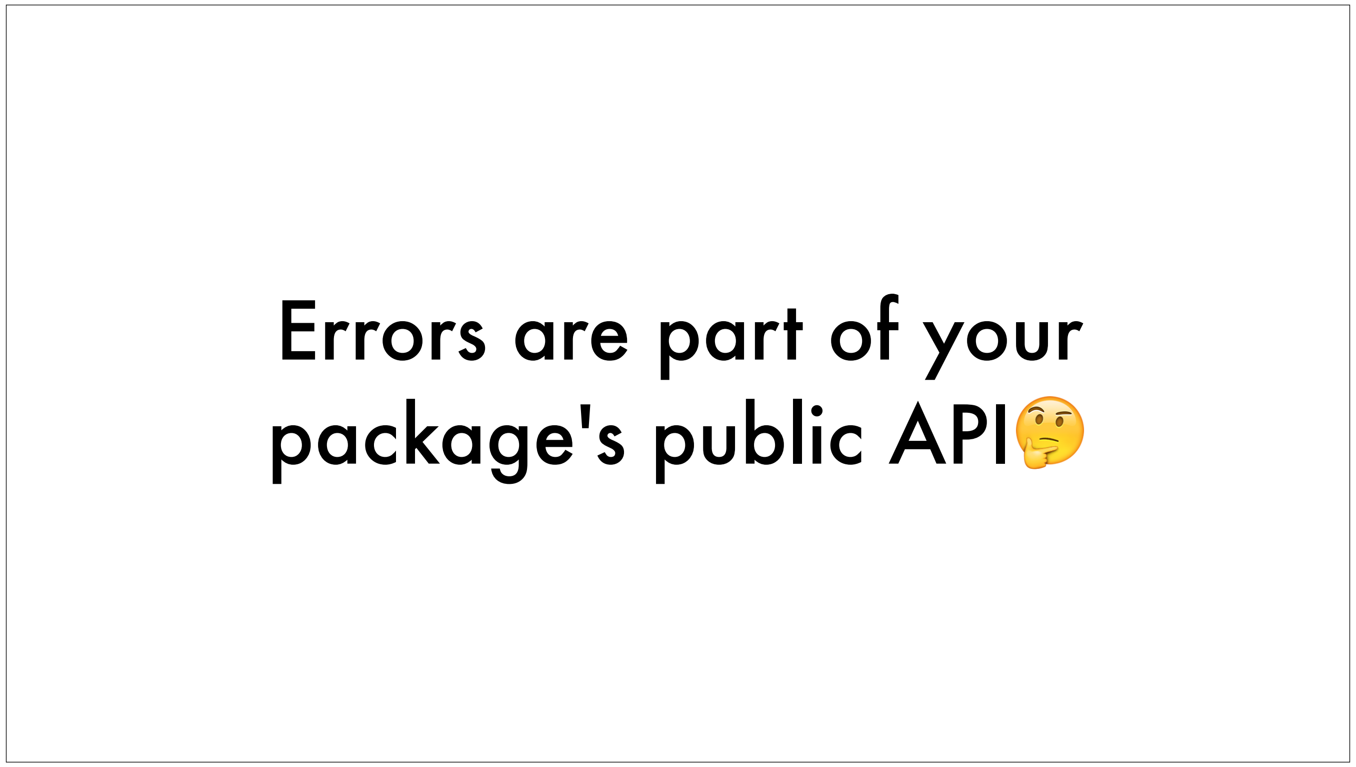
Annotating errors with errors.Wrap

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

Using the errors package gives you the ability to add context to error values, in a way that is inspectable by both a human and a machine.

Conclusion

In conclusion



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 type.

Minimise the use of sentinel error values in your program

Minimise the number of sentinel error values in your program.

Convert errors to opaque errors with errors.Wrap

Convert errors to opaque errors by wrapping them as soon as they occur.

Use errors.Cause to recover the underlying error

Use `errors.Cause` to recover the underlying error if you need to inspect it.

Proverbs are just stories
(with a lesson)

Proverbs aren't 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 message behind the proverb.

I hope that my explanations have been informative, and I will leave you to consider the meaning behind the other proverbs which I did not have time to discuss today.

I hope that you will watch Sensei Pike's video, I think it was a wonderful presentation and I cannot recommend it enough to each of you.

ありがとう

@davecheney

Thank you.