The Hidden Power of Humble Interfaces

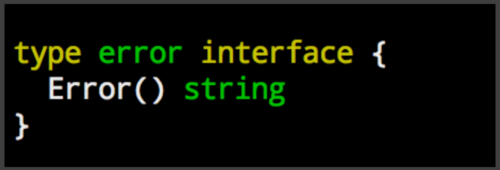
Aditya Mukerjee
@ChimeraCoder

GopherCon India 2016

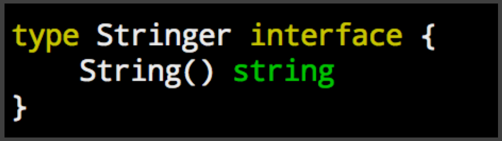
Why use interfaces?

- Write less code
- Write robust code
- Write flexible code

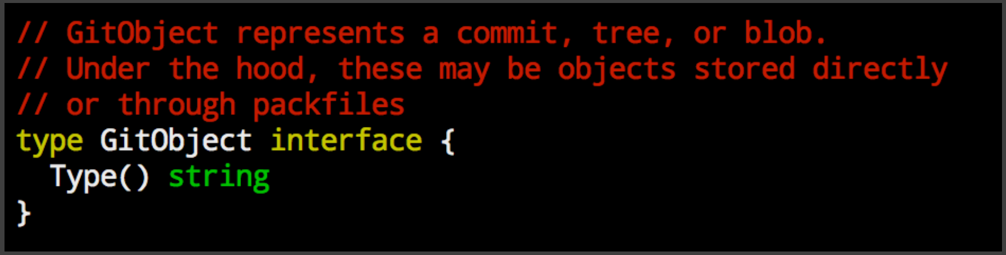
Interfaces: The Basics



```go
type error interface {
  Error() string
}
```

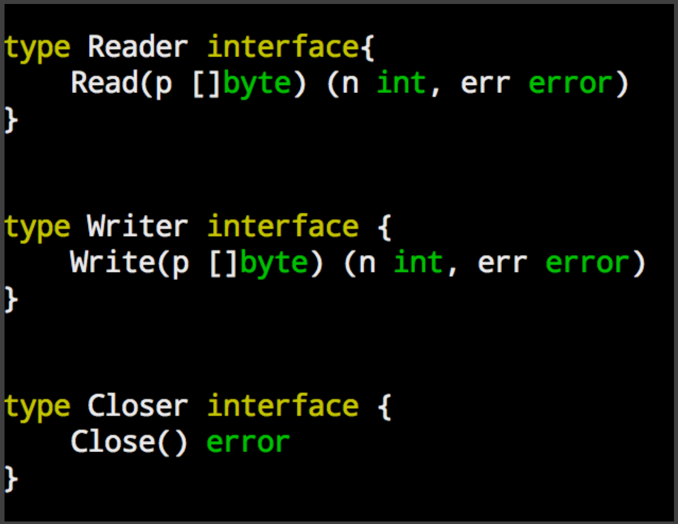


```go
type Stringer interface {
    String() string
}
```



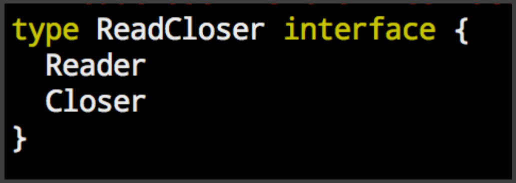
```go
// GitObject represents a commit, tree, or blob.
// Under the hood, these may be objects stored directly
// or through packfiles
type GitObject interface {
  Type() string
}
```

io.* interfaces

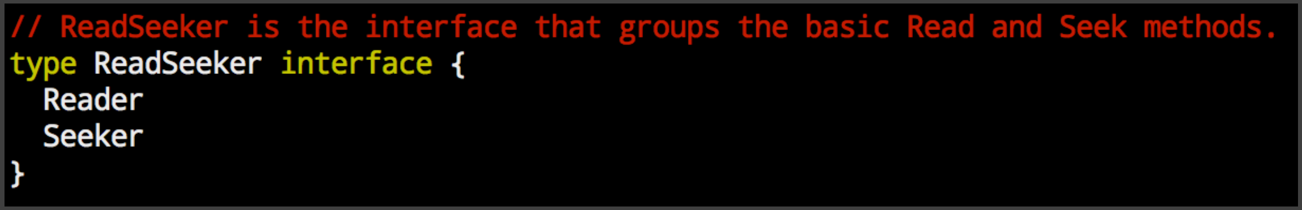


```
type Reader interface{
    Read(p []byte) (n int, err error)
}


type Writer interface {
    Write(p []byte) (n int, err error)
}


type Closer interface {
    Close() error
}
```

Composite interfaces

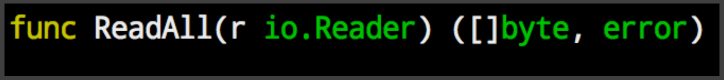


```
type ReadCloser interface {
  Reader
  Closer
}
```

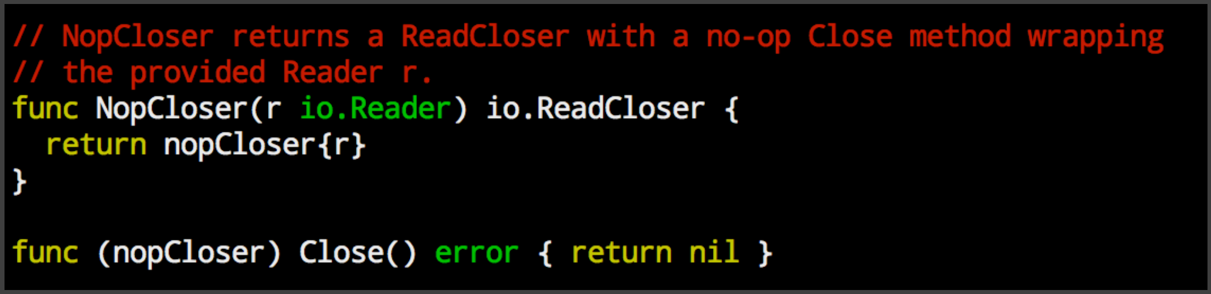


```
// ReadSeeker is the interface that groups the basic Read and Seek methods.
type ReadSeeker interface {
  Reader
  Seeker
}
```

Interface helper functions



```
func ReadAll(r io.Reader) ([]byte, error)
```



```
// NopCloser returns a ReadCloser with a no-op Close method wrapping
// the provided Reader r.
func NopCloser(r io.Reader) io.ReadCloser {
  return nopCloser{r}
}

func (nopCloser) Close() error { return nil }
```

What makes the `io` interfaces powerful?

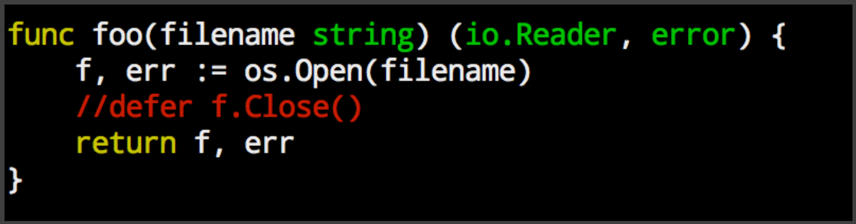
- Abstracting a lot of **common functionality**
- Lots of **granularity**
- Plethora of **helper functions**

What makes `error` powerful?

- Abstracts **no** functionality
- Provides **no** granularity
- Provides (almost) **no** helper functions

What makes `io.Reader` not powerful?

- Lifecycle management

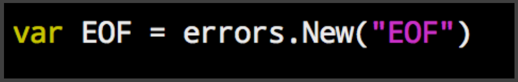


```go
func foo(filename string) (io.Reader, error) {
    f, err := os.Open(filename)
    //defer f.Close()
    return f, err
}
```

- Impedance mismatch of exact methods required

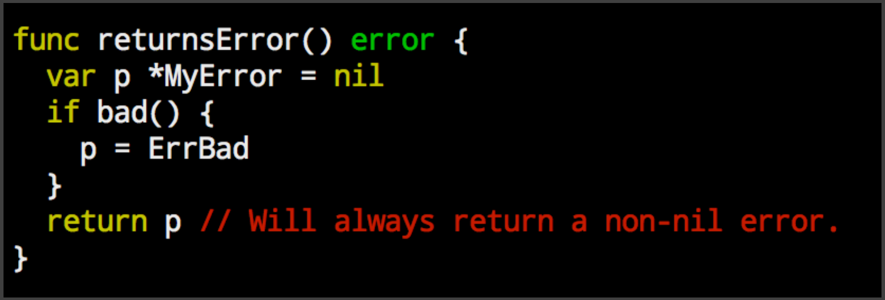
What makes `error` not powerful?

- Inconsistent convention around sentinel error values



```go
var EOF = errors.New("EOF")
```

- `error == nil`

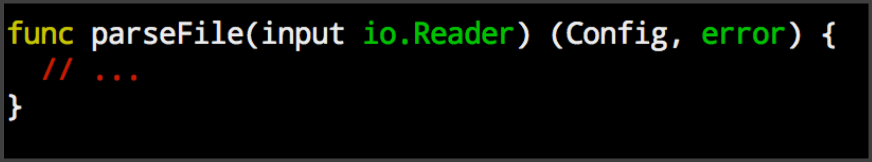


```go
func returnsError() error {
  var p *MyError = nil
  if bad() {
    p = ErrBad
  }
  return p // Will always return a non-nil error.
}
```

What makes a powerful interface?

Lessons we can learn

- Keep interfaces **humble**
  - Writing interfaces forces you to **define** the minimum required contract for using your types



```
func parseFile(input io.Reader) (Config, error) {
  // ...
}
```

- Keep interfaces **disciplined**
  - Writing interfaces allows the compiler to **enforce** this contract

```
type Foo ~~struct~~ interface {

}
```

Why Gophers avoid interfaces

- Afraid of writing an interface that is too complicated
- Afraid of specifying the wrong methods
- Optimizing for memory usage and garbage collection
- Easy initialization from a literal value
- Preventing others from implementing the interface

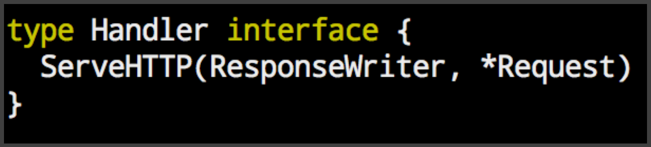
How I Learned To Stop Worrying and Love the `interface`

- Interfaces reveal the assumptions you’re already making
- “If it’s there, you will use it”

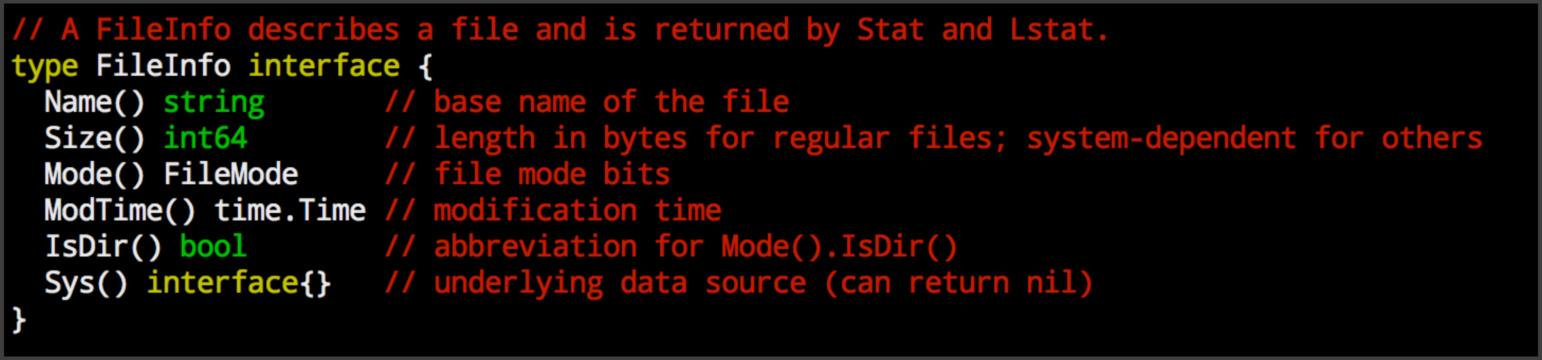
What kind of interface should I write?

Questions to ask yourself

- Is my interface **declarative** or **functional**?
- Do any interface methods require **complex transformation** of data?
- Will this interface have any closely-related ‘sibling’ interfaces?



```go
type Handler interface {
  ServeHTTP(ResponseWriter, *Request)
}
```



```go
// A FileInfo describes a file and is returned by Stat and Lstat.
type FileInfo interface {
  Name() string       // base name of the file
  Size() int64        // length in bytes for regular files; system-dependent for others
  Mode() FileMode     // file mode bits
  ModTime() time.Time // modification time
  IsDir() bool        // abbreviation for Mode().IsDir()
  Sys() interface{}   // underlying data source (can return nil)
}
```

And *how* should I design these interfaces?

Writing interfaces like `io.*`

- Don’t write helper methods until you actually find you need them
- The contracts provided by each should be **minimal**
- Create composite interface types

Writing interfaces like `error`

- Provide canonical sentinel values, if relevant
- Provide a default implementation, if relevant

Techniques for fine-tuning interfaces

- Use an unexported method in an interface to restrict implementation
- Pair exported structs with an interface type that is used in all function signatures
- Create unexported structs to implement your interfaces

What if structs could never be exported?

Aditya Mukerjee
@chimeracoder
https://github.com/ChimeraCoder