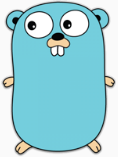
Programming in Go

Matt Holiday
Christmas 2020



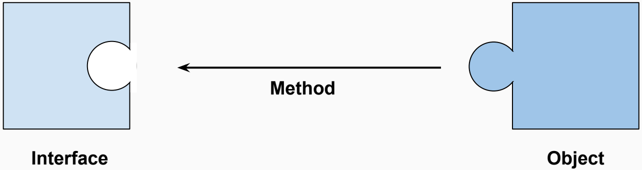
Methods and Interfaces

Why have methods?

An **interface** specifies *abstract* behavior in terms of **methods**

```go
type Stringer interface { // in "fmt"
    String() string
}
```

Concrete types offer methods that *satisfy* the interface



Methods are type-bound functions

A **method** is a special type of function (syntax from Oberon-2)

It has a **receiver** parameter *before* the function name parameter

```
type IntSlice []int

func (is IntSlice) String() string {
    strs []string

    for _, v := range is {
        strs = append(strs, strconv.Itoa(v))
    }

    return "[" + strings.Join(strs, ";") + "]"
}
```

Why have methods?

Only **methods** may be used to satisfy an **interface**

```
func main() {
    var v IntSlice = []int{1, 2, 3}
    var s fmt.Stringer = v

    for i, x := range v {
        fmt.Printf("%d: %d\n", i, x)
    }

    fmt.Printf("%T %[1]v\n", s)
    fmt.Printf("%T %[1]v\n", v)  // Uses String() method (if available)
}
```

An `IntSlice` value **"is a"** `fmt.Stringer` because it implements the `String()` method

Why interfaces?

Without interfaces, we'd have to write (many) functions
for (many) concrete types, possibly coupled to them

```
func OutputToFile(f *File, . . .) { . . . }
func OutputToBuffer(b *Buffer, . . .) { . . . }
func OutputToSocket(s *Socket, . . .) { . . . }
```

Better — we want to define our function in terms of *abstract behavior*

```
type Writer interface {
    Write([]byte) (int, error)
}

func OutputTo(w io.Writer, . . . ) { . . . }
```

Why interfaces?

An interface specifies required behavior as a **method set**

Any type that implements that method set satisfies the interface:

```
type Stringer interface { // in "fmt"
    String() string
}

func (is IntSlice) String() string {
    . . .
}
```

This is known as *structural* typing (“duck” typing)

No type will declare itself to implement `ReadWriter` explicitly

Not just structs

A method may be defined on any **user-declared** (named) type*

That means methods can't be declared on `int`, but

```
type MyInt int

func (i MyInt) String() string {
    . . .
}
```

The same method name may be bound to different types

* Some rules and restrictions apply, see package insert for details

Receivers

A method may take a *pointer* or *value* receiver, but not both

```go
type Point struct {
    X, Y float64
}

func (p Point) Offset(x, y float64) Point {
    return Point{p.x+x, p.y+y}
}

func (p *Point) Move(x, y float64) {
    p.x += x
    p.y += y
}
```

Taking a pointer allows the method to change the receiver (original object)

Interface variables

A variable of interface type can refer to any object that satisfies it

```
func Copy(w Writer, r Reader) (int, error) { // in "io"
    . . .
}

f1, err := os.Open("input.txt")
f2, err := os.Create("output.txt")

n, err := io.Copy(f2, f1)
```

Here `w` and `r` are references ultimately to files

But it could be a `File` and a `bytes.Buffer` source; it wouldn't care — all it needs is the specific behaviors (write & read)

Interface example

```go
type ByteCounter int

func (b *ByteCounter) Write(p []byte) (int, error) {
    *b += ByteCounter(len(p))  // conversion required
    return len(p), nil
}

var c ByteCounter

f, _ := os.Open("input.txt")
n, _ := io.Copy(&c, f)        // &c required

fmt.Println(n == int64(c))    // true
```

Lots of types are `Writers` and can be written/copied to;
see also Francesc Campoy Interfaces in Go (2019)

Interfaces and substitution

All the methods must be present to satisfy the interface

```go
var w   io.Writer
var rwc io.ReadWriteCloser

w = os.Stdout           // OK: *os.File has Write method
w = new(bytes.Buffer)   // OK: *bytes.Buffer has Write method
w = time.Second         // ERROR: no Write method

rwc = os.Stdout         // OK: *os.File has all 3 methods
rwc = new(bytes.Buffer) // ERROR: no Close method

w = rwc                 // OK: io.ReadWriteCloser has Write
rwc = w                 // ERROR: no Close method
```

Which is why it pays to keep interfaces small

Interface satisfiability

The **receiver** must be of the right type (pointer or value)

```
type IntSet struct { /* ... */ }

func (*IntSet) String() string

var _ = IntSet{}.String() // ERROR: String needs *IntSet (l-value)

var s IntSet
var _ = s.String()        // OK: s is a variable; &s used automatically

var _ fmt.Stringer = &s   // OK
var _ fmt.Stringer = s    // ERROR: no String method
```

We'll come back and talk about pointer vs value receivers in more detail

Interface composition

`io.ReadWriter` is actually defined by Go as two interfaces

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

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

type ReadWriter interface {
    Reader
    Writer
}
```

Small interfaces with **composition** where needed are more flexible

Interface declarations

All methods for a given type must be declared in the same
package where the type is declared

This allows a package importing the type to know all the methods at compile time

But we can always extend the type in a new package through embedding:

```go
type Bigger struct {
    my.Big                // get all Big methods via promotion
}

func (b Bigger) DoIt() {  // and add one more method here
    . . .
}
```

Interfaces in practice

1. Let **consumers** define interfaces
   (what *minimal* behavior do they require?)
2. Keep interface declarations small
   ("The bigger the interface, the weaker the abstraction")
3. Compose one-method interfaces into larger interfaces (if needed)
4. Avoid coupling interfaces to particular types/implementations
5. Accept interfaces, return concrete types (if possible *)
6. Re-use standard interfaces wherever possible

* Returning `error` is a good example of an exception to this rule