Level 5–1

Gophers & Friends

Interfaces

ON TRACK with GOLANG

Calling jump() on Multiple Gophers

Static typing allows the Go compiler to ensure every element on the collection returned by getList() responds to the jump() method.

```
type gopher struct { ... }
func (g gopher) jump() string { ... }

func main() {
  gopherList := getList()
  for _, gopher := range gopherList {
    fmt.Println(gopher.jump())
  }
}

func getList() []*gopher {

}
```

All gophers respond to jump()...

...and here we grab a collection of gophers...

...so we can safely call jump() on each and every element from this collection.

*The * symbol means return value is a slice of pointers to gopher.*

Returning a Collection of struct Pointers

From the getList() function, we create two gophers, grab their pointers, and return them as part of a slice.

```go
type gopher struct { ... }
func (g gopher) jump() string { ... }

func main() {
   ...
}

func getList() []*gopher {
   phil := &gopher{name: "Phil", age: 30}
   noodles := &gopher{name: "Noodles", age: 90}

   list := []*gopher{phil, noodles}
   return list
}
```

```
$ go run main.go
```

```
Phil can jump HIGH
Noodles can still jump
```

Creates two gophers and returns a pointer for each.

Creates a new slice with the gopher pointers and returns it from the function.

Other Types Can Also jump()

There can also be other structs, like horse, with different properties but **the same method signature.**

```go
type gopher struct { ... }
func (g gopher) jump() string { ... }

type horse struct {
   name   string
   weight int
}

func (h horse) jump() string {
   if h.weight > 2500 {
      return "I'm too heavy, can't jump..."
   }
   return "I will jump, neigh!!"
}
```

Different properties...

...but exact same method signature.

Different Types Are... Different!

We **cannot** combine both Gopher and Horse structs under a single slice of type *gopher.

```go
   type gopher struct { ... }
   func (g gopher) jump() string { ... }

   type horse struct { ... }
   func (h horse) jump() string { ... }

33 func getList() []*gopher {
34   phil := &gopher{name: "Phil", age: 30}
35   noodles := &gopher{name: "Noodles", age: 90}
36   barbaro := &horse{name: "Barbaro", weight: 2000}
37
38   list := []*gopher{gopher, noodles, barbaro}
39   return list
40 }
```

A horse is NOT a gopher!

```
$ go run main.go
```

```
main.go:38: cannot use horse (type *horse) as type
            *gopher in array or slice literal
```

Common Behavior With interface

(type)

Interfaces provide a way to **specify behavior:** *"If something can do* ***this****, then it can be used* ***here****"*.

(method)

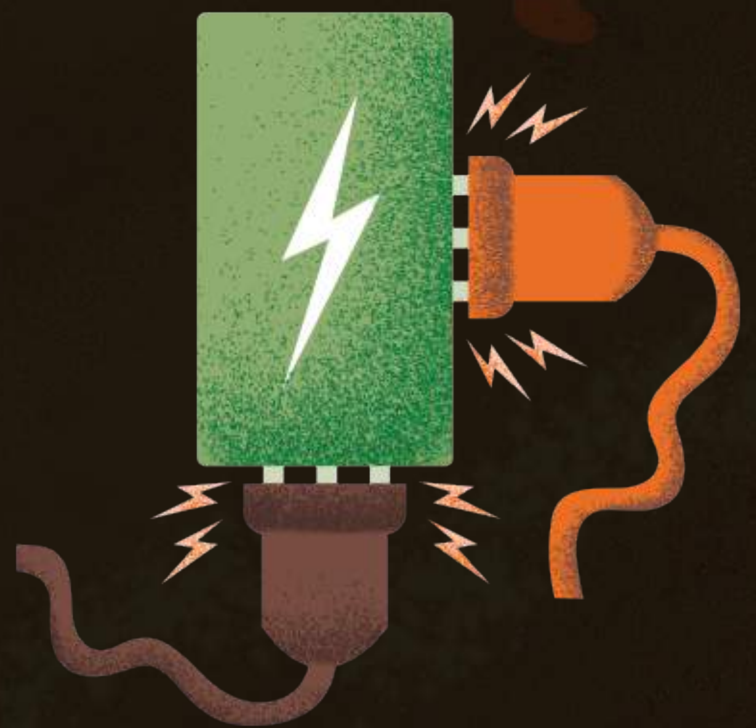
```go
...

type jumper interface {
   jump() string
}

func getList() []jumper {
   phil := &gopher{name: "Phil", age: 30}
   noodles := &gopher{name: "Noodles", age: 90}
   barbaro := &horse{name: "Barbaro", weight: 2000}

   list := []jumper{                              }
   return list
}
```

Method expected to be present in all types that implement this interface

Can be used as return type

*(The * symbol is not necessary when working with interfaces)*

All types part of this slice MUST respond to jump().



Combining Types That jump()

Types implement interfaces **implicitly**, simply by **implementing methods from the interface.**

```go
func (g gopher) jump() string { ... }
func (h horse) jump() string { ... }

type jumper interface {
  jump() string
}

func getList() []jumper {
  phil := &gopher{name: "Phil", age: 30}
  noodles := &gopher{name: "Noodles", age: 90}
  barbaro := &horse{name: "Barbaro", weight: 2000}

  list := []jumper{gopher, noodles, barbaro}
  return list
}
```

Because gopher and horse both implement** jump() **with <u>the exact same signature</u>...

***...we can add both types under the same slice of type** jumper.*

All Jumpers Can jump()

```go
type gopher struct { ... }
func (g gopher) jump() string { ... }
type horse struct { ... }
func (h horse) jump() string { ... }
type jumper interface {
  jump() string
}

func main() {
  jumperList := getList()
  for _, jumper := range jumperList {
    fmt.Println(jumper.jump())
  }
}

func getList() []jumper { ... }
```

Compiler does NOT care about naming, so these could be named something else too...

```go
list := getList()
for _, element := range list {
  fmt.Println(element.jump())
}
```

```
$ go run main.go
```

```
Phil can jump HIGH
Noodles can still jump
I will jump, Neigh!!
```

Naming Convention for interfaces

A convention for naming one-method interfaces in Go is to use the method name plus an **-er** suffix.

...plus -er suffix.

Method name...

```go
type jumper interface {
  jump() string
}
```

Examples from the Go standard library:

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

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

Visit **http://go.codeschool.com/go-io** for full documentation