Level 4–1

Adding Structure to the Data

Structs

ON TRACK with GOLANG

Young Gophers Can Jump HIGH

Not many people know this, but a gopher's ability to jump is based on their age.

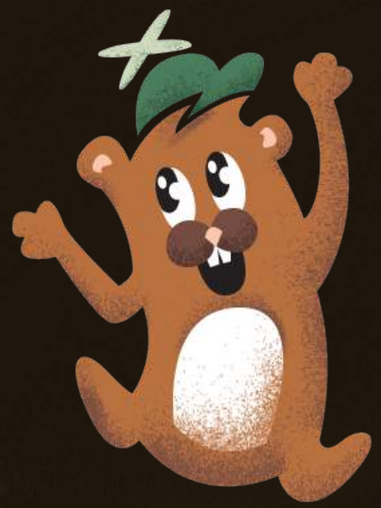
```
gopher1Name := "Phil"
gopher1Age := 30

if gopher1Age < 65 {
  highJump(gopher1Name)
} else {
  ...
}

func highJump(name string) {
  fmt.Println(name, "can jump HIGH")
}
```

Values for name and age are assigned to individual variables.

Phil can jump pretty high.

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

```
Phil can jump HIGH
```



Older Gophers Can Still Jump

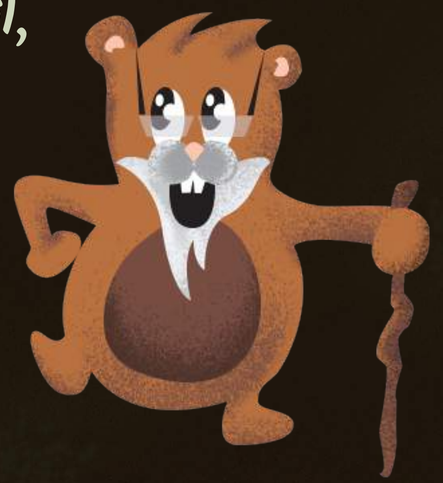
Despite the odds, the more experienced gophers can still keep up with the youngsters!

```
...
gopher2Name := "Noodles"
gopher2Age := 90

if gopher2Age < 65 {
  ...
} else {
  lowJump(gopher2Name)
}

func lowJump(name string) {
  fmt.Println(name, "can still jump")
}
```

Two new values and two new variables

Although not as young as Phil, Noodles can still jump.

```
$ go run main.go
Noodles can still jump
```



Too Much Code at Once

Things start looking confusing when we begin working with multiple gophers. This is a sign our code is **leaking logic details.**

```
gopher1Name := "Phil"
gopher1Age := 30
gopher2Name := "Noodles"
gopher2Age := 90
```

Each new gopher requires TWO independent variables...

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

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

```
if gopher1Age < 65 {
   ...
}
if gopher2Age < 65 {
   ...
}
```

...and an additional `if` statement.

Being part of the same file, logic rules are exposed to caller of this code.

```
func highJump(name string) { ... }
func lowJump(name string) { ... }
```

Hiding Details

Even in procedural languages like Go, there are ways we can **hide unnecessary implementation details** from the caller. This practice is also known as **encapsulation.**

```
gopher1 := gopher{name: "Phil", age: 30}
gopher2 := gopher{name: "Noodles", age: 90}

fmt.Println(gopher1.jump())
fmt.Println(gopher2.jump())
```

You might not know what these mean just yet, but I bet they look intuitive...

Leaking implementation details

Encapsulating implementation details

How do we get here?

Declaring a New struct

We'll declare a new struct type for a gopher. A struct is a built-in type that allows us to group properties under a single name.

The name of the struct

The underlying primitive type

```
type gopher struct {
  name string
  age  int
}
```

Properties are variables internal to the struct.

The type keyword indicates a new type is about to be declared.

Creating a struct

The most common way to allocate memory and assign values to a struct is by calling its name, followed by the initial data wrapped in curly braces.

```
type gopher struct {
  name string
  age  int
}

func main() {
  gopher1 := gopher{name: "Phil", age: 30}
  gopher2 := gopher{name: "Noodles", age: 90}
}
```

Must be placed outside the main function.

Allocates memory and assigns result to variables

Using struct for Encapsulation of Behavior

A struct contains behavior in the form of **methods**. The way we define methods on a struct is by writing a regular function and specifying **the** struct **as the explicit receiver.**

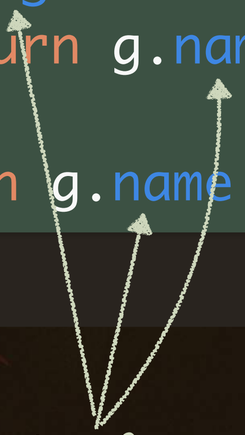
```
type gopher struct {
  name string
  age  int
}

func (g gopher) jump() string {


}

func main() {

}
```

This is how we specify an explicit receiver for this function.

Using struct for Encapsulation of Behavior

From inside the method, we can access properties from the struct via the **explicit receiver.**

```
type gopher struct {
  name string
  age  int
}

func (g gopher) jump() string {
  if g.age < 65 {
    return g.name + " can jump HIGH"
  }
  return g.name + " can still jump"
}
```



Properties from the struct

Calling Methods

We can now call jump() on all gophers and avoid exposing the “jump logic” to the caller of this method.

```
type gopher struct { ... }

func (g gopher) jump() string {
  if g.age < 65 {
    return g.name + " can jump HIGH"
  }
  return g.name + " can still jump"
}

func main() {
  gopher1 := gopher{name: "Phil", age: 30}
  fmt.Println(gopher1.jump())
  gopher2 := gopher{name: "Noodles", age: 90}
  fmt.Println(gopher2.jump())
}
```

The jump() method acts on its receiver.

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

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

The “Tell, Don’t Ask” Principle

Rather than **asking** for data and acting on it, we instead **tell** the program what to do.

Asking

Asking for age and checking...

```
if gopherAge < 65 {
   highJump(gopherName)
} else {
   lowJump(gopherName)
}

func highJump(name string) {
   ...
}
func lowJump(name string) {
   ...
}
```

...whether it has a high jump...

...or a low jump.

VS.

Telling

```
type gopher struct {
   ...
}

func (g gopher) jump() string {
   ...
}

gopher1.jump()
```

*Telling gopher **what to do. Logic is encapsulated and hidden from the caller.***