Level 4–2

Adding Structure to the Data

Working With Pointers

ON TRACK with GOLANG

Validating a Gopher's Age

A new function will set a value on a property from the struct passed as argument.

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

New property will determine whether a gopher is an adult.

A new function takes type gopher.

```
<new-function>(gopher)
```

If gopher is of age, property is set to true...

```
gopher.isAdult = true
```

...otherwise, it is set to false.

```
gopher.isAdult = false
```

This pattern of **modifying arguments passed to functions** can be found in parts of the Go standard library. The Scan() method from the database package is an example.

http://go.codeschool.com/go-db-sql

func (*Rows) Scan

```
func (rs *Rows) Scan(dest ...interface{}) error
```

Scan copies the columns in the current row into the values pointed at by dest. The number of values in dest must be the same as the number of columns in Rows.

New Property Defaults to Zero Value

The isAdult property from all new gophers defaults to **false** — the zero value for type bool, remember?

```go
type gopher struct {
  name string
  age  int
  isAdult bool
}

func main() {
  phil := gopher{name: "Phil", age: 35}
  fmt.Println(phil)
}
```

```
$ go run main.go
{Phil 35 false}
```

No value assigned to isAdult, so it defaults to false.

Writing a Validation Function

The new function takes one argument and writes to the isAdult property of this argument. The function does **NOT** return anything.

```
type gopher struct {
  ...
}

func main() {
  phil := gopher{name: "Phil", age: 35}
  validateAge(phil)

}

func validateAge(g  ???? ) {
  g.isAdult = g.age >= 21
}
```

Passing type gopher as argument

Must accept a compatible type.

Passing structs **by Value**

Passing a struct as argument **creates a copy** of all the values assigned to its properties.

```go
type gopher struct {
  name string
  age  int
  isAdult bool
}

func main() {
  phil := gopher{name: "Phil", age: 35}
  validateAge(phil)
  fmt.Println(phil)
}

func validateAge(g gopher) {
  g.isAdult = g.age >= 21
}
```

Creates a COPY of the original struct data...

...and the COPY of the data is received as argument.

Assigns true to the COPY of the data — not the original data!

```
$ go run main.go
{Phil 35 false}
```

Original value from struct is NOT changed.

Values and References in Music Playlists

Thinking about how playlists work can help us understand the difference between values and references.

Original songs from each artist's album

Albums

Are You Gonna Go My Way

1 - Song **a**

2 - Song **b**

3 - Song **c**

Milo Goes to College

1 - Song **d**

2 - Song **e**

3 - Song **f**

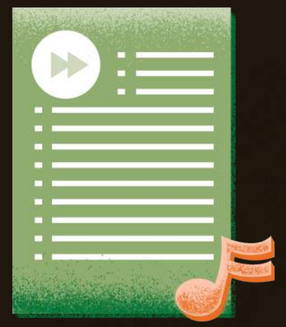
Playlist *(by value)*

Music for Programming

1 -

2 -

3 -



Favorite songs handpicked by us

Creating a Playlist With Values

We can implement a music playlist by **creating copies** of existing songs and storing those copies under each playlist.

Albums

Are You Gonna Go My Way

1 - Song **a**

2 - Song **b**

3 - Song **c**

Milo Goes to College

1 - Song **d**

2 - Song **e**

3 - Song **f**

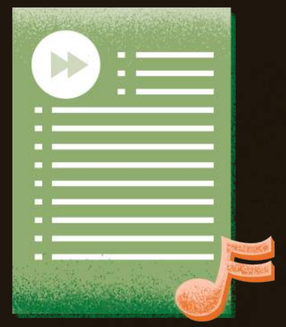
Playlist *(by value)*

Music for Programming

1 - Song **a**

2 - Song **c**

3 - Song **f**



Playlist contains copies of the original songs.

Creating a Playlist With References

A more efficient way to implement a playlist is by **storing references** to original songs. This avoids creating multiple copies of the same songs for each new playlist.

Albums

Are You Gonna Go My Way

1 - Song **a**

2 - Song **b**

3 - Song **c**

Milo Goes to College

1 - Song **d**

2 - Song **e**

3 - Song **f**

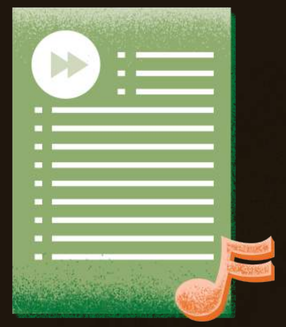
Playlist *(by reference)*

Music for Programming

1 -

2 -

3 -



Playlist slots point back to original songs.
No copies are created! (memory efficient)

Pass by Value

A) Passing Values *(default behavior)*

```
language := "Go"
```

1. Primitive value is assigned to new variable and stored in **new memory address** **0x10444444**.

```
favoriteLanguage := language
```

2. A **new memory address** **0x10555555** **is allocated** for the new variable that receives **a copy of the original data.**

Two different memory addresses are used to store exact copies of the data.

Pass by Reference

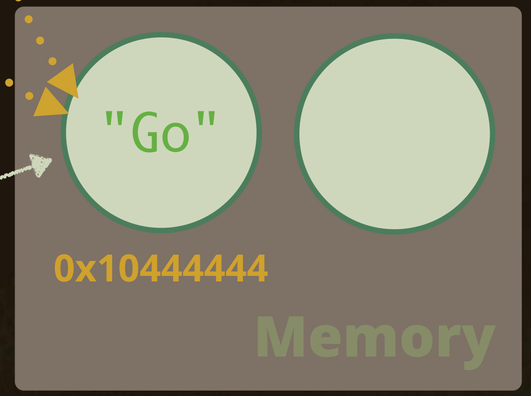
B) Passing References

1. Primitive value is assigned to new variable and stored in new memory address **0x10444444**.
2. Using the & operator, a **reference** to the existing memory address is assigned to the new variable.

```
language := "Go"
```

Returns a pointer

```
favoriteLanguage := &language
```



A single memory address is used. (memory efficient)

Passing structs **by Reference**

In order to assign a struct reference to a new variable, we use the & operator to return a **pointer**.

```go
type gopher struct {
  name string
  age  int
  isAdult bool
}

func main() {
  phil := &gopher{name: "Phil", age: 30}
  validateAge(phil)
  fmt.Println(phil)
}

func validateAge(          ) {
  g.isAdult = g.age >= 21
}
```

The & operator returns a pointer to this new struct.

Passes a reference to the original struct — NOT a copy of the values

Values and References Are Not the Same

Accepting references as function arguments requires a different syntax.

```go
type gopher struct {
  name string
  age  int
  isAdult bool
}

func main() {
  phil := &gopher{name: "Phil", age: 30}
  validateAge(phil)
  fmt.Println(phil)
}

func validateAge(g gopher) {
  g.isAdult = g.age >= 21
}
```

Wrong type!

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

```
main.go:15: cannot use phil (type *gopher) as
        type gopher in argument to validateAge
```

Reading struct **References**

We use the * operator to access the value that the pointer points to (a.k.a., “dereferencing a pointer variable”).

```go
type gopher struct {
  name string
  age  int
  isAdult bool
}

func main() {
  phil := &gopher{name: "Phil", age: 30}
  validateAge(phil)
  fmt.Println(phil)
}

func validateAge(g *gopher) {
  g.isAdult = g.age >= 21
}
```

***The * operator indicates a pointer to the type** gopher.*

```
$ go run main.go
&{Phil 30 true}
```

Original value from** struct **is changed

***Reference to underlying** struct*

Level 4–2

Adding Structure to the Data

Working With Pointers

ON TRACK with GOLANG