Scopes and Slices

Tom Arrell

December 16

Recap...

Last week we covered control structures

- **If** statement

This week

- Scopes, *global* and *local*
- Slices and arrays

Recap...

Last week we covered control structures

- **If** statement
- **For** loop

This week

- Scopes, *global* and *local*
- Slices and arrays

Recap...

Last week we covered control structures

- **If** statement
- **For** loop
- How to combine the two to make our programs more useful

This week

- Scopes, *global* and *local*
- Slices and arrays

Scope

What is it?

Scope defines the area of visibility within the program of variables and functions.

Example

```
a := 1
{
  a := 2
  fmt.Println(a)
}
fmt.Println(a, b)
```

Hint: This doesn't compile

Explanation

This is an example of **local** scope. Local scope works in only one direction.

i.e. You can only access local scope that was defined at a *higher* level than what you're currently using.

Types

There are two primary scopes in Golang.

- Global: variables defined here are accessible anywhere within the package that they were defined

Types

There are two primary scopes in Golang.

- Global: variables defined here are accessible anywhere within the package that they were defined
- Local: variables defined here and only accessible within their own *block* and below.

Global Scope

Golang makes it rather easy to spot everything in the global scope.

These things are defined at the *root* level, i.e. have nothing surrounding them, and therefore have no indentation.

```
package main

var hello = "Hello World"

func main() {
   ...
}
```

Local scope

Local scope is a little more nuanced. There is a heirarchy.

There is a concept of *"nested"* scopes. Therefore, we say that inner scopes have access to things only in their current and outer scopes.

They are able to read from and write to the variables that they have access to.

Shadowing

Shadowing is the term for when you redeclare a variable in a lower scope that is *already* declared in an outer scope.

This is sometimes useful for prevening lower scopes from having access to said variable.

Ararys and Slices

Most languages have a concept of *arrays*. These are essentially just a list of related things.

Ararys and Slices

In Golang, we have:

- Arrays
- Slices

Note: Slices use arrays underneath.

So what's the difference?

- Arrays have a fixed size

So what's the difference?

- Arrays have a fixed size
- Slices can grow

Example

```
...

func slices() {
    array := [6]int{1, 2, 3, 4, 5, 6}
    fmt.Println(array)

    slice := []int{1, 2, 3, 4, 5, 6, 7}
    fmt.Println(slice)
}
```

Slices are *most* common in practice, as they are less bug prone. We will continue looking are them.

Creating a slice

To create a slice in Go, you use one of the following syntaxes:

```
var := []type{}
var := make([]type, cap, len)
```

Both of which will create an empty slice, ready to put data into.

Range syntax

Go allows you to iterate over certain types of *collections*, slices being one of these.

```
for i, v := range mySlice {
  fmt.Println(i, v)
}
```

This is equivalent to the traditional for loop:

```
for i := 0; i < len(mySlice); i++ {
  fmt.Println(i, mySlice[i])
}
```

Note: the index operator `mySlice[i]` which is essentially saying `mySlice.Get(i)` where `Get(i int)` retrieves the value at index `i`.

Append

Golang has a useful built-in function for adding things to slices.

This is called the **Append** function.

You can use it like so:

```
mySlice := []int{1, 2, 3, 4}
mySlice = append(mySlice, 5, 6, 7)
```

Typed slices

Your slice doesn't just need to store integers. You can store any valid *Type*.

Slice example

```
type friend struct {
  Name    string
  Planet  string
  Pets    int
  IsClose bool
}

friends := []friend{
  friend{"Rey", "Tatooine", 1, true},
  friend{"Finn", "Jakku", 0, false},
  friend{"Han", "Corellia", 1, true},
}
```

This is an example of a custom *struct* type called “friend”, which we are constructing 3 of inside a slice.

Challenge time

Given an array of integers, return a new array with all the values doubled.

```
...

var input = []int{1, 2, 3, 4, 5}

func double(numbers []int) []int {
  // TODO
}

func main() {
  out := double(input)
  fmt.Println(out)
}
```

Challenge #2

We run a pizza shop, and we have a program which automatically calculates the total price of an order.

Given a (Go) slice of pizzas in an order, return the total price of the order.

- Pepperoni = $6.00
- Mozzarella = $5.00
- Vege = $5.50

```
...

func orderPrice(pizzas []string) int {
  // TODO
}

func main() {
  total := orderPrice([]string{
    "pepperoni",
    "mozzarella",
    "vege",
    "vege",
  })

  fmt.Println(total)
}
```

lesson 3, fin

If you had any trouble, now is the time to ask for help!

Questions?