Maps and Methods

Tom Arrell

January 20

Last week...

We went through some practical challenges. The answer to each is available on Github.

Did anyone go away and complete the fifth challenge?

This week

- Maps
  - introduction
  - creation
  - getters
  - setters
  - challenge
- Methods
  - introduction
  - challenge

What is a map?

A map is a *data structure* which contains **key-value pairs**.

e.g.

```
key: value
key: value
...
```

What is a map?

In almost all implementations of a map, you have at least two things that you can do with them.

1. Set a value with a specific key
2. Get a value with a specific key

All keys must be unique. i.e. A single key can only map to a single value.

A sufficient analogy is a phone book. Looking someone up by their name will give you their phone number.

Creating a Map

In Go, to create a map, you need to specify both the *key* and the *value* types.

Once we know that, we can then use the builtin function `make` to construct our map.

e.g.

```
myMap := make(map[string]int)
```

This will construct a map where all the keys are **strings**, and values are **integers**.

Map: Setter

Once you've got an initialised map, you are able to set and get values within it.

In order to set a given key to a specific value (given they are both of the correct type for the particular map), you can do the following.

```
myMap := make(map[string]int)
myMap["hello"] = 2
```

This will set the key `"hello"` to the value 2 within the map.

Map: Getter

Once you have values inside your map that you would like to retrieve, you can use a similar syntax.

e.g.

```
// Creation
myMap := make(map[string]int)
// Setting
myMap["hello"] = 2
// Getting & printing
val := myMap["hello"]
fmt.Println("The value set was %d", val)
```

Map Challenge

Given a paragraph of text, write a function which takes the text, and returns a map where the keys are each word, and the value is the count of the occurences of that word in the text.

```
func countWords(text string) map[string]int {
  // your code
}

func main() {
  text := "The quick brown fox might want to jump over..."

  fmt.Println(countWords(text))
}
```

What is a Method?

A method is syntactic sugar for a function, where the first argument is an instance of a type.

e.g.

```
type Person struct { ... }

func (p *Person) Greet() {
  fmt.Println("Hello, %s", p.Name)
}
```

We can see in the above that the greet function has access to the person, and can use the fields on the struct within the method.

What is a Method?

To call a method, we use the `.` operator on an instance of the type.

e.g.

...

```
func main() {
  me := Person{ Name: "Skywalker" }
  me.Greet() // prints: Hello, Skywalker
}
```

What is a Method?

So why is this *syntactic sugar*? Well, what we saw on the previous slide is equivalent to:

```
func Greet(p *Person) {
  fmt.Println("Hello, %s", p.Name)
}
```

Using a method means you don't have to pass in the instance to a function through the arguments.

Defining related sets of methods becomes important when we get to **interfaces**.

Methods on all types

Methods are very versatile, you can define them on any custom type.

e.g.

```
type Age int

func (a *Age) String() string {
  return fmt.Sprintf("%d years", a)
}
```

This allows you to define custom types with special properties, such as printing themselves in a unique manner.

We'll now take a quick look at `time.Time`.

Method challenge

Create a struct named `Circle` with a single `int` field called `Radius`. Write a method on that struct called `Area` which returns the area of the circle instance.

```
type Circle struct {
  Radius int
}

func main() {
  c := Circle{3}
  fmt.Println(c.Area()) // 28.274...
}
```

lesson 6, fin

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

Questions?