The for range Construct

This lesson explains how to use the for range construct to access key-value pairs in a map.

Explanation

The following construct can also be applied to maps:

```
for key, value := range map1 {
  ...
}
```

The `key` is the key of the map, and the `value` is the value for the `key`. They are local variables only known in the body of the for statement. If you are only interested in the values, use the form:

```
for _, value := range map1 {
  fmt.Printf("Value is: %d\n", value)
}
```

To get only the keys, you can use:

```
for key := range map1 {
  fmt.Printf("Key is: %d\n", key)
}
```

The order in which elements are visited when iterating over a map using a for range statement is *unpredictable*, even if the same loop is run multiple times with the same map. The first element in a map iteration is chosen at *random*. This behavior allows the map implementation to ensure better map balancing. Your code should not assume that the elements are visited in any particular order.

Implementation

Here is an example of a program that uses a for loop on a map.

```
package main
import "fmt"

func main() {
  map1 := make(map[int]float32)
  map1[1] = 1.0
  map1[2] = 2.0
  map1[3] = 3.0
  map1[4] = 4.0
  // for range
  for key, value := range map1 {
    fmt.Printf("key is: %d - value is: %f\n", key, value)
  }
}
```

For range on a Map

In the code above, in `main`, at line **5**, we make a map `map1`. The declaration of `map1` shows that its keys will be of *int* type and values associated with its keys will be of *float32* type. From **line 6** to line **9**, we are making *key-value* pairs (each pair line by line) for `map1`. At **line 6**, we create a key `1` and give the value **1.0** to it. At **line** 7, we create a key `2` and give the value **2.0** to it. At **line 8**, we create a key `3` and give the value **3.0** to it. At **line 9**, we create a key `4` and give the value **4.0** to it. Now, we have a for loop at **line 11**. It's reading all the key-value pairs from `map1` and printing a key-value pair in each iteration. The order of output may change every time you run the program.

We see that a map is not *key-ordered*, neither is it sorted on the values. It is even safe to delete pairs from a map while iterating over the map like:

```
package main
import "fmt"

func main() {
    map1 := make(map[int]float32)
    map1[1] = 1.0
    map1[2] = 2.0

    map1[3] = 3.0
    map1[4] = 4.0

    for key := range map1 {
      if key > 3 {
        delete(map1, key)   // deleting keys greater than and equal to 4
      }
    }
    // printing the modified map
    for key, value := range map1 {
      fmt.Printf("key is: %d - value is: %f\n", key, value)
    }
  }
```

In the code above, in `main` at line **5**, we made a map `map1`. The declaration of `map1` shows that its keys will be of *int* type and values associated with its keys will be of *float32* type. From **line 6** to line **9**, we are making *key-value* pairs (each pair line by line) for `map1`. At **line 6**, we create a key `1` and give the value **1.0** to it. At **line** 7, we created a key `2` and gave the value **2.0** to it. At **line 8**, we create a key `3` and give the value **3.0** to it. At **line 9**, we create a key `4` and give the value **4.0** to it.

Now we have a for loop at **line 11**. It's reading all the key-value pairs from `map1` and deleting the key-value pairs for which the `key` is greater than **3** using the `delete` function.

Then again, we have a for loop at **line 17** that is printing modified `map1`.

Now that you are familiar with accessing the keys and values from the maps using for loop, in the next lesson, you have to write a program to solve a problem.