

# Maps in Go

This lesson gives an introduction to maps, using the map literals and mutating maps in Go

## WE'LL COVER THE FOLLOWING ^

- Introduction
- Mutating maps
- Resources

## Introduction #

Maps are somewhat similar to what other languages call “dictionaries” or “hashes”.

A map *maps* keys to values. Here we are mapping string keys (actor names) to an integer value (age).

### Environment Variables ^

Key:	Value:
GOPATH	/go

```
package main

import "fmt"

func main() {
    celebs := map[string]int{ //mapping strings to integers
        "Nicolas Cage":      50,
        "Selena Gomez":      21,
        "Jude Law":           41,
        "Scarlett Johansson": 29,
    }

    fmt.Printf("%#v", celebs)
}
```



When not using map literals like above, maps must be created with `make` (not `new`) before use. The `nil` map is empty and cannot be assigned to.

Assignments follow the Go convention and can be observed in the example below.

Environment Variables

Key:	Value:
GOPATH	/go

```
package main

import "fmt"

type Vertex struct {
    Lat, Long float64
}

var m map[string]Vertex

func main() {
    m = make(map[string]Vertex)
    m["Bell Labs"] = Vertex{40.68433, -74.39967} //assignment
    fmt.Println(m["Bell Labs"])
}
```

In the example above the `map m` takes `string` as input and maps it to `vertex` which is a `struct` containing two variables of type `float64`. Hence, the string “Bell” gets mapped to the value “40.68433” and “Labs” gets mapped to “-74.39967”.

When using map literals, if the top-level type is just a type name, you can omit it from the elements of the literal.

Environment Variables

Key:	Value:
GOPATH	/go

```
package main
```

```
import "fmt"

type Vertex struct {
    Lat, Long float64
}

var m = map[string]Vertex{
    "Bell Labs": {40.68433, -74.39967},
    // same as "Bell Labs": Vertex{40.68433, -74.39967}
    "Google": {37.42202, -122.08408},
}

func main() {
    fmt.Println(m)
}
```



## Mutating maps #

Insert or update an element in map m:

```
m[key] = elem
```



Retrieve an element:

```
elem = m[key]
```



Delete an element:

```
delete(m, key)
```



Test that a key is present with a two-value assignment:

```
elem, ok = m[key]
```



Let's take a look at an example now:

### Environment Variables



Key:

Value:

GOPATH

/go



```
package main

import "fmt"

type Vertex struct {
    Lat, Long float64
}

var m = map[string]Vertex{
    "Bell Labs": {40.68433, -74.39967},
    // same as "Bell Labs": Vertex{40.68433, -74.39967}
    "Google": {37.42202, -122.08408},
}

func main() {
    m["Splice"] = Vertex{34.05641, -118.48175} //inserting a new (key,value) here
    fmt.Println(m["Splice"])
    delete(m, "Splice") //deleting the element
    fmt.Printf("%v\n", m)
    name, ok := m["Splice"] //checks to see if element is present
    fmt.Printf("key 'Splice' is present?: %t - value: %v\n", ok, name)
    name, ok = m["Google"]
    fmt.Printf("key 'Google' is present?: %t - value: %v\n", ok, name)
}
```



If **key** is in **m**, **ok** is true. If not, **ok** is false and **elem** is the zero value for the map's element type. Similarly, when reading from a map if the key is not present the result is the zero value for the map's element type.

## Resources #

- [Go team blog post on maps](#)
- [Effective Go - maps](#)

This marks the end of this chapter. Read on to the next one to learn more interesting concepts in Go.