

# The anatomy of maps in Go

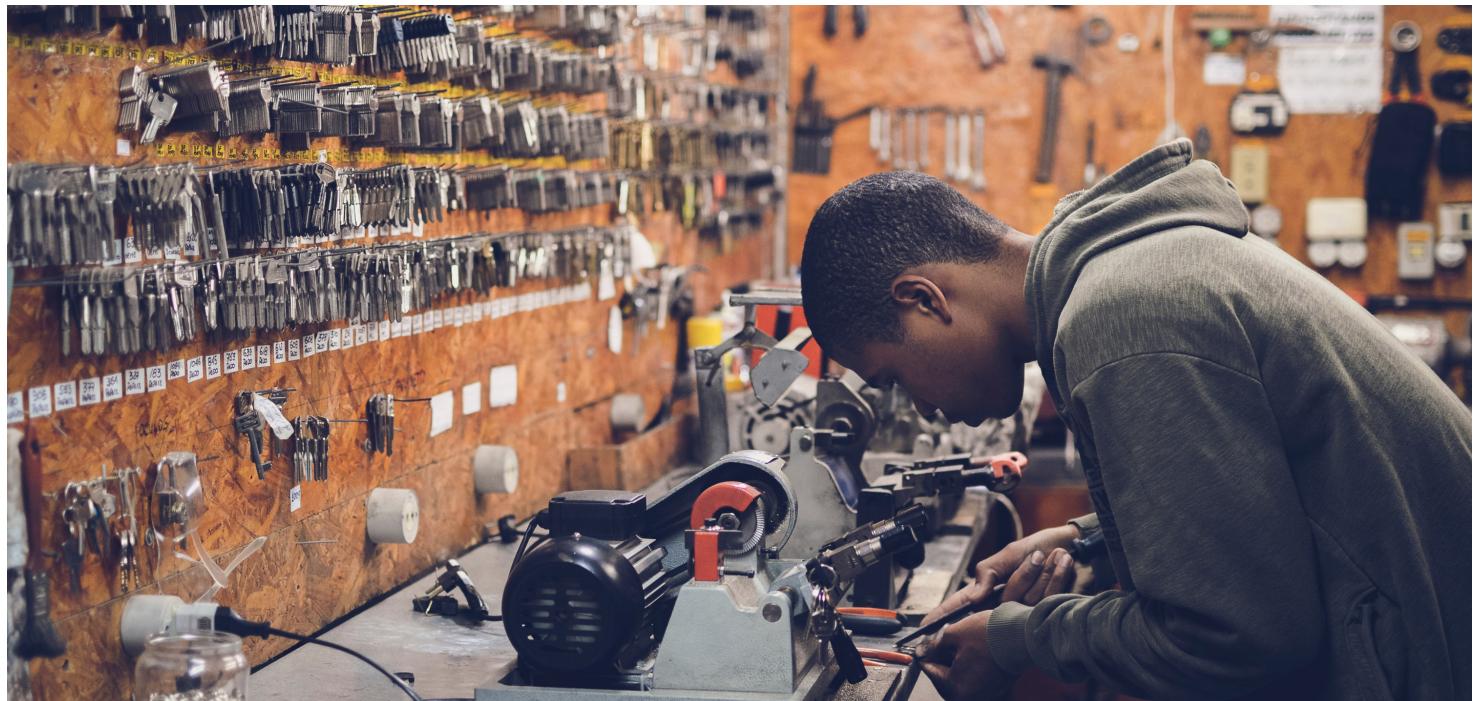
A map is composite data type that can hold data represented by key:value pairs.



Uday Hiwarale

[Follow](#)

Sep 27, 2018 · 6 min read



A map is like an apartment complex with many rooms. Every room has a key and key is necessary to find out what's inside that room. Key can be made of any fixed material (data type) and a room can contain any fixed material (data type).

## ➡ What is a map?

A map is like an `array` except, instead of an integer `index`, you can have `string` or any other data types as `key`. An illustration of a map looks like

```
{  
    stringKey: intValue,  
    stringKey: intValue  
    ...  
}
```

The syntax to define a map is

```
var myMap map[keyType]valueType
```

Where `keyType` is the data type of **map keys** while `valueType` is the data type of **map values**. A `map` is a **composite data type** because it is composed of primitive data types (*see variables lesson for primitive data types*).

Let's declare a simple map.

The screenshot shows the GoLand IDE interface. On the left is the Explorer sidebar with 'OPEN EDITORS' containing 'maps.go src' and 'MAIN' containing 'bin', 'pkg', and 'src'. The 'src' folder is expanded, showing 'maps.go'. The main area is a code editor titled 'maps.go — main' with the following content:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     var m map[string]int
7
8     fmt.Println(m)
9     fmt.Println("m == nil", m == nil)
10 }
```

Below the code editor is a terminal window titled 'TERMINAL' with the command 'go run src/maps.go' and its output:

```
main > go run src/maps.go
map[]
m == nil true
main > 
```

The status bar at the bottom shows 'Ln 7, Col 5' and other GoLand specific icons.

<https://play.golang.org/p/Rv6dFDthsoZ>

In the above program, we have declared a map `m` which is empty, because `zero-value` of a map is `nil`. But the thing about `nil` map is, we can't add values to it because like slices, `map` cannot hold any data, rather they reference internal data structure that holds the data. So, in case of `nil` map, the internal data structure is missing and assigning any data to it will cause runtime panic `panic: assignment to entry in nil map`. You can use `nil` map as a variable to store another non `nil` map.

## Creating an empty map

An `empty map` is like an empty `slice` with internal data structure defined so that we can use it to store some data. Like `slice`, we can use `make` function to create an empty `map`.

```
m := make(map[keyType]valueType)
```

Let's create a simple map `age` in which we can save age of people.

```
maps.go — main
EXPLORER
OPEN EDITORS
  maps.go src
MAIN
  bin
  pkg
  src
    maps.go
package main
import "fmt"
func main() {
    age := make(map[string]int)
    age["mina"] = 28
    age["john"] = 32
    age["mike"] = 55
    fmt.Println("age of john", age["john"])
}
TERMINAL
main > go run src/maps.go
age of john 32
main >
```

Ln 11, Col 2 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing

[https://play.golang.org/p/mrQ\\_Hw0lZI0](https://play.golang.org/p/mrQ_Hw0lZI0)

In the above program, we have created an empty map `age` which holds `int` data and referenced by `string` keys. You can access or assign a value of a map using key like `map[key]`.

Using that information, we assigned some age data of `mina`, `john` and `mike`. You can add as many values as you want, as `map` like `slice` can hold a variable number of elements.

## ☛ Initializing a map

Instead of creating `empty` map and assigning new data, we can create a map with some initial data, like `array` and `slice`.

```
maps.go — main
1 package main
2
3 import "fmt"
4
5 func main() {
6     age := map[string]int{
7         "mina": 28,
8         "john": 32,
9         "mike": 55,
10    }
11
12    fmt.Println(age)
13}
14
```

TERMINAL

```
main > go run src/maps.go
map[john:32 mike:55 mina:28]
main >
```

<https://play.golang.org/p/49hr11mmkqx>

## Accessing map data

In case `array` or `slice`, when you are trying to access out of index element (*when the index does not exist*), go will throw an error. But not in case of `map`. When you are trying to access the value by the `key` which is not present in the map, go will not throw an error, instead, it will return `zero` value of `valueType`.

```
maps.go — main
1 package main
2
3 import "fmt"
4
5 func main() {
6     age := map[string]int{
7         "mina": 28,
8         "john": 32,
9         "mike": 55,
10    }
11
12    fmt.Println(age["mina"])
13    fmt.Println(age["jessy"])
14}
```

```
13
14
TERMINAL
1: bash
main > go run src/maps.go
28
0
main >
```

Ln 11, Col 14 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing

<https://play.golang.org/p/fq6i39u8AeE>

28 is correct because that's the age of mina but since jessy is not in the map, go will return 0 as it's a zero-value of data type int.

So, to check if a key exists in the map or not, go provide another syntax which returns 2 values.

```
value, ok := m[key]
```

Let's see above example in a new syntax

```
maps.go — main
EXPLORER
OPEN EDITORS
maps.go src
MAIN
bin
pkg
src
maps.go
maps.go
package main
import "fmt"
func main() {
    age := map[string]int{
        "mina": 28,
        "john": 32,
        "mike": 55,
    }
    minaAge, minaOk := age["mina"]
    jessyAge, jessyOk := age["jessy"]

    fmt.Println(minaAge, minaOk)
    fmt.Println(jessyAge, jessyOk)
}

TERMINAL
1: bash
main > go run src/maps.go
28 true
0 false
main >
```

The screenshot shows the RunGo web-based Go development environment. At the top, there's a toolbar with icons for file operations and a status bar showing 'Ln 13, Col 5' and other settings. Below the toolbar is a URL bar with the address <https://play.golang.org/p/YYYT2qdiGue>. The main area contains a code editor with the following Go code:

```

maps.go — main
maps.go x
1 package main
2
3 import "fmt"
4
5 func main() {
6     age := map[string]int{
7         "mina": 28,
8         "john": 32,
9         "mike": 55,
10    }
11    fmt.Println("len(age) =", len(age))
12 }

```

Below the code editor is a terminal window labeled 'TERMINAL' with the command 'main > []'. The bottom status bar shows 'Ln 11, Col 40' and other settings.

So, we are getting extra information on whether a key exists or not. If the key exists, the second parameter will be `true` else it will be `false`.

## Length of map

We can find out how many elements contained in a `map` using `len` function, which we saw in `array` and `slice`.

The screenshot shows the RunGo web-based Go development environment. At the top, there's a toolbar with icons for file operations and a status bar showing 'Ln 11, Col 40' and other settings. Below the toolbar is a URL bar with the address <https://play.golang.org/p/94lOJeBXfq7>. The main area contains a code editor with the same Go code as before. The terminal window below shows the output of the program: 'len(age) = 3'. The bottom status bar shows 'Ln 11, Col 40' and other settings.

*There is nothing like capacity in `map` because go completely takes control of internal data structure of `map`. Hence don't try to use `cap` function on `map`.*

## Delete map element

Unlike `slice` where you need to use a **hack** to delete an element, go provide easier function `delete` to delete `map` element. Syntax of a delete function is as following

```
func delete(m map[Type]Type1, key Type)
```

`delete` function demands the first argument to be a `map` and second argument to be a `key` value.

```
maps.go — main
EXPLORER
OPEN EDITORS
  maps.go src
MAIN
  bin
  pkg
  src
    maps.go
maps.go x
1 package main
2
3 import "fmt"
4
5 func main() {
6     age := map[string]int{
7         "mina": 28,
8         "john": 32,
9         "mike": 55,
10    }
11
12    delete(age, "john")
13    delete(age, "jessy")
14
15    fmt.Println(age)
16 }
17

TERMINAL
1: bash
main > go run src/maps.go
map[mike:55 mina:28]
main > [REDACTED]

Ln 15, Col 1 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing ☺ 🔔
```

[https://play.golang.org/p/i\\_M0x76jleP](https://play.golang.org/p/i_M0x76jleP)

If the key does not exist in the map, like `jessy` in above example, go will not throw an error while executing `delete` function.

## maps comparison

Like `slice`, a map can be only compared with `nil`. If you are thinking to iterate over a map and match each element, you are in grave trouble. But if you are in dire need to compare two maps, use `reflect` package's `DeepEqual` function (<https://golang.org/pkg/reflect/>).

## map iteration

Since there are no index values in `map`, you can't use simple `for` loop with incrementing `index` value until it hits the end. You need to use `for range` to do it.

```
maps.go — main
1 package main
2
3 import "fmt"
4
5 func main() {
6     age := map[string]int{
7         "mina": 28,
8         "john": 32,
9         "mike": 55,
10    }
11
12    for key, value := range age {
13        fmt.Println(key, "=>", value)
14    }
15}
16
```

TERMINAL

```
main > go run src/maps.go
mina => 28
john => 32
mike => 55
main > █
```

<https://play.golang.org/p/MbIYk17zu6C>

`range` in `for` loop will return `key` and `value` of map element. You can also use `_` (blank identifier) to ignore either `key` or `value` in case you don't need it, just like `array` and `slice`.

*Order of retrieval of elements in map is random when used in for iteration. Hence there is no guarantee that each time, they will be in order. That explains why we can't compare two maps.*

## map with other data types

It's not necessary that only string types should be the keys of a map. All **comparable types** such as boolean, integer, float, complex, string etc. can also be keys. This should be very obvious, but `boolean` just blows my mind because `boolean` can only represent 2 values, either `true` or `false`. Let's see what happens where we can use it.

The screenshot shows the RunGo IDE interface. The left sidebar has icons for Explorer, Open Editors, Main, and Terminal. The 'OPEN EDITORS' section shows 'maps.go src'. The 'MAIN' section shows 'bin', 'pkg', and 'src'. Under 'src', 'maps.go' is selected. The main editor window displays the following Go code:

```

1 package main
2
3 import "fmt"
4
5 func main() {
6     age := map[bool]string{
7         true:  "YES",
8         false: "NO",
9     }
10
11    for key, value := range age {
12        fmt.Println(key, ">=>", value)
13    }
14}
15

```

The terminal window below shows the output of running the program:

```

main > go run src/maps.go
false => NO
true => YES
main >

```

The status bar at the bottom indicates: Ln 11, Col 34 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing.

<https://play.golang.org/p/Nbelj54VdES>

I guess we found ourselves a use case for `boolean` key values. But what if we add duplicate keys.

The screenshot shows the RunGo IDE interface. The left sidebar has icons for Explorer, Open Editors, Main, and Terminal. The 'OPEN EDITORS' section shows 'maps.go src 1'. The 'MAIN' section shows 'bin', 'pkg', and 'src'. Under 'src', 'maps.go' is selected. The main editor window displays the following Go code:

```

1 package main
2
3 import "fmt"
4
5 func main() {
6     age := map[bool]string{
7         true:  "YES",
8         false: "NO",
9         true:  "YEAH",
10    }
11
12    for key, value := range age {
13        fmt.Println(key, ">=>", value)
14    }
15}
16

```

The terminal window below shows the output of running the program, which includes an error message:

```

main > go run src/maps.go
# command-line-arguments
src/maps.go:9:7: duplicate key true in map literal
main >

```

The screenshot shows the RunGo IDE interface. At the top, there's a toolbar with icons for settings, file operations, and analysis tools. The status bar at the bottom indicates 'Ln 10, Col 6' and other settings like 'Tab Size: 4', 'UTF-8', 'LF', 'Go', and 'Analysis Tools Missing'. Below the toolbar is a URL bar with the address <https://play.golang.org/p/sOC6TpétrK>.

```
maps.go — main
maps.go x
package main
import "fmt"
func main() {
    var ages map[string]int
    age := map[string]int{
        "mina": 28,
        "john": 32,
        "mike": 55,
    }
    ages = age
    delete(ages, "john")
    fmt.Println("age", age)
    fmt.Println("ages", ages)
}
TERMINAL
1: bash
main > go run src/maps.go
age map[mina:28 mike:55]
ages map[mike:55 mina:28]
main >
```

This proves that we are not allowed to add duplicate keys in a map.

## ☞ Maps are referenced type

Like `slice`, map references an internal data structure. When you copy a map to a new map, the internal data structure is not copied, just referenced.

The screenshot shows the RunGo IDE interface. At the top, there's a toolbar with icons for settings, file operations, and analysis tools. The status bar at the bottom indicates 'Ln 18, Col 1' and other settings like 'Tab Size: 4', 'UTF-8', 'LF', 'Go', and 'Analysis Tools Missing'. Below the toolbar is a URL bar with the address <https://play.golang.org/p/n-U1AF4jy8M>.

```
maps.go — main
maps.go x
package main
import "fmt"
func main() {
    var age map[string]int
    age := map[string]int{
        "mina": 28,
        "john": 32,
        "mike": 55,
    }
    ages := age
    delete(ages, "john")
    fmt.Println("age", age)
    fmt.Println("ages", ages)
}
TERMINAL
1: bash
main > go run src/maps.go
age map[mina:28 mike:55]
ages map[mike:55 mina:28]
main >
```

As expected, `ages` map now has two elements because we deleted one element. Not only that, but we got same change in `age` map as well. This proves that, **don't mess with maps**.

To copy a map, you need to use `for` loop.

```

maps.go — main
EXPLORER
OPEN EDITORS
maps.go src
MAIN
bin
pkg
src
maps.go
maps.go
package main
import "fmt"
func main() {
    ages := make(map[string]int)
    age := map[string]int{
        "mina": 28,
        "john": 32,
        "mike": 55,
    }
    for key, value := range age {
        ages[key] = value
    }
    delete(ages, "john")
    fmt.Println("age", age)
    fmt.Println("ages", ages)
}
TERMINAL
1: bash
main > go run src/maps.go
age map[mina:28 john:32 mike:55]
ages map[mike:55 mina:28]
main >

```

<https://play.golang.org/p/uqwruAyVym4>

In the above case, we haven't copied map but used map **keys** and **values** to store in a different map which implements its own underlying data structure.

*Since `map` references internal data structure, `map` passed as function parameter share same internal data structure just like `slice`. Hence, make sure to follow same guidelines as explained in `slices` lesson.*