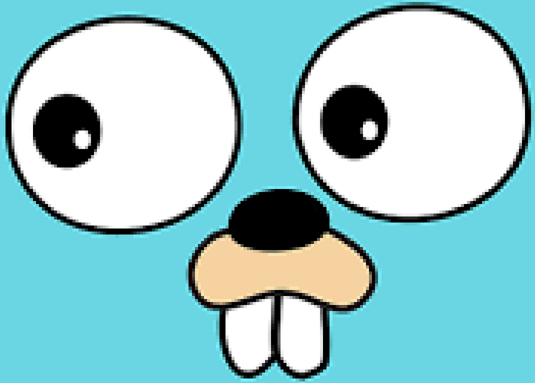


Phase 1: Introduction to Golang (Sessions 1-8)

Objective: Build a strong foundation in Golang basics.

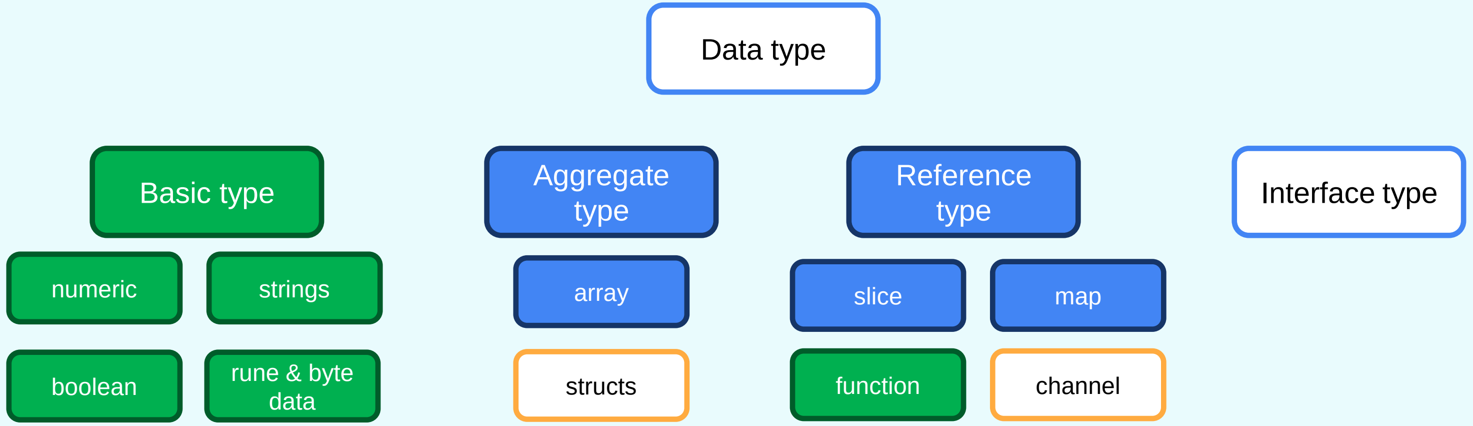
Session 6: Arrays, Slices, and Maps

Discussion Points

- Array basics
- Slices: creation, manipulation, and internal mechanics
- Maps: creating, accessing, and iterating

Array basics

What is next with Data Types?



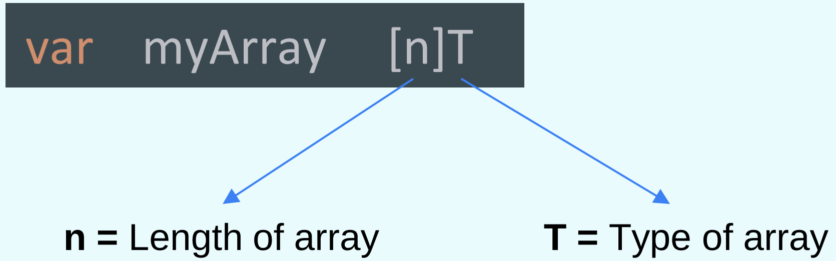
Array basics

What is Array?

Arrays are data structures that represent an ordered sequence of elements of the same type.

Since Go is a statically typed language, mixing different values belonging to different data types in an array is not allowed.

Array sizes are static, they are set when the array is defined and cannot be changed subsequently.



```go
var   myArray   [n]T
```

n = Length of array

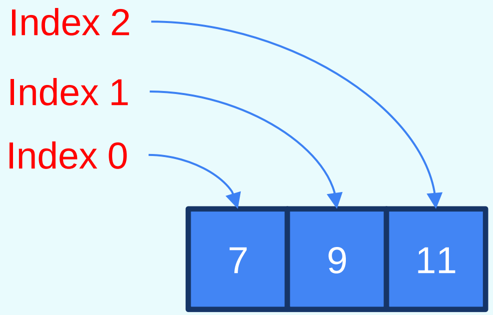
T = Type of array

Array basics

How to declare Array?

```go
var arrInteger [3]int
var arrString [3]string

arrIntegerV := [3]int{7, 9, 11}
arrStringV := [3]string{"Hi", "Gophers", "!"}
```



Array basics

Automatic Array length declaration?

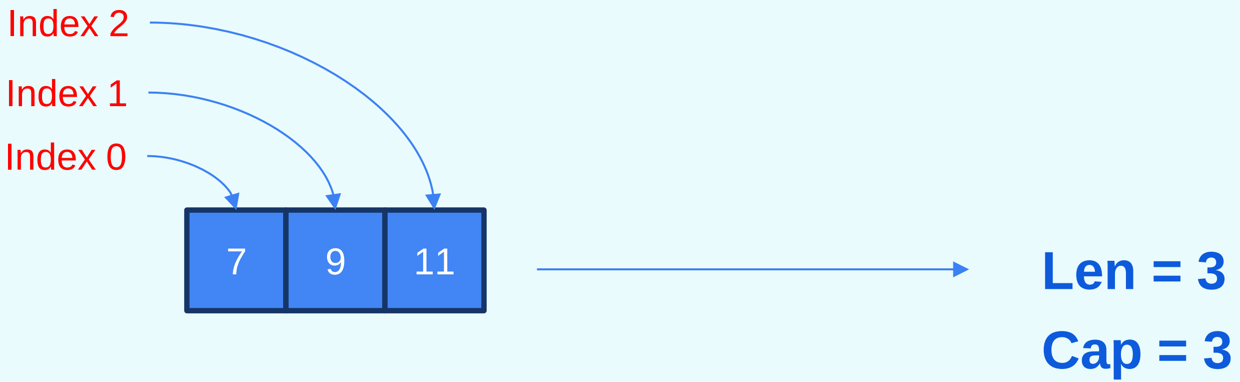
Sometimes, we don’t know the length of an array while typing its elements. Hence, Go provide ... operator to put in place of **n** in **[n]T** array type syntax. Go compiler will find the length on its own. You can only use this operator when you are defining an array with an initial value.

```
arr := [...]string{
                    ...
}
```

Array basics

Length and Capacity of an array Array?

Go provide a built-in function **len()** and **cap()** which is used to calculate the length and capacity of array.

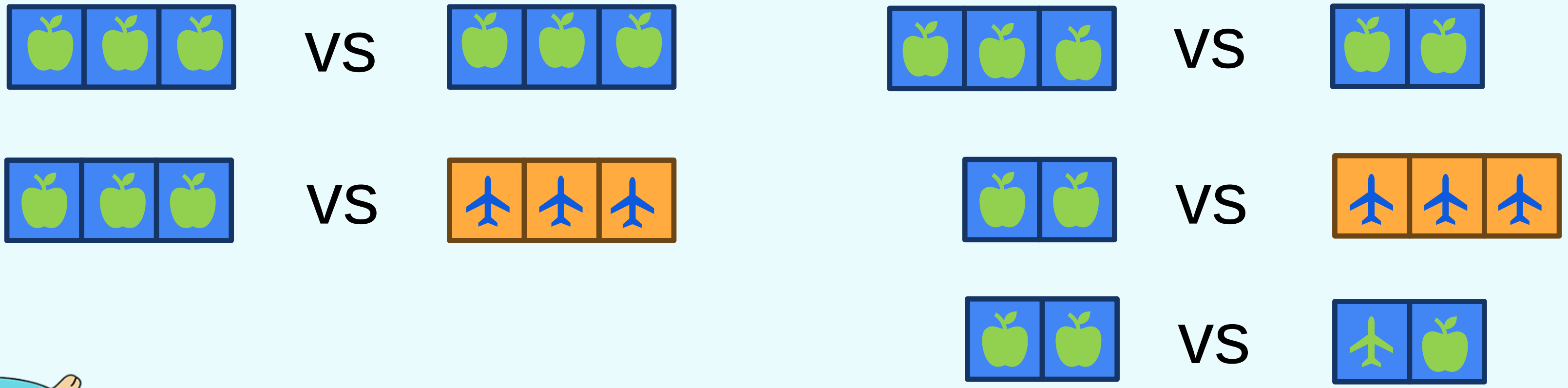


Array basics

Comparison of Array?

For an array to be the equal or the same as the second array, both arrays should be of the

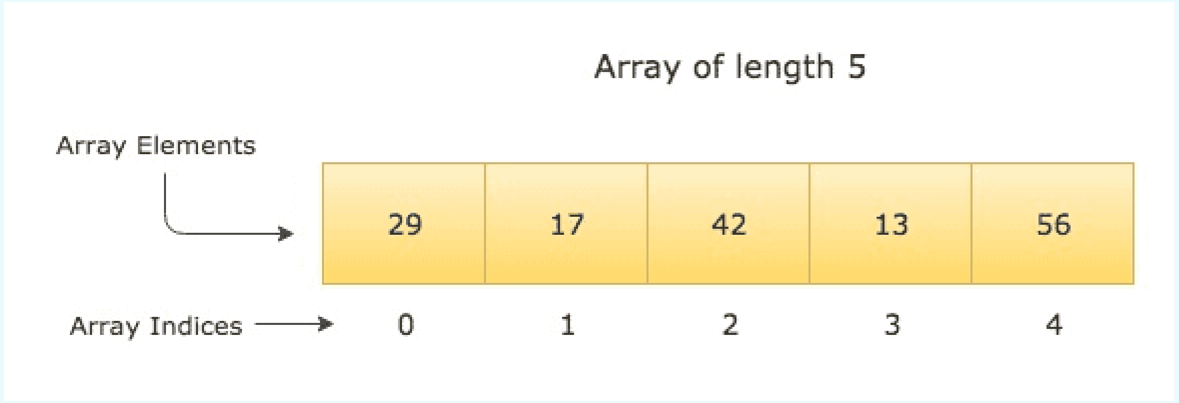
- same type
- must have the same elements in them
- all elements must be in the same order.



Array basics

Array iteration

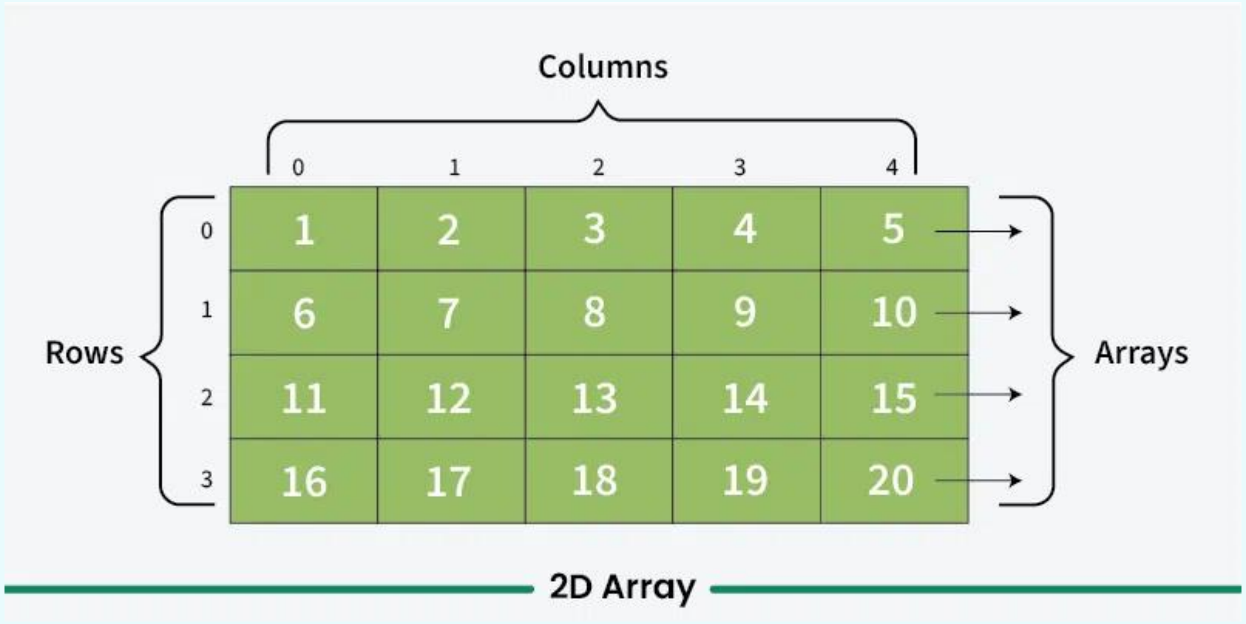
To iterate over an array, we can use **for** loop. Go provides **range** operator which returns index and value of each element of an array in for loop.



Array basics

Multi-dimensional Arrays

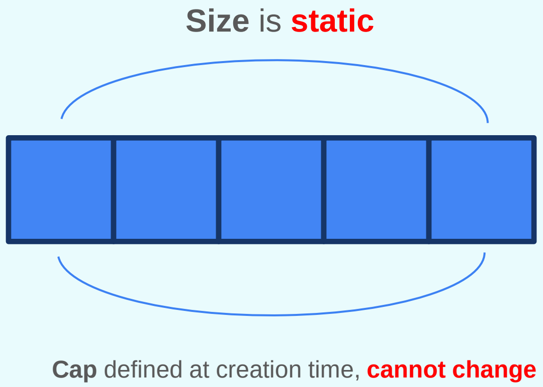
When elements of an **array** are **arrays**, then it's called a **multi-dimensional array**.



Array basics

Arrays are static

The capacity of an array is defined at creation time.
Once an array has allocated its size, the size can no longer be changed. Because the size of an array is static, it means that it only allocates memory once. This makes arrays somewhat rigid to work with. **Slices**, covered next, are similar to arrays, **but are much more flexible.**



Slices

What is Slices?

Go Slice is an **abstraction** over Go Array. A slice **doesn't store any data**; it just describes a section of an underlying array.

Slices **can change their size (capacity) dynamically**. Because a slice can be expanded to add more elements when needed, they are more commonly used than arrays.

Internally, a slice is a struct that consists of:

- a pointer to the array
- the length - is the total number of elements present in the array.
- the capacity - represents the maximum size up to which it can expand.

What is Slices?

Syntax to define a slice is pretty similar to that of an array but without specifying the elements count.

```
var  slice  []T
```

T = Type of slice

- The zero value of a slice is nil.
- A nil slice has a length and capacity of 0 and has no underlying array.

Slices

Creation of Slices? make function

Slices can be created with the built-in **make** function; this is how you create dynamically-sized arrays.

The make function **allocates an array** and **returns a slice** that refers to that array:

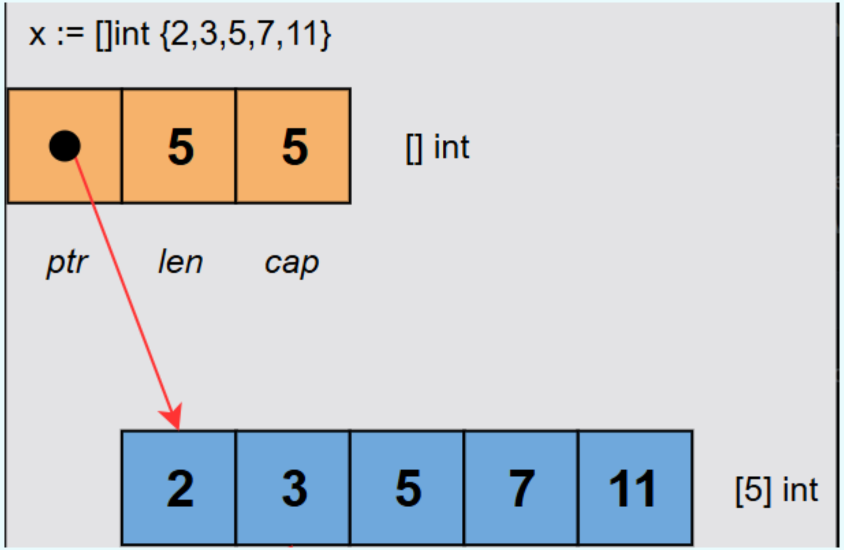
With **make**:
Defined type **default** values, manual **len** and **cap**

```
x := make([]int, 5, 5)
x := make([]int, 5, 9)
x := make([]int, 5)
```

Cap is not mandatory

Without **make**:
Defined **values**, automatic **len** and **cap**

```
x := []int{2, 3, 5, 7, 11}
```



Slices

Add to Slices? append function

A built-in **append** function appends new elements at the end of slice

Signature of append function is

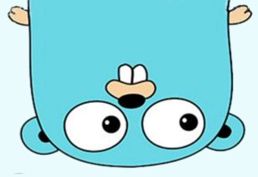
```
func append(slice []Type, elements ...Type) []Type
```

```
func append(slice []int, elements ...int) []int
```

Use append only to self assign the new slice like

s = append(s, ...)

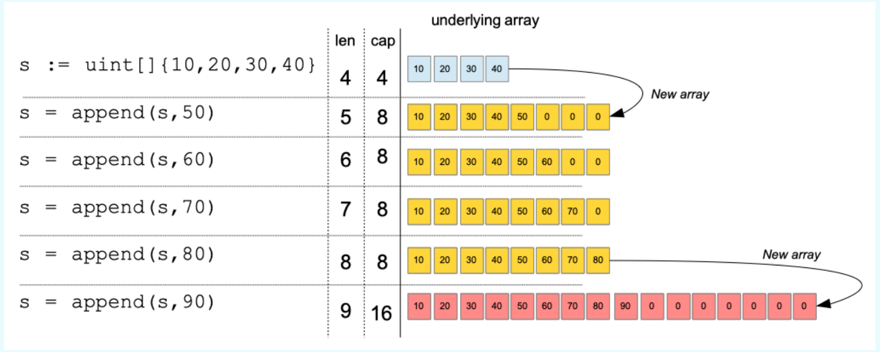
The append function will always increase the length of the new slice.



Slices

How append manages cap in Slices?

When there’s no available capacity in the underlying array for a slice, the append function will create a new underlying array, copy the existing values that are being referenced, and assign the new value.



	len	cap	underlying array
`s := uint[]{10,20,30,40}`	4	4	10 20 30 40
`s = append(s,50)`	5	8	10 20 30 40 50 0 0 0
`s = append(s,60)`	6	8	10 20 30 40 50 60 0 0
`s = append(s,70)`	7	8	10 20 30 40 50 60 70 0
`s = append(s,80)`	8	8	10 20 30 40 50 60 70 80
`s = append(s,90)`	9	16	10 20 30 40 50 60 70 80 90 0 0 0 0 0 0 0

- The initial growth rate is 2x: When the capacity is exceeded, Go typically doubles the capacity of the slice.
- For smaller slices (capacity < 1,024 elements), Go doubles the capacity each time (cap * 2).
- For larger slices (capacity >= 1,024 elements), Go switches to increasing the capacity by approximately 1.25x (a 25% increment) to avoid consuming too much memory too quickly.

Slices

Copy Slices?

Go provides built-in copy function to copy one slice into another.

It copies data to a destination slice from a source slice, and returns the number of elements copied.

Signature of copy function is as below

```
func copy(newSlice, oldSlice []Type) int
```

Slices

Extract operator in Slices?

In the below example, we have the simple slice s of integers starting from 0 to 9.

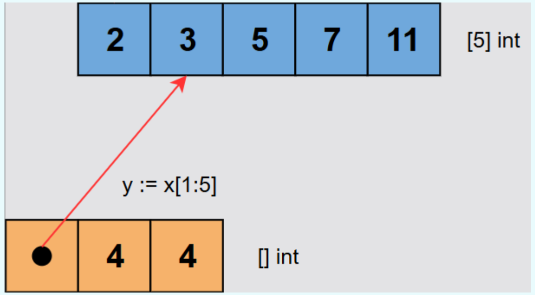
```
s:= []int{0,1,2,3,4,5,6,7,8,9}
```

s[:] means extract all elements of s **starting from 0** index **till the end**.
Hence returns all elements of s.

s[2:] means extract elements of s **starting from 2nd** index **till the end**.
Hence returns [2 3 4 5 6 7 8 9]

s[:4] means extract elements of s **starting from 0th** index **till 4th index but not including index 4**.
Hence returns [0 1 2 3]

s[2:4] means extract elements of s **starting from 2nd** index **till 4th index but not including index 4**.
Hence returns [2 3]



The important thing to remember is that any slice created by extract operator still references the same underneath array. You can use copy, make or append functions in conjugation to avoid this.

Multi-dimensional Slices?

Similar to array, slices can also be multi-dimensional. Syntax of defining multi-dimensional slices are pretty similar to arrays but without mentioning element size.

```
s := [][]int{
   {1, 2},
   {3, 4},
   {5, 6},
}
```

Slices vs Arrays?

Similar to array, slices can also be multi-dimensional. Syntax of defining multi-dimensional slices are pretty similar to arrays but without mentioning element size.

Feature	Arrays	Slices
Size	Fixed-size	No fixed length (dynamic)
Type	Strictly typed (e.g., [4]int, [3]string)	Strictly typed, but with dynamic length
Length as Part of Type	Yes, length is part of the type ([3]int is different from [4]int)	No, length is not part of the type
Memory Layout	Contiguous memory locations, elements are laid out sequentially	References an underlying array, doesn't store the data itself
Passing	Passing an array makes a copy, which can be expensive	Passing a slice only passes a reference, not the actual data
Modification of Elements	Modifying an array doesn't affect anything else	Modifying a slice affects its underlying array and other slices sharing it
Flexibility	Inflexible and expensive	Flexible and efficient, adjusts length dynamically
Building Block	A fundamental building block for slices	Built on top of arrays
Capacity	Fixed capacity	Capacity can grow dynamically with append()
Capacity Reporting	N/A	Capacity is reported using cap(slice)
Reallocation	N/A	Automatically reallocates if the capacity is exceeded (via append())

Maps

What is Map?

Maps are a special kind of data structure: an unordered collection of pairs of items, where one element of the pair is the key, and the other element, associated with the key, is the data or the value. Hence, they are also called associative arrays or dictionaries. They are ideal for looking up values, and given the key, the corresponding value can be retrieved very quickly.

A map is a reference type and declared in general as:

```
var myMap map[KeyType]ValueType
```

For example:

```
var myMap map[string]int
```

Creating a Map?

The built-in function **make** can be used to create a map

Empty map

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

Nil map

```
var countries map[string]int
```

With initial key/value pairs

```
countries := map[string]int{
   "AZ" : 994,
   "TR" : 90,
   "RU" : 7,
   "US" : 1,
}
```

Maps

Accessing Map data?

In Go, you can access data in a map by using the key associated with the value you want to retrieve. 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. Go will not throw an error, instead, it will return zero value of valueType. You can access the value associated with a specific key by providing the key in square brackets after the map variable's name.

```go
countries := map[string]int{
   "AZ" : 994,
   "TR" : 90,
   "RU" : 7,
   "US" : 1,
}

fmt.Println(countries["AZ"])
fmt.Println(countries["TR"])
fmt.Println(countries["RU"])
fmt.Println(countries["US"])
```

Maps

What is comma-ok idiom?

A syntax called "comma-ok idiom" is to check if a key exists in a map

```go
countries := map[string]int{
    "AZ": 994,
    "TR": 90,
    "RU": 7,
    "US": 1,
}

if value, ok := countries["AZ"]; ok {
    fmt.Printf("Phone country code is %d\n%v", value, ok)
} else {
    fmt.Println("Not found")
}
```

Maps

Delete a Map element?

In Go, you can delete an element from a map using the **delete** built-in function. Syntax of a delete function is as following

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

delete(countries, "RU")
```

If the key does not exist in the map, Go will not throw an error while executing delete function.

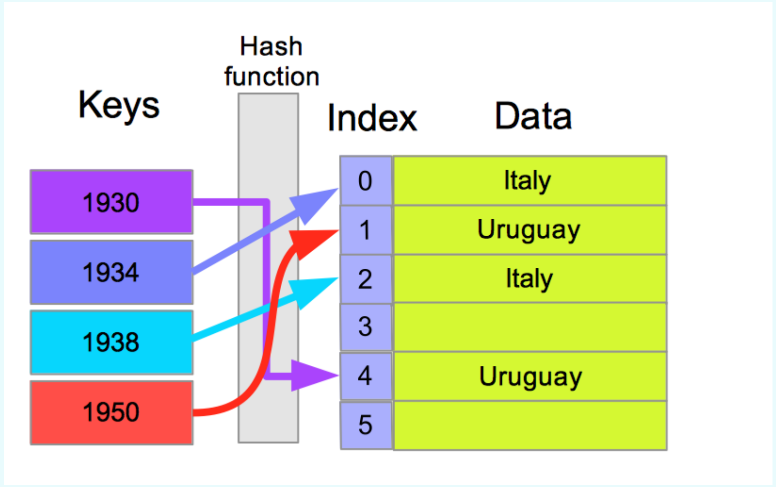
Maps

Map 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

Map iteration?

In Go, you can iterate over the elements of a map using a for loop. Maps are unordered collections of key-value pairs, so there's no guaranteed order for iteration. 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 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**.

Session Quiz
Go?

Join at
slido.com
# 3827691

Thank you.

Questions?

- Array basics
- Slices: creation, manipulation, and internal mechanics
- Maps: creating, accessing, and iterating

“Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning.”

- Albert Einstein