

%p - &data : to print the address of the data variable

%g - (used instead of %d or %e) it is mainly used to print the decimal points accurate no extra or no cut off.

%t - is to print the bool data types

%T - is used to print the type of the variable

%v - prints the values of any data type

%#v- prints the map,slice,array values and its types

%+v - useful for printing the struct types

Golang has no concept of inheritance i.e. no is-a relationship

Instead it has has-a relationship

Ex: one struct type can be used in other type by using it not by inheriting it. It has the type not it is the type.

In inheritance child class has a relationship with parent class.

```
type Title struct {  
    name string  
}
```

```
type Book struct{  
    title Title  
    auther string  
}
```

Here Book has a Title type.

An IS-A relationship is inheritance. The classes which inherit are known as sub classes or child classes. On the other hand, HAS-A relationship is composition.

In OOP, IS-A relationship is completely inheritance. This means, that the child class is a type of parent class.

On the other hand, composition means creating instances which have references to other objects.

<https://turnoff.us/geek/inheritance-versus-composition/>

You can't get the address of the map element but for the slices and arrays we can get

There is an switch type

v := interface{}

Here v has the value

```
switch v := v.(type) {  
    case int:  
        fmt.Println("its int")// inside this case the v value is of int type  
  
    case string:  
        fmt.Println("its string")// inside this case the v value is of string type  
    default:  
        // do soemthing  
}
```

```
5 func main() {  
6     var v interface{}  
7     v = 10.0  
8     switch v := v.(type) {  
9     case int:  
10        fmt.Printf("v: %v\n", v)  
11        fmt.Printf("type of v: %T\n", v)  
12  
13     case string:  
14        fmt.Printf("v: %v\n", v)  
15        fmt.Printf("type of v: %T\n", v)  
16  
17     case float64:  
18        fmt.Printf("v: %v\n", v)  
19        fmt.Printf("type of v: %T\n", v)  
20    }  
21 }  
22 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS E:\_Golang\go_programs\type_switch> go run .\main.go  
v: 10  
type of v: int  
PS E:\_Golang\go_programs\type_switch> go run .\main.go  
v: 10  
type of v: string  
PS E:\_Golang\go_programs\type_switch> go run .\main.go  
v: 10  
type of v: float64  
PS E:\_Golang\go_programs\type_switch> []
```

```
package main  
  
import "fmt"  
  
func main() {  
    var v interface{}  
    v = 10.0  
    switch v := v.(type) {  
    case int:  
        fmt.Printf("v: %v\n", v)  
        fmt.Printf("type of v: %T\n", v)  
    case string:  
        fmt.Printf("v: %v\n", v)  
        fmt.Printf("type of v: %T\n", v)  
    case float64:  
        fmt.Printf("v: %v\n", v)  
        fmt.Printf("type of v: %T\n", v)  
    }  
}
```

```
8 func main() {
9     unixTimestamp := time.Now()
10
11    fmt.Println(unixTimestamp)
12    // We stress that one must show how the reference time is formatted,
13    // not a time of the user's choosing. Thus each layout string is a
14    // representation of the time stamp,
15    // Jan 2 15:04:05 2006 MST
16    // here we can have any type of format but
17    // we need to use the reference time in our format
18    // here I need my time in the format of
19    // year % month % date
20    t := unixTimestamp.Format("2006 % Jan % 2")
21    fmt.Println(t)
```

Time format layout can be of any format but we need to use the reference time values in our own format.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS E:\_Golang\go_programs\time> go run .\main.go
2022-04-24 11:27:43.0592502 +0530 IST m=+0.006914701
2022 % Apr % 24
PS E:\_Golang\go_programs\time> █
```

String and Byte slice

```
func main() {
```

```
    var byteSlice []byte
```

```
    var stringValue string = "Logesh"
```

```
    // when I try to append the string to the bytes slice
```

```
    // its an error
```

```
        var stringValue string
```

cannot use stringValue (variable of type string) as byte value in argument to
append compiler([IncompatibleAssign](#))

[View Problem](#) No quick fixes available

```
    byteSlice = append(byteSlice, stringValue)
```

```
}
```

String is a byte slice behind the scene.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     var byteSlice []byte
7     var stringValue string = "Logesh"
8     // when I try to append the string to the bytes slice
9     // its an error
10    // but String is byte slice behind the scene
11    byteSlice = append(byteSlice, stringValue...)
12    fmt.Printf("byteSlice: %v\n", byteSlice)
13    fmt.Printf("string(byteSlice): %v\n", string(byteSlice))
14 }
15
```



```
PS E:\_Golang\go_programs\bytes>
PS E:\_Golang\go_programs\bytes> go run .\string_is_bytes_array.go
byteSlice: [76 111 103 101 115 104]
string(byteSlice): Logesh
PS E:\_Golang\go_programs\bytes>
```

String and []byte are interchangeably convertible

```
5 func main() {
6     var byteSlice []byte
7     var stringValue string = "Logesh"
8     // when I try to append the string to the bytes slice
9     // its an error
10    // but String is byte slice behind the scene
11    byteSlice = append(byteSlice, stringValue...)
12    fmt.Printf("byteSlice : %v\n", byteSlice)
13    fmt.Printf("string(byteSlice): %v\n", string(byteSlice))
14    fmt.Printf("byteSlice by %%s : %s\n", byteSlice)
15 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS E:\_Golang\go_programs\bytes> go run .\string_is_bytes_array.go
byteSlice : [76 111 103 101 115 104]
string(byteSlice): Logesh
byteSlice by %s : Logesh
PS E:\_Golang\go_programs\bytes>
```

When it comes to strings the **len()** function will return the **number bytes** that string variable holds not the number of characters it has. It is due to that the string may contain the non - english characters which may require some more bytes to store it.

To get the length of the string (number of characters) then we need to use **utf.RuneCountInString()** . this function returns the **number of characters**.

Range loop on the string variables will loop through the character by character.

```

func main() {
    str := "țără" // țără means country in Romanian
    // 't', 'a', 'r' and 'ă' are runes and each rune occupies between 1 and 4 bytes.

    //The len() built-in function returns the no. of bytes not runes or chars.
    fmt.Println("len(str):", len(str)) // -> 6, 4 runes in the string but the length is 6

    // returning the number of runes in the string
    m := utf8.RuneCountInString(str)
    fmt.Println("RuneCountInString :", m) // => 4
    // wont work in the way we expect, becoz this loops over byte by byte not char by char
    fmt.Printf("\n Using for loop with len(str)\n\n")
    for n := 0; n < len(str); n++ {
        fmt.Println("Loop count:", n)
        fmt.Printf("str[%d]: %c\n", n, str[n])
    }

    // range loop works as expected
    fmt.Printf("\n Using range loop\n\n")
    for idx, c := range str {
        fmt.Printf("index : %d , %c\n", idx, c)
    }
}

```



```

PS E:\_Golang\go_programs\strings\runes_strings>
PS E:\_Golang\go_programs\strings\runes_strings> go run .\main.go
len(str): 6
RuneCountInString : 4

```

Using for loop with len(str)

```

Loop count: 0
str[0]: Ȣ
Loop count: 1
str[1]: Ȣ
Loop count: 2
str[2]: Ȣ
Loop count: 3
str[3]: Ȣ
Loop count: 4
str[4]: Ȣ
Loop count: 5
str[5]:

```

Using range loop

```

index : 0 , Ȣ
index : 2 , Ȣ
index : 3 , Ȣ
index : 4 , Ȣ

```

```
PS E:\_Golang\go_programs\strings\runes_strings>
```



Good job!

Question 1:

Which byte slice below equals to the "**keeper**" string?

```
1 | // Here are the corresponding code points for the runes of "keeper":  
2 | // k => 107  
3 | // e => 101  
4 | // p => 112  
5 | // r => 114
```

`[]byte{107, 101, 101, 112, 101, 114}`

`[]byte{112, 101, 101, 112, 114, 101}`

`[]byte{114, 101, 112, 101, 101, 112}`

`[]byte{112, 101, 101, 114, 107, 101}`



Good job!

Question 2:

What does this code print?

```
1 // Code points:  
2 // g => 103  
3 // o => 111  
4 fmt.Println(string(103), string(111))
```

To convert the int to string (1 to "1")
We need to use the strconv.Itoa() function

103 111

g o

n o

"103 111"



Good job!

Question 3:

What does this code print?

```
1 | const word = "gökyüzü"
2 | bword := []byte(word)
3 |
4 | // ö => 2 bytes
5 | // ü => 2 bytes
6 | fmt.Println(utf8.RuneCount(bword), len(word), len(string(word[1])))
```

7 10 2

10 7 1

10 7 2

7 7 1



Good job!

Question 4:

Which one below is true?

for range loops over the bytes of a string

for range loops over the runes of a string



Good job!

This was discussed in Lecture 138: [How can you decode a string? >](#)

Question 5:

For a utf-8 encoded string value, which one below is true?

runes always start and end in the same indexes

runes may start and end in different indexes

bytes may start and end in different indexes

When we are given an interval we can say the interval upper and lower limits

Ex: $[0, n)$

- [- says that the number is included i.e. 0 is included
-) - says that the number is excluded i.e. n is excluded

[] - square brackets means included

() - parenthesis or round bracket means excluded



Good job!

Right. The square-brace means: "inclusion". The parenthesis means: "exclusion". So, $[0, 5)$ means: 0, 1, 2, 3, 4. It's called the "half open interval notation".

Question 4:

What does $[0, 5)$ mean?

Note that: This is not a Go syntax. It's mathematical notation. So, please do not try to use it in code.



A range of numbers between 0 and 5 (*excluding 5*)



A range of numbers between 0 and 5 (*including 5*)



Just 0 and 5



Just 0 and 4

func RuneCountInString

```
func RuneCountInString(s string) (n int)
```

RuneCountInString is like RuneCount but its input is a string.

▼ Example

```
package main

import (
    "fmt"
    "unicode/utf8"
)

func main() {
    str := "Hello, 世界"
    fmt.Println("bytes =", len(str))
    fmt.Println("runes =", utf8.RuneCountInString(str))
}
```

Output:

```
bytes = 13
runes = 9
```

Len returns the number of bytes the string takes not the number of characters it has.

To find the number of characters it has we can use the

[**RuneCountInString**](#) from the `utf8` package

In the switch statement we don't need break at the end of each case clause. And also we have the default clause. This default class can be written anywhere inside the switch statement.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println("Working with switch, with expression")
7     a := 10
8     // at the end of each case there is a virtual break
9     // so its not trying to execute next case like in other langs
10    switch a {
11        // the default clause can be in anywhere in the switch statement.
12        // it will be executed at the end if no case is statisfies the condition
13        default:
14            fmt.Println("a is", a)
15        case 2:
16            fmt.Println("a is 2, i found it")
17        case 10:
18            fmt.Println("a is 10, i found it man")
19    }
20 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS E:\_Golang\go_programs\switch_statements\default_clause> go run .\default_clause.go
Working with switch, with expression
a is 10, i found it man
PS E:\_Golang\go_programs\switch_statements\default_clause> █
```

```
-  
9 func main() {  
10     ar := os.Args[1]  
11     a, _ := strconv.Atoi(ar)  
12     // at the end of each case there is a virtual break  
13     // so its not trying to execute next case like in other langs  
14     switch a {  
15         // the default clause can be in anywhere in the switch statement.  
16         // it will be executed at the end if no case is statisfies the condition  
17         default:  
18             fmt.Println("a is", a)  
19         case 2, 4: // multiple conditions(this case will be executed if a is either 2 or 4)  
20             fmt.Println("a is 2, i found it")  
21     }  
22 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS E:\_Golang\go_programs\switch_statements\default_clause> go run .\default_clause.go 2  
a is 2, i found it  
PS E:\_Golang\go_programs\switch_statements\default_clause> go run .\default_clause.go 4  
a is 2, i found it  
PS E:\_Golang\go_programs\switch_statements\default_clause> go run .\default_clause.go 1  
a is 1  
PS E:\_Golang\go_programs\switch_statements\default_clause> █
```

fallthrough keyword is used in switch statement in golang. This keyword is used in switch case block. If the **fallthrough** keyword is present in the case block, then it will transfer control to the next case even though the current case might have matched.

fallthrough needs to be final statement within the switch block. If it is not then compiler raise error

`fallthrough statement out of place`

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     i := 45
7     switch {
8     case i < 10:
9         fmt.Println("i is less than 10")
10    fallthrough // if this line is executed then the next case block is executed without validating the expression
11    case i < 50:
12        fmt.Println("i is less than 50")
13        fallthrough
14    case i < 100:
15        fmt.Println("i is less than 100")
16    }
17 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PS E:_Golang\go_programs\switch_statements\fallthrough> go run .\main.go

i is less than 50

i is less than 100

PS E:_Golang\go_programs\switch_statements\fallthrough>

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     a := 100
7     // no initialize and post statement but the condition is hidden (default) true
8     // its an infinite loop
9     for {
10         fmt.Println("In Loop")
11         if a == 103 {
12             fmt.Println("Breaking the Loop")
13             break
14         }
15         a++
16     }
17 }
```

Here i have used the break statement else it will be an infinite loop

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS E:\_Golang\go_programs\for_loop> go run .\main.go
In Loop
In Loop
In Loop
In Loop
Breaking the Loop
PS E:\_Golang\go_programs\for_loop> █
```

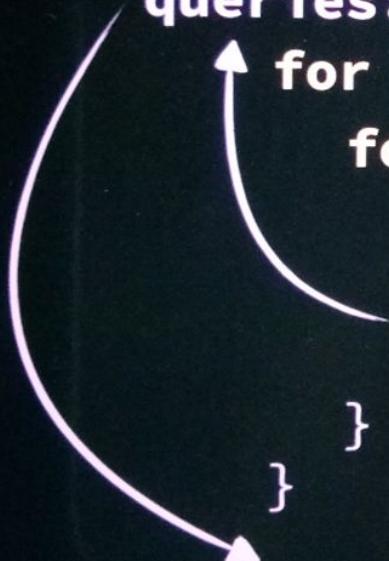
LABELED BREAK

Labeled break breaks the labeled statement

Here the parent (outer) loop is breaked since that loop is labeled as queries and we break with that label

queries:

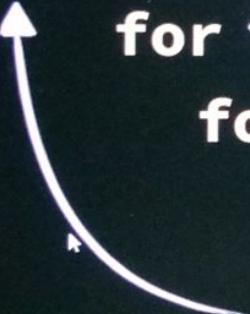
```
for _, q := range query {  
    for i, w := range words {  
        // ...  
        break queries  
    }  
}
```



LABELED CONTINUE

Labeled continue continues from the labeled loop

```
queries:  
for _, q := range query {  
    for i, w := range words {  
        // ...  
        continue queries  
    }  
}
```





Good job!

They can be used throughout the function, even before their declaration inside the function. This also what makes goto statement jump to any label within a function.

Question 1:

Which scope does a label belong to?

To the scope of the statement that it is in

To the body of the function that it is in

To the package scope that it is in

Yep. This is an unlabeled break. So, it breaks the closest statement, which in here, it's that switch statement. And, since it only breaks the switch, the loop will keep continue.

Question 3:

Given the following program, does the loop terminate after the break statement?

```
1 | package main
2 |
3 | func main() {
4 |     main:
5 |         for {
6 |             switch "A" {
7 |                 case "A":
8 |                     break // <- here!
9 |                 case "B":
10 |                     continue main
11 |             }
12 |         }
13 | }
```

Note: "to terminate" means "to quit". Remember, statements can also terminate. This means that the statement will finish executing. It doesn't mean that a program will finish.

No, the break will only terminate the switch but the loop will continue

Yes, the break will terminate the loop

Yes, the break will terminate the switch

Question 4:

Given the following program, does the loop ever end?

```
1 package main
2
3 func main() {
4     flag := "A"
5
6     main:
7     for {
8         switch flag {
9             case "A":
10                flag = "B"
11                break
12            case "B":
13                break main
14            }
15        }
16    }
```

No, the loop will never end.

Yes, the first break will terminate the loop.

Yes, the second break will terminate the loop.



Question 5:

Given the following program, what does the first break do?

Note that: There's an infinite loop.

```
1 | package main
2 |
3 | func main() {
4 |     for {
5 |         switcher:
6 |             switch 1 {
7 |                 case 1:
8 |                     switch 2 {
9 |                         case 2:
10 |                             break switcher
11 |                         }
12 |                     }
13 |                     break
14 |                 }
15 | }
```

It breaks from the 2nd switch causing the program will loop indefinitely

It breaks from the 2nd switch and then the 2nd break will terminate the loop

It breaks from the 1st switch and then the 2nd break will terminate the loop

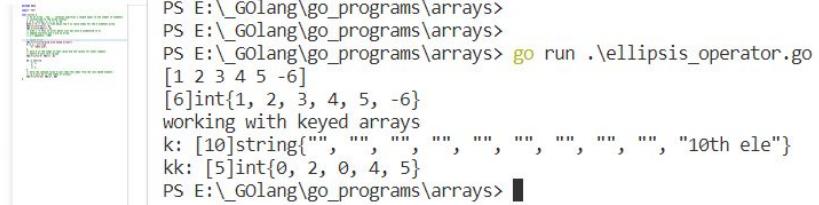
Go automatically sets the uninitialized variables to their zero values

Array is a **collection elements**. It stores the **same type** in **contiguous** memory locations

```

1 package main
2
3 import "fmt"
4
5 func main() {
6     // In an array , the ... notation specifies a length equal to the number of elements
7     // initialized in the array literal.
8     a := [...]int{1, 2, 3, 4, 5, 6}
9     a[5] = -6 // this is fine becoz the 5 is valid index for the 6 elements array
10    fmt.Printf("%v\n", a)
11    fmt.Printf("%#v\n", a)
12    // a[6] = -7 this is error becoz list has only 6 elements(0 to 5)
13    // append function won't wrk in array
14    // a = append(a, -100)
15
16    // keyed array
17    fmt.Println("working with keyed arrays")
18    k := [...]string{
19        9: "10th ele",
20    }
21    // here 9 is the index of that value and the values for other indexes
22    // before 9 were zero valued
23    fmt.Printf("k: %#v\n", k)
24
25    kk := [5]int{
26        3: 4,
27        5,      Unkeyed element gets the index from the last keyed element
28        1: 2,
29    }
30    // here the unkeyed value 5 will take the index from the last keyed element.
31    // so 5 will be in the index of 4(last)
32    fmt.Printf("kk: %#v\n", kk)
33}
34

```



PS E:_Golang\go_programs\arrays>
PS E:_Golang\go_programs\arrays> go run .\ellipsis_operator.go
[1 2 3 4 5 -6]
[6]int{1, 2, 3, 4, 5, -6}
working with keyed arrays
k: [10]string{"", "", "", "", "", "", "", "", "", "10th ele"}
kk: [5]int{0, 2, 0, 4, 5}
PS E:_Golang\go_programs\arrays>

Unkeyed element gets the index from the last keyed element

=GO rand

func Intn

```
func Intn(n int) int
```

Intn returns, as an `int`, a non-negative pseudo-random number in the half-open interval $[0, n)$ from the default Source. It panics if $n \leq 0$.

▼ Example

```
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    // Seeding with the same value results in the same random sequence each run.
    // For different numbers, seed with a different value, such as
    // time.Now().UnixNano(), which yields a constantly-changing number.
    rand.Seed(86)
    fmt.Println(rand.Intn(100))
    fmt.Println(rand.Intn(100))
    fmt.Println(rand.Intn(100))

}
```

To generate the random number we can use the rand package from math folder.
Also the rand doesn't generates the true random numbers. We can make the rand function to generate the true random numbers by Seed the rand package with the random number.

```
1 package main
2
3 import (
4     "fmt"
5     "math/rand"
6 )
7
8 func main() {
9     fmt.Println(rand.Intn(100))
10    fmt.Println(rand.Intn(100))
11    fmt.Println(rand.Intn(100))
12    // everytime i ran this code it will give the same results,
13    // which says the rand is not truly random
14
15 }
```

```
PS E:\_Golang\go_programs\random_numbers> PS E:\_Golang\go_programs\random_numbers> go run .\main.go  
81  
87  
47  
PS E:\_Golang\go programs\random numbers>
```

```
1 package main
2
3 import (
4     "fmt"
5     "math/rand"
6     "time"
7 )
8
9 func main() {
10    fmt.Println(rand.Intn(100))
11    fmt.Println(rand.Intn(100))
12    // everytime i ran this code it will give the same results,
13    // which says the rand is not truly random
14
15    // to make it truly random we need to randomize the rand seed
16    rand.Seed(time.Now().UnixNano())
17    // time is the only thing that is unique all time. so by giving the unique seed value
18    // everytime the rand will generate the true random numbers
19    fmt.Println("\nAfter seeded\n")
20    fmt.Println(rand.Intn(100))
21    fmt.Println(rand.Intn(100))
22
23 }
```



```
PS E:\_Golang\go_programs\random_numbers>
PS E:\_Golang\go_programs\random_numbers> go run .\main.go
81
87
After seeded
75
52
PS E:\_Golang\go_programs\random_numbers> go run .\main.go
81
87
After seeded
31
73
PS E:\_Golang\go_programs\random_numbers> go run .\main.go
81
87
After seeded
62
81
PS E:\_Golang\go_programs\random_numbers> go run .\main.go
81
87
After seeded
18
36
PS E:\_Golang\go_programs\random_numbers>
```

Almost random values

%g

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     a := 10.78
7     fmt.Printf("a is : %f\n", a)
8     fmt.Printf("a is : %e\n", a)
9     fmt.Printf("a is : %g\n", a)
10    // %g is used in place of %e and %f
11    // where we need to have the
12    // decimal values as given
13    // no extra or no cutoff decimal values
14 }
15
```



```
PS E:\_Golang\go_programs\format_verbs>
PS E:\_Golang\go_programs\format_verbs> go run .\percent_g.go
a is : 10.780000
a is : 1.078000e+01
a is : 10.78
PS E:\_Golang\go_programs\format_verbs> □
```



Good job!

Yes! There are no elements in the element list. So, Go sets the length of the array to 0.

This was discussed in Lecture 72: [What is an array in Go? >](#)

Question 2:

What is the type and length of the gadgets array?

```
1 | gadgets := [...]string{}
```

[0]string and 0

[0]string{} and 0

[1]string and 1

[1]string{} and 1





Good job!

Yes! gadget's type is [3]string whereas gears's type is [1]string.

Question 4:

Are the following arrays comparable?

```
1 | gadgets := [3]string{"Confused Drone"}  
2 | gears    := [...]string{"Confused Drone"}  
3 |  
4 | fmt.Println(gadgets == gears)
```

Yes, because they have identical types and elements

No, because their types are different

No, because their elements are different



Good job!

Yes! When you assign an array, Go creates a copy of the original array. So, gadgets and gears arrays are not connected. Changing one of them won't effect the other one.

Question 5:

What does this program print?

```
1 | gadgets := [3]string{"Confused Drone", "Broken Phone"}  
2 | gears     := gadgets  
3 |  
4 | gears[2] = "Shiny Mouse"  
5 |  
6 | fmt.Printf("%q\n", gadgets)
```

["Confused Drone" "Broken Phone" "Shiny Mouse"]

["Confused Drone" "Broken Phone" ""]

["" "" "Shiny Mouse"]

["" "" ""]

Slice

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     // uninitialized slice is nil
7     var s1 []string
8     // initialized slice ith zero element
9     var s2 = []string{}
10
11    fmt.Printf("s1: %#v\n", s1)
12    fmt.Printf("s2: %#v\n", s2)
13
14    if s1 == nil {
15        fmt.Println("s1 is nil")
16    } else {
17        fmt.Println("s1 is not nil")
18    }
19
20    if s2 == nil {
21        fmt.Println("s2 is nil")
22    } else {
23        fmt.Println("s2 is not nil")
24    }
25    // An Empty slice is an initialized slice
26    // An Nil slice is an un-initialized slice
27    fmt.Println("len of s1:", len(s1))
28    fmt.Println("len of s2:", len(s2))
29    //Both the slice has same length - 0
30 }
```



```
PS E:\_Golang\go_programs\slices\empty>
PS E:\_Golang\go_programs\slices\empty> go run .\main.go
s1: []string(nil)
s2: []string{}
s1 is nil
s2 is not nil
len of s1: 0
len of s2: 0
PS E:\_Golang\go_programs\slices\empty> []
```

Can't sort an array using sort.Ints() since Ints() accepts only the slice

```
func main() {
    var a1 = [5]int{5, 3, 9, 2, 6}
    var s1 = [5]int{5, 3, 9, 2, 6}

    fmt.Println(a1)
    fmt.Println(s1)

    sort.Ints(a1)
    sort.Ints(s1)

    fmt.Println("\nAfter sorting\n\n")
    fmt.Printf("a1: %#v\n", a1)
    fmt.Printf("s1: %#v\n", s1)
}
```

cannot use a1 (variable of type [5]int) as []int value in argument to sort.Ints compiler([IncompatibleAssign](#))

[View Problem](#) No quick fixes available

Solution

```
1 package main
2
3 import (
4     "fmt"
5     "sort"
6 )
7
8 func main() {
9     var a1 = [5]int{5, 3, 9, 2, 6}
10    var s1 = []int{5, 3, 9, 2, 6}
11
12    fmt.Printf("a1: %#v\n", a1)
13    fmt.Printf("s1: %#v\n", s1)
14
15    sort.Ints(a1[:]) ← Blue arrow pointing here
16    sort.Ints(s1)
17    fmt.Println("\nAfter sorting\n\n")
18    fmt.Printf("a1: %#v\n", a1)
19    fmt.Printf("s1: %#v\n", s1)
20
21 }
22
```

```
PS E:\_GOLang\go_programs\slices\array_as_slice>
PS E:\_Golang\go_programs\slices\array_as_slice> go run .\passing_array_as_slice.go
a1: [5]int{5, 3, 9, 2, 6}
s1: []int{5, 3, 9, 2, 6}

After sorting

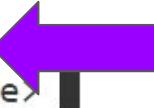
a1: [5]int{2, 3, 5, 6, 9}
s1: []int{2, 3, 5, 6, 9}
PS E:\_Golang\go_programs\slices\array_as_slice> 
```

Now we sliced our array to make it like an slice.

When we slice an array then the return is an slice

```
5 func main() {  
6     var a1 = [5]int{5, 3, 9, 2, 6}  
7     // var s1 = []int{5, 3, 9, 2, 6}  
8  
9     fmt.Printf("By Index - a1[0] : %d and the Type is %T\n", a1[0], a1[0])  
10    fmt.Printf("By Slice - a1[0:1]: %d and the Type is %T\n", a1[0:1], a1[0:1])  
11}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS E:\_Golang\go_programs\slices\array_as_slice> go run .\passing_array_as_slice.go  
By Index - a1[0] : 5 and the Type is int  
By Slice - a1[0:1]: [5] and the Type is []int   
PS E:\_Golang\go_programs\slices\array_as_slice>
```

Here we can see that the sliced array result is an Slice.



Good job!

That's right. This is an empty slice, it doesn't contain any elements, so its length is zero.

Question 4:

What is the length of the following slice?

1 | []uint64{}

64

1

0

Error



Good job!

Yep! A slice stores its elements in a backing that the slice references (or points to).

Question 1:

Where does a slice store its elements?

In the slice value

In a global backing array that is shared by all the slices

In a backing array that is specific to a slice

In a backing array that the slice references



Good job!

Yes! Slicing returns a new slice that references to some segment of the same backing array.

Question 2:

When you slice a slice, what value does it return?

```
1 // example:  
2 s := []string{"i'm", "a", "slice"}  
3 s[2:] // <-- slicing
```

- It returns a new slice value with a new backing array
- It returns the existing slice value with a new backing array
- It returns a new slice value with the same backing array



Good job!

Yes! When a slice is created by slicing an array, that array becomes the backing array of that slice.

Question 4:

Which one is the backing array of "slice2"?

```
1 | arr := [...]int{1, 2, 3}  
2 | slice1 := arr[2:3]  
3 | slice2 := slice1[:1]
```

arr

slice1

slice2

A hidden backing array



Good job!

That's right. A slice literal always creates a new backing array.

Question 5:

Which answer is correct for the following slices?

- 1 | slice1 := []int{1, 2, 3}
- 2 | slice2 := []int{1, 2, 3}

Their backing array is the same.

Their backing arrays are different.

They don't have any backing arrays.

```
5 func main() {
6     // var a1 = [5]int{5, 3, 9, 2, 6}
7     var s1 = []int{} // initialized but empty
8     var s2 []int      // uninitialized so nil
9     fmt.Printf("s1: %#v\n", s1)
10    fmt.Printf("len(s1): %d\n", len(s1))
11    fmt.Printf("s2: %#v\n", s2)
12    fmt.Printf("len(s2): %d\n", len(s2))
```

[PROBLEMS](#)[OUTPUT](#)[DEBUG CONSOLE](#)[TERMINAL](#)

```
PS E:\_Golang\go_programs\slices\array_as_slice> go run .\passing_array_as_slice.go
s1: []
len(s1): 0
s2: []
len(s2): 0
PS E:\_Golang\go_programs\slices\array_as_slice> █
```



Good job!

That's right. `array[:5]` returns a slice with the first 5 elements of the `array` (len is 5), but there are 5 more elements in the backing array of that slice, so in total its capacity is 10.

Question 3:

Which slice value does the following slice header describe?

```
1 SLICE HEADER:  
2 + Pointer : 100th  
3 + Length  : 5  
4 + Capacity: 10  
5  
6 Assume that the backing array is this one:  
7 var array [10]string
```

`array[5:]`

`array[:5]`

`array[3:]`

`array[100:]`



Good job!

`array` is $1000 \times \text{int64}$ (8 bytes) = 8000 bytes. Assigning an array copies all its elements, so `array2` adds additional 8000 bytes. A slice doesn't store anything on its own. Here, it's being created from array2, so it doesn't allocate a backing array as well. A slice header's size is 24 bytes. So in total: This program allocates 16024 bytes.

Question 5:

What is the total memory usage of this code?

```
1 | var array [1000]int64
2 |
3 | array2 := array
4 | slice := array2[:]
```

1024 bytes

2024 bytes

3000 bytes

16024 bytes

Backing array

```
5 func main() {
6     var s1 = []int{1, 2, 3, 4, 5}
7
8     s2 := s1[0:2]
9
10    fmt.Printf("len(s2) : %d\n", len(s2))
11    fmt.Printf("s2: %#v\n", s2)
12    fmt.Printf("Getting 4th index from s2: %d\n", s2[4]) ←
13    fmt.Printf("len(s2) : %d\n", len(s2))
14 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS E:\_Golang\go_programs\slices\backing_array> go run .\main.go
len(s2) : 2
s2: []int{1, 2}
panic: runtime error: index out of range [4] with length 2
goroutine 1 [running]:
main.main()
    E:/_Golang/go_programs/slices/backing_array/main.go:12 +0x106
exit status 2
PS E:\_Golang\go_programs\slices\backing_array>
```

```
5 func main() {
6     var s1 = []int{1, 2, 3, 4, 5}
7
8     s2 := s1[0:2]
9
10    fmt.Printf("len(s2) : %d\n", len(s2))
11    fmt.Printf("s2: %#v\n", s2)
12    fmt.Printf("Getting 4th index from s2: %d\n", s2[4:5]) ←
13    fmt.Printf("len(s2) : %d\n", len(s2))
14 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS E:\_Golang\go_programs\slices\backing_array> go run .\main.go
len(s2) : 2
s2: []int{1, 2}
Getting 4th index from s2: [5]
len(s2) : 2
PS E:\_Golang\go_programs\slices\backing_array>
```

In the above example even though the length of the slice is 2 We can access the 4th index since the slice has the backing array. The Backing array length is 5 in our case. So i can access the last index(4).

We could access that only by the **slice expression** not by the normal indexing

```
5 func main() {
6     var s1 = []int{1, 2, 3, 4, 5}
7
8     s2 := s1[0:2]      We can see our backing array length that can be accessed or visible for this slice by using the cap(slice)
9
10    fmt.Printf("len(s2) : %d\t cap(s2) : %d\n", len(s2), cap(s2))
11    fmt.Printf("s2: %#v\n", s2)
12    fmt.Printf("Getting 4th index from s2: %d\n", s2[4:5])
13    fmt.Printf("len(s2) : %d\t cap(s2) : %d\n", len(s2), cap(s2))
14 }
15
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS E:\_Golang\go_programs\slices\backing_array> go run .\main.go
len(s2) : 2      cap(s2) : 5
s2: []int{1, 2}  ██████████
Getting 4th index from s2: [5]
len(s2) : 2      cap(s2) : 5
PS E:\_Golang\go_programs\slices\backing_array> █
```



Good job!

Question 1:

What is the difference between the length and capacity of a slice?

They are the same

The length is always greater than the capacity

The capacity is always greater than the capacity

The length describes the length of a slice but a capacity describes the length of the backing array beginning from the first element of the slice



Good job!

Question 2:

What is the capacity of a nil slice?

It is equal to its length + 1

It is nil

0

1



Good job!

Right! `words[0:]` slices for the rest of the elements, which in turn returns a slice with the same length as the original slice. Beginning from the first array element, the `words` slice's backing array contains 6 elements; so its capacity is also 6.

This was discussed in Lecture 109: [What is the capacity of a slice? >](#)

Question 4:

What are the length and capacity of the 'words' slice?

```
1 | words := []string{"lucy", "in", "the", "sky", "with", "diamonds"}  
2 | words = words[:0]
```

Length: 0 - Capacity: 0

Length: 6 - Capacity: 6

Length: 0 - Capacity: 6

Length: 5 - Capacity: 10



Good job!

Right! `words[0:]` slices for the rest of the elements, which in turn returns a slice with the same length as the original slice: 6. Beginning from the first array element, the `words` slice's backing array contains 6 elements; so its capacity is also 6.

Question 5:

What are the length and capacity of the 'words' slice?

```
1 | words := []string{"lucy", "in", "the", "sky", "with", "diamonds"}  
2 | words = words[0:]
```

Length: 0 - Capacity: 0

Length: 6 - Capacity: 6

Length: 0 - Capacity: 6

Length: 5 - Capacity: 10

```
5 func main() {  
6     var s1 = []int{1, 2, 3, 4, 5}  
7     fmt.Printf("s1 : %#v\n", s1)  
8     s1 = append(s1, s1[2:]...)  
9     fmt.Printf("s1 : %#v\n", s1)  
10    s1 = append(s1[:2], []int{9, 16, 25}...)  
11    fmt.Printf("s1 : %#v\n", s1)  
12    fmt.Printf("s1[:cap(s1)]: %#v\n", s1[:cap(s1)])  
13  
14 }
```

Inserting elements in middle of the slice

[PROBLEMS](#) [OUTPUT](#) [DEBUG CONSOLE](#) [TERMINAL](#)

```
PS E:\_Golang\go_programs\slices\insert> go run .\insert_middle.go  
s1 : []int{1, 2, 3, 4, 5}  
s1 : []int{1, 2, 3, 4, 5, 3, 4, 5}  
s1 : []int{1, 2, 9, 16, 25}  
s1[:cap(s1)]: []int{1, 2, 9, 16, 25, 3, 4, 5, 0, 0}  
PS E:\_Golang\go_programs\slices\insert> □
```



Good job!

Yes, it overwrites the last element, then it adds two element. However, there is not enough space to do that, so it allocates a new backing array.

Question 1:

Which append call below does need to allocate a new backing array?

```
1 | words := []string{"lucy", "in", "the", "sky", "with", "diamonds"}
```

words = append(words[:3], "crystals")

words = append(words[:4], "crystals")

words = append(words[:5], "crystals")

words = append(words[:5], "crystals", "and", "diamonds")



Good job!

line #2 overwrites the 2nd and 3rd elements. line #3 appends ["with" "diamonds"] after the ["lucy" "is" "everywhere"].

Question 2:

What does the program print?

```
1 | words := []string{"lucy", "in", "the", "sky", "with", "diamonds"}  
2 | words = append(words[:1], "is", "everywhere")  
3 | words = append(words, words[len(words)+1:cap(words)]...)
```

lucy in the sky with diamonds

lucy is everywhere in the sky with diamonds

lucy is everywhere with diamonds

lucy is everywhere

Slice Expression

The syntax has been introduced in Go 1.2, as I mentioned in "[Re-slicing slices in Golang](#)". It is documented in [Full slice expressions](#):

```
a[low : high : max]
```

constructs a slice of the same type, and with the same length and elements as the simple slice expression `a[low : high]`.

Additionally, it controls the resulting slice's capacity by setting it to `max - low`. Only the first index may be omitted; it defaults to 0.

After slicing the array `a`:

```
a := [5]int{1, 2, 3, 4, 5}
t := a[1:3:5]
```

https://go.dev/ref/spec#Slice_expressions

the slice `t` has type `[]int`, length 2, capacity 4, and elements

```
t[0] == 2
t[1] == 3
```

https://go.dev/ref/spec#Making_slices_maps_and_channels

Making slices, maps and channels

The built-in function `make` takes a type `T`, optionally followed by a type-specific list of expressions. The [core type](#) of `T` must be a slice, map or channel. It returns a value of type `T` (not `*T`). The memory is initialized as described in the section on [initial values](#).

Call	Core type	Result
<code>make(T, n)</code>	<code>slice</code>	<code>slice of type T with length n and capacity n</code>
<code>make(T, n, m)</code>	<code>slice</code>	<code>slice of type T with length n and capacity m</code>
<code>make(T)</code>	<code>map</code>	<code>map of type T</code>
<code>make(T, n)</code>	<code>map</code>	<code>map of type T with initial space for approximately n elements</code>
<code>make(T)</code>	<code>channel</code>	<code>unbuffered channel of type T</code>
<code>make(T, n)</code>	<code>channel</code>	<code>buffered channel of type T, buffer size n</code>

Each of the size arguments `n` and `m` must be of [integer type](#), have a [type set](#) containing only integer types, or be an untyped [constant](#). A constant size argument must be non-negative and [representable](#) by a value of type `int`; if it is an untyped constant it is given type `int`. If both `n` and `m` are provided and are constant, then `n` must be no larger than `m`. If `n` is negative or larger than `m` at run time, a [run-time panic](#) occurs.

```
s := make([]int, 10, 100)      // slice with len(s) == 10, cap(s) == 100
s := make([]int, 1e3)          // slice with len(s) == cap(s) == 1000
s := make([]int, 1<<63)       // illegal: len(s) is not representable by a value of type int
s := make([]int, 10, 0)        // illegal: len(s) > cap(s)
c := make(chan int, 10)        // channel with a buffer size of 10
m := make(map[string]int, 100) // map with initial space for approximately 100 elements
```

Calling `make` with a map type and size hint `n` will create a map with initial space to hold `n` map elements. The precise behavior is implementation-dependent.

https://go.dev/ref/spec#Appending_and_copying_slices

The function `copy` copies slice elements from a source `src` to a destination `dst` and returns the number of elements copied. The `core types` of both arguments must be slices with `identical` element type. The number of elements copied is the minimum of `len(src)` and `len(dst)`. As a special case, if the destination's core type is `[]byte`, `copy` also accepts a source argument with core type `string`. This form copies the bytes from the string into the byte slice.

```
copy(dst, src []T) int  
copy(dst []byte, src string) int
```

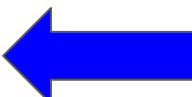
Examples:

```
var a = [...]int{0, 1, 2, 3, 4, 5, 6, 7}  
var s = make([]int, 6)  
var b = make([]byte, 5)  
n1 := copy(s, a[0:])          // n1 == 6, s == []int{0, 1, 2, 3, 4, 5}  
n2 := copy(s, s[2:])          // n2 == 4, s == []int{2, 3, 4, 5, 4, 5}  
n3 := copy(b, "Hello, World!") // n3 == 5, b == []byte("Hello")
```

```

1 package main
2
3 import "fmt"
4
5 func main() {
6     a := []int{1, 2, 3, 4, 5}
7     fmt.Printf("a: %v\n", a)
8     // backup a to aa
9     aa := make([]int, len(a))
10    copy(aa, a)
11    fmt.Printf("aa: %v\n", aa)
12    fmt.Println("\nChanging value of a\n\n")
13    a[0] = 100 // only a changes since the backing array of a and aa is different
14    fmt.Printf("a: %v\n", a)
15    fmt.Printf("aa: %v\n", aa)
16
17    fmt.Println("\nAnother way to copy the slice\n\n")
18    b := []int{100, 200, 300, 400, 500}
19    fmt.Printf("b: %v\n", b)
20    bb := append([]int(nil), b...)
21    fmt.Printf("bb: %v\n", bb)
22    fmt.Println("\nChanging value of b\n\n")
23    b[0] = -1
24    fmt.Printf("b: %v\n", b)
25    fmt.Printf("bb: %v\n", bb)
26 }

```




```

PS E:\_Golang\go_programs\slices\backup>
PS E:\_Golang\go_programs\slices\backup> go run .\main.go
a: [1 2 3 4 5]
aa: [1 2 3 4 5]

Changing value of a

a: [100 2 3 4 5]
aa: [1 2 3 4 5]

Another way to copy the slice

b: [100 200 300 400 500]
bb: [100 200 300 400 500]

Changing value of b

b: [-1 200 300 400 500]
bb: [100 200 300 400 500]
PS E:\_Golang\go_programs\slices\backup>

```

Options

- make() and copy
- Append to nil slice (this is best)

At the end both the option tries to create the new backing array for the second slice to backup the original data.



Good job!

General Formula: `'[low:high:max]` => `length = high - max` and `capacity = max - low`. `lyric[1:3]` is `["me" "my"]`. `lyric[1:3:5]` is `["me" "my" "silver" "lining"]`. So, `[1:3]` is the returned slice, length: 2. `[1:3:5]` limits the capacity to four because after the 1st element there are only four more elements.

Question 1:

What are the length and capacity of the 'part' slice?

```
1 | lyric := []string{"show", "me", "my", "silver", "lining"}  
2 | part  := lyric[1:3:5]
```

Length: 1 - Capacity: 5

Length: 1 - Capacity: 3

Length: 3 - Capacity: 5

Length: 2 - Capacity: 4



Good job!

`lyric[:2:2]` = `["show" "me"]`. After the append the part becomes: `["show" "me" "right" "place"]` — so it allocates a new backing array. `lyric` stays the same: `["show" "me" "my" "silver" "lining"]`.

Question 2:

What are the lengths and capacities of the slices below?

```
1 | lyric := []string{"show", "me", "my", "silver", "lining"}  
2 | part  := lyric[:2:2]  
3 | part  = append(part, "right", "place")
```

lyric's len: 5, cap: 5 — part's len: 5, cap: 5

lyric's len: 3, cap: 1 — part's len: 2, cap: 3

lyric's len: 5, cap: 5 — part's len: 4, cap: 4

lyric's len: 3, cap: 1 — part's len: 2, cap: 3



Good job!

Yes! You can use the make function to preallocate a backing array for a slice upfront.

Question 3:

When you might want to use the make function?

To preallocate a backing array for a slice with a definite length

To create a slice faster

To use less memory



Good job!

`make([]string, 2)` creates a slice with len: 2 and cap: 2, and it sets all the elements to their zero-values.

`append()` appends after the length of the slice (after the first two elements). That's why the first two elements are zero-valued strings but the last two elements are the newly appended elements.

Question 4:

What does the program print?

```
1 tasks := make([]string, 2)
2 tasks = append(tasks, "hello", "world")
3
4 fmt.Printf("%q\n", tasks)
```



["" "" "hello" "world"]



["hello" "world"]



["hello" "world" "" """]



Good job!

`copy` copies a newly created slice with four elements (`make([]string, 4)`) onto `lyric` slice. They both have 4 elements, so the `copy` copies 4 elements. Remember: `make()` initializes a slice with zero-values of its element type. Here, this operation clears all the slice elements to their zero-values.

Question 5:

What does the program print?

```
1 | lyric := []string{"le", "vent", "nous", "portera"}  
2 | n := copy(lyric, make([]string, 4))  
3 |  
4 | fmt.Printf("%d %q\n", n, lyric)  
5 |  
6 | // -- USEFUL INFORMATION (but not required to solve the question) --  
7 | //     In the following `copy` operation, `make` won't allocate  
8 | //     a slice with a new backing array up to 32768 bytes  
9 | //     (one string value is 8 bytes on a 64-bit machine).  
10| //  
11| //     This is an optimization made by the Go compiler.
```

4 ["le" "vent" "le" "vent"]

4 ["le" "vent" "nous" "portera"]

4 ["" "" "" ""]



Map

```
package main

import (
    "fmt"
    "os"
)

func main() {

    key := os.Args[1]
    a := map[string]int{}
    a["zero"] = 0
    a["one"] = 1

    // how to check whether the key exists or not
    value, ok := a[key]
    if ok {
        fmt.Printf("The key %q exists\n", key)
        fmt.Printf("a[%q] : %v", key, value)
    } else {
        fmt.Printf("The key %q not exists\n", key)
        fmt.Printf("The zero value of a[%q] : %v", key, value)
    }
}
```



```
PS E:\_Golang\go_programs\maps\imp>
PS E:\_Golang\go_programs\maps\imp>
PS E:\_Golang\go_programs\maps\imp>
PS E:\_Golang\go_programs\maps\imp> go run .\main.go one
The key "one" exists
a["one"] : 1
PS E:\_Golang\go_programs\maps\imp>
PS E:\_Golang\go_programs\maps\imp>
PS E:\_Golang\go_programs\maps\imp> go run .\main.go no
The key "no" not exists
The zero value of a["no"] : 0
PS E:\_Golang\go_programs\maps\imp>
PS E:\_Golang\go_programs\maps\imp>
```

Comparing maps using == we get error

```
}
```

```
var a map[string]int
```

```
// cannot compare a == b (operator == not defined for
```

```
b : map[string]int) compiler(UndefinedOp)
```

```
View Problem No quick fixes available
```

```
if a == b {
```

```
}
```

```
func main() {  
  
    // key := "something"  
    a := map[string]int{}  
    a["zero"] = 0  
    a["one"] = 1  
  
    // how to check whether the key exists or not  
    // value, ok := a[key]  
    // if ok {  
    //     fmt.Printf("The key %q exists\n", key)  
    //     fmt.Printf("a[%q] : %v\n", key, value)  
    // } else {  
    //     fmt.Printf("The key %q not exists\n", key)  
    //     fmt.Printf("The zero value of a[%q] : %v\n", key, value)  
    // }  
  
    // to compare 2 maps  
    b := map[string]int{"one": 1, "zero": 0}  
  
    aStr := fmt.Sprintf("%v", a) ← blue arrow pointing here  
    bStr := fmt.Sprintf("%v", b)  
    if aStr == bStr {  
        fmt.Println("both are equal")  
        fmt.Println(a)  
        fmt.Println(b)  
    } else {  
        fmt.Println("both are not equal")  
        fmt.Println(a)  
        fmt.Println(b)  
    }  
}
```

```
PS E:\_Golang\go_programs\maps\imp>  
PS E:\_Golang\go_programs\maps\imp>  
PS E:\_Golang\go_programs\maps\imp> go run .\main.go  
both are not equal  
map[one:1 zero:0]  
map[one:1 zero:1]  
PS E:\_Golang\go_programs\maps\imp>  
PS E:\_Golang\go_programs\maps\imp> go run .\main.go  
both are equal  
map[one:1 zero:0]  
map[one:1 zero:0]  
PS E:\_Golang\go_programs\maps\imp>  
PS E:\_Golang\go_programs\maps\imp>
```



Good job!

That's right. Maps work in $O(1)$ in average for fast-lookup.

Question 1:

Why are maps used for?

For example, here is an inefficient program that uses a loop to find an element among millions of elements.

```
1 | millions := []int{/* millions of elements */}
2 | for _, v := range millions {
3 |     if v == userQuery {
4 |         // do something
5 |     }
6 | }
```



Maps allow fast-lookup for map keys in $O(1)$ time



Maps allow fast-lookup for map keys in $O(n)$ time



Maps allow fast-traversal on map keys in $O(1)$ time



Good job!

Right! Looping over map keys happen in $O(n)$ time. So, maps are the worst data structures for key traversing.

Question 2:

When should you not use a map?

To find an element through a key

To loop over the map keys

To add structured data to your program



Good job!

Slices, maps, and function values are not comparable. So, they cannot be map keys.

Question 3:

Which type below cannot be a map key?

`map[string]int`

`[]string`

`[]int`

`[]bool`

All of them



Good job!

The map contains other maps. The element type of a map can be of any type.

Question 4:

Which are the key and element types of the map below?

`map[string]map[int]bool`

Key: string Element: bool

Key: string Element: int

Key: string Element: map[int]

Key: string Element: map[int]bool



Good job!

That's right. Maps are complex data structures. However, each map value is only a pointer to a map header (which is a more complex data structure).

Question 5:

What is a map value behind the scenes?

A map header

A pointer to a map header

Tiny data structure with 3 fields: Pointer, Length and Capacity

We can sort the map only by making all the keys to the slice and then sorting the key slice and then looping over the key slice and then print the map values.

So this can only be used to print in the sorted order. It doesn't actually sort the map. Also map is an unorder data type.

```

func main() {
    population := map[string]int{
        "Australia": 24982688,
        "Qatar": 2781677,
        "Wales": 3139000,
        "Burundi": 11175378,
        "Guinea": 12414318,
        "Niger": 22442948,
        "Brazil": 209469333,
        "Malta": 484630,
        "Peru": 31989256,
        "Yemen": 28498687,
        "Ireland": 4867309,
        "Kenya": 51393010,
        "Montserrat": 5900,
        "Cuba": 11338138,
        "Nicaragua": 6465513,
        "Jordan": 9956011,
        "Gabon": 2119275,
    }

    keys := make([]string, 0, len(population))
    for k := range population {
        keys = append(keys, k)
    }
    sort.Strings(keys)

    for _, k := range keys {
        fmt.Println(k, population[k])
    }
}

```



```

PS E:\_Golang\go_programs\maps\sorting_map> go run .\sort_by_keys.go
Australia 24982688
Brazil 209469333
Burundi 11175378
Cuba 11338138
Gabon 2119275
Guinea 12414318
Ireland 4867309
Jordan 9956011
Kenya 51393010
Malta 484630
Montserrat 5900
Nicaragua 6465513
Niger 22442948
Peru 31989256
Qatar 2781677
Wales 3139000
Yemen 28498687
PS E:\_Golang\go_programs\maps\sorting_map> go run .\sort_by_keys.go
Australia 24982688
Brazil 209469333
Burundi 11175378
Cuba 11338138
Gabon 2119275
Guinea 12414318
Ireland 4867309
Jordan 9956011
Kenya 51393010
Malta 484630
Montserrat 5900
Nicaragua 6465513
Niger 22442948
Peru 31989256
Qatar 2781677
Wales 3139000
Yemen 28498687
PS E:\_Golang\go_programs\maps\sorting_map>
PS E:\_Golang\go_programs\maps\sorting_map>

```

**Good job!**

The Text() method only returns the last scanned token. A token can be a line or a word and so on.

Question 2:

What does the program print?

```
1 | in := bufio.Scanner(os.Stdin)
2 |
3 | in.Scan() // user enters: "hi!"
4 | in.Scan() // user enters: "how are you?"
5 |
6 | fmt.Println(in.Text())
```

hi and how are you?

hi

how are you?

Nothing



Good job!

That's right. bufio has a few splitters like ScanWords such as ScanLines (the default), ScanRunes, and so on.

This was discussed in Lecture 156: [Use maps as sets](#) >

Question 4:

How can you configure bufio.Scanner to only scan for the words?

```
1 | in := bufio.Scanner(os.Stdin)
2 | // ...
```

in = bufio.NewReader(in, bufio.ScanWords)

in.Split(bufio.ScanWords)

in.ScanWords()



Good job!

"Must" prefix is a convention. If a function or method may panic (= crash a program), then it's usually being prefixed with a "must" prefix.

Question 5:

The following function uses the "Must" prefix, why?

```
regexp.MustCompile("...")
```

It's only being used for readability purposes

It's a guarantee that the function will work, no matter what

The function may crash your program

JSON and Structs

Json package only encodes the exported fields

Becoz only the exported fields can be seen by other packages. So, that means our struct fields should be exported inorder for the json package to see it and encode it.



Good job!

Right! Go initializes a struct's fields to zero-values depending on their type.

Question 1:

What is the zero-value of the following struct value?

```
1 | var movie struct {  
2 |     title, genre string  
3 |     rating      float64  
4 |     released    bool  
5 | }
```



{}



{title: "", genre: "", rating: 0, released: false}



{title: "", genre: "", rating: 0, released: true}



{"title, genre": "", rating: 0, released: false}

```

package main

import (
    "encoding/json"
    "fmt"
)

type permissions map[string]bool // #3

type user struct {
    Name      string      `json:"username"`
    Password  string      `json:"-"` // this will make this field to be skipped in encoding
    Permissions permissions `json:"perms,omitempty"`
    // the omit empty option will make this field not to encode when this field is empty
}

func main() {
    users := []user{ // #2
        {"inanc", "1234", nil}, // here permission field is nil so it wont be encode due to theomitempty option
        {"god", "42", permissions{"admin": true}},
        {"devil", "66", permissions{"write": true}},
    }

    out, err := json.MarshalIndent(users, "", "\t")
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(string(out))
}

```

This are field tags

```

PS E:\_Golang\go_programs\structs\extras> go run .\main.go
[{"username": "inanc"}, {"username": "god", "perms": {"admin": true}}, {"username": "devil", "perms": {"write": true}}]

```

https://www.youtube.com/watch?v=_SCRvMunkdA

Struct fields explanation



Good job!

Right! Field names and types are part of a struct's type.

Question 2:

What is the type of the following struct?

```
1 | avengers := struct {
2 |     title, genre string
3 |     rating      float64
4 |     released    bool
5 |
6 |     "avengers: end game", "sci-fi", 8.9, true,
7 |
8 |
9 |     fmt.Printf("%T\n", avengers)
```



struct{}



struct{ string; string; float64; bool }



struct{ title string; genre string; rating float64; released bool }



{title: "avengers: end game"; genre: "sci-fi"; rating: 8.9; released: true}



Good job!

When creating a struct value, it doesn't matter whether you use the field names or not. So, they are equal.

Question 3:

Are the following struct values equal?

```
1 | type movie struct {
2 |     title, genre string
3 |     rating        float64
4 |     released      bool
5 |
6 |
7 |     avengers := movie{"avengers: end game", "sci-fi", 8.9, true}
8 |     clone       := movie{
9 |         title: "avengers: end game", genre: "sci-fi",
10 |         rating: 8.9, released: true,
11 |     }
```

The only limitation when we can't even == the struct when the struct fields have map or slice data types



There is a syntax error

Only same struct types can be compared



Yes



No



Good job!

Right! Types with different names cannot be compared. However, you can convert one of them to the other because they have the same set of fields. `movie{} == movie(performance{})` is ok, or vice versa.

This was discussed in Lecture 162: [When can you compare struct values?](#) >

Question 4:

Do the movie and performance struct types have the same types?

```
1 | type item      struct { title string }
2 | type movie     struct { item }
3 | type performance struct { item }
```

Yes: They have the same set of fields

No : They have different type names

No : An embedded field cannot be compared



Good job!

Right! It's just a string value. It's only meaningful when other code reads it. For example, the json package can read it and encode/decode depending on the field tag's value.

This was discussed in Lecture 165: [Encode values to JSON](#) >

Question 5:

Which answer below is correct about a field tag?

- It needs to be typed according to some rules
- You can change it to a different value in runtime
- It's just a string value, and it doesn't have a meaning on its own

Functions



Good job!

Question 4:

How and why does the following return statement work?

```
1 | func spread(samples int, P int) (estimated float64) {  
2 |     for i := 0; i < P; i++ {  
3 |         estimated += estimate(i, P)  
4 |     }  
5 |     return  
6 | }
```



estimated is a named result value. So the naked return returns **estimated** automatically.



return statement is not necessary there. Go automatically returns **estimated**.



Result values cannot have a name. This code is incorrect.



Good job!

Map values are pointers. So, `incrAll` can update the map value.

Question 5:

Does the following code work? If so, why?

IT SHOULD PRINT: `map[1:11 10:3]`

```
1 func main() {  
2     stats := map[int]int{1: 10, 10: 2}  
3     incrAll(stats)  
4     fmt.Println(stats)  
5 }  
6  
7 func incrAll(stats map[int]int) {  
8     for k := range stats {  
9         stats[k]++  
10    }  
11 }
```



No, it doesn't work: Go is a pass by value language. `incrAll` cannot update the map value.



Yes, it works: `incrAll` can update the map value.

Pointers

```
package main

import "fmt"

func main() {
    var a int = 100

    p := &a
    fmt.Printf("p: %v\n", p)
    fmt.Printf("*p: %v\n", *p)

    fmt.Printf("a: %v\n", a)
    aa := *p
    fmt.Printf("aa: %v\n", aa)
    aa = 1111
    fmt.Printf("aa: %v\n", aa)

    fmt.Printf("*p: %v\n", *p)
    fmt.Printf("a: %v\n", a)
    *p = 999
    fmt.Printf("*p: %v\n", *p)
    fmt.Printf("a: %v\n", a)
}
```



```
PS E:\_Golang\go_programs\pointers>
PS E:\_Golang\go_programs\pointers>
PS E:\_Golang\go_programs\pointers>
PS E:\_Golang\go_programs\pointers>
PS E:\_Golang\go_programs\pointers> go run .\main.go
p: 0xc0000aa058
*p: 100
a: 100
aa: 100
aa: 1111
*p: 100
a: 100
*p: 999
a: 999
PS E:\_Golang\go_programs\pointers>
```

```

type house struct {
    name string
    rooms int
}

func structs() {
    myHouse := house{name: "My House", rooms: 5}

    addRoom(myHouse)

    // fmt.Printf("%+v\n", myHouse)
    fmt.Printf("structs() : %p %+v\n", &myHouse, myHouse)

    addRoomPtr(&myHouse)
    fmt.Printf("structs() : %p %+v\n", &myHouse, myHouse)
}

func addRoomPtr(h *house) {
    h.rooms++ // same: (*h).rooms++
    fmt.Printf("addRoomPtr() : %p %+v\n", h, h)
    fmt.Printf("&h.name : %p\n", &h.name)
    fmt.Printf("&h.rooms : %p\n", &h.rooms)
}

func addRoom(h house) {
    h.rooms++
    fmt.Printf("addRoom() : %p %+v\n", &h, h)
}

```

Pointer for structs

Structs also needs the pointer to change its value in the function.

Since structs are bare type like int and string

```

type house struct {
    name string
    rooms int
}

func main() {
    structs()
}

func structs() {
    myHouse := house{name: "My House", rooms: 5}

    addRoom(myHouse)

    // fmt.Printf("%+v\n", myHouse)
    fmt.Printf("structs()      : %p %+v\n", &myHouse, myHouse)

    addRoomPtr(&myHouse)
    fmt.Printf("structs()      : %p %+v\n", &myHouse, myHouse)
}

func addRoomPtr(h *house) {
    h.rooms++ // same: (*h).rooms++
    fmt.Printf("addRoomPtr()   : %p %+v\n", h, h)
    fmt.Printf("&h.name       : %p\n", &h.name)
    fmt.Printf("&h.rooms      : %p\n", &h.rooms)
}

func addRoom(h house) {
    h.rooms++
    fmt.Printf("addRoom()     : %p %+v\n", &h, h)
}

```

The terminal window shows the execution of a Go program named `main.go`. The output displays memory addresses and corresponding values for various pointers and their dereferenced values.

```

PS E:\_Golang\go_programs\pointers\struct_pointers>
PS E:\_Golang\go_programs\pointers\struct_pointers>
PS E:\_Golang\go_programs\pointers\struct_pointers> go run .\main.go
addRoom()      : 0xc000004090 {name:My House rooms:6}
structs()      : 0xc000004078 {name:My House rooms:5}
addRoomPtr()   : 0xc000004078 &{name:My House rooms:6}
&h.name        : 0xc000004078
&h.rooms       : 0xc000004088
structs()      : 0xc000004078 {name:My House rooms:6}
PS E:\_Golang\go_programs\pointers\struct_pointers>

```



Good job!

A pointer is just another value that can contain a memory address of a value.

This was discussed in Lecture 175: [What is a pointer? >](#)

Question 1:

What is a pointer ?

A variable that contains an hexadecimal value

A variable that contains a memory address

A value that can contain a memory address of a value

A value that points to a function



Good job!

a and b are nil at the beginning, so they are equal. However, after that, they get two different memory addresses from the composite literals, so their addresses are not equal but their values (that are pointed by the pointers) are equal.

Question 4:

What is the result of the following code?

```
1 | type computer struct {
2 |     brand string
3 | }
4 |
5 | var a, b *computer
6 | fmt.Println(a == b)
7 |
8 | a = &computer{"Apple"}
9 | b = &computer{"Apple"}
10| fmt.Println(" ", a == b, " ", *a == *b)
```



1 | false false false



1 | true true true



1 | true false true



Good job!

Every time a func runs, it creates new variables for its input params and named result values (if any). There two pointer variables: `a` and `b`. Then there happen two more pointer variables: `'c'`. It's because: `change()` is called twice.

This was discussed in Lecture 176: [Learn the pointer mechanics >](#)

Question 5:

How many variables are there in the following code?

```
1 type computer struct {
2     brand string
3 }
4
5 func main() {
6     a := &computer{"Apple"}
7     b := a
8     change(b)
9     change(b)
10 }
11
12 func change(c *computer) {
13     c.brand = "Indie"
14     c = nil
15 }
```



Methods

```
package main

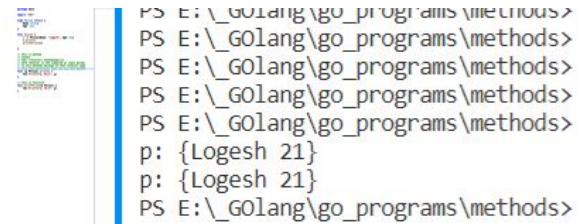
import "fmt"

type Person struct {
    Name string
    Age int
}

func main() {
    l := Person{Name: "Logesh", Age: 21}
    l.print()
    printPerson(l)
}
```

```
// this is method
// syntax
// func (receiver) methodname(){}
// here receiver is the compared as input params
// so this methods belongs to that receiver type
// only that particular type can use this method
func (p Person) print() {
    fmt.Printf("p: %v\n", p)
}
```

```
// this is function
func printPerson(p Person) {
    fmt.Printf("p: %v\n", p)
}
```



```
PS E:\_Golang\go_programs\methods>
PS E:\_Golang\go_programs\methods>
PS E:\_Golang\go_programs\methods>
PS E:\_Golang\go_programs\methods>
PS E:\_Golang\go_programs\methods> go run .\person.go
p: {Logesh 21}
p: {Logesh 21}
PS E:\_Golang\go_programs\methods>
```

Even this is the method of Person type it can't change the value

receiver value, we can change it by passing the pointer way

p.Name = "Logesh Vel"
This doesn't changes caller variable

```
type Person struct {
    Name string
    Age int
}

func main() {
    l := Person{Name: "Logesh", Age: 21}
    l.print()
    l.up()
    l.print()
}
```

```
func (p Person) print() {
    fmt.Printf("%+v\n", p)
}
```

```
// Pointer receiver to modify the values
func (p *Person) up() {
    p.Name = strings.ToUpper(p.Name)
}
```



```
PS E:\_Golang\go_programs\methods>
PS E:\_Golang\go_programs\methods>
PS E:\_Golang\go_programs\methods> go run .\pointer_receiver.go
{Name:Logesh Age:21}
{Name:LOGESH Age:21}
PS E:\_Golang\go_programs\methods>
```

For the pointer receiver method we don't need to pass the & the Go will automatically pass it.



```
1 package main
2
3 import "fmt"
4
5 type money float64
6
7 func main() {
8     var balance money
9     balance = 19956789799.989
10    balance.show()
11 }
12
13 // here we have created the method for money type
14 // which has float64 as the underlying type
15 func (m money) show() {
16     fmt.Printf("$ %.2f", m)
17 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS E:\_Golang\go_programs\methods\methods_to_non_struct_types> go run .\main.go
$ 19956789799.99
PS E:\_Golang\go_programs\methods\methods_to_non_struct_types> █
```

`strings.Builder`

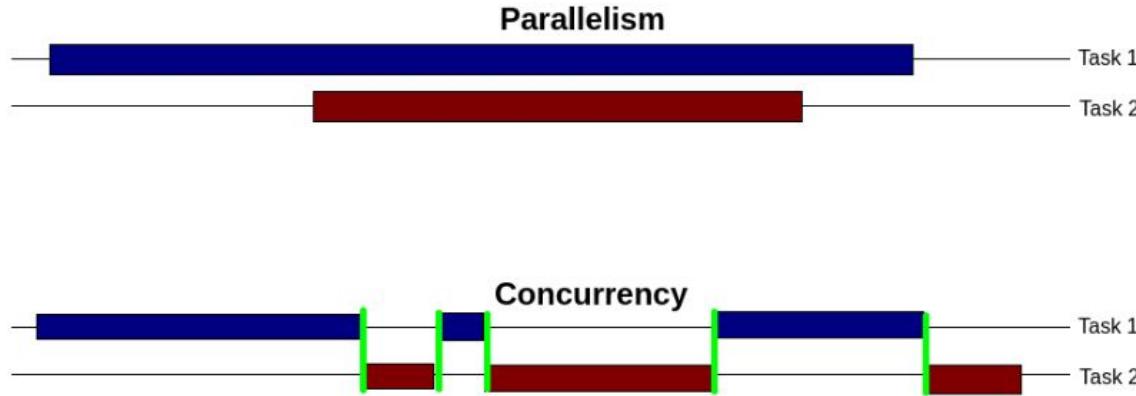
```
outerloop:  
    for _, ele1 := range s1 {  
        for _, ele2 := range s2 {  
            for _, ele3 := range s3 {  
                // ...  
                continue outerloop  
            }  
        }  
    }  
}
```

Goroutines are not threads

You might have heard that goroutines are not the same as threads, but do you know what the differences are?

First thing first, *what is concurrency*? Concurrency is being able to run other tasks even if a task is not completed. For example, while a concurrent app is waiting for input from the terminal, it can do another calculation. For Go, we use goroutines for concurrency. A goroutine waits for input while another goroutine does the calculations.

Note that concurrency is not *parallelism*. Parallelism means the ability to run more than one task at the same time. It is possible if you have more than one CPU. In concurrency, the apps switch between goroutines (or threads if you use a different language).



Visual explanation of parallelism and concurrency

You can see green lines in the image. These lines are *context switches*, which means the moment when the app switch to run the other task.

For concurrency, we use threads in most programming languages. Go has a different mechanism instead. This mechanism is goroutines. Basically, both threads and goroutines have the same purpose; concurrency. However, goroutines are better in some points. Let's check what they are.

Speed

While threads are *managed by the kernel*, goroutines are *managed by go runtime*. Since we don't need kernel system calls for goroutines as much as threads, goroutines are faster.

Memory Size

Threads have fixed *stack size*. When your app is run, the operating system allocates a memory block for the stack. In order not to face problems about insufficient memory, the size of this block cannot be small.

On the other hand, the stack size of goroutines can grow or shrink. Therefore, go allocate a small memory block at the start. Whenever your app requires more, go runtime can allocate it. Hence, goroutines use way lesser memory than threads.

