Date No

Arrays and Slices.

An array is a collection of elements that belong to the same type. mixing different types is not allowed.

Declaration.

There are different ways to declare arrays.

```
func main() {
  var a [3] int  // int array with length of 3
  fmt. Println (a)   all the elements in array automati-
}  →[0 0 0]        cally assigned to zero
```

index of an arrg starts from 0 and ends at length-1

```
func main() {
  var arr [3] int  // int array with 3 length
  arr [0] = 12  // index starts at 0
  arr [1] = 15
  arr [2] = 11
  fmt. Println (arr)
} → [12  15  11 ]
```

Array short hand declaration

```
func main() {   var arr = [3] {2,4,6}
  arr := [3] int { 12, 78, 15 } // Shorthand declaration.
  fmt. Println (arr)
} → [12  78  15 ]
```

* it is not necessary that all elements in array have

Date No

Can even ignore the length of the array in the declaration and replace it with ... compiler find the length.

```
func main () {
  arr := [...] {2, 5, 7, 1}
  fmt.Println(arr)
} -> [2 5 7 1]
```

The size of the array is a part of the type. Hence [5]int and [25]int are distinct type. Because of this array cannot be resized.

```
func main () {
  a := [3]int {2, 5, 9}
  var b [5] int
  b = a
}   // not Possible since [3]int and [5]int are
      distinct types
```

Arrays are value type.

Array is Go are value type not reference types. This means, when they are assigned to a new variable, a copy of the original array is assigned to new variable. If changes are made to new variable, it will not be reflected in the original array.

Date No

```
func main() {                          [0]    [1]    [2]
  arr_old := [...]String {"Canada", "USA", "UAE"}
  arr_new := arr_old
  arr_new[0] = "Thai"
  fmt.Println("Old Array ", arr_old)
  fmt.Println("New Array", arr_new)
}
```

→ Old Array [canada USA UAE]
New Array [Thai USA UAE]

When array are passing to function as parameters, they are passed by value and the original is unchanged.

```
func changeLocal (num [5] int) {
    num [5] = 55
    fmt.Print ("inside function", num)
}

func main ()                                    (5)
    num := [...] int { 2, 3, 4, 8, 8 }
    fmt.Println ("Before Passing to func ", num)
    changeLocal (num)
    fmt.Println ("After Passing to func", num)
}
```

→ Before Passing to func [2 3 4 8 8]
inside function [55 3 4 8 8]
After Passing to func [2 3 4 8 8]

Date No

Length of an Array.

Length of the array is found by passing the array as parameter to the len function.

```
func main() {
  arr := [...] int {5, 10, 15, 20}
  fmt.Println("Length is", len(arr))
} -> Length is 4
```

iterating arrays using range

for loop can be used to array iteration

```
func main() {
  arr := [...] int {10, 20, 30, 40}
  for i := 0; i < len(arr); i++ {
    fmt.Printf("%d th element of array is %d", i, arr[i])
}
-> 0th element of arry is 10
   1th element of array is 20
        :
```

Go provides a better and concise way to iterate over an array by using the range form of the for loop. range returns both the index and the value at that index.

```
func main() {
  arr := [...] int { 2, 4, 3, 9, 0}
  for index, value := range arr {
    fmt.Printf("\n index %d is %d", index, value)
```

Date No

in case you want only the value and want to ignore the index, use blank identifier as follows —

put blank identifier in index

```
for _, val := range arr
```

Multidimensional Arrays.

```
func printarray (a[3][2] string) {
    for _, v1 := range a {
        for _, v2 := range v1 {
            fmt.Printf("%s", v2)
        }
        fmt.Println()
    }
}
func main() {
    a := [3][2] String {
        {"lion", "tiger"},
        {"cat", "dog"},
        {"cow", "rat"},
    }
    printarray(a)
```

rows → [3] columns → [2]

[0][0]	lion	tiger	[0][1]
[1][0]	cat	dog	[1][1]
[2][0]	cow	rat	[2][1]

Declaration & initialization together.

```
var b[3][2] String ← Declaration
b[0][0] = "apple"
b[0][1] = "samsang"
b[1][0] = "microsoft"
b[1][1] = "google"
b[2][0] = "nokia"
```

initialization

→ lion tiger
cat dog
cow rat

apple samung

Date No

Slices

A Slice is convenient, flexible and powerful wrapper on top of array. Slices do not own any data on their own. They are just references to existing array.

Creating a Slice.

```
func main() {
    a := [5]int {76, 77, 78, 79, 80}
    var b []int = a [1:4]
            // create a slice from a[1] to a[3]
    fmt.Println(b)
}  -->[77 78 79]
```

The syntax a [start : end] create a slice from array a starting from index start to index end-1 (no-1)

In the above program a [1:4] creates a slice representation of the array a starting from index 1 through 3.

Another way to create Slice

```
func main() {
    c := []int {3, 4, 5} // create an array and
    fmt.Print(c)          returns a slice reference.
}
```