Q 1. How are OO principles implemented in Golang? Explain.

Ans. Go is a hybrid of old and new programming concepts. In terms of object-orientedness, Go has no classes, objects, exceptions and templates. The most notable feature of OOP that Go doesn't have is inheritance. Go however has garbage collection and concurrency (a very important concept used in Go).

Struct:

To balance absence of classes, Go has structs with methods (Remember, methods is not same as functions in Go). Structs hold only state and no behavior. For example, here is a Student struct. It has a Name field and a RollNo field of type int64.

```
type Creature struct {

  Name string

  RollNo int64

}
```

Here, the struct acts as a class (not entirely) in Go.

Methods:

Methods are basically functions that work on a particular "type". Methods are functions with special receiver arguments, bound to a particular "type". We have something called "receiver" that's tied to these methods.

Functions Syntax:

```
func FunctionName(Parameters...) ReturnTypes...

For eg:
func add(x int, y int) int
```

Methods Syntax:

```
func (t ReceiverType...)FunctionName(Parameters...) ReturnTypes...

For eg:
func (t MyType) add(int x, int y) int
```

Embedding:

Anonymous types can be embedded inside each other. If a nameless struct is embedded inside another struct, the embedded struct provides it's state (and methods) available to the main struct directly.

Interfaces:

Interfaces are the hallmark of Go's object oriented support. Interfaces are types that declare methods. It has no implementation like other programming languages. If all methods of an interface are implemented by a "type", the interface gets "implemented" without using the keyword "implements" implicitly, since Go doesn't support inheritance of any kind. It's a convention to have interfaces names ending in "er".

Encapsulation:

Go encapsulates things at package level. It has a very absurd way of providing access protection. Names starting with lowercase are only visible to that package it is defined in. Whereas names beginning in uppercase are visible to code outside package. For eg:

```
type foo struct {
}

func (f foo) Foo1() {
    fmt.Println("Foo1() here")
}

func (f foo) Foo2() {
    fmt.Println("Foo2() here")
}

func (f foo) Foo3() {
    fmt.Println("Foo3() here")
}

func NewFoo() Fooer {
    return &Foo{}
}
```

You can access NewFoo() from another package and then access Fooer interface implemented by internal foo type.

Q 2. "Don't communicate by sharing memory, share memory by communicating." Comment on the above with suitable example/s.

Ans. "Don't communicate by sharing memory, share memory by communicating." is a famous quote by Rob Pike on Golang.

Let's break it apart:

## Don't communicate by sharing memory.

Concurrency is a (some might argue for the) key feature of Go. Concurrent processing allows multiple computing streams to advance independent of each other. The language feature that Go

uses to allow for concurrent processing is called goroutine. Frequently, there is a need for coordination between these goroutines. This coordination is the communication we are concerned with.

In some languages, the way to communicate is by using some resource that the concurrent steams can all access. This resource could be a simple variable or a complex structure. This works. There are even features provided by the operating system to allow shared objects between independent processes. However, there are problems here. If two streams try to modify the shared object at the same time, the final state depends on the order in when things happen. And this order is not guaranteed. This is a **race condition**. The solution to a race condition is controlling entry in to the parts of the code that use the shared resource, but this adds overhead and complexity. With Go, there are alternatives.

### Share memory by communicating.

There is still the frequent requirement to coordinate. Go provides another language feature, the *channel*. Channels allow communication between concurrent streams, the **goroutines**. When creating the goroutines, channels are created to allow for the communication that is needed. The source of the information writes to the channel when the information is ready. The receiver is waiting for the information to be delivered by the channel. Once written, it will read it and do whatever processing is needed. A power characteristic of channels is that there can be multiple readers and/or writers. The channel mechanism ensures that each write and each read is atomic. There are many details related to channels that are quite powerful, but the Go documentation is the place to read about that.

A primary goal of the Go language is to make the code as simple and clean as possible. By using goroutines and channels, concurrent processes can be expressed with less visible complexity than if one needed to provide shared memory and synchronization explicitly.