Question 1. Polymorphism in Go **[4 points]**

Go is not an object-oriented language but it is still possible to implement polymorphism and to define methods associated to types.

You will find with this assignment, a set of Java classes representing a hierarchy of classes. You are asked to translate this program into Go. Your Go program must also implement polymorphism among all types of employees. It must also produce the same output.

You must also adhere to the following guidelines:

- *All employees created in main function must be in an array of StaffMember type.*
- *Java References should be replaced by Go pointers*
- *You must use the constructor pattern in Go that consists in defining a NewThing(parameters…) function instead of the new Thing(parameters…) call to constructor used in Java*
- *To replace Java inheritance, you must use type embedding*
- *Polymorphism must be implemented using Go interfaces*

Question 2. The Go Function **[4 points]**

A series of numbers in a slice must be processed and the results displayed. The processing is performed by calling the `process` *function. In order to accelerate the processing, it is decided to create a separate thread for each number to be processed. Here is the first version of this program :*

```
package main
 import "fmt"
import "sync"
import "time"
 func process(v int) int
{

       time.Sleep(1500*time.Millisecond) // simulate compute time
return 2*v
}

func main() {

        var wg sync.WaitGroup

        for _,value := range []int{9,35,27,56,88,80} {
           wg.Add(1)             go func()
{                defer wg.Done()
      fmt.Println(process(value))
           }()
        }
        wg.Wait()
}
```

Unfortunately, the results obtained are not the ones expected…

```
160
160
160
160
160
160
```

Correct the problem by making few modifications to the program (not more than three two or four lines). The result should be as below (the display order is not important).

```
176 70
112
18
54 160
```

Question 3. Concurrent threads [4 points]

Prime numbers play an important role in cryptography. The following program generates 10000 random prime numbers and inserts them into a slice. This program would be much more efficient if prime number generation would be done in multiple concurrent threads. Modify this program such that 5 go threads concurrently generate prime numbers and write the results into a channel. The slice is then filled by reading from this channel. Make sure all threads are terminated before ending the program. Check the impact of this change on the execution time.

```
 package main
import "fmt"
import "math"
import "time"
import "math/rand"

// returns true if number is prime func
isPrime(v int64) bool {

   sq:= int64(math.Sqrt(float64(v)))+1
var i int64    for i=2; i<sq; i++ {
       if v%i == 0 {
return false
      }
}
   return true
}
```

```
// get a random prime number between 1 and maxP
func getPrime(maxP int64) int64 {

   var i int64
   for i=0; i<maxP; i++ {

     n:= rand.Int63n(maxP)
if isPrime(n) {
return n
     }
   }
   return 1 // just in case
}
func main() {

   var primes []int64 // slice of prime numbers
   const maxPrime int64 = 10000000 // max value for primes
    start:= time.Now()
for i:=0; i<10000; i++ {
      p:= getPrime(maxPrime)  // add a new prime
primes= append(primes,p)
   }
   end:= time.Now()

   fmt.Println("End of program.",end.Sub(start)) }
```