The defer Pattern

This lesson briefly discusses the uses of the defer pattern in Go language.

WE'LL COVER THE FOLLOWING

Using `defer` ensures that all resources are properly closed or given back to the *pool* when the resources are not needed anymore. Secondly, it is paramount in Go to recover from panicking.

Closing a file stream

```
// open a file f
defer f.Close()
```

Unlocking a locked resource (a mutex)

```
mu.Lock()
defer mu.Unlock()
```

Closing a channel (if necessary)

```
ch := make(chan float64)
defer close(ch)
```

or with 2 channels:

```
answerα, answerβ := make(chan int), make(chan int)
defer func() { close(answerα); close(answerβ) }()
```

Recovering from a panic

```
defer func() {
  if err := recover(); err != nil {
    log.Printf("run time panic: %v", err)
}()
```

Stopping a ticker

```
tick1 := time.NewTicker(updateInterval)
defer tick1.Stop()
```

Release of a process

```
p, err := os.StartProcess(..., ..., ...)

defer p.Release()
```

Stopping CPUprofiling and flushing the info

```
pprof.StartCPUProfile(f)
defer pprof.StopCPUProfile()
```

A goroutine signaling a WaitGroup

```
func HeavyFunction1(wg *sync.WaitGroup) {
defer wg.Done()
// Do a lot of stuff
}
```

It can also be used when not forgetting to print a footer in a report.

That's it about the `defer` pattern, and its uses. The next lesson brings you the highlights of playing around operators without violating the visibility rule.