Using recover with goroutines

This lesson covers the advantages of recovering when using goroutines and how it doesn't affect the rest of the program.

WE'LL COVER THE FOLLOWING

Recovering with goroutines

One application of `recover` is to shut down a failing goroutine inside a server without killing the other executing goroutines.

```
func server(workChan <-chan *Work) {
  for work := range workChan {
    go safelyDo(work) // start the goroutine for that work
  }
}

func safelyDo(work *Work) {
  defer func() {
   if err := recover(); err != nil {
     log.Printf("work failed with %s in %v:", err, work)
     // the failing goroutine is stopped
   }
 }()
do(work)
}
```

In the code snippet above, if `do(work)` panics, the error will be logged and the goroutine will exit cleanly without disturbing the other goroutines.

Because `recover` always returns *nil* unless called directly from a deferred function, the deferred code can call the library routines that themselves use `panic` and `recover` without failing. As an example, the deferred function in `safelyDo()` might call a logging function before calling recover, and that

`safelyDo()` might call a logging function before calling recover, and that logging code would run unaffected by the panicking state. With our recovery pattern in place, the do function (and anything it calls) can get out of any bad situation cleanly by calling `panic`. However, the recovery has to take place inside the panicking goroutine. It cannot be recovered by a different goroutine.

Now that you are familiar with recovering from panicking goroutines, in the next lesson, you'll learn how to choose the best pattern and the method of communication between goroutines, depending on the scenarios.