Debugging with Closures

This lesson discusses how debugging can be made easy with Go using closures.

WE'LL COVER THE FOLLOWING

- Debugging using `runtime`
- Debugging using `log`

When analyzing and debugging complex programs with myriads of functions in different code-files calling one another, it can often be useful to know which file is executing at certain points in the program and the line number of it. This can be done by using special functions from the packages `runtime` or `log`.

Debugging using `runtime`

In `runtime`, the function `Caller()` provides this information, so the closure `where()` could be defined, and then be invoked wherever it is needed:

```
package main
import "fmt"
import "runtime"
import "log"

func main() {
    where := func() {   // debugging function
        _, file, line, _ := runtime.Caller(1)
        log.Printf("%s:%d", file, line)
    }
    where()
    // some code
    a := 2*5
    where()
    // some more code
    b := a+100
    fmt.Println("a: ",a)
    fmt.Println("b: ",b)
    where()
}
```

Debugging using runtime

Debugging using `log`

The same can be achieved by setting a *flag* in the `log` package:

```
log.SetFlags(log.Llongfile)
log.Print("")
```

or if you like the brevity of `where`:

```
package main
import "fmt"
import "log"

func main() {
    var where = log.Print

    where()
    // some code
    a := 2*5
    where()

    // some more code
    b := a+100
    fmt.Println("a: ",a)
    fmt.Println("b: ",b)
    where()
}
```

Debugging using log

That's how Go uses closures to make debugging less tricky and provides efficiency. In the next lesson, you'll learn how to *time* a function and optimize the program.