Defer and Tracing

This lesson discusses important concepts of deferring and tracing functions in Go.

WE'LL COVER THE FOLLOWING

- The `defer` keyword
- Tracing with `defer`
- Logging parameters with `defer`

The `defer` keyword

The `defer` keyword allows us to postpone the execution of a statement or a function until the end of the enclosing (calling) function. Defer executes something (a function or an expression) when the enclosing function returns. This happens after every return, even when an error occurs in the midst of executing the function, not only a return at the end of the function, but before the **}**. But why after every return? This happens because the return statement itself can be an expression that does something instead of only giving back 1 or more variables. The *defer* resembles the *finally-block* in OO-languages as Java and C#; in most cases, it also serves to free up allocated resources. However, keep in mind that a defer-call is function scoped, while a finally-call is block scoped.

Run the following program to see how defer works.

```
package main
import "fmt"

func main() {
    Function1()
}

func Function1() {
    fmt.Printf("In Function1 at the top\n")
    defer Function2() // function deferred, will be executed after Function1 is done.
    fmt.Printf("In Function1 at the bottom!\n")
}

func Function2() {

    fmt.Printf("Function2: Deferred until the end of the calling function!")
}
```

Deferring

As you can see in the above code, in `main` we call `Function1` at **line 5**, so control will transfer to **line 8**. **Line 9** will be executed, and **In Function1 at the top** will be printed on the screen. At **line 10**, we defer calling `Function2()` with the statement: `defer Function2()`. Due to the `defer` keyword, control will not transfer to `Function2`. Then, **line 11** will be executed, and **In Function1 at the bottom!** will be printed. Now, all the statements other than `defer Function2()` are executed, and `Function1` will return. `Function2` will be executed, and **Function2: Deferred until the end of the calling function!** will be printed. Now, control will go back to `main` at **line 6**, and the program will terminate.

When many defer's are issued in the code, they are executed at the end of the function in the inverse order (like a stack or LIFO), which means the last defer is first executed, and so on. This is illustrated in the following code:

```
package main
import "fmt"

func main() {
    for i := 0; i < 5; i++ {
        defer fmt.Printf("%d ", i)
    }
}
```

Deferred Order of Execution

The `defer` allows us to guarantee that certain clean-up tasks are performed before we return from a function, for example:

- Closing a file stream
- Unlocking a locked resource (a mutex)

- Unlocking a locked resource (a mutex)
- Printing a footer in a report
- Closing a database connection

The `defer` can be helpful to keep the code cleaner and often shorter.

Tracing with `defer`

A primitive but sometimes effective way of tracing the execution of a program is printing a message when entering and leaving certain functions. This can be done with the following 2 functions:

```
func trace(s string) { fmt.Println("entering:", s) }
func untrace(s string) { fmt.Println("leaving:", s) }
```

The following program demonstrates how the `untrace` function is called with the `defer` keyword.

```
package main
import "fmt"

func trace(s string) {
        fmt.Println("entering:", s)
    } // entering func.
func untrace(s string) {
        fmt.Println("leaving:", s)
    } // leaving func.

func a() {
    trace("a")
    defer untrace("a") // untracing via defer
    fmt.Println("in a")
}

func b() {
    trace("b")
    defer untrace("b") // untracing via defer
    fmt.Println("in b")
    a()
}

func main() {
    b()
}
```

Tracing with defer

In the above code, the entry point is at **line 24**, where the `main` function is written. We are calling `b` at **line 25**. Now control will transfer to **line 17**. The `trace` function will be called at **line 18**, and **entering: b** will be printed on screen. Now, we are deferring the `untrace` function at **line 19**, which means that **line 19** will be executed after the `b` function returns. **Line 20** will gain control, and **in b** will be printed. At **line 21**, we are calling the function `a`. Control will transfer to **line 11**. The `trace` function will be called at **line 12**, and **entering: a** will be printed on screen. Now, we are deferring `untrace` function at **line 13**, which means that **line 13** will be executed after `a` returns. Now, function `a` will return, and the `untrace` function will gain control. The message **leaving: a** will be printed. Control will go back to `b`, and it will return and `untrace` will gain control. The message **leaving: b** will be printed. Finally, from there, control will go back to `main` at **line 26**, and the program will terminate.

Logging parameters with `defer`

This is another possible use of `defer`, which might come in handy while debugging. Look at the following program.

```
package main
import (
    "log"
    "io"
)

func func1(s string)(n int, err error) {

    defer func() {
        log.Printf("func1(%q) = %d, %v", s, n, err)
    }()

    return 7, io.EOF
}

func main() {
    func1("Go")
}
```

Logging Parameters with defer

As you can see, in `main` at **line 17**, we call a function `func1` with a parameter **Go**. Control will go to **line 7**. Then at **line 9**, we defer `func` function so that the execution of this line will be suspended. Control will transfer directly to **line 13**, and `func1` will return *named values*: `n` of type `int` and `err` of type `error`. Now again, control will go to **line 9**. Where line **func1("Go")=7, EOF** will be printed because `s` is **Go**, `n` is 7, and `err` is **EOF** (End Of File). Finally, the control will go back to `main` at **line 18**, and the program will terminate.

That's it about defer and tracing. Now, you'll study some *built-in* functions.