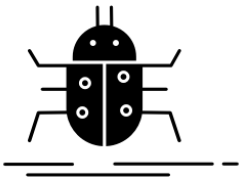
Testing for Errors on Functions

In this lesson, you'll learn about errors that originate from functions and how to resolve them.

WE'LL COVER THE FOLLOWING

Testing support



Sometimes, functions in Go are defined so that they return *two* results. One is the value and the other is the status of the execution. For example, the function will return a value and *true* in case of successful execution. Whereas, it will return a value (probably nil) and *false* in case of an unsuccessful execution.

Instead of true and false, an error-variable can be returned. In the case of successful execution, the error is nil. Otherwise, it contains the error information. It is then obvious to test the execution with an `if` statement because of its notation; this is often called the `comma, ok` pattern.

Consider the following example:

```
v, ok = sample_function(parameter)
```

Here, the `comma, ok` pattern is being followed. The value goes to `v`, and the `ok` parameter holds the status of the error during the execution. If there would be no error, then `sample_function` will return *true* to `ok` otherwise, it will return an error value.

Tracking error in a function

In the previous chapter while converting to and from a string, the function `strconv.Atoi` converts a string to an integer. There, we discarded a possible error-condition with:

```
anInt, _ = strconv.Atoi(origStr)
```

If `origStr` cannot be converted to an integer, the function returns 0 for `anInt`, and the `_` absorbs the error, and the program continues to run.

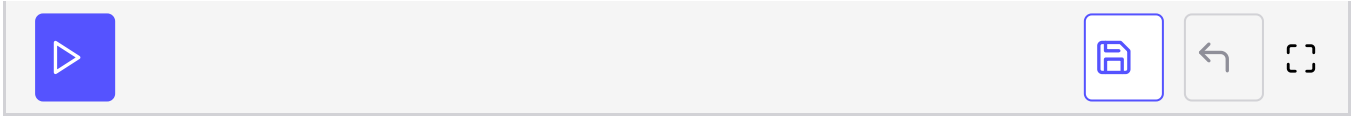
A program should test for every occurring error and behave accordingly by, at least informing the user (or world) of the error-condition and returning from the function or even halting the program.

Let's write another program to implement testing for errors.

```
package main
import (
"fmt"
"strconv"
)

func main() {
    var orig string = "ABC"
    var an int
    var err error
    // storing integer and error information
    an, err = strconv.Atoi(orig)

    if err != nil { // if it was an error, discontinue
        fmt.Printf("orig %s is not an integer - exiting with error\n", orig)
        return
    }
    fmt.Printf("The integer is %d\n", an)
    // rest of the code
}
```



String Conversion with Error Testing

In the above code, at **line 9** and **line 10**, we declare two variables `an` for value and `err` for error information. Then, we call the function `strconv.Atoi` for string `orig`. At **line 14**, we make a condition `if err != nil`. If this condition is *true* it means there was an error during the execution of the function, and due

to the `return` statement, the program will terminate. If the condition is *false*, then **line 18** and the rest of the code will be executed. In this case, the program will terminate with an error display message because `orig` isn't an integer.

We could also have used the form of return which returns a variable like `return err`. In this case, the calling function can examine the error `err`.

```
value, err := pack1.Function1(param1)
if err != nil {
    fmt.Printf("An error occurred in pack1.Function1 with parameter %v", p
aram1)
    return err
}
// normal case, continue execution:
```

In the above program, it was *main()* executing so the program stops. If we do want the program to stop in case of an error, we can use the function `Exit` from package `os` instead of `return`:

```
if err != nil {
    fmt.Printf("Program stopping with error %v", err)
    os.Exit(1)
}
```

Look at the following code where we tried to open a file:

```
f, err := os.Open(fname)
if err !=nil {
    return err
}
doSomething(f) // In case of no error, the file f is passed to a functio
n doSomething
```

Sometimes, the above idiom is repeated a number of times in succession. No else branch is written. If there is no error-condition, the code simply continues execution after the `if { }`.

> **Remark:** More information on `os` package can be found here.

Now you know how to handle errors. In the next lesson, you'll study another control structure called *switch case*.