Solution Review: Make a Stack with Variable Internal Types

This lesson discusses the solution to the challenge given in the previous lesson.

Environment Variables

Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

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

var st1 mystack.Stack

func main() {
        st1.Push("Brown")
        st1.Push(3.14)
        st1.Push(100)
        st1.Push([]string{"Java", "C++", "Python", "C#", "Ruby"})
        for {
                item, err := st1.Pop()
                if err != nil {
                        break
                }
                fmt.Println(item)
        }
}
```

In this program, we develop a generic `stack` type using a slice holding elements of type `interface{ }`. This is done in **mystack.go** in the folder **mystack**. The stack type is defined at **line 4** as: `type Stack []interface{}`.

This type is an array of items of a generic type `interface{}`; that means the items can be of any type like *ints*, *strings*, *booleans*, *arrays*, etc. The following functions are implemented in the file **mystack.go**:

- The `Len()` method amounts to `len` of the `stack` array (see **line** 7).

- The `IsEmpty()` method is defined as true when the `len(stack)` equals **0** (see **line 15**).

- Look at the header of the `Push()` method: `func (stack *Stack) Push(e interface{})` at **line 18**. It takes a `*Stack` as a receiver variable stack, and a parameter `e` of type `interface{}`. At **line 19**, it defines the new value of stack (`*stack`) by appending `e` to the old value of the stack.

- Look at the header of the `Top()` method: `func (stack Stack) Top() (interface{}, error)` at **line 22**. It takes a `Stack`. As a receiver variable stack, has no parameters, but it returns an item of the `Stack` type or an *error*: `(interface{}, error)`. It first tests whether the stack is empty; in this case, it returns `nil` for the item and an error: `errors.New("stack is empty").` If not empty, it returns the stack item at the end (or top) and `nil` for the `error: stack[len(stack)-1], nil`.

- Look at the header of the `Pop()` method: `func (stack *Stack) Pop() (interface{}, error)` at **line 29**. It has the same return type as `Top()`, but it has a `*Stack` as a receiver variable stack because this method has to change stack. It dereferences this pointer to a local variable `stk` (see **line 30**) so that the code is somewhat easier to read. The same test is performed on an empty stack as in `Top()`, and the stack item at the end (or top) is named `top`. Then, at **line 35**, the stack is changed by setting its new value `*stack` to `stk[:len(stk)-1]`.

In the **main.go** file a variable `st1` of the `mystack.Stack` type is declared at **line** 7. In order to be able to use the package `mystack`, we have to first import it at **line 4**. Then, in `main()` we use the `Push()` method to add a number of items of different types from **line 10** to **line 13**. Then, we loop over the array with *for*. We `Pop()` each item from the stack `st1` at **line 15**. If the stack becomes empty, the error is not `nil`, and we `break` from the loop at **line 17**. In the case there was still an item left, we print it out at **line 19**.

That's it about the solution. In the next lesson, OO behavior in Go is summarized.