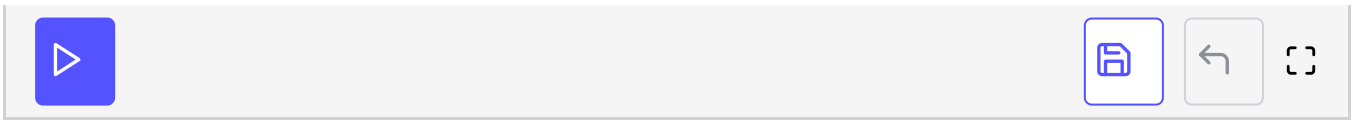
Solution Review: Implement Stack Data Structure

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

```
package main

import (
        "fmt"
        "strconv"
)

const LIMIT = 4 // DONOT CHANGE IT!

type Stack struct {
        ix   int // first free position, so data[ix] == 0
        data [LIMIT]int
}

func (st *Stack) Push(n int) {
        if st.ix == LIMIT {
                return // stack is full!
        }
        st.data[st.ix] = n
        st.ix++         // increment total no. of elements present
}

func (st *Stack) Pop() int {
        if st.ix >0{    // if stack not empty
                st.ix--         // decrease amount of elements present
                element := st.data[st.ix]
                st.data[st.ix] = 0
                return element
        }
        return -1       // if stack already empty
}

func (st Stack) String() string {
        str := ""
        for ix := 0; ix < st.ix; ix++ {
                str += "[" + strconv.Itoa(ix) + ":" + strconv.Itoa(st.data[ix]) + "] "
        }
        return str
}

func main() {
        st1 := new(Stack)
        fmt.Printf("%v\n", st1)
        st1.Push(3)     // function call to Push
        fmt.Printf("%v\n", st1)
        st1.Push(7) // function call to Push
        fmt.Printf("%v\n", st1)
        st1.Push(10) // function call to Push
```

```
        st1.Push(10) // function call to Push
        fmt.Printf("%v\n", st1)
        st1.Push(99) // function call to Push
        fmt.Printf("%v\n", st1)
        p := st1.Pop() // function call to Pop
        fmt.Printf("Popped %d\n", p)
        fmt.Printf("%v\n", st1)
        p = st1.Pop() // function call to Pop
        fmt.Printf("Popped %d\n", p)
        fmt.Printf("%v\n", st1)
        p = st1.Pop() // function call to Pop
        fmt.Printf("Popped %d\n", p)
        fmt.Printf("%v\n", st1)
        p = st1.Pop() // function call to Pop
        fmt.Printf("Popped %d\n", p)
        fmt.Printf("%v\n", st1)
}
```



Implement Stack Data Structure

In the above code, look at **line 8**. As per the requirement to generalize the implementation, we make a constant `LIMIT` and set its value to **4**. This means that our stack can, at maximum hold *four* values. Now that we have defined the limit of the stack, we make a struct of type `Stack` at **line 10**. It has two fields. The first field is an *integer* field `ix` that keeps track of how many elements are currently in the stack. The next field is an array of *integers* `data` of capacity `LIMIT`. It keeps track of the elements that are present in a stack. When you declare a `Stack` type variable, `ix` will be **0** and the `data` array will hold four *zeros*, initially. Now, let's see how the basic functionalities (push and pop) of the stack are implemented.

Look at the header of method `Push` at **line 15**: `func (st *Stack) Push(n int)`. The header makes it clear that only the *pointer* to the stack object can call this method, and `n` is the element that is to be pushed in the stack. Now, the question is, would you push every time? No, you can't push every single time because what if the stack is full (`ix` equals `LIMIT`) and you are trying to push an element at the *top*. In that case, it would give an error. Look at **line 16**, we start the implementation from this condition: `if st.ix == LIMIT`. If this condition is satisfied(`ix` of the stack `st` is equal to `LIMIT`), then control will exit from the function (see **line 17**). If not, then we'll add `n` in the stack at the *top*. Here, top means the index that is free from the end. Consider an example, if a stack is empty then `ix` will be **0**. So, `n` should be pushed at position **0**. If there is already *one* element in the stack, then `ix` will be one. So, `n` will be

inserted at index **1**. This means that every time `n` will be inserted at the `ix` position. See **line 19**. We are inserting `n` at the `ix` position of the `data` array. In the next line, we are incrementing `ix` because we need to create space for an empty top for the next push operation until the stack is full.

Look at the header of method `Pop` at **line 23**: `func (st *Stack) Pop() int`. The header makes it clear that only the *pointer* to the stack object can call this method, and the popped element is returned from the method. Now, the question is: would you pop every time? No, you can't pop every single time because what if the stack is empty (`ix` equals `0`), and you are trying to pop an element at the *top*. This would give an error. Look at **line 24**. We start the implementation from this condition: `st.ix >0`. If this condition is not satisfied (stack is empty), then control will exit from the function (see **line 30**). If satisfied, then we'll pop the element at the *top*. Here, top means the index that is free from the end. Consider an example. If there is already *one* element in the stack, then `ix` will be one, and the new element can be inserted at position **1**. So, this means to pop an element you have to remove the element from position **0**; that is `ix-1`. Look at **line 25**. We are decrementing `ix` by **1** to access the recently added value, and after the pop function, it's obvious we have to decrease `ix` because the number of elements decreases by 1. Before popping, we store the popped element in `element`, a separate variable. Before returning `element` we placed **0** at the position of the popped element, so that position looks empty. In this case, when the stack is empty we are returning **-1**.

Before going to `main`, let's see debugging function `String()`. Look at the header of `String` method at **line 33**: `func (st Stack) String() string`. It shows that the method can be called by the object of type `Stack`, and returns a string. The format of the returned string is `[Index: Value at Index]`. To make a string that shows the stack, we have to run a loop till `ix` times (see **line 35**), so that this pattern can be obtained: `[0:Value at Index 0] [1:Value at Index 1] ... [ix-1:Value at Index ix-1]`.

Now, look at `main`. At **line 42**, we make a *pointer* type variable `st1` of type `Stack` using the `new` function. In the next line, we call `Printf` function. This call will automatically transfer control to the `String` method declared at **line 33**. As the string is empty, an empty string will be printed on the screen.

At **line 44**, we are calling the `Push` method on `st1` for `n` equals to **3**. In the

next line, we call `Printf` function. This call will automatically transfer control to the `String` method declared at **line 33**. As the stack contains **3** at position **0**, **[0:3]** will be printed on screen. At **line 46**, we are calling the `Push` method on `st1` for `n` equals to 7. In the next line, we call the `Printf` function. As the stack now contains 7 at position **1**, **[0:3] [1:7]** will be printed on screen. At **line 48**, we are calling the `Push` method on `st1` for `n` equals to **10**. In the next line, we call `Printf` function. As the stack now contains **10** at position **2**, **[0:3] [1:7] [2:10]** will be printed on screen. At **line 50**, we are calling the `Push` method on `st1` for `n` equals to **99**. In the next line, we call the `Printf` function. As the stack now contains **99** at position **3**, **[0:3] [1:7] [2:10] [3:99]** will be printed on screen.

Now, at **line 52**, we are calling `Pop` on `st1`. With this pop, the last inserted value, which is **99**, will be popped and returned. The popped value is printed in the next line. At **line 54**, we call the `Printf` function. As **99** from position **3** was removed, **[0:3] [1:7] [2:10]** will be printed on screen. Now, at **line 55**, we are calling `Pop` on `st1`. With this pop, the last inserted value, which is **10**, will be popped and returned. The popped value is printed in the next line. At **line 57**, we call `Printf` function. As **10** from position **2** was removed, **[0:3] [1:7]** will be printed on screen. Now, again at **line 58**, we are calling `Pop` on `st1`. With this pop, the last inserted value, which is 7, will be popped and returned. The popped value is printed in the next line. At **line 60**, we call the `Printf` function. As 7 from position **1** was removed, **[0:3]** will be printed on screen. Now, again at **line 61**, we are calling `Pop` on `st1`. With this pop, the last inserted value, which is **3**, will be popped and returned. The popped value is printed in the next line. At **line 63**, we call the `Printf` function. As **3** from the position **0** was removed, an empty string will be printed because the stack is empty now.

That's it about the solution. In the next lesson, you'll study the concept of garbage in Go.