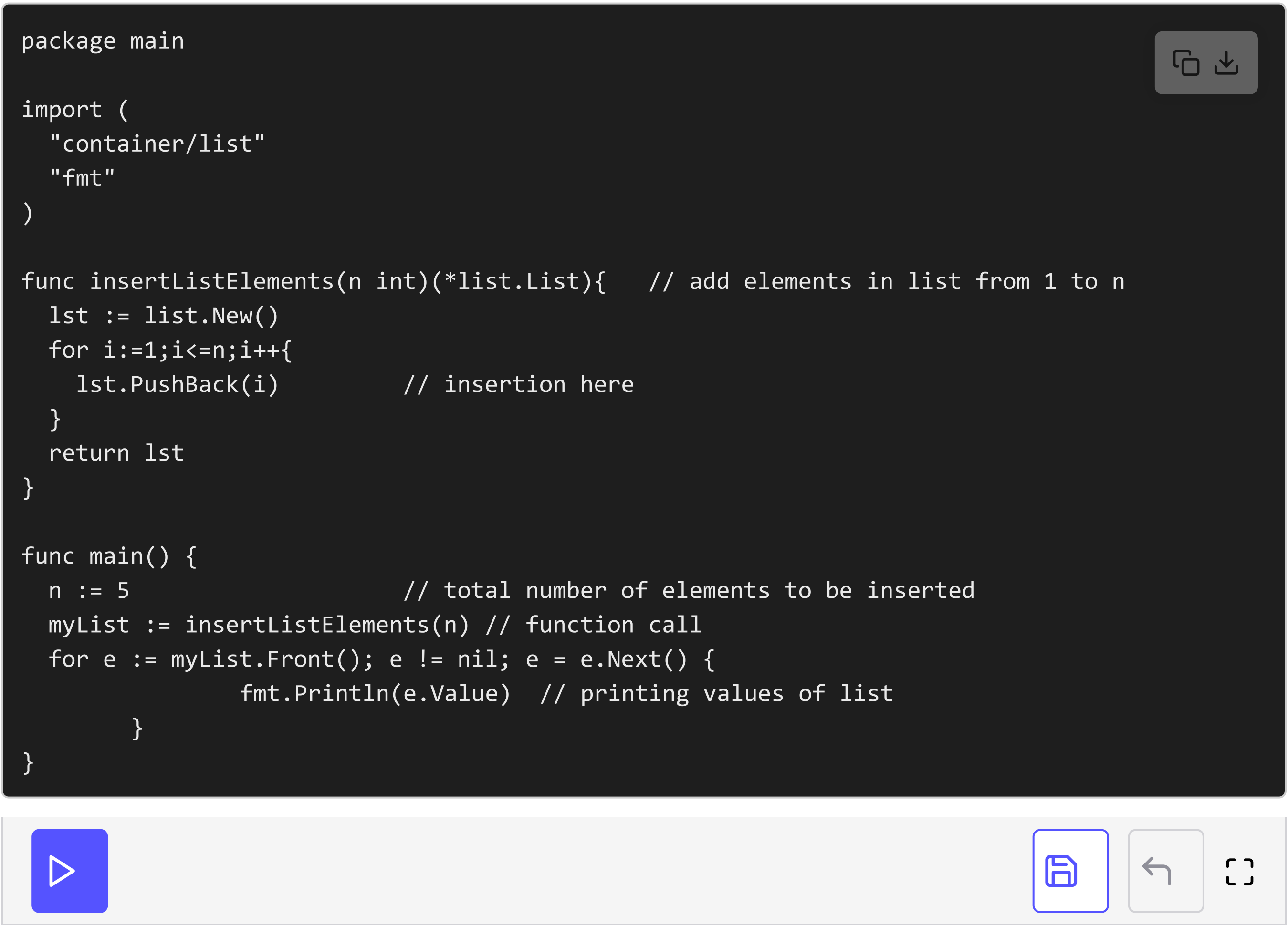
Solution Review: Construct a Double-Linked List

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



```go
package main

import (
  "container/list"
  "fmt"
)

func insertListElements(n int)(*list.List){   // add elements in list from 1 to n
  lst := list.New()
  for i:=1;i<=n;i++{
    lst.PushBack(i)         // insertion here
  }
  return lst
}

func main() {
  n := 5                    // total number of elements to be inserted
  myList := insertListElements(n) // function call
  for e := myList.Front(); e != nil; e = e.Next() {
                fmt.Println(e.Value)  // printing values of list
        }
}
```

Double Linked List

In the code above, at **line 4**, we import a package `container/list` because it is used for the implementation of doubly-linked lists. Look at the header of function `insertListElements` at **line 8**: `func insertListElements(n int)(*list.List)`. This function takes a parameter `n` and is returning a double-linked list. At **line 9**, we are making a new but empty doubly-linked list `lst` using `list.New()` function. Now we have a for loop that will run `n` times. The iterator `i` starts from **1** and ends at `n` because we have to add integers from **1** to `n`. **Line 11** is inserting values in `lst` in every iteration as: `lst.PushBack(i)`. By the end of the for loop, `n` values will be inserted, and we return `lst` from the function.

Now, look at the `main` function. We declare a variable `n` and initialize it with **5**. Remember `n` of `main` and `n` of `insertListElements` function are different variables because they have different scope. In the next line (**line 18**), we call the function `insertListElements` with `n` as a parameter, and store the return value from the function in `myList` list. Then we have a for loop at **line 19** to verify the insertions. We have `e` as an iterator. The iterator `e` acts as a node. It's obvious that we'll start from the front of the node and will visit all the nodes until we reach a `nill` value (end of the list).

The package `container/list` provides a function `Front()`, which returns the starting point of a list. So `e` was initialized with `myList.Front()`. The loop will continue until we reach `nill`. The question is how to move across the list. Previously in this course using an integer iterator, we simply increment or decrement it by **1** or any other integer. In the case of `e` acting as a node, this is not possible. Again the package `container/list` comes to rescue, by providing the `Next()` function. By doing `e = myList.Next()` function, we can change `e` to the next node in the `myList` list. To print the value of a node, you cannot directly print `e`, as `e` is the node. To print the value residing in `e`, you have to print `e.Value` (see line **20**). As output, **1** to **5** integers are printed on the screen.

That's it about the solution. In the next lesson, you'll be studying a package called `regexp`.