Stack Recitation

In this recitation we will be working to implement a Stack Abstract Data Type (ADT) and using a Stack to solve a symbol matching problem.

Part 1

Create the following generic interface `IStack.java`

```
public interface IStack<T>
{
    // adds the T parameter to the top of the stack
    public void push(T val);

    // removes and returns the top element of the stack
    // throws an EmptyStackException if there are 0 elements
    public T pop();

    // returns the top element of the stack
    // throws an EmptyStackException if there are 0 elements
    public T peek();

    // returns the current number of elements stored in the Stack
    public int size();
 }
```

Part 2

Complete the Stack class that implements the IStack Interface. This stack should use generics to store any valid type in the Stack. For example, we could create a Stack that stores Integers or Strings.

```
...
Stack<Integer> iStack = new Stack<>();
Stack<String> sStack = new Stack<>();
...
```

Features

1. The Stack internally is implemented using a simplified singly linked list of `Node` objects. For this we will define a private inner class in `Stack.java`

   ```
   private class Node
   {
       T val;
       Node next;
   }
   ```

   A Node consists of the a value of type `T` (the data being stored in the Stack) and a reference to the next Node in the list.

2. **Push**

   Should create a new `Node` and store the function parameter into the Node's `val`. Then "link" the new node as the new head of the list

3. **Pop**

   Should "unlink" the first node of the list (the head node) and return its value. If there are no elements in the Stack, throw an EmptyStackException

4. **Peek**

   Should return the value of the first node of the list (the head node). If there are no elements in the Stack, throw an ExmptyStackException

5. **Size**

   Should return the current size (number of elements) of the Stack.

Starter Code

```
import java.util.EmptyStackException;

public class Stack<T> implements IStack<T>
{
    // Inner class to represent a Node in the Stack
    private class Node
    {
        T val;
        Node next;
    }

    private Node head;
    // Add any other variables as needed

    // Your Code here

}
```

Part 3

Write a program (`MatchingSymbols`) that reads a string from standard input and prints `valid` if there are matching close symbols for each open symbol or `invalid` otherwise. Your implementation should correctly determine valid strings with nested symbols as well.
Your program only needs to handle the following symbols:

1. Open symbols: `({[<`
2. Close symbols: `)}]>`

A string is valid if there is a matching close symbol for each open symbol.
For example, `{abcd}`, is valid because there is one '}' for the opening '{'.
Other valid strings include:

- `{123[456]}`
- `()()()(){}[[[[]]]]`
- `abcdef(g)`

A string is invalid is there is not a matching close symbol for each open symbol or there are too many open symbols.
For example, `(()` is invalid since the first '(' has no matching ')'
Other invalid string include:

- `(asd>`
- `[[[]]`
- `asdasdasdasd}`
- `{`

Hint: Use a stack to keep track of the open symbols in the string, when you find a close symbol, look at the top symbol in the stack to check if it is the corresponding open symbol.

Sample Output (Valid)

```
$ java MatchingSymbols
input: <123<456>>
valid
```

Sample Output (Invalid)

```
$ java MatchingSymbols
input: (asd>
invalid
```