Laboratorio 3 – Persistent Stack

Este laboratorio tiene dos objetivos fundamentales:

1. Trabajar con una implementación de pilas.
2. Trabajar con nodos enlazados.

Tarea

El laboratorio 3 consiste en la creación de una pila inmutable en memoria dinámica utilizando nodos enlazados. ¿Qué significa inmutable? Que cada pila generada no puede ser modificada. Para ello, las operaciones que realicen una "modificación" sobre la pila, *push* y *pop,* deben devolver una nueva pila, de manera que realmente no alterarán aquella pila sobre la que se apliquen, sino que crearán una nueva pila basada en la original con la modificación pertinente. En otras palabras, si realizamos, por ejemplo, la operación *push* sobre una instancia de pila llamada *pila1*, es decir, añadimos un elemento a la cima de *pila1*, la operación no debe modificar *pila1* añadiendo el elemento a su cima, sino que debe devolver una nueva pila (*pila2*) que contendrá los elementos de *pila1* más el elemento nuevo que será su cima. Para poder hacer esto sin malgastar grandes cantidades de memoria, los elementos comunes a ambas pilas serán compartidos. En concreto, en el ejemplo anterior, ¿qué elementos comparten *pila1* y *pila2*? La respuesta es que ambas comparten los elementos que contiene *pila1*, teniendo *pila2* un nuevo elemento, que se corresponde con su cima. La cima de *pila1* será el segundo elemento en *pila2* (el que hay después de su cima).

Además, hay que tener en cuenta que no sólo la pila debe ser inmutable sino que los elementos que contenga, también deben serlo para que la implementación tenga sentido.

Interfaz Stack<E>

En primer lugar, será necesario definir una interfaz, que denominaremos *Stack<E>*:

```
public interface Stack<E> {
    Stack<E> push(E elem);
    Stack<E> pop() throws StackError;
    E top() throws StackError;
    boolean isEmpty();
}
```

Las operaciones de *Stack<E>* son:

- **push**: devuelve una nueva pila basada en aquella desde la que se realiza la llamada, pero añadiendo un elemento a la cima.
- **pop**: devuelve una nueva pila basada en aquella desde la que se realiza la llamada, pero eliminando el elemento de la cima. Si la pila está vacía lanza *StackError*.
- **top**: devuelve el elemento de la cima de la pila, sin modificarla, o lanza *StackError* si ésta está vacía.
- **isEmpty:** devuelve *true* si la pila está vacía o *false* en caso contrario.

Como se puede comprobar *pop* y *top* lanzan *StackError*, clase que representa diferentes errores y que es la siguiente:

```
public class StackError extends Exception {
    public StackError(String msg) {
        super(msg);
      }
}
```

Como se trata de una excepción comprobada (es decir, no extiende *Error* o *RuntimeException*), debemos declararla en la cabecera de los métodos que pueden lanzarla (*pop* y *top*).

Clase SharedStack<*E*>

El siguiente paso será implementar la interfaz que hemos definido. Esto lo hará la clase *SharedStack<E>* y utilizará nodos enlazados (memoria dinámica), tal y como se han utilizado en el tema 3 en la implementación de *LinkedList<E>*. Se hará uso de una clase anidada estática, llamada *Node<E>*, que representará cada nodo y en cuyo interior contendrá un enlace al elemento de la pila que contiene, así como otro al nodo que contenga el elemento insertado previamente en la pila:

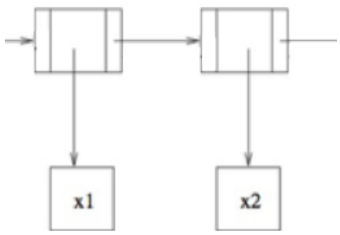

Figura 1. Ejemplo de representación de nodos con elementos de la pila

El aspecto de la clase *SharedStack*<*E*> podría ser el siguiente:

```
public class SharedStack<E> implements Stack<E> {

    private final Node<E> topOfStack;

    private static class Node<E> {
        private final E elem;
        private final Node<E> next;

        // ¿?

    }

    public SharedStack() {
        // ¿?
    }

    // Otros Constructores
        //¿?

    @Override
    public SharedStack<E> push(E e) {
        // ¿?
    }

    @Override
    public SharedStack<E> pop() throws StackError {
        // ¿?
    }


    @Override
    public E top() throws StackError {
        // ¿?
    }

    @Override
    public boolean isEmpty() {
        // ¿?
    }
}
```

Entrega

El resultado obtenido debe ser entregado por medio de un proyecto IntelliJ por cada grupo, comprimido y con el nombre "Lab3_NombreIntegrantes". En el proyecto deberán estar presentes los test realizados con JUnit 4.

Además, se debe entregar un documento de texto, máximo de cuatro páginas, en el cual se explique la solución a cada una de las tareas.

Consideraciones de Evaluación

A la hora de evaluar el laboratorio se tendrán en cuenta los siguientes aspectos:

- El código de cada proyecto ofrece una solución a cada una de las tareas planteadas.

- Realización de test con JUnit 4.0. Además de comprobar el funcionamiento de la pila, debéis comprobar que realmente se esté compartiendo memoria entre las pilas. Para ello, cread una clase mutable que os permita comprobar que, al modificar un elemento de la pila, todas las comparticiones de este han cambiado. Para diseñar los tests es muy conveniente dibujar los diferentes nodos y enlaces (y esos diagramas deberían ser incluidos en la documentación).

- Explicación adecuada del trabajo realizado, es decir, esfuerzo aplicado al documento de texto solicitado.

- Calidad y limpieza del código.