Ejercicios Genericidad

Semana del 30 de Marzo

1) Modifica la clase `semiGroup` que has hecho en ejercicios anteriores, para que pueda contener elementos de cualquier tipo, no necesariamente enteros. Para ello, tendrás que hacer la interfaz `IOperation` genérica y ten en cuenta que **las excepciones no pueden ser genéricas**.

Por ejemplo, el siguiente código debería producir la salida indicada más abajo:

```
enum Colour { RED, BLUE, GREEN; }
class ColourShifter implements IOperation<Colour> {
  @Override public Colour operate(Colour a, Colour b) {
       int idx = (a.ordinal()+b.ordinal())%Colour.values().length;
       return Colour.values()[idx];
  }
}

public class SemiGroupMain {
  public static void main(String[] args) {
    try {
         SemiGroup<Colour> sg = new SemiGroup<>(new ColourShifter(),
                                                Arrays.asList(Colour.values()));
         System.out.println( "Operating "+Colour.BLUE+
                             " and "+Colour.GREEN+
                             " yields "+sg.calculate(Colour.BLUE, Colour.GREEN));
    }
    catch (SemiGroupException ex) {
         System.err.println(ex);
    }
  }
}
```

Salida esperada:

```
Operating BLUE and GREEN yields RED
```

Solución:

```
public interface IOperation<T> {
	T operate (T a, T b);
}

public class SemiGroup<T> {
	private Set<T> set = new LinkedHashSet<>();
	private IOperation<T> oper;

	public SemiGroup(IOperation<T> oper, Collection<T> elems)
				throws IllFormedSemiGroupException{
		this.oper = oper;
		this.set.addAll(elems);
		this.checkSemiGroup();
	}

	private void checkSemiGroup() throws IllFormedSemiGroupException{
		T res = null;
		for (T p1 : this.set)
			for (T p2 : this.set)
				try {
					res = this.calculate(p1, p2);
				} catch (SemiGroupException e) {
					throw new IllFormedSemiGroupException(p1, p2, e);
				}
	}

	public T calculate(T i, T j) throws SemiGroupException{
		if (!this.set.contains(i)) throw new ParameterNotInGroupException(i);
		if (!this.set.contains(j)) throw new ParameterNotInGroupException(j);
		T res = this.oper.operate(i, j);
		if (!this.set.contains(res)) throw new ResultOutOfGroupException(res);
		return res;
	}

	@Override public String toString() {
		return this.set.toString()+" with operation "+this.oper;
	}
}

public class AddModulo implements IOperation<Integer> {
	private int modulo;
	public AddModulo(int m) { this.modulo = m; }
	@Override public Integer operate(Integer a, Integer b) {
		return (a+b) % this.modulo;
	}
	@Override public String toString() { return "+ modulo "+this.modulo; }

}

// The following is a bit ugly, but exceptions cannot be generic, and so we use Objects instead
public abstract class SemiGroupException extends Exception {
	public SemiGroupException(String msg) { super(msg); }
}

public class IllFormedSemiGroupException extends SemiGroupException {
	public IllFormedSemiGroupException(Object p1, Object p2, SemiGroupException ex) {
		super ("Ill formed semigroup found when operating "+
				p1+" and "+p2+"\n"+ex.toString());
	}
}

public class ParameterNotInGroupException extends SemiGroupException {
	public ParameterNotInGroupException(Object e) {
		super("Parameter "+e+" is not an element of the semigroup");
	}
}

public class ResultOutOfGroupException extends SemiGroupException {
	public ResultOutOfGroupException(Object e) {
		super("Result "+e+" is not an element of the semigroup");
	}
}
```

2) Crea una clase `Tree` genérica que permita representar árboles cuyos nodos tengan un objeto de un tipo parametrizable. La clase debe tener un método add para añadir hijos a los nodos del árbol, y un método `getChildren` que obtenga una lista no modificable con todos los hijos de un nodo del árbol. Cada nodo del árbol mantiene una lista con sus hijos ordenados por orden natural. Así, el siguiente main debe producir la salida de más abajo.

```
 public static void main(String[] args) {
   Tree<Integer> myTree = new Tree<Integer>(4);
   Tree<Integer> child = myTree.add(10); // Creates node 10, adds to myTree and returns it
   myTree.add(6);
   myTree.add(-1).add(7); // adds -1 to the root, and then 7 to node -1

   System.out.println("tree: "+myTree);

   List<Tree<Integer>> children = myTree.getChildren();
   System.out.println(children);
   children.add(new Tree<Integer>(8)); // we cannot modify this list
 }
```

Salida esperada

```
tree: 4 [-1 [7], 6, 10]
[-1 [7], 6, 10]
Exception in thread "main" java.lang.UnsupportedOperationException
	at java.util.Collections$UnmodifiableCollection.add(Unknown Source)
	at tree.TreeMain.main(TreeMain.java:17)
```

Solución:

```
public class Tree<T extends Comparable<T>> implements Comparable<Tree<T>>{
	private T value;
	private List<Tree<T>> children = new ArrayList<>();

	public Tree(T elem) {
		this.value = elem;
	}

	public Tree<T> add(T elem) {
		Tree<T> node = new Tree<T>(elem);
		this.children.add(node);
		Collections.sort(this.children);
		return node;
	}

	public List<Tree<T>> getChildren() {
		return Collections.unmodifiableList(this.children);
	}

	@Override public int compareTo(Tree<T> node) {
		return this.value.compareTo(node.value);
	}

	@Override
	public String toString() {
		String children = this.children.isEmpty() ? "" : " "+this.children.toString();
		return this.value+children;
	}
}
```

3) **[Del examen final del año 2013]** Se desea diseñar un sistema genérico para crear flujos de trabajo en Java. Un flujo de trabajo (workflow) está definido por varias tareas (tipo Task) conectadas de forma que al introducir información de entrada en la primera tarea se obtiene una salida que es a su vez la entrada para la segunda tarea, y así sucesivamente.

Cada tarea tendrá un método que recibe una entrada y la transforma en una salida. Para conectar tareas se crearán conectores que son tipos especiales de tareas que implementarán la lógica de diversas formas de conexión.

Tu programa debe considerar dos tipos de conexiones:

a) `SerialConnector`, que conecta dos tareas, de forma que la salida de la primera es la entrada de la segunda. Esta clase usa tres parámetros genéricos, el tipo de la entrada, el tipo intermedio y el tipo de salida.
b) `ListConnector`, que aplica una misma tarea a una lista de entradas, produciendo una lista de salidas que tendrá el mismo tamaño que la lista de entradas. Los parámetros genéricos son el tipo de los elementos de la lista de entrada y el tipo de elementos de la lista de salida.

Dadas las siguientes clases de tareas de utilidad `Max` y `String2Double`, **se pide**: completar el siguiente programa, para que la salida sea la de más abajo, implementando las clases o interfaces necesarias y rellenando los espacios indicados

```
public class Max<T extends Comparable<T>> implements Task<List<T>, T> {
	@Override
	public T execute(List<T> input) {
		T result = null;
		for (T i:input)
			if (result == null || result.compareTo(i) < 0)
				result = i;
		return result;
	}
}
```

```
public class String2Double implements Task<String, Double> {
	@Override public Double execute(String input) { return Double.parseDouble(input); }
}
```

```
public class Main { // completar espacios subrayados
	public static void main(String[] args) {
		Max<Double> maxDouble= new Max<Double>();
		Max<String> maxString= new Max<String>();
		Task<__ String, Double __> toDouble = new String2Double();
		Task<__ List<String>, Double___> workflow =
			new SerialConnector<List<String>, __ List<Double>_____  , Double>(
					new ListConnector<String, Double>(toDouble),
					maxDouble);
		String[] test = {"3.2", "1", "19", "0"};
		System.out.println(workflow.execute(Arrays.asList(test)));
		System.out.println(maxString.execute(Arrays.asList(test)));
	}
}
```

Salida esperada

```
19.0
3.2
```

```java
public class ListConnector<I, O> implements Task<List<I>, List<O>>{
	final private Task<I, O> task;
	public ListConnector(Task<I, O> task){
		this.task=task;
	}
	@Override
	public List<O> execute(List<I> input) {
		List<O> result= new ArrayList<O>(input.size());
		for (I i: input){
			result.add(task.execute(i));
		}
		return result;
	}
}
```

```java
public class SerialConnector<I, T, O> implements Task<I, O>{
	final private Task<I, T> in;
	final private Task<T, O> out;
	public SerialConnector(Task<I, T> in, Task<T, O> out){
		this.in=in;
		this.out=out;
	}
	@Override
	public O execute(I input) {
		return out.execute(in.execute(input));
	}
}
```

```java
public interface Task <I, O>{
	O execute (I input);
}
```