Solución a los ejercicios sobre Interfaces

Semana del 16 de Marzo

1) Diseñar el API de una clase de utilidad con un método de ordenación "sort", para ordenar listas de String. El método sort debe ordenar los elementos de la lista por un criterio de orden a definir por el usuario, y que se le pasa como parámetro.

// Un ejemplo de código cliente de la interfaz

```
public class Main {
	public static void main (String...args) {
		List<String> miLista = Arrays.asList("hola", "un", "dos");
		// queremos poder hacer algo como esto
		ListaUtil.sort(miLista, new MenorAMayor());
	}
}
```

```
import java.util.List;

public class ListaUtil {
	// CriterioOrdenacion es una interfaz que permite definir el criterio de ordenacion
	public static void sort(List<String> lista, CriterioOrdenacion crit) {
		//…
	}
}
```

```
// Esta es la interfaz que todos los criterios de ordenación para Strings deben seguir
public interface CriterioOrdenacion {
	int compara(String s1, String s2);
	// devuelve < 0 si s1<s2, ==0 si s1=s2, >0 si s1>s2
}
```

```
// Un ejemplo de criterio de orden, que order de menor a mayor longitud
public class MenorAMayor implements CriterioOrdenacion{
	@Override
	public int compara(String s1, String s2) {
		return s1.length() - s2.length();
	}
}
```

Posibles mejoras:

- Podemos mejorar el método sort para que acepte listas de cualquier tipo, no solo de String.
- Esto hace que la interfaz CriterioOrdenacion tenga que ser más general, para permitir criterios aplicables a cualquier tipo de dato. Esta mayor generalidad se consigue con una interfaz genérica:

```
public interface CriterioOrdenacion<T> {
	int compara(T s1, T s2);
}
```

Y la clase MenorAMayor, tendría que modificarse así:

```
public class MenorAMayor implements CriterioOrdenacion<String>{
	@Override public int compara(String s1, String s2) {
		return s1.length() - s2.length();
	}
}
```

2) Crea una clase de utilidad `PrettyPrinter` para imprimir por pantalla estructuras en forma de árbol de manera indentada. La clase de utilidad ha de ser altamente reutilizable, por ejemplo con la clase Directorio del ejericio de la sección anterior.

// Esta clase imprime de manera indentada estructuras en forma de árbol, que sean conformes a la
// interfaz Tree

```
public class PrettyPrinter {
	private static final String INDENT = "    ";

	public static void print(Tree t) {
		print(t, "");
	}
	private static void print(Tree t, String indent) {
		System.out.println(indent+t.valor());
		for (Tree child : t.children())
			print(child,indent+INDENT);
	}
}
```

```
import java.util.List;
public interface Tree {
	String valor();
	// Podríamos no exigir este método y usar toString(), pero puede que las clases
	// que implementen Tree quieran dar una representación distinta a toString()

	List<? extends Tree> children();
	// <? extends Tree> significa: Un tipo que sea conforme a Tree (Tree, o cualquier
	// clase que lo implement).
	// Permite a una clase Directorio que implemente Tree devolver List<Directorio>
}
```

Usando la clase de utilidad:

- Usa la interfaz Tree para imprimir directorios de manera indentada (ejercicio sistema de ficheros de la sección anterior)

3) Se quiere construir una clase `SemiGroup`, que modele el concepto matemático de semigrupo. Un semigrupo es un conjunto con una operación que toma dos elementos del conjunto y produce otro elemento del conjunto. Por simplificar, consideraremos que los semigrupos son conjuntos de enteros y no haremos ninguna suposición adicional sobre la operación (que debería ser asociativa).

Así pues, la clase `SemiGroup` debe tener un constructor que tome como parámetros la operación y una colección de enteros. La operación debe ser conforme a la siguiente interfaz.

```
interface IOperation {
   Integer operate(Integer a , Integer b );
}
```

Un semigrupo debe tener un método `calculate`, que invoca la operación y devuelve null si los parámetros o el resultado no pertenecen al conjunto. Como ves en el código de más abajo, la operación `AddModulo` implementa la interfaz `IOperation` realizando la suma módulo el número que se le pasa como parámetro en el constructor.

Completa el siguiente programa para que se obtenga la salida de más abajo

```
package semiGroup;
import java.util.Arrays;

public class SemiGroupMain {
	public static void main(String[] args) {
		SemiGroup sg = new SemiGroup(new AddModulo(4), Arrays.asList(0, 1, 2, 3));

		System.out.println(sg.calculate(1, 2));
		System.out.println(sg.calculate(3, 4));
		System.out.println(sg);
	}
}
```

Salida esperada

```
3
null
[0, 1, 2, 3] with operation + modulo 4
```

Nota: Una mejor idea para señalar los dos tipos de errores es generar excepciones en lugar de devolver null (agregaremos esta característica en la próxima lección).

// Esta clase es conforme a IOperation

```
public class AddModulo implements IOperation {
	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;
	}
}
```

// La clase simplemente tiene un conjunto de enteros, y una operacion

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

	public SemiGroup(IOperation oper, Collection<Integer> elems) {
		this.oper = oper;
		this.set.addAll(elems);
	}

	public Integer calculate(Integer i, Integer j) {
		if (!this.set.contains(i) || !this.set.contains(j)) return null;
		Integer res = this.oper.operate(i, j);
		if (!this.set.contains(res)) return null;
		return res;
	}

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