Ejercicios: Nuevos conceptos Java 8 y expresiones Lambda

Semana del 20 de Abril

1) **Métodos default en interfaces**: Dada la siguiente interfaz genérica `Arbol<T>`:

```
public interface Arbol<T>{
  T getElemento();
  Arbol<T> hijoIzq();
  Arbol<T> hijoDer();
  default boolean esHoja() {
       return this.hijoIzq().esVacio() && this.hijoDer().esVacio();
  }
  boolean esVacio();
  default Arbol<T> search(T o) {
       if (this.getElemento().equals(o)) return this;
       else {
             Arbol<T> result = null;
             if (! this.hijoIzq().esVacio() ) result = this.hijoIzq().search(o);
             if (result != null) return result;
             if (! this.hijoDer().esVacio() ) result = this.hijoDer().search(o);
             if (result != null) return result;
       }
       return null;
  }
}
```

Añadir un método estático a la interfaz que devuelva un Comparator que:

- Compare por el valor de los nodos raíz (si uno es vacío y el otro no, es mayor el no vacío).
- Si las raíces no son nulas, compara por el orden natural.

Solución:

```
public interface Arbol<T>{
       //…
       static <T extends Comparable<T>> Comparator<Arbol<T>> getComparator() {
             return new Comparator<Arbol<T>>() {
                   @Override
                   public int compare(Arbol<T> o1, Arbol<T> o2) {
                         if (o1.esVacio() && !o2.esVacio()) return -1;
                         if (o2.esVacio() && !o1.esVacio()) return 1;
                         return o1.getElemento().compareTo(o2.getElemento());
                   }

             };

       }
       //…
}
```

2) **Expresiones lambda:** Diseña una clase que almacene secuencialmente datos de cualquier tipo, y que pueda devolver una secuencia filtrada de dichos datos por un criterio configurable.

```
import java.util.*;
import java.util.function.Predicate;

public class Store<T> {
       private Predicate<? super T> filter;
       private List<T> data = new ArrayList<>();
```

```
	public Store(Predicate<? super T> filter) { this.filter = filter; }
	public Store<T> add(T i) {
		if (this.filter.test(i)) this.data.add(i);
		return this;
	}
	@Override public String toString() { return this.data.toString(); }
	public static void main(String[] args) {
		Store<Integer> store = new Store<Integer>(x -> x%2==0);
		System.out.println(store.add(3).add(2).add(5).add(6));
	}
} // Similar al ejercicio 1 de la hoja de clases anónimas
```

3) **Expresiones lambda**: En el ámbito de programación funcional, el currying es una técnica muy utilizada para reducir los parámetros de una función. Así, dada una función f: X × Y → Z, curry(f) devuelve una función h: X → (Y → Z). Esto es, h toma un argumento de tipo X, y devuelve una función Y → Z, definida de tal manera que h(x)(y) = f(x, y).

Se pide:

- Implementar *curry* utilizando expresiones lambda.
- Pista: la interfaz BiFunction<X, Y, Z> puede usarse para modelar f, y Function<Y, Z> para modelar h.

Solución:

```
public class Currying {
  private <X, Y, Z> Function<X, Function<Y, Z>> curry(BiFunction<X, Y, Z> f) {
      return x -> (y -> f.apply(x, y));
  }

  public static void main(String ...args) {
      System.out.println(((BiFunction<Integer, Integer, Integer>)(x, y) -> x + y).
                         apply(3, 4)); // a bifunction needs two arguments
      Currying c = new Currying();
      Integer result = c.<Integer, Integer, Integer>curry((x, y) -> x + y).
                         apply(3).
                         apply(4); // curry allows evaluating the bifunction step-by-step
      System.out.println(result);
  }
}
```

4) **Expresiones lambda:** Usando lambdas, crea un simulador para máquinas de estados. Una máquina de estados está formada por estados y una serie de variables, que asumimos de tipo Integer. Se pasa de un estado a otro cuando sucede un evento (de tipo String). Los estados pueden tener acciones asociadas, que se ejecutan al salir del estado debido a algún evento, y pueden por ejemplo modificar las variables.

Así, el siguiente programa:

```
public class Main {
  public static void main(String[] args) {
    StateMachine sm = new StateMachine("Light", "num"); // nombre y variable
    State s1 = new State("off");
    State s2 = new State("on");
    s1.addEvent("switch", s2);  // una transición de s1 a s2 por el evento switch
    s2.addEvent("switch", s1);  // una transición de s2 a s1 por el evento switch
    // Una lambda con dos parámetros: el estado actual y el evento que se recibe
    s1.action((State s, String e) -> s.set("num", s.get("num")+1) ); // aumenta num
    s2.action((State s, String e) -> s.set("num", s.get("num")+1) );
    sm.addStates(s1, s2); // añadimos los dos estados a la máquina de estados
    sm.setInitial(s1);    // s1 es el estado actual
    System.out.println(sm);
    MachineSimulator ms = new MachineSimulator(sm);
    ms.simulate(Arrays.asList("switch", "switch")); // simulamos la máquina con 2 eventos
   }
```

```
}
```

Debe producir la siguiente salida:

```
Machine Light : [off, on]
switch: from [off] to [on]
Machine variables: {num=1}
switch: from [on] to [off]
Machine variables: {num=2}
```

Solución:

```
public class StateMachine {
	private String n;
	private Map<String, Integer> variables = new LinkedHashMap<>();
	private List<State> states = new ArrayList<>();
	private State initial;

	public StateMachine(String nombre, String ...variables) {
		this.n = nombre;
		Arrays.asList(variables).forEach(v -> this.variables.put(v, 0));
	}
	public void addStates(State ...states) {
		Arrays.asList(states).forEach( s -> {
			this.states.add(s);
			s.setMachine(this);
		});
	}
	public void setInitial(State s1) { this.initial = s1; }
	@Override public String toString() { return "Machine "+this.n+" : "+this.states; }
	public State initial() { return this.initial;	}
	public Map<String, Integer> getVariables() { return this.variables; }

}

public class State implements Function<String, State>, Consumer<String>
{
	private String nombre;
	private Map<String, State> transiciones = new LinkedHashMap<>();
	private BiConsumer<State, String> action;
	private StateMachine sm;

	public State(String n) { this.nombre = n; }
	public void addEvent(String ev, State st) { this.transiciones.put(ev, st); }
	public void action(BiConsumer<State, String> action) { this.action = action; }
	public int get(String var) { return sm.getVariables().get(var); }
	public void setMachine(StateMachine m) { this.sm = m; }
	public void set(String v, int i) { this.sm.getVariables().put(v, i); }
	@Override public String toString() {return this.nombre; }
	@Override public State apply(String e) { return this.transiciones.get(e); }
	@Override public void accept(String e) {
		if (this.action!=null) this.action.accept(this, e);
	}
}

public class MachineSimulator {
	private StateMachine machine;

	public MachineSimulator(StateMachine sm) { this.machine = sm; }
	public void simulate(List<String> eventos) {
		State current = this.machine.initial();
		for (String e : eventos) {
			current.accept(e);
```

```
                    System.out.println(e+" : from ["+current+"] to ["+current.apply(e)+"]");
                    current = current.apply(e);
                    System.out.println("Machine variables "+this.machine.getVariables());
                }
        }
}
```

5) **Expresiones Lambda (del examen final de 2019)**: Se quiere diseñar una clase `MapTools`, con métodos útiles para potenciar el uso del método `map` de `Stream`. Debes añadir el método `enRango` que permitirá usar `map` con una función *base* y dos funciones *limitadoras*, una *limitadora superior* y otra *limitadora inferior*. Sin considerar las funciones limitadoras, el resultado final sería el mismo que utilizando el método `map` directamente con la función base. En cambio, cada función limitadora impone un límite (uno superior y otro inferior) al valor resultante de la función base antes de que sea tratado por el método `map` en su forma habitual. Es decir, si el valor de la función base supera al de la limitadora superior, `map` deberá recibir el valor de la limitadora superior para operar con él, y si es inferior al de la limitadora inferior deberá recibir el de esta limitadora. En cualquier otro caso, `map` debe recibir el valor de la función base. Ver ejemplos de uso abajo.

Se pide: Implementar el método `enRango` para la clase `MapTools` de la forma más genérica y flexible que sea razonable para que el siguiente programa ejemplo produzca la salida indicada abajo.

```
import java.util.*;
import java.util.stream.*;

public class Ej4 {
    public static Integer doble(Number n) { return 2 * n.intValue(); }

    public static void  main(String[] args) {
     Collection<Integer> lista = Arrays.asList(2, -3, 5, 32, -10, -20);
     System.out.println( lista.stream()
                              .map( Ej4::doble )  // sin limitadoras
                              .collect( Collectors.toList() ));

     System.out.println( lista.stream()  // con dos limitadoras, inferior y superior
                               .map( MapTools.enRango(Ej4::doble, x -> x-3,  x -> x+3) )
                               .collect( Collectors.toList() ));

     Collection<String> palabras = Arrays.asList( "bata", "buey", "paz", "zoo" );
     System.out.println( palabras.stream()
                                 .map( MapTools.enRango(x->x,  x->"boy",  x->"que") )
                                 .collect( Collectors.toList() ));
   }
}
```

Salida esperada:

```
[4, -6, 10, 64, -20, -40]
[4, -6, 8, 35, -13, -23]
[boy, buey, paz, que]
```

Solución:

```
class MapTools {
    public static <T extends Comparable<T>>
            Function<T,T> enRango( Function<? super T, ? extends T>  base,
                                   Function<? super T, ? extends T> bottom,
                                   Function<? super T, ? extends T> top) {
        return x -> Collections.min( Arrays.asList( top.apply(x),
                                     Collections.max( Arrays.asList( bottom.apply(x),
                                                                     base.apply(x)    )) ));
    }
}
```

Otra solución:

```
class MapTools {
```

```
    public static <T extends Comparable<? super T>>
              Function<T, T> enRango( Function<? super T, ? extends T>  base,
                                      Function<? super T, ? extends T> bottom,
                                      Function<? super T, ? extends T> top) {
       return x -> {
                     T r = base.apply(x);
                     T max = top.apply(x);
                     T min = bottom.apply(x);
                    return (max.compareTo(r) > 0) ? max : (min.compareTo(r) < 0) ? min : r;
                   };
    }
}
```

6) **Expresiones Lambda (del examen final de 2018)**: Al procesar un *stream* es muy frecuente hacer filtros elementales para combinarlos en secuencia (como en s.filter(f1).filter(f2)...) y eso equivale a hacer la conjunción lógica de los filtros. En cambio, en este ejercicio te pedimos que añadas a la clase `STools` un método `filterOr` que, teniendo como parámetros un *stream*, `s`, y un número indeterminado de filtros, devuelva un *stream* que contiene solo los elementos de `s` que cumplen al menos uno de los filtros dados, es decir, aplica la disyunción de filtros al *stream* `s`, como se ve en el ejemplo de abajo.

Además, surge la idea de que cualquier conjunto también podría ser empleado para componer filtros como se ha hecho arriba. Es decir, un conjunto visto como filtro acepta o deja pasar solamente aquellos elementos que contiene. Así pues, como nos interesan los filtros disyuntivos, debes añadir a la clase `STools` un método `filterSets` que, como se ve en el ejemplo, es similar al método del párrafo anterior *pero acepta cualquier tipo de conjuntos como filtros*. Nótese que, en el ejemplo, "Ja" sale dos veces porque está dos veces en la entrada y el conjunto se usa solamente para filtrar, no para almacenar las palabras.

Bien pensado, si alguien prevé usar su conjunto como filtro, podría crear un tipo especial de filtro, que fuese combinable, en disyunción, no solo con otros filtros de su tipo, sino con cualquier otro filtro que sea aceptado por el método `filterOr` que se describió en el primer párrafo. Es decir, define la clase `TestSet` que se usa en el programa ejemplo, para que se pueda pasar como argumento al método `filterOr` de forma que el programa de abajo (tercera línea de salida) pueda combinar disyuntivamente filtros de varios tipos.

Se pide:

Implementar las clases e interfaces necesarias para que el código dado produzca la salida de abajo. Se valorará específicamente la calidad del diseño, la flexibilidad en el uso de genéricos y la concisión del código.

```
public class Main {
   public static boolean longWord(String s) { return s.length() > 4; }

   public static void main(String[] args) {
      Collection<String> pal = Arrays.asList("Ja", "Ja", "Reírse", "es", "algo", "saludable");

      Stream<String> ss = STools.filterOr(pal.stream(),
                                          s -> s.startsWith("J"),
                                          s -> s.length() > 4);
      System.out.println(ss.collect(Collectors.toList()));

      System.out.println(STools.filterSets(pal.stream(),
                                           Set.of("Ja", "Je", "Jo"),
                                           Collections.emptySet(),
                                           Set.of("es", "no") )
                               .collect(Collectors.toList()));

      System.out.println(STools.filterOr(pal.stream(),
                                           new TestSet<>(Set.of("Ja", "Je", "Jo")),
                                           Main::longWord,
                                           new TestSet<>(Set.of("es", "no")) )
                               .collect(Collectors.toList()));
   }
}
```

Salida esperada:

```
[Ja, Ja, Reírse, saludable]
[Ja, Ja, es]
[Ja, Ja, Reírse, es, saludable]
```

Solución:

```
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

class TestSet<T> extends HashSet<T> implements Predicate<T> {
    public TestSet(Collection<? extends T> c) { super(c); }
    @Override public boolean test(T t) { return this.contains(t); }
}

class STools {
    public static  <T> Stream<T> filterOr(Stream<T> s,
                                  Predicate<? super T>... ps) {
        return s.filter( t -> { for (Predicate<? super T> p : ps)
                                      if (p.test(t)) return true;
                                return false; } );
    }
    public static  <T> Stream<T> filterSets(Stream<T> s,  Set<? super T>... sets) {
        return s.filter( t -> { for (Set<? super T> p : sets)
                                     if (p.contains(t)) return true;
                                return false; } );
    }
}
```

Otra solución:

```
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

class TestSet<T> implements Predicate<T> {
    public Set<T> set;
       public TestSet(Collection<? extends T> c) { this.set = new HashSet<>(c); }
    @Override public boolean test(T t) { return this.set.contains(t); }
}

class STools {
    public static <T> Stream<T> filterOr(Stream<? extends T> s,
                                  Predicate<? super T>... ps) {

        Predicate<T> combinedPred  = x -> false;
        for (Predicate<? super T> p : ps)
                 combinedPred  = combinedPred.or(p);
        return (Stream<T>) s.filter( combinedPred );
    }
    public static  <T> Stream<T> filterSets(Stream<T> s,
                                            Set<? super T>... sets) {
        return s.filter( t -> { for (Set<? super T> p : sets)
                                     if (p.contains(t)) return true;
                                return false; } );
    }
}
```