EVALUACION NO CONTINUA

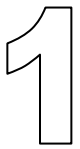
Análisis y Diseño de Software (2013/2014)

Responde a cada apartado en hojas separadas

Apartado 1. (2 puntos). Colecciones

Dadas las siguientes clases Java, **se pide** completar los espacios señalados (cuando sea necesario hacerlo) para que el método `main` produzca la salida indicada abajo. El propósito de la clase `ConexionesAereas` es almacenar, en orden alfabético, los nombres de las aerolíneas que ofrecen vuelos directos entre cada dos aeropuertos dados. Para cada trayecto directo entre dos aeropuertos, debe evitarse almacenar por duplicado la misma información para el trayecto en sentido inverso, es decir, intercambiando aeropuerto origen y aeropuerto destino. Además, según muestra la salida esperada del programa, la información de todas las conexiones aéreas almacenadas se mostrará ordenada por los trayectos directos almacenados (primero por aeropuerto de origen y después por aeropuerto de destino), y las aerolíneas que sirven cada trayecto, también han de presentarse por orden alfabético. La clase `ConexionesAereas` debe también tener un método para borrar una aerolínea de un trayecto.

```
public enum Aeropuerto { BCN, CDG, JFK, MAD; }
public class TrayectoDirecto { // completar la clase (1) si es necesario
    private Aeropuerto origen, destino;
    public Aeropuerto getOrigen() { return origen; }
    public Aeropuerto getDestino() { return destino; }
    public String toString() { return "(" + origen + "<>" + destino + ")"; }
```



```
} // end clase TrayectoDirecto


public class ConexionesAereas { // completar la clase (2) si es necesario
```

```
} // end clase ConexionesAereas


public class Ejercicio1 {
  public static void main(String[] args) {
       ConexionesAereas c = new ConexionesAereas();
       c.add( new TrayectoDirecto( Aeropuerto.MAD, Aeropuerto.JFK ), "NeverCrash" );
       c.add( new TrayectoDirecto( Aeropuerto.JFK, Aeropuerto.MAD ), "NeverCrash" );
       c.add( new TrayectoDirecto( Aeropuerto.MAD, Aeropuerto.JFK ), "EspaFlai" );
       c.add( new TrayectoDirecto( Aeropuerto.MAD, Aeropuerto.BCN ), "NeverCrash" );
       c.add( new TrayectoDirecto( Aeropuerto.BCN, Aeropuerto.MAD ), "EspaFlai" );
       c.add( new TrayectoDirecto( Aeropuerto.CDG, Aeropuerto.JFK ), "SubidonFree" );
       c.add( new TrayectoDirecto( Aeropuerto.JFK, Aeropuerto.CDG ), "FlaiJai" );
       c.add( new TrayectoDirecto( Aeropuerto.MAD, Aeropuerto.BCN ), "EspaFlai" );
       c.add( new TrayectoDirecto( Aeropuerto.BCN, Aeropuerto.MAD ), "EspaFlai" );

       if (c.remove(new TrayectoDirecto( Aeropuerto.JFK, Aeropuerto.CDG ), "FlaiJai"))
             System.out.println("FlaiJai borrado");
       System.out.println(c);
  } // end main
} // end clase Ejercicio1
```

Salida esperada:

```
FlaiJai borrado
{(CDG<>JFK)=[SubidonFree], (MAD<>BCN)=[EspaFlai, NeverCrash], (MAD<>JFK)=[EspaFlai, NeverCrash]}
```

Apartado 1. Colecciones 2 puntos

```
package examen.colecciones.continua;  /* sombreado sólo afecta a versión de Ev. No Continua */
import java.util.*;
public enum Aeropuerto { BCN, CDG, JFK, MAD; }

public class TrayectoDirecto implements Comparable<TrayectoDirecto> { /* para usar la clase como clave del TreeMap */
    private Aeropuerto origen, destino;
    public Aeropuerto getOrigen() { return origen; }
    public Aeropuerto getDestino() { return destino; }
    public String toString() { return "(" + origen + "<>" + destino + ")"; }
    public TrayectoDirecto(Aeropuerto origen, Aeropuerto destino) {
        this.origen = origen; this.destino = destino;
    }
    @Override       /* método IMPRESCINDIBLE ya que los objetos de clase Trayecto se usarán como clave en un Map */
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (! (obj instanceof TrayectoDirecto)) return false;
        TrayectoDirecto t = (TrayectoDirecto) obj;

        return getOrigen().equals(t.getOrigen()) &&  getDestino().equals(t.getDestino())
           ||  getOrigen().equals(t.getDestino()) &&  getDestino().equals(t.getOrigen());
    }
    @Override  /* método IMPRESCINDIBLE para mantener coherencia entre la igualdad y el hashing usado en el Map */
    public int hashCode() {
        return getOrigen().hashCode() + 101 * getDestino().hashCode();
    }
    @Override  /* método IMPRESCINDIBLE para ordenar objetos de la clase Trayecto usados como clave del TreeMap */
    public int compareTo(TrayectoDirecto t) {
        if ( this.equals(t) ) return 0;
        else if  (getOrigen().equals(t.getOrigen())) return getDestino().compareTo(t.getDestino());
        else return getOrigen().compareTo(t.getOrigen());
    }
}

public class ConexionesAereas {
        private Map<TrayectoDirecto, Set<String>> conexiones = new TreeMap<TrayectoDirecto,Set<String>>();

        public void add(TrayectoDirecto trayecto, String aerolinea) {
                if (! conexiones.containsKey(trayecto))   conexiones.put(trayecto, new TreeSet<String>());
                conexiones.get(trayecto).add(aerolinea);
        }

        public String toString() { return conexiones.toString(); }

        // AÑADIDO SOLO EN VERSION DE NO CONTINUA  ************************
        public boolean remove(TrayectoDirecto trayecto, String aerolinea) {
                if  (conexiones.containsKey(trayecto))
                        return conexiones.get(trayecto).remove(aerolinea);
                return false;
        }
        // ***************************************************************
}

public class Ejercicio1NoContinua {
        public static void main(String[] args) {
                ConexionesAereas c = new ConexionesAereas();
                c.add( new TrayectoDirecto( Aeropuerto.MAD, Aeropuerto.JFK ), "NeverCrash" );
                c.add( new TrayectoDirecto( Aeropuerto.JFK, Aeropuerto.MAD ), "NeverCrash" );
                c.add( new TrayectoDirecto( Aeropuerto.MAD, Aeropuerto.JFK ), "EspaFlai" );
                c.add( new TrayectoDirecto( Aeropuerto.MAD, Aeropuerto.BCN ), "NeverCrash" );
                c.add( new TrayectoDirecto( Aeropuerto.BCN, Aeropuerto.MAD ), "EspaFlai" );
                c.add( new TrayectoDirecto( Aeropuerto.CDG, Aeropuerto.JFK ), "SubidonFree" );
                // sgte línea, SOLO EN Version NO Continua
                c.add( new TrayectoDirecto( Aeropuerto.JFK, Aeropuerto.CDG ), "FlaiJai" );
                c.add( new TrayectoDirecto( Aeropuerto.MAD, Aeropuerto.BCN ), "EspaFlai" );
                c.add( new TrayectoDirecto( Aeropuerto.BCN, Aeropuerto.MAD ), "EspaFlai" );
                // sgte instrucción, SOLO EN Version NO Continua
                if (c.remove(new TrayectoDirecto( Aeropuerto.JFK, Aeropuerto.CDG ), "FlaiJai"))
                        System.out.println("FlaiJai borrado");
                System.out.println(c);
        }
}
```

Salida esperada:

```
FlaiJai borrado     /* esta línea solamente en versión No Continua */
{(CDG<>JFK)=[SubidonFree], (MAD<>BCN)=[EspaFlai, NeverCrash], (MAD<>JFK)=[EspaFlai, NeverCrash]}
```

Apartado 2. (2 puntos). Java Básico

Se desea diseñar y programar una aplicación Java para gestionar los pedidos y facturación de una pizzería a domicilio con los siguientes requisitos. Hay tres tipos de productos Pizza, Complemento y Bebida. Todos los productos tienen un nombre o descripción. Por otro lado, cada producto, según sea su tipo y su composición, tendrá una forma concreta de calcular su coste, así como unas reglas para saber si se trata de un producto de especialidad, y otras para saber si se trata de un producto promocional. Concretamente, las pizzas nunca son promocionales, y una pizza de especialidad se crea mediante nombre y precio exclusivamente, pero una pizza que no sea de especialidad se puede crear con una masa y una lista de ingredientes (de tamaño arbitrario), y su precio se calculará como 2.00€ por la masa más 0.50€ por cada ingrediente. Las bebidas se venden por unidades con su precio unitario pero con un descuento fijo del 50% a partir de la sexta unidad (inclusive); además las bebidas nunca son de especialidad pero siempre son promocionales. Un complemento se crea siempre con un nombre y precio; se le considera de especialidad si el precio es superior a 5.5€, y promocional si el precio es inferior a 2.5. Los pedidos se crean vacíos pero se les puede añadir y eliminar productos (previamente configurados) hasta que se cierra el pedido. Una vez cerrado un pedido no se pueden añadir ni eliminar productos al mismo. Al cerrar un pedido, se calcula y devuelve su precio final (con 10% de IVA incluido en la facturación). Antes de aplicar el IVA, el precio final de cualquier pedido se calcula sumando el coste de todos su productos pero aplicándole un descuento del 50% sobre el coste de todos los productos promocionales cuando el coste total de los productos de especialidad sea superior a 7.00€.

Se pide: el código Java de las clases Java necesarias para que el programa de abajo produzca la salida esperada.

```
public class Ejercicio2 {
   public static void main(String[] args) {    // Producto.TRACE = true;
        Producto p1 = new Pizza(TipoMasa.FINA,"jamón", "ternera", "cebolla", "pimiento");
        Producto b1 = new Bebida("refresco", 9, 1.0);
        Producto p2 = new Pizza("jaguayana", 6.25);
        Producto b2 = new Bebida("cerveza", 4, 1.25);
        Producto c1 = new Complemento("alitas", 7.75);
        Producto c2 = new Complemento("helado", 2.25);

        // ejemplos de uso de productos
        System.out.println(p1 + " promocional: " + p1.promocional());
        System.out.println(b1 + " coste: " + b1.coste()); // con descuento por cantidad
        System.out.println(c1 + " especialidad: " + c1.especialidad());
        System.out.println(c2 + " promocional: " + c2.promocional());


        // creación de pedidos: pe1 con descuento a promocionales, pe2 sin dto. promocionales
        Pedido pe1 = new Pedido(),  pe2 = new Pedido();
        double precio1 = pe1.añadir(p1).añadir(b1).añadir(p2).añadir(c1).añadir(b2).cerrar();
        System.out.println("precio1 = "+precio1);

        pe2.añadir(p1).añadir(b1).añadir(p2).añadir(c2);

        // facturación
        System.out.println(Pedido.facturacion());
        pe2.cerrar();
        System.out.println(Pedido.facturacion());

    }// end método main
}// end clase Ejercicio2
```

Salida espera:

```
masa FINA con:[jamón, ternera, cebolla, pimiento]=4.0 promocional: false
refresco=7.0 coste: 7.0
alitas=7.75 especialidad: true
helado=2.25 promocional: true
precio1 = 26.40
Total pedidos = 26.40
Total pedidos = 47.85
```

Apartado 2. Java Básico 2 puntos.

```
public class Pedido {
        private static final Double TASA_IVA = 0.10;
        private static final Double MIN_ESPECIALIDADES_PARA_DESCUENTO = 7.0;
        private static final Double PORCENTAJE_DESCUENTO_PROMOCION = 50.0;

        private static Double totalPedidos = 0.0;

        private List<Producto> productos = new ArrayList<Producto>();
        private boolean cerrado = false;

        public Pedido añadir(Producto p) { if (! cerrado) productos.add(p); return this; }
        public Pedido eliminar(Producto p) {  if (! cerrado)  productos.remove(p); return this; }

        public static String facturacion() { return "Total pedidos = " + totalPedidos; }

        public double cerrar() {
                double precioFinal = 0.0;
                if (! cerrado) { precioFinal = precioFinal(); cerrado = true; totalPedidos += precioFinal;  }
                return precioFinal;
        }

        private Double precioFinal() {
                Double total = 0.0;
                for (Producto p : productos) total += p.coste();
                if ( costeEspecialidades() > MIN_ESPECIALIDADES_PARA_DESCUENTO)
                        total = total - PORCENTAJE_DESCUENTO_PROMOCION * costePromocionales() / 100.0;
                return total * (1 + TASA_IVA);
        }

        private  Double costeEspecialidades() {
           Double total = 0.0; for (Producto p : productos) if (p.especialidad()) total += p.coste(); return total; }
        private  Double costePromocionales() {
            Double total = 0.0; for (Producto p : productos) if (p.promocional()) total += p.coste(); return total; }
}
```

```
public abstract class Producto  {
        private final String descripcion;
        private final double precio;
        abstract public boolean especialidad();
        abstract public boolean promocional();
        public final Double coste() { return precio; }
        public Producto(String desc, double precio) { descripcion = desc; this.precio = precio; }
        public final String toString() { return descripcion + "=" + this.coste(); }
}
```

```
public enum TipoMasa { CLASICA, FINA,  PAN; }
```

```
public class Pizza extends Producto {
    private static final Double PRECIO_MASA = 2.0;
    private static final Double PRECIO_INGREDIENTE= 0.5;
    private boolean especialidad;
    public Pizza(String nombre, Double precio) { super(nombre, precio);  this.especialidad = true; }

    public Pizza(TipoMasa masa, String... ingredientes) {              /* con número variable de ingredientes  */
        super("masa " + masa + " con:" +Arrays.asList(ingredientes).toString(),   /* llamada correcta a super   */
                PRECIO_MASA + PRECIO_INGREDIENTE* ingredientes.length);
        this.especialidad = false;
    }
        public boolean especialidad() { return especialidad; }  /* incluyendo incialización correcta  */
        public boolean promocional() { return false; }
}
```

```
public class Bebida extends Producto {
        private static final Integer UNIDADES_SIN_DESCUENTO = 5;
        private static final Double DESCUENTO_POR_UNIDADES= 0.5;

        public Bebida(String nombre, int unidades, Double precioUnidad) {
           super(nombre,  precioUnidad * unidades - DESCUENTO_POR_UNIDADES* Math.max(0, unidades -
UNIDADES_SIN_DESCUENTO));
        }                  /*   cálculo correcto del precio con  descuento por unidades */
        public boolean especialidad() { return false; }
        public boolean promocional() { return true; }
}
```

```
public class Complemento extends Producto {
        private static final Double COSTE_MAX_PARA_PROMOCIONAL = 2.5;
        private static final Double COSTE_MIN_PARA_ESPECIALIDAD = 5.5;
        public Complemento(String nombre, Double precio) { super(nombre,precio); }
        public boolean especialidad() { return coste() > COSTE_MIN_PARA_ESPECIALIDAD; }
        public boolean promocional() { return coste() < COSTE_MAX_PARA_PROMOCIONAL; }
}
```

Apartado 3. (2 puntos). Genericidad y Excepciones

Se quiere construir una clase de utilidad `CombinaSeries` que almacene un número variable de series de elementos. A su vez, cada serie tiene un número arbitrario de elementos. Todas las series tienen elementos del mismo tipo, y cada serie de `CombinaSeries` es de un tipo base, o un subtipo. El programa ha de diseñarse para permitir utilizar la clase `CombinaSeries` con distintos tipos de manera flexible. La clase debe lanzar una excepción si se intenta añadir una serie de tamaño 0.

Se pide:
Diseñar la clase `CombinaSeries`, la excepción `SerieVaciaExcepcion` y completar el siguiente programa, para producir la salida de más abajo.

```
public class Ejercicio3 {

    public static void main(String[] args) {
       CombinaSeries<Number> cs = new CombinaSeries<Number>();

       _____1_____ {                                    // completar 1
             cs.addSerie(Arrays.asList(0,  1, 2));
             cs.addSerie(Arrays.asList(3.4, 4.6, 7.7, 1.1));
             cs.addSerie(Arrays.asList(5L,  6L));
             cs.addSerie(new ArrayList<Integer>());
       } _____2_____ ( SerieVaciaExcepcion sv ) {    // completar 2
             System.out.println(sv);
       }

       List<_____3_____> serie0 = cs.get(0);          // completar 3
       System.out.println(serie0);
       System.out.println(cs);
    }
}
```

```
Error: se intentó añadir una serie vacía en la posición 4
[0, 1, 2]
Series:
[0, 1, 2]
[3.4, 4.6, 7.7, 1.1]
[5, 6]
```

```java
public class SerieVaciaExcepcion extends Exception {
	private int pos;
	public SerieVaciaExcepcion(int pos) { this.pos = pos; }
	public String toString() {
	   return "Error: se intentó añadir una lista vacia en la posición " + pos;
	}
}
```

```java
import java.util.*;
public class CombinaSeries<T extends Number> {
	private List<List<? extends T>> contenido = new ArrayList<List<? extends T>>();

	public void addSerie(List<? extends T> serie) throws SerieVaciaExcepcion {
		if (serie.size() == 0) throw new SerieVaciaExcepcion( contenido.size()+1 );
		contenido.add(serie);
	}
	public List<? extends T> get(Integer indice) { return contenido.get(indice); }

	public String toString() {
		System.out.println("Series:");
		for (List<? extends T> s : contenido) System.out.println(s);
		return "";
	}
}
```

```java
import java.util.*;
public class Ejercicio3 {

	public static void main(String[] args) {
		CombinaSeries<Number> cs = new CombinaSeries<Number>();

		try {                                                    // completar 1
			cs.addSerie(Arrays.asList(0, 1, 2));
			cs.addSerie(Arrays.asList(3.4, 4.6, 7.7, 1.1));
			cs.addSerie(Arrays.asList(5L,  6L));
			cs.addSerie(new ArrayList<Integer>());

		} catch ( SerieVaciaExcepcion sv ) {                    // completar 2
			System.out.println(sv);
		}

		List< ? extends Number > serie0 = cs.get(0);             // completar 3
		System.out.println(serie0);
		System.out.println(cs);
	    }

}
```

Apartado 4. (2 puntos). Patrones de diseño

Continuando el ejercicio anterior, queremos obtener la lista de elementos resultante de recorrer en paralelo las series que contiene un objeto `CombinaSeries`. Para ello, construiremos una clase `RecorreParalelo`, que facilite realizar dicho recorrido, y devuelva una lista (con un elemento de cada una de las series) con los elementos actuales del recorrido. Si una serie tiene menos elementos, se usa un elemento de relleno configurable (de un tipo compatible), como se muestra en el siguiente listado. La clase `RecorreParalelo` debe diseñarse de tal manera que permita añadir nuevas series durante la iteración.

Se pide:

a) Utilizando los patrones de diseño más adecuado, diseña la clase `RecorreParalelo` (modificando o extendiendo la clase `CombinaSeries` del ejercicio anterior, si es necesario), y completa la línea de puntos (si es necesario) para que el siguiente programa produzca la salida de más abajo.

b) ¿Qué patrón o patrones de diseño has usado?

```
public class Ejercicio4 {
  public static void main(String[] args) ..... { // completar 1
       CombinaSeries<Number> cs = new CombinaSeries<Number>();

       cs.addSerie(Arrays.asList(0,  1, 2));
       cs.addSerie(Arrays.asList(3.4, 4.6, 7.7, 1.1));
       cs.addSerie(Arrays.asList(5L,  6L));

       RecorreParalelo<Number> is = cs.getRecorreParalelo(42); // 42 es valor relleno

       int index = 0;
       while (is.hasNext()) {
         index++;
         if (index == 2) cs.addSerie(Arrays.asList(2, 1, 0)); // en 2ª iteración añadimos
                                                              // una nueva serie
         System.out.println(is.current());
         is.next();
       }
       System.out.println(is.current());
  }
}
```

Salida:

```
[0, 3.4, 5]
[1, 4.6, 6, 1]
[2, 7.7, 42, 0]
[42, 1.1, 42, 42]
```

```java
import java.util.*;
public class RecorreParalelo<T extends Number> { // Esta solución se base en el patrón ITERATOR

	private List<List<? extends T>> contenido = new ArrayList<List<? extends T>>();

	private List<Iterator<? extends T>> iteradores = new ArrayList<Iterator<? extends T>>();
	private List<T> actual = new ArrayList<T>();
	private T relleno;
	private int maxSize = 0;
	private int indice = 0;

	public RecorreParalelo(List<List<? extends T>> contenido, T relleno) {
		this.contenido = contenido;
		this.relleno = relleno;
		this.indice = 0;
		for (List<? extends T> s : this.contenido) {
			if (s.size() > maxSize) maxSize = s.size();
		}
	}

	public List<? extends T> current() {
		actual.clear();
		if (  indice == maxSize ) return actual;
		for (List<? extends T> s : this.contenido) {
			if (s.size() <= indice) actual.add( relleno );
			else actual.add( s.get(indice) );
		}
		return actual;
	}

	public boolean hasNext() {	return indice < maxSize - 1;	}

	public void next() {
	    if (hasNext()) indice++;
    }
}
```

```
La clase SerieVaciaExcepcion del apartado anterior no cambia en nada
```

```java
// añadir este método a la clase CombinaSeries del apartado anterior
	public RecorreParalelo<T> getRecorreParalelo(T relleno) {
		return new RecorreParalelo<T>(contenido, relleno);
	}
```

```java
public class Ejercicio4v2 {
	public static void main(String[] args) throws SerieVaciaExcepcion { // completar
		CombinaSeries<Number> cs = new CombinaSeries<Number>();

		cs.addSerie(Arrays.asList(0,  1, 2));
		cs.addSerie(Arrays.asList(3.4, 4.6, 7.7, 1.1));
		cs.addSerie(Arrays.asList(5L,  6L));

		RecorreParalelo<Number> is = cs.getRecorreParalelo(42); // 42 es valor relleno

		int index = 0;
		while (is.hasNext()) {
		  index++;
		  if (index == 2) cs.addSerie(Arrays.asList(2, 1, 0)); // en 2ª iteración añadimos
		                                                       // una nueva serie
		  System.out.println(is.current());
		  is.next();
		}
		System.out.println(is.current());
	}
}
```

Apartado 5. (2 puntos) Diagramas de clase

Se desea diseñar un sistema de gestión de archivos en internet, que permita a los usuarios compartir contenido de distinto tipo con otros usuarios.

Cada usuario tiene un directorio raíz, la raíz de su sistema de archivos, que no puede ser compartido con otros usuarios, ni estar contenido en otro directorio o borrarse. El sistema de archivos tiene una estructura jerárquica, de forma que cada directorio contiene una serie de archivos y/o directorios.

Los elementos, directorios o archivos, pueden estar en varios directorios simultáneamente. La excepción son los directorios raíz de cada usuario, que no están contenidos en ningún directorio, y se generará una excepción si se intentan incluir en otro directorio.

De esta forma, un mismo usuario puede tiene varias veces el mismo directorio o archivo, en sitios diferentes de sus sistema de archivos. Un directorio o archivo puede estar compartido, por ejemplo si está en directorios de distintos usuarios.

Un usuario sólo puede mover un elemento de su sistema de archivos a otro directorio al que tenga acceso, y que por lo tanto esté en su sistema de archivos. Cuando un usuario quiere compartir un archivo con otro usuario, el sistema lo coloca en la raíz del sistema de archivos de ese nuevo usuario.

Además de los atributos necesarios para modelar el sistema:

- Los usuarios tienen un nombre, una dirección de correo electrónico y una contraseña.
- Cada archivo o directorio tiene un nombre, una fecha de creación, y una fecha de última modificación.
- Cada archivo tiene también un contenido binario, en forma de Array de bytes.

Se pide:

(a) Representa en UML las clases y relaciones del sistema anterior. En el diagrama no es necesario incluir los métodos o constructores.

(b) Para las siguientes operaciones, indica la clase más apropiada para contenerla, su nombre, argumentos, tipo de resultado y excepciones, en caso de poder generarlas.

a. Compartir un archivo o directorio con otro usuario
b. Incluir un elemento en un directorio

Nota: Solo se pide el diseño de la lógica del sistema, sin interfaz de usuario, ni código.