Examen Final de Convocatoria Extraordinaria

Análisis y Diseño de Software (2010/2011)

Contesta cada apartado en hojas separadas

Apartado 1. (1.5 puntos)

Indica razonadamente la salida que produciría el siguiente programa, tras explicar los errores de compilación o de ejecución que pueda contener y eliminar las líneas que los produzcan.

```
class Mamifero {
        public String nombre;
        public Mamifero(String nombre) {
           this.nombre = nombre;
        }
        public Mamifero() {}

        public String hacerRuido() { return " gruñe "; }
        public String getNombre() { return nombre; }
}

class Elefante extends Mamifero {
        String nombre = " Elefante con trompa ";
        public String hacerRuido() { return " barrita "; }
}

public class PruebaZoo {

        public static void main(String[] args) {
           new PruebaZoo().vamos();
        }

        void vamos() {
           Mamifero n = new Mamifero("perro");
           System.out.println(n.nombre + n.hacerRuido() + n.getNombre());
           Mamifero m = new Elefante();
           System.out.println(m.nombre + m.hacerRuido() + m.getNombre());
           Elefante p = new Elefante();
           System.out.println(p.nombre + p.hacerRuido() + p.getNombre());
           Elefante q = (Elefante) new Mamifero("lobo");
           System.out.println(q.nombre + q.hacerRuido() + q.getNombre());

        }
}
```

Contesta cada apartado en hojas separadas

Apartado 2. (3.5 puntos)

Completa el siguiente programa para que su salida sea la que se indica más abajo. El objetivo es calcular las comisiones a pagar a los vendedores por el importe de sus ventas. Los vendedores, identificados por su nombre completo, pueden ser solamente de dos tipos: fijos y temporales. Cada vendedor fijo tendrá asignado un porcentaje de comisión sobre sus ventas, que se aplicará desde el primer Euro de venta. Cada vendedor temporal tendrá un porcentaje de comisión (posiblemente más alto) y un importe mínimo de ventas, de forma que no se le pagará ninguna comisión hasta que no haya *acumulado ese importe mínimo de ventas y a partir de ese importe se aplicará el porcentaje de comisión asignado*. Además, cada vendedor puede estar asociado a otro vendedor, su jefe de zona que cobrará comisiones por todas las ventas realizadas por los vendedores de su zona; cada vez que se calcule una comisión a pagar a un vendedor, se le sumará a su jefe de zona (si lo tiene) un 2,5% sobre dicha comisión en concepto de comisión de zona. Como se indica abajo, el programa deberá poder imprimir los datos de todos los vendedores (hayan o no realizado ventas), y también imprimir los vendedores dependientes de un jefe de zona. Las comisiones se calcularán y acumularán según se vayan realizando ventas, para que en cualquier momento se pueda solicitar la generación de comisiones a pagar hasta el momento, y a partir de se instante se deberán restablecer a cero todos los totales acumulados de ventas y comisiones.

```
public class Apartado2 {
	public static void main(String[] args) throws VendedorYaExisteException {
                        // Comision 0.3% sobre ventas acumuladas por encima de 10000
		Vendedor v1 = new VendedorTemporal("Luis Moreno", 0.3, 10000);
                        // Comision 0.2% desde el primer Euro de venta
		VendedorFijo v2 = new VendedorFijo("Ricardo Paz", 0.2);
                         // Comision 0.15% desde el primer Euro de venta
		Vendedor v3 = new VendedorFijo("Manuel Gomez", 0.15);
                        // Comision 0.5% sobre ventas acumuladas por encima de 20000
		VendedorTemporal v4 = new VendedorTemporal("Mario Ruiz", 0.5, 20000);
		Vendedor v5 = new VendedorFijo("Alvaro Lopez", 0.4);
		try {
		   Vendedor.imprimeVendedores();
		   // El vendedor v3 acumulara 0.05% de comision por cada venta en su zona
		   v2.asignarJefeZona(Vendedor.llamado("Manuel Gomez"));
		   v1.asignarJefeZona(Vendedor.llamado("Manuel Gomez"));
		   System.out.println(v3.getMisVendedores()); System.out.println();

		   Comisiones c = new Comisiones();
		   c.ventaRealizada("Mario Ruiz", 11000); // no acumula comision todavia
		   c.ventaRealizada("Mario Ruiz", 15000); // acumula 0.5% comision sobre 6000 EUR
		   c.ventaRealizada("Luis Moreno", 5000); // no acumula comision todavia
		   c.ventaRealizada("Luis Moreno", 7000); // acumula 0.3% comision sobre 2000 EUR
		   c.ventaRealizada("Ricardo Paz ", 1000); // acumula 0.2% comision sobre 1000 EUR

		   c.imprimirComisionesVendedor("Manuel Gomez");
		   c.pagarComisiones();
		   c.pagarComisiones();
		   c.imprimirComisionesVendedor("John Smith");
		} catch (VendedorNoExisteException e) { System.out.println(e.getMessage()); }
	}
}
```

Salida:

```
Lista de vendedores:
Ricardo Paz con comision 0.2% fijo.
Luis Moreno con comision 0.3% temporal, con 10000 de venta minima.
Manuel Gomez con comision 0.15% fijo.
Mario Ruiz con comision 0.5% temporal, con 20000 de venta minima.
Alvaro Lopez con comision 0.4% fijo.

[Ricardo Paz con comision 0.2% fijo., Luis Moreno con comision 0.3% temporal y 10000 de venta minima.]

Manuel Gomez con comision 0.15% fijo. Total comisiones:  0,20

Comisiones a pagar (en orden alfabético):
Luis Moreno con comision 0.3% temporal, con 10000 de venta minima. Total comisiones:  6,00
Manuel Gomez con comision 0.15% fijo. Total comisiones:  0,20
Mario Ruiz con comision 0.5% temporal, con 20000 de venta minima. Total comisiones: 30,00
Ricardo Paz con comision 0.2% fijo. Total comisiones:  2,00

No hay comisiones pendientes de pagar.
Nombre de vendedor no existe: John Smith
```

Contesta cada apartado en hojas separadas

Apartado 3. (2.5 puntos)

Frecuentemente las comunicaciones entre procesos son más eficientes cuando se intercambian datos por lotes. Por ejemplo, es común obtener bloques de datos al leer de disco duro (en bloques o sectores en lugar de bytes), o al acceder a un servicio web donde se obtiene una lista parcial de resultados, ya que la lista completa puede requerir demasiado tiempo y/o memoria.

Este comportamiento se puede modelar de forma sencilla con una interfaz en Java, `BatchList<T>`, que representa una lista de tipo T (tienes la definición de dicha interfaz más abajo). La interfaz contiene dos métodos: uno que devuelve el tamaño real de la lista, y otro método (`getListFrom`) que devuelve una lista parcial de resultados desde la posición `from`. El parámetro `from` puede tomar valores entre `0` y `size()-1`, y en caso de que se exceda de los límites el método `getListFrom` lanzará la excepción `IndexOutOfBoundsException`. El tamaño del resultado parcial no es determinante, aunque normalmente será mucho menor que el tamaño total de la lista completa, que puede estar en disco, en la red, etc.

Fichero BatchList.java

```
public interface BatchList<T>{
  int size();
  List<T> getListFrom(int from) throws IndexOutOfBoundsException;
}
```

Con el objetivo de simplificar al máximo la gestión de datos, lo más sencillo es crear un adaptador, que permita usar los objetos `BatchList` como listas normales de Java, y leer sus elementos de uno en uno. De esta forma se puede trabajar con los resultados como una lista estándar, con la ventaja de no tener que obtener todos los elementos antes de procesarlos, lo cual podría ser una operación demasiado costosa en tiempo y memoria.

Se pide:

Implementa una clase llamada `ListAdapter<T>`, como subclase de `AbstractList<T>`, para crear una lista de sólo lectura que permita obtener los elementos de una `BatchList<T>` de uno en uno. `ListAdapter` tendrá un constructor con un solo argumento, un objeto `BatchList`, que utilizará para obtener la información. Hay que tener en cuenta que usando la interfaz `List` los elementos se obtendrán de uno en uno, mientras que `BatchList` los devuelve por lotes, por lo tanto se ha de optimizar `ListAdapter` para no tener que obtener un nuevo lote de resultados si el elemento buscado ya estaba en el último lote obtenido.

Nota: Como recordarás, `AbstractList` es una clase predefinida de la librería de colecciones de Java. Es una clase abstracta que permite implementar la interfaz `List` de forma sencilla, mediante subclasificación. Para el caso de listas de solo lectura, únicamente requiere implementar `get(int)` y `size()`. A continuación se presenta un resumen de `AbstractList.java`:

Fichero AbstractList.java

```
package java.util;

public abstract class AbstractList<E> implements List<E> {

  // Método de utilidad, lanza una excepción del tipo indexOutOfBoundsException
  protected static void indexOutOfBounds(int index, int size) {
    throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size);
  }

  protected AbstractList() {}  //Constructor vacío

  public abstract E get(int index);
  public abstract int size();
  …
  …
 /* Se ha ocultado el resto de código, que implementa los métodos necesarios
  * para crear una lista de solo lectura basada en los métodos abstractos get(int) y size().
  * Al intentar modificar la lista se lanzan excepciones del tipo UnsupportedOperationException
  */
}
```

Contesta cada apartado en hojas separadas

Apartado 4. (2.5 puntos)

Se quiere construir una aplicación que simule edificios inteligentes. Estos son edificios que contienen sensores de diversos tipos (por ejemplo de luz, de movimiento) que se pueden configurar para que realicen ciertas acciones (por ejemplo, cuando haya poca luz exterior, encender la luz de la habitación y la televisión).

Para ello, se quiere construir un diagrama de clases UML que modele una versión simplificada de dicha aplicación. De esta manera, vamos a considerar que una habitación (que suponemos rectangular, con ciertas dimensiones) puede tener asociado un número arbitrario de dispositivos. Consideraremos dos tipos de dispositivos: *sensores* y *actuadores*. Los primeros son dispositivos que reciben un estímulo del exterior (mediante un método `setEntrada(valor:double)` que recibe la "fuerza" o valor de dicho estímulo de entrada) y lo almacenan. Por simplicidad consideraremos sólo dos tipos de sensores: de luz y de movimiento.

Los actuadores son dispositivos que pueden realizar alguna acción. Por simplicidad, consideraremos dos tipos: lámpara y televisión. Para mayor flexibilidad, todos los actuadores deben devolver una lista de propiedades. Así, la televisión deberá devolver el volumen y el canal, mientras que la lámpara debe devolver su luminosidad. De manera adicional, debe ser posible modificar el valor de estas propiedades mediante un método setter. Todos los dispositivos deben poder encenderse y apagarse.

Para configurar las acciones del edificio inteligente, simplemente consideraremos que podemos asociar actuadores a los sensores, y que los sensores se pueden configurar con un valor umbral. De esta manera, cada vez que el sensor recibe una invocación al método `setEntrada`, si el valor del parámetro es mayor que el umbral, el sensor encenderá los actuadores asociados que no estén encendidos. Cuando el valor de entrada tenga un valor menor que el umbral, se apagarán los actuadores asociados que no estuvieran ya apagados. Un actuador puede asociarse a más de un sensor, y tu diseño debe soportar asociar y quitar actuadores de los sensores.

Se pide:

a) El **diagrama de clases** UML que modele el enunciado. No olvides añadir los **métodos** necesarios para obtener la funcionalidad que se describe en el enunciado.

b) El **código Java** del método `setEntrada` de los sensores.