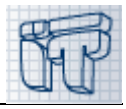


NOMBRE:
APELLIDOS:
NIA:
GRUPO:

Examen Final Extraordinario

2ª Parte: Problemas (7 puntos sobre 10)

Duración: 180 minutos
Puntuación máxima: 7 puntos
Fecha: 28 junio 2018

Instrucciones para el examen:

- No se permite el uso de libros o apuntes, ni tener teléfonos móviles u otros dispositivos electrónicos encendidos. Incumplir cualquiera de estas normas puede ser motivo de expulsión inmediata del examen.
- Rellena tus datos personales antes de comenzar a realizar el examen.
- El examen debe rellenarse con bolígrafo azul o negro. No está permitido entregar el examen con lapicero.

Problema 1 (2 / 7 puntos)

Apartado 1 (0,5 puntos)

En una compañía telefónica han desarrollado un nuevo servicio de televisión por cable. Como miembro de la compañía, estás encargado del diseño de algunas clases en Java para modelar este servicio. En primer lugar, se quieren modelar los programas de la televisión mediante la clase `Programme`. Un programa incluye su nombre (`name`), una cadena de texto para indicar la fecha y hora en la que se emite (`time`) y la calidad en la que se emite (`quality`). Todos los atributos podrán ser visibles en cualquier clase dentro del paquete, pero no podrán ser accedidos desde clases en otro paquete. Respecto a la calidad, ésta se modelará con constantes, que pueden tomar los siguientes valores:

- `SD = 1` `HD = 2`

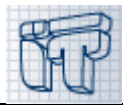
Esta clase tendrá un constructor que recibe por parámetro los valores iniciales de los atributos. En caso de que el parámetro que se refiere a la calidad no tenga ningún valor de los propuestos, se deberá asignar la calidad SD automáticamente. Por otro lado, la clase dispone de un método `void description()` para indicar la descripción del programa, aunque en esta clase no se dispone de información suficiente para saber cómo debe ser el código del método `description()`, ya que la descripción dependerá del tipo de programa. En la plataforma se definen cuatro tipos de programas: `News`, `Sport`, `Film` y `Series`. Sin embargo, el código de los tipos de programas será programado por otro equipo. En este apartado, **se pide escribir el código únicamente** de la clase `Programme` con las especificaciones dadas.

Apartado 2 (0,7 puntos)

Por otro lado, estás encargado de la gestión de los clientes. Inicialmente se incluirán dos funcionalidades, que se incluyen en la siguiente interfaz, que **debe ser implementada por todos los tipos de clientes**:

```
public interface Functions {
      void record(String name) throws TVException;
      void generateIncidence(String incidencia);
}
```

Dado que no todos los clientes pueden contratar la televisión, se definen dos tipos de clientes: `Client` y `ClientTV`. El primero de ellos **no** estará suscrito a la televisión (puede tener fijo, móvil, etc., pero no TV). Este cliente se identificará por un identificador (`id`) que le otorgará la plataforma de forma incremental (el primer cliente tendrá id 1, el segundo id 2 y así sucesivamente), un nombre (`name`), que será una cadena de caracteres y, además, un `ArrayList` de cadenas de texto para almacenar sus incidencias. Todos los atributos solamente serán visibles dentro de la clase. En este apartado se pide escribir el código de la clase `Client` en la que, aparte de declarar los atributos, deberás incluir al menos:



- Un constructor en el que se inicialicen los atributos. Este constructor solo recibe por parámetro el nombre del cliente.
- El método `record(String name)`, que en este caso lanzará directamente una `TVException` (puedes suponer que esta clase ya está programada) con el mensaje `"Recordings are not available in your subscription"`.
- El método `generateIncidence(String incidencia)`, que añade una nueva incidencia al `ArrayList` de incidencias.

Apartado 3 (0,8 puntos)

Por último, se pide programar la clase `ClientTV`, que modela los clientes que sí tienen contratada la televisión. Estos clientes, además de tener nombre, id y lista de incidencias, disponen de un array con capacidad para grabar hasta 100 programas (`Programme`), sean del tipo que sean. Las posiciones vacías de este array estarán marcadas con `null`. Para poder grabar los programas, y como se comentará posteriormente, se necesitará hacer uso de la guía de televisión (`TVGuide`), que es una clase que ha sido creada por otro equipo. Esta guía dispone de un método `static Programme searchProgramme(String name)`, que devuelve un programa de TV a partir de su nombre si lo encuentra en la guía o `null` si no lo encuentra.

Además, en tu clase `ClientTV` debes incluir:

- Un constructor que inicialice todos los atributos. Además, dentro de las grabaciones se debe incluir una grabación promocional. Ésta está catalogada como noticia (`News`) con nombre `"TV info"`, con fecha `"28/06/2018 10:30"` y está disponible en HD. Puedes suponer que la clase que implementa las noticia no necesita ningún parámetro adicional respecto a un programa genérico.
- El método `record(String programme)`, que se encargará de grabar un programa. Para ello, buscará si el programa existe en la guía. En caso de que no exista, se lanzará una `TVException` indicando `"Programme not found"`. En caso de que sí se encuentre el programa en la guía, se intentará insertar el programa en el primer hueco disponible del array de grabaciones (puedes asumir que el programa nunca ha sido ya programado con anterioridad y no va a estar repetido). Si el array estuviera lleno con las 100 grabaciones (ninguna posición con valor `null`), se lanzaría una `TVException` con el mensaje `"There is not space for more recordings"`. Finalmente, si la operación es exitosa, se mostrará un mensaje con el siguiente formato:

  `Well done, <name>. Your programme has been scheduled for recording successfully.`
- El método `generateIncidence(String incidencia)`, no cambia su comportamiento con respecto al cliente que no está suscrito a la televisión.

NOTA 1: No debes preocuparte por importar la clase `ArrayList<E>` en la primera línea de tu programa.
NOTA 2: La clase `ArrayList<E>` tiene los siguientes métodos, algunos de los cuales pueden ser de utilidad:

- `boolean add(E e)`
- `void add(int index, E element)`
- `E get(int index)`
- `boolean isEmpty()`

Problema 2 (1 / 7 puntos)

La clase `ExamUtil` expone el siguiente método;

```
public static String gradeAlpha(float grade) {
      if (grade < 0.0f || grade > 10.0f)
            throw new IllegalArgumentException("Illegal grade");
      if (grade < 5.0f)             return "Failed";
      else if (grade < 7.0f)        return "Passed";
      else if (grade < 9.0f)        return "Outstanding";
      else                          return "Remarkable";
}
```

NOTA: Se asume que las notas tienen **solamente un decimal válido.** Es decir, no hay que considerar notas con más de un decimal.

Apartado 1 (0,1 puntos)

Se pide programar el test que pruebe el comportamiento de este método con el mayor grade negativo inválido.

Apartado 2 (0,1 puntos)

Se pide programar el test que pruebe el comportamiento de este método con el menor grade positivo inválido.

Apartado 3 (0,8 puntos)

Se pide programar **un test para cada posible resultado** que compruebe tanto la nota mínima como la nota máxima que dan ese resultado. Es decir, que se pruebe el suspenso más bajo y el más alto, el aprobado más bajo y el más alto, el notable más bajo y el más alto y, finalmente el sobresaliente más bajo y el más alto.

Problema 3 (2 / 7 puntos)

Dada una clase `Node` con los siguientes métodos y una clase que implementa una clase lista (`LinkedList`) formada por un conjunto de nodos:

Apartado 1 (1,25 puntos)

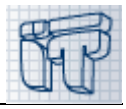
Programa el método `replaceZerosByPrevSum()` incluido en la clase `LinkedList`. Este método modifica la lista actual, obteniéndose una lista formada por las sumas de los elementos previos a cada "0". Observa los ejemplos que aparecen a continuación para comprender mejor el funcionamiento del método.

```
Ejemplo 1. Lista acabada en "0":
In: 5 -> 5 -> 5 -> 0 -> 17 -> 17 -> 0 -> 4 -> 4 -> 0 -> null
Out: 15 -> 34 -> 8 -> null

Ejemplo 2. Lista no acabada en "0":
In: 5 -> 5 -> 5 -> 0 -> 17 -> 17 -> 0 -> 4 -> 4 -> null
Out: 15 -> 34 -> 4 -> 4 -> null

Ejemplo 3. Lista que comienza por "0":
In: 0- > 5 -> 5 -> 5 -> 0 -> 17 -> 17 -> 0 -> 4 -> 4 -> null
Out: 15 -> 34 -> 4 -> 4 -> null
```

NOTA 1. En la lista de entrada se garantiza que no aparecerán dos elementos consecutivos con valor "0".
NOTA 2. Se sugiere reutilizar los nodos de la lista original para guardar las sumas, no hace falta crear nuevos nodos ni listas auxiliares.



```
public class Node {
    private Node next;
    private int num;

    public Node(int val, Node next){
        this.num = val;
        this.next = next;
    }
    public Node(int val) {
        this(val, null);
    }
    public int getInfo() {
        return this.num;
    }
    public Node getNext() {
        return this.next;
    }
    public void setInfo(int info) {
        this.num = info;
    }
    public void setNext(Node next) {
        this.next = next;
    }
 }
```

```
public class LinkedList {
    Node head;
    public LinkedList(int val) {
        this.head = new Node(val);
    }
    public LinkedList() { this.head = null;}
    public void insert(int val) {
       Node newNode = new Node(val);
       newNode.setNext(this.head);
       head = newNode;
    }
    public void print() {
        Node tmpNode = head;
        while (tmpNode != null) {
            System.out.print(tmpNode.getInfo()
                             + " -> ");
            tmpNode = tmpNode.getNext();
        }
        System.out.print("null");
    }
    public void replaceZerosByPrevSum() {
        // APARTADO 1
    }
}
```

Apartado 2 (0,75 puntos)

Escriba el código de la clase `Main` (con únicamente un método `main`) para ejecutar y mostrar por pantalla el siguiente ejemplo:

```
In: 5 -> 5 -> 5 -> 0 -> 17 -> 17 -> 0 -> 4 -> 4 -> 0 -> null
Out: 15 -> 34 -> 8 -> null
```

Problema 4 (2 / 7 puntos)

Para el siguiente problema, considere la interfaz `BTree<E>` y las clases `LBNode<E>` y `LBTree<E>` con los siguientes métodos ya implementados, con la misma semántica que la vista en las clases de laboratorio y sin preocuparse de las excepciones.

```
public interface BTree<E> {
    static final int LEFT = 0;
    static final int RIGHT = 1;

    boolean isEmpty();
    E getInfo();
    BTree<E> getLeft();
    BTree<E> getRight();
    void insert(BTree<E> tree,
                int side);
    BTree<E> extract(int side);
    int size();
    int height();
    boolean equals(BTree<E> tree);
    boolean find(BTree<E> tree);
}
```

```
public class LBNode<E> {
    private E info;
    private BTree<E> left;
    private BTree<E> right;
    LBNode(E info, BTree<E> left, BTree<E> right) { ... }
    E getInfo() { ... }
    void setInfo(E info) { ... }
    BTree<E> getLeft() { ... }
    void setLeft(BTree<E> left) { ... }
    BTree<E> getRight() { ... }
    void setRight(BTree<E> right) { ... }
}
```

```
public class LBTree<E> implements BTree<E> {
    private LBNode<E> root;
    public LBTree() { ... }
    public LBTree(E info) { ... }
    // ...
}
```

Se pide añadir un método `public boolean someSubtreeEqualsToSomeSubtreeOf(BTree<E> other)` a la clase `LBTree<E>`, que dado el propio árbol `this` y el otro árbol `other` que se reciba por parámetro, devuelva:

- `true` cuando se pueda encontrar algún subárbol no vacío de `other` que sea igual a algún subárbol no vacío de `this`
- `false` en caso contrario

La búsqueda debe hacerse empezando por los subárboles más grandes y continuando por los subárboles más pequeños. Para resolverlo, es suficiente con utilizar los métodos existentes (incluido este nuevo método si se resuelve de manera recursiva), aunque puedes añadir nuevos métodos a las clases y/o a la interfaz si lo consideras oportuno. El algoritmo deberá parar cuando encuentre la primera coincidencia.

SOLUCIONES DE REFERENCIA (Varias soluciones a cada uno de los problemas son posibles)

PROBLEMA 1

Apartado 1 (0,5 puntos)

```
public abstract class Programme {
	static final int SD = 1;
	static final int HD = 2;

	String name;
	String time;
	int quality;

	public Programme(String name, String time, int quality){
		this.name = name;
		this.time = time;
		if(quality != SD && quality != HD)
			quality = SD;
		this.quality = quality;
	}

	public abstract void description();
}
```

Apartado 2 (0,7 puntos)

```
public class Client implements Functions {
	private static int id_counter;
	private int id;
	private String name;
	private ArrayList<String> incidences;

	public Client(String name){
		this.name = name;
		id_counter++;
		this.id = id_counter;
		incidences = new ArrayList<String>();
	}

	public void record(String name) throws TVException{
		throw new TVException ("Recordings are not available in your
subscription");
	}

	public void generateIncidence(String incidencia){
		incidences.add(incidencia);
	}

	public String getName() {
		return name;
	}
}
```

Apartado 3 (0,8 puntos)

```
public class ClientTV extends Client{
	private Programme[] grabaciones;

	public ClientTV(String name){
		super(name);
		grabaciones = new Programme[100];
		grabaciones[0] = new News("TV Info", "28/06/2018 10:30", Programme.HD);
	}

	public void record(String name) throws TVException{
		Programme p = TVGuide.searchProgramme(name);
		if(p == null){
			throw new TVException("Programme not found");
		}
		boolean inserted = false;
		for(int i=0; !inserted && i<grabaciones.length;i++){
			if(grabaciones[i] == null){
				grabaciones[i] = p;
				inserted = true;
			}
		}
		if(!inserted){
			throw new TVException("There is not space for more recordings");
		}
		else{
			System.out.println("Well done, " + getName() + ". Your programme
has been scheduled for recording successfully");
		}
	}
}
```

PROBLEMA 1

Apartado 1 (0,5 puntos)

- 0,1: Declaración de la clase
- 0,1: Declaración de las constantes
- 0,1: Atributos. Si está mal la visibilidad, entonces 0
- 0,1: Constructor
- 0,1: Método abstracto
- Los errores significativos están sujetos a sanciones adicionales

Apartado 2 (0,7 puntos)

- 0,1: Declaración de la clase implementando la interfaz
- 0,1: Gestión del id con los dos atributos (uno de ellos estático) en la parte de atributos. Si el atributo estático se declara también como constante (final), entonces 0
- 0,1: Declaración del nombre y el ArrayList de incidencias
- 0,1: Constructor
- 0,1: Método record()
- 0,1: Método generateIncidence()
- 0,1: Método getName()
- Los errores significativos están sujetos a sanciones adicionales

Apartado 3 (0,8 puntos)

- 0,1: Declaración de la clase. Si no hereda de Client o implementa la interfaz, entonces 0.
- 0,1: Atributo grabaciones.
- 0,1: Constructor
- 0,5: Método record()
  - 0,1: Búsqueda del programa en la guía. Si hace la llamada a partir de un objeto TVGuide, en lugar de TVGuide.searchProgramme (al ser un método estático), entonces 0
  - 0,2: Inserción en el array en la primera posición disponible
  - 0,1: Lanzamiento de excepciones en los dos casos indicados. Si en vez de lanzar excepciones se imprimen mensajes de error, entonces 0
  - 0,1: Mostrar el mensaje si la operación es satisfactoria. Si accede directamente a name sin el getName(), entonces 0
- Respecto al método generateIncidence(), se dará por válido tanto si se implementa con super.generateIncidence(indicencia) o si no se sobreescribe. Cualquier otra solución será penalizada con 0,1.
- Los errores significativos están sujetos a sanciones adicionales

PROBLEMA 2

Apartado 1 (0,1 puntos)

```
@Test(expected = IllegalArgumentException.class)
public void testUnderGrade() {
        ExamUtil.gradeAlpha(-0.1f);
}
```

Apartado 2 (0,1 puntos)

```
@Test(expected = IllegalArgumentException.class)
public void testOverGrade() {
        ExamUtil.gradeAlpha(10.1f);
}
```

Apartado 3 (0,8 puntos)

```
@Test
public void testFailedLow() {
        assertEquals(ExamUtil.gradeAlpha(0.0f), "Failed");
}
@Test
public void testFailedHigh() {
        assertEquals(ExamUtil.gradeAlpha(4.9f), "Failed");
}
@Test
public void testPassedLow() {
        assertEquals(ExamUtil.gradeAlpha(5.0f), "Passed");
}
@Test
public void testPassedHigh() {
        assertEquals(ExamUtil.gradeAlpha(6.9f), "Passed");
}
@Test
public void testOutstandingLow() {
        assertEquals(ExamUtil.gradeAlpha(7.0f), "Outstanding");
}
@Test
public void testOutstandingHigh() {
```

```
            assertEquals(ExamUtil.gradeAlpha(8.9f), "Outstanding");
    }
    @Test
    public void testRemarkableLow() {
            assertEquals(ExamUtil.gradeAlpha(9.0f), "Remarkable");
    }
    @Test
    public void testRemarkableHigh() {
            assertEquals(ExamUtil.gradeAlpha(10.0f), "Remarkable");
    }
```

Apartado 1 (0,1 puntos)

- 0,05: Llamar a gradeAlpha adecuadamente con -0.1
- 0.05: Comprobar que devuelve la excepción esperada
- Los errores significativos están sujetos a sanciones adicionales

Apartado 2 (0,1 puntos)

- 0,05: Llamar a gradeAlpha adecuadamente con 10.1
- 0.05: Comprobar que devuelve la excepción esperada
- Los errores significativos están sujetos a sanciones adicionales

Apartado 3 (0,8 puntos)

- 0,30: Llamar a gradeAlpha adecuadamente con los valores mínimos de cada clase de equivalencia
- 0,30: Llamar a gradeAlpha adecuadamente con los valores máximos de cada clase de equivalencia
- 0.20: Comprobar que los resultados obtenidos son los correctos
- Los errores significativos están sujetos a sanciones adicionales

PROBLEMA 3

Apartado 1 (1,25 puntos)

```
public void replaceZerosByPrevSum() {
   if(head != null) {
      if(head.getInfo() == 0) {
           head = head.getNext();
      }
      Node res = head;
      Node temp = head;
      int sum = 0;
      while (temp != null) {
         if (temp.getInfo() != 0) {
             sum += temp.getInfo();
         }
         else {
             res.setInfo(sum);
             res.setNext(temp.getNext());
             res = res.getNext();
             sum = 0;
         }
         temp = temp.getNext();
      }
   }
}
```

Apartado 2 (0,75 puntos)

```
public class Main {
    public static void main(String[] args) {
        LinkedList myList = new LinkedList(0);
        myList.insert(4);
        myList.insert(4);
        myList.insert(0);
        myList.insert(17);
        myList.insert(17);
        myList.insert(0);
        myList.insert(5);
        myList.insert(5);
        myList.insert(5);
        myList.print();
        System.out.println("\n");
        myList.replaceZerosByPrevSum();
        myList.print();
    }
}
```

Apartado 1 (1,25 puntos)

- 0,25: Bucle para recorrer la lista
- 0,50: Suma de los nodos hasta que se llega a algún 0
- 0,50: Actualización de las variables auxiliares para el siguiente ciclo
- Los errores significativos están sujetos a sanciones adicionales

Apartado 2 (0,75 puntos)

- 0,50: Formar la lista
- 0,25: Llamada a la función
- Los errores significativos están sujetos a sanciones adicionales

PROBLEMA 4

```
public boolean someSubtreeEqualsToSomeSubtreeOf(BTree<E> other) {
       return !other.isEmpty() &&
              (find(other)
               || someSubtreeEqualsToSomeSubtreeOf(other.getLeft())
               || someSubtreeEqualsToSomeSubtreeOf(other.getRight())
              );
}
```

- 0,50: Devolver false cuando se llega a los subárboles vacíos
- 0,50: Comprobar primero si el árbol completo other es igual que el propio this o que alguno de los subárboles de this (llamando a this.find(other), o de alguna otra manera).
- 0,50: Aplicar de manera correcta el mismo algoritmo a los subárboles izquierdo y derecho de other.
- 0,50: Combinar los subresultados de forma apropiada de manera que se empiece buscando el mayor subárbol posible y continuar intentando con subárboles más pequeños.
- Se permiten soluciones no recursivas siempre que estén correctas.
- Penalización de -0,25 si, a pesar de haber ya encontrado una pareja de subárboles iguales, se siguen creando particiones de los árboles para encontrar más parejas
- Dar por válido tanto si se aplica la búsqueda a this y las particiones a other, o viceversa. Lo mismo con la verificación del árbol vacío, que puede ser sobre this o sobre other.
- Los errores significativos están sujetos a sanciones adicionales.