NOMBRE:
APELLIDOS:
NIA:
GRUPO:

Convocatoria Extraordinaria

2ª Parte: Problemas (7 puntos sobre 10)

Duración: 115 minutos
Puntuación máxima: 7 puntos
Fecha: 25 de junio de 2021

Instrucciones para el examen:

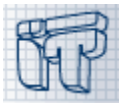
- No se permite el uso de libros o apuntes, ni tener teléfonos móviles u otros dispositivos electrónicos encendidos. Incumplir cualquiera de estas normas puede ser motivo de expulsión inmediata del examen.

Problema 1 OO + testing (3 puntos)

Una empresa líder en el sector de la logística le ha pedido que desarrolle un programa para la gestión de un almacén. El almacén distribuye productos (`Product`) que pueden ser de tres categorías diferentes representadas por tres constantes de tipo carácter: `f-FOOD` (alimentación), `M-MEDICAL` (productos médicos) y `m-MISCELLANY` (miscelánea). NOTA: Puedes asumir que todos los métodos get y set están implementados (salvo el del atributo category). Dada la clase `Product` se pide:

```
public class Product {
    private String name;
    private String brand;
    private char category;
    private int numUnits;
    private double costPerUnit;
    private double pricePerUnit;

    public Product(String name, String brand, char category, int numUnits,
            double costPerUnit, double pricePerUnit) { ... }

    //setters and getters
    //TODO: constants f-FOOD, M-MEDICAL, m-Miscellanea
    //TODO: setCategory
}
```

Apartado 1.1 Constantes y método setCategory de la clase Product (0,75 puntos)
Declarar tres constantes para representar los tres tipos de categorías (`category`) de producto (`Product`) permitidos en el almacén. Codificar el método `public void setCategory (char category) throws Exception` de la clase `Product` . El método debe comprobar que la categoría asignada es uno de los tres valores permitidos y, en caso contrario, debe lanzar una excepción de tipo `Exception`.



Apartado 1.2 Clase MedicalProduct (0,75 puntos)

Debido a la situación sanitaria actual la empresa necesita distinguir especialmente los productos médicos que necesitan ser suministrados con urgencia. Para ello se pide implementar una nueva clase (`MedicalProduct`) que sea una especialización de producto (`Product`), que además de contener toda la información de cualquier producto tenga un atributo llamado `lab` para guardar la información sobre el laboratorio proveedor del medicamento (`Provider`). Todos los productos médicos pertenecerán a la categoría `MEDICAL`.

Apartado 1.3 método getBenefit de la clase MedicalProduct (0,5 puntos)

Todos los productos médicos (`MedicalProduct`) ven mermado su beneficio por una comisión del 15% que se paga al laboratorio proveedor para que lo fabrique con urgencia. Se pide implementar el método `public double getBenefit()` en la clase `MedicalProduct` para que calcule el beneficio del producto teniendo en cuenta la comisión del 15% que debe pagar al laboratorio. (NOTA: Recuerda que el beneficio de un producto se puede calcular a partir del coste del producto para el almacén, su precio de venta al público y el número de unidades de dicho producto).

Apartado 1.4 método testGetBenefit (0,5 puntos)

Codifique un método de testing llamado `testGetBenefit()` para el método `getBenefit` de la clase `MedicalProduct`. El test debe verificar que un producto médico con 10 unidades, con 5 euros de coste y 10 euros de precio de venta al público proporciona un beneficio después de aplicar la comisión al laboratorio de 42,5 euros .

Apartado 1.5 método testSetCategory (0,5 puntos)

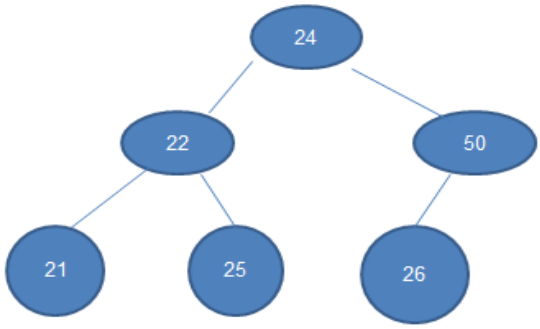
Codifique un método de testing llamado `testSetCategory()` para el método `setCategory` de la clase `Product`. Dicho método debe comprobar que se lanza la excepción en caso de asignar a un producto una categoría incorrecta.

Problema 2 (2 puntos)

Apartado 2. 1 método sumMultiplyOf (2 puntos)

Dadas la clase `TreeMain` y la interfaz BTree de la figura se pide implementar el método **recursivo** `public int sumMultipleOf (int number, BTree<Integer> tree)` de la clase `TreeTest` que dado un árbol de enteros (`tree`) y un número cualquiera (`number`) sume todos los enteros que sean múltiplos del número dado. Para hacerlo deberá recorrer el árbol sumando sólo aquellos nodos cuya contenido sea múltiplo del número dado. (Nota: Recuerda que debes manejar las excepciones que lancen los métodos de BTree). No se admitirán soluciones no recursivas.

Ejemplo. Dado el árbol que se muestra en la figura y el número `5` la llamada a `sumMultipleOf(5)` devolvería como resultado `75` puesto que hay dos nodos cuyo contenido es múltiplo de `5` (el nodo `50` y el nodo `25`).

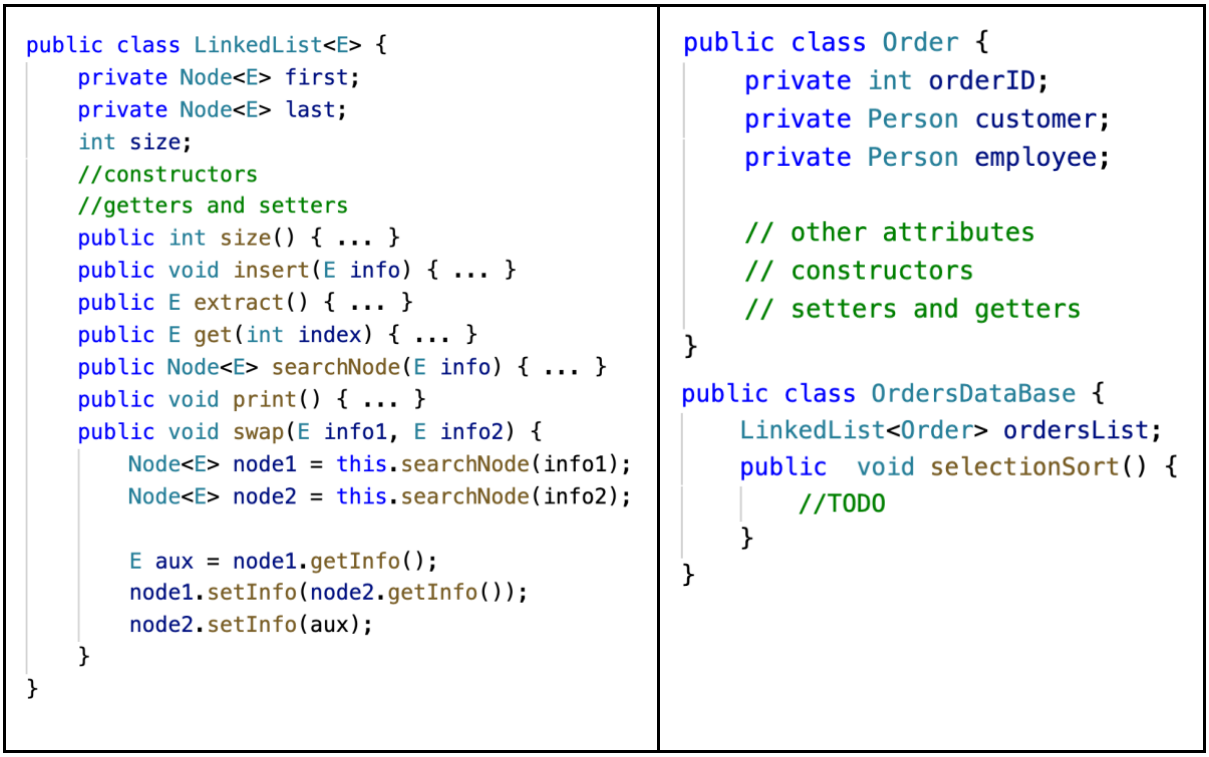



```java
public interface BTree<E> {

    static final int LEFT = 0;
    static final int RIGHT = 1;

    boolean isEmpty();

    E getInfo() throws BTreeException ;
    BTree<E> getLeft() throws BTreeException;
    BTree<E> getRight() throws BTreeException;

    //other methods
}
```

```java
public static void main(String[] args) {
    BTree e24 = new LBTree(24);
    BTree e22 = new LBTree(22);
    BTree e50 = new LBTree(50);
    BTree e21 = new LBTree(21);
    BTree e25 = new LBTree(25);
    BTree e26 = new LBTree(26);
    BTree<Integer> numbersTree = e24;
    try {
        e50.insert(e26, BTree.LEFT);
        e22.insert(e21, BTree.LEFT);
        e22.insert(e25, BTree.RIGHT);
        e24.insert(e22, BTree.LEFT);
        e24.insert(e50, BTree.RIGHT);
        numbersTree = e24;
        System.out.println(sumMultipleOf(5, numbersTree));
    } catch (BTreeException bte) {
        System.out.println(bte);
    }
}

public static int sumMultipleOf(int number, BTree<Integer> tree) {
    //TODO
}
```

Problema 3 (2 puntos)

Se quiere optimizar el proceso de consulta del histórico de pedidos de un almacén y para ello se ha creado una nueva clase llamada `OrdersDataBase` que contiene un listado de pedidos (`LinkedList<Order> ordersList`). Dadas las clases `LinkedList`, `Order` y `OrdersDataBase`, se pide:



```java
public class LinkedList<E> {
    private Node<E> first;
    private Node<E> last;
    int size;
    //constructors
    //getters and setters
    public int size() { ... }
    public void insert(E info) { ... }
    public E extract() { ... }
    public E get(int index) { ... }
    public Node<E> searchNode(E info) { ... }
    public void print() { ... }
    public void swap(E info1, E info2) {
        Node<E> node1 = this.searchNode(info1);
        Node<E> node2 = this.searchNode(info2);

        E aux = node1.getInfo();
        node1.setInfo(node2.getInfo());
        node2.setInfo(aux);
    }
}
```

```java
public class Order {
    private int orderID;
    private Person customer;
    private Person employee;

    // other attributes
    // constructors
    // setters and getters
}
public class OrdersDataBase {
    LinkedList<Order> ordersList;
    public  void selectionSort() {
        //TODO
    }
}
```

Apartado 3.1 Método compareTo de la clase Order (0,5 puntos)

Programar el método `compareTo` de la clase `Order` que permite comparar dos pedidos (`Order`) en función de su identificador (`orderID`), teniendo en cuenta el manejo de la excepción `ClassCastException`. NOTA: Recordar que para que un objeto de tipo `Object` pueda utilizar métodos de una clase que hereda de ella es necesario hacer un casting.

Apartado 3.2 Método selectionSort de la clase OrdersDataBase (1,5 puntos)

Implementar el método `selectionSort()` de la clase `OrdersDataBase` que permite ordenar los pedidos de la lista (`ordersList`) en orden ascendente (de menor a mayor) en función de su identificador de pedido (`orderID`). NOTA: puedes asumir que la clase `LinkedList` tiene los métodos que se indican en la figura como por ejemplo el método `swap` que permite intercambiar dos nodos de la lista a partir de la información que contienen.

SOLUCIÓN DE REFERENCIA (Varias soluciones son posibles)

Problema 1. OO + Testing (3 Puntos)

Apartado 1.1 Constantes y método setCategory (0,75 puntos)

Rúbrica:

- 0 si no tiene sentido.
- 0,25 por declarar las constantes.
- 0,5 por codificar el método set usando las constantes y lanzando correctamente la excepción (0,2 el lanzamiento de la excepción).
- Los errores significativos están sometidos a penalizaciones adicionales.

Solución:

```
public static final char FOOD = 'f';
public static final char MISCELLANY = 'm';
public static final char MEDICAL = 'M';
public void setCategory(char category) throws Exception {
	switch (category) {
		case FOOD:
		case MISCELLANY:
		case MEDICAL:
			this.category = category;
			break;
		default:
			throw new Exception(
				"The category must be: 'f' FOOD, 'm' Miscellanea, 'M' 
Medical");
		}
}
```

Apartado 1.2 clase MedicalProduct (0,75 puntos)

Rúbrica:

- 0 si no tiene sentido.
- 0,25 por declarar la clase correctamente heredando de Product y el atributo de la clase Provider.
- 0,2 por declarar correctamente el constructor y asignar el atributo lab.
- 0,3 por invocar correctamente a super() con los parámetros correctos, teniendo en cuenta que la categoría es una constante que no debe ser argumento.
- Los errores significativos están sometidos a penalizaciones adicionales.

Solución:

```
public class MedicalProduct extends Product{
```

```
	private Provider lab;

	public MedicalProduct(String name, String brand, int numUnits, double
	costPerUnit, double pricePerUnit, Provider lab) {

		super(name, brand, MEDICAL, numUnits, costPerUnit, pricePerUnit);
		this.lab = lab;

	}

}
```

Apartado 1.3 método getBenefit (0,5 puntos)

Rúbrica:

- 0 si no tiene sentido.
- 0,25 por aplicar correctamente el cálculo de la comisión.
- 0,25 por calcular correctamente el valor del beneficio y devolverlo.
- Los errores significativos están sometidos a penalizaciones adicionales.

Solución:

```
public double getBenefit(){
	return ((getPricePerUnit()-getCostPerUnit())*getNumUnits())*0.85;
}
```

Apartado 1.4 método testGetBenefit (0,5 puntos)

Rúbrica:

- 0 si no tiene sentido.
- 0,1 por usar la anotación @test.
- 0,15 por instanciar el objeto MedicalProduct con los valores correctos.
- 0,25 por usar correctamente assertEquals, incluyendo el último argumento de precisión.
- Los errores significativos están sometidos a penalizaciones adicionales.

Solución:

```
@Test
public void testGetBenefit() {
	MedicalProduct medicalProduct = new MedicalProduct("Aspirin", "Bayern", 10,
5, 10, new Provider());
	assertEquals(medicalProduct.getBenefit(), 42.5, 0.1);
}
```

Apartado 1.5 método testSetCategory (0,5 puntos)

Rúbrica:

- 0 si no tiene sentido.
- 0,1 por usar la anotación @test.
- 0,15 por instanciar el objeto Product con los valores correctos.
- 0,25 por usar correctamente assertThrows o el try/catch llamando a setCategory con un valor no permitido para que salte la excepción.
- Los errores significativos están sometidos a penalizaciones adicionales.

Solución:

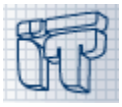
```
@Test
public void testSetCategory() {
	Product product = new Product("p1","b1",Product.FOOD,10,20,30);

	// Solution 1 JUNit5
	assertThrows(Exception.class, ()->{product.setCategory('X');});

	// Solution 2
	try {
		product.setCategory('X');
		fail("Failed test");
	} catch (Exception e) {
		e.printStackTrace();
	}
}
```

Problema 2 (2 puntos)

Rúbrica:

- 0 si no se hace recursivo
- 0,25 inicialización y devolución del resultado correcto
- 0,25 manejo de excepciones (try/catch). Considerar correcto también si dentro del catch ponen un System.out.println
- 1 rama del if
  - 0,5 condición
  - 0,5 asignación del resultado
- 0,5 rama del else (asignación del resultado)

Solución:



```java
public static int sumMultipleOf(int number, BTree<Integer> tree) {
    int result = 0;
    try {
        if (tree.isEmpty()) {
            result = 0;
        } else if (tree.getInfo() % number == 0) {
            result = tree.getInfo() + sumMultipleOf(number, tree.getLeft())
                        + sumMultipleOf(number, tree.getRight());
        } else {
            result = sumMultipleOf(number, tree.getLeft()) +
                    sumMultipleOf(number, tree.getRight());
        }
    } catch (BTreeException bte) {
        bte.printStackTrace();
    }
    return result;
}
```

Problema 3 (2 puntos)

Apartado 3. 1 Método compareTo de la clase Order (0,5 puntos)

Rúbrica:

- 0 si no tiene sentido. Cada criterio sólo puntúa si se cumple completamente
- 0,1 inicializar y devolver el tipo de retorno
- 0,1 Hacer el casting a Order y manejar la Excepción
- 0,1 Rama 0 correcta (condición y resultado)
- 0,1 Rama 1 correcta (condición y resultado)
- 0,1 Rama -1 correcta

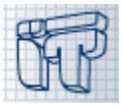
Solución:

```java
public int compareTo(Object other) {
    int result;
    // convert other to Order
    Order otherOrder = null;
    try {
      otherOrder = (Order) other;
    }catch(ClassCastException cce) {
        cce.printStackTrace();
    }
    if (this.getOrderID() == otherOrder.getOrderID()) {
        result = 0;
    } else if (this.getOrderID() < otherOrder.getOrderID()) {
        result = -1;
    } else {
        result = 1;
    }
    return result;
}
```

Apartado 3.2 Método selectionSort de la clase OrdersDataBase (1,5 puntos)

Rúbrica:

- 0 si no tiene sentido



    - 0,1 en total si han memorizado el método de las transparencias pero no lo adaptan al ejercicio.
- 0,25 Bucle externo (inicialización, condición y actualización)
- 0,2 inicialización de m
- 0,25 Bucle interno (inicialización, condición y actualización)
- 0,35 condición del if
    - 0,25 llamada correcta a compareTo con los índices y símbolo < en el orden adecuado
    - 0,1 llamada correcta a los elementos de la lista con get
- 0,2 actualización de m
- 0,25 llamada correcta a swap de ordersList.. 0 si hay algún fallo.

Solución:

```
public  void selectionSort() {
    for (int i = 0; i < ordersList.size(); i++) {
        int m = i;
        for (int j = i; j < ordersList.size(); j++) {
            if (ordersList.get(j).compareTo(ordersList.get(m)) < 0) {
                m = j;
            }
        }
        ordersList.swap(ordersList.get(i), ordersList.get(m));
    }
}
```