NOMBRE:
APELLIDOS:
NIA:
GRUPO:

Primer parcial

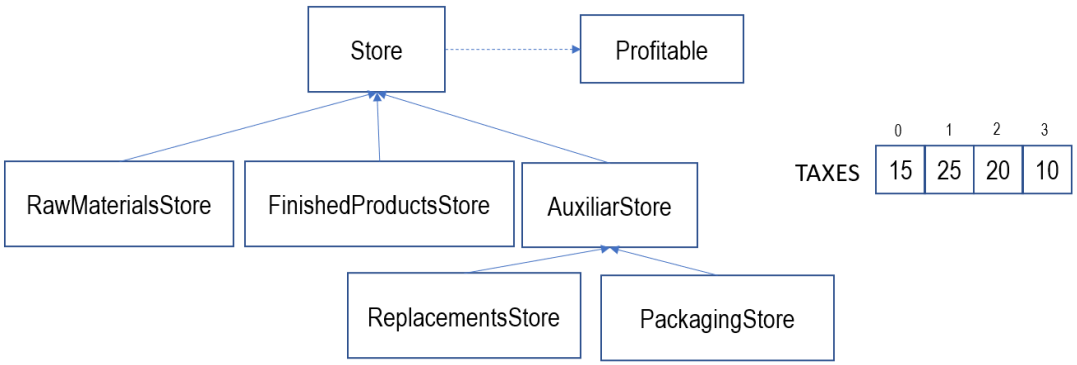
2ª Parte: Problemas (7 puntos sobre 10)

Duración: 70 minutos
Puntuación máxima: 7 puntos
Fecha: 26 de marzo 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.

Ejercicio 1 (5 / 7 puntos)



El almacén con el que has trabajado en el proyecto ha diversificado su actividad creando 4 tipos (type) distintos de almacén (Store): Uno para materias primas (`RawMaterialsStore`), otro para productos terminados (`FinishedProductsStore`) y otros dos auxiliares: uno para piezas de repuesto (`RepacementStore`) y otro para embalajes (`PackagingStore`). Se pide:

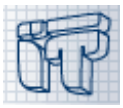
Interfaz Profitable

- Implementa la interfaz `Profitable` (rentable) que es aplicable a cualquier producto con el que se pueda obtener un beneficio gracias a su venta. Esta interfaz tiene tres métodos uno para calcular los ingresos (`calculateIncomes()`), otro para calcular los gastos (`calculateExpenses()`) y otro para calcular los beneficios (`calculateBenefit()`)

Clase Store. Declaración y atributos de clase

Declara la clase `Store` que implementa la interfaz `Profitable` e implementa sus atributos teniendo en cuenta las siguientes especificaciones:

- Cada almacén tiene un identificador numérico (`storeID`) que se asigna automáticamente en el momento de su creación. El identificador coincide con el número de almacenes existentes contando el recién creado (`numStores`).



- El número de almacenes existentes (`numStore`) es el mismo para todos los objetos de la clase y se encarga de llevar la cuenta de todos los almacenes creados hasta la fecha independientemente de su tipo.
- Cada almacén tiene un atributo que indica el tipo al que pertenece (`type`). El tipo sólo puede tomar uno de estos cuatro valores numéricos: `0-RAW_MATERIALS, 1-FINISHED_PRODUCTS, 2_REPLACEMENTS, 3_PACKAGING`.
- Crea un array de constantes (`TAXES`) como el que se indica en la figura que guarde los impuestos que tiene que pagar cada almacén en función del tipo al que pertenezca. Las materias primas pagan el 15%, los productos terminados el 25%, etc.

Clase Store. Constructores getters y setters

- Implementa los métodos `getType`, `setType` y `setTaxes` de la clase `Store` teniendo en cuenta que debes hacer comprobaciones para garantizar que el tipo y las tasas asignadas cumplen las restricciones indicadas. No es necesario que implementes ningún otro método `set/get`
- Implementa el constructor de la clase `Store public Store (int type)` que asigne valor a todos los atributos respetando las restricciones indicadas.

Clase Store. Otros métodos

- Implementa el método `calculateBenefit()` que permite calcular el beneficio como ingresos - gastos - ingresos *(%impuestos).
- Esta clase no tiene información suficiente para implementar los métodos que permiten calcular los ingresos y los gastos

Clase AuxiliarStore

Implementa la clase `AuxiliarStore` según la jerarquía de herencia indicada en la figura.

- Implementa el constructor de la clase `AuxiliarStore` que recibe un único parámetro.
- Esta clase no tiene información suficiente para calcular los ingresos y los gastos

Clase ReplacementStore

Implementa la clase `ReplacementsStore` según la jerarquía de herencia indicada en la figura

- Implementa un constructor sin parámetros que llame al constructor de la clase padre
- Implementa un método `toString` que devuelve el String `"Replacements Store ID: "` seguido del ID del almacén.
- Puedes asumir que los métodos `calculateIncomes`() y `calculateExpenses`() están implementados en esta clase (no es necesario que los implementes).

Ejercicio 2.Testing (0,5 / 7 Puntos)

Dado el siguiente Test para probar el método `toString` de la clase `PackagingStoreTest` indica el código que falta en los huecos indicados.

```
import static org.junit.jupiter.api.Assertions.*;

class PackagingStoreTest {
    PackagingStore p = new PackagingStore();
    @Test
    void testToString() {
        (1)("Packaging Store ID: 1", (2));
    }
}
```

Ejercicio 3. Recursión (1,5 / 7 Puntos)

Dado el siguiente código, programa el método recursivo `public void showReverse(int [] elements, int pos)` que muestra por pantalla los elementos del array de forma inversa a como están insertados.

En el ejemplo mostrará por pantalla: 3-2-1-

```
public class Recursion{
    public void showReverse(int [] elements, int pos){
    // Tú código aquí
    }

    public static void main(String [] args){
        int [] elements = {1,2,3};
        Recursion r = new Recursion();
        r.showReverse(elements, 0);
    }
}
```

SOLUCIÓN DE REFERENCIA (Varias soluciones son posibles)

Ejercicio 1. Solución (5 / 7 puntos)

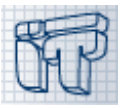
- **Interfaz Profitable (0,5 puntos)**
  - 0,25: Declaración interfaz
  - 0,25: Declaración de métodos

```
public interface Profitable {
    public double calculateIncomes();
    public double calculateExpenses();
    public double calculateBenefit();
}
```

- **Clase Store.Declaración y atributos de clase (1,25 puntos)**
  - 0,25 declaración de clase
  - 0,25 atributos numStores y storeID
  - 0,25 atributos type y taxes
  - 0,25 array de constantes
  - 0,25 resto de constantes

```
public abstract class Store implements Profitable {
    private static int numStores;
    private int storeID;
    public static final double[] TAXES = {15, 25, 20, 10};
    public static final int RAW_MATERIALS = 0;
    public static final int FINISHED_PRODUCTS = 1;
    public static final int REPLACEMENTS = 2;
    public static final int PACKAGING = 3;
    private int type;
    private double taxes;
    //...
```

- **Clase Store. Constructores getType, setType, setTaxes (1,5 puntos)**
  - 0,8 constructor
  - 0,2 getType
  - 0.25 setType
  - 0.25 setTaxes



```
public Store(int type) {
    setType(type);
    setTaxes();
    storeID = ++numStores;
}
public int getStoreID() {
    return storeID;
}

public int getType() {
    return type;
}

public void setType(int type) {
    if (type == RAW_MATERIALS || type == FINISHED_PRODUCTS ||
        type == REPLACEMENTS  || type == PACKAGING) {
        this.type = type;
    } else {
        System.err.println("Invalid type");
    }
}
private void setTaxes() {
    taxes = TAXES[type] ;
}
private double getTaxes() {
    return taxes;
}
```

- **Clase Store. CalculateBenefit (0,25 puntos)**

```
public double  calculateBenefit() {
    return calculateIncomes()-calculateIncomes()*taxes/100 - calculateExpenses();
}
```

- **Clase AuxiliarStore (0,75 puntos)**
  - 0,25 declaración de clase con extends
  - 0,25 métodos abstractos o abstract en declaración de la clase
  - 0,25 constructor

```
public abstract class AuxiliarStore extends Store {
    public AuxiliarStore(int type) {
        super(type);
    }
}
```

- **Clase ReplacementStore (0,75 puntos)**
  - 0,25 declaración de clase
  - 0,25 constructor
  - 0,25 método

```
public class ReplacementsStore extends AuxiliarStore {
    public ReplacementsStore() {
        super(REPLACEMENTS);
    }
    public String toString() {
        return "Replacements Store ID: " + getStoreID();
    }
}
```

Ejercicio 2. Solución (0.5 / 7 puntos)

- 0.25 assertEquals
- 0.25 p.toString()

Solución:

```
import static org.junit.jupiter.api.Assertions.*;

class PackagingStoreTest {
    PackagingStore p = new PackagingStore();
    @Test
    void testToString() {
        assertEquals("Packaging Store ID: 1", p.toString());
    }
}
```

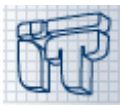
Ejercicio 3. Solución (1,5 / 7 Puntos)

- 0,5 puntos :: condición de salida correcta
- 0,5 puntos :: llamada recursiva correcta (valores de los parámetros)
- 0,5 puntos :: Impresión correcta de los valores.
- No puntúa nada si no es recursivo

Solución-A

```
public class Recursion {
    public void showReverse(int[] elements, int pos) {
        int value;
        if (pos < elements.length) {
            value = elements[pos];
            showReverse(elements, pos + 1);
            System.out.print(value + "-");
        }
    }

    public static void main(String[] args) {
        int[] elements = { 1, 2, 3 };
        Recursion r = new Recursion();
        r.showReverse(elements, 0);
    }
}
```

Solucion-B

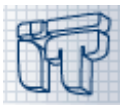


```java
public class Recursion4 {
    public void showReverse(int[] elements, int pos) {
        if(pos == elements.length) {
            return;
        }else{
            showReverse(elements, pos + 1);
            System.out.print(elements[pos] + "-");
        }
    }

    public static void main(String[] args) {
        int[] elements = { 1, 2, 3 };
        Recursion4 r = new Recursion4();
        r.showReverse(elements, 0);
    }
}
```

Solución-C: // En este caso la llamada es distinta y el método es static

```java
public class Recursion {
    public static void showReverse(int[] elements, int pos) {
        if(pos < 0) {
            return;
        }else{
            System.out.print(elements[pos] + "-");
            showReverse(elements, pos - 1);

        }
    }
    public static void main(String[] args) {
        int[] elements = { 1, 2, 3 };
        Recursion.showReverse(elements, elements.length-1);
    }
}
```

Solución-D: // En este caso la llamada es distinta y el método es static



```java
public class Recursion2 {
    public static void showReverse(int[] elements, int pos) {
        if(pos == elements.length) {
            return;
        }else{
            showReverse(elements, pos + 1);
            System.out.print(elements[pos] + "-");
        }
    }

    public static void main(String[] args) {
        int[] elements = { 1, 2, 3 };
        Recursion2.showReverse(elements, 0);
    }
}
```