Ejercicio 1. Unidades de Almacenaje (3 puntos)

En un Sistema Operativo hay *Unidades de Almacenaje*. Las *Unidades de Almacenaje* se caracterizan por su *nombre* y *tamaño*. Las *Unidades de Almacenaje* pueden ser *Ficheros* o *Directorios*. Los *Ficheros* **son** *Unidades de Almacenaje* simple. Los *Directorios* son *Unidades de Almacenaje* que **se componen** a su vez de un conjunto de *Unidades de Almacenaje* (*Ficheros* o *Directorios*).

SE PIDE:

Codificar las clases `UnidadAlmacenaje`, `Fichero`, `Directorio` y los métodos necesarios para que, dado un stock de *Unidades de Almacenaje*, podamos generar una lista (`ArrayList`) que contenga la descripción (`nombre`, `tamaño`) de aquellos ficheros que superen un tamaño determinado independientemente de la profundidad a la que se encuentren como se indica en la clase `ListadoUnidadesPorTamaño`.

```
import java.util.ArrayList;

public class ListadoUnidadesPorTamaño {
    private ArrayList<UnidadAlmacenaje> stock;


    public ListadoUnidadesPorTamaño(ArrayList<UnidadAlmacenaje> stock) {
        this.stock = stock;
    }

    public  void listar(double tamaño) {
        ArrayList<String> resultado = new ArrayList<>();
        for (UnidadAlmacenaje item: stock) {
            resultado.addAll(item.mayorQue(tamaño));
        }
        for (String elemento:resultado) {
            System.out.println(elemento);
        }
    }
}
```

El método `mayorQue` con la siguiente definición:

```
public ArrayList<String> mayorQue (double tamaño);
```

Es el encargado de obtener los ficheros que superen un `tamaño` determinado independientemente de la profundidad a la que se encuentren.

Examen de Programación Orientada a Objetos (Plan 2014) — **17 de diciembre de 2021**

Nº matrícula:_____ **Grupo:**_____ **Nombre:** _____

Apellidos: _____

Ejercicio 1. Unidades de Almacenaje (3 puntos)

(1 punto) Clase `UnidadAlmacenaje`

```
public abstract class UnidadAlmacenaje {
    private String nombre;
    private double tamaño;


    public UnidadAlmacenaje(String nombre, double tamaño) {
       this.nombre = nombre;
       this.tamaño = tamaño;
    }

    public double getTamaño() {
        return tamaño;
    }

    @Override
    public String toString() {
        return "UnidadAlmacenaje{" + "nombre='" + nombre + '\'' +", tamaño=" +
tamaño +'}';

    public abstract ArrayList<String> mayorQue(double tamaño);
}
```

(1 punto) Clase `File`

```
import java.util.ArrayList;
public class File extends UnidadAlmacenaje{

    public File(String nombre, double tamaño) {
        super(nombre, tamaño);
    }

    @Override
    public ArrayList<String> mayorQue(double tamaño) {
        ArrayList<String> resultado = new ArrayList<>();
        if(this.getTamaño()>tamaño)
            resultado.add(this.toString());
        return resultado;
    }

    @Override
    public String toString() {
        return "File  " + super.toString();
    }
}
```

(1 punto) Clase `Folder`

```
import java.util.ArrayList;

public class Folder extends UnidadAlmacenaje{
    private ArrayList<UnidadAlmacenaje> items;

    public Folder(ArrayList<UnidadAlmacenaje> items) {
        this.items = items;
    }

    @Override
    public ArrayList<String> mayorQue(double tamaño) {
        ArrayList<String> resultado = new ArrayList<>();
        for (UnidadAlmacenaje item:items) {
            resultado.addAll(item.mayorQue(tamaño));
        }
        return resultado;
    }

    @Override
    public String toString() {
        return "Folder{" +
                "items=" + items +
                '}'+ super.toString();
    }
}
```

Ejercicio 2. Expresiones aritméticas (3 puntos)

Una aplicación implementa un sencillo sistema de evaluación de *expresiones aritméticas de números enteros positivos*. Las operaciones que tiene implementadas son *suma*, *resta*, *producto* y *cociente*. El sistema lee por teclado la expresión que introduce el usuario y devuelve el resultado de evaluar dicha expresión siempre y cuando sea correcta; en caso contrario, lanzará una excepción. Por simplificar los cálculos, todas las operaciones aritméticas tienen la misma prioridad, por lo que se evalúa de izquierda a derecha sin importar cuál es el operador.

Ejemplo	*Resultado*
12 – 4 + 7 * 3 / 5	9 equivale a (((12 – 4) + 7) * 3) / 5
12 – 4 +	excepción por expresión incorrecta
* 12 – 4	excepción por expresión incorrecta
12 % 4	excepción por operador incorrecto

El sistema utiliza el método `evaluar` que toma como argumento un `string` con la expresión que ha introducido el usuario y devuelve el resultado como un `int`.

```
public int evaluar(String expresion)
```

El método `evaluar` utiliza el método privado `trocear` el cual toma como argumento un `string` con la expresión y devuelve una cola (`LinkedList`) cuyos elementos son enteros que representan los operandos y también los operadores codificados como enteros de la siguiente manera (`0:suma`, `1:resta`, `2:producto` y `3:cociente`).

```
private LinkedList<Integer> trocear(String expresion)
```

El método `trocear` analiza el `string`, pasado por parámetro, devolviendo en trozos o *tokens* cada uno de los elementos asegurando que el primer elemento es un operando y que no hay operadores incorrectos, de lo contrario, lanza una excepción.

Ejemplo	*Resultado*
12 – 4 + 7 * 3 / 5	12, 1, 4, 0, 7, 2, 3, 3, 5
12 – 4 +	12, 1, 4, 0
* 12 – 4	excepción por expresión incorrecta
12 % 4	excepción por operador incorrecto

El método `evaluar` también utiliza el método `operar` que recibe *dos operandos* y *un operador* y devuelve el resultado de aplicar el operador a los operandos.

```
private int operar(int operando1, int operando2, int operador)
```

Por ejemplo, `operar(12, 4, 1)` equivale a `12 – 4` y devolvería `8`.

Examen de Programación Orientada a Objetos (Plan 2014) — **17 de diciembre de 2021**

Nº matrícula:_____ **Grupo:**_____ **Nombre:** _____

Apellidos: _____

Ejercicio 2. Expresiones aritméticas (3 puntos)

SE PIDE

A) **(0.5 puntos)** Codificar las clases `ExpresionIncorrectaException` y `OperadorIncorrectoException`.

```
public class ExpresionIncorrectaException extends Exception {
     public ExpresionIncorrectaException() {
             super("ExpresionIncorrectaException");
     }
}

public class OperadorIncorrectoException extends Exception {
     public OperadorIncorrectoException() {
             super("OperadorIncorrectoException");
     }
}
```

B) **(0.5 puntos)** Completar la cabecera de `trocear`.

```
private LinkedList<Integer> trocear(String expresion)
             throws ExpresionIncorrectaException, OperadorIncorrectoException
```

C) **(2 puntos)** Suponiendo que ya están implementados los métodos `trocear` y `operar`, codificar el método `evaluar`.

NOTA: Tener en cuenta los siguientes métodos disponibles en la clase `LinkedList`.

```
public E poll() - Recupera y elimina el primer elemento de la lista.
public int size() - Devuelve el número de elementos de la lista.
public boolean isEmpty() - Devuelve true si la colección está vacía.
```

```
public int evaluar(String expresion)
     throws ExpresionIncorrectaException, OperadorIncorrectoException {

     int res, operando, operador;
     LinkedList<Integer> trozos = null;
     try {
             trozos = trocear(expresion);
             if (trozos.size() % 2 == 0)
                     throw new ExpresionIncorrectaException();
             res = trozos.poll();
             while (!trozos.isEmpty()) {
                     operador = trozos.poll();
                     operando = trozos.poll();
                     res = operar(res, operador, operando);
             }
             return res;
     }
     catch (ExpresionIncorrectaException e1) {
             System.out.println("La expresión es incorrecta.");
             throw new ExpresionIncorrectaException();
     }
     catch (OperadorIncorrectoException e2) {
             System.out.println("La expresión tiene un operador incorrecto");
             throw new OperadorIncorrectoException();
     }
}
```

Ejercicio 3. Calificación de exámenes (4 puntos)

Los profesores de la asignatura de POO deciden automatizar el proceso de elaboración y calificación de los exámenes de tipo test de la asignatura. Para ello se definen las siguientes clases:

Clase `Alumno`:

```
public class Alumno {
    private String nombre;
    private float nota;

    public Alumno(String nombre) {
      this.nombre = nombre;
      nota = 0.0f;
    }

    public void setNota(float nota) { this.nota = nota; }
    public float getNota() { return nota;    }
    public String getNombre() { return nombre; }
    public void mostrar() { System.out.println(nombre + ": " + nota); }
}
```

Clase `Pregunta`:

```
public class Pregunta {
    private String enunciado;
    private String[] opciones;
    private int correcta;

    public Pregunta () {
      enunciado = generaEnunciado();
      opciones = new String[4];
      for (int i = 0; i < 4; i++) opciones[i] = generaOpcion();
      correcta = asignaCorrecta();
    }

    /* Resto de declaraciones */
}
```

Donde, los métodos `generaEnunciado`, `generaOpcion` y `asignaCorrecta` están implementados y devuelven los datos de cada pregunta solicitándolos al profesor que prepara el examen.

Clase `Examen`:

```
import java.util.Collections;
import java.util.ArrayList;

public class Examen {
    private ArrayList<Pregunta> preguntas;

    public Examen () {
      preguntas = new ArrayList<Pregunta>();
      for (int i = 0; i < 10; i++)
            preguntas.add(new Pregunta());
    }

    /* Resto de declaraciones */
}
```

Clase `GestorExamen`:

```
import java.util.ArrayList;
import java.util.TreeSet;

public class GestorExamen {
    private String identificador;
    private Examen examen;
    private ArrayList<Alumno> alumnos;

    public GestorExamen (String identificador) {
      this.identificador = identificador;
      alumnos = leerAlumnosMatriculados();
      examen = new Examen();
    }

    public Alumno getAlumno(String nombre) {
      for (Alumno a : alumnos)
          if (a.getNombre().equals(nombre))
             return a;
      return null;
    }

    /* Resto de declaraciones */
}
```

Nº matrícula:_____ **Grupo:**_____ **Nombre:** _____

Apellidos: _____

Ejercicio 3. Calificación de exámenes (4 puntos)

SE PIDE

A) **(0.5 puntos)** En la clase `Pregunta`, suponiendo que tenemos implementado un método `mostrarPregunta` que muestra la pregunta con las distintas opciones al alumno y devuelve un `int` con la opción elegida (entre `0` y `3`), se pide: implementar el método `evaluar` que devuelva un `float` con valor `1.0f` si la opción elegida por el alumno es la correcta y `0.0f` en caso contrario.

```
public float evaluar() {
   int opcion = mostrarPregunta();
   if (opcion == correcta)
      return 1.0f;
   return 0.0f;
}
```

B) **(1 punto)** En la clase `Examen`, implementar el método `evaluar` que presente una a una todas las preguntas al alumno de forma aleatoria y devuelva un `float` con la nota final del alumno (con la suma de la nota obtenida en cada pregunta). Tener en cuenta la existencia del siguiente método de la clase `Collections`:

```
public static void shuffle(List<?> list) - Permuta aleatoriamente la lista
especificada.
```

```
public float evaluar() {
   float nota = 0.0f;
   Collections.shuffle(preguntas);
   for (Pregunta p : preguntas)
      nota += p.evaluar();
   return nota;
}
```

C) **(0.5 puntos)** En la clase `GestorExamen`, implementar el método `evaluar` que evalúe uno a uno a todos los alumnos matriculados en la asignatura y les asigne la nota correspondiente llamando al método `setNota` de la clase Alumno.

```
public void evaluar() {
    for (Alumno a : alumnos)
       a.setNota(examen.evaluar());
}
```

D) **(1 punto)** En la clase `GestorExamen`, empleando un `TreeSet` como estructura auxiliar, implementar el método `mostrarNotasPorNombre` que muestre el resultado de la evaluación de todos los alumnos ordenados por nombre.

```
public void mostrarNotasPorNombre() {
   TreeSet<String> porNombre = new TreeSet<String>();
   for (Alumno a : alumnos)
      porNombre.add(a.getNombre());
   for (String s : porNombre)
      getAlumno(s).mostrar();
}
```

E) **(0.5 puntos)** ¿Qué habría que cambiar para contemplar otro tipo de preguntas? Por ejemplo, preguntas de test con opción múltiple.

En lugar de tener una única clase Pregunta, podríamos tener una jerarquía de clases que derivaran de la clase Pregunta (PreguntaOpcionUnica, PreguntaOpcionMultiple…) sobreescribiendo el método evaluar en cada una de ellas.

F) **(0.5 puntos)** Qué habría que cambiar y/o añadir para incluir una clase que gestionara todos los exámenes de la asignatura en un curso académico?

Bastaría con añadir una clase más "GestorExamenesAsignatura" que contuviera una instancia de la clase GestorExamen para cada uno de los exámenes realizados a lo largo del curso