# Ejercicios TADs

## Pregunta 1 [Lista+Cola]

Se ha decidido diseñar el TAD `Oficina`. Una `Oficina` se define como una colección de empleados. Cada empleado de la oficina está identificado por un código y tiene que trabajar sobre una cola de expedientes que tiene asignados. Para ello se ha decidido diseñar el TAD `Oficina`, con las operaciones que se indican a continuación:

```
public interface Oficina extends Iterable<Empleado> {
    public int numEmpleados();
    // Produce: devuelve el número de empleados de la oficina.

    public Empleado get(int codigo) throws IllegalArgumentException;
    // Produce: devuelve el empleado con el código que se pasa como parámetro. Si
no existe, lanza la excepción IllegalArgumentException.

    public void insertar(int codigo);
    // Modifica: this
    // Produce: añade un empleado a la oficina con el código que se pasa como
parámetro y una cola de expedientes vacía.

    public boolean baja(int codigo);
    // Modifica: this
    // Produce: si el empleado tiene expedientes en su cola, retorna false; en
otro caso, elimina un empleado de la oficina y retorna true.

    public boolean asignarExpediente(Expediente x);
    // Modifica: this
    // Produce: si la oficina está vacía, devuelve falso; en otro caso, asigna el
expediente x al empleado con la cola de expedientes de menor tamaño y retorna
true. Si varios empleados tienen el mismo número de expedientes en su cola y son
los de menor tamaño se le asigna a cualquiera de ellos.

    public boolean eliminarExpediente(Empleado e);
    // Modifica: this
    // Produce: si la cola de expedientes del empleado e está vacía retorna
false; en otro caso, elimina un expediente de la cola de expedientes del empleado
e y retorna true.
}
```

Para implementar la interfaz `Oficina` se ha decidido utilizar la siguiente clase cuya estructura de datos es una `Lista`:

```
public class OficinaImp implements Oficina {
    private Lista<Empleado> listEmpleados;
}
```

Así como las clases `Empleado` y `Expediente`:

```
 public class Empleado {
    private int codigo;
```

```
    private Cola<Expediente> colaExp;

    public Empleado(int c) {
        codigo = c;
        colaExp = new EnlazadaCola<>();
    }
    public int getCodigo() {
        return codigo;
    }
    public Cola<Expediente> getExp() {
        return colaExp;
    }

    // Otros métodos
}

public class Expediente {
    private int numero;
    private String texto;

    public Expediente(int n, String t) {
        numero = n;
        texto = t;
    }

    public int gerNumero() {
        return numero;
    }

    public String getTexto() {
        return texto;
    }
}
```

Se pide:

1. Continuar con la implementación de la **clase** `OficinaImp` escribiendo el código de la operación `asignarExpediente`:

```
public boolean asignarExpediente(Expediente x)
// Modifica: this
// Produce: si la oficina está vacía, devuelve falso; en otro caso asigna el
expediente x al empleado con la cola de expedientes de menor tamaño y retorna
true. Si varios empleados tienen el mismo número de expedientes en su cola y
son los de menor tamaño se le asigna a cualquiera de ellos.
```

2. Haciendo uso de la **interfaz** `Oficina`, resuelve el siguiente método:

```
public static Lista<Integer> listarExpedientes(int código, Oficina of)
// Produce: devuelve una lista con los números de expediente que están en la
cola del empleado con el código que se pasa como parámetro.
```

# Pregunta 2 [Lista+Pila]

Los compiladores de lenguajes de bloques utilizan lo que se denomina una **tabla de símbolos**. Estas tablas se caracterizan porque, cuando el compilador entra en un nuevo bloque B, todos los objetos que se declaran en él esconden las apariciones de objetos con el mismo nombre que aparecen en los bloques que envuelvan a B, y cuando el compilador sale del bloque, todos los objetos que estaban escondidos vuelven a ser visibles.

Una posible implementación del **TDA Tabla de Símbolos** es considerar que es una pila de listas. Cada lista guarda elementos que son pares `<identificador, característica>`, donde identificador es una cadena de caracteres que representa el nombre de la variable y característica es una cadena de caracteres que representa el tipo de la variable en cuestión. En esta lista no puede haber dos variables con el mismo identificador.

Ejemplo de tabla de símbolos:

La interfaz del TAD Tabla de símbolos es la siguiente:

```
public interface TSimbolos {
    public void entrar();
        // Modifica:    this
        // Produce:     añade una lista vacía en la cima de la Tabla de Símbolos

    public void salir();
        // Modifica:    this
        // Produce:     elimina la lista que está en la cima de la Tabla de
Símbolos.

    public boolean declarar(String identificador, String tipo);
        // Modifica:    this
        // Produce:     si el par (identificador, tipo) ya existe en la cima de
la Tabla de símbolos devuelve falso; en otro caso lo añade a la lista de la cima
y devuelve verdadero.

    public String consultar(String identificador);
        // Produce:     devuelve el tipo de la variable cuyo nombre es
‘identificador’ o “NO_DECLARADO" si no se encuentra. Empieza a buscarlo a partir
de la cima de la tabla de símbolos.
}
```

Para implementar la interfaz `TSimbolos` se ha decidido utilizar la siguiente clase cuya estructura de datos es una **pila de listas de pares**:

```
public class TSimboloImp implements TSimbolo {
    private Pila<Lista<Par>> tabla;

}

La clase Par se muestra a continuación:
public class Par {
    private String identificador, tipo;

    public Par(String identificador, String tipo) {
```

```
        this.identificador = identificador;
        this.tipo = tipo;
    }

    public String getIdentificador() {
        return identificador;
    }

    public String getTipo() {
        return tipo;
    }
}
```

Se pide continuar con la implementación de la **clase** `TSimboloImp` escribiendo el código de las operaciones:

```
public boolean declarar(String identificador, String tipo);
// Modifica:    this
// Produce:     si el par (identificador, tipo) ya existe en la lista de la cima de la Tabla de símbolos devuelve falso; en otro caso, lo añade a la lista de la cima y devuelve verdadero.

public String consultar(String identificador);
// Produce:     devuelve el tipo de la variable cuyo nombre es 'identificador' o "NO_DECLARADO" si no se encuentra. Empieza a buscarlo a partir de la cima de la tabla de símbolos.
```

# Pregunta 3 [Pila+Conjunto]

Dada la interfaz `Conjunto<E>`:

```
public interface Conjunto<E> extends Iterable<E> {
    public int size();
    public boolean pertenece(E e);
    public void insertar(E e);
    public void borrar(E e);
}
```

Implementada mediante la clase `ImpConjunto<E>`:

```
public class ImpConjunto<E> implements Conjunto<E> {

    public ImpConjunto() // crea un conjunto vacío

}
```

Y dada la interfaz `MultiConjunto<E>`:

```
public interface MultiConjunto<E> {
    public boolean estaTodosConjuntos(E e);
    //Produce: devuelve cierto si el elemento e está en todos los conjuntos.

    public void quitarTodosConjuntos(E e);
```

```
    //Modifica: this
    //Produce: quita un elemento de todos los conjuntos de la pila.

    public Conjunto<E> moverTodos();
    //Modifica: this
    //Produce: Crea un único conjunto con todos los elementos de los conjuntos de
la pila.

    public void quitarVacios();
    //Modifica: this
    //Produce: quita todos los conjuntos vacíos de la pila.

    public void añadirConjunto();
    //Modifica: this
    //Produce: añade un conjunto vacío a la pila.

    public void añadirElemento(E e);
    //Modifica: this
    //Produce: añade un elemento a la cima de la pila.
}
```

Esta última implementada mediante una `Pila<Conjunto<E>>`, es decir:

```
public class ImpMultiConjunto<E> implements MultiConjunto<E> {
    private Pila<Conjunto<E>> conjuntos;
}
```

Se pide:

1. Implementar, haciendo uso de la estructura de datos indicada, la operación del TAD `MultiConjunto`:

   ```
   public Conjunto<E> moverTodos()
   ```

2. Implementar, haciendo uso de la estructura de datos indicada, la operación del TAD `MultiConjunto`:

   ```
   public void quitarVacios()
   ```

# Pregunta 4 [Incompleta]

Un restaurante dispone de una carta, la cual se presenta como **cola de platos**. De cada plato se indica el nombre, precio y la **pila de ingredientes** que se emplean en su elaboración. De cada ingrediente se muestra el nombre, la cantidad que se usa en el plato y el coste.

Debido a la subida de precios, el restaurante ha decidido modificar la carta de manera que va a quitar de aquellos platos cuyo precio supere los 50€ el ingrediente utilizado en mayor cantidad (solo hay uno). El nuevo precio del plato debe quedar reflejado en la carta.

Las clase `Plato` e `Ingrediente` se muestran a continuación:

```
```

```java
public class Plato {
    private String nombre;
    private double precio;
    private Pila<Ingrediente> ingredientes;

    public Plato (String nombre, double precio, Pila<Ingrediente> pila) {
        this.nombre = nombre;
        this.precio = precio;
        this.ingredientes = pila;
    }

    public String getNombre() {
        return this.nombre;
    }

    public double getPrecio() {
        return this.precio;
    }

    public Pila<Ingrediente> getIngredientes() {
        return this.ingredientes;
    }

    public void setPrecio(double precio) {
        this.precio = precio;
    }

    public void setIngredientes(Pila<Ingrediente> pila) {
        this.ingredientes = pila;
    }
}
```

```java
public class Ingrediente {
    private String nombre;
    private int cantidad;
    private double precio;

    public Ingrediente(String nombre, int cantidad, double precio) {
        this.nombre = nombre;
        this.cantidad = cantidad;
        this.precio = precio;
    }

    public String getNombre() {
        return this.nombre;
    }

    public int getCantidad() {
        return this.cantidad;
    }

    public double getPrecio() {
        return this.precio;
```

```
    }
}
```

## Pregunta 5 [Listas]

Una matriz dispersa es una matriz de gran tamaño en la que la mayor parte de sus elementos son cero. Para optimizar el almacenamiento de estas matrices en memoria se ha diseñado un TAD que solo almacena los valores distintos de cero. La interfaz de dicho TAD se muestra a continuación:

```
public interface MatrizEnteros {
    public int numFilas();
     // Produce: devuelve el número de filas de la matriz.

    public int numCols();
     // Produce: devuelve el número de columnas de la matriz.

    public int recupera(int i, int j) throws IndexOutOfBoundsException;
     // Produce: si i<=0 o i>n o j<=0 o j>m lanza la excepcion
IndexOutOfBoundException
     //          en caso contrario devuelve el valor almacenado en la fila i,
columna j.

    public void asigna(int i, int j, int nuevoValor) throws
IndexOutOfBoundsException;
     // Modifica: this
     // Produce: si i<=0 o i>n o j<=0 o j>m lanza la excepcion
IndexOutOfBoundException
     //          en caso contrario asigna nuevoValor en la posicion dada por la
fila i, columna j.
}
```

Para su implementación se va a utilizar una **lista** que contiene los números de fila con datos distintos de cero en la matriz y para cada fila se guarda una **lista** que contiene los pares número de columna y el valor distinto de cero almacenado en la matriz.

Por ejemplo, dada la matriz:

Se guardaría como:

```
1 → (1,1), (2,2), (5,3), (6,4)
2 → (2,5), (4,6)
3 → (2,7), (3,8), (4,9)
4 → (1,1), (4,2)
5 → (2,3), (5,4), (6,5)
6 → (3,6), (5,7), (6,8)
7 → (7,9)
```

La clase que implementa dicha interfaz se llama `ListaMatrizEnteros`, y se muestra a continuación:

```
public class ListaMatrizEnteros implements MatrizEnteros {
    private int numFilas;
    private int numCol;
    private Lista<ElementoFila> filas;
}
```

La clase `ElementoFila` se define como:

```
public class ElementoFila {
    private int fila;
    private Lista<ElementoColumna> columnas;

    public ElementoFila(int fila) {
        this.fila = fila;
        this.columnas = new ListaEnlazada<>();
    }

    public int getFila() {
        return this.fila;
    }

    public Lista<ElementoColumna> getColumnas() {
        return this.columnas;
    }
}
```

La clase `ElementoColumna` se define como:

```
class ElementoColumna {
    private int columna;
    private int valor;

    public ElementoColumna(int columna, int  valor) {
        this.columna = columna;
        this.valor = valor;
    }

    public int getColumna() {
        return this.columna;
    }

    public int getValor() {
        return this.valor;
    }
}
```

Se pide que, haciendo uso de la estructura de datos de la clase `ListaMatrizEnteros`, se implemente el método *recupera* de la interfaz `MatrizEnteros`.

```
public int recupera(int i, int j) throws IndexOutOfBoundsException
// Produce: si i<=0 o i>n o j<=0 o j>m lanza la excepcion
IndexOutOfBoundException
//          en caso contrario devuelve el valor almacenado en la fila i, columna
j.
```

## Pregunta 6 [Lista]

Un estudiante de informática pertenece a un club de baloncesto de su localidad y el presidente le ha pedido que diseñe una aplicación para el mantenimiento de los/as socios/as del club. El estudiante ha decidido definir una clase llamada `Abonado` para almacenar la información de cada uno de los/as abonados/as al club, y un TAD llamado `Club`, que almacena un conjunto ordenado de abonados/as sin repetición. La clase `Abonado` está disponible en el anexo.

```
public interface Club extends Iterable<Abonado> {
        public boolean esVacio();
        // Produce: cierto si el club no tiene abonados; falso, en caso
contrario.

        public int cardinal();
        // Produce: el número de abonados del club.

        public boolean esSocio(Abonado abonado) throws NullPointerException;
        // Produce: si abonado es null lanza una excepcion; en caso contrario,
devuelve cierto si abonado pertenece al club; falso, en caso contrario.

        public Abonado recuperar() throws ClubVacioException;
        // Produce: lanza la excepción si el club está vacío; en otro caso,
recupera un abonado cualquiera del club.

        public boolean alta(Abonado abonado) throws NullPointerException;
        // Modifica: this
        // Produce: si abonado es null lanza una excepcion; en otro caso, si
abonado no está en el club lo añade y devuelve true; si ya es socio del club no
hace nada y devuelve falso.

        public boolean baja(Abonado abonado) throws NullPointerException;
        // Modifica: this
        // Produce: si abonado es null lanza una excepción; en otro caso,
devuelve cierto si elimina el abonado abonado del club, falso en caso contrario.
}
```

Se pide:

1. (2 puntos) Haciendo uso de la interfaz anterior, la clase `Abonado`, la interfaz `Lista<E>` y la clase `ListaEnlazada<E>` disponibles en el anexo, escribe el siguiente método:

   ```
   public static Lista<Abonado> getMorosos(Club miClub)
   // Produce: devuelve una lista con los abonados morosos del club.
   ```

2. (4 puntos) Para implementar la interfaz `Club` se va a utilizar una estructura lineal enlazada simple con referencia al primer y último nodo de la estructura y sin centinela. Los abonados se guardarán en dicha estructura ordenados ascendentemente por número de abonado.

De esta clase se pide escribir **exclusivamente** la representación o **atributos** utilizados y el método `alta()` especificado en la interfaz. Puedes hacer uso de la clase `Nodo<E>` del anexo.

```
public class EnlazadoClub implements Club {

    // atributos

    public boolean alta(Abonado ab) throws NullPointerException {

    }
}
```

**Importante**: si para implementar el método alta haces uso de algún otro método de la interfaz `Club`, debes implementarlo también.