

Taller 2

Fecha: Agosto de 2024

Indicador de logro a medir: Aplicar los conceptos del manejo dinámico de la memoria en el desarrollo de una aplicación con interfaz gráfica de usuario basada en el lenguaje **Java** y un IDE adecuado.

NOTAS:

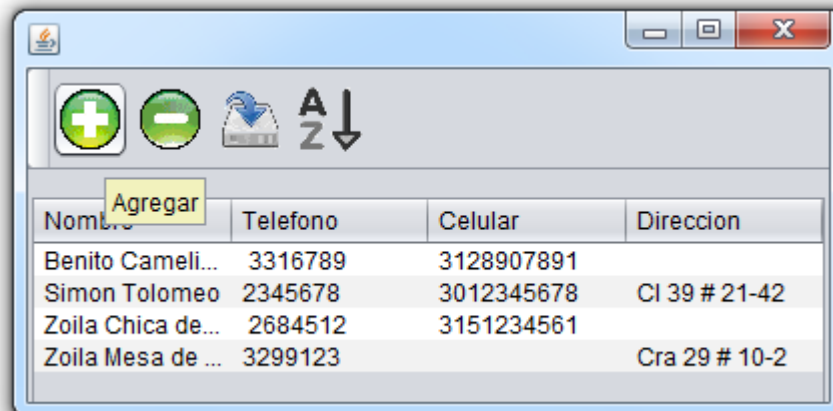
- Este taller se debe hacer como preparación para el quiz. En ningún caso representará una calificación.
- Se entregan ejercicios resueltos como ejemplo para el desarrollo de los demás.

Elaborar el diagrama de clases básico (sin las clases correspondientes a la interface de usuario según el lenguaje de implementación) y la respectiva aplicación en un lenguaje orientado a objetos para los siguientes enunciados:

1. Implementar un directorio de contactos personal que permita realizar las operaciones básicas de actualización y consulta:
 - Agregar
 - Modificar
 - Eliminar
 - Listar
 - Ordenar

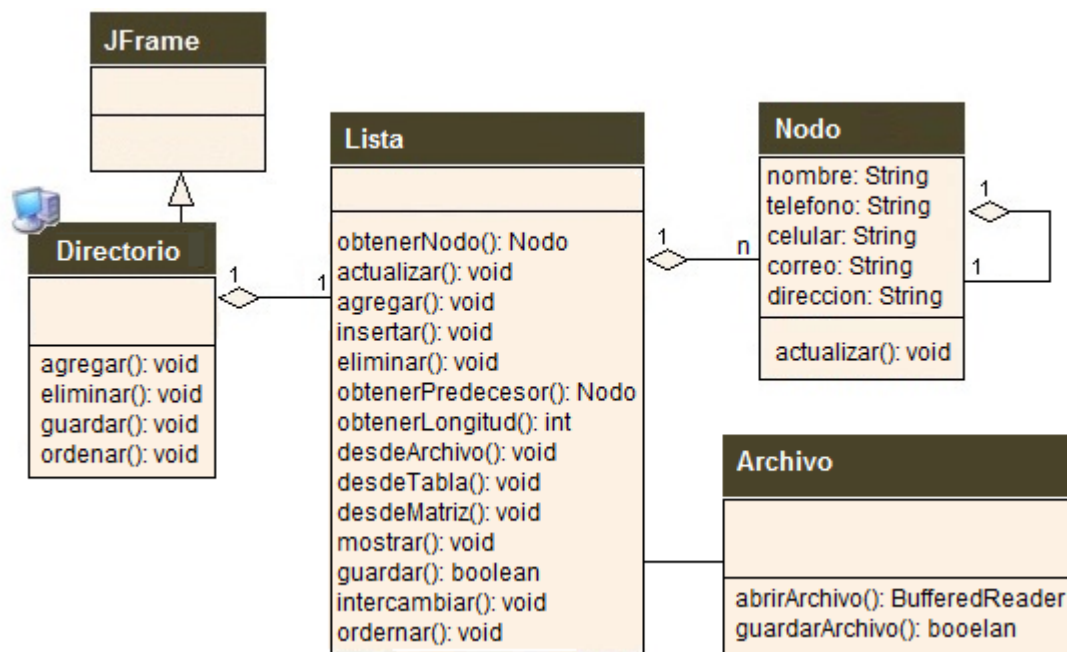
La información deberá almacenarse en un archivo plano y cargarse en memoria en una lista ligada.

La ejecución podría lucir así:



R/

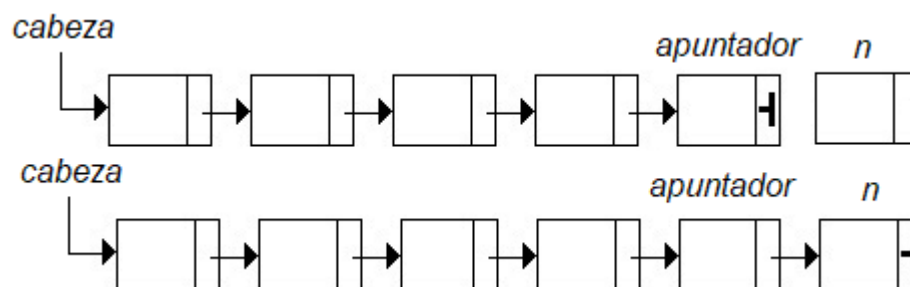
- El modelado bajo el paradigma Orientado a Objetos se ilustra en el siguiente diagrama de clases:



Para comprender este diagrama y el ejercicio, es importante tener en cuenta los siguientes fundamentos:

Agregar Nodo

Para agregar un nodo, este se realiza la final de la lista, requiriendo hacer un recorrido desde el nodo *Cabeza* hasta el último nodo, para hacer que el apuntador de este último nodo sea hacia el nuevo nodo:

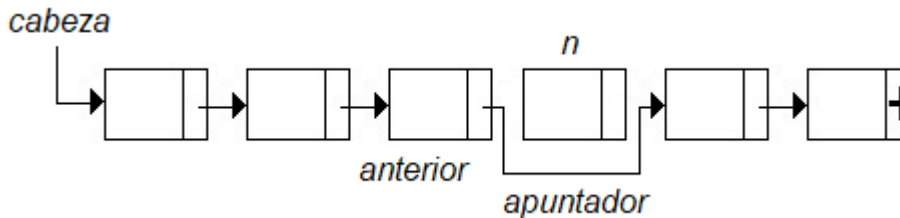


```

public void agregar(Nodo n) {
    if (n != null) {
        if (cabeza == null) {
            cabeza = n;
        } else {
            Nodo apuntador = cabeza;
            while (apuntador.siguiente != null) {
                apuntador = apuntador.siguiente;
            }
            apuntador.siguiente = n;
            n.siguiente = null;
        }
    }
}
  
```

Eliminar Nodo

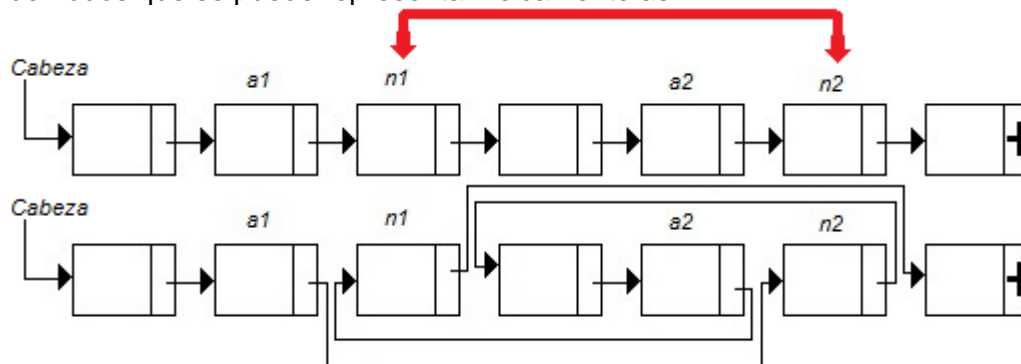
Para eliminar un nodo, se debe obtener el nodo que apunta a éste (denominado *anterior*) para hacer que apunte al siguiente del nodo a eliminar:



```
public void eliminar(Nodo n) {
    if (n != null && cabeza != null) {
        //Buscar el nodo
        boolean encontrado = false;
        Nodo apuntador = cabeza;
        Nodo anterior = null;
        while (apuntador != null && !encontrado) {
            if (apuntador == n) {
                encontrado = true;
            } else {
                anterior = apuntador;
                apuntador = apuntador.siguiente;
            }
        }
        if (encontrado) {
            if (anterior == null) {
                cabeza = apuntador.siguiente;
            } else {
                anterior.siguiente = apuntador.siguiente;
            }
        }
    }
}
```

Intercambio de Nodos

Para el ordenamiento de los datos en la lista ligada se requiere una operación de **Intercambio** de nodos que se puede representar físicamente así:



Donde:

a1 Nodo antecesor al primer nodo a intercambiar

- n1** Primer nodo a intercambiar
- a2** Nodo antecesor al segundo nodo a intercambiar
- n2** Segundo nodo a intercambiar

Lo cual equivale a las siguientes instrucciones:

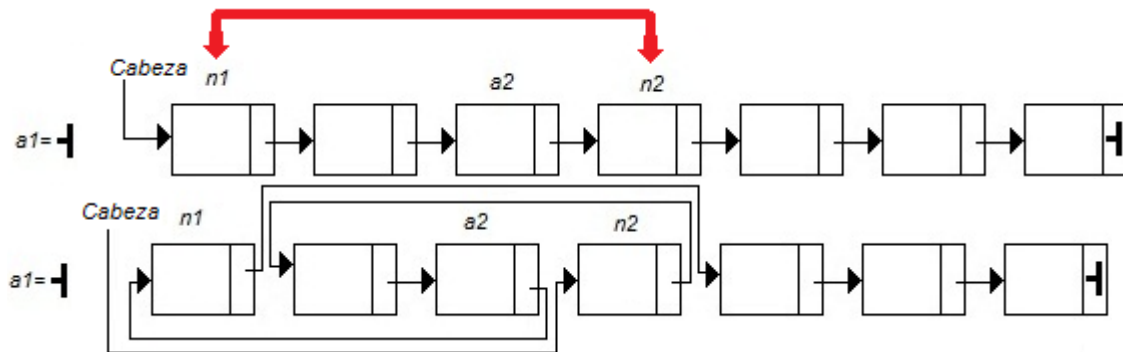
```

a1.siguiente = n2;
//Se guarda temporalmente el apuntador siguiente del segundo nodo
t = n2.siguiente;
n2.siguiente = n1.siguiente;
a2.siguiente = n1;
n1.siguiente = t;

```

El asunto es que se pueden presentar situaciones donde no se puedan ejecutar todas las anteriores instrucciones o se deba prever algunas situaciones:

a. El nodo **n1** es el nodo **Cabeza**



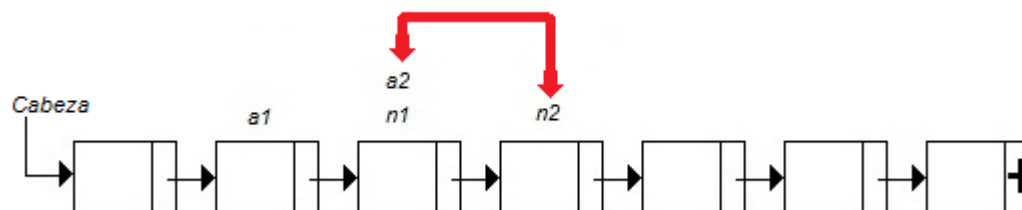
Se debe condicionar la asignación del nodo siguiente al antecesor del nodo **n1** (que en este caso no existe), así como cambiar el nodo **cabeza**.

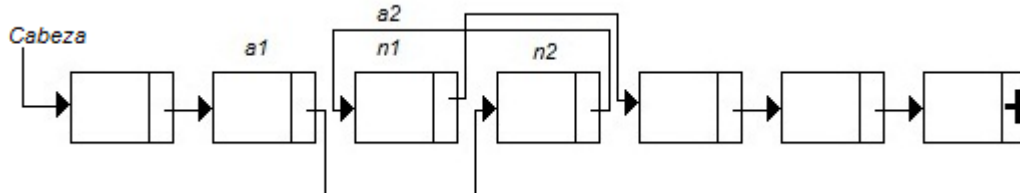
```

if (a1 != null) {
    a1.siguiente = n2;
} else {
    cabeza = n2;
}
//Se guarda temporalmente el apuntador siguiente del segundo nodo
t = n2.siguiente;
n2.siguiente = n1.siguiente;
a2.siguiente = n1;
n1.siguiente = t;

```

b. El antecesor del nodo **n2** es el nodo **n1**

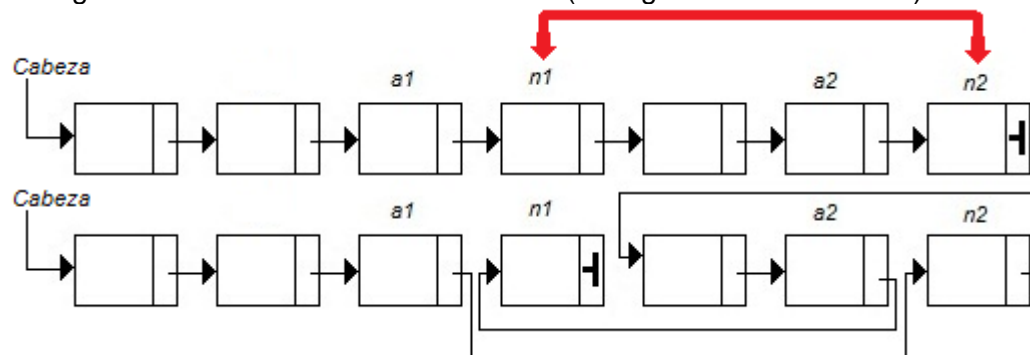




Se condiciona adicionalmente la asignación de los siguientes nodos del nodo **n2** y de su antecesor **a2** para evitar redundancia de asignaciones:

```
if (a1 != null) {
    a1.siguiente = n2;
} else {
    cabeza = n2;
}
//Se guarda temporalmente el apuntador siguiente del segundo nodo
t = n2.siguiente;
if (n1 != a2) {
    n2.siguiente = n1.siguiente;
    a2.siguiente = n1;
}
else
{
    n2.siguiente = n1;
}
n1.siguiente = t;
```

c. El segundo nodo **n2** es el último de la lista (Su siguiente nodo es nulo):



El código no requiere ningún cambio

Finalmente, el siguiente sería el código completo del método para realizar el intercambio de nodos:

```
//metodo para intercambio de nodos
public void intercambiar(Nodo n1, Nodo n2,
    Nodo a1, Nodo a2) {
    if (cabeza != null && n1 != n2
        && n1 != null && n2 != null) {
        if (a1 != null) {
            a1.siguiente = n2;
        } else {
            cabeza = n2;
        }
        //Se guarda temporalmente el apuntador siguiente del segundo nodo
        Nodo t = n2.siguiente;
```

```

    if (n1 != a2) {
        n2.siguiente = n1.siguiente;
        a2.siguiente = n1;
    }
    else
    {
        n2.siguiente = n1;
    }
    n1.siguiente = t;
}
}

```

Se puede observar en este código que toda la operación de intercambio se condiciona a que haya nodos en la lista, que los nodos a intercambiar no sean los mismos, y que los nodos no sean nulos.

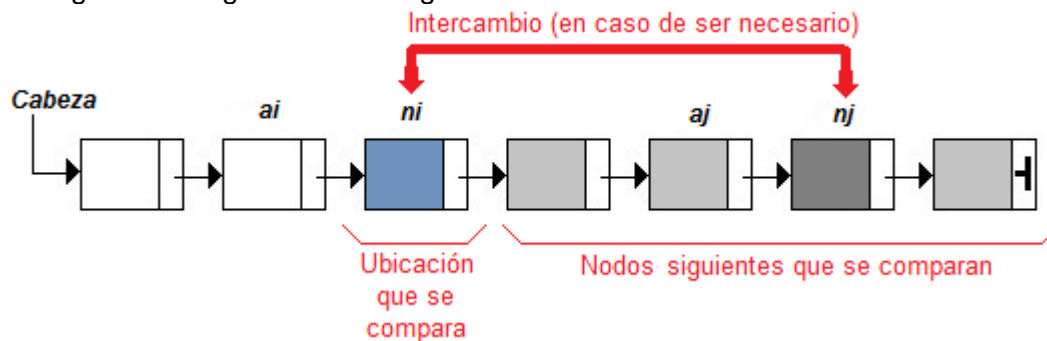
Ordenamiento de los Nodos de la Lista

Tomando como base el método de ordenamiento de la burbuja, se comparan los valores almacenados en los nodos de la siguiente manera:

Se inicia con el nodo **cabeza**, el cual se compara contra los nodos restantes (Lo que realmente se compara es alguno de los valores del nodo). Para que la lista quede ordenada ascendentemente, se verifica que el valor del primer nodo sea mayor que el del segundo, y en caso afirmativo, se hace el intercambio de nodos. Al final de estas comparaciones se garantiza que en la primera posición quede el nodo con el menor valor.

El anterior ciclo de comparaciones se repite para los demás nodos de la lista, hasta llegar al penúltimo nodo que será el último que tenga nodos restantes con quién compararse (el último nodo de la lista).

La siguiente imagen ilustra el algoritmo en un momento dado



Donde:

- ai** Nodo antecesor al nodo en la ubicación que se está comparando
- ni** Nodo en la ubicación que se está comparando
- aj** Nodo antecesor de alguno de los restantes nodos que se compara
- nj** Alguno de los restantes nodos que se compara

El siguiente código resuelve el ordenamiento de los nodos:

```

//metodo para ordenar los nodos
public void ordenar() {
    Nodo ni = cabeza;
    Nodo ai = null;
    while (ni.siguiente != null) {
        Nodo nj = ni.siguiente;
        Nodo aj = ni;
    }
}

```

```

while (nj != null) {
    if (ni.nombre.compareTo(nj.nombre) > 0) {
        intercambiar(ni, nj, ai, aj);
        Nodo t=ni;
        ni=nj;
        nj=t;
    }
    aj = nj;
    nj = nj.siguiente;
}
ai = ni;
ni = ni.siguiente;
}

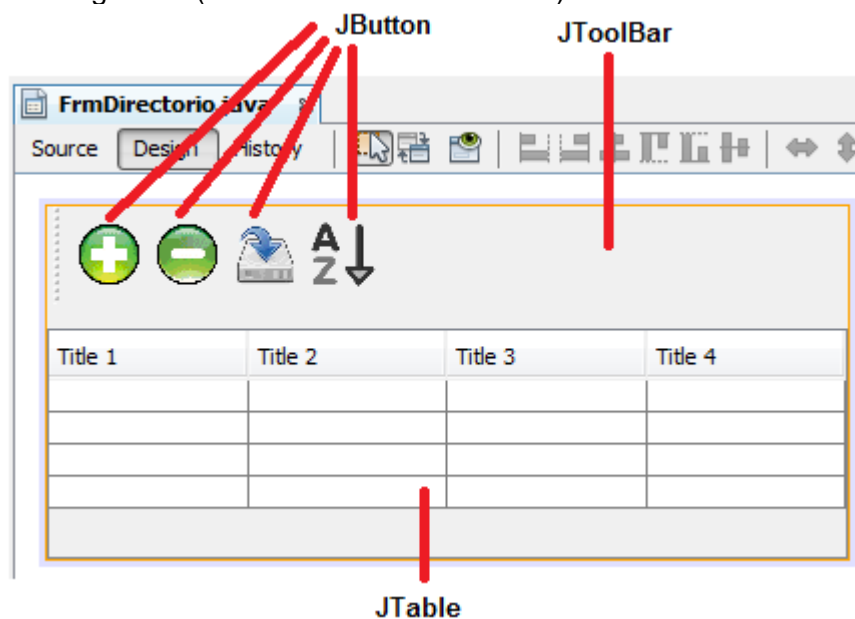
```

Se puede observar lo siguiente en este código:

- La condición para el primer ciclo busca evitar que se llegue al último nodo. Esto permite que siempre se pueda iniciar el primer nodo restante (**nj**) en el siguiente
- En caso de intercambio de los nodos, también se debe hacer intercambio de las variables apuntadoras (**ni** y **nj**)
- El ordenamiento se hará por el valor almacenado en el campo **nombre** de los nodos.

Programa

Para la implementación del aplicativo en *Java* se debe comenzar con el diseño de un formulario como el siguiente (Utilizando el *IDE Netbeans*):



Este formulario corresponde a la clase *Directorio* del anterior diagrama de clases. Al ser un formulario, hereda de la clase *JFrame*. La siguiente tabla relaciona los objetos a añadir con las propiedades cuyos valores deben ser cambiados:

Tipo Control	Nombre	Otras Propiedades
JLabel	<i>jToolBar1</i>	
JButton	<i>btnAgregar</i>	toolTipText = "Agregar" icon= "Agregar.gif"

	<i>btnEliminar</i>	toolTiptext = " <i>Eliminar</i> " icon= " <i>Eliminar.gif</i> "
	<i>btnGuardar</i>	toolTiptext = " <i>Guardar</i> " icon= " <i>Guardar.gif</i> "
	<i>btnOrdenar</i>	toolTiptext = " <i>Ordenar</i> " icon= " <i>Ordenar.gif</i> "
JTable	<i>tblDirectorio</i>	

De acuerdo al modelo de clases planteado, se debe editar la clase *Nodo* la cual tendrá la funcionalidad asociada a la estructura base de la lista ligada. Esta se compondrá de las siguientes propiedades:

Nodo
<i>nombre</i> : String <i>telefono</i> : String <i>celular</i> : String <i>correo</i> : String <i>direccion</i> : String
<i>actualizar()</i> : void

- *nombre*: El nombre del contacto
- *telefono*: El número de teléfono fijo del contacto
- *celular*: El número de teléfono móvil del contacto
- *correo*: El dato del correo del contacto
- *dirección*: La dirección física del contacto

Y el siguiente método:

- *actualizar()*: El cual permite asignar valores a las propiedades del objeto

El siguiente es el código completo de la clase en *Java*:

```
public class Nodo {

    String nombre;
    String telefono;
    String celular;
    String direccion;
    String correo;
    Nodo siguiente;

    public Nodo() {
        nombre = "";
        telefono = "";
        celular = "";
        direccion = "";
        correo="";
        siguiente = null;
    }

    public Nodo(String nombre,
                String telefono,
                String celular,
                String direccion,
                String correo) {
        this.nombre = nombre;
        this.telefono = telefono;
        this.celular = celular;
        this.direccion = direccion;
        this.correo=correo;
        siguiente = null;
    }
}
```



```

public void actualizar(String nombre,
    String telefono,
    String celular,
    String direccion,
    String correo) {
    this.nombre = nombre;
    this.telefono = telefono;
    this.celular = celular;
    this.direccion = direccion;
    this.correo=correo;
}
}

```

Ahora bien, la clase *Lista*, la cual implementa toda la funcionalidad relacionada con la lista ligada en sí, se compondrá de los siguientes métodos:

Lista
obtenerNodo(): Nodo actualizar(): void agregar(): void insertar(): void eliminar(): void obtenerPredecesor(): Nodo obtenerLongitud(): int desdeArchivo(): void desdeTabla(): void desdeMatriz(): void mostrar(): void guardar(): boolean intercambiar(): void ordenar(): void

- *obtenerNodo()*: Devuelve el nodo ubicado en una posición específica
- *actualizar()*: Permite cambiar los valores de los atributos de un nodo ubicado en una posición específica
- *agregar()*: Permite agregar un nuevo nodo a la lista, ubicándolo al final
- *insertar()*: Permite insertar un nodo en la lista después de un nodo específico
- *eliminar()*: Permite quitar un nodo de la lista
- *obtenerPredecesor()*: Permite obtener el nodo que precede a otro en la lista
- *obtenerLongitud()*: Devuelve el total de nodos de la lista ligada
- *desdeArchivo()*: Permite crear la lista ligada con información proveniente de un archivo plano

- *desdeMatriz()*: Permite crear la lista ligada con información proveniente de una matriz
- *desdeTabla()*: Permite crear la lista ligada con información proveniente de una rejilla de datos
- *mostrar()*: Permite mostrar los datos de la lista en una rejilla de datos
- *guardar()*: Guarda los datos de la lista ligada en un archivo plano
- *intercambiar()*: Permite intercambiar dos nodos en la lista ligada
- *ordenar()*: Permite ordenar los nodos de la lista por el atributo del nombre mediante el algoritmo de ordenamiento de la burbuja

El siguiente es el código completo de la clase en *Java*:

```

import java.io.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.event.*;

public class Lista {

    Nodo cabeza;

    //Metodo constructor que crea una lista vacia

```

```
public Lista() {
    cabeza = null;
}

//Devuelve el nodo ubicado en una posicion
public Nodo obtenerNodo(int posicion) {
    int p = 0;
    Nodo apuntador = cabeza;
    while (apuntador != null && p != posicion) {
        apuntador = apuntador.siguiente;
        p++;
    }
    if (apuntador != null && p == posicion) {
        return apuntador;
    } else {
        return null;
    }
}

//cambia los valores de un nodo dada la posición
public void actualizar(int posicion,
    String nombre,
    String telefono,
    String celular,
    String direccion,
    String correo) {
    Nodo n = obtenerNodo(posicion);
    if (n != null) {
        n.actualizar(nombre, telefono, celular, direccion, correo);
    }
}

//Agrega un nodo al final de la lista
public void agregar(Nodo n) {
    if (n != null) {
        if (cabeza == null) {
            cabeza = n;
        } else {
            Nodo apuntador = cabeza;
            while (apuntador.siguiente != null) {
                apuntador = apuntador.siguiente;
            }
            apuntador.siguiente = n;
            n.siguiente = null;
        }
    }
}

//Inserta un nodo en medio de la lista
public void insertar(Nodo n, Nodo predecesor) {
    if (n != null && predecesor != null) {
        if (predecesor != null) {
            n.siguiente = predecesor.siguiente;
            predecesor.siguiente = n;
        } else {
            n.siguiente = cabeza;
        }
    }
}
```

```
        cabeza = n;
    }
}

//Elimina un nodo de la lista
public void eliminar(Nodo n) {
    if (n != null && cabeza != null) {
        //Buscar el nodo
        boolean encontrado = false;
        Nodo apuntador = cabeza;
        Nodo anterior = null;
        while (apuntador != null && !encontrado) {
            if (apuntador == n) {
                encontrado = true;
            } else {
                anterior = apuntador;
                apuntador = apuntador.siguiente;
            }
        }
        if (encontrado) {
            if (anterior == null) {
                cabeza = apuntador.siguiente;
            } else {
                anterior.siguiente = apuntador.siguiente;
            }
        }
    }
}

//Obtener nodo predecesor
public Nodo obtenerPredecesor(Nodo n) {
    Nodo predecesor = null;
    if (n != null && cabeza != null && !cabeza.equals(n)) {
        predecesor = cabeza;
        while (predecesor != null && !predecesor.siguiente.equals(n)) {
            predecesor = predecesor.siguiente;
        }
    }
    return predecesor;
}

//Devuelve el numero de nodos en la lista
public int obtenerLongitud() {
    int totalNodos = 0;
    Nodo apuntador = cabeza;
    while (apuntador != null) {
        totalNodos++;
        apuntador = apuntador.siguiente;
    }
    return totalNodos;
}

//Llena la lista desde un archivo plano
public void desdeArchivo(String nombreArchivo) {
    cabeza = null;
```

```
BufferedReader br = Archivo.abrirArchivo(nombreArchivo);
try {
    String linea = br.readLine();
    while (linea != null) {
        String[] textos = linea.split("\t");
        if (textos.length >= 5) {
            Nodo n = new Nodo(textos[0],
                               textos[1],
                               textos[2],
                               textos[3],
                               textos[4]);
            agregar(n);
        }
        linea = br.readLine();
    }
} catch (IOException ex) {
}

//Llena la lista desde una tabla
public void desdeTabla(JTable tbl) {
    DefaultTableModel dtm = (DefaultTableModel) tbl.getModel();
    String[][] datos = new String[dtm.getRowCount()][dtm.getColumnCount()];
    for (int i = 0; i < dtm.getRowCount(); i++) {
        for (int j = 0; j < dtm.getColumnCount(); j++) {
            datos[i][j] = (String) dtm.getValueAt(i, j);
        }
    }
    desdeMatriz(datos);
}

//Llena la lista desde una matriz de textos
public void desdeMatriz(String[][] datos) {
    cabeza = null;
    for (int i = 0; i < datos.length; i++) {
        Nodo n = new Nodo(datos[i][0],
                           datos[i][1],
                           datos[i][2],
                           datos[i][3],
                           datos[i][4]);
        agregar(n);
    }
}

//Muestra la lista en una tabla
public void mostrar(JTable tbl) {
    int filas = obtenerLongitud();
    String[][] datos = new String[filas][5];
    Nodo apuntador = cabeza;
    filas = 0;
    while (apuntador != null) {
        datos[filas][0] = apuntador.nombre;
        datos[filas][1] = apuntador.telefono;
        datos[filas][2] = apuntador.celular;
        datos[filas][3] = apuntador.direccion;
    }
}
```

```
        datos[filas][4] = apuntador.correo;
        apuntador = apuntador.siguiente;
        filas++;
    }
    DefaultTableModel dtm = new DefaultTableModel(datos,
        new String[]{"Nombre", "Telefono", "Celular", "Direccion",
"Correo"});
    dtm.addTableModelListener(
        new TableModelListener() {

            public void tableChanged(TableModelEvent evt) {
                int p = evt.getFirstRow();
                DefaultTableModel dtm = (DefaultTableModel)
evt.getSource();

                actualizar(p,
                    (String) dtm.getValueAt(p, 0),
                    (String) dtm.getValueAt(p, 1),
                    (String) dtm.getValueAt(p, 2),
                    (String) dtm.getValueAt(p, 3),
                    (String) dtm.getValueAt(p, 4));
            }
        });
    tbl.setModel(dtm);
}

public boolean guardar(String nombreadarchivo) {
    int tn = obtenerLongitud();
    if (tn > 0) {
        String[] lineas = new String[tn];
        Nodo n = cabeza;
        int i = 0;
        while (n != null) {
            lineas[i] = n.nombre + "\t" + n.telefono + "\t" + n.celular +
"\t"
                + n.direccion + "\t" + n.correo;
            i++;
            n = n.siguiente;
        }
        return Archivo.guardarArchivo(nombreadarchivo, lineas);
    } else {
        return false;
    }
}

//metodo para intercambio de nodos
public void intercambiar(Nodo n1, Nodo n2,
    Nodo a1, Nodo a2) {
    if (cabeza != null && n1 != n2
        && n1 != null && n2 != null) {
        if (a1 != null) {
            a1.siguiente = n2;
        } else {
            cabeza = n2;
        }
        //Se guarda temporalmente el apuntador siguiente del segundo nodo
        Nodo t = n2.siguiente;
```

```
        if (n1 != a2) {
            n2.siguiente = n1.siguiente;
            a2.siguiente = n1;
        } else {
            n2.siguiente = n1;
        }
        n1.siguiente = t;
    }
}

//metodo para ordenar los nodos
public void ordenar() {
    Nodo ni = cabeza;
    Nodo ai = null;
    while (ni.siguiente != null) {
        Nodo nj = ni.siguiente;
        Nodo aj = ni;
        while (nj != null) {
            if (ni.nombre.compareTo(nj.nombre) > 0) {
                intercambiar(ni, nj, ai, aj);
                Nodo t = ni;
                ni = nj;
                nj = t;
            }
            aj = nj;
            nj = nj.siguiente;
        }
        ai = ni;
        ni = ni.siguiente;
    }
}
}
```

En este código es importante observar estos detalles:

- Los componentes de la lista ligada son objetos de la clase *Nodo*
- El archivo plano es cargado en memoria en objetos de la clase *BufferedReader* el cual permite recorrer su contenido para luego ser utilizado para crear nodos de la lista ligada
- La rejilla de datos para mostrar la información de la lista ligada se implementa mediante objetos de la clase *JTable*, la cual requiere estratégicamente pasar la información de la lista ligada a una matriz. Por otra parte la implementación de un *JTable* que permita la actualización de sus datos requiere sobrecargar el método *tableChanged()*
- El guardado de la lista ligada requiere un paso intermedio de crear un vector de cadenas de texto con la información contenida en los nodos de la lista

La clase *Archivo* incluye la funcionalidad para operar con archivos planos. Los métodos que se implementan son los siguientes:

Archivo
abrirArchivo(): BufferedReader guardarArchivo(): boolean

- *abrirArchivo()*: Permite abrir el contenido de un archivo plano y dejarlo disponible en memoria para su lectura
- *guardarArchivo()*: Permite guardar en un archivo la información contenida en un vector de cadenas de texto

El siguiente es el código completo de la clase en Java:

```
import java.io.*;

public class Archivo {

    public static BufferedReader Abrir(String nombreArchivo) {

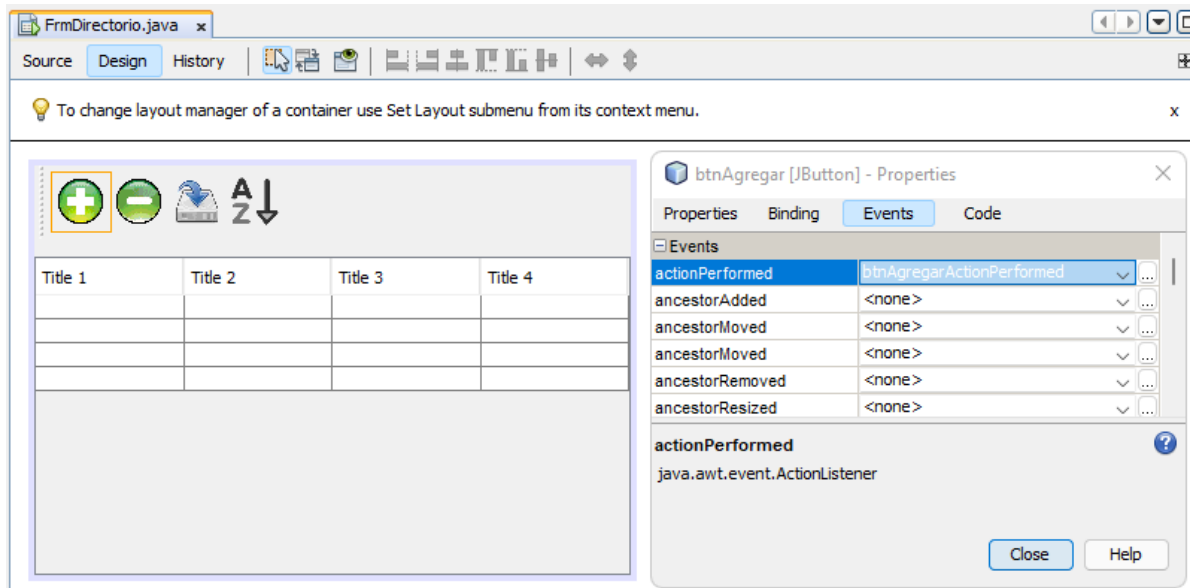
        //verificar la existencia del archicvo
        File f = new File(nombreArchivo);
        if (f.exists()) {
            try {
                FileReader fr = new FileReader(f);
                return new BufferedReader(fr);
            } catch (Exception ex) {
                return null;
            }
        } else {
            return null;
        }
    }
}
```

En este código es importante observar estos detalles:

- La clase *File* permite verificar la existencia de un archivo mediante el método *exists()*. Esto sirve para poder abrir un archivo sin generar errores de no existencia
- Un archivo plano se lee mediante objetos de la clase *FileReader*. El resultado de la lectura se almacena en objetos de la clase *BufferedReader* quien almacena el contenido del archivo en memoria
- Un archivo plano se guarda mediante objetos de la clase *BufferedWriter*. Para ello se tienen los métodos *write()*, *newLine()* y *close()* para grabar, agregar nueva línea y cerrar respectivamente

Ahora bien, para continuar con la codificación, se deben programar los métodos *agregar()*, *eliminar()*, *guardar()* y *ordenar()* de la clase *Directorio* y que corresponderán a los eventos de los botones de comando agregados al formulario.

Para agregar un nuevo contacto (y por ende un nuevo nodo a la lista) se hará mediante un método que responde al evento ***actionPerformed*** del botón de comando ***btnAgregar***.

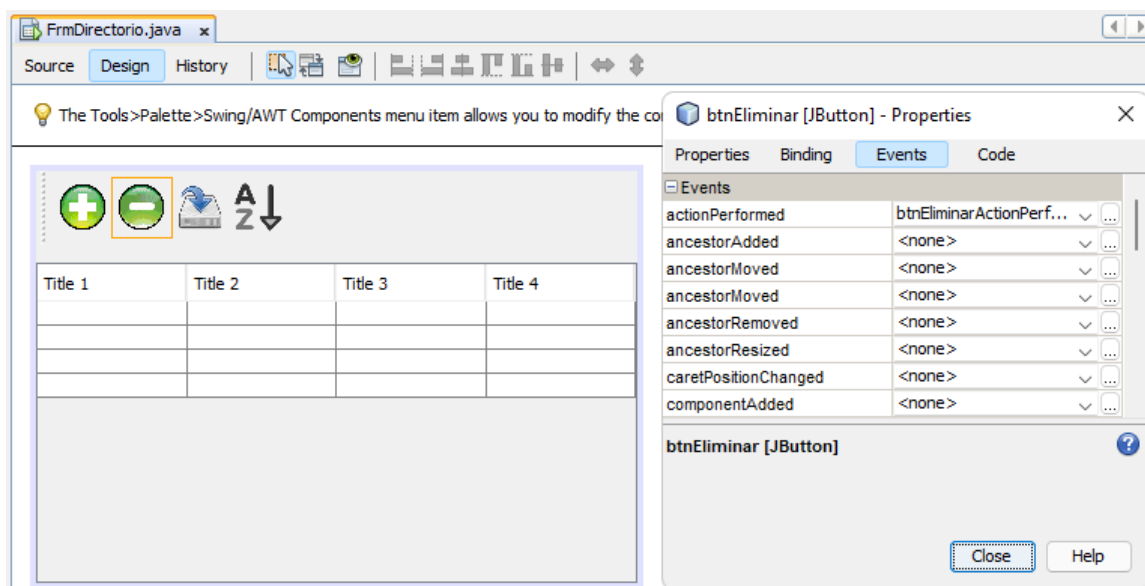


El código respectivo sería el siguiente:

```
private void btnAgregarActionPerformed(java.awt.event.ActionEvent evt) {
    l.agregar(new Nodo());
    l.mostrar(tblDirectorio);
}
```

En este código, se agrega un nodo vacío a la lista ligada y luego se muestra en la rejilla de datos.

Para eliminar un contacto existente, (y por ende quitar el nodo de la lista) se hará mediante un método que responde al evento **actionPerformed** del botón de comando **btnEliminar**.



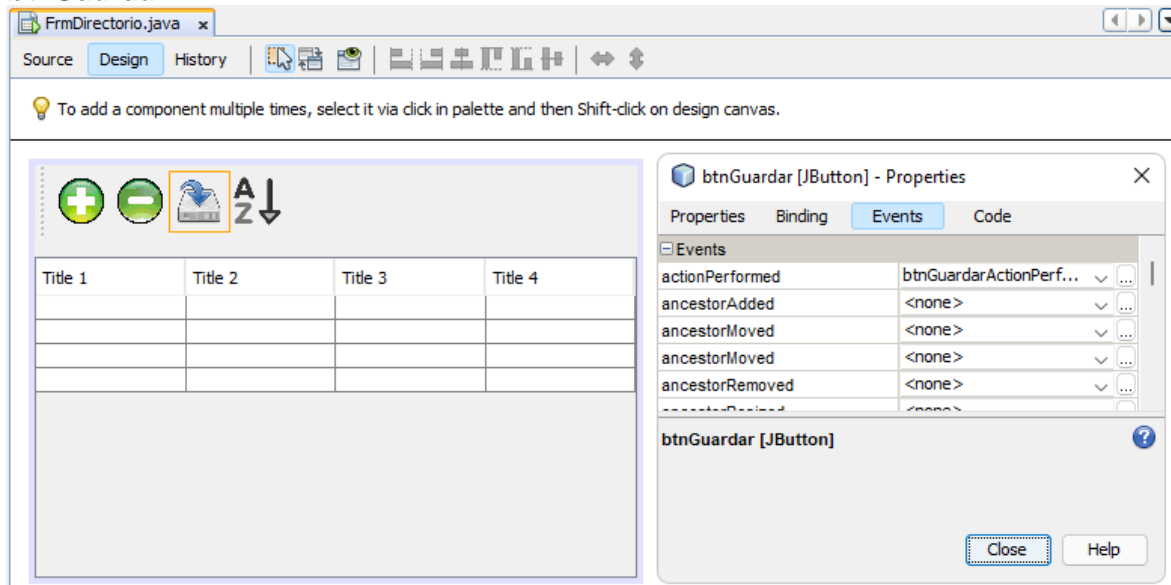
El código respectivo sería el siguiente:


```
private void btnEliminarActionPerformed(java.awt.event.ActionEvent evt) {
    if (tblDirectorio.getSelectedRow() >= 0)
        l.eliminar(l.obtenerNodo(tblDirectorio.getSelectedRow()));
    l.mostrar(tblDirectorio);
}
```

En este código primero se verifica si hay una fila seleccionada en la rejilla de datos. Con esta posición luego se obtiene el nodo que se va a retirar de la lista.

Una vez removido, se actualiza de nuevo el despliegue de la lista en la rejilla de datos

Para guardar la información de los contactos (la que está almacenada en los nodos de la lista) se hará mediante un método que responde al evento **actionPerformed** del botón de comando **btnGuardar**.



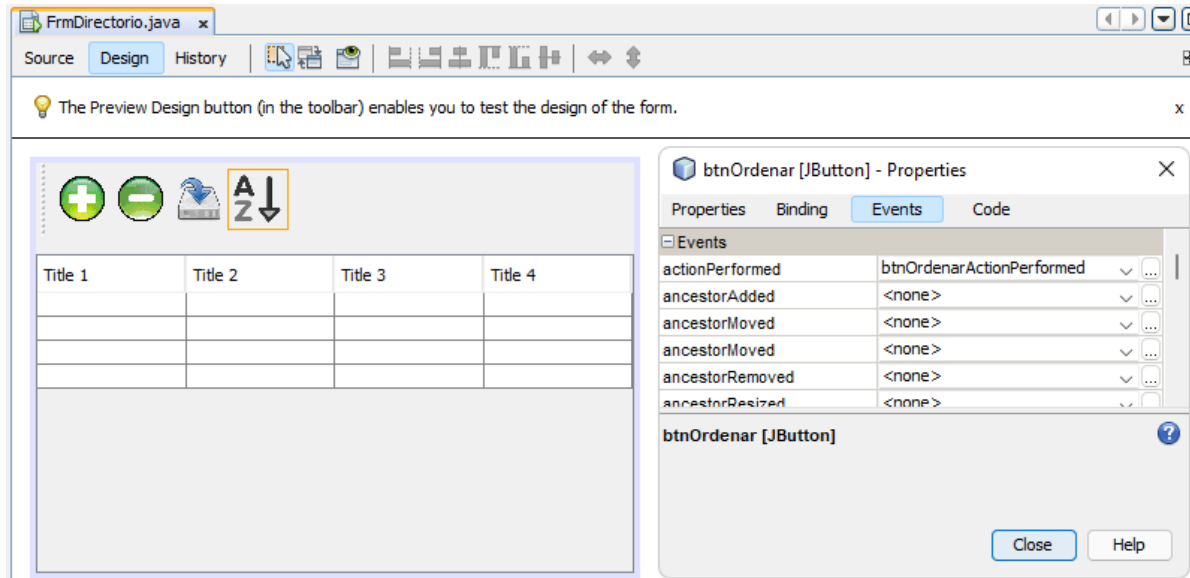
El código respectivo sería el siguiente:

```
private void btnGuardarActionPerformed(java.awt.event.ActionEvent evt) {
    String nombreArchivo = System.getProperty("user.dir")
        + "/src/Datos/Datos.txt";
    if (l.guardar(nombreArchivo))
        JOptionPane.showMessageDialog(new JFrame(), "Los datos fueron guardados");
    else
        JOptionPane.showMessageDialog(new JFrame(), "No fueron guardados los datos");
}
```

En este código se crea la ruta completa del archivo combinando la ruta del programa con la carpeta y nombre de archivo de destino.

Luego se llama al método que guarda la lista en un archivo plano, mostrándose un mensaje de haberse podido efectuar el proceso de guardado.

Para ordenar la información de los contactos (y por ende cambiar el orden de los nodos de la lista) se hará mediante un método que responde al evento **actionPerformed** del botón de comando **btnOrdenar**:



El código respectivo sería el siguiente:

```
private void btnOrdenarActionPerformed(java.awt.event.ActionEvent evt) {
    l.ordenar();
    l.mostrar(tblDirectorio);
}
```

En este código básicamente se llama al método que ordena la lista ligada por el atributo nombre y luego se actualiza el despliegue en la rejilla de datos.

Por último, el código que corresponde el método constructor de la clase y que permite desplegar el estado inicial de la lista a partir de la información almacenada en el archivo plano, es el siguiente:

```
public class FrmDirectorio extends JFrame {

    Lista l=new Lista();

    public FrmDirectorio() {
        initComponents();

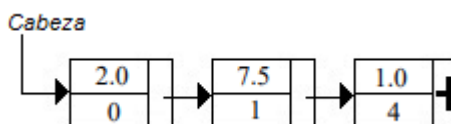
        String nombreArchivo = System.getProperty("user.dir")
            + "/src/Datos/Datos.txt";

        l.desdeArchivo(nombreArchivo);
        l.mostrar(tblDirectorio);
    }

    ...
}
```

2. Un polinomio puede ser representado por una lista ligada en la que cada nodo contiene un coeficiente (de tipo REAL), el valor del exponente (un valor CARDINAL) y un puntero hacia el siguiente elemento.

Las listas ligadas que representan los polinomios estarán ordenadas en forma ascendente por el grado de los nodos. Por ejemplo, el polinomio $2 + 7.5x + x^4 = 0$ se representa así:



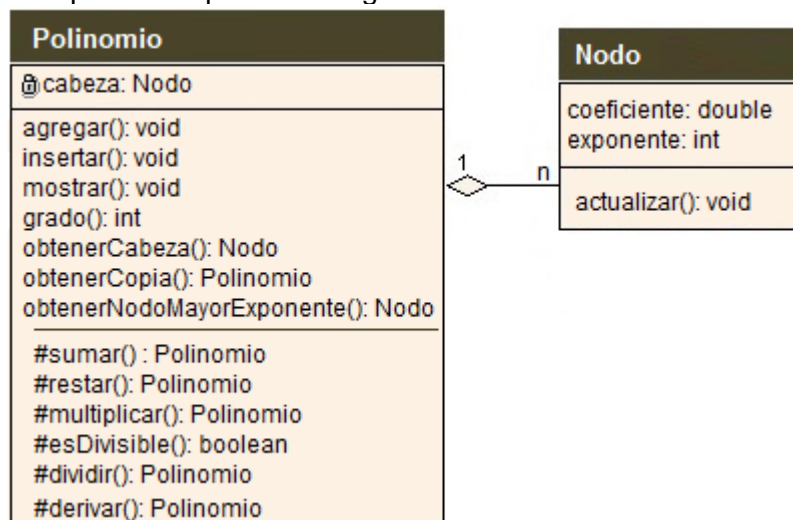
Escribir un aplicativo que permita:

- Crear polinomios
- Añadir términos a un polinomio
- Mostrar un polinomio en pantalla
- Sumar dos polinomios
- Restar dos polinomios
- Calcular el producto de dos polinomios
- Calcular el producto de un polinomio por un monomio
- Calcular la derivada de un polinomio dado

Las operaciones de suma, resta, producto y derivada deben generar listas nuevas (los nodos que las forman deben crearse)

R/

Para solucionar lo requerido se plantea el siguiente modelo de clases básico:



Donde se plantea el polinomio como una clase que representa una lista ligada de nodos.

La clase *Nodo* sigue el esquema tradicional de los componentes de una lista ligada (información y apuntador al siguiente nodo), sólo que su estructura interior la componen los atributos *coeficiente* y *exponente*. El código sería el siguiente:

```
public class Nodo {  
  
    double coeficiente;  
    int exponente;  
    Nodo siguiente;  
  
    public Nodo() {  
        coeficiente = 0;  
        exponente = 0;  
        siguiente = null;  
    }  
  
    public Nodo(double coeficiente,  
                int exponente) {  
        this.coeficiente = coeficiente;  
        this.exponente = exponente;  
        siguiente = null;  
    }  
  
    public void actualizar(double coeficiente,  
                           int exponente) {  
        this.coeficiente = coeficiente;  
        this.exponente = exponente;  
    }  
}
```

Se procederá ahora programar la clase que contendrá tanto la definición de la funcionalidad correspondiente a la lista de monomios (mediante variables y métodos de clase), como la funcionalidad para operar con varias instancias de ella misma (mediante variables y métodos estáticos):

Inicialmente la clase tendrá solo un atributo privado:

Polinomio
@ cabeza: Nodo
agregar(): void insertar(): void mostrar(): void grado(): int obtenerCabeza(): Nodo obtenerCopia(): Polinomio obtenerNodoMayorExponente(): Nodo

- *cabeza*: Instancia de *Nodo* que corresponde al primer elemento del polinomio (el de menor grado)

Y los siguientes métodos (además del método constructor):

- *agregar()*: El cual permite insertar un nuevo nodo a la lista ligada, de acuerdo al orden que le corresponde el exponente. Si el nodo del exponente ya existe, entonces se suma el coeficiente al existente. Si la suma da 0, se retira el nodo
- *insertar()*: Permite agregar un nodo después de otro que se debe especificar.
- *mostrar()*: Permite desplegar la escritura del polinomio, basado en los nodos que componen la lista
- *grado()*: Devuelve el exponente contenido en el último nodo de la lista
- *obtenerCabeza()*: Devuelve el primer nodo de la lista
- *obtenerCopia()*: Crea una nueva instancia de la clase a partir de los nodos que componen la actual
- *obtenerNodoMayorExponente()*: Devuelve el último nodo de la lista

El código sería el siguiente:

```
import java.awt.Font;
import javax.swing.JLabel;

public class Polinomio {

    Nodo cabeza;

    //Metodo constructor que crea un Polinomio vacio
    public Polinomio() {
        cabeza = null;
    }

    //Agrega un nodo en la ubicación que le corresponde en el polinomio
    public void agregar(Nodo n) {
        if (n != null) {
            if (cabeza == null) {
                cabeza = n;
            } else {
                Nodo apuntador = cabeza;
                Nodo predecedor = null;
                int encontrado = 0;
                while (apuntador != null && encontrado == 0) {
                    if (n.exponente == apuntador.exponente) {
                        encontrado = 1;
                    } else if (n.exponente < apuntador.exponente) {
                        encontrado = 2;
                    } else {
                        predecedor = apuntador;
                        apuntador = apuntador.siguiente;
                    }
                }
                if (encontrado == 1) {
                    //se acumulan los coeficientes
                    double coeficiente = apuntador.coeficiente +
n.coeficiente;

                    if (coeficiente == 0) {
                        //quitar el nodo
                        if (predecedor == null) {
                            cabeza = apuntador.siguiente;
                        } else {
                            predecedor.siguiente = apuntador.siguiente;
                        }
                    } else {
                        apuntador.coeficiente = coeficiente;
                    }
                } else {
                    insertar(n, predecedor);
                }
            }
        }
    }

    //Inserta un nodo en medio de la lista
    public void insertar(Nodo n, Nodo predecesor) {
        if (n != null) {
```

```
        if (predecesor != null) {
            n.siguiente = predecesor.siguiente;
            predecesor.siguiente = n;
        } else {
            n.siguiente = cabeza;
            cabeza = n;
        }
    }
}

//Muestra el polinomio como un texto en un JLabel
public void mostrar(JLabel lbl) {
    String espacio = "&nbsp;";
    String lineal = "";
    String linea2 = "";
    Nodo apuntador = cabeza;
    while (apuntador != null) {
        String texto = String.valueOf(apuntador.coeficiente) + " X";
        if (apuntador.coeficiente >= 0) {
            texto = "+" + texto;
        }
        lineal += String.format("%0" + texto.length() + "d",
0).replace("0", espacio);
        linea2 += texto;

        texto = String.valueOf(apuntador.exponente);
        linea2 += String.format("%0" + texto.length() + "d",
0).replace("0", espacio);
        lineal += texto;

        apuntador = apuntador.siguiente;
    }
    if (!linea2.equals("")) {
        linea2 += " = 0";
    }
    lbl.setFont(new Font("Courier New", Font.PLAIN, 12));
    lbl.setText("<html>" + lineal + "<br>" + linea2 + "</html>");
}

//Devuelve el mayor exponente
public int grado() {
    if (cabeza != null) {
        Nodo apuntador = cabeza;
        while (apuntador.siguiente != null) {
            apuntador = apuntador.siguiente;
        }
        return apuntador.exponente;
    }
    return -1;
}

//Devuelve el primer nodo
public Nodo obtenerCabeza() {
    return cabeza;
}
```

```
//Crea una copia del actual polinomio
public Polinomio obtenerCopia() {
    Polinomio p = new Polinomio();
    if (cabeza != null) {
        Nodo apuntador = cabeza;
        while (apuntador != null) {
            Nodo n = new Nodo(apuntador.coeficiente,
                                apuntador.exponente);
            p.agregar(n);
            apuntador = apuntador.siguiente;
        }
    }
    return p;
}

public Nodo obtenerNodoMayorExponente() {
    if (cabeza != null) {
        Nodo apuntador = cabeza;
        while (apuntador.siguiente != null) {
            apuntador = apuntador.siguiente;
        }
        return apuntador;
    }
    return null;
}
}
```

Ahora se agregará a esta clase los siguientes métodos estáticos:

Polinomio
#sumar(): Polinomio
#restar(): Polinomio
#multiplicar(): Polinomio
#esDivisible(): boolean
#dividir(): Polinomio
#derivar(): Polinomio

- *sumar()*: Devuelve un nuevo polinomio correspondiente a la suma de los dos polinomios entrantes
- *restar()*: Devuelve un nuevo polinomio correspondiente a la resta de los dos polinomios entrantes
- *multiplicar()*: Devuelve un nuevo polinomio correspondiente al producto de los dos polinomios entrantes. Este método tendrá una sobrecarga que permite multiplicar un polinomio por un monomio
- *esDivisible()*: Indica si un monomio (representado por un objeto *Nodo*) es divisible por otro
- *dividir()*: Devuelve dos nuevos polinomios (Cociente y Residuo) correspondiente al resultado de la división de los dos polinomios entrantes
- *derivar()*: Devuelve un polinomio resultado de derivar el polinomio entrante

El código sería el siguiente:

```
public class Polinomio {
    ...

    //*****Métodos estáticos
    public static Polinomio Sumar(Polinomio p1, Polinomio p2) {
        Polinomio pR = new Polinomio();

        Nodo apuntador1 = p1.obtenerCabeza();
```

```
Nodo apuntador2 = p2.obtenerCabeza();

while (!(apuntador1 == null && apuntador2 == null)) {
    Nodo n = new Nodo();
    if (apuntador1 != null && apuntador2 != null
        && apuntador1.exponente == apuntador2.exponente) {
        n.exponente = apuntador1.exponente;
        n.coeficiente = apuntador1.coeficiente +
apuntador2.coeficiente;
        apuntador1 = apuntador1.siguiente;
        apuntador2 = apuntador2.siguiente;
    } else if ((apuntador2 == null)
        || (apuntador1 != null && apuntador1.exponente <
apuntador2.exponente)) {
        n.exponente = apuntador1.exponente;
        n.coeficiente = apuntador1.coeficiente;
        apuntador1 = apuntador1.siguiente;
    } else {
        n.exponente = apuntador2.exponente;
        n.coeficiente = apuntador2.coeficiente;
        apuntador2 = apuntador2.siguiente;
    }

    pR.agregar(n);
}
return pR;
}

public static Polinomio Restar(Polinomio p1, Polinomio p2) {
    Polinomio pR = new Polinomio();

    Nodo apuntador1 = p1.obtenerCabeza();
    Nodo apuntador2 = p2.obtenerCabeza();

    while (!(apuntador1 == null && apuntador2 == null)) {
        Nodo n = new Nodo();
        if (apuntador1 != null && apuntador2 != null
            && apuntador1.exponente == apuntador2.exponente) {
            n.exponente = apuntador1.exponente;
            n.coeficiente = apuntador1.coeficiente -
apuntador2.coeficiente;
            apuntador1 = apuntador1.siguiente;
            apuntador2 = apuntador2.siguiente;
        } else if ((apuntador2 == null)
            || (apuntador1 != null && apuntador1.exponente <
apuntador2.exponente)) {
            n.exponente = apuntador1.exponente;
            n.coeficiente = apuntador1.coeficiente;
            apuntador1 = apuntador1.siguiente;
        } else {
            n.exponente = apuntador2.exponente;
            n.coeficiente = -apuntador2.coeficiente;
            apuntador2 = apuntador2.siguiente;
        }
        if (n.coeficiente != 0) {
            pR.agregar(n);
        }
    }
}
```



```
        }
    }
    return pR;
}

public static Polinomio Multiplicar(Polinomio p1, Polinomio p2) {
    Polinomio pR = new Polinomio();

    return pR;
}

public static Polinomio Multiplicar(Polinomio p, Nodo n) {
    Polinomio pR = new Polinomio();

    return pR;
}

public static boolean esDivisible(Nodo n1, Nodo n2) {
    return n1.coeficiente % n2.coeficiente == 0 && n1.exponente >=
n2.exponente;
}

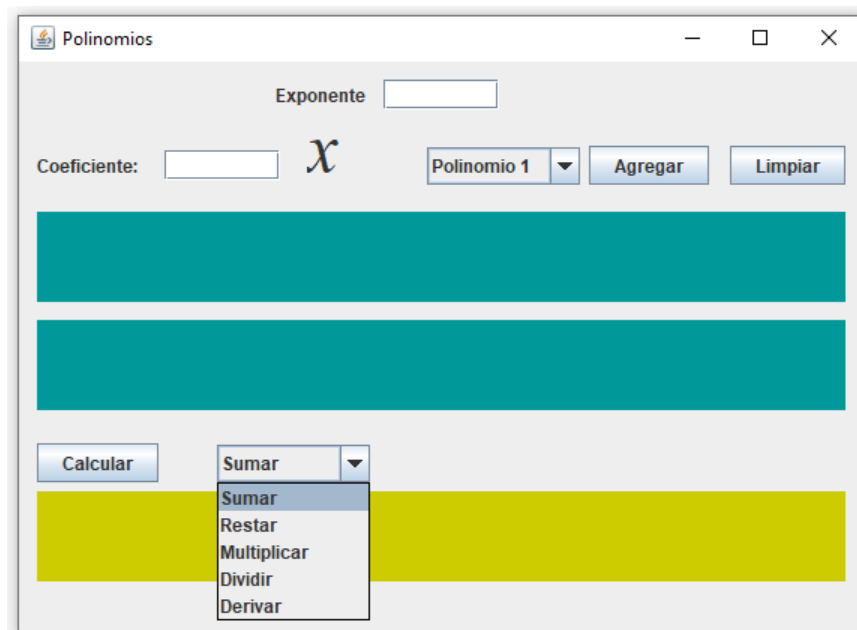
public static Polinomio[] Dividir(Polinomio p1, Polinomio p2) {
    Polinomio[] pR = new Polinomio[2];

    return pR;
}

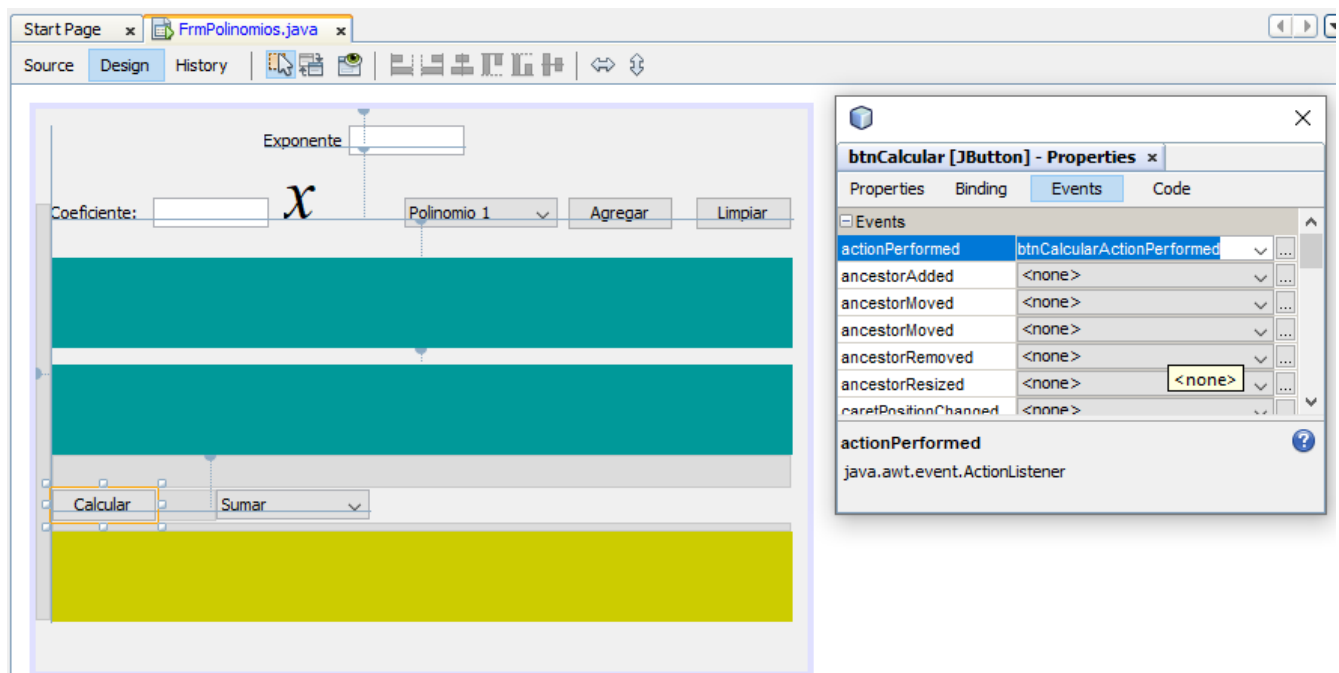
public static Polinomio Derivar(Polinomio p) {
    Polinomio pR = new Polinomio();

    return pR;
}
}
```

Para hacer uso práctico de la anterior funcionalidad, se puede diseñar un formulario como el de la siguiente ilustración, donde se pueda editar 2 polinomios y permitir calcular las operaciones:



Para poder ejecutar cualquiera de los cálculos escogidos mediante la lista desplegable, se hará mediante un método que responde al evento **actionPerformed** del botón de comando **btnCalcular**.



El código respectivo sería el siguiente:

```
Polinomio p1, p2, pR;
```

```
private void btnCalcularActionPerformed(java.awt.event.ActionEvent evt)
    switch (cmbOperacion.getSelectedIndex()) {
        case 0:
```

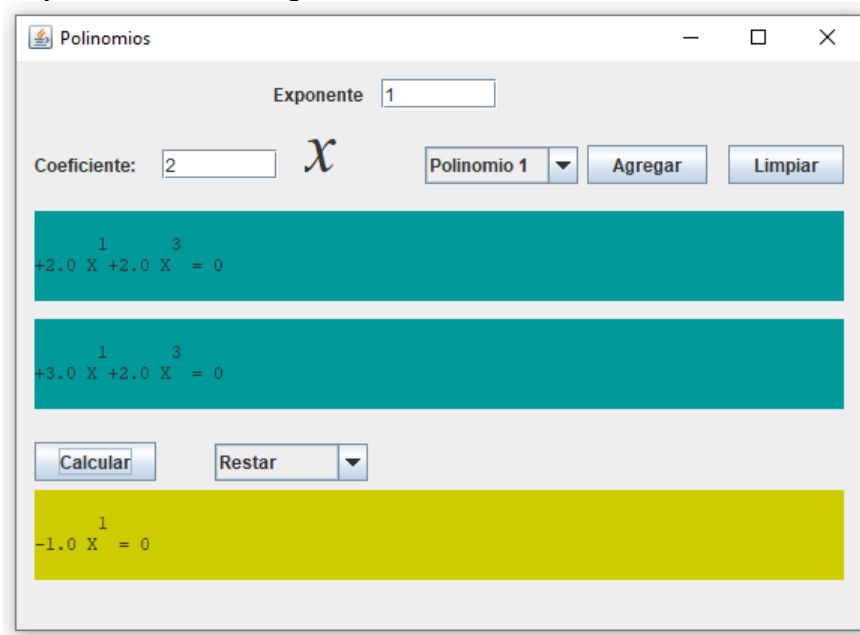
```

        pR = Polinomio.Sumar(p1, p2);
        pR.mostrar(lblPolinomioR);
        break;
    case 1:
        pR = Polinomio.Restar(p1, p2);
        pR.mostrar(lblPolinomioR);
        break;
    case 2:
        pR = Polinomio.Multiplicar(p1, p2);
        pR.mostrar(lblPolinomioR);
        break;
    case 3:
        Polinomio[] pRD=Polinomio.Dividir(p1, p2);
        pRD[0].mostrar(lblPolinomioR);
        break;
    case 4:
        if(cmbPolinomio.getSelectedIndex()==0) {
            pR = Polinomio.Derivar(p1);
        } else {
            pR = Polinomio.Derivar(p2);
        }
        pR.mostrar(lblPolinomioR);
        break;
    }
}

```

En este código, se verifica cual es la operación seleccionada por el usuario, luego se llaman los métodos correspondientes a cada operación y se muestra el polinomio resultante en el objeto *JLabel* destinado para ello.

Un ejemplo de ejecución sería el siguiente:



- Una matriz dispersa es una matriz donde muchos de sus elementos tienen valor cero. En predicción meteorológica, por ejemplo, es necesario manejar matrices de este tipo con

grandes dimensiones. Si se utiliza una matriz bidimensional se está desperdiciando una gran cantidad de memoria. Una solución a este problema es utilizar listas enlazadas.

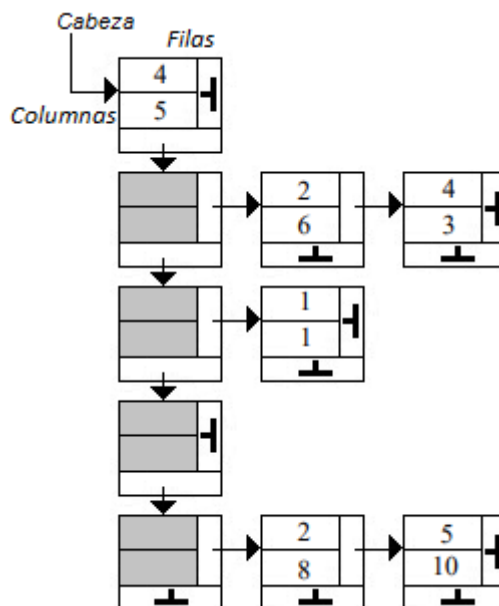
Una matriz se representa por un registro de tres campos:

- El primero es el número total de filas de la matriz, el segundo el de columnas y el tercero una lista ligada vertical con tantos elementos como filas tenga la matriz.
- Cada nodo de la lista vertical es un puntero a una lista enlazada con los nodos no nulos de la fila. Estos nodos tienen dos campos, además del puntero: la columna a la cual corresponde el dato y el dato. Las listas horizontales están ordenadas de menor a mayor según el contenido del primer campo (la columna a la cual corresponde el dato).

Por ejemplo, la siguiente matriz:

$$\begin{bmatrix} 0 & 6 & 0 & 3 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & 10 \end{bmatrix}$$

Se puede representar mediante la siguiente lista ligada:



Escribir un módulo de biblioteca donde se defina el tipo MATRIZ y las siguientes operaciones:

- **Crear:** toma como argumentos el número total de filas y columnas de una matriz y devuelve una matriz vacía
- **Asignar:** toma como argumentos una matriz, una fila, una columna y un valor. Asigna este valor a la correspondiente posición de la matriz. Para ello hay que buscar si existe un nodo en la matriz correspondiente a la posición. Si es así, se asigna el nuevo valor. En otro caso se crea un nuevo nodo con los valores adecuados y se enlaza en su posición correspondiente
- **Elemento:** toma una matriz, una fila y una columna y devuelve el valor del elemento que ocupa dicha posición en la matriz. Si no existe el correspondiente nodo en la matriz, se devolverá cero
- **Mostrar Matriz:** Muestra el contenido de la matriz en pantalla

- **Leer Matriz:** permite crear una matriz y leer sus componentes desde la pantalla

Como aplicación de la biblioteca, escribir un programa que lea dos matrices y calcule su producto y lo muestre por pantalla

4. En el primer ejercicio, implementar la funcionalidad de ordenamiento, utilizando el algoritmo *QuickSort*. Mostrar de alguna manera que este algoritmo es más rápido que el algoritmo de la burbuja
5. En el ejercicio 2, calcular la división de dos polinomios.

Para dividir polinomios donde el dividendo y divisor son polinomios con por lo menos dos términos cada uno, se sugiere los siguientes pasos:

- Divida el primer término del dividendo entre el primer término del divisor para determinar el primer término del cociente
- El primer término del cociente obtenido en el paso anterior multiplíquelo a cada término del divisor
- Reste el producto anterior del dividendo y se obtiene un nuevo polinomio, que será el residuo
- Repita el proceso con el nuevo polinomio hasta que no se pueda hacer una división