Momento II: Guía de Proyecto.

2023

Semestre 3

Ingeniería de Sistemas
Creado por: Valentina Ojeda y Sarah Montes

Requisitos

Herramientas para la guía

Para una mejor comprensión de esta guía, se recomienda tener instalada la versión más reciente de **Apache NetBeans**. También es esencial contar con las herramientas de desarrollo para **Java (JDK**) en su versión 8 para garantizar un funcionamiento estable de otro de los componentes vitales, **JavaFX**, que es una plataforma requerida para este proyecto.

Por medio de las previamente mencionadas, se crea un proyecto compuesto por ventanas que contienen varios elementos capaces de ejercer multiples funciones a describir mas adelante. Por esto también se estara usando **Scene Builder.** A continuación se muestran las herramientas con sus respectivos vinculos a sus paginas con más información aparte de la descarga.

Página de Java Fx

Página de Apache NetBeans

Página de JDK 8

Página de Scene Builder

Nota:

Dado el caso no se cuenten con las herramientas de desarrollo para Java en versión 8, se necesitara crear una cuenta en Oracle para acceder al archivo para realizar la descarga.

Cabe resaltar que se requiere un computador portatil o un ordenador, pues estos programas poseen una interfaz compleja y sirven multiples funciones, además de tener previos conocimientos de pseudocodigo y lenguaje de programación java, pues la plataforma a utilizar depende vitalmente del uso de este lenguaje, que compone a esta.

Indicio

1. Introducción

Propositos y utilidad de la guía

A continuación, la guía utilizara instrucciones y gráficas para explicar el proceso de creación un sistema de tienda o catalogo sencillo, capaz de realizar múltiples y distintas tareas como si el usuario accediera como un administrador, que provee más información aparte de los elementos principales; este provee información sobre los usuarios que han hecho previo uso de dicho catálogo.

El sistema, como se menciono anteriormente, sera creado por medio de Netbeans y otras herramientas, con las cuales se creara un proyecto compuesto por clases ordenadas según la funcion que cumplan, además que se estará utilizando un creador de ventanas (**Scene Builder**), que permite que se facilite aún más la creación del sistema, pues con solo java fx tomaría mucho tiempo.

En terminos más específicos, se tendrán 2 ventanas principales, con tablas, botones y otros controles del usuario, que ejecutan funciones tales como:

- **Introducción de datos de un producto:** ID, fecha de lote y vencimiento, nombre y precio que corresponden a un producto, estos datos quedrán almacenados para posteriormente realizar multiples operaciones que describen más adelante.
- **Operaciones:** Sacar promedio, el precio mayor/menor, sumar, reorganizar los datos según como el usuaro lo requisite por medio de las herramientas que provee el sistema que se crea.
- **Muestra de datos y tabla:** Según la información que se vaya introduciendo al sistema.

2. Clases

Métodos, Atributos y Paso a paso

2.1. Package: Código

2.1.1. Pila 

Es una implementación de una pila (**stack**) en Java. En una pila, los elementos se almacenan y se acceden de manera que el último elemento agregado es el primero en ser retirado, lo que se conoce como el principio "Last In, First Out" (LIFO).

Atributos:

- private Stack<tienda> Producto: Esta es una pila que almacena objetos del tipo tienda.

Constructor:

- public Pila(): Este es el constructor de la clase Pila que inicializa la pila de productos.

Métodos:

- public void setPush(tienda Tienda): Este método permite agregar un objeto de tipo tienda a la pila.

- public tienda setPop(): Este método permite quitar y devolver el elemento superior de la pila. Si la pila está vacía, devuelve null.

- public boolean Buscarid(String IdP): Este método busca un producto por su identificador y devuelve true si lo encuentra, de lo contrario, devuelve false.

- public boolean BuscarNombre(String NomP): Este método busca un producto por su nombre y devuelve true si lo encuentra, de lo contrario, devuelve false.

- public tienda getInfoId (String IdP): Este método devuelve el objeto de tipo tienda correspondiente a un identificador dado.

Métodos:

- public tienda getInfoFechaLote (LocalDate FechaL): Este método devuelve el objeto de tipo tienda correspondiente a una fecha de lote dada.

- public tienda getInfoNombre (String NomP): Este método devuelve el objeto de tipo tienda correspondiente a un nombre dado.

- public tienda getInfoFechaV (LocalDate FechaVencimiento): Este método devuelve el objeto de tipo tienda correspondiente a una fecha de vencimiento dada.

- public tienda getInfoPrecio (double Precio): Este método devuelve el objeto de tipo tienda correspondiente a un precio unitario dado.

- public double Promedio(): Este método calcula el promedio de los precios unitarios de los productos en la pila.

- public List<tienda> getMayorPromedio(): Este método devuelve una lista de productos con precios unitarios mayores que el promedio.

- public List<tienda> getMenorPromedio(): Este método devuelve una lista de productos con precios unitarios menores que el promedio.

- public tienda getMayorPrecio(): Este método devuelve el producto con el precio unitario más alto en la pila.

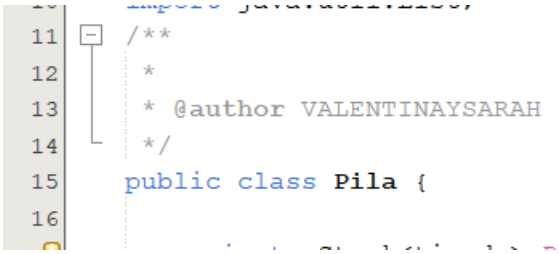
- public tienda getMenorPrecio(): Este método devuelve el producto con el precio unitario más bajo en la pila.

- public List<tienda> getProductosPorMesLote(int MesL): Este método devuelve una lista de productos con fechas de lote correspondientes a un mes dado.

- public List<tienda> getMesVencimiento(int MesV): Este método devuelve una lista de productos con fechas de vencimiento correspondientes a un mes dado.

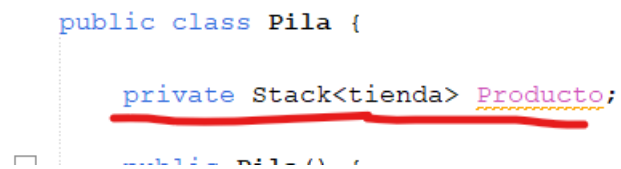
Paso a Paso

1. **Declaración de la Clase:** Empieza nombrando la clase Pila.



```
/**
 *
 * @author VALENTINAYSARAH
 */
public class Pila {
```

2. **Declaración del Atributo:** Declara el atributo Producto como una pila de objetos de tipo tienda.



```
public class Pila {

    private Stack<tienda> Producto;
```

3. **Constructor de la Clase**: Crea el constructor de la clase Pila para inicializar el atributo Producto.

```
public Pila() {
    this.Producto = new Stack<>();
}
```

4. **Método setPush**: Define el método setPush para agregar un objeto tienda a la pila.

```
public void setPush(tienda Tienda) {
    Producto.push(item: Tienda);
}
```

5. **Método setPop:** Implementa el método setPop para eliminar y devolver el elemento en la parte superior de la pila, o null si está vacía.

```
public tienda setPop() {
    if (!Producto.isEmpty()) {
        return Producto.pop();
    } else {
        return null;
    }
}
```

Paso a Paso

6. **Método Buscarid:** Crea el método Buscarid para buscar un objeto tienda en la pila basándose en un valor de identificación (IdP).

```
33      public boolean Buscarid(String IdP) {
34          for (tienda Tienda : Producto) {
            if (Tienda.getIdProducto()== IdP) {
36              return true;
37          }
38      }
39      return false;
40  }
```

7. **Método BuscarNombre:** Define el método BuscarNombre para buscar un objeto tienda en la pila basándose en el nombre del producto (NomP).

```
42      public boolean BuscarNombre(String NomP) {
43          for (tienda Tienda : Producto) {
44              if (Tienda.getNombreProducto() == null ? NomP == null : Tienda.getNombreProducto().equals(anObject: NomP)) {
45                  return true;
46              }
47          }
48          return false;
49      }
```

8. **Método getInfoId:** Este método busca un objeto tienda en la pila en función de un valor de identificación (IdP) y devuelve ese objeto si se encuentra.

```
50      public tienda getInfoId (String IdP){
51      for (tienda Tienda : Producto) {
52
            if (Tienda.getIdProducto()== IdP) {
54
55              return Tienda;
56          }
57      }
58      return null;
59  }
```

9. **Método getInfoNombre:** Este método busca un objeto tienda en la pila basándose en el nombre del producto (NomP) y devuelve ese objeto si se encuentra.

```
60      public tienda getInfoNombre (String NomP){
61      for (tienda Tienda : Producto) {
62
63          if (Tienda.getNombreProducto() == null ? NomP == null : Tienda.getNombreProducto().equals(anObject: NomP)) {
64
65              return Tienda;
66          }
67      }
68      return null;
69  }
```

Paso a Paso

10. **Método getInfoFechaLote:** Busca un objeto tienda en la pila en función de una fecha de lote (FechaL) y devuelve ese objeto si se encuentra.

```
public tienda getInfoFechaLote (LocalDate FechaL){
for (tienda Tienda : Producto) {
    if (Tienda.getFechaLoteD().getValue().equals(obj:FechaL)) {
        return Tienda;
    }
}
return null;
}
```

11. **Método getInfoFechaV:** Busca un objeto tienda en la pila en función de una fecha de vencimiento (FechaVencimiento) y devuelve ese objeto si se encuentra.

```
78      public tienda getInfoFechaV (LocalDate FechaVencimiento){
79      for (tienda Tienda : Producto) {
80          if (Tienda.getFechadeVencimiento().getValue().equals(obj:FechaVencimiento))
81              return Tienda;
82          }
83      }
84      return null;
85      }
```

12. **Método getInfoPrecio:** Busca un objeto tienda en la pila en función de un precio (Precio) y devuelve ese objeto si se encuentra.

```
86      public tienda getInfoPrecio (double Precio){
87      for (tienda Tienda : Producto) {
88
89              if (Tienda.getPrecioUnitario()==Precio) {
90
91                  return Tienda;
92              }
93          }
94          return null;
95      }
```

13. **Método Promedio:** Calcula el promedio de los precios unitarios de todos los productos en la pila y lo devuelve.

```
96      public double Promedio(){
97     double Suma = 0;
98
99          for (tienda Tienda : Producto) {
100             Suma= Tienda.getPrecioUnitario();
101         }
102
103         if (!Producto.isEmpty()) {
104             return Suma / Producto.size();
105         }
106
107         return 0;
108     }
```

Paso a Paso

14. **Método getMayorPromedio**: Devuelve una lista de objetos tienda que tienen un precio unitario mayor que el promedio de todos los productos en la pila.

```
109     public List<tienda> getMayorPromedio() {
110         double MayorPromedio = Promedio();
111         List<tienda> pMayorPromedio = new ArrayList<>();
112 
113         for (tienda Tienda : Producto) {
114             if (Tienda.getPrecioUnitario()> MayorPromedio) {
115                 pMayorPromedio.add(e: Tienda);
116             }
117         }
118 
119         return pMayorPromedio;
120     }
```

15. **Método getMenorPromedio**: Devuelve una lista de objetos tienda que tienen un precio unitario menor que el promedio de todos los productos en la pila.

```
121     public List<tienda> getMenorPromedio() {
122         double menorPromedio = Promedio();
123         List<tienda> pMenorPromedio = new ArrayList<>();
124 
125         for (tienda Tienda : Producto)
126           if (Tienda.getPrecioUnitario()< menorPromedio) {
127               pMenorPromedio.add(e: Tienda);
128           }
129         return pMenorPromedio;
130     }
```

16. **Método getMayorPrecio:** Busca y devuelve el objeto tienda con el precio unitario más alto en la pila.

```
131     public tienda getMayorPrecio() {
132         if (Producto.isEmpty()) {
133             return null;
134         }
135     tienda MayorPrecio = Producto.get(index: 0);
136 
137         for (tienda Tienda : Producto) {
138             if (Tienda.getPrecioUnitario()> MayorPrecio.getPrecioUnitario()) {
139                 MayorPrecio =Tienda;
140             }
141         }
142 
143         return MayorPrecio;
144     }
```

Paso a Paso

17. **Método getMenorPrecio:** Busca y devuelve el objeto tienda con el precio unitario más bajo en la pila.

```
145     public tienda getMenorPrecio() {
146         if (Producto.isEmpty()) {
147             return null;
148         }
149         tienda MenorPrecio = Producto.get(index: 0);
150 
151         for (tienda Tienda : Producto) {
152             if (Tienda.getPrecioUnitario()< MenorPrecio.getPrecioUnitario()) {
153                 MenorPrecio =Tienda;
154             }
155         }
156 
157         return MenorPrecio;
158     }
```

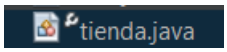
18. **Método getProductosPorMesLote:** Devuelve una lista de objetos tienda que tienen una fecha de lote correspondiente al mes especificado (MesL).

```
159     public List<tienda> getProductosPorMesLote(int MesL) {
160     List<tienda> MesLote = new ArrayList<>();
161     for (tienda Tienda : Producto) {
162         if (Tienda.getFechaLoteD().getValue().getMonthValue() == MesL) {
163             MesLote.add(e: Tienda);
164         }
165     }
166     return MesLote;
167     }
```

19. **Método getMesVencimiento**: Devuelve una lista de objetos tienda que tienen una fecha de vencimiento correspondiente al mes especificado (MesV).

```
168     public List<tienda> getMesVencimiento(int MesV) {
169     List<tienda> MesVencimiento = new ArrayList<>();
170     for (tienda Tienda : Producto) {
171         if (Tienda.getFechadeVencimiento().getValue().getMonthValue() == MesV)
172             MesVencimiento.add(e: Tienda);
173         }
174     }
175     return MesVencimiento;
176     }
177 
178 }
```

- Los métodos proporcionados en esta clase permiten acceder y manipular estos objetos de manera efectiva, lo que facilita la gestión de productos en el sistema. La estructura de pila (stack) asegura que los elementos se manejen siguiendo la regla de último en entrar, primero en salir, lo que puede ser útil en diversos contextos.

2.1.2. Tienda 

Representa objetos que describen productos en el sistema, y contiene varios atributos y métodos que permiten acceder y manipular información relacionada a dichos productos.

Atributos

- **IdProducto:** Representa una identificación única para el producto.
- **NombreProducto:** Almacena el nombre del producto.
- **FechaLoteD y FechadeVencimiento:** Son objetos de tipo ObjectProperty<LocalDate> que almacenan fechas relacionadas con el producto, como la fecha de lote y la fecha de vencimiento.
- **PrecioUnitario:** Guarda el precio unitario del producto**.**

Constructores

La clase tienda tiene dos constructores:

- El **primero** es un constructor sin argumentos que inicializa algunos de los atributos con valores predeterminados, como un valor de -1 para IdProducto, una cadena vacía para NombreProducto, 1 para PrecioUnitario, y las fechas actuales para FechaLoteD y FechadeVencimiento.
- El **segundo constructor** acepta argumentos para inicializar todos los atributos de la clase a partir de los valores proporcionados al crear un objeto tienda.

Métodos

- **Tienda:** proporciona métodos getters y setters para todos sus atributos. Esto permite acceder y modificar los valores de los atributos desde fuera de la clase.
- Los **getters** permiten obtener los valores de los atributos.
- Los **setters** permiten establecer nuevos valores para los atributos.

Paso a Paso

1. **Importa las clases necesarias:** Se importan las clases **LocalDate** y **SimpleObjectProperty** para poder trabajar con fechas y propiedades de objetos en el código. Se puede hacerlo al comienzo del archivo Java.

```
6   package Codigo;
7   import java.time.LocalDate;
8   import javafx.beans.property.ObjectProperty;
9   import javafx.beans.property.SimpleObjectProperty;
10
```

2. Se crea una instancia de la clase tienda utilizando el constructor con los respectivos argumentos.

```
15    public class tienda {
16    String IdProducto;
17    String NombreProducto;
18    public ObjectProperty<LocalDate> FechaLoteD;
19    public ObjectProperty<LocalDate> FechadeVencimiento;
20    double PrecioUnitario;
21
```

Entonces..

```
22  public tienda() {
23          int IdProducto=-1;
24          NombreProducto="";
25          PrecioUnitario=1;
26          FechaLoteD=FechadeVencimiento=new SimpleObjectProperty<>(t: LocalDate.now());
27      }
28
29      public tienda(String IdProducto, String NombreProducto, LocalDate FechaLoteD, LocalDate FechadeVencimiento, double PrecioUnitario) {
30          this.IdProducto = IdProducto;
31          this.NombreProducto = NombreProducto;
32          this.FechaLoteD = new SimpleObjectProperty<>(t: FechaLoteD);
33          this.FechadeVencimiento = new SimpleObjectProperty<>(t: FechadeVencimiento);
34          this.PrecioUnitario = PrecioUnitario;
35      }
36
37      public String getIdProducto() {
38          return IdProducto;
39      }
40
41      public void setIdProducto(String IdProducto) {
42          this.IdProducto = IdProducto;
43      }
44
45      public String getNombreProducto() {
46          return NombreProducto;
47      }
48
49      public void setNombreProducto(String NombreProducto) {
50          this.NombreProducto = NombreProducto;
51      }
52
53      public ObjectProperty<LocalDate> getFechaLoteD() {
54          return FechaLoteD;
55      }
```

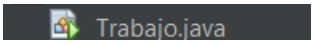
...

```
56
57      public void setFechaLoteD(ObjectProperty<LocalDate> FechaLoteD) {
58          this.FechaLoteD = FechaLoteD;
59      }
60
61      public ObjectProperty<LocalDate> getFechadeVencimiento() {
62          return FechadeVencimiento;
63      }
64
65      public void setFechadeVencimiento(ObjectProperty<LocalDate> FechadeVencimiento) {
66          this.FechadeVencimiento = FechadeVencimiento;
67      }
68
69      public double getPrecioUnitario() {
70          return PrecioUnitario;
71      }
72
73      public void setPrecioUnitario(double PrecioUnitario) {
74          this.PrecioUnitario = PrecioUnitario;
75      }
76
77
78
79  }
```

3. **Accede a los atributos de la instancia:** Una vez que creada la instancia de tienda, se puede acceder a sus atributos y métodos.

   - Una vez que has creada la instancia de **tienda**, se acceden a los atributos de la clase para obtener información sobre el producto. Esto permite trabajar con objetos de la clase tienda del programa que se esta creando.
   - Se ajustan los valores y atributos según las necesidades específicas de la aplicación y el usuario.

2.2. Package: JavaFxController

2.2.1. Clase Principal: Trabajo 

Esta clase es la clase principal de la aplicación y se encarga de inicializar la aplicación JavaFX.

Atributos:

No hay atributos específicos en esta clase aparte de los heredados de **Application**.

Métodos:

- **start(Stage stage):** Método override de Application que se encarga de iniciar la aplicación JavaFX y mostrar la ventana gráfica.
- **main(String[] args):** Método estático que sirve como punto de entrada de la aplicación JavaFX. Se ejecuta para iniciar la aplicación llamando a launch(args).

Paso a paso:

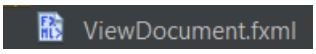
1. **Declaración de la clase:** Se define la clase Trabajo como una subclase de Application. Esto significa que Trabajo es una clase que extiende y personaliza la funcionalidad proporcionada por la clase Application.

```
16    public class Trabajo extends Application {
17
```

2. **Método start(Stage stage):** Este método start es una anulación (override) del método start de la clase Application. Se ejecuta cuando se inicia la aplicación JavaFX. Dentro de este método, se realizan las siguientes acciones:

    - Se carga la interfaz de usuario desde un archivo FXML llamado "ViewDocument.fxml".
    - Se crea una escena gráfica (Scene) que contiene la interfaz de usuario.
    - Se establece esta escena en la ventana (stage).
    - Finalmente, se muestra la ventana.

2.3. Package: View

2.3.1. Ventana: ViewDocument.fxml 

- Como esta es una aplicación distinta, no se estará describiendo los atributos y metodos, pues estos están previamente descritos en las otras clases.
- Se estará haciendo uso del software llamaddo “Scene builder”, que se halla al principio de la guía su respectiva página.

Paso a paso:

1. **Abre Scene Builder**: Inicia **Scene Builder**, ya descargado en la computadora.

2. **Selecciona un Diseño Base**: Se elige un diseño base para la ventana.

3. **Arrastra y Suelta Elementos:** Desde la barra de herramientas de la izquierda, arrastra y suelta elementos como botones, etiquetas o campos de texto en el área de diseño. Se colocan estos elementos en la posición deseada.

4. **Ajusta Propiedades**: Hacer clic en los elementos agregados para seleccionarlos y ajustar sus propiedades.

5. **Guarda el Diseño**: Ir a "Archivo" y selecciona "Guardar" para guardar el diseño en un archivo FXML en tu sistema.

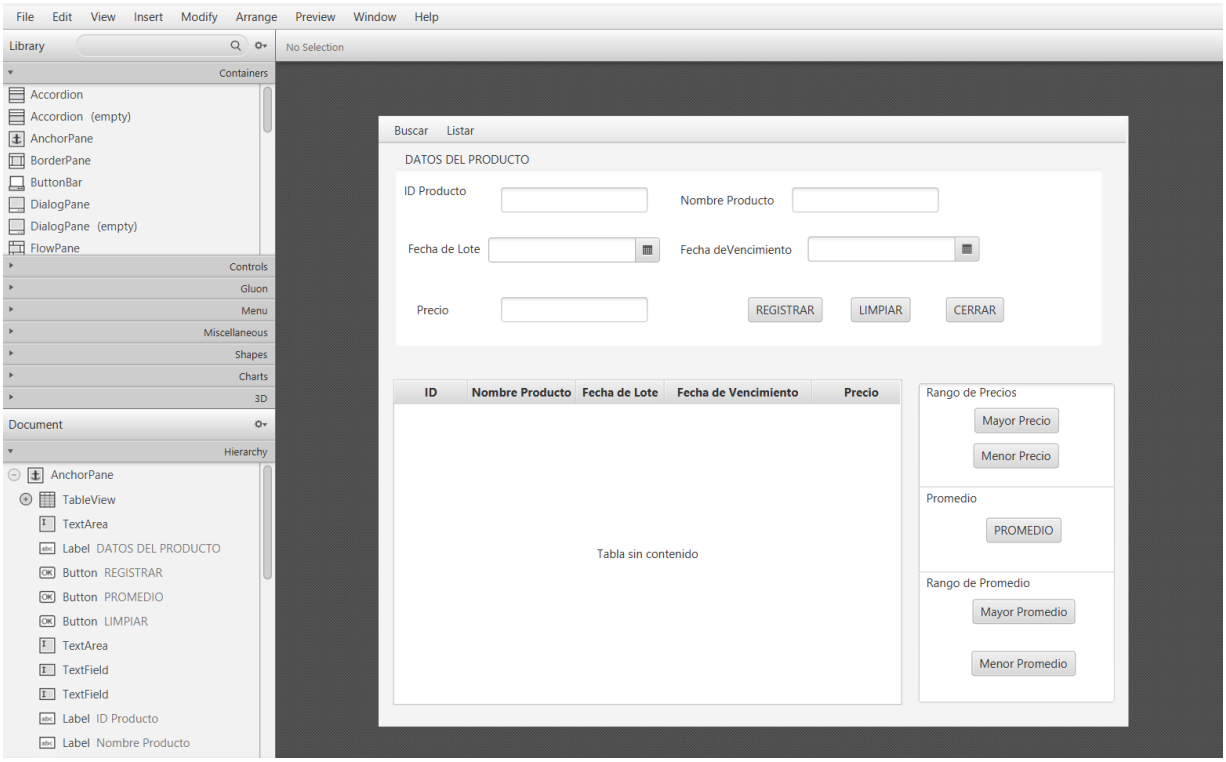
6. **Generar el Controlador** (Opcional): Si se desea agregar interacciónes a los elementos (por ejemplo, para manejar eventos de botones), puedes generar un controlador. Para hacerlo, ir a "Ver" > "Controladores" y especificar el controlador que se desea usar. Luego, Scene Builder generará un archivo de controlador FXML relacionado.

7. **Cierra Scene Builder**: Una vez que hayas diseñada la ventana, cierra Scene Builder.

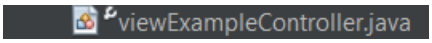
8. **Usa Scene Builder en la Aplicación**: En la aplicación, se usa un FXMLLoader para cargar el archivo FXML creado con Scene Builder. Luego, muestra la interfaz de usuario en una ventana, como se describe en el paso a paso anterior.

Paso a paso:

Ejemplo a continuación.



2.4. Package: Controller

2.4.1. Clase: viewExampleController.java 

La clase se encarga de gestionar la lógica y la interacción en una aplicación de gestión de productos.

Atributos:

- **private Pila Codigo**: Un objeto de la clase Pila, que se utiliza para almacenar y gestionar productos. La Pila es una estructura de datos que puede apilar y desapilar objetos de tipo tienda. Se usa para realizar operaciones como búsqueda y cálculos de precios.
- **private ObservableList<tienda> Producto:** Esta es una lista observable que almacena objetos de tipo tienda. Se utiliza para mostrar los datos de los productos en una tabla en la interfaz de usuario. La lista observable permite una actualización dinámica de la interfaz cuando se agregan o eliminan productos.
- **Varios atributos @FXML** relacionados con componentes de la interfaz de usuario, como botones, menús, tablas y campos de entrada.

Algunos atributos se anotan con @FXML para que puedan ser inyectados por JavaFX y se conecten con los elementos correspondientes en el archivo FXML de la interfaz de usuario.

- **private MenuBar menu:** El menú de la aplicación.
- **private Menu buscar:** Un menú para realizar operaciones de búsqueda.
- **private MenuItem idp, npb, flb, fvb, pub**: Elementos del menú "buscar" para buscar productos por diferentes criterios.
- **private Menu listar:** Un menú para realizar operaciones de listado de productos.
- **private MenuItem fechalote, fechavencimiento:** Elementos del menú "listar" para listar productos por fecha.
- **private Button cerrarboton**: Un botón que cierra la aplicación.
- **private Button botonLim:** Un botón para limpiar la tabla de productos.
- **@FXML private Button BotonR:** Un botón que se utiliza para registrar nuevos productos.
- **@FXML private Button botonp:** Un botón para buscar productos por precio.
- **@FXML private Button mayboton:** Un botón para buscar productos con el precio más alto.

Atributos:

El resto de los atributos descritos son de anotación **@FXML**.

- **@FXML private Button menboton:** Un botón para buscar productos con el precio más bajo.
- **@FXML private Button meboton:** Un botón para buscar productos por mes de lote.
- **@FXML private Button maboton:** Un botón para buscar productos por mes de vencimiento.
- **@FXML private Label texto**: Un componente de etiqueta utilizado para mostrar información en la interfaz de usuario.
- **@FXML private DatePicker FechaVencimientoDate:** Un selector de fecha para buscar productos por fecha de vencimiento.
- **@FXML private DatePicker FechaLoteDate:** Un selector de fecha para buscar productos por fecha de lote.
- **@FXML private TableView<tienda> tablavis:** Una tabla en la interfaz de usuario utilizada para mostrar productos y sus detalles.
- **@FXML private TableColumn<tienda, String> IdProducto:** Columna que muestra el ID del producto en la tabla.
- **@FXML private TableColumn<tienda, String> NombreProducto:** Columna que muestra el nombre del producto en la tabla.
- **@FXML private TableColumn<tienda, LocalDate> FechaLoteD:** Columna que muestra la fecha de lote del producto en la tabla.
- **@FXML private TableColumn<tienda, LocalDate> FechadeVencimiento**: Columna que muestra la fecha de vencimiento del producto en la tabla.
- **@FXML private TableColumn<tienda, Double> PrecioUnitario:** Columna que muestra el precio unitario del producto en la tabla.
- **@FXML private TextField IdProductoTextField:** Un campo de entrada de texto para ingresar el ID de un nuevo producto.
- **@FXML private TextField NombreProductoTextField**: Un campo de entrada de texto para ingresar el nombre de un nuevo producto.
- **@FXML private TextField PrecioUnitarioTextField:** Un campo de entrada de texto para ingresar el precio unitario de un nuevo producto.

Métodos:

- **public void limpiarTabla(ActionEvent event):** Este método se invoca cuando se presiona el botón "Limpiar". Su función es limpiar la tabla de productos.
- **public void cerrar(ActionEvent event):** Este método se invoca cuando se presiona el botón "Cerrar". Su función es cerrar la aplicación.
- **public void Action(ActionEvent event):** Este método se utiliza para buscar un producto por su ID.
- **public void Action1(ActionEvent event):** Este método se utiliza para buscar un producto por su nombre.
- **public void Action2(ActionEvent event):** Este método se utiliza para buscar un producto por su fecha de lote.
- **public void Action3(ActionEvent event):** Este método se utiliza para buscar un producto por su fecha de vencimiento.
- **public void Action4(ActionEvent event):** Este método se utiliza para buscar un producto por su precio.
- **public void mayorp(ActionEvent event):** Este método se utiliza para buscar el producto con el precio más alto.
- **public void menorpr(ActionEvent event):** Este método se utiliza para buscar el producto con el precio más bajo.
- **public void ProductosMesLote(ActionEvent event):** Este método se utiliza para buscar productos por mes de lote.
- **public void ProductosMesVencimiento(ActionEvent event):** Este método se utiliza para buscar productos por mes de vencimiento.
- **public void promedio(ActionEvent event):** Este método se utiliza para calcular y mostrar el precio promedio de los productos.
- **public void Mepromedio(ActionEvent event**): Este método se utiliza para buscar productos con precios por debajo del promedio.
- **public void mayorpro(ActionEvent event):** Este método se utiliza para buscar productos con precios por encima del promedio.
- **public void eventKey(KeyEvent event**): Este método se encarga de controlar la entrada de datos en los campos de texto, permitiendo solo caracteres numéricos o alfabéticos según corresponda.
- **public void Registrar(ActionEvent event):** Este método se utiliza para registrar un nuevo producto en la lista.

- **public void initialize(URL url, ResourceBundle rb):** Este método se ejecuta al inicializar el controlador y se encarga de configurar la tabla de productos y sus columnas.

Paso a Paso

1. **Diseño y Configuración de la Ventana en Scene Builder y Configuración de Eventos**

    - Abrir Scene Builder y crea el diseño de la ventana.
    - Ajusta la apariencia, disposición y colores de la ventana y los controles.
    - Configurar eventos y acciones en la pestaña "Código" de Scene Builder.
    - Guardar el diseño en un archivo .fxml.

2. **Creación de la Ventana Principal en JavaFX**

    - Crear una clase JavaFX que extienda javafx.application.Application.
    - Implementar el método start para cargar la ventana principal desde el archivo .fxml creado en el **Paso 1**.

3. **Carga del Archivo .fxml en la Ventana Principal**

    - En el método start, cargar el archivo .fxml usando FXMLLoader.
    - Asignar la escena y muestra la ventana principal.

4. **Eventos y Acciones en la Ventana Principal**

    - En la clase principal de la aplicación, se definen los métodos que manejan los eventos y acciones configurados en Scene Builder.
    - Vincula estos métodos a los controles en el archivo .fxml.

5. **Ejecución de la Aplicación**

    - En el método main de la aplicación, lanza la aplicación JavaFX usando launch.

6. Compilación y Ejecución de la Aplicación

Compilar las clases Java con javac.
Ejecuta la aplicación JavaFX.

3. Conclusiones

En este proyecto de Java, se han presentado varios componentes y códigos que abarcan distintas áreas de desarrollo de software. Desde la creación de ventanas de interfaz de usuario en JavaFX hasta la implementación de una clase para el manejo de datos, estos elementos pueden ser útiles en diferentes tipos de aplicaciones.

El uso de JavaFX y Scene Builder para crear interfaces gráficas proporciona una forma eficiente de diseñar ventanas y controles interactivos. El archivo .fxml permite definir la estructura de la interfaz y conectarla a la lógica de la aplicación. Además, la capacidad de gestionar eventos y acciones a través de Scene Builder simplifica el desarrollo de aplicaciones interactivas.

La clase **Pila** permitió organizar y gestionar datos en una aplicación. En este caso, se utiliza una estructura de pila para mantener un registro de productos con sus atributos. Esta clase también proporciona métodos para buscar, calcular promedios y listar productos según criterios específicos.

El proyecto y componentes reflejan la versatilidad de Java en términos de desarrollo de aplicaciones. Pueden formar parte de un sistema más amplio o adaptarse para satisfacer aun más necesidades específicas. Al comprender cómo se relacionan estos componentes y cómo funcionan en conjunto, se tienen una base sólida para crear otras aplicaciones personalizadas y permiten mayor efectividad.