**Resumen de mi proyecto y Algoritmo**

**28/07/2024**

**Introducción**

El proyecto es un Sistema de Gestión de Inventarios desarrollado en Java utilizando Apache NetBeans. Este sistema permite a las empresas o negocios llevar un registro detallado de sus productos en stock, facilitando el control de las existencias, la realización de pedidos de compra y la toma de decisiones estratégicas relacionadas con el inventario.

**Desarrollo del Proyecto**

**1. Configuración Inicial del Proyecto**

**Creación del Proyecto en NetBeans:**

Iniciaste NetBeans y creaste un nuevo proyecto Java utilizando "Java with Ant".

Configuraste el proyecto con las estructuras básicas necesarias.

**Configuración del Repositorio en GitHub:**

Creaste un repositorio público en GitHub para versionar el código del proyecto.

Configuraste el repositorio en NetBeans utilizando las opciones de Versioning y Git.

**2. Implementación del Sistema de Inicio de Sesión**

**ALGORITMO DE TODO EL SISTEMA DE GESTION DE INVENTARIOS**

**Clase** LoginScreen:

Creaste una clase LoginScreen que define una interfaz gráfica para el inicio de sesión.

Utilizaste JFrame para crear la ventana principal y JPanel para organizar los componentes.

Añadiste campos de texto para el nombre de usuario y la contraseña, estilizados con colores específicos y bordes.

Implementaste un botón de inicio de sesión que, al ser presionado, verifica las credenciales (usuario: "admin", contraseña: "admin").

4/08/2024

**3. Implementación de la Pantalla de Sistema de Inventarios**

1. **Clase SistemaInventariosScreen:**

Creaste una clase SistemaInventariosScreen que define una interfaz gráfica con botones para acceder a diferentes funcionalidades del sistema.

Añadiste botones para "Gestión de Productos", "Control de Existencias", "Pedidos de Compra" y "Informes y Estadísticas".

Configuraste acciones para cada botón, que actualmente muestran un mensaje de confirmación.

**Configuración del Repositorio en GitHub**

**Creación del Repositorio:**

Creaste un repositorio público en GitHub con un nombre adecuado para el proyecto.

Configuraste el repositorio local en tu proyecto de NetBeans.

**Commits y Pushes:**

Realizaste el primer commit del proyecto con los archivos iniciales.

Subiste los cambios al repositorio remoto en GitHub utilizando las herramientas de Git integradas en NetBeans.

**Funcionamiento del Sistema**

**Inicio de Sesión:**

El usuario abre la aplicación y se le presenta una pantalla de inicio de sesión.

El usuario introduce su nombre de usuario y contraseña, y si las credenciales son correctas, se le dirige a la pantalla principal del sistema.

**Pantalla Principal del Sistema de Inventarios:**

La pantalla principal muestra botones para cada una de las funcionalidades del sistema: Gestión de Productos, Control de Existencias, Pedidos de Compra e Informes y Estadísticas.

Al hacer clic en cualquiera de estos botones, se muestra un mensaje de confirmación (esta parte se puede expandir para abrir ventanas específicas para cada funcionalidad).

5/08/2024

Gestión de Productos

Definición de Categorías:

 **JFrame Configuración**: Se configura el JFrame principal para la gestión de productos.

 **Panel de Categorías**: Un JPanel para mostrar y gestionar las categorías de productos.

 **Lista de Categorías**: Se utiliza un JList con un DefaultListModel para manejar la lista de categorías.

 **Botones de Categorías**: Botones para agregar, eliminar y modificar categorías.

 **Panel Principal**: Este es el panel donde se agregarán más componentes para las demás funcionalidades (alta, baja, modificación, características y especificaciones).

 **Acciones de los Botones**: Se añaden ActionListener a los botones para manejar las acciones de agregar, eliminar y modificar categorías.

 **Validación de Nombre Vacío**:

* Se verifica si el campo nombre está vacío y se muestra un mensaje de error si es así.

 **Validación de Nombre Existente**:

* Se recorre el modelo de la lista de categorías (modeloCategorias) y se compara cada nombre existente con el nombre ingresado, ignorando mayúsculas y minúsculas.
* Si se encuentra una coincidencia, se muestra un mensaje de error y se detiene el proceso de agregar la categoría.

14/08/2024

 **Botón "Guardar"**:

* Se añadió un botón btnGuardarCategoria a la interfaz, y se agregó al panelBotonesCategorias.

 **Acción del Botón "Guardar"**:

* Al hacer clic en el botón "Guardar", se invoca el método guardarCategoriaEnArchivo, que recorre todas las categorías en modeloCategorias y las guarda en un archivo de texto categorias.txt.

 **Método guardarCategoriaEnArchivo()**:

* Este método abre el archivo categorias.txt en modo de escritura y guarda cada categoría en una nueva línea.
* Se muestra un mensaje de confirmación si la operación es exitosa o un mensaje de error si algo falla

25/08/2024

 **Interfaz de Modificación**: Utilizamos un JOptionPane similar al de agregar categorías para modificar el nombre y la descripción de una categoría existente.

 **Validaciones**: Aseguramos que el nuevo nombre no esté vacío y que no sea un duplicado de otra categoría existente.

 **Guardado en Archivo**: Se actualizan los cambios en el archivo categorias.txt tanto al agregar, modificar, como al eliminar categorías.

 **Carga de Categorías**: Al iniciar la aplicación, se cargan todas las categorías desde el archivo categorias.txt para que estén disponibles en la interfaz.

 **Validación del Nombre Vacío**: Se agrega una verificación para asegurarse de que el nuevo nombre de la categoría no esté vacío.

 **Verificación de Duplicados**: Antes de modificar, el sistema verifica si ya existe una categoría con el mismo nombre (ignorando el nombre actual de la categoría seleccionada).

 **Actualización del Archivo**: Una vez modificada la categoría en el modelo, se llama a guardarCategoriasEnArchivo() para actualizar el archivo de texto

26/08/2024

 **modificarCategoria()**: Este método es el encargado de modificar el nombre de una categoría seleccionada por el usuario. Tras modificar la categoría, se llama al método guardarCategoriasEnArchivo() para asegurarse de que los cambios se guarden en el archivo.

 **guardarCategoriasEnArchivo()**: Este método sobrescribe el archivo categorias.txt con todas las categorías actuales del modelo. Utiliza un BufferedWriter para escribir cada categoría en una nueva línea.

 **categoriaYaExiste()**: Este método auxiliar verifica si una categoría con el mismo nombre ya existe en el modelo, ayudando a evitar duplicados.

 **Eliminación de una categoría existente**: Selecciona una categoría y haz clic en el botón "Eliminar". Confirma la acción y verifica si la categoría desaparece tanto de la interfaz gráfica como del archivo categorias.txt.

 **Intentar eliminar sin seleccionar una categoría**: Intenta hacer clic en el botón "Eliminar" sin seleccionar una categoría para asegurarte de que el programa muestra un mensaje adecuado.

 **Validación de eliminación exitosa**: Después de eliminar una categoría, abre el archivo categorias.txt para asegurarte de que la categoría eliminada ya no está presente.

 **Verificación de Selección**: Primero, el método comprueba si el usuario ha seleccionado una categoría en la lista (listaCategorias). Si no se ha seleccionado nada, se muestra un mensaje de advertencia.

 **Cuadro de Diálogo de Confirmación**: Si hay una categoría seleccionada, el método JOptionPane.showConfirmDialog se utiliza para mostrar un cuadro de diálogo de confirmación al usuario. Este diálogo tiene un mensaje, un título ("Confirmar eliminación") y dos opciones: YES y NO.

 **Acción Basada en la Respuesta del Usuario**:

* **Si el usuario selecciona "Sí"** (JOptionPane.YES\_OPTION), la categoría seleccionada se elimina de la lista (modeloCategorias) y el archivo se actualiza llamando al método guardarCategoriasEnArchivo().
* **Si el usuario selecciona "No"** (JOptionPane.NO\_OPTION), no se realiza ninguna acción adicional y el cuadro de diálogo se cierra.

 **Verificación de Selección**: Primero, el método comprueba si el usuario ha seleccionado una categoría en la lista (listaCategorias). Si no se ha seleccionado nada, se muestra un mensaje de advertencia.

 **Cuadro de Diálogo de Confirmación**: Si hay una categoría seleccionada, el método JOptionPane.showConfirmDialog se utiliza para mostrar un cuadro de diálogo de confirmación al usuario. Este diálogo tiene un mensaje, un título ("Confirmar eliminación") y dos opciones: YES y NO.

 **Acción Basada en la Respuesta del Usuario**:

* **Si el usuario selecciona "Sí"** (JOptionPane.YES\_OPTION), la categoría seleccionada se elimina de la lista (modeloCategorias) y el archivo se actualiza llamando al método guardarCategoriasEnArchivo().
* **Si el usuario selecciona "No"** (JOptionPane.NO\_OPTION), no se realiza ninguna acción adicional y el cuadro de diálogo se cierra.

 **Método eliminarCategoriaDelArchivo:**

* **Lectura del Archivo:** Se utiliza un BufferedReader para leer las categorías actuales desde el archivo original.
* **Escritura Condicional:** Se usa un BufferedWriter para escribir todas las líneas de nuevo al archivo temporal excepto la línea que coincide con la categoría a eliminar.
* **Reemplazo del Archivo Original:** El archivo original se elimina y se reemplaza con el archivo temporal actualizado, lo que efectivamente elimina la categoría del archivo de texto de acceso secuencial.

 **Validación y Seguridad:**

* **Confirmación:** Antes de eliminar, se solicita al usuario una confirmación.
* **Archivos Temporales:** El uso de un archivo temporal garantiza que si algo sale mal durante el proceso de eliminación, el archivo original no se corrompa.

11/09/2024

 **eliminarAsociacionesDeProductos**:

* Recorre la lista de productos y asigna la categoría "Sin categoría" a los productos que pertenecían a la categoría eliminada.
* Si prefieres eliminar los productos en lugar de asignar una categoría, puedes usar iterator.remove().

 **guardarCategoriaEnArchivo y guardarProductosEnArchivo**:

* Puedes completar estas funciones para escribir las categorías y productos en sus respectivos archivos.

 **cargarProductosSimulados**:

* Simula algunos productos para probar la eliminación de categorías. En tu sistema real, los productos deberían cargarse desde un archivo o base de datos.

Definición de Características:

*  **Botón en la ventana principal**: En la ventana principal de gestión de productos, hemos añadido un botón llamado "Características de Productos".
*  **Abrir nueva ventana**: Al hacer clic en el botón, se abre una nueva ventana donde se gestionarán las características de los productos.
*  **Nueva ventana de características**: La nueva ventana está creada con su propio JFrame y puede ser extendida para incluir funcionalidades como agregar, modificar y eliminar características.

 **Cerrar la ventana**:

* Un botón "Cerrar" permite al usuario salir de la ventana cuando lo desee.
*  **JList**: Se usa para mostrar la lista de características.
*  **DefaultListModel**: Se usa para manejar los elementos de la lista. Aquí se añadirán las características que el usuario defina.
*  **Botones de acción**: Los botones permiten agregar, eliminar o modificar características

 **JOptionPane con múltiples entradas**: Utilizamos un JOptionPane que presenta dos campos de texto para que el usuario ingrese tanto el nombre como la descripción.

 **Validación**: Verifica que el campo de nombre no esté vacío. Si el usuario no ingresa una descripción, solo se agrega el nombre de la característica.

 **Agregar al modelo**: Si ambos campos están completos, se añade la característica en el formato Nombre - Descripción o solo Nombre si la descripción está vacía.

13/09/2024

 **Validación del nombre**: El sistema verifica que el campo de nombre no esté vacío.

 **Verificación de duplicados**: Se busca en la lista de características existentes para asegurarse de que no se repita el nombre.

 **Guardar en archivo**: Si la característica pasa las validaciones, se guarda en un archivo de texto.

14/09/2024

 **BufferedWriter**: Usamos esta clase para escribir caracteres en un archivo de forma eficiente.

 **FileWriter**: Se usa con append = true para que el archivo no se sobrescriba, sino que se agreguen nuevas líneas al final del archivo.

 **Estructura del archivo**: Cada característica se guarda en una línea nueva, donde el nombre y la descripción están separados por el delimitador.

 **Lectura del archivo original:** Se abre el archivo caracteristicas.txt y se lee línea por línea. Cada línea se separa en partes usando el delimitador | para extraer el nombre de la característica.

 **Verificación de la coincidencia:** Si el nombre en la línea coincide con el nombreOriginal, entonces se escribe la nueva línea con el nuevoNombre y la nuevaDescripcion en un archivo temporal.

 **Escritura en archivo temporal:** Si no coincide, se escribe la línea sin modificar en un archivo temporal.

 **Reemplazo del archivo original:** Una vez que todo se ha leído y modificado, se borra el archivo original y se renombra el archivo temporal para que lo reemplace.

 **Validar que el campo no esté vacío:**

* Antes de modificar la característica, se verifica que el campo txtNombreCaracteristica no esté vacío con el método isEmpty(). Si está vacío, se muestra una advertencia.

 **Validar que el nuevo nombre no exista:**

* Se llama al método validarNuevoNombreUnico() para verificar que el nuevo nombre no exista ya en el archivo. Si el nombre ya está en uso, se muestra una advertencia y se cancela el proceso de modificación.

 **Modificar si todo es válido:**

* Si ambas validaciones pasan, se llama a modificarCaracteristicaEnArchivo() para hacer la modificación y luego se recarga la lista con las características actualizadas.

 **Lectura y escritura en archivos:**

* Se abre el archivo caracteristicas.txt con un BufferedReader para leer cada línea.
* Simultáneamente, se crea un archivo temporal caracteristicas\_temp.txt con un BufferedWriter donde se van a escribir las características modificadas.

 **Buscar y modificar la característica:**

* Mientras se lee el archivo original línea por línea, se verifica si el nombre de la característica coincide con el nombre que queremos modificar (nombreOriginal).
* Si coincide, se escribe la nueva versión de la característica con el nuevoNombre y la nuevaDescripcion. Si no coincide, se escribe la línea tal cual al archivo temporal.

 **Reemplazar el archivo original:**

* Al finalizar la lectura del archivo, se elimina el archivo original y se renombra el archivo temporal como el archivo original. Esto asegura que los cambios se guarden de manera efectiva.

 **Manejo de errores:**

* Se utiliza un bloque try-catch para manejar cualquier excepción que pueda ocurrir al manipular los archivos, y se muestra un mensaje de error si algo sale mal.

22/09/2024

*  El usuario selecciona la característica a eliminar.
*  El sistema solicita confirmación para eliminar la característica.
*  Si se confirma, el sistema elimina la característica del archivo.
*  Se actualiza la lista de características visibles.
* **Selección de la característica**: Se obtiene la característica seleccionada de la lista.
* **Confirmación**: Se utiliza JOptionPane.showConfirmDialog con los parámetros:
  + **this**: Se refiere al componente principal (puede ser la ventana principal).
  + **Mensaje de confirmación**: Se pregunta si realmente desea eliminar la característica seleccionada.
  + **Título del cuadro de diálogo**: "Confirmar Eliminación".
  + **Opciones**: JOptionPane.YES\_NO\_OPTION, mostrando dos botones: "Sí" y "No".
  + **Mensaje de advertencia**: Se puede añadir un ícono de advertencia utilizando JOptionPane.WARNING\_MESSAGE para resaltar la importancia de la acción.
* **Respuesta del usuario**:
  + Si el usuario selecciona "Sí", se procede con la eliminación de la característica.
  + Si el usuario selecciona "No", la eliminación se cancela y se muestra un mensaje informativo.
* **Eliminación**: Se llama al método eliminarCaracteristicaDelArchivo para eliminar la característica del archivo de texto.

**Confirmación en acción:**

Cada vez que el usuario intente eliminar una característica, verá una ventana de confirmación para evitar eliminaciones accidentales.

Este es el comportamiento estándar para confirmar la eliminación de datos importantes y asegurar que el usuario tenga una segunda oportunidad para cancelar la acción.

1. **Archivo temporal**: Se utiliza un archivo temporal para escribir todas las características excepto la que se va a eliminar. Este archivo reemplazará al archivo original una vez que se elimine la característica.
2. **Lectura del archivo original**:
   * Se lee el archivo línea por línea usando un BufferedReader.
   * Cada línea representa una característica en el archivo de texto.
   * Si el nombre de la característica no coincide con la que se desea eliminar, se escribe en el archivo temporal.
3. **Eliminar el archivo original**:
   * Una vez que se han escrito todas las líneas excepto la característica a eliminar en el archivo temporal, se borra el archivo original.
4. **Renombrar el archivo temporal**:
   * Después de borrar el archivo original, el archivo temporal se renombra para que tome el lugar del original.

**Integración con la confirmación de eliminación:**

Este método debe ser llamado en el flujo del cuadro de confirmación que ya has implementado anteriormente. Después de la confirmación del usuario, si elige "Sí", se llama al método eliminarCaracteristicaDelArchivo() para proceder con la eliminación de la característica.

 **Archivo temporal**: Al igual que en la eliminación de la característica, usamos un archivo temporal para almacenar las modificaciones, que luego reemplazará al archivo original de productos.

 **Lectura y procesamiento**:

* Se lee el archivo de productos línea por línea.
* Se separa cada línea en sus componentes (ID del producto, nombre, lista de características).
* Se procesan las características para eliminar la que se desea.

 **Actualizar la lista de características**:

* Se reconstruye la lista de características sin la característica eliminada.
* Si la lista estaba vacía o solo contenía la característica eliminada, quedará vacía.

 **Escribir la nueva línea en el archivo temporal**: Se escribe la línea con la lista de características actualizada en el archivo temporal.

23/09/2024

**Especificaciones de Productos**

 **Archivo de especificaciones (especificaciones.txt)**: Cada especificación debe estar guardada en una línea diferente. El sistema leerá el archivo y cargará las especificaciones en el JList.

 **cargarEspecificacionesDesdeArchivo**: Este método abre el archivo de especificaciones y carga cada línea en el DefaultListModel, que se muestra en la interfaz gráfica.

 **Campos de entrada**: Se añaden campos para el nombre, descripción y tipo de dato.

 **Validaciones**: El nombre no debe estar vacío y debe ser único.

 **Guardado**: Se guarda la especificación en un archivo de texto con un formato sencillo.

 **Recarga**: Al agregar una nueva especificación, la lista de especificaciones se actualiza automáticamente.

 El usuario intenta agregar una nueva especificación.

 El sistema primero valida que el nombre no esté vacío.

 Luego, se verifica que el nombre no esté duplicado en el archivo de especificaciones.

 Si ambas validaciones pasan, el sistema guarda la nueva especificación en el archivo de texto y actualiza la lista visible en la interfaz.

 Seleccionas una especificación de la lista.

 Se cargan los datos de la especificación en los campos.

 El usuario edita los datos y hace clic en "Modificar".

 Se valida el nuevo nombre y los datos.

 Se guarda la especificación modificada en el archivo reemplazando la línea correspondiente.

 **Leer el archivo original**: Se utiliza un BufferedReader para leer línea por línea el archivo especificaciones.txt.

 **Buscar la especificación original**: Si el nombre de la especificación coincide con el nombre que se quiere modificar, se escribe la versión modificada en un archivo temporal.

 **Guardar la especificación modificada**: Si se encuentra la especificación, se escribe en el archivo temporal con los nuevos datos.

 **Reemplazar el archivo original**: Una vez que todas las líneas se han procesado, el archivo temporal reemplaza al archivo original.

 **Leer y procesar el archivo**: El código lee el archivo productos.txt línea por línea. Cada línea tiene el formato Producto|Especificaciones.

 **Eliminar la especificación**: Si una línea contiene la especificación que deseas eliminar, se excluye esa especificación de la lista de especificaciones.

 **Escribir el nuevo archivo**: Si la lista de especificaciones del producto no queda vacía, se vuelve a escribir la línea. Si queda vacía, se escribe el producto sin especificaciones.

 **Reemplazar el archivo original**: Al final, el archivo original se reemplaza por el nuevo archivo modificado.

 **Relación entre productos y especificaciones**: Debes manejar una relación entre productos y especificaciones, almacenada en archivos o base de datos.

 **Eliminar asociaciones**: Al eliminar una especificación, también debes eliminarla de los productos que la utilizaban.

 **Confirmación adicional**: Puedes agregar una advertencia que indique que las asociaciones con productos también serán eliminadas.

25/09/2024

Asignación de Categorías, Características y Especificaciones a

Productos:

 **JComboBox**: Creamos un JComboBox (comboCategorias) que mostrará las categorías para que el usuario las seleccione al crear un nuevo producto.

 **Carga de Categorías**: El método cargarCategoriasDesdeArchivo carga las categorías desde un archivo de texto (por ejemplo, categorias.txt) y las añade al modelo del JComboBox.

 **Guardado del Producto**: Cuando se selecciona una categoría y se ingresa un nombre de producto, se guarda esta información en un archivo (productos.txt).

 **Características y Especificaciones**: Se añaden dos JList para que el usuario pueda seleccionar múltiples características y especificaciones.

 **Validación**: Se valida que el usuario haya seleccionado al menos una característica y una especificación antes de guardar el producto.

 **Guardado**: Se guarda el producto junto con la categoría seleccionada y las características y especificaciones en un archivo de texto.

29/09/2024

 **Ingreso de valores:** Cuando el usuario selecciona características y especificaciones, el botón "Ingresar Valores" abrirá un JOptionPane para que el usuario ingrese los valores personalizados.

 **Guardado:** Se guardan las características, especificaciones, y los valores correspondientes en un archivo de texto (productos.txt).

 **Estructura de datos:** Usamos LinkedHashMap para almacenar las características y especificaciones junto con los valores ingresados.

 **Carga de productos:** El método cargarProductosDesdeArchivo() lee el archivo productos.txt y carga los productos en una lista que el usuario puede seleccionar.

 **Modificación de producto:** Al seleccionar un producto, el método cargarProductoSeleccionado() llena los campos con los datos actuales del producto, permitiendo la modificación de la categoría, características y especificaciones.

 **Guardar los cambios:** Una vez modificados, guardarProductoModificado() reemplaza el producto seleccionado con los nuevos datos, y actualiza el archivo productos.txt

 **Carga de características:**

* Verificamos si el archivo caracteristicas.txt existe. Si no, lo creamos.
* Si ya existe o se ha creado, lo leemos línea por línea y añadimos las características al modelo DefaultListModel.

 **Carga de especificaciones:**

* Similar a las características, verificamos si el archivo especificaciones.txt existe o lo creamos si es necesario.
* Luego, leemos el archivo y añadimos las especificaciones al modelo correspondiente.

 **Asignación de los modelos:** Después de cargar los datos, asignamos los modelos a los componentes visuales JList correspondientes (listaCaracteristicas y listaEspecificaciones).

 **Carga de tipos de datos:** El método cargarTiposDatosDesdeArchivo() lee un archivo de texto que contiene los tipos de datos esperados para cada característica o especificación.

 **Validación de tipos de datos:** El método validarTiposDeDatos() recorre los valores ingresados por el usuario y verifica que cumplan con el tipo de dato especificado (por ejemplo, si se espera un int, valida que el valor sea un número entero).

 **Guardado de producto:** Antes de guardar, el sistema valida que todos los valores ingresados sean del tipo correcto. Si no lo son, muestra un mensaje de error y detiene el guardado.

 **JTextArea:** Se usa un JTextArea para permitir que el usuario ingrese una descripción larga. Puedes ajustar el tamaño (filas y columnas) según lo que necesites.

 **JScrollPane:** Al envolver el JTextArea en un JScrollPane, si la descripción es más larga que el área visible, aparecerá una barra de desplazamiento.

 **Validación:** El sistema puede validar que la descripción no esté vacía antes de guardar el producto.

 **Guardar la descripción:** Al guardar el producto, la descripción se escribe junto con el resto de los datos en el archivo productos.txt.

 El usuario selecciona las características del producto de una lista.

 El sistema solicita al usuario que ingrese un valor para cada característica seleccionada.

 Las características y sus valores se guardan correctamente en el archivo productos.txt.

 **Agregar un campo para ingresar el precio**:

* Puedes utilizar un JTextField para que el usuario ingrese el precio.
* Asegúrate de que el precio se valide como un número.

 **Validar el tipo de dato**:

* El precio debe ser un valor numérico. Puedes usar una verificación para asegurarte de que el valor ingresado es un número válido.

 **Guardar el precio en el archivo de texto**:

* Una vez validado, guarda el precio junto con los demás datos del producto.
*  **Nombre del producto**: Se utiliza isEmpty() para verificar si el campo está vacío.
*  **Categoría seleccionada**: Se asegura de que una categoría ha sido seleccionada de la lista desplegable.
*  **Descripción del producto**: Se valida que el campo de texto no esté vacío.
*  **Características y Especificaciones**: Las listas deben tener al menos una característica y una especificación seleccionada.
*  **Precio de venta**: Se convierte el valor de cadena a double y se valida que sea mayor que 0.
*  **Cantidad de stock**: Se convierte el valor de cadena a int y se valida que sea mayor o igual a 0.
*  **Estructura de validación**: Se verifica uno por uno cada campo del formulario, mostrando un JOptionPane con un mensaje de error si alguna validación falla, deteniendo la ejecución del método con un return.
*  **Guardado del producto**: Si todas las validaciones son correctas, los datos se guardan en el archivo productos.txt de manera correcta.
*  **UUID**: Utiliza UUID.randomUUID().toString() para generar un identificador único alfanumérico para cada producto. Este identificador se añade al principio de cada línea en el archivo de productos.
*  **Archivo productos.txt**: Ahora cada producto almacenado en el archivo tendrá un identificador único asociado en la primera columna.
*  Los UUIDs son prácticamente únicos en todos los contextos y aseguran que no haya colisiones de identificadores.
*  Se puede utilizar en múltiples sistemas sin necesidad de coordinar un contador centralizado.

 **new FileWriter("productos.txt", true)**: El argumento true asegura que el archivo se abra en modo **append** (agregar al final del archivo), lo que garantiza que cada vez que se guarde un producto, se agregue una nueva línea sin sobrescribir la información anterior.

 **writer.newLine()**: Asegura que cada producto se escriba en una nueva línea en el archivo de texto.

*  **Formato secuencial**: Cada línea representa un producto con su información separada por delimitadores (| en este caso)
*  **JOptionPane.showMessageDialog** muestra una ventana emergente con un mensaje al usuario.
*  En este caso, el mensaje indica que el **producto se ha guardado correctamente** en el archivo, cumpliendo con el requerimiento de informar al usuario sobre la creación exitosa del producto.

 Este código muestra una lista de productos usando un JComboBox. Los productos cargados desde el archivo productos.txt aparecerán en la lista.

*  Luego, el usuario podrá seleccionar uno para eliminar en el próximo paso (1.6.2, eliminar el producto seleccionado).

 **JOptionPane.showConfirmDialog**: Este cuadro de diálogo muestra un mensaje al usuario y le da la opción de confirmar con "Sí" o "No".

* El tercer argumento "Confirmar eliminación" es el título del cuadro de diálogo.
* El cuarto argumento JOptionPane.YES\_NO\_OPTION indica que el cuadro mostrará dos botones: "Sí" y "No".

 **confirmacion == JOptionPane.YES\_OPTION**: Si el usuario elige "Sí", se procede con la eliminación del producto; de lo contrario, no se realiza ninguna acción.

 **BufferedReader**: Se utiliza para leer el archivo original línea por línea.

 **BufferedWriter**: Se utiliza para escribir en un archivo temporal (productos\_temp.txt).

 **Comparación**: Si la línea actual contiene el nombre del producto a eliminar (productoAEliminar), se omite esa línea, lo que elimina el producto del archivo.

 **Reemplazo del archivo**: Después de procesar todas las líneas, el archivo original se elimina y el archivo temporal se renombra para reemplazar al original.

 **Confirmación de eliminación**: Si el producto fue encontrado y eliminado, se muestra un mensaje confirmando la acción; si no, se notifica al usuario que no se encontró el producto.

 El mensaje de éxito se muestra al usuario si el producto fue eliminado correctamente.

 Si el producto no fue encontrado en el archivo, se muestra un mensaje de error.

Modificación de Productos

 **Leer productos existentes desde el archivo**: Usa un método similar al que ya usas para leer las categorías, características, etc.

 **Mostrar los productos en una lista**: Puedes utilizar un JList o un JComboBox para mostrar los productos.

 **Permitir la selección**: Una vez seleccionado un producto, debes llenar los campos con los datos correspondientes para permitir su modificación.

 **Cargar los datos del producto**: El producto seleccionado se obtiene de la lista y se separa en sus componentes (nombre, categoría, características, especificaciones, precio, stock) usando el delimitador que se haya utilizado al guardar.

 **Mostrar los datos actuales**:

* El nombre del producto se muestra en txtNombreProducto.
* La categoría seleccionada se muestra en comboCategorias.
* Las características y especificaciones se separan por comas y se muestran en los JList correspondientes.
* El precio y la cantidad de stock se cargan en los campos de texto.
*  **Campo del código único:** Asegúrate de que el código único se muestre pero no sea editable.
*  **Modificar datos:** Permitir la modificación de los demás datos (nombre, categoría, características, especificaciones, precio y stock).
*  **Validar datos:** Valida que los datos ingresados por el usuario sean correctos.
*  **Guardar cambios:** Actualiza el archivo de productos sin cambiar el código único.
*  **Nombre vacío:** Si el nombre del producto está vacío, se muestra una advertencia.
*  **Categoría:** Si no se ha seleccionado ninguna categoría, también se muestra una advertencia.
*  **Características y especificaciones:** Se verifica que el usuario haya seleccionado al menos una característica y una especificación.
*  **Precio:** Se asegura de que el precio sea un valor numérico y positivo (o cero).
*  **Stock:** Se valida que el stock sea un número entero positivo (o cero).
*  **Guardar producto modificado:** Si todas las validaciones pasan, el producto se guarda en el archivo.

 **Validación:** Se validan los datos ingresados por el usuario antes de intentar guardar el producto modificado.

 **Lectura del archivo:** El sistema lee cada línea del archivo productos.txt. Cada línea representa un producto.

 **Modificación:** Cuando el sistema encuentra la línea que contiene el producto a modificar (basándose en el código único), reescribe esa línea con los nuevos datos del producto. Si no lo encuentra, muestra un mensaje de error.

 **Reescribir el archivo:** Todos los productos se escriben en un archivo temporal (productos\_temp.txt), que luego reemplaza el archivo original.

*  **Confirmación:** Si todo sale bien, se muestra un mensaje confirmando que el producto fue modificado.

Consideraciones adicionales:

 **Estructura de la categoría**: Cada categoría debe tener un nombre y posiblemente un padre. Esto permite crear relaciones jerárquicas.

 **Interfaz de usuario**: Para mostrar las jerarquías, podrías usar un JTree en Swing, ya que permite representar estructuras jerárquicas de manera visual.

 **Almacenamiento**: Las categorías deben almacenarse en un archivo de texto, manteniendo la relación jerárquica. Un formato como padre|hijo|nieto puede ser útil para almacenar esta información.

 El usuario podrá crear jerarquías seleccionando categorías padre.

 La jerarquía se mostrará en un JTree y se almacenará en un archivo de texto.

 Asegúrate de que al agregar categorías, se mantenga la relación padre-hijo, permitiendo crear estructuras como "ropa > ropa de mujer > blusas".

 **Modelo de Característica y Valores**:

* Una clase Caracteristica que contendrá el nombre de la característica y una lista de valores asociados (como una lista de colores para la característica "Color").

 **Interfaz gráfica**:

* Un JList o JComboBox para permitir la selección de múltiples valores para una característica específica.
* Opcionalmente, un botón para añadir nuevos valores a la lista de valores de una característica.

 **Guardado en archivo**:

* Guardar cada característica junto con sus valores en un archivo de texto, separando los valores con algún delimitador (por ejemplo, | o ,).

#### 1. Modelo de Caracteristica con múltiples valores

Primero, definimos una clase Caracteristica que contendrá el nombre de la característica y una lista de valores.

Para implementar la funcionalidad de **asociar múltiples valores a una misma característica**, por ejemplo, permitir que una característica como "Color" tenga valores como "Rojo", "Azul", "Verde", puedes seguir estos pasos:

**Diseño de la Solución:**

1. **Modelo de Característica y Valores**:
   * Una clase Caracteristica que contendrá el nombre de la característica y una lista de valores asociados (como una lista de colores para la característica "Color").
2. **Interfaz gráfica**:
   * Un JList o JComboBox para permitir la selección de múltiples valores para una característica específica.
   * Opcionalmente, un botón para añadir nuevos valores a la lista de valores de una característica.
3. **Guardado en archivo**:
   * Guardar cada característica junto con sus valores en un archivo de texto, separando los valores con algún delimitador (por ejemplo, | o ,).

**Implementación:**

**1. Modelo de Caracteristica con múltiples valores**

Primero, definimos una clase Caracteristica que contendrá el nombre de la característica y una lista de valores.

**2. Interfaz Gráfica**

En la interfaz gráfica, puedes usar un JList o JComboBox para permitir que el usuario seleccione varios valores o añadir nuevos valores.

#### 3. Guardar las características con múltiples valores en un archivo de texto

Para guardar las características con sus múltiples valores, puedes escribir las características en un archivo de texto, separando los valores con comas o algún delimitador.

#### 4. Cargar las características y valores desde el archivo

Cuando inicies el sistema, deberás cargar las características y sus valores desde el archivo para mostrarlas al usuario.

 El usuario puede agregar valores a una característica mediante la interfaz gráfica (JList o JComboBox).

 Los valores de cada característica se guardan en un archivo de texto, usando un formato delimitado (por ejemplo, "Color|Rojo,Azul,Verde").

 El sistema carga las características con sus respectivos valores cuando se inicia, para que el usuario pueda seguir trabajando con ellos.

 **Modelo de Especificación**:

* Crear una clase EspecificacionNumerica que representará una especificación numérica y su correspondiente unidad de medida.
* Esta clase contendrá el valor numérico y la unidad (por ejemplo, "kg" para peso o "cm" para altura).

 **Interfaz gráfica**:

* Añadir un JComboBox para permitir al usuario seleccionar la unidad de medida de una lista predefinida (como "kg", "cm", "m", "litros", etc.).
* Un campo de texto para ingresar el valor numérico correspondiente.

 **Guardado en archivo**:

* Guardar la especificación con su valor y unidad en un archivo de texto o base de datos, usando un formato como: "peso: 75 kg".

#### 1. Modelo de EspecificacionNumerica

Primero, creamos una clase EspecificacionNumerica que contendrá el valor numérico y la unidad de medida

**2. Interfaz gráfica**

Para permitir al usuario definir especificaciones numéricas con unidades de medida, puedes crear una interfaz gráfica que incluya:

* Un campo de texto para ingresar el valor numérico.
* Un JComboBox para seleccionar la unidad de medida.

#### 3. Guardar las especificaciones numéricas en un archivo

Para guardar las especificaciones numéricas, puedes escribir el nombre de la especificación, su valor y la unidad en un archivo de texto.

#### 4. Cargar las especificaciones numéricas desde el archivo

Para cargar las especificaciones numéricas y mostrarlas al usuario, puedes leer el archivo y parsear las líneas para extraer los valores y las unidades.

 **Interfaz gráfica**:

* El usuario ingresa un valor numérico en un campo de texto.
* Selecciona la unidad de medida de un JComboBox con las opciones predefinidas.
* Opcionalmente, el usuario puede agregar nuevas especificaciones con su unidad correspondiente.

 **Guardado de datos**:

* Las especificaciones numéricas se guardan en un archivo de texto con el formato: "peso: 75 kg", permitiendo recuperar esa información en futuras sesiones.

 **Carga de datos**:

* El sistema lee las especificaciones desde el archivo de texto al iniciar, cargando los valores y unidades de cada una.

Control de existencias

 **Nuevo método registrarEntrada**: La lógica de registrar una entrada para un producto se mueve a un método separado llamado registrarEntrada. Este método toma como parámetros el código del producto y la cantidad que se va a ingresar. De esta manera, la lógica de procesamiento está mejor encapsulada y es más fácil de leer.

 **Acción del botón "Registrar Entrada"**: Ahora, el botón simplemente llama al método registrarEntrada después de realizar las validaciones básicas de entrada de usuario, como verificar que el código no esté vacío y que la cantidad sea mayor a 0.

 **Manejo de errores**: El código maneja adecuadamente los errores de entrada de usuario (como ingresar una cantidad inválida) y problemas de lectura/escritura de archivos. Los mensajes de error se muestran mediante JOptionPane.

 **Verificación del código**: El sistema primero comprueba si el campo de código está vacío. Si lo está, muestra un mensaje pidiendo al usuario que lo ingrese.

 **Llamada a buscarProducto**: Si el código no está vacío, se llama a buscarProducto para ver si el producto existe en el archivo.

 **Mensaje si no está registrado**: Si buscarProducto retorna false (lo que significa que no encontró el producto), se muestra un mensaje al usuario diciendo que el producto no está registrado, y no se sigue con la lógica de registrar la entrada.

 **Búsqueda del producto**: Se verifica si el producto existe en el archivo usando el método buscarProducto. Si no existe, se muestra un mensaje de error.

 **Ingreso de cantidad**: Si el producto existe, se le solicita al usuario que ingrese la cantidad a través del campo txtCantidad.

 **Validación de la cantidad**: Se valida que la cantidad sea un número mayor a 0. Si no lo es, se muestra un mensaje de error y no se sigue adelante.

 **Registro exitoso**: Si todo está correcto, el sistema procede con la lógica para actualizar el stock y registrar la entrada.

 **Lectura del archivo de productos**: Usa un BufferedReader para leer cada línea del archivo productos.txt. Cada producto está separado por | en diferentes campos (por ejemplo, el código, nombre, stock, etc.).

 **Búsqueda del producto**: Mientras lees cada línea, divides los datos por | y comparas el código del producto con el código ingresado.

 **Actualización del stock**: Si el código coincide, extraes el stock actual, sumas la cantidad ingresada y actualizas el valor en la línea de texto correspondiente.

 **Guardar el archivo actualizado**: Después de leer todo el archivo y actualizar el stock, escribes todas las líneas (incluyendo la que ha sido modificada) de vuelta al archivo, sobrescribiéndolo.

 **Mensajes de éxito o error**: Muestra mensajes al usuario según si el producto fue encontrado y si el archivo fue actualizado correctamente.

Registro de Salida de Productos:

 **Interfaz gráfica**: Similar a la pantalla de registro de entrada, aquí se crean componentes para el ingreso del código del producto y la cantidad a retirar.

 **Botón "Registrar Salida"**: Este botón será el encargado de registrar la salida de productos cuando se haga clic en él.

 **Validaciones iniciales**: Se validan el código del producto y la cantidad para asegurarse de que el usuario haya ingresado valores correctos.

 **Búsqueda en el archivo**: El archivo productos.txt se lee línea por línea, buscando un producto cuyo código coincida con el ingresado por el usuario.

 **Verificación del stock**: Se verifica si la cantidad de stock actual es suficiente para realizar la salida. Si no es suficiente, se muestra un mensaje de error.

 **Actualización del stock**: Si el stock es suficiente, se actualiza la cantidad de stock en la línea correspondiente y se guarda en una lista.

 **Escritura del archivo**: Después de encontrar el producto y actualizar su stock, el archivo se reescribe con los cambios.

 **Mensaje de éxito**: Si todo sale bien, se muestra un mensaje indicando que la salida fue registrada correctamente.

 **productoEncontrado** es una variable booleana que inicialmente es false. Si se encuentra una coincidencia con el código del producto, la variable se actualiza a true.

 Si al terminar la lectura del archivo el valor sigue siendo false, significa que no se encontró el producto, y por lo tanto, se muestra el mensaje de error.

 **Solicitar la cantidad**: Usamos JOptionPane.showInputDialog para pedirle al usuario la cantidad de productos que desea vender.

 **Validar la cantidad**: Verificamos que la cantidad sea mayor a 0 y que sea un número válido.

 **Comprobar el stock**: Se compara la cantidad solicitada con el stock actual del producto para asegurarse de que el usuario no venda más de lo que hay en existencia.

 **Actualizar el stock**: Si la venta es válida, se actualiza el stock restando la cantidad solicitada.

 **Verificación del stock**: El sistema compara la cantidad que el usuario desea vender (cantidadSalida) con el stock disponible (stockActual).

 **Manejo de error**: Si el usuario intenta vender más de lo que hay en stock, se muestra un mensaje de error y el proceso se detiene.

 **Verificación de stock**: El sistema verifica si la cantidad a vender es mayor que el stock actual.

 **Mensaje de error**: Si la cantidad a vender es mayor al stock, el sistema muestra un mensaje diciendo que no hay suficiente stock.

 **Actualización del stock**: Si la cantidad a vender es válida, se actualiza el archivo de productos restando la cantidad vendida del stock actual.

 **Verificación de la cantidad de productos**: Se comprueba si la cantidad solicitada a vender es menor o igual al stock disponible.

 **Actualización del stock**: Si la cantidad es válida, se actualiza el stock restando la cantidad vendida del stock actual.

 **Escritura en el archivo**: Después de actualizar el stock, se reescribe el archivo de productos con los nuevos valores.

 **LocalDateTime.now()**: Obtiene la fecha y hora actuales.

 **DateTimeFormatter**: Formatea la fecha y hora en un formato legible.

 **BufferedWriter**: Abre el archivo ventas.txt en modo append (true para agregar nuevas líneas sin sobrescribir el contenido existente).

 **Registro de venta**: Incluye la fecha, hora, código del producto y cantidad de productos vendidos.

Generación de Alertas de Stock Bajo

 **TimerTask**: Esta clase permite ejecutar una tarea repetidamente en intervalos de tiempo definidos. El método verificarStock se ejecutará cada 24 horas (puedes cambiar este intervalo según lo que necesites).

 **Verificación del stock**: Cada vez que se ejecute el temporizador, se leerá el archivo productos.txt y se verificará si algún producto tiene menos de 10 unidades de stock (puedes ajustar este nivel según lo que desees).

 **Generación de alertas**: Si el stock es bajo, se mostrará una alerta al usuario y también se guardará en un archivo de texto (alertas\_stock.txt)

 **Archivo de productos**: Ahora, el archivo productos.txt contiene un nuevo campo para el **punto de reorden** de cada producto, lo cual permite al sistema saber cuándo generar una alerta de stock bajo.

 **Verificación de stock contra el punto de reorden**: La línea if (stockActual <= puntoReorden) es clave para verificar si el stock es bajo y si debe activarse una alerta.

 **Generación de alertas**: Si el stock está por debajo del punto de reorden, se genera una alerta y se guarda un registro en el archivo alertas\_stock.txt

 **Verificación periódica del stock**: El sistema sigue revisando el archivo de productos cada 24 horas (o cualquier intervalo que prefieras) para verificar el stock de cada producto.

 **Notificación visual al usuario**: Cuando se detecta que el stock de un producto está por debajo o igual al punto de reorden, el sistema muestra una notificación emergente (popup) usando JOptionPane.showMessageDialog con un ícono de advertencia.

 **Registro de la alerta**: Además de la notificación emergente, la alerta también se registra en un archivo de texto (alertas\_stock.txt), que puede ser útil para revisar el historial de alertas de stock bajo.

*  El mensaje de alerta contiene:
  + **Nombre del producto**: 'nombreProducto'.
  + **Código del producto**: 'codigoProducto'.
  + **Stock actual**: stockActual (en unidades).
  + **Punto de reorden**: puntoReorden (en unidades).

Esto hace que el mensaje sea claro y contenga toda la información relevante que el usuario necesita para tomar decisiones.

 **Notificación visual**:

* Usa JOptionPane.showMessageDialog con un ícono de advertencia (WARNING\_MESSAGE) para que el usuario vea de inmediato que se trata de una alerta de stock bajo.

 **Historial de alertas**:

* Además de mostrar la alerta en pantalla, se guarda el mensaje en un archivo (alertas\_stock.txt), lo que permite llevar un control de todas las alertas generadas.

 **Lista de productos (JList)**:

* Se crea un JList que muestra los productos cargados desde el archivo productos.txt. Solo se muestra el nombre de cada producto.

 **Cargar productos desde el archivo**:

* El método cargarProductosDesdeArchivo() lee el archivo productos.txt y carga los nombres de los productos en la lista. Se asume que el formato del archivo sigue el patrón de campos separados por |, y el nombre del producto está en la segunda posición (datosProducto[1]).

 **Botón "Ver Historial"**:

* El botón permite al usuario ver el historial de movimientos del producto seleccionado. Se obtiene el nombre del producto seleccionado en la lista y luego se busca ese nombre en el archivo de historial de movimientos (movimientos\_stock.txt).

 **Mostrar el historial**:

* Si se encuentran movimientos en el archivo movimientos\_stock.txt que contienen el nombre del producto, estos se muestran en un cuadro de diálogo con un JTextArea. Si no hay movimientos registrados, se informa al usuario.
*  **Registro de movimientos**: Cada vez que haya una operación de entrada o salida, se agregará al archivo de historial con los detalles necesarios.
*  **Visualización del historial**: El usuario ingresa el código del producto, y el sistema busca en el archivo de historial y muestra todas las entradas relacionadas con ese producto en un JTextArea.

Pedidos de Compra

Este código crea una ventana en la que el usuario puede seleccionar un proveedor y presionar un botón para crear un pedido. El siguiente paso sería conectar esta funcionalidad a tu sistema para que el pedido se registre correctamente.

1. **Proveedor**: Se selecciona en un JComboBox.
2. **Productos**: Se utiliza un JList con selección múltiple para que el usuario pueda seleccionar varios productos.
3. **Botón Crear Pedido**: Al presionar el botón, se valida que haya un proveedor seleccionado y al menos un producto seleccionado. Si todo es correcto, se muestra un mensaje de confirmación.
4.  **Map para productos y cantidades**: Se crea un Map<String, Integer> para asociar cada producto seleccionado con la cantidad ingresada.
5.  **JOptionPane para ingresar la cantidad**: Después de seleccionar los productos, el sistema solicita la cantidad para cada uno utilizando un JOptionPane.showInputDialog().
6.  **Validación de cantidad**: Se verifica que la cantidad ingresada sea válida (entero mayor que 0) y se maneja cualquier error de entrada.
7.  **Confirmación del pedido**: Al final, se muestra un mensaje con el proveedor seleccionado y los productos con sus cantidades.

 **Mapa de precios**: He agregado un Map<String, Double> llamado preciosProductos para almacenar el precio unitario de cada producto.

 **Cálculo del total**: Mientras el usuario ingresa la cantidad para cada producto, se multiplica la cantidad por el precio del producto y se acumula en la variable totalPedido.

**Mostrar el total del pedido**: Una vez que se han ingresado todas las cantidades, se muestra el total del pedido en un cuadro de diálogo junto con el proveedor y los productos.

 **Construcción de un resumen del pedido**: Se utiliza un StringBuilder para generar un resumen que contiene el proveedor, los productos seleccionados, las cantidades, los precios unitarios y el total del pedido.

 **Confirmación del pedido**: Antes de proceder, el sistema muestra un cuadro de diálogo con la opción de confirmar o cancelar el pedido. Si el usuario selecciona "Sí" (JOptionPane.YES\_OPTION), el sistema procede con el pedido, de lo contrario, el pedido es cancelado.

 **Proceso de guardar el pedido**: En el bloque donde el usuario confirma el pedido, puedes implementar la lógica para guardar el pedido en un archivo o base de datos, dependiendo de tu implementación.

 **UUID.randomUUID().toString()** genera un código único que puede ser utilizado como identificador para cada pedido.

 Cada vez que se crea un nuevo pedido, se invoca el método generarCodigoPedido() para asignar un código diferente.

 El código generado es una cadena alfanumérica única.

 **Generación de código único**: El método generarCodigoPedido() usa UUID para garantizar que cada pedido tenga un código único.

 **Guardar pedido**: El método guardarPedidoEnArchivo() guarda la información del pedido en un archivo de texto (pedidos.txt), agregando el código del pedido, el proveedor, los productos, las cantidades y el total.

 **Confirmación**: Después de guardar el pedido, el sistema muestra un mensaje de éxito utilizando JOptionPane.showMessageDialog(), indicando que el pedido se ha creado correctamente junto con su código.

 **Carga de datos**: El código lee el archivo pedidos.txt y agrega cada línea (pedido) a un DefaultListModel, que luego se muestra en un JList.

 **Interfaz gráfica**: El JList está dentro de un JScrollPane para permitir desplazarse si hay muchos pedidos.

 **Cerrar la ventana**: El botón "Cerrar" permite al usuario cerrar la ventana una vez que ha revisado los pedidos.

 **Panel de Filtros**: Se agrega un panel con campos de texto para el proveedor, fecha y producto, y un combo box para seleccionar el estado del pedido (Pendiente, En curso, Completado o Todos).

 **Botón Filtrar**: Al presionar el botón "Filtrar", el sistema lee los pedidos desde el archivo de texto, aplica los criterios de búsqueda y muestra solo los pedidos que coincidan con los filtros.

 **Botón Limpiar Filtros**: Limpia todos los filtros para que se carguen nuevamente todos los pedidos.

 **Formato de archivo**: El archivo pedidos.txt podría tener el formato

 **JSplitPane**: Usamos un JSplitPane para dividir la pantalla en dos partes: la lista de pedidos a la izquierda y los detalles del pedido a la derecha.

 **JList para pedidos**: La lista de pedidos muestra los pedidos cargados desde el archivo pedidos.txt. Cada elemento tiene la información del pedido en formato CódigoPedido | Proveedor | Fecha | Estado | Producto | Total.

 **Acción de selección**: Al seleccionar un pedido, se extrae su información y se muestran los detalles en el área de texto de la derecha (JTextArea).

 **Carga de pedidos**: Se cargan los pedidos desde el archivo pedidos.txt y se añaden al modelo de la lista (DefaultListModel).

 **Formato del archivo de pedidos**: El archivo pedidos.txt debería tener el formato adecuado, por ejemplo:

P001 | Proveedor A | 2024-10-15 | Pendiente | Producto X | 500.00

**Formato de archivo de pedidos**: El archivo pedidos.txt ahora debe incluir más información para cada pedido, en este formato:

CódigoPedido | Proveedor | FechaCreación | FechaEntrega | Producto1:Cantidad,Producto2:Cantidad | Total

 **Mostrar productos y cantidades**:

* En el código, los productos solicitados y sus cantidades se almacenan como un campo de texto en formato Producto:Cantidad, separados por comas para varios productos.
* Al mostrar los detalles, se hace un bucle para dividir los productos y sus cantidades y mostrarlos en el área de texto.

 **Nuevos campos**:

* Se agregaron las fechas de creación y entrega estimada, que se extraen del archivo de texto y se muestran al usuario.
* El campo de productos solicitados también incluye las cantidades de cada producto.

 **Visualización del estado actual**:

* En la sección de detalles del pedido, se muestra el estado actual del pedido y se actualiza el JComboBox con el estado correspondiente.

 **Modificación del estado**:

* Cuando el usuario selecciona un nuevo estado en el combo box y presiona el botón "Modificar Estado", se actualiza el estado en el archivo de texto, y la lista se recarga con los cambios.

 **Actualización en el archivo**:

* Se lee el archivo completo, se busca la línea correspondiente al pedido seleccionado, y se reemplaza esa línea por la nueva con el estado modificado.

 **Notificación al usuario**:

* Después de modificar el estado, se muestra un mensaje de confirmación al usuario indicando que el estado ha sido actualizado correctamente
*  **Filtro de pedidos pendientes**: El método cargarPedidosPendientesDesdeArchivo() carga solo los pedidos cuyo estado es "Pendiente", para que el usuario pueda seleccionar uno de ellos para su recepción.
*  **Recepción del pedido**: Al presionar el botón "Recibir Pedido", el sistema verifica si el estado del pedido es "Pendiente". Si lo es, cambia el estado a "Completado" y actualiza el archivo de texto.
*  **Actualización del archivo**: Se usa el método actualizarPedidoEnArchivo() para modificar el estado del pedido en el archivo de texto y guardar los cambios.
*  **Filtro de pedidos pendientes**: El método cargarPedidosPendientesDesdeArchivo() carga solo los pedidos cuyo estado es "Pendiente", para que el usuario pueda seleccionar uno de ellos para su recepción.
*  **Recepción del pedido**: Al presionar el botón "Recibir Pedido", el sistema verifica si el estado del pedido es "Pendiente". Si lo es, cambia el estado a "Completado" y actualiza el archivo de texto.
*  **Actualización del archivo**: Se usa el método actualizarPedidoEnArchivo() para modificar el estado del pedido en el archivo de texto y guardar los cambios.
*  **Tabla para ingresar cantidades recibidas**: Se ha añadido una tabla donde el usuario puede modificar la cantidad de productos recibidos.
*  **Validación de cantidades**: Antes de procesar la recepción del pedido, el sistema verifica que las cantidades recibidas no excedan las solicitadas.
*  **Actualización del pedido**: Las cantidades recibidas se actualizan en el pedido, que luego se guarda en el archivo de texto
*  **Validación de cantidades**: En el bucle que recorre la tabla de productos, se verifica si la cantidad recibida es mayor a la solicitada. Si lo es, se establece cantidadesCorrectas = false y se muestra un mensaje de error mediante un JOptionPane.
*  **Bloqueo de la recepción**: Si alguna cantidad es incorrecta, el proceso de recepción se detiene, y no se procede con la actualización del estado ni con el almacenamiento de los productos en el archivo de texto.
*  **Notificación de éxito**: Si todas las cantidades son correctas, se procede con la actualización y se informa al usuario que el pedido ha sido recibido correctamente.
*  **Detección de exceso de productos**: El sistema ahora verifica si la cantidad recibida es mayor que la cantidad solicitada en cada iteración del bucle. Si esto ocurre, se muestra una advertencia mediante un JOptionPane que notifica al usuario sobre el exceso de productos.
*  **Mensaje de advertencia**: El mensaje informa al usuario cuál es la cantidad solicitada y la cantidad recibida, destacando el exceso.
*  **Control de flujo**: Dependiendo de la lógica de negocio, puedes decidir si el proceso debe detenerse (como está implementado en este ejemplo) o si el usuario puede seguir con la recepción a pesar del exceso. Si decides continuar, puedes eliminar el break para permitir que el proceso siga actualizando el pedido.

 **Detección de productos faltantes**: Se añade una verificación para ver si la cantidad recibida es menor que la cantidad solicitada. Si esto ocurre, el sistema muestra un mensaje de advertencia informando al usuario de los productos faltantes.

 **Mensaje de advertencia**: Al igual que con el exceso de productos, se muestra un JOptionPane que detalla cuántos productos se solicitaron y cuántos se recibieron, indicando la diferencia.

 **Control de estado del pedido**: Si hay productos faltantes, se puede marcar el estado del pedido como "En curso", indicando que el pedido no está completamente recibido. Si no faltan productos, el estado se cambia a "Completado".

 **Comportamiento del sistema**: Dependiendo de las necesidades del sistema, podrías permitir que el proceso de recepción continúe aunque falten productos, manteniendo el pedido en estado "En curso" hasta que se complete.

 **Bandera todasCantidadesIguales**: Se ha añadido una nueva bandera para verificar si todas las cantidades recibidas coinciden exactamente con las cantidades solicitadas. Si al final del bucle, esta bandera sigue siendo true y no se detectan faltantes ni excesos, el estado del pedido se cambiará a "Completado".

 **Estado del pedido**: Si todas las cantidades recibidas son iguales a las solicitadas, el sistema actualiza el estado del pedido a "Completado". Si hay productos faltantes, el sistema cambia el estado a "En curso". El sistema no permitirá completar el pedido si hay productos en exceso.

 **Mensajes de advertencia**: El sistema continúa mostrando mensajes de advertencia si hay productos en exceso o faltantes, y no permitirá completar el pedido hasta que los datos sean correctos.

 **Leer productos desde archivo**: Leemos el archivo productos.txt línea por línea, separando los datos de cada producto usando el delimitador |.

 **Actualizar stock**: Verificamos si cada producto en el pedido recibido está presente en la lista de productos del inventario. Si lo está, sumamos la cantidad recibida al stock actual.

 **Reescribir archivo**: Una vez que todas las cantidades de stock se han actualizado, volvemos a escribir el archivo de productos con los valores modificados.

 **Mensajes de confirmación**: El sistema muestra mensajes de confirmación o error según corresponda.

 **Fecha y hora actual**: Se obtiene la fecha y hora usando LocalDateTime.now() y se formatea con DateTimeFormatter.

 **Generar registro**: Se crea una cadena que incluye la fecha, el código del pedido y los productos recibidos. La información de cada producto se extrae del arreglo productosRecibidos, que contiene los productos en el formato codigoProducto:cantidad.

 **Guardar en archivo**: El registro se guarda en un archivo de texto (recepciones.txt) que acumula los registros de recepción.

15/10/2024

Informes y Estadísticas

Generación de Informes de Inventario Actual

 **Clase de interfaz gráfica**: Creamos una nueva ventana InformeInventarioScreen que contendrá una tabla o lista para mostrar los productos.

 **Cargar datos de inventario**: Leer los productos desde el archivo productos.txt y mostrarlos en la tabla.

 **Filtrado**: Añadir una opción para filtrar por categoría.

El valor total se calcula en la columna de "Valor Total Stock" multiplicando el precio de venta por la cantidad de stock

El código incluye un filtrado por categoría y el informe se puede ordenar modificando el modelo de tabla o implementando una opción adicional para ordenar.

El método exportarA\_CSV implementa la exportación de los datos actuales de la tabla a un archivo CSV.

Generación de Informes de Movimientos de Stock

 **Combo de productos**: Se cargan los productos desde el archivo productos.txt.

 **Rango de fechas**: El usuario introduce manualmente las fechas de inicio y fin.

 **Generación del informe**: Filtra el archivo movimientos\_stock.txt para mostrar las entradas correspondientes al producto seleccionado y al rango de fechas.

 **Exportación a CSV**: Crea un archivo CSV con los datos filtrados.

Generación de Informes de Compras:

 **Selección de proveedor y fechas**: El usuario selecciona un proveedor y define un rango de fechas para generar el informe.

 **Carga de compras**: Se filtran las compras por proveedor y fechas, y se muestran en una tabla.

 **Exportar a CSV**: El informe mostrado en la tabla se puede exportar a un archivo CSV.

Visualización de Gráficas y Estadísticas

 **Cargar datos:** El método cargarInventarioDesdeArchivo lee los datos del archivo productos.txt y extrae el nombre del producto y la cantidad en stock.

 **Crear el gráfico:** Utiliza **JFreeChart** para crear un gráfico de barras vertical.

 **Panel de gráficos:** El gráfico se inserta dentro de un ChartPanel y se muestra en la ventana.

 Usa botones o JComboBox para permitir al usuario seleccionar filtros como **categoría**, **proveedor** o **rango de fechas**.

 Por ejemplo, al seleccionar una categoría, el gráfico puede actualizarse para mostrar solo los productos dentro de esa categoría.

 **Interacción:** Asegúrate de que el usuario pueda seleccionar filtros y modificar la vista de las gráficas dinámicamente.

 **Datos correctos:** Revisa que los archivos como productos.txt, ventas.txt, compras.txt y movimientos\_stock.txt estén actualizados y con el formato correcto.

 **JFreeChart:** Si usas **JFreeChart**, asegúrate de agregar la dependencia en tu proyecto.

Seguridad

**Cifrado de contraseñas**: Almacenar contraseñas encriptadas en lugar de texto plano en tu archivo de usuarios.

* Utiliza un algoritmo de hash, como SHA-256, para cifrar las contraseñas.
* Verifica las contraseñas comparando los hashes almacenados.
*  **Clase Usuario**: Incluye un campo de tipo String o enum para el rol.
*  **Control de Acceso**: Dependiendo del rol, mostrar u ocultar opciones en la interfaz gráfica.

**Restricción de Acceso en Pantallas**:

* En tu LoginScreen, después de validar las credenciales, puedes obtener el rol del usuario.
* En las diferentes pantallas del sistema, verifica el rol para habilitar/deshabilitar funcionalidades específicas:

**Ejemplo de Registro de Accesos:**

1. **Método para registrar accesos**: Cada vez que un usuario inicia o cierra sesión, puedes escribir la información en un archivo.

**Llamada al Registro**:

* Llama a RegistrarAcceso al iniciar sesión y a RegistrarSalida cuando el usuario cierra sesión o cierra la aplicación.
*  **Seguridad**: Contraseñas cifradas con SHA-256, autenticación de usuarios.
*  **Roles**: Implementación de roles y permisos para cada funcionalidad.
*  **Registro de Actividades**: Archivo de texto log\_accesos.txt para registrar entradas y salidas de usuarios.

Seguridad

**Cifrado de contraseñas**: Almacenar contraseñas encriptadas en lugar de texto plano en tu archivo de usuarios.

* Utiliza un algoritmo de hash, como SHA-256, para cifrar las contraseñas.
* Verifica las contraseñas comparando los hashes almacenados.

 **Clase Usuario**: Incluye un campo de tipo String o enum para el rol.

 **Control de Acceso**: Dependiendo del rol, mostrar u ocultar opciones en la interfaz gráfica.

**Restricción de Acceso en Pantallas**:

* En tu LoginScreen, después de validar las credenciales, puedes obtener el rol del usuario.
* En las diferentes pantallas del sistema, verifica el rol para habilitar/deshabilitar funcionalidades específicas:

**Método para registrar accesos**: Cada vez que un usuario inicia o cierra sesión, puedes escribir la información en un archivo.

Llama a RegistrarAcceso al iniciar sesión y a RegistrarSalida cuando el usuario cierra sesión o cierra la aplicación.

Escabilidad

 **Diseño modular**: Asegúrate de que tu código esté organizado en módulos o paquetes bien definidos, como ya lo estás haciendo. Esto facilita la expansión y modificación de las funcionalidades.

 **Principio SOLID**: Aplica principios de diseño como SOLID para que tu código sea fácil de mantener y escalar a medida que crezcan las necesidades del negocio.

 **Control de concurrencia**: Si planeas agregar más usuarios, es importante diseñar mecanismos que eviten colisiones de datos en la base de datos o bloqueos en los archivos de texto.

 **Estructuras de datos eficientes**: Usa estructuras de datos adecuadas, como listas enlazadas, mapas hash o árboles binarios, para manejar productos y consultas internas.

 **Carga diferida (lazy loading)**: Si tienes una interfaz que muestra una lista de productos, implementa la carga diferida para no cargar todos los datos de una sola vez. Solo los productos visibles en la pantalla deben cargarse al inicio, y el resto debe cargarse bajo demanda.

 **Paginación**: Implementa paginación en las tablas de productos en la interfaz gráfica. Esto reducirá la carga inicial y mejorará la experiencia del usuario.

 **Interfaz simple y clara**: Sigue los principios de diseño de interfaces simples, con botones bien organizados y accesibles. Evita sobrecargar al usuario con demasiadas opciones en un solo lugar.

 **Accesibilidad**: Asegúrate de que la interfaz sea accesible, con textos legibles, colores contrastantes y componentes de fácil interacción. Puedes agregar manuales o tutoriales de ayuda en caso de que los usuarios menos experimentados lo necesiten.

 **Personalización por rol**: Según el rol de usuario (administrador, vendedor, encargado de bodega), adapta las opciones disponibles para que la interfaz no sea abrumadora. Solo muestra las funciones relevantes para el rol.

*  El sistema de gestión de inventarios debe poder conectarse a otros sistemas de la empresa, como el de ventas o contabilidad, mediante APIs o intercambio de archivos (CSV, XML, JSON).
* Deben definirse endpoints seguros para la comunicación entre sistemas si se utilizan APIs RESTful.
* Las credenciales de autenticación y métodos de autorización para los otros sistemas deben estar correctamente configurados.

 **Flujos de integración:**

* Se debe definir el flujo de datos entre el sistema de inventarios y los sistemas externos. Por ejemplo:
  + Cuando se registre una venta, el sistema de ventas puede notificar al sistema de inventarios para actualizar el stock.
  + El sistema de contabilidad puede recibir informes sobre movimientos de stock para hacer un seguimiento financiero.

 **Ejemplo de integración con una API RESTful:**

* Para integrar el sistema de inventarios con un sistema de ventas, podrías tener una clase que se encargue de enviar y recibir información a través de una API.
*  El sistema de gestión de inventarios permite la exportación e importación de datos para integrarse con otros sistemas.
* Los datos pueden ser compartidos en formatos como CSV o JSON para facilitar la integración.
* Se pueden realizar solicitudes automáticas para la actualización del stock desde el sistema de ventas, y generar reportes para el sistema de contabilidad.

 **Guía de uso:**

* El usuario puede generar un informe de inventario que será enviado automáticamente al sistema de contabilidad al final del día.
* El sistema de ventas puede comunicarse directamente con el sistema de gestión de inventarios para actualizar el stock después de cada venta.

FINAlGORITMO