**EXPLICACION PROGRAMA M3**

1. **Importaciones de paquetes:** El código comienza con una serie de importaciones que traen **clases y métodos** necesarios para trabajar con **interfaces gráficas de usuario (GUI), manejo de eventos, manipulación de archivos, manejo de fechas** y otras operaciones.
2. **Declaración de variables de instancia:** Se definen las variables de instancia para los **componentes de la interfaz de usuario**, como **etiquetas (JLabel)**, **campos de texto (JTextField), botones (JButton)** y otros componentes necesarios para la aplicación.
3. **Inicialización del GUI:** El método **initComponents()** se encarga de inicializar y configurar los componentes de la **interfaz de usuario,** como la **disposición de los elementos** y la **asignación de acciones a los botones**.
4. **Manejo de eventos:** Se definen manejadores de eventos para los botones "Calcular" y "Ver Beneficios". Estos manejadores de eventos **ejecutan las** **acciones necesarias cuando el usuario interactúa con los componentes de la interfaz de usuario.**
5. **Cálculo y gestión de beneficios:** El método calcularBeneficios() realiza el cálculo de los beneficios basado en los datos ingresados por el usuario en los campos de texto y los muestra en una etiqueta. Además, este método acumula los beneficios en un mapa utilizando fechas como claves.
6. **Carga y guardado de beneficios desde y hacia un archivo:** Se implementan métodos para cargar y guardar los beneficios acumulados en un archivo de texto. Esto permite que los beneficios se conserven entre sesiones de ejecución de la aplicación.
7. **Método main():** Este método inicia la aplicación Swing en un hilo de eventos, lo que garantiza que todas las operaciones de la interfaz de usuario se realicen en el hilo de despacho de eventos de Swing.

En resumen, este código representa una aplicación de Java Swing para calcular y administrar los beneficios de la venta de automóviles, con funcionalidades como cálculo de beneficios, visualización de beneficios en un rango de fechas, y almacenamiento persistente de beneficios en un archivo.

**POR PARTES:**

**IMPORTS:**

1. **javax.swing.\*:** Este paquete proporciona **clases** y **componentes** para **crear interfaces gráficas de usuario (GUI)** en Java. Incluye clases **como JFrame, JLabel, JTextField, JButton**, entre otras, que son utilizadas para construir y manipular la **interfaz de usuario**.
2. **javax.swing.event.DocumentEvent** // **javax.swing.event.DocumentListener:**

**DocumentEvent**: Un evento DocumentEvent se **genera** cada vez que hay un **cambio en el contenido**, como cuando el usuario **escribe texto, elimina texto o modifica el formato** del texto.

**DocumentListener**: Cuando implementas un DocumentListener y lo agregas a un campo de texto, puedes **detectar cuándo se realizan cambios** en ese campo de texto y actuar en consecuencia. La interfaz DocumentListener tiene **tres métodos principales**: **insertUpdate**, **removeUpdate** y **changedUpdate**, que se llaman cuando se inserta, se elimina o se modifica respectivamente el texto en el campo de texto.

Por lo tanto, significa que estos **DocumentListener** están configurados para **observar los cambios que el usuario realiza en los campos de texto y ejecutar cierto código cada vez que se produce un cambio.** Esto podría ser útil para actualizar otras partes de la interfaz de usuario, realizar validaciones en tiempo real, o realizar cálculos automáticos basados en la entrada del usuario.

1. **javax.swing.text.Document:** Esta clase representa un modelo de **documento de texto editable o no. Se utiliza con los listeners** de documentos para **detectar cambios en los campos de texto**.
2. **java.awt.event.ActionEvent y java.awt.event.ActionListener:** se utilizan para **manejar eventos de acción generados por los componentes de la interfaz de usuario, como los botones**. El ActionListener escucha los eventos de acción y realiza las acciones correspondientes cuando se activan.

Los **eventos de acción** en Java son eventos **generados** por la **interacción del usuario con** los componentes de la **interfaz de usuario**, y se manejan **mediante ActionListener y ActionEvent** para realizar acciones específicas en respuesta a esos eventos.

1. **java.io.\*:** Este paquete proporciona clases para **realizar operaciones de entrada y salida, como lectura y escritura de archivos**. En este contexto, se utiliza para manejar la lectura y escritura de datos desde y hacia archivos.
2. **java.text.DecimalFormat y java.text.DecimalFormatSymbols:** se utilizan para **formatear números decimales.** Permiten especificar el **formato** de cómo se deben mostrar los **números,** como la cantidad de decimales y el separador de miles.
3. **java.time.LocalDate y java.time.format.DateTimeFormatter:** se utilizan para **trabajar con fechas**. **LocalDate** representa una **fecha solamente**, mientras que **DateTimeFormatter se utiliza para formatear y analizar fechas y horas**.
4. **java.util.HashMap y java.util.Map:** Estas clases son parte del paquete java.util y se utilizan para **trabajar con colecciones de tipo mapa**. **HashMap** es una implementación de la interfaz Map que **almacena pares de clave-valor**. En este contexto, se utiliza para **almacenar los beneficios acumulados en un mapa donde las claves son las fechas y los valores son los beneficios.**
5. **com.toedter.calendar.JCalendar:** Esta clase es parte de una **biblioteca externa** y se utiliza para **mostrar un calendario en la interfaz de usuario**. Es probable que se utilice para permitir al usuario **seleccionar fechas** en la aplicación.

public class BeneficiosUsuarioUI extends JFrame {

private JLabel lblTipoCoche, lblCostoAuto, lblPrecioVenta, lblGastosMantenimiento, lblImpuestos, lblBeneficios;

private JTextField txtCostoAuto, txtPrecioVenta, txtGastosMantenimiento, txtImpuestos;

private JButton btnCalcular, btnVerBeneficios;

private JCalendar calendarInicio, calendarFin;

private Map<LocalDate, Double> beneficiosPorFecha;

private DecimalFormat formatoDecimal;

private JComboBox<String> comboBoxMarcas;

private JTextField txtTipoCoche;

private static final String FILENAME = "beneficios.txt";

1. **JLabel**: Representa una **etiqueta de texto** que se utiliza para **mostrar texto en la interfaz de usuario**. En este caso, se utilizan para **etiquetar** diferentes **campos de entrada** y **mostrar información al usuario**.
2. **JTextField**: Es un **campo de texto** que permite al usuario **ingresar datos**. Estos campos se utilizan para ingresar el costo del automóvil, el precio de venta, los gastos de mantenimiento, los impuestos, y para mostrar el tipo de coche seleccionado.
3. **JButton**: Representa un **botón** en la interfaz de usuario que el usuario puede hacer clic para **realizar ciertas acciones**. En este caso, hay dos botones: uno para calcular los beneficios y otro para ver los beneficios acumulados.
4. **JCalendar**: Es un **componente** que permite al usuario **seleccionar una fecha de un calendario**. Se utiliza en este caso para **seleccionar el rango de fechas para ver los beneficios.**
5. **Map<LocalDate, Double> beneficiosPorFecha**: Es **un mapa que almacena los beneficios acumulados para cada fecha**. La clave del **mapa es un objeto LocalDate que representa la fecha**, y el **valor es un número decimal (Double)** que representa los **beneficios** acumulados para esa fecha.
6. **DecimalFormat**: Se utiliza para **formatear los números decimales en un formato específico**. En este caso, se utiliza para **mostrar los beneficios**
7. **String FILENAME**: Es una constante que representa el **nombre del archivo** donde se **guardarán los beneficios** acumulados.

En resumen, estas variables y componentes representan los elementos de la interfaz de usuario y las estructuras de datos necesarias para calcular y mostrar los beneficios de la venta de automóviles en la aplicación.

public BeneficiosUsuarioUI() {

initComponents();

cargarBeneficios();

formatoDecimal = new DecimalFormat("#,##0.000", new DecimalFormatSymbols(Locale.US));

}

1. **public BeneficiosUsuarioUI()**: Esto define el **constructor de la clase** **BeneficiosUsuarioUI**. Un **constructor** es un **método especial** en una **clase** que se **llama automáticamente cuando creas un objeto de esa clase**.

En este caso, cuando se crea un objeto de la clase **BeneficiosUsuarioUI**, este constructor se ejecuta automáticamente.

1. **initComponents()**: Este es un método que **se llama desde el constructor** para **inicializar todos los componentes de la interfaz de usuario**, como etiquetas, campos de texto, botones, etc.

En otras palabras, **configura la apariencia inicial de la ventana de la aplicación**.

1. **cargarBeneficios()**: Este es otro **método** **que se llama desde el constructor**. Se utiliza para **cargar cualquier información preexistente o almacenada en el programa**. En este caso, parece que se **están cargando beneficios previamente guardados** (probablemente en un **archivo llamado "beneficios.txt"**)**.**
2. **formatoDecimal = new DecimalFormat("#,##0.000", new DecimalFormatSymbols(Locale.US))**: En esta línea, se crea un objeto de la clase **DecimalFormat**, que se utiliza para **formatear números decimales en un formato específico.**

En resumen, el constructor **BeneficiosUsuarioUI()** se encarga de **inicializar los componentes de la interfaz de usuario**, **cargar información guardada** y **configurar el formato decimal**. Es una parte importante del proceso de inicialización de la aplicación.

private void initComponents() {

setTitle("Calculadora de Beneficios del Usuario");

setSize(400, 400);

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

setLocationRelativeTo(null);

lblTipoCoche = new JLabel("Tipo de coche:");

lblCostoAuto = new JLabel("Costo del automóvil:");

lblPrecioVenta = new JLabel("Precio de venta:");

lblGastosMantenimiento = new JLabel("Gastos de mantenimiento:");

lblImpuestos = new JLabel("Impuestos:");

lblBeneficios = new JLabel("Beneficios:");

txtTipoCoche = new JTextField(10);

txtCostoAuto = new JTextField(10);

txtPrecioVenta = new JTextField(10);

txtGastosMantenimiento = new JTextField(10);

txtImpuestos = new JTextField(10);

btnCalcular = new JButton("Calcular");

btnCalcular.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

calcularBeneficios();

}

});

btnVerBeneficios = new JButton("Ver Beneficios");

btnVerBeneficios.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

verBeneficios();

}

});

calendarInicio = new JCalendar();

calendarFin = new JCalendar();

JPanel panel = new JPanel();

GroupLayout layout = new GroupLayout(panel);

panel.setLayout(layout);

layout.setAutoCreateGaps(true);

layout.setAutoCreateContainerGaps(true);

layout.setHorizontalGroup(layout.createSequentialGroup()

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)

.addComponent(lblTipoCoche)

.addComponent(lblCostoAuto)

.addComponent(lblPrecioVenta)

.addComponent(lblGastosMantenimiento)

.addComponent(lblImpuestos)

.addComponent(lblBeneficios))

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)

.addComponent(txtTipoCoche)

.addComponent(txtCostoAuto)

.addComponent(txtPrecioVenta)

.addComponent(txtGastosMantenimiento)

.addComponent(txtImpuestos)

.addComponent(btnCalcular)

.addComponent(btnVerBeneficios)

.addComponent(calendarInicio)

.addComponent(calendarFin))

);

layout.setVerticalGroup(layout.createSequentialGroup()

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)

.addComponent(lblTipoCoche)

.addComponent(txtTipoCoche))

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)

.addComponent(lblCostoAuto)

.addComponent(txtCostoAuto))

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)

.addComponent(lblPrecioVenta)

.addComponent(txtPrecioVenta))

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)

.addComponent(lblGastosMantenimiento)

.addComponent(txtGastosMantenimiento))

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)

.addComponent(lblImpuestos)

.addComponent(txtImpuestos))

.addComponent(btnCalcular)

.addComponent(btnVerBeneficios)

.addComponent(lblBeneficios)

.addComponent(calendarInicio)

.addComponent(calendarFin)

);

add(panel);

}

Este código **pertenece a una clase** que **implementa una interfaz gráfica de usuario (GUI)** en Java, **utilizando Swing**.

1. **private void initComponents() {**: Este es el método donde **se inicializan todos los componentes de la interfaz de usuario**, como **etiquetas** (JLabels), **campos de texto** (JTextFields) y **botones** (JButtons).
2. **setTitle("Calculadora de Beneficios del Usuario");**: Establece el **título** de la ventana de la aplicación como "Calculadora de Beneficios del Usuario".
3. **setSize(400, 400);**: Establece el **tamaño de la ventana** en píxeles.
4. **setDefaultCloseOperation(EXIT\_ON\_CLOSE);**: Establece la operación que ocurrirá cuando se cierre la ventana. En este caso, la **aplicación** **se cerrará cuando se cierre la ventana**.
5. **setLocationRelativeTo(null**);: **Centra la ventana** en la pantalla del usuario.
6. Luego, **se crean** **etiquetas** **(JLabels)** para mostrar texto descriptivo en la interfaz de usuario. Por ejemplo, **"Tipo de coche:"**, **"Costo del automóvil:"**, etc.
7. Se **crean campos de texto (JTextFields)** para que el usuario pueda ingresar datos, como el tipo de coche, costo del automóvil, etc.
8. Se **crea un botón** ("Calcular") para que el usuario pueda realizar alguna acción, como calcular los beneficios.
9. **btnCalcular.addActionListener(new ActionListener() {...});**: Se añade un "escuchador" al botón "Calcular". Esto significa que cuando el usuario haga clic en el botón, se ejecutará el código dentro del método **actionPerformed(ActionEvent e)**. En este caso, el método **calcularBeneficios()** se llamará cuando se haga clic en el botón.

**btnVerBeneficios = new JButton("Ver Beneficios");**: Esta línea de código crea un nuevo botón con el texto "Ver Beneficios". Este botón será visible en la interfaz de usuario y permitirá al usuario interactuar con él.

**btnVerBeneficios.addActionListener(new ActionListener() { ... });**: Esta parte del código está agregando un "escuchador" al botón **btnVerBeneficios**. Un "escuchador" es un objeto que está atento a ciertos eventos (como hacer clic en un botón) y ejecuta cierto código cuando ocurre ese evento.

**new ActionListener() { ... }**: Aquí se crea una instancia anónima de la interfaz **ActionListener**. Esta interfaz define un método llamado **actionPerformed(ActionEvent e)** que se ejecutará cuando ocurra un evento, como hacer clic en el botón.

**public void actionPerformed(ActionEvent e) { ... }**: Este es el método **actionPerformed**, que se ejecutará cuando el botón sea presionado. En este caso, el método llama a **verBeneficios()**. Esto significa que cuando el botón "Ver Beneficios" sea presionado, se ejecutará el código dentro del método **verBeneficios()**.



1. Se crean dos objetos **JCalendar** llamados **calendarInicio** y **calendarFin**. Estos objetos representan los componentes de calendario que permitirán al usuario seleccionar fechas.
2. Se crea un objeto **JPanel** llamado **panel**. Un **JPanel** es un contenedor ligero que se utiliza para organizar otros componentes de la GUI.
3. Se crea un objeto **GroupLayout** llamado **layout**. Este objeto se utilizará para especificar cómo se organizarán y colocarán los componentes dentro del **JPanel**.
4. Se establece que el **GroupLayout** debe crear automáticamente espacios entre los componentes y entre el contenedor y los componentes interiores (**layout.setAutoCreateGaps(true)** y **layout.setAutoCreateContainerGaps(true)**).
5. Se define cómo se organizarán horizontalmente los componentes dentro del **JPanel** utilizando el método **setHorizontalGroup()** del **GroupLayout**. En este caso, los componentes se colocan en dos columnas, donde las etiquetas (**JLabels**) se colocan en la columna izquierda y los campos de texto (**JTextFields**), botones (**JButtons**) y calendarios (**JCalendar**) se colocan en la columna derecha.
6. Se define cómo se organizarán verticalmente los componentes dentro del **JPanel** utilizando el método **setVerticalGroup()** del **GroupLayout**. En este caso, los componentes se colocan uno debajo del otro en el mismo orden que se definió en la organización horizontal.
7. Finalmente, el **panel** se agrega al contenedor principal de la ventana utilizando el método **add(panel)**.

En resumen, este método **initComponents()** **inicializa** todos los **componentes visuales de la interfaz de usuario** y **define las acciones que ocurren** cuando el **usuario interactúa con estos componentes**, como hacer clic en un botón.

private void calcularBeneficios() {

// Obtener la fecha actual

LocalDate fechaActual = LocalDate.now();

// Calcular beneficios

double precioVenta = Double.parseDouble(txtPrecioVenta.getText());

double costoAuto = Double.parseDouble(txtCostoAuto.getText());

double gastosMantenimiento = Double.parseDouble(txtGastosMantenimiento.getText());

double impuestos = Double.parseDouble(txtImpuestos.getText());

double beneficios = precioVenta - costoAuto - gastosMantenimiento - impuestos;

// Mostrar los beneficios calculados

lblBeneficios.setText("Beneficios: $" + formatoDecimal.format(beneficios));

// Verificar si ya hay beneficios acumulados para la fecha actual

Double beneficiosAcumulados = beneficiosPorFecha.getOrDefault(fechaActual, 0.0);

// Acumular los beneficios

beneficiosAcumulados += beneficios;

// Actualizar el mapa con los beneficios acumulados

beneficiosPorFecha.put(fechaActual, beneficiosAcumulados);

// Mostrar los beneficios totales hasta la fecha actual

double beneficiosTotales = calcularBeneficiosTotales();

lblBeneficios.setText("Beneficios totales: $" + formatoDecimal.format(beneficiosTotales));

// Guardar los beneficios acumulados

guardarBeneficios();

}

Este método llamado **calcularBeneficios()** se encarga de realizar cálculos relacionados con los beneficios financieros. Aquí está una explicación paso a paso:

1. **LocalDate fechaActual = LocalDate.now();**: Esta línea obtiene la fecha actual del sistema utilizando la clase **LocalDate** de Java 8. Esta fecha se utiliza más adelante para realizar un seguimiento de los beneficios acumulados para la fecha actual.
2. Se obtienen los valores ingresados por el usuario desde los campos de texto (**JTextField**). Estos valores representan el precio de venta, el costo del automóvil, los gastos de mantenimiento y los impuestos asociados a la venta.
3. Se calculan los beneficios restando el costo total del automóvil (incluidos los gastos de mantenimiento e impuestos) al precio de venta. Esto proporciona la ganancia neta que se obtiene de la venta del automóvil.
4. Se muestra el resultado de los beneficios calculados en un **JLabel** llamado **lblBeneficios**. El resultado se muestra como un mensaje de texto concatenado con el formato adecuado utilizando **formatoDecimal**.
5. Se verifica si ya existen beneficios acumulados para la fecha actual en el mapa **beneficiosPorFecha**, que parece ser una estructura de datos que almacena los beneficios acumulados para diferentes fechas.
6. Se acumulan los beneficios actuales con los beneficios acumulados previamente para la fecha actual.
7. Se actualiza el mapa **beneficiosPorFecha** con los beneficios acumulados actualizados.
8. Se calculan los beneficios totales hasta la fecha actual llamando a un método llamado **calcularBeneficiosTotales()**. Presumiblemente, este método calcula los beneficios acumulados desde el inicio del tiempo hasta la fecha actual.
9. Se muestra el resultado de los beneficios totales hasta la fecha actual en el mismo **JLabel** **lblBeneficios**. Es importante tener en cuenta que este paso sobrescribe el mensaje anterior de los beneficios calculados, lo que puede ser un error. Es posible que este paso estuviera destinado a mostrar los beneficios totales en otro componente de la interfaz de usuario.
10. Se guarda el mapa **beneficiosPorFecha** actualizado, lo que presumiblemente implica almacenar los beneficios acumulados en algún tipo de almacenamiento persistente, como un archivo o una base de datos.

En resumen, este método calcula los beneficios financieros asociados a la venta de un automóvil, muestra los resultados en la interfaz de usuario y guarda los beneficios acumulados para su posterior recuperación.

private double calcularBeneficiosTotales() {

// Calcular los beneficios totales acumulados hasta la fecha actual

double beneficiosTotales = 0;

for (double beneficio : beneficiosPorFecha.values()) {

beneficiosTotales += beneficio;

}

return beneficiosTotales;

Este método llamado **calcularBeneficiosTotales()** se encarga de calcular los beneficios totales acumulados hasta la fecha actual. Aquí está una explicación paso a paso:

1. **double beneficiosTotales = 0;**: Se inicializa una variable llamada **beneficiosTotales** con el valor 0. Esta variable se utilizará para acumular los beneficios totales.
2. **for (double beneficio : beneficiosPorFecha.values()) { ... }**: Se inicia un bucle **for** que recorre todos los valores almacenados en el mapa **beneficiosPorFecha**. Cada valor representa los beneficios acumulados para una fecha específica.
3. **beneficiosTotales += beneficio;**: Dentro del bucle **for**, se suma cada valor de beneficio al valor acumulado en **beneficiosTotales**. Esto permite calcular la suma total de todos los beneficios acumulados para todas las fechas.
4. **return beneficiosTotales;**: Una vez que se han sumado todos los beneficios acumulados, el método devuelve el valor total de beneficios acumulados.

En resumen, este método recorre todos los valores de beneficios almacenados en el mapa **beneficiosPorFecha**, los suma y devuelve el total acumulado de beneficios hasta la fecha actual. Esto proporciona una manera de calcular y mostrar los beneficios totales obtenidos desde el inicio del tiempo hasta la fecha actual.

private void verBeneficios() {

// Obtener el rango de fechas seleccionado

LocalDate fechaInicio = calendarInicio.getDate().toInstant().atZone(java.time.ZoneId.systemDefault()).toLocalDate();

LocalDate fechaFin = calendarFin.getDate().toInstant().atZone(java.time.ZoneId.systemDefault()).toLocalDate();

// Calcular los beneficios dentro del rango de fechas

double beneficiosEnRango = 0;

for (Map.Entry<LocalDate, Double> entry : beneficiosPorFecha.entrySet()) {

LocalDate fecha = entry.getKey();

if (fecha.isEqual(fechaInicio) || fecha.isEqual(fechaFin) ||

(fecha.isAfter(fechaInicio) && fecha.isBefore(fechaFin))) {

beneficiosEnRango += entry.getValue();

}

}

// Mostrar los beneficios en el rango de fechas

JOptionPane.showMessageDialog(this, "Beneficios desde " + fechaInicio + " hasta " + fechaFin + ": $" + formatoDecimal.format(beneficiosEnRango));

}

Este método llamado **verBeneficios()** se encarga de mostrar los beneficios acumulados dentro de un rango de fechas seleccionado por el usuario. Aquí está una explicación paso a paso:

1. **LocalDate fechaInicio = calendarInicio.getDate().toInstant().atZone(java.time.ZoneId.systemDefault()).toLocalDate();**: Esta línea obtiene la fecha seleccionada por el usuario en el componente **calendarInicio** y la convierte en un objeto **LocalDate**, que es un tipo de dato utilizado para representar fechas en Java 8 y versiones posteriores.
2. **LocalDate fechaFin = calendarFin.getDate().toInstant().atZone(java.time.ZoneId.systemDefault()).toLocalDate();**: De manera similar, esta línea obtiene la fecha seleccionada por el usuario en el componente **calendarFin** y la convierte en un objeto **LocalDate**.
3. Se inicializa una variable llamada **beneficiosEnRango** con el valor 0. Esta variable se utilizará para acumular los beneficios dentro del rango de fechas seleccionado por el usuario.
4. Se inicia un bucle **for** que recorre todas las entradas (pares clave-valor) en el mapa **beneficiosPorFecha**. Cada entrada representa una fecha y su respectivo beneficio acumulado.
5. Dentro del bucle **for**, se obtiene la fecha de cada entrada y se compara con el rango de fechas seleccionado por el usuario (**fechaInicio** y **fechaFin**).
6. Si la fecha de la entrada está dentro del rango de fechas seleccionado, se suma su beneficio correspondiente al valor acumulado en **beneficiosEnRango**.
7. Después de recorrer todas las entradas en el mapa, se muestra un cuadro de diálogo emergente (**JOptionPane.showMessageDialog()**) que informa al usuario sobre los beneficios acumulados dentro del rango de fechas seleccionado.
8. El mensaje del cuadro de diálogo incluye la fecha de inicio y fin del rango seleccionado, así como el total de beneficios acumulados dentro de ese rango, formateado utilizando **formatoDecimal**.

En resumen, este método permite al usuario seleccionar un rango de fechas y luego muestra los beneficios acumulados dentro de ese rango. Esto proporciona una manera conveniente para que el usuario pueda visualizar los beneficios en un período específico de tiempo.

private void cargarBeneficios() {

beneficiosPorFecha = new HashMap<>();

File file = new File(FILENAME);

if (!file.exists()) {

return;

}

try (BufferedReader reader = new BufferedReader(new FileReader(FILENAME))) {

String line;

while ((line = reader.readLine()) != null) {

String[] parts = line.split(",");

LocalDate fecha = LocalDate.parse(parts[0]);

double beneficio = Double.parseDouble(parts[1]);

// Acumular los beneficios en lugar de sobrescribirlos

beneficiosPorFecha.merge(fecha, beneficio, Double::sum);

}

} catch (IOException e) {

e.printStackTrace();

}

}

Este método llamado **cargarBeneficios()** se encarga de cargar los beneficios acumulados desde un archivo de texto y almacenarlos en una estructura de datos llamada **beneficiosPorFecha**, que parece ser un mapa que relaciona fechas (**LocalDate**) con beneficios acumulados (**Double**). Aquí está una explicación paso a paso:

1. **beneficiosPorFecha = new HashMap<>();**: Se crea un nuevo objeto **HashMap** llamado **beneficiosPorFecha**. Un **HashMap** es una estructura de datos que almacena pares clave-valor, donde cada clave está asociada con un valor.
2. Se crea un objeto **File** llamado **file** que representa el archivo de texto donde se almacenan los beneficios.
3. Se verifica si el archivo especificado por **FILENAME** (presumiblemente una constante que contiene la ruta del archivo) existe en el sistema. Si el archivo no existe, el método se detiene y retorna, ya que no hay beneficios que cargar.
4. Se abre el archivo de texto para lectura utilizando un **BufferedReader**. Un **BufferedReader** permite leer texto desde un flujo de entrada de manera eficiente.
5. Se lee cada línea del archivo de texto una a una hasta que se alcanza el final del archivo.
6. Para cada línea leída, se divide en partes utilizando la coma como delimitador mediante el método **split(",")**. Esto separa la línea en un arreglo de **String** llamado **parts**, donde **parts[0]** contiene la fecha y **parts[1]** contiene el beneficio acumulado.
7. Se convierte la cadena de texto que representa la fecha en un objeto **LocalDate** utilizando el método **LocalDate.parse()**. Esto permite almacenar la fecha en un formato adecuado para su posterior uso en Java.
8. Se convierte la cadena de texto que representa el beneficio acumulado en un valor numérico de tipo **double** utilizando el método **Double.parseDouble()**. Esto permite realizar cálculos numéricos con los beneficios.
9. Se utiliza el método **merge()** del **HashMap** **beneficiosPorFecha** para agregar el beneficio acumulado al mapa. Este método combina el valor existente para la fecha especificada (si hay alguno) con el nuevo beneficio acumulado sumándolos juntos. Esto asegura que los beneficios acumulados se acumulen en lugar de sobrescribirlos si ya existen beneficios para la misma fecha.
10. Si ocurre alguna excepción durante el proceso de lectura del archivo, como un error de E/S (Entrada/Salida), se imprime el rastreo de la excepción (**e.printStackTrace()**) para ayudar en la depuración del problema.

En resumen, este método carga los beneficios acumulados desde un archivo de texto y los almacena en una estructura de datos para su posterior uso en la aplicación. Esto permite que los beneficios acumulados se conserven entre ejecuciones del programa y se utilicen para realizar cálculos y visualizaciones.

private void guardarBeneficios() {

try (BufferedWriter writer = new BufferedWriter(new FileWriter(FILENAME))) {

for (Map.Entry<LocalDate, Double> entry : beneficiosPorFecha.entrySet()) {

LocalDate fecha = entry.getKey();

double beneficio = entry.getValue();

writer.write(fecha.toString() + "," + beneficio);

writer.newLine();

}

} catch (IOException e) {

e.printStackTrace();

}

}

Este método llamado **guardarBeneficios()** se encarga de guardar los beneficios acumulados en un archivo de texto en el disco duro. Aquí está una explicación paso a paso:

1. Se utiliza un **try-with-resources** para abrir y manejar automáticamente un recurso externo (**BufferedWriter**) dentro de un bloque **try**. Este recurso es una forma de escritura de texto que facilita la escritura en un archivo.
2. Se crea un nuevo objeto **BufferedWriter** que se conecta a un archivo de texto utilizando un **FileWriter**. Esto permite escribir en el archivo **FILENAME** (presumiblemente una constante que contiene la ruta del archivo).
3. Se inicia un bucle **for** que recorre todas las entradas (pares clave-valor) en el mapa **beneficiosPorFecha**. Cada entrada representa una fecha (**LocalDate**) y su respectivo beneficio acumulado (**Double**).
4. Dentro del bucle **for**, se obtiene la fecha y el beneficio de cada entrada.
5. Se escribe la fecha y el beneficio en una línea del archivo utilizando el método **write()** del **BufferedWriter**. La fecha se convierte en una cadena de texto utilizando el método **toString()**, seguida por una coma (**,**) que separa la fecha del beneficio acumulado. Luego se escribe el beneficio acumulado.
6. Después de escribir cada línea, se agrega un nuevo salto de línea utilizando el método **newLine()** del **BufferedWriter**. Esto asegura que cada línea se escriba en una nueva línea del archivo.
7. Una vez que se han escrito todas las líneas en el archivo, el bloque **try-with-resources** se encarga de cerrar automáticamente el **BufferedWriter** y liberar los recursos asociados.
8. Si ocurre alguna excepción durante el proceso de escritura en el archivo, como un error de E/S (Entrada/Salida), se imprime el rastreo de la excepción (**e.printStackTrace()**) para ayudar en la depuración del problema.

En resumen, este método guarda los beneficios acumulados en un archivo de texto en el disco duro, donde cada línea del archivo representa una fecha y su respectivo beneficio acumulado. Esto permite que los beneficios se conserven entre ejecuciones del programa y se puedan cargar nuevamente en el futuro.

public static void main(String[] args) {

SwingUtilities.invokeLater(new Runnable() {

public void run() {

new BeneficiosUsuarioUI().setVisible(true);

}

});

}

}

Este fragmento de código es el método principal (**main**) de una aplicación Java que utiliza la biblioteca Swing para crear una interfaz gráfica de usuario (GUI). Aquí está una explicación paso a paso:

1. **public static void main(String[] args) { ... }**: Este es el punto de entrada de la aplicación Java. Cada programa Java debe tener un método **main**, ya que es el punto de partida para la ejecución del programa.
2. **SwingUtilities.invokeLater(new Runnable() { ... });**: **SwingUtilities.invokeLater()** es un método de utilidad de Swing que se utiliza para ejecutar tareas en el hilo de despacho de eventos de Swing (Event Dispatch Thread, EDT). El EDT es responsable de manejar todos los eventos de la interfaz de usuario en una aplicación Swing.
3. **new Runnable() { ... }**: Se crea un nuevo objeto **Runnable**. Un **Runnable** es una interfaz funcional en Java que representa una tarea que puede ejecutarse.
4. **public void run() { ... }**: Se sobrescribe el método **run()** de la interfaz **Runnable**. Este método contiene el código que se ejecutará en el hilo de despacho de eventos de Swing cuando se invoque el método **run()**.
5. **new BeneficiosUsuarioUI().setVisible(true);**: Se crea una nueva instancia de la clase **BeneficiosUsuarioUI**, que probablemente sea la ventana principal de la aplicación. Luego, se llama al método **setVisible(true)** para hacer visible la ventana en la pantalla.
6. **});**: Cierra el bloque de código del **Runnable**.

En resumen, este método **main** se encarga de iniciar la aplicación Swing, asegurándose de que toda la interfaz de usuario se construya y se ejecute correctamente en el hilo de despacho de eventos de Swing. Esto garantiza un comportamiento seguro y adecuado de la interfaz de usuario de la aplicación.

**CONEXIÓN**

Este código Java es un ejemplo básico de cómo establecer una conexión a una base de datos MySQL utilizando la biblioteca JDBC (Java Database Connectivity). Permíteme explicar paso a paso lo que hace:

1. Importa las clases necesarias de la biblioteca JDBC para trabajar con bases de datos en Java.
2. Define una clase llamada Conexión que extiende javax.swing.JFrame, lo que sugiere que es una ventana de una aplicación de escritorio.
3. Declara variables privadas estáticas para la conexión (con), la URL de la base de datos (url), el nombre de usuario (user), y la contraseña (pass) para la base de datos MySQL.
4. En el constructor de la clase Conexión, se inicializa la ventana (initComponents()) y se llama al método conectar() para establecer la conexión a la base de datos.
5. El método conectar() intenta cargar el controlador JDBC de MySQL (com.mysql.cj.jdbc.Driver) utilizando Class.forName(). Luego, utiliza DriverManager.getConnection() para establecer la conexión a la base de datos MySQL utilizando la URL, el nombre de usuario y la contraseña proporcionados. Si la conexión se establece con éxito, muestra un mensaje indicando que la conexión está establecida. Si hay algún error durante la conexión, muestra un mensaje de error.
6. El código dentro del método initComponents() está generado automáticamente por la herramienta de diseño de GUI de NetBeans y define la apariencia y los componentes de la ventana de la aplicación.
7. El método main() es el punto de entrada de la aplicación. Crea una instancia de la clase Conexión y la hace visible.

En resumen, este código Java crea una aplicación de escritorio que intenta conectarse a una base de datos MySQL al iniciarse. Si la conexión se establece correctamente, muestra un mensaje indicando que la conexión está establecida. Si no puede establecer la conexión, muestra un mensaje de error.

private static Connection con; private String url = "jdbc:mysql://localhost:3306/netbeans\_ct"; private String user = "CarlosTarre"; private String pass = "1234";

* Aquí se declaran tres variables:
  + **con**: es una variable de tipo **Connection** que se utilizará para establecer la conexión a la base de datos.
  + **url**: es una cadena que contiene la URL de la base de datos a la que te conectarás. En este caso, es una base de datos MySQL que se encuentra en **localhost** (el mismo ordenador) en el puerto **3306**, y el nombre de la base de datos es **netbeans\_ct**.
  + **user** y **pass**: son el nombre de usuario y la contraseña que se utilizarán para acceder a la base de datos.

1. **Método conectar()**:

javaCopy code

private void conectar() { try { Class.forName("com.mysql.cj.jdbc.Driver"); con = DriverManager.getConnection(url, user, pass); if (con != null) { jLabel2.setText("Conexion establecida"); } } catch (ClassNotFoundException | SQLException e) { jLabel2.setText("Error de conexion: " + e.getMessage()); } }

* Este método intenta establecer una conexión a la base de datos MySQL.
  + **Class.forName("com.mysql.cj.jdbc.Driver")**: Carga el controlador JDBC necesario para establecer la conexión con MySQL. Es una parte necesaria para usar JDBC.
  + **DriverManager.getConnection(url, user, pass)**: Intenta establecer una conexión utilizando la URL de la base de datos, el nombre de usuario y la contraseña proporcionados.
  + **if (con != null) {...}**: Verifica si la conexión se estableció correctamente. Si es así, muestra un mensaje indicando que la conexión está establecida.
  + **catch (ClassNotFoundException | SQLException e) {...}**: Maneja cualquier excepción que pueda ocurrir durante el proceso de conexión, como errores de clase no encontrada o errores de SQL.

1. **Método main()**:

javaCopy code

public static void main(String args[]) { /\* Código para configurar la apariencia de la interfaz de usuario (no relevante para la conexión a la base de datos) \*/ java.awt.EventQueue.invokeLater(new Runnable() { public void run() { new Conexión().setVisible(true); } }); }

* Este es el punto de entrada de la aplicación Java. Crea una nueva instancia de la clase **Conexión** (la ventana de la aplicación) y la hace visible para el usuario.