# Índice

**Punto 1: estructuras de datos** ..... **2**

**Punto 2: lectura de los archivos** ..... **3**

**Punto 3: lectura en segundo plano** ..... **6**

**Punto 4: tabla de películas** .....

Inciso a: componentes, eventos y llenado de la tabla ..... **7**

Inciso b: manejo de los eventos y posibles alternativas ..... **8**

Inciso c: actualización del histograma según datos elegidos ..... **9**

Inciso d: filtros de ordenación por los diferentes campos ..... **10**

**Punto 5: implementación del histograma** ..... **11**

**Punto 6: características distintivas del proyecto** ..... **15**

**Imágenes del programa en funcionamiento** ..... **16**

## Desarrollo del trabajo

1) ¿En qué estructura de datos guardan los datos leídos de los archivos movies.csv y ratings.csv?, ¿Cuál fue el criterio usado para elegir dicha estructura de datos y no otra, qué consideraciones hicieron para elegirla?

**Respuesta:** para guardar toda la información relevante de los archivos fuente (.csv) se creó una clase ***"Datos"***, que tiene en sus atributos una **lista** de objetos *"Película"*.

```java
public class Datos {
    private final List<Pelicula> peliculas;
    private Iterator<Pelicula> iterator;
    private int cantUsuarios;

    public Datos() {
        this.peliculas = new ArrayList<>();
        this.cantUsuarios = 0;
    }
```

Figura 1: atributos y constructor de la clase Datos

```java
public class Pelicula implements Comparable<Pelicula> {
    private String titulo, movieId;

    private final int[] votos;
    private int cantUsuarios;

    // Variables de acceso rápido (para el ordenamiento)
    private float ratingPromedio = -1;
    private int cantVotos;
```

Figura 2: atributos de la clase Película.

Como se puede observar en la Figura 1, en el constructor se define que la lista es una instancia de **ArrayList.** En un principio, debido a que los datos están ordenados por su ***"movieId"*** en los archivos fuente, éstos se agregan en la lista siguiendo dicho criterio.

Las principales razones por la que se eligió **ArrayList** son las siguientes:

1. Es una estructura ordenable, ya que el método estático **sort()** de la clase *Collections* es aplicable para todas las clases que implementen **List**. Esto es importante para permitir el ordenamiento según un atributo en particular.

2. Es una estructura de datos **dinámica**, por lo que se pueden ir agregando más "películas" a medida que se van leyendo los archivos, sin especificar el límite.

3. Los métodos que brinda la interfaz pública de las listas en JAVA, por ejemplo, el método **get(int)**, muy importante en la selección de ítems en las tablas.

**¿Por qué no se eligió otra estructura de datos?**

- **Árbol de búsqueda:** como los datos ya están ordenados por movieId de forma ascendente, los elementos siempre se agregarían en la rama derecha del árbol haciendo que la **altura** sea igual a la cantidad de nodos, por lo tanto, sería ineficiente para búsquedas y se comportaría como una lista enlazada.

- **Set (conjunto):** no permite el ordenamiento directo en base a un criterio.

- **Array (arreglo):** es ordenable, pero de tamaño fijo, por lo que no resulta conveniente si no se conoce a priori la cantidad de películas a procesar.

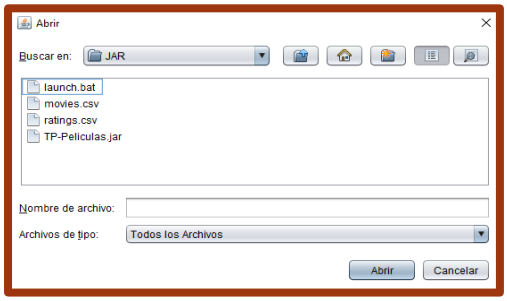
- **Lista enlazada:** satisface las necesidades planteadas anteriormente, pero resulta ineficiente para el método **get(int)** al ser de acceso secuencial.

2) ¿Cómo resolvieron la lectura de los archivos movies.csv y ratings.csv?, ¿Qué clases JAVA usaron para leer los datos?, ¿Qué decisiones tomaron en caso que los archivos no se encuentren en el filesystem o no cuenten con permisos de lectura? Copie el fragmento de código que permita mostrar la solución a la lectura de los archivos.

**Respuesta:** en primer lugar, se realiza la lectura del archivo **movies.csv** para armar los objetos *"Película"*, con sus respectivos IDs y títulos; y posteriormente se lee el archivo **ratings.csv** para procesar los votos realizados a las películas por cada usuario.

Los métodos específicos para leer los archivos se encuentran en una clase ***"Lector"***, que utiliza a la clase **BufferedReader**, que a su vez utiliza una instancia de **FileReader**, dentro de un bloque **try** para manejar las posibles excepciones (código más adelante).

Los archivos .csv son externos al JAR, ubicados preferentemente en el mismo directorio que dicho ejecutable. En caso de que no sean localizados por tener un nombre distinto al esperado o estar en un directorio diferente, se produce la excepción **FileNotFoundException**, y se muestra la advertencia de la izquierda al usuario:


Sin embargo, esto no producirá el cierre de la aplicación; luego de cerrar el cuadro de diálogo se muestra un **JFileChooser**, como se ve a la derecha, para que el usuario pueda examinar la ubicación del archivo faltante para continuar con el procesamiento de los datos.

A continuación, se muestra el código implementado para la lectura de **movies.csv:**

```
private boolean leerPeliculas(Datos datos, File archivo) {
    try (BufferedReader br = new BufferedReader(new FileReader(archivo))) {
        String linea;
        br.readLine();      // Saltear encabezado

        while ((linea = br.readLine()) != null) {
            datos.agregarPelicula(filtrarTitulo(linea), filtrarID(linea));
            int progreso = datos.getCantPeliculas() * PORCE_ONE / 10000;
            progressBar.setValue(progreso);
        }

        progressBar.setValue(PORCE_ONE);
        return true;
    } catch (FileNotFoundException e) {
        this.mostrarErrorArchivoNoEncontrado(archivo.getAbsolutePath());
        File archivoIntento = examinarArchivo();    // JFileChooser
        if (archivoIntento != null) return leerPeliculas(datos, archivoIntento);
    } catch (IOException e) {
        this.mostrarErrorLectura(archivo.getAbsolutePath());
        e.printStackTrace();
    }

    return false;
}
```

El archivo es leído línea por línea, en cada una está el título y movie ID.

A medida que se crean "Películas" de forma interna en Datos, también se actualiza la barra de progreso que se muestra en la interfaz gráfica

**Código respectivo a la extracción del título e ID de una película:**

```java
private String filtrarTitulo(String line) {
    Pattern pattern = Pattern.compile("\"(.*?)\"");
    Matcher matcher = pattern.matcher(line);
    if (matcher.find()) return matcher.group(1);
    else return line.split( regex: ",")[1];
}
```

```java
private String filtrarID(String line){
    return line.split( regex: ",")[0];
}
```

**Código respectivo a los cuadros de diálogos por posibles errores:**

```java
private void mostrarErrorArchivoNoEncontrado(String filepath){
    JOptionPane.showMessageDialog( parentComponent: null,  message: "No se pudo encontrar el archivo " +
                    filepath, Aplicacion.NOMBRE, JOptionPane.ERROR_MESSAGE);
}

private void mostrarErrorLectura(String filepath){
    JOptionPane.showMessageDialog( parentComponent: null,  message: "Ocurrio un problema al leer " +
                    filepath, Aplicacion.NOMBRE, JOptionPane.WARNING_MESSAGE);
    barraError();
}

private void mostrarErrorGeneral(String filepath, String info){
    JOptionPane.showMessageDialog( parentComponent: null,  message: "Ocurrio un error en ejecucion (" +
                    info + ") mientras se procesaba el archivo " + filepath,
            Aplicacion.NOMBRE, JOptionPane.WARNING_MESSAGE);
    barraError();
}
```

**Código respectivo a JFileChooser:**

```java
private File examinarArchivo(){
    File archivo = null;
    JFileChooser chooser = new JFileChooser();
    chooser.setFileSelectionMode(JFileChooser.FILES_ONLY);
    chooser.setCurrentDirectory(new File( pathname: "."));
    if (chooser.showOpenDialog( parent: null) == JFileChooser.OPEN_DIALOG) {
        archivo = chooser.getSelectedFile();
        if ((archivo == null) || (archivo.getName().equals(""))) {
            JOptionPane.showMessageDialog( parentComponent: null,  message: "Nombre de archivo invalido",
                    Aplicacion.NOMBRE, JOptionPane.ERROR_MESSAGE);
            barraError();
        }
    } else barraError();
    return archivo;
}
```

Respecto a un posible error debido a falta de permisos de lectura, la excepción lanzada es también **FileNotFoundException**, al intentar abrir el archivo en el **FileReader**, por lo que la aplicación lo interpretará como si el archivo no existiera (es inaccesible). El método **setReadable(boolean)** de la clase **File** permite cambiar dicho permiso, pero esto no fue implementado en la solución debido a que no era solicitado en el trabajo.

En la siguiente página se muestra el código completo para el procesamiento del archivo **ratings.csv**, que es posterior al de movies.csv ya que requiere de que los IDs de las películas estén cargados para poder asociarlos con los votos de los usuarios.

```java
private boolean leerRatings(Datos datos, File archivo) {
    try (BufferedReader br = new BufferedReader(new FileReader(archivo))){
        String ultimoId = "";
        String linea;
        br.readLine();  // Saltear encabezado
        int votosLeidos = 0;

        // PRECONDICION: La columna de userId esta ordenada de menor a mayor
        while ((linea = br.readLine()) != null){
            // PRECONDICION: Cada fila es un voto diferente
            String[] params = linea.split( regex: ",");
            float rating = getRating(params);

            // En caso de encontrar alguna inconsistencia
            if (rating == -1) throw new Exception("Formato invalido");

            // PRECONDICION: La primer columna es el userId
            String actualId = params[0];
            if (!actualId.equals(ultimoId)){
                datos.incrementarUsuarios();
                datos.reiniciarIterador();
                ultimoId = actualId;
            }

            // Agregar voto
            Pelicula pelicula = datos.getPelicula(params[1]);
            pelicula.incrementarUsuarios();
            pelicula.addVoto(rating);

            if (++votosLeidos % UPDATE_RATE == 0){
                // Actualizar progreso (suponemos 101000 votos)
                int progreso = (votosLeidos) * (PORCE_TWO - PORCE_ONE) / 101000;
                progressBar.setValue(progreso + PORCE_ONE);
            }
        }

        progressBar.setValue(PORCE_TWO);
        return true;
    } catch (FileNotFoundException e) {
        this.mostrarErrorArchivoNoEncontrado(archivo.getAbsolutePath());
        e.printStackTrace();
        // pedirle al usuario que busque el archivo correcto
        File archivoIntento = examinarArchivo();
        if (archivoIntento != null) return leerRatings(datos, archivoIntento);
    } catch (IOException e) {
        this.mostrarErrorLectura(archivo.getAbsolutePath());
        e.printStackTrace();
    } catch (Exception e){
        this.mostrarErrorGeneral(archivo.getAbsolutePath(), e.getMessage());
        e.printStackTrace();
    }

    return false;
}
```

El método para la extracción del rating (requiere convertirlo a Float) es:

```java
// PRECONDICION: La tercera columna existe y es el rating (float)
private float getRating(String[] params) {
    try {
        return Float.parseFloat(params[2]);
    } catch (NumberFormatException e){
        System.err.println("Se encontró el rating " + params[2] + ", que no es un float");
        return -1;
    }
}
```

3) El enunciado del trabajo solicita que la lectura de los archivos se haga bajo demanda y en segundo plano para evitar "freezar" la interfaz de usuario. ¿Cómo resolvió este requerimiento? Copie el fragmento de código que permita mostrar la solución.

**Respuesta:** la solución fue utilizar un **Thread** para la lectura y procesamiento de los archivos, que se ejecuta de forma independiente al *Main Thread*. En este último se delega la interacción con los elementos de la interfaz gráfica de usuario.

El hilo definido para la lectura es un atributo privado de la clase "*PanelPrincipal*", y es instanciado pasándole como parámetro una instancia de "*ProcesamientoDatos*", una clase privada definida en *PanelPrincipal* que implementa la interfaz **Runnable**.

```
public PanelPrincipal() {
    this.datos = new Datos();
    this.setFrame();
    this.setMainPanel();
    this.setProcessPanel();
    this.setContadores();
    this.setTablePanel();
    this.setHistograma();

    this.histograma.revalidate();
    thread = new Thread(new ProcesamientoDatos());
}
```

Figura 3: constructor de la clase PanelPrincipal

```
private void procesarDatos(){
    if (thread.getState() == Thread.State.NEW){
        thread.start();
    } else if (thread.getState() == Thread.State.TERMINATED){
        thread = new Thread(new ProcesamientoDatos());
        thread.start();
    }
}
```

Figura 4: método privado que inicia o vuelve a instanciar el Thread cuando se requiera

En la interfaz gráfica existe un botón de *"Procesar Datos"*, el cual tiene añadido un listener que llama al método ***procesarDatos()*** al hacer clic en él.

```
private class ProcesamientoDatos implements Runnable {

    @Override
    public void run() {
        Lector lector = new Lector(pgbProcesarDatos);
        btnProcesarDatos.setEnabled(false);
        datos = lector.getDatos();

        // actualizar UI con los datos totales
        lblCantUsuarios.setText(String.valueOf(datos.getCantUsuarios()));
        lblCantPeliculas.setText(String.valueOf(datos.getCantPeliculas()));
        lblCantVotos.setText(String.valueOf(datos.getCantVotos()));

        // Habilitar botones
        btnTotalHistograma.setEnabled(true);
        btnProcesarDatos.setEnabled(true);
        btnProcesarDatos.setText(PROCESS_BTN_RETRY);

        model.setPeliculas(datos.getPeliculas());
        model.fireTableDataChanged();
        mostrarHistogramaGeneral();
    }
}
```

```
public Datos getDatos() {
    Datos datos = new Datos();
    progressBar.setForeground(Colores.VERDE.color().brighter());
    progressBar.setValue(0);
    if (!leerPeliculas(datos, new File(MOVIES_PATH))) return datos;
    if (!leerRatings(datos, new File(RATINGS_PATH))) return datos;
    Collections.sort(datos.getPeliculas());
    progressBar.setValue(100);
    return datos;
}
```

Para la lectura se instancia un *Lector*, pasando la barra de progreso de la interfaz como parámetro para que los métodos de lectura la puedan actualizar.

Antes de iniciar la lectura se deshabilita el botón de Procesar Datos hasta que la tarea haya finalizado.

En *getDatos()* de *Lector* se invocan los métodos vistos en la respuesta del inciso 2, y devuelve la información correctamente procesada. Notar que, incluso si se produce un error de lectura, siempre se devolverá una instancia no nula de *Datos*.

Todos los componentes se actualizan según los valores cargados en dicho objeto.

4) Sobre la visualización de la tabla de películas:

a) ¿Qué componentes de GUI utilizan?, ¿Qué eventos de interfaz de usuario se contemplan? Dentro de las componentes utilizadas, se pidió el uso de una tabla, ¿cómo llenan los campos de la tabla?

**Respuesta:** para representar la tabla de películas se utilizó una **JTable** contenida dentro de un **JScrollPane**, de forma que, en caso de tener muchas películas cargadas para visualizar en pantalla, se muestre una barra vertical en la parte derecha para poder desplazarse a lo largo de la tabla.

Para representar la cantidad de películas que el usuario desee visualizar en la tabla se utilizó un **JComboBox**, acompañado de un **JLabel** para describir su función. Se permite seleccionar entre 6 cantidades distintas: 5, 10, 20, 100, 1000, TODAS.


Figura 5: vista de la tabla al finalizar la lectura

Los eventos contemplados son dos:

- **MouseEvent,** cuando se selecciona una o más películas en la tabla. Mediante la adición de un *MouseListener* al componente *JTable*, se muestra el histograma correspondiente de la(s) película(s) seleccionada(s) al soltar el clic.

```
tabla.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseReleased(MouseEvent e) {
        mostrarHistogramaIndividual(tabla.getSelectedRows());
    }
});
```

Figura 5: adición del MouseListener a la tabla

- **ActionEvent,** cuando se selecciona una opción del selector. Mediante la adición de un *ActionListener* al componente *JComboBox*, cuando se hace clic en alguna de las opciones se invoca al método ***actualizarTabla()***, para que la misma limite o expanda la cantidad de elementos a visualizar en la tabla.

```
selector.addActionListener(e -> actualizarTabla());
```

```
private void actualizarTabla(){
    if (selector.getSelectedIndex() == selector.getItemCount()-1){
        // está seleccionada la ultima opción, que es TODOS
        model.setCantidadAMostrar(datos.getPeliculas().size());
    } else {
        // está seleccionada una opción numérica (filtro)
        String selectedOption = (String) selector.getSelectedItem();
        if (selectedOption != null) try {
            int cantInteres = Integer.parseInt(selectedOption);
            model.setCantidadAMostrar(cantInteres);
        } catch (NumberFormatException e){
            System.err.println("El filtro seleccionado no es un número! \n\n" + e.getMessage());
        }
    }

    model.fireTableDataChanged();
}
```

Figura 6: método actualizarTabla(), se invocan métodos del model (pág. siguiente)

En la instanciación del componente *JTable*, se le asignó un *model* personalizado, el cual posee la lista de películas cargadas. Estos son los atributos de la clase model:

```
public class PeliculasTableModel extends AbstractTableModel {
    private final String[] columnNames = {"Nombre de película", "Usuarios", "Rating"};
    private List<Pelicula> peliculas;

    private static final int COLUMN_TITLE   = 0;
    private static final int COLUMN_USERS   = 1;
    private static final int COLUMN_SCORE   = 2;
    private int cantidadAMostrar = 5;
```

La lista de películas y la cantidad de mostrar se pueden "setear" en todo momento. El método ***getRowCount()*** está sobreescrito para devolver adecuadamente la cantidad de filas según la cantidad a mostrar que se haya seteado previamente.

```
@Override
public int getRowCount() { return Math.min(peliculas.size(), cantidadAMostrar); }
```

Respondiendo a la pregunta, para llenar los campos está sobreescrito el método ***getValueAt(int, int)***, de modo que devuelva el objeto correcto para cada celda:

```
@Override
public Object getValueAt(int rowIndex, int columnIndex) {
    Pelicula pelicula = peliculas.get(rowIndex);

    switch (columnIndex){
        case COLUMN_TITLE:
            return pelicula.getTitulo();
        case COLUMN_USERS:
            return pelicula.getCantUsuarios();
        case COLUMN_SCORE:
            return String.format("%.2f", pelicula.getRatingPromedio());
        default:
            throw new IllegalArgumentException("Índice de columna " + columnIndex + " inválido");
    }
}
```

```
@Override
public Class<?> getColumnClass(int columnIndex) {
    if (peliculas.isEmpty()) return Object.class;
    else return getValueAt( rowIndex: 0, columnIndex).getClass();
}
```

Este "llenado" de celdas se realiza cuando desde la clase *PanelPrincipal* se invoca el método ***fireTableDataChanged()*** definido en la superclase del model, para notificar el cambio de datos al finalizar la lectura o si la lista cambia su tamaño u orden.

b) ¿Qué solución eligió para manejar los eventos?, ¿Qué otras formas existen?

**Respuesta:** por un lado, para el manejo del **MouseEvent** en la tabla se utiliza una clase anónima de **MouseAdapter**, como se observa en la *Figura 5* (pág. anterior). Se sobrescribió el método ***mouseReleased(MouseEvent)*** para que muestre el histograma correspondiente según las filas seleccionadas de la tabla. Esto es posible porque los *Adapters* son clases que implementan los métodos del *Listener* asociado, pero sus cuerpos están vacíos, entonces se debe sobrescribir los que sean necesarios.

La razón de por qué se utilizó una clase anónima para este evento es que las sentencias a ejecutar tienen utilidad solo al seleccionar películas. Las otras soluciones son:

- Crear una **clase interna** que extienda de **MouseAdapter** (preferentemente) y agregar una instancia de la misma en el método *addMouseListener* de **JTable**.

- Ídem anterior, pero que la clase no sea interna. Sería necesario pasar los componentes a actualizar, en este caso, el histograma; y también la tabla con su model para identificar las películas seleccionadas (no recomendable).

- Hacer que la clase *PanelPrincipal* implemente *MouseListener*. Sin embargo, los métodos a implementar pueden "solaparse" con otros eventos del mismo tipo generados por otros componentes; y además se deben dejar métodos vacíos. La sentencia donde se añade el listener sería: *tabla.addMouseListener(this)*.

Respecto al manejo del **ActionEvent** generado al elegir una opción del *JComboBox*, en este caso también se utiliza una clase anónima. Se utiliza una expresión lambda porque la clase **ActionListener** sólo tiene un método abstracto, entonces del lado izquierdo de la expresión se ubican los parámetros (el evento) y del lado derecho la función (sentencias) a ejecutar, que por simplificación es la invocación a otro método.

```
selector.addActionListener(e -> actualizarTabla());
```

En la *Figura 6* se detalla el cuerpo del método *actualizarTabla()*. Nuevamente, la solución alternativa que se puede implementar es la creación de una clase interna, debe implementar **ActionListener** y definir el comportamiento de ***actionPerformed***.

c) ¿Es posible elegir los datos de una película y actualizar dinámicamente el histograma con dichos datos? Si lo implementaron, copien el fragmento de código y describan su solución.

**Respuesta:** sí, es posible. Este el código correspondiente al método invocado:

```
private void mostrarHistogramaIndividual(int[] filasSeleccionadas){
    int[] votos = new int[CANT_BARRAS_HISTOGRAMA+1];
    Pelicula peliculaSeleccionada = null;

    for (int numeroFila: filasSeleccionadas) {
        peliculaSeleccionada = model.getPeliculas().get(tabla.convertRowIndexToModel(numeroFila));
        for(int i=1; i <=CANT_BARRAS_HISTOGRAMA; i++) votos[i] += peliculaSeleccionada.getCantVotos(i);
    }

    for (int i=1; i <=CANT_BARRAS_HISTOGRAMA; i++) histograma.cambiarValorBarra(i, votos[i]);

    if (filasSeleccionadas.length > 1)
        lblHistograma.setText("Histograma Combinado para las " +
                filasSeleccionadas.length + " películas seleccionadas");
    else if (peliculaSeleccionada != null)
        lblHistograma.setText("Histograma para " + peliculaSeleccionada.getTitulo());

    histograma.formalizarHistograma();
    histograma.revalidate();
}
```

Como se mencionó en el inciso **a)**, cuando el usuario selecciona una o varias películas de la tabla, luego de soltar el mouse se invoca al método anterior. Recibe un arreglo de enteros que representan los índices de las filas que hayan sido seleccionadas.

La tabla permite obtener las instancias de las películas deseadas realizando la conversión del índice de fila a índice de lista (es la misma que se encuentra en *Datos*).

Por cada película seleccionada se obtienen los votos y se almacenan en un arreglo local según el rating (1-5 estrellas) a modo de conteo para poder representarlos en el histograma. Se prosigue con la actualización de las barras del histograma. Su implementación interna se explica con más detalles en el punto 5 de este trabajo.

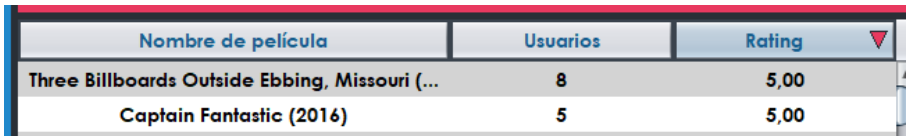
Existe una etiqueta (*JLabel*) llamada *lblHistograma*, situada inmediatamente encima del histograma, a modo de descripción de lo que se muestra en el mismo. Si el usuario seleccionó una única película en la tabla se indicará que el histograma mostrado es para dicha película, mostrando el título; en caso contrario, al seleccionar varias se indica que el histograma es una combinación (suma) de los votos de N películas.

Como paso final, se formaliza el histograma, también explicado en el punto 5, y se invoca al método *revalidate()* por si algún atributo afecta las dimensiones del Layout.

d) ¿Es posible aplicar filtros de ordenación por los diferentes campos: nombre de la película, #usuarios, #votos? Si lo implementaron, describan qué componente de GUI usaron y qué eventos de interfaz de usuario manejaron.

**Respuesta:** sí, es posible. Cuando el usuario hace clic en un encabezado de la tabla se realiza el ordenamiento de los datos según la columna seleccionada, de manera ascendente o descendente alternativamente. Esta función fue implementada mediante la adición a la tabla de una instancia de **TableRowSorter**, al cual se le pasa el model de la tabla en su constructor para que se puedan aplicar los filtros de forma adecuada.

```
TableRowSorter<PeliculasTableModel> sorter = new TableRowSorter<>(model);
tabla.setRowSorter(sorter);
```



| Nombre de película | Usuarios | Rating |
|---|---|---|
| Three Billboards Outside Ebbing, Missouri (... | 8 | 5,00 |
| Captain Fantastic (2016) | 5 | 5,00 |

El único componente de GUI en cuestión son *JTable*, e internamente su *JTableHeader* cuando se realiza el clic en él. Es posible manejar los eventos de forma personalizada mediante la adición de un **RowSorterListener**, como se muestra a continuación:

```
table.getRowSorter().addRowSorterListener(new RowSorterListener() {
    @Override
    public void sorterChanged(RowSorterEvent e) {
        // Sorting changed
    }
});
```

Aunque se apliquen filtros de ordenación, no es necesario modificar la lista de películas contenida en el model, ya que mediante el método ***convertRowIndexToModel(int)*** del componente **JTable** es posible obtener el índice original de carga. Este método se invoca para mostrar el histograma correspondiente, como se explicó en el inciso **c).**

5) ¿Cómo implementaron el histograma?, ¿Qué componente de GUI usaron?

**Respuesta:** se utilizó una clase **Histograma** que extiende de **JPanel**, de modo que se hereden sus comportamientos. El histograma contiene, a su vez, dos JPanel, de los cuales uno está dedicado a las barras y el otro a las etiquetas debajo de cada barra; también tiene una lista de **Barras**, modelizadas como objeto, y un **JLabel** con el total de votos que suman las películas que hayan sido seleccionadas para mostrar.

```
public class Histograma extends JPanel{
    public static final Font FUENTE = new Font( name: "Default", Font.BOLD,  size: 12);
    private final List<Barra> barras = new ArrayList<>();
    private final JPanel panelBarra;
    private final JPanel panelEtiqueta;
    private final JLabel lblTotalVotos;
```

Figura 7: atributos de la clase Histograma

La clase Barra es interna al histograma (privada), y contiene los siguientes atributos:

- **etiqueta (String):** representa el texto que se mostrará debajo de la barra, para este proyecto es la cantidad de estrellas que le corresponde a la barra.
- **valor (int):** es el número (altura) que representa la barra, para este proyecto es la cantidad de votos correspondientes a una cantidad de estrellas.
- **color (Color):** contiene el color asignado para representar la barra.

```
private static class Barra {

    private final String etiqueta;
    private int valor;
    private final Color color;
```

Estas barras, en forma gráfica, se representan mediante íconos personalizados que se setean en componentes **JLabel**, dentro de un método **formalizarHistograma**().

```
private static class ColorIcon implements Icon {

    private final Color color;
    private final int ancho;
    private final int alto;

    public ColorIcon(Color color, int ancho, int alto) {
        this.color = color;
        this.ancho = ancho;
        this.alto = alto;
    }

    public int getAncho() { return ancho; }

    public int getAlto() { return alto; }

    @Override
    public void paintIcon(Component c, Graphics g, int x, int y) { dibujarParalelepipedo(g, x, y); }

    @Override
    public int getIconWidth() { return getAncho(); }

    @Override
    public int getIconHeight() { return getAlto(); }
```

Figura 8: atributos, constructor y métodos públicos de la clase ColorIcon, usada para dibujar las barras

Al panel Histograma se le setea un **BorderLayout** en el constructor, que nos permite ubicar las componentes de forma intuitiva. También se agregan las componentes ya mencionadas y se las configura, como se muestra en el código a continuación:

```
public Histograma() {
    setBorder(new EmptyBorder( top: 10,  left: 10,  bottom: 10,  right: 10));
    setBackground(Colores.OSCURO.color().brighter());
    setLayout(new BorderLayout());

    int gapEntreBarra = 5;
    panelBarra = new JPanel(new GridLayout( rows: 1,  cols: 0, gapEntreBarra,  vgap: 0));
    Border externo = new MatteBorder( top: 3,  left: 3,  bottom: 3,  right: 3, Colores.OSCURO.color().brighter());
    Border interno = new EmptyBorder( top: 10,  left: 10,  bottom: 0,  right: 10);
    Border compuesto = new CompoundBorder(externo, interno);
    panelBarra.setBorder(compuesto);

    panelEtiqueta = new JPanel(new GridLayout( rows: 1,  cols: 0, gapEntreBarra,  vgap: 0));
    panelEtiqueta.setBackground(Colores.OSCURO.color().brighter());
    panelEtiqueta.setBorder(new EmptyBorder( top: 5,  left: 10,  bottom: 0,  right: 10));

    lblTotalVotos = new JLabel();
    lblTotalVotos.setFont(Aplicacion.FUENTE);
    lblTotalVotos.setForeground(Colores.BLANCO.color());

    add(lblTotalVotos, BorderLayout.NORTH);
    add(panelBarra, BorderLayout.CENTER);
    add(panelEtiqueta, BorderLayout.PAGE_END);
}
```

Figura 9: constructor de la clase Histograma

Nótese que a los paneles de barras y etiquetas se les establece un **GridLayout**, con una única fila y un *gap* horizontal, de modo que al agregar componentes éstos aparezcan uno al lado del otro con un espacio de separación determinado por el *gap*.

El histograma permite agregar nuevas barras o cambiar el valor de las mismas:

```
public void agregarColumna(String etiqueta, int valor, Color color) {
    Barra barra = new Barra(etiqueta, valor, color);
    barras.add(barra);
}
```

```
public void cambiarValorBarra(int pos, int valor){
    barras.get(pos-1).setValor(valor);
}
```

Sin embargo, los cambios tendrán efecto una vez que se formalice el histograma:

```
public void formalizarHistograma() {
    panelBarra.removeAll();
    panelEtiqueta.removeAll();

    int suma = 0;

    int max = 0;
    for (Barra barra : barras)
        max = Math.max(max, barra.getValor());


    for (Barra barra : barras) {
        JLabel etiqueta = new JLabel(Integer.toString(barra.getValor()));
        etiqueta.setFont(Aplicacion.FUENTE);

        etiqueta.setHorizontalTextPosition(JLabel.CENTER);
        etiqueta.setHorizontalAlignment(JLabel.CENTER);

        etiqueta.setVerticalTextPosition(JLabel.TOP);
        etiqueta.setVerticalAlignment(JLabel.BOTTOM);

        int alturaBarra;
        int alturaHistograma = 200;
        int anchoBarra = 100;
```

En primer lugar, se limpian las componentes gráficas mediante el método *removeAll()* pertinente a cada **JPanel**, ya que puede haber barras vacías con valor 0, o bien, de una selección anterior.

Luego se busca la barra más alta de las actuales, y a partir de este máximo se setean las alturas de todas las barras por regla de tres.

El **JLabel** local “etiqueta” tendrá el ícono que representa a la barra y como texto el valor que tiene almacenado dicha barra.

```
        suma += barra.getValor();

        if(max <= 0)
            alturaBarra = 0;
        else
            alturaBarra = (barra.getValor() * alturaHistograma) / max ;

        Icon icon = new ColorIcon(barra.getColor(), anchoBarra, alturaBarra);
        etiqueta.setIcon(icon);
        etiqueta.updateUI();

        panelBarra.add(etiqueta);

        JLabel etiquetaBarra = new JLabel(barra.getEtiqueta());
        etiquetaBarra.setFont(FUENTE);
        etiquetaBarra.setForeground(Colores.DORADO.color().brighter());
        etiquetaBarra.setHorizontalAlignment(JLabel.CENTER);
        panelEtiqueta.add(etiquetaBarra);
    }

    lblTotalVotos.setText("Total de votos = " + suma);
}
```

A su vez también se va llevando a cabo la suma de los valores de las barras para calcular la cantidad de votos totales representados en este histograma, visualizado en el **JLabel** superior "lblTotalVotos".

Se agrega cada barra en el panel de barras, y se agregan las etiquetas (cantidad de estrellas) que van debajo de las mismas en el panel de etiquetas. De esta forma existe una coherencia entre barra y etiqueta, alineadas verticalmente, pero en 2 paneles.

Como último paso se muestra la suma antes mencionada en el **JLabel** correspondiente.

El histograma es actualizado en tres ocasiones: cuando finaliza la lectura de los archivos, cuando el usuario selecciona películas de la tabla, y cuando se hace clic en un botón de "Mostrar votos totales", que vuelve a mostrar el histograma "poscarga".

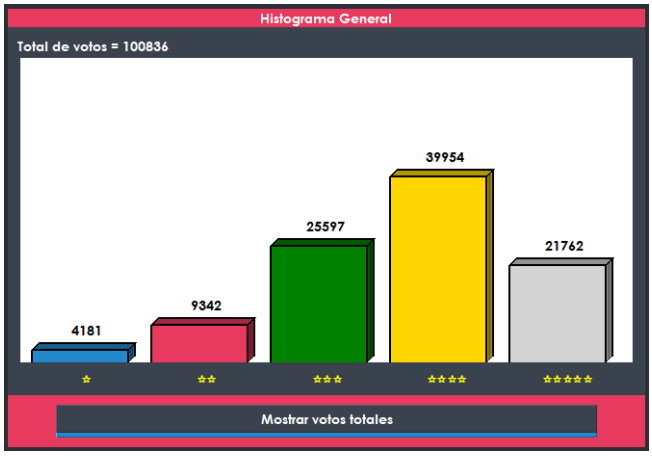



Figura 10: histograma mostrado al finalizar la lectura

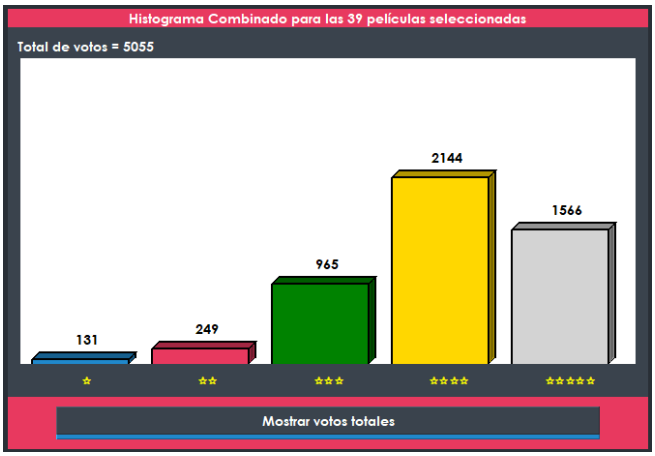



Figura 11: histograma para selección múltiple

Para poder dibujar las barras se invoca al método **dibujarParalelepipedo(Graphics g, int x, int y)** - ver figura 8 - donde se realiza un casteo de **g** a **Graphics2D** para poder luego setear el grosor de las líneas de los bordes. Para dibujar los rectángulos tridimensionales que representan a las barras se usa el método **fillRect(int x, int y, int width, int height)** y se crean polígonos para dibujar los bordes de esos rectángulos:

```
private void dibujarParalelepipedo(Graphics g, int x, int y){
    // Graphics a 2D
    Graphics2D g2 = (Graphics2D) g;

    // Medida de desborde
    int gap = 7;

    // Grosor de linea
    int thick = 2;

    // Dibuja el rectangulo oscuro
    g2.setColor(color.darker());
    g2.fillRect(x, y, ancho, alto);
```

```
    // Porción para darle perspectiva, une los vertices de los rectangulos principales
    Polygon unidorPunta = new Polygon();
    unidorPunta.addPoint(x, y);
    unidorPunta.addPoint( x: x - gap,  y: y + gap);
    unidorPunta.addPoint(x,  y: y + gap);
    g2.fillPolygon(unidorPunta);

    // Dibuja el rectangulo claro
    g2.setColor(color);
    g2.fillRect( x: x - gap,  y: y + gap, ancho, alto);

    // Borde de la cara superior
    Polygon borde1 = new Polygon();
    borde1.addPoint(x, y);
    borde1.addPoint( x: x - gap,  y: y + gap);
    borde1.addPoint( x: x - gap + ancho,  y: y + gap);
    borde1.addPoint( x: x + ancho, y);
// Sirve para ocultar la punta del extremo inferior derecho, para darle perspectiva
Polygon ocultadorPunta = new Polygon();
ocultadorPunta.addPoint( x: x-gap+ancho,  y: y + alto);
ocultadorPunta.addPoint( x: x+ancho,  y: y+alto - gap);
ocultadorPunta.addPoint( x: x+ancho,  y: y+alto);

// Borde para la izquierda del rectangulo claro
Polygon borde2 = new Polygon();
borde2.addPoint( x: x - gap ,  y: y + gap);
borde2.addPoint( x: x - gap,  y: y + alto + gap);

// Borde para la cara derecha
Polygon borde3 = new Polygon();
borde3.addPoint( x: x + ancho - gap ,  y: y + gap );
borde3.addPoint( x: x + ancho, y);
borde3.addPoint( x: x + ancho,  y: y + alto - gap);
borde3.addPoint( x: x + ancho - gap,  y: y + alto);

// Relleno para la cara derecha
Polygon rellenoCaraDerecha = new Polygon();
rellenoCaraDerecha.addPoint( x: x + ancho - gap + thick,  y: y + gap);
rellenoCaraDerecha.addPoint( x: x + ancho - thick,  y: y + thick);
rellenoCaraDerecha.addPoint( x: x + ancho - thick,  y: y + alto - gap - thick);
rellenoCaraDerecha.addPoint( x: x + ancho - gap + thick,  y: y + alto - thick);

g2.setColor(Color.BLACK);
g2.setStroke(new BasicStroke(thick));

// Dibujo el borde de la cara superior
g2.drawPolygon(borde1);

// Dibujo el borde para la izquierda del rectangulo claro
g2.drawPolygon(borde2);
    if(alto > 0) {
        // Dibujo el borde para la cara derecha
        g2.drawPolygon(borde3);

        g2.setColor(color.darker().darker());

        // Dibujo el relleno para la cara derecha
        g2.fillPolygon(rellenoCaraDerecha);
    }

    if(alto > 5) {
        // Recomiendo que el ocultador de punta tenga el mismo color que el background
        g2.setColor(Colores.BLANCO.color());

        // Dibujo el ocultador de punta
        g2.fillPolygon(ocultadorPunta);
    }
}
```

6) Comenten una característica distintiva de su proyecto.

**Respuesta:** nuestro proyecto cuenta con varias características/funciones adicionales a lo solicitado en el trabajo, entre las cuales podemos destacar las siguientes.

- **Posibilidad de actualizar los archivos .csv:** se ubicaron los archivos fuente de forma externa al JAR, de modo que los mismos puedan ser modificados o reemplazados por el usuario en caso de necesitar incorporar más películas o votos, por ejemplo. La única limitación es que se debe respetar el mismo formato para no afectar el funcionamiento de nuestro programa.

- **Posibilidad de examinar los archivos .csv:** mediante la implementación del componente **JFileChooser**, en caso de que el programa no pueda localizar los archivos fuente se le pedirá al usuario que los examine (explicado en punto 2), sin obligarlo a que mueva los archivos al directorio donde se encuentra el JAR. Si el usuario los localiza, el programa continuará con la lectura y procesamiento de los archivos de forma normal, como si ningún error hubiese ocurrido.

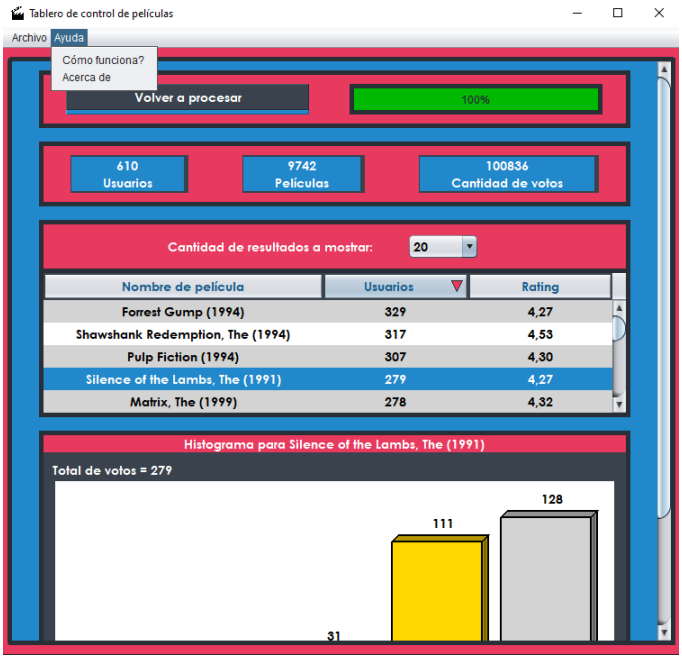
- **Posibilidad de recargar los archivos .csv:** el botón de "Procesar Datos" puede ser pulsado antes y después de una lectura de los archivos, de modo que el usuario puede modificar o reemplazar los archivos fuente y volver a procesarlos sin reiniciar el programa. El texto del botón es cambiado a "Volver a procesar".

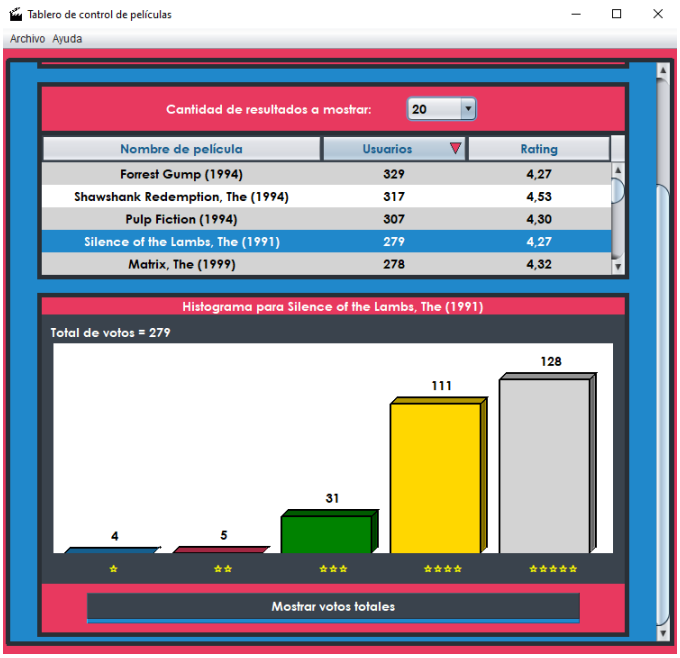

- **Diferentes tipos de histograma:** se puede mostrar al usuario un histograma **general** que representa los votos sumados de todas las películas cargadas; un histograma **individual** para los votos de una película específica que el usuario haya seleccionado en la tabla; o bien, un histograma **combinado** con los votos sumados de varias películas que hayan sido seleccionadas desde la tabla. Esta última opción no era requerida en el trabajo, pero a nosotros nos pareció una característica interesante para agregar en caso de selección múltiple de filas. También existe un botón "Mostrar votos totales" debajo del histograma en caso de que el usuario desee restaurar el gráfico general mostrado por defecto.

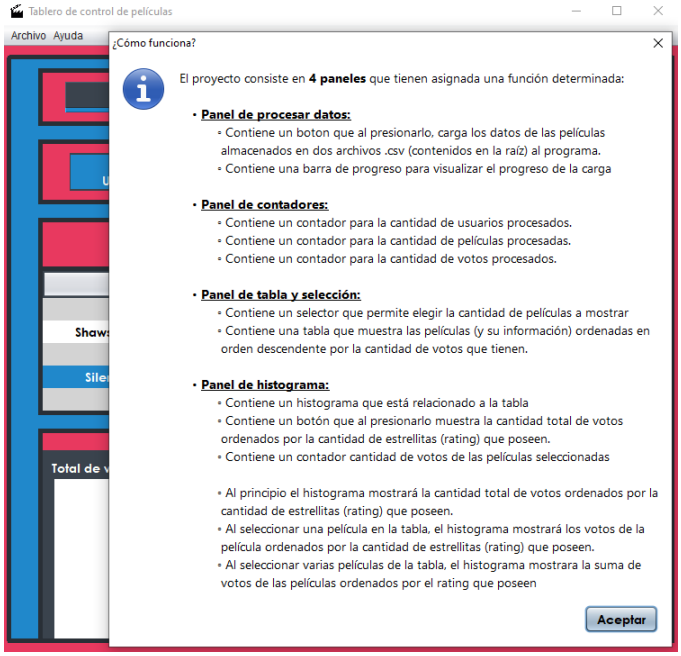
- **Implementación de JMenuBar:** se agregó este componente, desde el cual se puede acceder a los ítems "Salir", "¿Cómo funciona?" y "Acerca de". Estos dos últimos muestran diálogos de tipo informativo (ver figuras en pág. siguiente)

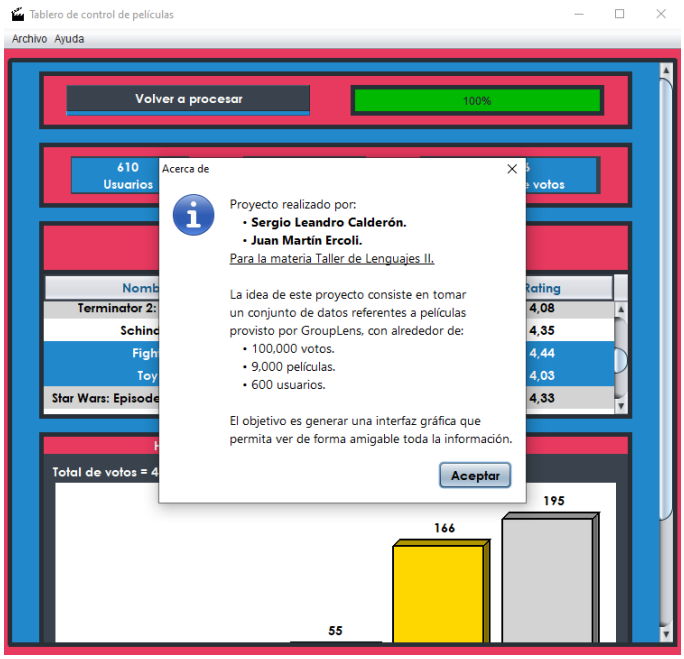
- **Ícono de aplicación:** debido a que los archivos están ubicados fuera del JAR, se decidió que el Frame del programa tenga un ícono de "película", para hacer uso del método getClass().getResourceAsStream() y la clase **BufferedImage**.

- **Estilo Nimbus y colores personalizados:** se le da a la interfaz gráfica una apariencia más amigable mediante *Look And Feel*. Sin embargo, se tuvo que crear las flechas de sort porque el estilo las invisibilizaba en los encabezados.




```java
private void setAppIcon(){
    try {
        InputStream inputStream = getClass().
                getResourceAsStream( name: "/tallerlenguajes2/tp3/peliculas/res/iconoPelicula.png");

        BufferedImage bufferedImage = ImageIO.read(inputStream);
        super.setIconImage(bufferedImage);
    } catch (IOException e){
        System.err.println("Ocurrió un error durante la lectura del ícono");
        e.printStackTrace();
    } catch (IllegalArgumentException | NullPointerException e){
        System.err.println("El icono no se encuentra en el directorio especificado");
        e.printStackTrace();
    }
}
```