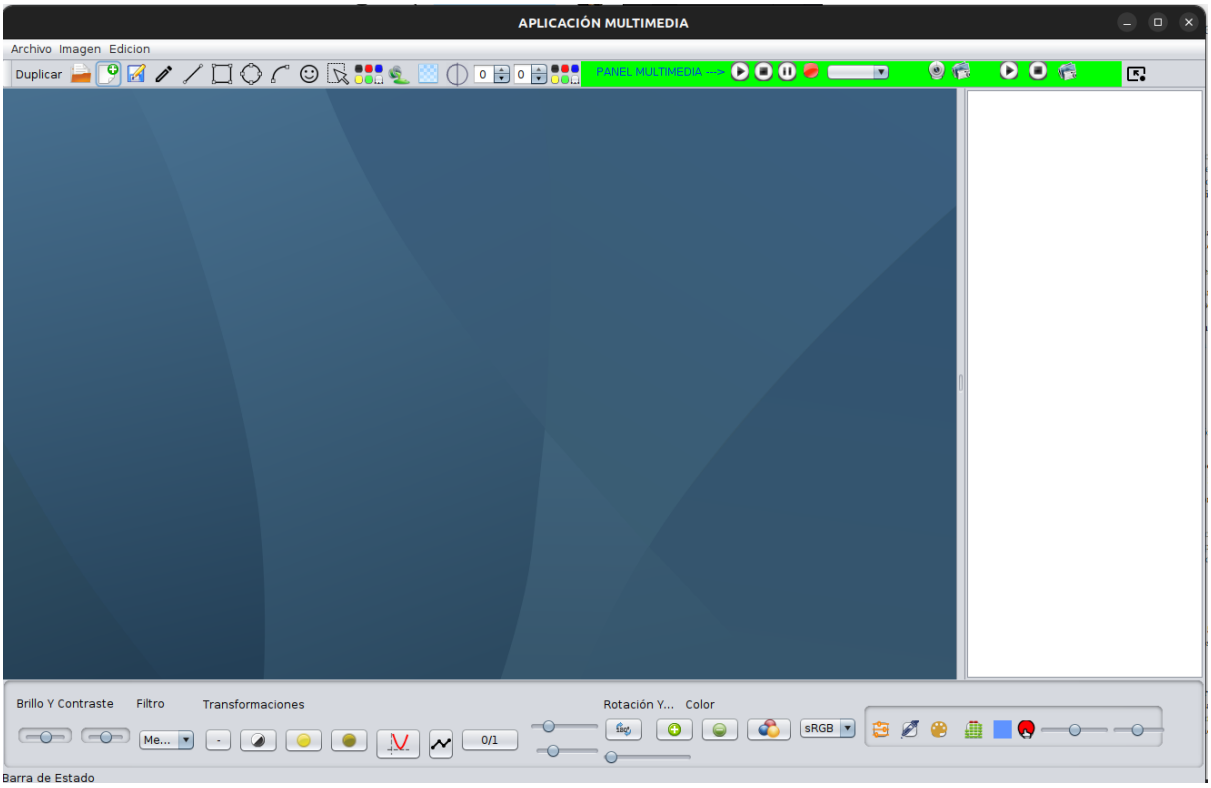
Raúl Ramírez Abril
Práctica de evaluación final- Sistemas Multimedia

# Sistemas Multimedia.
# Universidad de Granada.
# Curso 2022/2023.

## Práctica final de evaluación: Aplicación multimedia funcional.




**Proyecto realizado por: Raúl Ramírez Abril**

# Introducción:

El objetivo de esta práctica de evaluación es la realización de una aplicación multimedia que nos permita la gestión de varios tipos de medios. Dividiremos las estructura de la aplicación en cinco bloques principales: interfaz, gráficos, imágenes, sonido y video. Para cada tipo de medio, se abordarán funcionalidades específicas, proporcionándonos una aplicación multimedia completa y funcional. Cabe destacar que esta es una práctica de evaluación final de la asignatura, por lo que abordaremos todos los conceptos dados desde las práctica 1-14 y las explicaciones en las clases teóricas.

Esta práctica se abordará usando el lenguaje de programación JAVA. Para las interfaces se usará la swing y se realizará en el entorno de desarrollo Apache Netbeans 18 debido a su integración con la swing de java. Además, para distintas funcionalidades será necesario el uso de las librerías, algunas proporcionadas en la asignatura y otras externas a la asignatura pero relacionadas con ella.

# -2. Requisitos:

Comentaremos brevemente por cada bloque los requisitos indispensables y las soluciones planteadas.

**2.1 Interfaz:**

1. **Diseño claro y sencillo**: La interfaz debe ser fácil de entender a primera vista. Las funciones de cada componente de la interfaz deben ser obvias para el usuario:

   Se han implementado una serie de paneles bastante intuitivos en los que claramente se distingue la parte gráfica, la de multimedia y la de edición de fotografías.

2. **Navegación intuitiva**: Los usuarios deben ser capaces de moverse a través de la aplicación fácilmente, encontrar la información o función que buscan rápidamente. Para lidiar con esto, se ha incluido etiquetas desplegables en cada botón.

3. **Consistencia en el diseño**: Todos los elementos visuales de la interfaz deben seguir una misma línea de diseño. Esto incluye los colores, formas, tamaños, etc. También se refiere a la coherencia en la

ubicación y comportamiento de los elementos interactivos. Cada botón se ha hecho siguiendo el mismo patrón, en una zona concreta de la interfaz y usando un icono para que sea más representativo.

4. **Controles multimedia de usuario**: Debe haber controles para reproducir, pausar, y detener contenido. En este caso se ha distinguido entre los controles para la sección de audio y de video. A nivel de código la aplicación es totalmente capaz de manejar distintos tipos de archivos multimedia en función a su extensión, pero se ha preferido la distinción clara de botones a modo de como implementación futura tener capacidad de reproducir audios y videos a la vez distinguiendo los canales de sonido. Además, se ha implementado la funcionalidad de poder realizar una snapshot tanto de nuestra webcam como de cualquier video que reproduzcamos para así poder mostrarla inmediatamente en el escritorio de nuestra aplicación.

5. **Abrir, guardar, nuevo:** El usuario debe ser capaz de poder manejar de una manera intuitiva estas tres funciones indispensables. Se ha implementado un panel que los gestione. En cuanto al botón de abrir, el usuario podrá abrir cualquier tipo de archivo permitido indistintamente de su extensión, la aplicación interpretará que tipo de archivo es. Para nuevo, se creará un nuevo **lienzo.** La opción guardar nos dejará guardar nuestro archivo en el directorio que queramos de nuestro pc personal.

   El botón nuevo nos mostrará un diálogo en el que podremos elegir el tamaño de nuestro lienzo.

6. **Lista lateral:** La aplicación proporcionará una lista lateral en el borde derecho en el que podemos ver los trazos realizados en nuestro lienzo.

   La solución ha sido añadir un manejador de evento que se encargará de controlar cuando se añaden figuras para así poder añadirlos a la lista. En todo momento sabrá que figura de la lista es exactamente la que hay dibujada. Además, es desde esta lista desde donde podremos seleccionar nuestras figuras para el volcado (explicado más adelante).

**2.2 Gráficos:**

1. **Selector de herramientas:** En una barra superior, se deben presentar los íconos de las herramientas disponibles: trazo libre, línea, cuadrado, elipse, curva y smile. Al seleccionar cada una de estas herramientas, se deben activar los ajustes correspondientes para esa figura en particular.

2. **Propiedades de las figuras:** Una vez seleccionada la herramienta, debes proporcionar controles para ajustar sus propiedades. Se usarán usar spinners la discontinuidad y la transparencia. Para el color y el color de relleno, puedes usar selectores de color. Habrá una casilla de verificación para habilitar o deshabilitar el antialiasing, el relleno o la transparencia. La solución es implementada en la biblioteca SM.RRA.BIBLIOTECA, que se explicará más adelante.
3. **Dibujar figuras:** El usuario será capaz de dibujar los tipos de figuras explicados en los puntos 1 y 2.
4. **Mover y Volcar las figuras:** Se permitirá que los usuarios arrastren y suelten las figuras en la posición deseada en el lienzo. Para volcar las figuras y que queden como parte de la imagen del lienzo, estará el botón “volcar” que realice esa acción. La solución es implementada en la biblioteca SM.RRA.BIBLIOTECA, que se explicará más adelante.
5. **Barra de Estado:** En la parte inferior habrá una barra de estado que muestre información útil, como la herramienta actualmente seleccionada.

**2.3 Imágenes.**

En esta sección se establecerán, aparte de los requisitos, los diseños de muchos de estos, ya que la amplia mayoría no han requerido de usar la biblioteca propia del proyecto, sino que se han hecho con el uso de operadores vistos en la asignatura. Las operaciones que hayan requerido de una clase propia sí serán especificadas con más detenimiento en la sección de diseño.

1. **Duplicar imagen**: Una característica esencial es otorgar al usuario la capacidad de crear un duplicado de su lienzo/imagen. Por ejemplo, esto podría permitirle replicar su imagen antes de ejecutar una operación adicional sobre uno de los duplicados, con el objetivo de preservar la versión original. Para hacer esto, inicialmente adquirimos

el modelo de color y el raster del lienzo/imagen en uso, y luego comprobamos si tiene un canal alfa. Finalmente clonamos esa imagen y la mostramos en una nueva ventana.

2. **Ajustar el brillo y el contraste utilizando deslizadores:** Tendremos tres eventos. El evento inicial se refiere a cuando el deslizador obtiene el foco (método focusGained()). Lo que hacemos es generar una copia de la imagen, de la misma manera que lo hicimos anteriormente, pero en lugar de vincularla a una ventana y mostrarla, la guardamos en una variable de tipo BufferedImage llamada img. La segunda operación común se da en el caso contrario, cuando el deslizador pierde el foco. Llamamos al método focusLost(), el cual regresa el botón del deslizador a su valor medio (0 en el caso específico del brillo).

   Nuestra operación consistirá en seleccionar un valor entre 0 y 100 del slider (para representar el brillo máximo y mínimo) y crear una operación RescaleOP donde el valor de la variable a será 1 (porque el contraste no influye), y b será el valor obtenido del deslizador.

   Para el contraste, se puede seguir un proceso similar al del brillo, pero en la operación RescaleOP, se modificaría la variable 'a' con el valor obtenido del deslizador de contraste, y 'b' permanecería en 0.

3. **Selección de Filtros**: Tendremos un selector de filtros que nos permitirá aplicar unos filtros predeterminados a nuestras imágenes en el lienzo.
   1. **Media**: Suaviza la imagen reduciendo el ruido y los detalles al reemplazar el valor de un píxel por el valor medio de los píxeles vecinos.
   2. **Binomial**: Similar al filtro de la media, pero utiliza una matriz de pesos que sigue una distribución binomial para obtener un efecto de suavizado más refinado.
   3. **Enfoque**: Incrementa la nitidez de la imagen, acentuando los bordes y los detalles.
   4. **Relieve**: Destaca los bordes y las texturas de la imagen para dar la ilusión de profundidad o relieve.
   5. **Laplaciano**: Detecta los bordes de la imagen resaltando las zonas de cambio brusco de intensidad.
   6. **Horizontal 5x1, 7x1 y 10x1**: Aplica un filtro de suavizado o detección de bordes de forma horizontal con las dimensiones especificadas.

    7. **Medio 5x5 y 7x7**: Aplica un filtro de suavizado que promedia los valores de los píxeles en una vecindad de las dimensiones especificadas.
4. **Contraste Normal:** Se aplicará un efecto de ajuste de contraste usando una LookupTable predefinida. Este efecto se aplicará directamente a la imagen original. Una vez realizada esta acción, la imagen se actualizará automáticamente en la ventana interna.
5. **Iluminación**: Se aplicará un efecto de corrección gamma utilizando una LookupTable generada con el método LookupTableProducer.createLookupTable. El efecto se aplicará directamente a la imagen original y, posteriormente, se actualizará la vista en la ventana interna.

6. **Oscurecimiento**: Se aplicará un efecto de potenciación usando una LookupTable generada con el método LookupTableProducer.createLookupTable. Esta modificación se aplicará directamente sobre la imagen original y, una vez aplicada, la vista de la imagen se actualizará en la ventana interna.
7. **Operador Propio: Umbralización binaria:** La umbralización binaria es una técnica de procesamiento de imágenes que convierte una imagen en escala de grises a una imagen binaria (negro y blanco). Esto se logra definiendo un umbral de intensidad: los píxeles con intensidad por debajo del umbral se convierten en negros y los que están por encima se vuelven blancos. Explicado con más detalle en la sección diseño.
8. **Operador de función cuadrática:** Función cuadrática $f(x) = 1/100(x-m)^2$ siendo $0 <= m <= 255$ Esta función recibe un parámetro m y crea y devuelve una lookup table utilizada para aplicar una transformación cuadrática a los valores de los píxeles. Utiliza una fórmula cuadrática para calcular los nuevos valores de los píxeles basados en el parámetro m y los valores originales de los píxeles.
9. **Operador spline lineal con punto de inflexión:** Dependeremos de dos sliders, que tendrán valores oscilantes entre 0 y 255. Los llamaremos a y b. Haremos un ajuste lineal de contraste con inflexión en la imagen, es decir, aplicaremos un ajuste en el brillo y contraste de la imagen dividiendo el rango de niveles de grises en dos segmentos, cada uno con su propia pendiente.

   Los dos segmentos se separan en un punto de inflexión 'a', y 'b' representa el valor de intensidad del píxel en el punto de inflexión. Para

los valores de píxeles por debajo de 'a', se realiza un ajuste de brillo. Para los valores de píxeles por encima de 'a', se realiza un ajuste de contraste.

Hay un parámetro 'm' que es la pendiente de la línea después del punto de inflexión 'a'. Si 'a' no está en el extremo máximo del rango (255), entonces la pendiente 'm' se calcula para ajustar la intensidad del píxel hacia el valor máximo (255).

Por lo tanto, esta función devuelve un objeto `LookupOp` que se puede aplicar a una imagen para ajustar su contraste y brillo de manera lineal con un punto de inflexión.

10. **Rotación, zoom in y zoom out:** Se ha hecho uso de la clase AffineTransform para poder realizar estas transformaciones. Podemos hacer zoom hacia adentro y hacia afuera de la foto gracias a getScaleInstance. Para la rotación tenemos dos opciones, usar un slider para giro libre o rotar 180 grados pulsando el botón. La solución ha sido usar getScaleInstance de AffineTransform para estas operaciones. Destacar la importancia del método filter que es el encargado de aplicar la transformación en la imagen destino (la que sobreescribirá la por defecto).
11. **Extracción de bandas**: Este requisito consiste en proporcionar al usuario la opción de "Muestra Banda". Se trata de iterar a través de las bandas de la imagen y se utiliza el método "getImageBand" para obtener una nueva imagen que representa cada banda. Por cada banda la mostramos en una imagen distinta.

    El método getImageBand toma como entrada una imagen (BufferedImage) y un número de banda (int) y devuelve una nueva imagen que contiene solo la banda específica extraída de la imagen original. La nueva imagen se representa en el espacio de color GRAY.

    Para lograr esto, el método crea un modelo de color basado en el espacio de color GRAY y un modelo de color de componente (ComponentColorModel). Luego, utiliza el raster de la imagen original para crear un nuevo raster que contiene únicamente la banda especificada. Finalmente, se crea una nueva imagen (BufferedImage)

utilizando el modelo de color de componente y el raster de la banda extraída, y se devuelve como resultado.

12. **Espacios de color:** El usuario podrá elegir el espacio de color entre RGB, PYCC y GRAY. Se usa la biblioteca ColorSpace de java y el operador ColorConvertOp para aplicar el cambio de espacio de color.
13. **Combinar:** Este requisito tiene como objetivo la aplicación de combinación de bandas en una imagen. Para esto, se hace uso del operador BandCombineOp.
14. **Tintado:** Se trata de poder aplicar un efecto de tinte a nuestra imagen en función al color que tengamos marcado en nuestro lienzo. Se hace uso del operador TintOp que se encarga de filtrar la imagen y aplicar el tintado en ella misma.
15. **Sepia**: Funcionamiento similar al tintado. Esta vez se hace uso del operador SeipaOp para aplicar el efecto de sepia.
16. **Ecualización:** Consiste en aplicar un efecto de ecualización de histograma. El proceso es similar a los operadores anteriores, pero esta vez se usa el operador EqualizationOp que es quien se encarga de aplicar el filtro.
17. **Operador propio: Recalca Azules:** Operación similar a las anteriores. Se usa el operador AzulOp (implementado por mi y explicado más adelante).
18. **Recalca Rojos:** Similar al operador anterior. Recalca los rojos de la mano del operador RojoOp (implementación propia explicada más adelante).
19. **Variar tono de color:** Contaremos con un slider que permitirá variar el tono de los colores. Se usará el operador TonoOp de la biblioteca RRA (Explicada más adelante).
20. **Posterizar**: Similar al anterior. Uso del operador PosterizarOp implementado en mi biblioteca.

### 2.4: Sonido.

1. **Almacenar los sonidos:** La implementación contará con un combobox donde se cargarán los sonidos que abramos.
2. **Reproducir, pausar y parar los sonidos:** Se usará la biblioteca SMClipPlayer para implementar estas funciones.

3. **Grabar audios:** Tendremos un botón que nos permitirá realizar nuestras grabaciones. Usaremos SMSoundRecorder.

**2.5 Video.**

1. **Uso de webcam:** Nuestra aplicación será capaz de abrir nuestra webcam e incluso de tomar snapshots de esta. Se hará uso de VentanaInternaCamara.
2. **Reproducir, pausar y parar videos:** Se usarán las VentanasIternaVideo para poder realizar estos requerimientos.
3. **Snapshot de video**: La aplicación será capaz de tomar capturas de instantes del video. Se usará el método getImage que a su vez usa el método getSnapshot que nos devuelve una BufferedImage.

## -Diseño:

En esta sección abordaremos con detalle la implementación de nuestra aplicación. Si bien es cierto que los apartados de imágenes, sonido y audio han sido tratados con detalle en la sección de requisitos, en nigún momento hemos entrado en detalle con la librería propia del proyecto: SM.RRA.BIBLIOTECA.

En primer la estructura de nuestro proyecto y la asociación que tienen las clases entre ellas y después procederemos a explicar la estructura de nuestra librería, haciendo hincapié en el paquete gráficos.

AVISO AL LECTOR: Para poder comprender los tecnicismos que se van a explicar a continuación, es de vital importancia que conozcamos conceptos de programación y diseño orientado a objetos en los que se sustenta java: Jerarquía de clases, herencia, tipo de dato estático y dinámico. Clases abstractas e interfaces. Polimorfismo y ligadura dinámica.
Consulte el manual oficial de java para más información.

1. **Paquete principal:**
   **Ventana principal:** Es la parte central de la aplicación, el lugar donde se estructura toda la interfaz de usuario. Aquí es donde se encuentran todos los elementos y métodos relacionados con las imágenes que fueron explicados en la sección anterior.
   A destacar tenemos el método getFileExtension que es el que me ha permitido distinguir las extensiones de los archivos para así poder interpretar el tipo de archivo a la hora de abrirlo.
   En la parte central de la ventana tendremos un panel-escritorio que es donde albergamos nuestras ventanas internas.
   Aquí tendremos una gran variedad de botones iconizados con todas las funcionalidades establecidas en los requisitos. Cabe a destacar que si la funcionalidad está relacionada con el apartado gráfico, lo que hará el código es establecer la funcionalidad en el lienzo, es decir, que invocamos a **getLienzoSeleccionado()** y estableceremos la funcionalidad que corresponda.
   Añadir que para la funcionalidad de audio implementamos un manejador de audio dentro de esta clase.

   **Ventana Interna de Imagen:** Esta ventana interna es fundamental para el apartado de gráficos, ya que es la que albergará el **Lienzo2D.** Es fundamental entender el concepto de cómo es este concepto, ya que cuando nosotros queremos realizar cualquier acción (editar una imagen, pintar etc) será este el lienzo donde nosotros hagamos esa acción. Accederemos a el a través del método **getLienzoSeleccionado()** ya que es el "puente" entre la ventana principal al "lienzo" y sólo ahí es donde podremos realizar las acciones que deseemos, la ventana interna de imagen simplemente se encarga de alojar el lienzo.
   Por último, esta ventana interna cuenta con un manejador para el lienzo que será imprescindible para gestionar los eventos (la lista lateral).

   **Ventana interna de Video:** Contamos con dos atributos, uno de tipo File que se encargará de alojar el archivo de video seleccionado y otro de tipo EmbeddedMediaPlayer que se encargará de reproducirlo. Imprescindible tener VLC MEDIA PLAYER instalado en el sistema.
   Contamos con los métodos **play, stop, internalFrameClosing y getImage** (este último sirve para tomar snapshots del video).

**Ventana interna de cámara:** Encargado de hacer a modo de webcam en la ventana interna. Contamos con el atributo de tipo Webcam que se encarga de gestionar lo relacionado con la apertura de la webcam. Además, se ha implementado al igual que en la ventana anterior el método **getImage()** que devuelve una ventana interna de imagen con una captura de pantalla de la webcam en el instante que se hizo click.

## 2. Biblioteca SM.RRA.BIBLIOTECA

Esta biblioteca ha sido desarrollada a lo largo del curso y culminada en esta práctica final. Es el motor del proyecto y contiene la amplia mayoría de las funcionalidades relacionadas con el proyecto.

### Paquete de gráficos:

El problema principal que teníamos a lo largo del curso era que todo lo relacionado a gráficos era gestionado por el Lienzo, por lo que todo lo que pintábamos tenía las mismas propiedades.
Para hacerle frente a este problema se ha implementado la siguiente jerarquía de clases:

Superclase **abstracta** Forma:
Esta es una clase abstracta, por lo que no podrá ser instanciada pero si nos servirá a modo de reutilizar todos los atributos y métodos comunes, ya que todas las clases del paquete **heredarán de ella,** será el tipo estático que tendremos en el lienzo y que instanciamos (tipo dinámico) en nuestro lienzo.
La clase Forma tiene los siguientes atributos:

-**relleno**: un booleano que indica si la forma debe tener relleno.
-**mover**: un booleano que indica si la operación consiste en dibujar una forma o moverla.
-**transparente**: un booleano que indica si se debe aplicar un filtro de transparencia.
-**liso**: un booleano que indica si se debe aplicar un filtro de antialiasing.
-**grosor**: un entero que representa el grosor al dibujar.
-**discontinuidad**: un entero que representa la discontinuidad en el trazo.

Además, la clase tiene los siguientes atributos relacionados con el dibujo:

**color**: el color utilizado para dibujar la forma.
**colorRelleno**: el color utilizado para el relleno de la forma.
**stroke**: el trazo utilizado para dibujar la forma.
**alphaComposite**: el objeto utilizado para controlar la transparencia.

Esta clase proporciona métodos de acceso (getters) y de modificación (setters) para todos los atributos.

La clase también contiene **métodos abstractos que deben ser implementados por las clases derivadas**:

-**figura()**: devuelve el objeto Shape que representa la forma. Este método aunque es muy simple será uno de los métodos más usados ya que nos facilitará el diseño del lienzo en muchos aspectos.
**-pintar(Graphics g)**: Se utiliza para pintar la forma en un objeto Graphics. Aplica las propiedades de dibujo configuradas, como color, transparencia, suavidad, grosor y discontinuidad, y luego dibuja el trazo.
-**contains(Point2D p)**: verifica si un punto específico está contenido dentro de la forma.

**Se han implementado estas clases que heredarán de Forma:**

## Clase Linea:

Esta es una subclase de la clase Forma que representa una línea en un proyecto. Aquí tienes una explicación de la clase:

La clase Linea cuenta con el atributo Line2D linea, que será el que pintaremos.

**¿Y dónde están los demás atributos?** Pues como estamos heredando de la clase Forma los tenemos heredados y así nos evitamos repetir lineas de código.

**Muy importante entender esto**: Nuestra clase línea no extiende de Line2D, no es una línea. Tiene un atributo de tipo Line2D que será el que usaremos para pintar en nuestro lienzo.

La clase Linea proporciona varios constructores para crear instancias de la línea con diferentes configuraciones de color, transparencia, suavidad, grosor y discontinuidad. Estos constructores llaman a los métodos correspondientes de la clase base Forma para configurar las propiedades.

La clase Linea también implementa los métodos pintar(Graphics g), toString(), isNear(Point2D p) y contains(Point2D p).

El método pintar(Graphics g) se utiliza para dibujar la línea en un objeto Graphics utilizando las propiedades configuradas en la clase. Aplica el grosor, discontinuidad, color, transparencia y suavidad según corresponda.

El método toString() devuelve una representación en cadena de la línea, en este caso, simplemente devuelve la palabra "Linea".

El método isNear(Point2D p) verifica si un punto específico está cerca de la línea. Comprueba la distancia entre el punto y la línea, y devuelve true si la distancia es menor o igual a 2.0.

El método contains(Point2D p) verifica si un punto específico está contenido dentro de la línea. En este caso, simplemente llama al método isNear(p).

Además, la clase Linea implementa el método abstracto figura(), que devuelve el objeto Line2D que representa la línea.

**IMPORTANTE**
Una vez entendido esto, si tenemos un mínimo de conocimiento nos preguntaremos, **¿Pero cómo hacemos para que nuestros atributos seleccionados en la ventana principal sean los que tenga nuestra línea?** La respuesta está en el Lienzo, clase que anteriormente expliqué de forma simple y que luego extenderé. La línea será pintada en el lienzo, por lo que lo que tenemos que hacer es que cuando

nuestra en nuestra ventana principal seleccionemos un atributo, establezcamos este en el lienzo, ya ya el lienzo (como veremos más adelante) será el encargado de gestionar estos atributos. Es de vital importancia entender esto, sino no tiene sentido seguir leyendo.

Además de la línea, se han implementado otras demás clases para dibujar otras formas, en estas formas no se hará tanto incapié ya que la idea es la misma pero para cada tipo de Shape (superclase) de Line2D.

-Clase Curva: Igual que la línea, atributo de tipo QuadCurve2D lo único a tener en cuenta es la gestión de eventos que deberemos tener en cuenta que hay un punto de control para establecer la curvatura.

-Dibujo Libre: Se usa un atributo de tipo GeneralPath.

-Elipse: Similar a las anteriores, a tener en cuenta que esta vez ponemos en uso el atributo color relleno, ya que el método paint distinguirá el color del borde y el relleno. Para esto graphics2d tiene el método **fill**.

-Rectángulo: Similar a la elipse pero con Rectangle2D.

Smile: El más “distinto” ya que es de tipo Area y lo que haremos erá que en el constructor crearemos los ojos y la boca que tras hacer **.substract** en el area representarán un smile a la hora de pintar.

**Paquete de IU:**

**Clase Lienzo2D:** Por fin llegamos al corazón de la parte de gráficos del proyecto, el lienzo. A modo de recordatorio, este es el famoso lienzo del que se ha hablado varias veces en la práctica, el que es contenido en la ventana interna de imagen, el que contiene los atributos que marcaremos desde nuestra ventana principal en su interfaz de usuario y el que contendrá las formas. Además, gestionará los eventos de ratón pressed, dragged y released.

En primer lugar, tenemos un atributo de tipo **Forma 'figura' y un ArrayList de tipo Forma,** que será el que contendrá todas las figuras que dibujemos. Además, antes hablábamos de que la ventana interna de imagen contendría las imágenes que habíamos pero realmente en ningún momento se habló acerca de esto más en profundidad, ya que en realidad es el lienzo quien lo contiene.
Como mencionamos anteriormente, nuestras figuras que pintemos cogerán los atributos del lienzo, es por esto que el lienzo cuenta con todos estos atributos con sus getters y setters correspondientes declarados:

```
boolean relleno;
boolean mover;
boolean transparente;
boolean liso;
int grosor = 1;
int discontinuidad = 0;
tipos tipo = tipos.LINEA;  //Tipo de forma por defecto es linea
Color color = Color.BLACK;
Color colorRelleno = Color.BLACK;
BufferedImage imagen;
Stroke stroke;
AlphaComposite alphaComposite;
```

Así cuando en nuestra ventana principal seleccionamos un color por ejemplo o subamos el spiner de valor lo que haremos será:

```
this.getLienzoSeleccionado.set/*lo que seteemos*/;
```

Así accederemos a nuestra clase lienzo y marcaremos un atributo o un tipo de trazo.

Además, contamos con una variedad de atributos auxiliares declarados de tipo punto, double o booleano que nos servirán para poder mover figuras de una forma más sencilla sin tener que implementar métodos nuevos.

Métodos relacionados con las figuras.
**getFiguraSeleccionada(Point2D p):** este método será muy interesante al querer mover una figura en el lienzo manejando los eventos del ratón. Básicamente dado un punto (el pressed del ratón) nos devuelve la figura más cercana a donde hagamos clic si es que la hay. Hace uso del método **contains** que se implementó en el apartado de gráficos para saber si estamos conteniendo una figura o la tenemos cerca, en función del tipo de figura obviamente.
**vuelca(int numFigura)** : Recordemos que en nuestra ventana principal teníamos la posiblidad de volcar figuras en el lienzo, pues este es el método que lo hace realidad.
En el action performed del botón volcar guardamos un array con los índices de las figuras seleccionadas en la lista lateral para volcar y lo que hacíamos era **iterar sobre él desde el final** para no así aplastar índices y por cada iteración llamábamos a este método. Pues el método vuelca se encarga de, teniendo en cuenta todos los atributos que tiene la figura, volcarla sobre la imagen del lienzo para así hacerla permanente y eliminarla del vector de formas.

Apartado de imagen:
-Tenemos un atributo de tipo BufferedImage imagen.
-Método setImage: Se encargará de setear la imagen seleccionada en la imagen principal y la establecerá en el lienzo.
-Método getImagen(boolean pintaVector): Es el encargado de “tomar” la imagen del lienzo con las figuras dibujadas si pasamos true como parámetro, ya que usa Graphics2D para poder hacer esto una realidad.

**Método paint:**
Este método se encarga de pintar las formas que tenemos usando Graphics2D. Lo que cambia de esta implementación respecto a la del curso, es que la responsabilidad de pintar figuras de la derivamos al método pintar que tiene cada objeto de tipo Forma, porque con recorrer el vector de Formas

entero y invocar a pintar(g2d) será este el que se encargue de dibujar las figuras con sus atributos particulares.

Por último, tenemos una de las partes más importantes de la clase Lienzo, que es la gestión de eventos de ratón.
**Dividiremos en dos bloques, mover figuras y pintar figuras.**

**Para pintar figuras:**
Debemos saber que tenemos un atributo enumerado llamado tipo que es el que será establecido desde la ventana principal cuando seleccionemos una opción de dibujo, tipos.ELIPSE, tipos.LINEA etc.
En el **pressed** lo que haremos es, en función del tipo, instanciar nuestro atributo de tipo forma 'figura', esto es, hacerlo una linea, una elipse el tipo que sea, pasando los atributos del lienzo como parámetro. Aquí es cuando añadimos al vector de formas nuestra nueva instancia. Con el dragged es cuando veremos pintar nuestra figura en el lienzo. Gracias a la capacidad de hacer casting de nuestra forma al tipo de figura que instanciamos, llamaremos a su método figura() o bien a su get del atributo de tipo Shape (la Line2D para el caso de la línea) y así podremos usar los métodos ya definidos en la superclase **Shape**, que nos facilitará el poder dibujar la figura y a la vez nos ahorra definir métodos propios de tanto para definir como para mover.

**Para moverlas:**
Tendremos que tener el atributo mover establecido a true.
Para el pressed simplemente llamaremos al getFiguraSeleccionada y si hay figura en el dragged repetiremos el proceso que hicimos en el pressed pero esta vez usaremos los métodos como setLine, setRect o setFrameFromDiagonal que nos proporciona Shape y nos auxiliaremos en las variables auxiliares que necesitemos para hacer el movimiento posible.

**Clase LienzoListener:** Interfaz que deben cumplir los manejadores de eventos
**Clase LienzoAdapter:** Adapter asociado a LienzoListener.
**Clase LienzoEvent:** Clase que representa los eventos lanzdos en la clase Lienzo2D.
**Clase MiManejadorLienzo**: Hereda de LienzoAdapter y su único método se shappeAdded que sirve para gestionar cuando añadimos una figura.

**Paquete de Imágenes.**
Por último llegamos al paquete de imágenes. Estas clases han sido usadas para la edición de imágenes en el proyecto. Cada una tiene una funcionalidad concreta y se usan en un botón distinto.

**AzulOp:**
Umbral recomendado: 20
Operador de diseño propio que recalca el color azul a partir de un umbral preestablecido. Tiene un **constructor parametrizado con el umbral** y el método filter que es el que se encarga de recalcar el color azul.
El método **filter(BufferedImage src, BufferedImage dest)** se utiliza para aplicar la operación de la imagen. Aquí, src es la imagen de entrada y dest es la imagen de salida. Si dest es null, se crea una nueva imagen compatible con src. Luego, se recorre cada píxel de la imagen de entrada y se recogen los componentes de color de cada píxel. Sumamos la intensidad del azul y restamos la de los otros colores. Si el valor que obtuvimos es menor que el umbral, se hace una media entera de los tres componentes del color para cada componente del píxel en la imagen de destino. Si no, el píxel se copia tal cual a la imagen de destino. Siempre devolvemos la imagen destino pase lo que pase.

**RojoOp:**
Similar al operador anterior pero esta vez usando la intensidad del color rojo en vez de la del azul.
Umbral recomendado: 30.

**PosterizarOp:**
Operador que se encarga de posterizar una imagen con un nivel de posterizado.
El método filter hace una técnica de procesamiento de imágenes que reduce la cantidad colores de la imagen.

Se recorren todos los píxeles de la imagen y ajusta cada banda para cada píxel. El parámetro niveles determina el número de colores disponibles después de la posterización. El valor k = 256.0f / niveles es el intervalo de intensidad para cada nivel de color.

Para cada grupo de color de cada píxel, se obtiene la intensidad del color y se reduce a uno de los niveles especificados. Esto se hace dividiendo sample

por k, restando el número entero (que reduce la potencia a uno de los valores absolutos) y luego multiplicando  por k nuevamente. El resultado es una aproximación de la intensidad original que está limitada a uno de los niveles posibles, y ya cuando hemos hecho eso el valor modificado se pone como el nuevo tamaño del grupo de colores en el píxel correspondiente en la imagen de destino. Repetimos para todos los píxeles y todas las bandas de color, lo que nos da una imagen posterizada.

**TonoOp:** Por último tenemos el operador TonoOp, que se encargará de cambiar el tono de color de una imagen. Básicamente sigue este procedimiento por cada pixel:
-Pasa de RGB a HSB.
-Realiza esta transformación: hsb[0] = (((hsb[0] * 360) + tono) % 360) / 360;
-Vuelve a pasar de HSB a RGB

**Reminder: Mirar el operador propio explicado en los requisitos de imágenes, la umbralización binaria.**

Revisar JavaDocs del proyecto para ver el código con más detalle.