**Java GUI Project**

**1. Introduction**

This Java GUI project, named "GuiDemo," is a graphical user interface application designed to create and customize images with text, background images, and stamp-like icons. The project utilizes the Swing library for creating the GUI components.

**2. Features**

* **Text Customization:** Allows users to set the text, font size, font style (bold, italic), and text justification.
* **Background Customization:** Provides options for selecting background images, solid colors, or gradient overlays.
* **Stamp-like Icons:** Users can select stamp-like icons from a predefined set and place them on the canvas.
* **File Operations:** Supports creating a new canvas, saving the current image, and exiting the application.
* **Resizable Canvas:** The canvas size can be adjusted to accommodate various image dimensions.

**3. Project Structure**

The project is structured into several Java classes and packages. Key components include **GuiDemo** (the main class), **DrawPanel** (the canvas for drawing), **IconSupport** (support for stamp-like icons), and **TextMenu** (menu for text customization).

**4. Classes and Packages**

* **GuiDemo**: The main class that initializes the GUI components and sets up the application window.
* **DrawPanel**: Represents the drawing canvas, including text, background, and stamp-like icons.
* **IconSupport**: Manages stamp-like icons and provides a toolbar for selecting and placing them.
* **TextMenu**: Handles the text customization menu.

**5. How to Use**

1. **Text Customization:**
   * Click on "Text" in the menu bar to access text customization options.
   * Change the text, font size, style, and justification.
2. **Background Customization:**
   * Click on "Background" in the menu bar to change the canvas background.
   * Select from built-in images, use a solid color, or set a custom image.
3. **Stamp-like Icons:**
   * Click on "Stamps" in the menu bar to access stamp-like icons.
   * Choose an icon from the toolbar and click on the canvas to place it.
4. **File Operations:**
   * Use the "File" menu for creating a new canvas, saving the image, and quitting the application.

**6. Customization**

* **Canvas Size:** The canvas size can be adjusted by resizing the application window.
* **Stamp-like Icons:** Additional icons can be added by modifying the **iconNames** array in **IconSupport**.

**7. Program**

package guidemo;

import java.awt.\*;

import java.awt.event.\*;

import java.awt.image.BufferedImage;

import java.io.File;

import javax.imageio.ImageIO;

import javax.swing.\*;

/\*\*

 \* A frame that displays a multiline text, possibly with a background image

 \* and with added icon images, in a DrawPanel, along with a variety of

 \* controlls.

 \*/

public class GuiDemo extends JFrame {

    /\*\*

     \* The main program just creates a GuiDemo frame and makes it visible.

     \*/

    public static void main(String[] args) {

        JFrame frame = new GuiDemo();

        frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

        frame.setVisible(true);

    }

    private DrawPanel drawPanel;

    private SimpleFileChooser fileChooser;

    private TextMenu textMenu;

    private JCheckBoxMenuItem gradientOverlayCheckbox = new JCheckBoxMenuItem("Gradient Overlay", true);

    /\*\*

     \* The constructor creates the frame, sizes it, and centers it horizontally on

     \* the screen.

     \*/

    public GuiDemo() {

        super("GUI Project"); // Specifies the string for the title bar of the window.

        JPanel content = new JPanel(); // To hold the content of the window.

        content.setBackground(Color.LIGHT\_GRAY);

        content.setLayout(new BorderLayout());

        setContentPane(content);

        // Add a background toolbar to the NORTH position of the layout

        content.add(makeToolbar(), BorderLayout.NORTH);

        // Create the DrawPanel that fills most of the window, and customize it.

        drawPanel = new DrawPanel();

        drawPanel.getTextItem().setText(

                "Kindness is a language understood by all.\n" +

                        "Spread it generously,\n" +

                        "and watch the world transform.");

        drawPanel.getTextItem().setFontSize(36);

        drawPanel.getTextItem().setJustify(TextItem.LEFT);

        drawPanel.setBackgroundImage(Util.getImageResource("resources/images/mandelbrot.jpeg"));

        content.add(drawPanel, BorderLayout.CENTER);

        // Add an icon toolbar to the SOUTH position of the layout

        IconSupport iconSupport = new IconSupport(drawPanel);

        content.add(iconSupport.createToolbar(true), BorderLayout.SOUTH);

        // Create the menu bar and add it to the frame. The TextMenu is defined by

        // a separate class. The other menus are created in this class.

        JMenuBar menuBar = new JMenuBar();

        menuBar.add(makeFileMenu());

        textMenu = new TextMenu(drawPanel);

        menuBar.add(textMenu);

        menuBar.add(iconSupport.createMenu());

        menuBar.add(makeBackgroundMenu());

        setJMenuBar(menuBar);

        // Set the size of the window and its position.

        pack(); // Size the window to fit its content.

        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();

        setLocation((screenSize.width - getWidth()) / 2, 50);

        // Create and customize the file chooser that is used for file operations.

        fileChooser = new SimpleFileChooser();

        try { // I'd like to use the Desktop folder as the initial folder in the file chooser.

            String userDir = System.getProperty("user.home");

            if (userDir != null) {

                File desktop = new File(userDir, "Desktop");

                if (desktop.isDirectory())

                    fileChooser.setDefaultDirectory(desktop);

            }

        } catch (Exception e) {

        }

    } // end constructor

    private JToolBar makeToolbar() {

        JToolBar toolbar = new JToolBar("Background");

        toolbar.add(new ChooseBackgroundAction("Mandelbrot"));

        toolbar.add(new ChooseBackgroundAction("Earthrise"));

        toolbar.add(new ChooseBackgroundAction("Sunset"));

        toolbar.add(new ChooseBackgroundAction("Cloud"));

        toolbar.add(new ChooseBackgroundAction("Eagle\_nebula"));

        toolbar.addSeparator();

        toolbar.add(new ChooseBackgroundAction("Custom..."));

        toolbar.addSeparator();

        toolbar.add(new ChooseBackgroundAction("Color..."));

        // Add existing actions newPictureAction and saveImageAction to the toolbar

        toolbar.addSeparator();

        toolbar.add(newPictureAction);

        toolbar.add(saveImageAction);

        return toolbar;

    }

    /\*\*

     \* Create the "File" menu from actions that are defined later in this class.

     \*/

    private JMenu makeFileMenu() {

        JMenu menu = new JMenu("File");

        menu.add(newPictureAction);

        menu.add(saveImageAction);

        menu.addSeparator();

        menu.add(quitAction);

        return menu;

    }

    /\*\*

     \* Create the "Background" menu, using objects of type ChooseBackgroundAction,

     \* a class that is defined later in this file.

     \*/

    private JMenu makeBackgroundMenu() {

        JMenu menu = new JMenu("Background");

        menu.add(new ChooseBackgroundAction("Mandelbrot"));

        menu.add(new ChooseBackgroundAction("Earthrise"));

        menu.add(new ChooseBackgroundAction("Sunset"));

        menu.add(new ChooseBackgroundAction("Cloud"));

        menu.add(new ChooseBackgroundAction("Eagle\_nebula"));

        menu.addSeparator();

        menu.add(new ChooseBackgroundAction("Custom..."));

        menu.addSeparator();

        menu.add(new ChooseBackgroundAction("Color..."));

        menu.addSeparator();

        menu.add(gradientOverlayCheckbox);

        gradientOverlayCheckbox.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent evt) {

                if (gradientOverlayCheckbox.isSelected())

                    drawPanel.setGradientOverlayColor(Color.WHITE);

                else

                    drawPanel.setGradientOverlayColor(null);

            }

        });

        return menu;

    }

    private AbstractAction newPictureAction = new AbstractAction("New",

            Util.iconFromResource("resources/action\_icons/fileopen.png")) {

        public void actionPerformed(ActionEvent evt) {

            drawPanel.clear();

            gradientOverlayCheckbox.setSelected(true);

            textMenu.setDefaults();

        }

    };

    private AbstractAction quitAction = new AbstractAction("Quit",

            Util.iconFromResource("resources/action\_icons/exit.png")) {

        public void actionPerformed(ActionEvent evt) {

            System.exit(0);

        }

    };

    private AbstractAction saveImageAction = new AbstractAction("Save Image...",

            Util.iconFromResource("resources/action\_icons/filesave.png")) {

        public void actionPerformed(ActionEvent evt) {

            File f = fileChooser.getOutputFile(drawPanel, "Select Ouput File", "saying.jpeg");

            if (f != null) {

                try {

                    BufferedImage img = drawPanel.copyImage();

                    String format;

                    String fileName = f.getName().toLowerCase();

                    if (fileName.endsWith(".png"))

                        format = "PNG";

                    else if (fileName.endsWith(".jpeg") || fileName.endsWith(".jpg"))

                        format = "JPEG";

                    else {

                        JOptionPane.showMessageDialog(drawPanel,

                                "The output file name must end wth\n.png or .jpeg.");

                        return;

                    }

                    ImageIO.write(img, format, f);

                } catch (Exception e) {

                    JOptionPane.showMessageDialog(drawPanel, "Sorry, the image could not be saved.");

                }

            }

        }

    };

    /\*\*

     \* An object of type ChooseBackgroudnAction represents an action through which

     \* the

     \* user selects the background of the picture. There are three types of

     \* background:

     \* solid color background ("Color..." command), an image selected by the user

     \* from

     \* the file system ("Custom..." command), and four built-in image resources

     \* (Mandelbrot, Earthrise, Sunset, and Eagle\_nebula).

     \*/

    private class ChooseBackgroundAction extends AbstractAction {

        String text;

        ChooseBackgroundAction(String text) {

            super(text);

            this.text = text;

            if (!text.equals("Custom...") && !text.equals("Color...")) {

                putValue(Action.SMALL\_ICON,

                        Util.iconFromResource("resources/images/" + text.toLowerCase() + "\_thumbnail.jpeg"));

            }

            if (text.equals("Color..."))

                putValue(Action.SHORT\_DESCRIPTION,

                        "<html>Use a solid color for background<br>instead of an image.</html>");

            else if (text.equals("Custom..."))

                putValue(Action.SHORT\_DESCRIPTION, "<html>Select an image file<br>to use as the background.</html>");

            if (text.equals("Color...")) {

                putValue(Action.SMALL\_ICON, createColorIcon());

                // putValue(Action.SHORT\_DESCRIPTION,

                // "<html>Use a solid color for background<br>instead of an image.</html>");

            } else if (text.equals("Custom...")) {

                putValue(Action.SHORT\_DESCRIPTION, "<html>Select an image file<br>to use as the background.</html>");

            } else {

                putValue(Action.SHORT\_DESCRIPTION, "Use this image as the background.");

            }

        }

        private ImageIcon createColorIcon() {

            int iconSize = 32;

            BufferedImage colorImage = new BufferedImage(iconSize, iconSize, BufferedImage.TYPE\_INT\_RGB);

            Graphics g = colorImage.getGraphics();

            g.setColor(Color.DARK\_GRAY);

            g.fillRect(0, 0, iconSize, iconSize);

            g.setColor(Color.LIGHT\_GRAY);

            g.fillRect(4, 4, iconSize - 8, iconSize - 8);

            g.dispose();

            return new ImageIcon(colorImage);

        }

        public void actionPerformed(ActionEvent evt) {

            if (text.equals("Custom...")) {

                File inputFile = fileChooser.getInputFile(drawPanel, "Select Background Image");

                if (inputFile != null) {

                    try {

                        BufferedImage img = ImageIO.read(inputFile);

                        if (img == null)

                            throw new Exception();

                        drawPanel.setBackgroundImage(img);

                    } catch (Exception e) {

                        JOptionPane.showMessageDialog(drawPanel, "Sorry, couldn't read the file.");

                    }

                }

            } else if (text.equals("Color...")) {

                Color c = JColorChooser.showDialog(drawPanel, "Select Color for Background", drawPanel.getBackground());

                if (c != null) {

                    drawPanel.setBackground(c);

                    drawPanel.setBackgroundImage(null);

                }

            } else {

                Image bg = Util.getImageResource("resources/images/" + text.toLowerCase() + ".jpeg");

                drawPanel.setBackgroundImage(bg);

            }

        }

    }

}

package guidemo;

import java.applet.AudioClip;

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

import java.awt.image.BufferedImage;

import java.util.ArrayList;

/\*\*

 \* A panel that can display a background image, a gradient over the image that

 \* changes

 \* from almost transparent at the top to almost opaque at the bottom, a

 \* multiline text,

 \* and a list of small images on top of everything else. The small images are

 \* placed by

 \* clicking with the mouse. The image that is placed is determined by the

 \* currentDrawImage

 \* property; if this property is null, then clicking an existing image with the

 \* mouse will

 \* remove that image.

 \*/

public class DrawPanel extends JPanel {

    private TextItem text = new TextItem(); // The TextItem displayed in this image.

                                            // It can be retrieved with getTextItem but can't be set.

    private Image backgroundImage = null; // Seven properties that have "get" and "set" methods.

    private Color borderColor = Color.DARK\_GRAY;

    private int borderThickness = 3;

    private Color gradientOverlayColor = Color.WHITE;

    private boolean horizontalOverlay = false;

    private BufferedImage currentDrawImage;

    private ArrayList<ImageItem> images = new ArrayList<ImageItem>(); // three objects for internal use only

    public DrawPanel() {

        setPreferredSize(new Dimension(800, 600));

        setBackground(Color.DARK\_GRAY);

        setBorder(BorderFactory.createLineBorder(borderColor, borderThickness));

        setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR\_CURSOR));

        addMouseListener(new MouseAdapter() {

            AudioClip clink = Util.getSound("resources/sounds/clink.wav");

            AudioClip lase = Util.getSound("resources/sounds/lase.wav");

            public void mousePressed(MouseEvent evt) {

                int x = evt.getX();

                int y = evt.getY();

                if (currentDrawImage != null) {

                    if (clink != null)

                        clink.play();

                    images.add(new ImageItem(currentDrawImage, x, y));

                    repaint();

                } else {

                    for (int i = images.size() - 1; i >= 0; i--)

                        if (images.get(i).contains(x, y)) {

                            if (lase != null)

                                lase.play();

                            images.remove(i);

                            repaint();

                            break;

                        }

                }

            }

        });

    }

    protected void paintComponent(Graphics g1) {

        super.paintComponent(g1);

        Graphics2D g2 = (Graphics2D) g1;

        g2.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);

        if (backgroundImage != null)

            g2.drawImage(backgroundImage, 0, 0, getWidth(), getHeight(), this);

        if (gradientOverlayColor != null) {

            int r = gradientOverlayColor.getRed();

            int b = gradientOverlayColor.getBlue();

            int g = gradientOverlayColor.getGreen();

            Color startColor = new Color(r, g, b, 50);

            Color endColor = new Color(r, g, b, 200);

            if (horizontalOverlay)

                g2.setPaint(new GradientPaint(0, 0, startColor, getWidth(), 0, endColor, false));

            else

                g2.setPaint(new GradientPaint(0, 0, startColor, 0, getHeight(), endColor, false));

            g2.fillRect(0, 0, getWidth(), getHeight());

        }

        text.draw(g2, getWidth() / 2, getHeight() / 2);

        for (ImageItem img : images)

            img.draw(g2);

    }

    public Image getBackgroundImage() {

        return backgroundImage;

    }

    public void setBackgroundImage(Image backgroundImage) {

        this.backgroundImage = backgroundImage;

        repaint();

    }

    public Color getBorderColor() {

        return borderColor;

    }

    public void setBorderColor(Color borderColor) {

        this.borderColor = borderColor;

        setBorder(BorderFactory.createLineBorder(borderColor, borderThickness));

        repaint();

    }

    public int getBorderThickness() {

        return borderThickness;

    }

    public void setBorderThickness(int borderThickness) {

        this.borderThickness = borderThickness;

        setBorder(BorderFactory.createLineBorder(borderColor, borderThickness));

        repaint();

    }

    public Color getGradientOverlayColor() {

        return gradientOverlayColor;

    }

    public void setGradientOverlayColor(Color gradientOverlayColor) {

        this.gradientOverlayColor = gradientOverlayColor;

        repaint();

    }

    public boolean isHorizontalOverlay() {

        return horizontalOverlay;

    }

    public void setHorizontalOverlay(boolean horizontalOverlay) {

        this.horizontalOverlay = horizontalOverlay;

        repaint();

    }

    public BufferedImage getCurrentDrawImage() {

        return currentDrawImage;

    }

    public void setCurrentDrawImage(BufferedImage currentDrawImage) {

        this.currentDrawImage = currentDrawImage;

    }

    public TextItem getTextItem() {

        return text;

    }

    /\*\*

     \* Create and return a BufferedImage containing the same picture that is

     \* shown in this panel.

     \*/

    public BufferedImage copyImage() {

        BufferedImage copy = new BufferedImage(getWidth(), getHeight(), BufferedImage.TYPE\_INT\_RGB);

        Graphics g = copy.createGraphics();

        paintComponent(g);

        g.dispose();

        return copy;

    }

    /\*\*

     \* Return this panel to its default state. (The text will be "Hello World", on a

     \* gray

     \* background.)

     \*/

    public void clear() {

        text = new TextItem();

        backgroundImage = null;

        setBackground(Color.DARK\_GRAY);

        gradientOverlayColor = Color.WHITE;

        horizontalOverlay = false;

        borderThickness = 3;

        setBorderColor(Color.DARK\_GRAY);

        images.clear();

        repaint();

    }

}

package guidemo;

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

/\*\*

 \* Defines a modal dialog for inputing multiline text.

 \*/

public class GetTextDialog extends JDialog {

    private boolean canceled = false;

    private JTextArea text;

    /\*\*

     \* Display the dialog box, wait for the user to dismiss it, and return the

     \* user's input, or null if the user cancels the dialog.

     \*

     \* @param parent      A component whose frame is the parent of the dialog box.

     \* @param initialText the initial contents of the inputs box; if null, the box

     \*                    is initially empty.

     \* @return the text from the input box, or null if the user cancels the dialog.

     \*         Note that the return can be a blank string if the user clicks "OK"

     \*         without entering

     \*         any text.

     \*/

    public static String showDialog(Component parent, String initialText) {

        GetTextDialog dialog = new GetTextDialog(frameAncestor(parent), initialText);

        dialog.setVisible(true);

        if (dialog.canceled)

            return null;

        else

            return dialog.text.getText();

    }

    private static Frame frameAncestor(Component c) {

        while (c != null && !(c instanceof Frame))

            c = c.getParent();

        return (Frame) c;

    }

    /\*\*

     \* Creates, but does not show, a dialog box.

     \*/

    private GetTextDialog(Frame parent, String initialText) {

        super(parent, "Input Your Text", true);

        JPanel content = new JPanel();

        setContentPane(content);

        content.setBackground(Color.LIGHT\_GRAY);

        content.setLayout(new BorderLayout(3, 3));

        text = new JTextArea(10, 50);

        text.setMargin(new Insets(6, 6, 6, 6));

        if (initialText != null)

            text.setText(initialText);

        content.add(text, BorderLayout.CENTER);

        JPanel bottom = new JPanel();

        content.add(bottom, BorderLayout.SOUTH);

        JButton cancel = new JButton("Cancel");

        cancel.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent evt) {

                canceled = true;

                dispose();

            }

        });

        JButton ok = new JButton("OK");

        ok.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent evt) {

                dispose();

            }

        });

        bottom.add(cancel);

        bottom.add(ok);

        pack();

        setDefaultCloseOperation(JDialog.DO\_NOTHING\_ON\_CLOSE);

    }

}

package guidemo;

import java.awt.\*;

import java.awt.event.\*;

import java.awt.image.BufferedImage;

import javax.swing.\*;

import java.util.ArrayList;

/\*\*

 \* Contains a set of Actions that can be used to select images that can

 \* be added to a DrawPanel (in the form of ImageItems). Can create a

 \* toolbar containing a button for each Action in the set. A button

 \* shows an ImageIcon with the image that is selected by that button.

 \* Clicking one of the buttons also sets the cursor in the DrawPanel

 \* to be a (rough) copy of the image.

 \*/

public class IconSupport {

    private DrawPanel panel;

    private ArrayList<BufferedImage> iconImages = new ArrayList<BufferedImage>();

    private ArrayList<Action> actions = new ArrayList<Action>();

    public IconSupport(DrawPanel owner) {

        panel = owner;

        String[] iconNames = { "bell", "camera", "flower", "star", "check", "crossout",

                "tux", "bomb", "keyboard", "lightbulb", "tv" };

        for (String name : iconNames) {

            BufferedImage img = Util.getBufferedImageResource("resources/icons/" + name + ".png");

            if (img != null) {

                iconImages.add(img);

                actions.add(new SelectIconAction(name, iconImages.size() - 1));

            }

        }

        actions.add(new NoIconAction());

    }

    public JMenu createMenu() {

        JMenu menu = new JMenu("Stamps");

        for (Action action : actions) {

            menu.add(action);

        }

        return menu;

    }

    /\*\*

     \* Return a toolbar containing buttons representing the images that can be added

     \* to the DrawPanel.

     \*

     \* @param horizontal a value of JToolBar.HORIZONTAL or JToolBar.VERTICAL tells

     \*                   whether the toolbar is meant to have horizontal or vertical

     \*                   orientation.

     \*/

    public JToolBar createToolbar(boolean horizontal) {

        JToolBar tbar = new JToolBar(horizontal ? JToolBar.HORIZONTAL : JToolBar.VERTICAL);

        for (int i = 0; i < actions.size() - 1; i++)

            tbar.add(actions.get(i));

        tbar.addSeparator(new Dimension(15, 0));

        tbar.add(actions.get(actions.size() - 1));

        return tbar;

    }

    private class NoIconAction extends AbstractAction {

        NoIconAction() {

            super("Eraser");

            BufferedImage del = new BufferedImage(32, 32, BufferedImage.TYPE\_INT\_ARGB);

            Graphics g = del.createGraphics();

            g.setColor(Color.WHITE);

            g.fillRect(0, 0, 32, 32);

            g.setColor(Color.RED);

            g.drawString("DEL", 5, 20);

            g.dispose();

            putValue(Action.SMALL\_ICON, new ImageIcon(del));

            putValue(Action.SHORT\_DESCRIPTION, "Use Mouse to Erase Icons"); // tooltip

        }

        public void actionPerformed(ActionEvent evt) {

            panel.setCurrentDrawImage(null);

            panel.setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR\_CURSOR));

        }

    }

    private class SelectIconAction extends AbstractAction {

        int iconNumber;

        SelectIconAction(String name, int n) {

            // Note: The name is surpressed in toolbars, but not in menus.

            super(name, new ImageIcon(iconImages.get(n)));

            iconNumber = n;

            putValue(Action.SHORT\_DESCRIPTION, "Use Mouse to Stamp this Icon"); // tooltip

        }

        public void actionPerformed(ActionEvent evt) {

            BufferedImage image = iconImages.get(iconNumber);

            panel.setCurrentDrawImage(image);

            Cursor c = Util.createImageCursor(image, image.getWidth() / 2, image.getHeight() / 2);

            panel.setCursor(c);

        }

    }

}

package guidemo;

import java.awt.Graphics;

import java.awt.image.BufferedImage;

/\*\*

 \* Represents an image, drawn with its center at a specified point.

 \*/

public class ImageItem {

    private BufferedImage image;

    private int centerX, centerY;

    public ImageItem(BufferedImage image, int centerX, int centerY) {

        this.image = image;

        this.centerX = centerX;

        this.centerY = centerY;

    }

    public void draw(Graphics g) {

        g.drawImage(image, centerX - image.getWidth() / 2, centerY - image.getHeight() / 2, null);

    }

    public BufferedImage getImage() {

        return image;

    }

    public void setImage(BufferedImage image) {

        if (image == null)

            throw new IllegalArgumentException("Null image not allowed");

        this.image = image;

    }

    public int getCenterX() {

        return centerX;

    }

    public void setCenterX(int centerX) {

        this.centerX = centerX;

    }

    public int getCenterY() {

        return centerY;

    }

    public void setCenterY(int centerY) {

        this.centerY = centerY;

    }

    public void setPosition(int x, int y) {

        centerX = x;

        centerY = y;

    }

    public boolean contains(int x, int y) {

        int w = image.getWidth();

        int h = image.getHeight();

        return x > centerX - w / 2 && x < centerX + w / 2 && y > centerY - h / 2 && y < centerY + h / 2;

    }

}

package guidemo;

import java.awt.Component;

import java.io.File;

import javax.swing.JFileChooser;

import javax.swing.JOptionPane;

/\*\*

 \* This class provides a slightly simplified interface to one of Java's

 \* standard JFileChooser dialogs. An object of type SimpleFileChooser

 \* has methods that allow the user to select files for input or output.

 \* If the object is used several times, the same JFileChooser is used

 \* each time. By default, the dialog box is set to the user's home

 \* directory the first time it is used, and after that it remembers the

 \* current directory between one use and the next. However, methods

 \* are provided for setting the current directory. (Note: On Windows,

 \* the user's home directory will probably mean the user's "My Documents"

 \* directory".)

 \*/

public class SimpleFileChooser {

    private JFileChooser dialog; // The dialog, which is created when needed.

    /\*\*

     \* Reset the default directory in the dialog box to the user's home

     \* directory. The next time the dialog appears, it will show the

     \* contents of that directory.

     \*/

    public void setDefaultDirectory() {

        if (dialog != null)

            dialog.setCurrentDirectory(null);

    }

    /\*\*

     \* Set the default directory for the dialog box. The next time the

     \* dialog appears, it will show the contents of that directory.

     \*

     \* @param directoryName A File object that specifies the directory name.

     \*                      If this name is null, then the user's home directory

     \*                      will be used.

     \*/

    public void setDefaultDirectory(String directoryName) {

        if (dialog == null)

            dialog = new JFileChooser();

        dialog.setCurrentDirectory(new File(directoryName));

    }

    /\*\*

     \* Set the default directory for the dialog box. The next time the

     \* dialog appears, it will show the contents of that directory.

     \*

     \* @param directoryName The name of the new default directory. If the

     \*                      name is null, then the user's home directory will be

     \*                      used.

     \*/

    public void setDefaultDirectory(File directory) {

        if (dialog == null)

            dialog = new JFileChooser();

        dialog.setCurrentDirectory(directory);

    }

    /\*\*

     \* Show a dialog box where the user can select a file for reading.

     \* This method simply returns <code>getInputFile(null,null)</code>.

     \*

     \* @see #getInputFile(Component, String)

     \* @return the selected file, or null if the user did not select a file.

     \*/

    public File getInputFile() {

        return getInputFile(null, null);

    }

    /\*\*

     \* Show a dialog box where the user can select a file for reading.

     \* This method simply returns <code>getInputFile(parent,null)</code>.

     \*

     \* @see #getInputFile(Component, String)

     \* @return the selected file, or null if the user did not select a file.

     \*/

    public File getInputFile(Component parent) {

        return getInputFile(parent, null);

    }

    /\*\*

     \* Show a dialog box where the user can select a file for reading.

     \* If the user cancels the dialog by clicking its "Cancel" button or

     \* the Close button in the title bar, then the return value of this

     \* method is null. Otherwise, the return value is the selected file.

     \* Note that the file has to exist, but it is not guaranteed that the

     \* user is allowed to read the file.

     \*

     \* @param parent      If the parent is non-null, then the window that contains

     \*                    the parent component becomes the parent window of the

     \*                    dialog box. This

     \*                    means that the window is "blocked" until the dialog is

     \*                    dismissed. Also,

     \*                    the dialog box's position on the screen should be based on

     \*                    the position of

     \*                    the window. Generally, you should pass your application's

     \*                    main window or

     \*                    panel as the value of this parameter.

     \* @param dialogTitle a title to be displayed in the title bar of the dialog

     \*                    box. If the value of this parameter is null, then the

     \*                    dialog title will

     \*                    be "Select Input File".

     \* @return the selected file, or null if the user did not select a file.

     \*/

    public File getInputFile(Component parent, String dialogTitle) {

        if (dialog == null)

            dialog = new JFileChooser();

        if (dialogTitle != null)

            dialog.setDialogTitle(dialogTitle);

        else

            dialog.setDialogTitle("Select Input File");

        int option = dialog.showOpenDialog(parent);

        if (option != JFileChooser.APPROVE\_OPTION)

            return null; // User canceled or clicked the dialog's close box.

        File selectedFile = dialog.getSelectedFile();

        return selectedFile;

    }

    /\*\*

     \* Show a dialog box where the user can select a file for writing.

     \* This method simply calls <code>getOutputFile(null,null,null)</code>

     \*

     \* @see #getOutputFile(Component, String, String)

     \* @return the selcted file, or null if no file was selected.

     \*/

    public File getOutputFile() {

        return getOutputFile(null, null, null);

    }

    /\*\*

     \* Show a dialog box where the user can select a file for writing.

     \* This method simply calls <code>getOutputFile(null,null,null)</code>

     \*

     \* @see #getOutputFile(Component, String, String)

     \* @return the selcted file, or null if no file was selected.

     \*/

    public File getOutputFile(Component parent) {

        return getOutputFile(parent, null, null);

    }

    /\*\*

     \* Show a dialog box where the user can select a file for writing.

     \* This method calls <code>getOutputFile(parent,dialogTitle,null)</code>

     \*

     \* @see #getOutputFile(Component, String, String)

     \* @return the selcted file, or null if no file was selected.

     \*/

    public File getOutputFile(Component parent, String dialogTitle) {

        return getOutputFile(parent, dialogTitle, null);

    }

    /\*\*

     \* Show a dialog box where the user can select a file for writing.

     \* If the user cancels the dialog by clicking its "Cancel" button or

     \* the Close button in the title bar, then the return value of this

     \* method is null. A non-null value indicates that the user specified

     \* a file name and that, if the file exists, then the user wants to

     \* replace that file. (If the user selects a file that already exists,

     \* then the user will be asked whether to replace the existing file.)

     \* Note that it is not quaranteed that the selected file is actually

     \* writable; the user might not have permission to create or modify the file.

     \*

     \* @param parent      If the parent is non-null, then the window that contains

     \*                    the parent component becomes the parent window of the

     \*                    dialog box. This

     \*                    means that the window is "blocked" until the dialog is

     \*                    dismissed. Also,

     \*                    the dialog box's position on the screen should be based on

     \*                    the position of

     \*                    the window. Generally, you should pass your application's

     \*                    main window or

     \*                    panel as the value of this parameter.

     \* @param dialogTitle a title to be displayed in the title bar of the dialog

     \*                    box. If the value of this parameter is null, then the

     \*                    dialog title will

     \*                    be "Select Input File".

     \* @param defaultFile when the dialog appears, this name will be filled in

     \*                    as the name of the selected file. If the value of this

     \*                    parameter is null,

     \*                    then the file name box will be empty.

     \* @return the selected file, or null if the user did not select a file.

     \*/

    public File getOutputFile(Component parent,

            String dialogTitle, String defaultFile) {

        if (dialog == null)

            dialog = new JFileChooser();

        if (dialogTitle != null)

            dialog.setDialogTitle(dialogTitle);

        else

            dialog.setDialogTitle("Select Output File");

        if (defaultFile == null)

            dialog.setSelectedFile(null);

        else

            dialog.setSelectedFile(new File(defaultFile));

        while (true) {

            int option = dialog.showSaveDialog(parent);

            if (option != JFileChooser.APPROVE\_OPTION)

                return null; // User canceled or clicked the dialog's close box.

            File selectedFile = dialog.getSelectedFile();

            if (!selectedFile.exists())

                return selectedFile;

            else { // Ask the user whether to replace the file.

                int response = JOptionPane.showConfirmDialog(parent,

                        "The file \"" + selectedFile.getName()

                                + "\" already exists.\nDo you want to replace it?",

                        "Confirm Save",

                        JOptionPane.YES\_NO\_CANCEL\_OPTION,

                        JOptionPane.WARNING\_MESSAGE);

                if (response == JOptionPane.CANCEL\_OPTION)

                    return null; // User does not want to select a file.

                if (response == JOptionPane.YES\_OPTION)

                    return selectedFile; // User wants to replace the file

                // A "No" response will cause the file dialog to be shown again.

            }

        }

    }

}

package guidemo;

import java.awt.\*;

import java.util.ArrayList;

import java.util.Scanner;

/\*\*

 \* Represents a multiline text, with various properties that can be

 \* set. A draw() method is included that will draw the text in a

 \* graphics context, centered at a specified point.

 \*/

public class TextItem {

    public final static int CENTER = 0; // Constants for use with setJustify()

    public final static int LEFT = 1;

    public final static int RIGHT = 2;

    private String text = "Hello\nWorld"; // the displayed text, with '\n' indicating line breaks.

    private Color color = Color.BLACK;

    private double lineHeightMultiplier = 1;

    private boolean bold;

    private boolean italic;

    private int fontSize = 30;

    private String fontName = "Serif";

    private int justify = LEFT;

    private String[] lines = { "Hello", "World" }; // same as text, but broken into individual lines.

    public void draw(Graphics g, int centerX, int centerY) {

        Color saveColor = g.getColor();

        Font saveFont = g.getFont();

        int style;

        if (italic && bold)

            style = Font.BOLD | Font.ITALIC;

        else if (italic)

            style = Font.ITALIC;

        else if (bold)

            style = Font.BOLD;

        else

            style = Font.PLAIN;

        Font font = new Font(fontName, style, fontSize);

        g.setFont(font);

        FontMetrics fm = g.getFontMetrics(font);

        double lineHeight = fm.getHeight() \* lineHeightMultiplier;

        int totalHeight = (int) (lineHeight \* (lines.length - 1)) + fm.getAscent() + fm.getDescent();

        if (color != null)

            g.setColor(color);

        int[] widths = new int[lines.length];

        int totalWidth = 0;

        for (int i = 0; i < lines.length; i++) {

            widths[i] = fm.stringWidth(lines[i]);

            if (widths[i] > totalWidth)

                totalWidth = widths[i];

        }

        for (int i = 0; i < lines.length; i++) {

            int x;

            if (justify == CENTER)

                x = centerX - widths[i] / 2;

            else if (justify == LEFT)

                x = centerX - totalWidth / 2;

            else

                x = centerX + totalWidth / 2 - fm.stringWidth(lines[i]);

            int y = centerY - totalHeight / 2 + fm.getAscent() + (int) (i \* lineHeight);

            g.drawString(lines[i], x, y);

        }

        g.setColor(saveColor);

        g.setFont(saveFont);

    }

    public String getText() {

        return text;

    }

    public void setText(String newText) {

        Scanner reader = new Scanner(newText);

        ArrayList<String> s = new ArrayList<String>();

        while (reader.hasNextLine()) {

            s.add(reader.nextLine());

        }

        while (s.size() > 0 && s.get(0).trim().length() == 0)

            s.remove(0); // remove blank lines from front

        while (s.size() > 0 && s.get(s.size() - 1).trim().length() == 0)

            s.remove(s.size() - 1); // remove blank lines from end

        if (s.size() == 0)

            throw new IllegalArgumentException("Text can't be empty.");

        lines = new String[s.size()];

        for (int i = 0; i < lines.length; i++)

            lines[i] = s.get(i);

        text = newText;

    }

    public Color getColor() {

        return color;

    }

    public void setColor(Color color) {

        this.color = color;

    }

    public double getLineHeightMultiplier() {

        return lineHeightMultiplier;

    }

    public void setLineHeightMultiplier(double lineHeightMultiplier) {

        if (lineHeightMultiplier < 0)

            throw new IllegalArgumentException("Line height multiplier cannot be negative.");

        this.lineHeightMultiplier = lineHeightMultiplier;

    }

    public boolean isBold() {

        return bold;

    }

    public void setBold(boolean bold) {

        this.bold = bold;

    }

    public boolean isItalic() {

        return italic;

    }

    public void setItalic(boolean italic) {

        this.italic = italic;

    }

    public int getFontSize() {

        return fontSize;

    }

    public void setFontSize(int fontSize) {

        if (fontSize <= 0)

            throw new IllegalArgumentException("Font size must be positive.");

        this.fontSize = fontSize;

    }

    public String getFontName() {

        return fontName;

    }

    public void setFontName(String fontName) {

        this.fontName = fontName;

    }

    public int getJustify() {

        return justify;

    }

    public void setJustify(int justify) {

        if (justify != CENTER && justify != RIGHT && justify != LEFT)

            throw new IllegalArgumentException("Justify can only be CENTER, LEFT, or RIGHT");

        this.justify = justify;

    }

}

package guidemo;

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;;

/\*\*

 \* A menu full of commands that affect the text shown

 \* in a DrawPanel.

 \*/

public class TextMenu extends JMenu {

    private final DrawPanel panel; // the panel whose text is controlled by this menu

    private JCheckBoxMenuItem bold; // controls whether the text is bold or not.

    private JCheckBoxMenuItem italic; // controls whether the text is italic or not.

    // New variables for text justification

    private JRadioButtonMenuItem leftJustify;

    private JRadioButtonMenuItem rightJustify;

    private JRadioButtonMenuItem centerJustify;

    /\*\*

     \* Constructor creates all the menu commands and adds them to the menu.

     \*

     \* @param owner the panel whose text will be controlled by this menu.

     \*/

    public TextMenu(DrawPanel owner) {

        super("Text");

        this.panel = owner;

        final JMenuItem change = new JMenuItem("Change Text...");

        change.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent evt) {

                String currentText = panel.getTextItem().getText();

                String newText = GetTextDialog.showDialog(panel, currentText);

                if (newText != null && newText.trim().length() > 0) {

                    panel.getTextItem().setText(newText);

                    panel.repaint();

                }

            }

        });

        final JMenuItem size = new JMenuItem("Set Size...");

        size.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent evt) {

                int currentSize = panel.getTextItem().getFontSize();

                String s = JOptionPane.showInputDialog(panel, "What font size do you want to use?", currentSize);

                if (s != null && s.trim().length() > 0) {

                    try {

                        int newSize = Integer.parseInt(s.trim()); // can throw NumberFormatException

                        panel.getTextItem().setFontSize(newSize); // can throw IllegalArgumentException

                        panel.repaint();

                    } catch (Exception e) {

                        JOptionPane.showMessageDialog(panel, s + " is not a legal text size.\n"

                                + "Please enter a positive integer.");

                    }

                }

            }

        });

        final JMenuItem color = new JMenuItem("Set Color...");

        color.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent evt) {

                Color currentColor = panel.getTextItem().getColor();

                Color newColor = JColorChooser.showDialog(panel, "Select Text Color", currentColor);

                if (newColor != null) {

                    panel.getTextItem().setColor(newColor);

                    panel.repaint();

                }

            }

        });

        // New JMenuItem for setting line height

        final JMenuItem setLineHeight = new JMenuItem("Set Line Height...");

        setLineHeight.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent evt) {

                setLineHeightMultiplier();

            }

        });

        makeJustifyMenu();

        italic = new JCheckBoxMenuItem("Italic");

        italic.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent evt) {

                panel.getTextItem().setItalic(italic.isSelected());

                panel.repaint();

            }

        });

        bold = new JCheckBoxMenuItem("Bold");

        bold.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent evt) {

                panel.getTextItem().setBold(bold.isSelected());

                panel.repaint();

            }

        });

        add(change);

        addSeparator();

        add(size);

        add(color);

        add(setLineHeight);

        add(italic);

        add(bold);

        addSeparator();

        add(makeFontNameSubmenu());

    }

    /\*\*

     \* Reset the state of the menu to reflect the default settings for text

     \* in a DrawPanel. (Sets the italic and bold checkboxes to unselected.)

     \* This method is called by the main program when the user selects the

     \* "New" command, to make sure that the menu state reflects the contents

     \* of the panel.

     \*/

    public void setDefaults() {

        italic.setSelected(false);

        bold.setSelected(false);

        // Set the default justification to "Left"

        leftJustify.setSelected(true);

        rightJustify.setSelected(false);

        centerJustify.setSelected(false);

    }

    /\*\*

     \* Create a menu containing a list of all available fonts.

     \* (It turns out this can be very messy, at least on Linux, but

     \* it does show the use what is available and lets the user try

     \* everything!)

     \*/

    private JMenu makeFontNameSubmenu() {

        ActionListener setFontAction = new ActionListener() {

            public void actionPerformed(ActionEvent evt) {

                panel.getTextItem().setFontName(evt.getActionCommand());

                panel.repaint();

            }

        };

        JMenu menu = new JMenu("Font Name");

        String[] basic = { "Serif", "SansSerif", "Monospace" };

        for (String f : basic) {

            JMenuItem m = new JMenuItem(f + " Default");

            m.setActionCommand(f);

            m.addActionListener(setFontAction);

            m.setFont(new Font(f, Font.PLAIN, 12));

            menu.add(m);

        }

        menu.addSeparator();

        String[] fonts = GraphicsEnvironment.getLocalGraphicsEnvironment().getAvailableFontFamilyNames();

        if (fonts.length <= 20) {

            for (String f : fonts) {

                JMenuItem m = new JMenuItem(f);

                m.addActionListener(setFontAction);

                m.setFont(new Font(f, Font.PLAIN, 12));

                menu.add(m);

            }

        } else { // Too many items for one menu; divide them into several sub-sub-menus.

            char ch1 = 'A';

            char ch2 = 'A';

            JMenu m = new JMenu();

            int i = 0;

            while (i < fonts.length) {

                while (i < fonts.length && (Character.toUpperCase(fonts[i].charAt(0)) <= ch2 || ch2 == 'Z')) {

                    JMenuItem item = new JMenuItem(fonts[i]);

                    item.addActionListener(setFontAction);

                    item.setFont(new Font(fonts[i], Font.PLAIN, 12));

                    m.add(item);

                    i++;

                }

                if (i == fonts.length || (m.getMenuComponentCount() >= 12 && i < fonts.length - 4)) {

                    if (ch1 == ch2)

                        m.setText("" + ch1);

                    else

                        m.setText(ch1 + " to " + ch2);

                    menu.add(m);

                    if (i < fonts.length)

                        m = new JMenu();

                    ch2++;

                    ch1 = ch2;

                } else

                    ch2++;

            }

        }

        return menu;

    }

    private void makeJustifyMenu() {

        JMenu justifyMenu = new JMenu("Justify");

        ButtonGroup justifyGroup = new ButtonGroup();

        leftJustify = new JRadioButtonMenuItem("Left");

        leftJustify.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent evt) {

                panel.getTextItem().setJustify(TextItem.LEFT);

                panel.repaint();

            }

        });

        justifyGroup.add(leftJustify);

        rightJustify = new JRadioButtonMenuItem("Right");

        rightJustify.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent evt) {

                panel.getTextItem().setJustify(TextItem.RIGHT);

                panel.repaint();

            }

        });

        justifyGroup.add(rightJustify);

        centerJustify = new JRadioButtonMenuItem("Center");

        centerJustify.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent evt) {

                panel.getTextItem().setJustify(TextItem.CENTER);

                panel.repaint();

            }

        });

        justifyGroup.add(centerJustify);

        justifyMenu.add(leftJustify);

        justifyMenu.add(rightJustify);

        justifyMenu.add(centerJustify);

        add(justifyMenu);

    }

    private void setLineHeightMultiplier() {

        String inputValue = JOptionPane.showInputDialog(panel, "Enter Line Height Multiplier (e.g., 1.5):");

        if (inputValue != null && !inputValue.trim().isEmpty()) {

            try {

                double multiplier = Double.parseDouble(inputValue.trim());

                if (multiplier > 0) {

                    panel.getTextItem().setLineHeightMultiplier(multiplier);

                    panel.repaint();

                } else {

                    JOptionPane.showMessageDialog(panel,

                            "Please enter a positive number for the line height multiplier.");

                }

            } catch (NumberFormatException e) {

                JOptionPane.showMessageDialog(panel, "Invalid input. Please enter a valid number.");

            }

        }

    }

}

package guidemo;

import java.applet.AudioClip;

import java.awt.Cursor;

import java.awt.Image;

import java.awt.Point;

import java.awt.Toolkit;

import java.awt.image.BufferedImage;

import java.net.URL;

import javax.imageio.ImageIO;

import javax.swing.ImageIcon;

import javax.swing.JApplet;

/\*\*

 \* This class provides some static utility functions for working

 \* with resources (to avoid having to look up all the messy details).

 \* Resources are stored somewhere on the class path, usually in their

 \* own package. They are located by paths to files, such as

 \* "resources/images/mandelbrot.jpeg".

 \*/

public class Util {

    /\*\*

     \* Load an image resource. In this case, the data will actually

     \* be read into memory only when the Image is first drawn.

     \*

     \* @param pathToResource the path to the resource.

     \* @return the image, or null if the resource can't be located.

     \*/

    public static Image getImageResource(String pathToResource) {

        ClassLoader cl = Util.class.getClassLoader();

        URL loc = cl.getResource(pathToResource);

        if (loc == null)

            return null;

        Image img = Toolkit.getDefaultToolkit().createImage(loc);

        return img;

    }

    /\*\*

     \* Load a buffered image from a resource. In this case, the method

     \* does not return until the image data has been read and stored

     \* in memory.

     \*

     \* @param pathToResource the path to the resource.

     \* @return the image, or null if the resource can't be loaded.

     \*/

    public static BufferedImage getBufferedImageResource(String pathToResource) {

        ClassLoader cl = Util.class.getClassLoader();

        URL loc = cl.getResource(pathToResource);

        if (loc == null)

            return null;

        try {

            return ImageIO.read(loc);

        } catch (Exception e) {

            return null;

        }

    }

    /\*\*

     \* Create an ImageIcon from an image that is stored as a resource.

     \*

     \* @param pathToResource the path to the resource.

     \* @return the ImageIcon, or null if the resource can't be located.

     \*/

    public static ImageIcon iconFromResource(String pathToResource) {

        Image img = getImageResource(pathToResource);

        if (img == null)

            return null;

        else

            return new ImageIcon(img);

    }

    /\*\*

     \* Play a sound that is stored as a resource file. If the resource

     \* can't be located or can't be played, no sound is played, and

     \* no exception is thrown.

     \*

     \* @param pathToResource the path to the resource.

     \*/

    public static void playSoundResource(String pathToResource) {

        try {

            ClassLoader cl = Util.class.getClassLoader();

            URL loc = cl.getResource(pathToResource);

            AudioClip sound = JApplet.newAudioClip(loc);

            sound.play();

        } catch (Exception e) {

            System.out.println("Can't play soucd " + pathToResource);

        }

    }

    /\*\*

     \* Load an AudioClip from a resource file. The clip can be played

     \* by calling its play() method.

     \*

     \* @param pathToResource the path to the resource.

     \* @return the audio clip, or null if the resource can't be loaded.

     \*/

    public static AudioClip getSound(String pathToResource) {

        ClassLoader cl = Util.class.getClassLoader();

        URL loc = cl.getResource(pathToResource);

        if (loc == null)

            return null;

        try {

            return JApplet.newAudioClip(loc);

        } catch (Exception e) {

            return null;

        }

    }

    /\*\*

     \* Create a cursor from an image, with hot point at the upper left

     \* corner (0,0).

     \*

     \* @param image the image; can't be null.

     \* @return a cursor that will show the image.

     \*/

    public static Cursor createImageCursor(Image image) {

        return createImageCursor(image, 0, 0);

    }

    /\*\*

     \* Create a cursor from an image resource file, with hot point at the

     \* upper left corner (0,0).

     \*

     \* @param pathToResource the path to the resource.

     \* @return the cursor or, if the resource can't be loaded, the

     \*         default cursor.

     \*/

    public static Cursor createImageCursor(String pathToResource) {

        return createImageCursor(pathToResource, 0, 0);

    }

    /\*\*

     \* Create a cursor from a resource file, with hot point at

     \* (hotSpotX, hotSpotY).

     \*

     \* @param pathToResource the path to the resource.

     \* @return a cursor that will show the image.

     \*/

    public static Cursor createImageCursor(String pathToResource, int hotSpotX, int hotSpotY) {

        Image img = getImageResource(pathToResource);

        if (img == null)

            return Cursor.getDefaultCursor();

        else

            return Toolkit.getDefaultToolkit().createCustomCursor(

                    img, new Point(hotSpotX, hotSpotY), pathToResource);

    }

    /\*\*

     \* Create a cursor from an image, with hot point at

     \* (hotSpotX, hotSpotY).

     \*

     \* @param image the image; can't be null.

     \* @return a cursor that will show the image.

     \*/

    public static Cursor createImageCursor(Image image, int hotSpotX, int hotSpotY) {

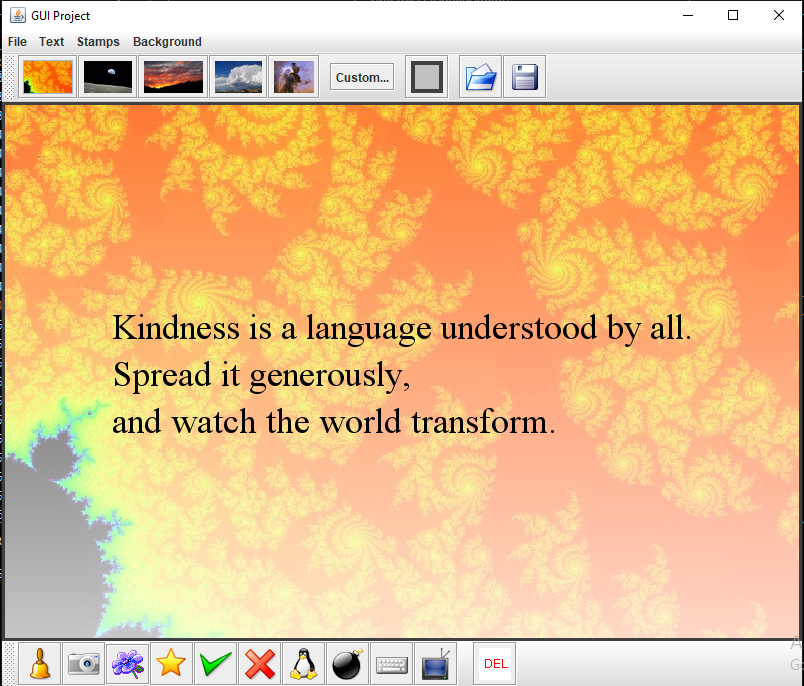
        return Toolkit.getDefaultToolkit().createCustomCursor(

                image, new Point(hotSpotX, hotSpotY), null);

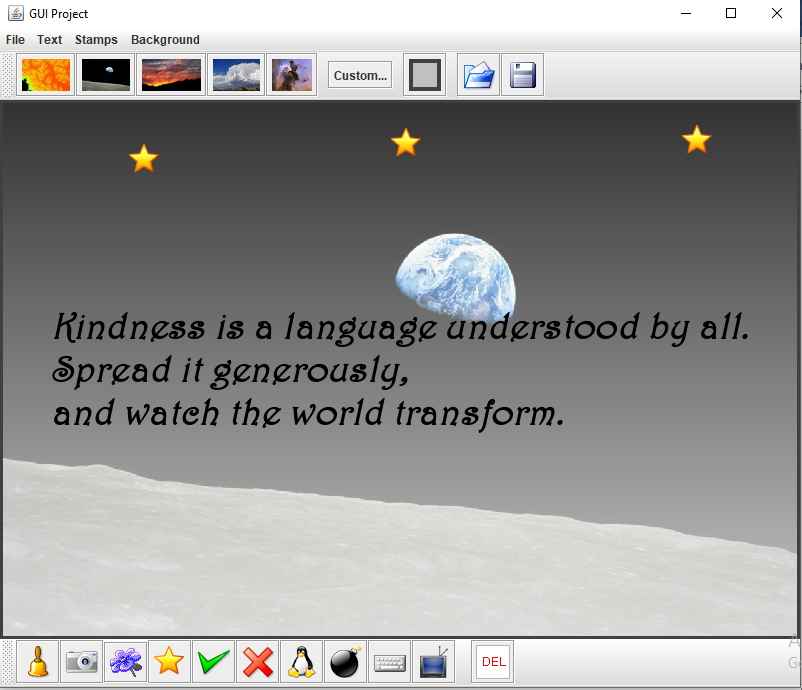
    }

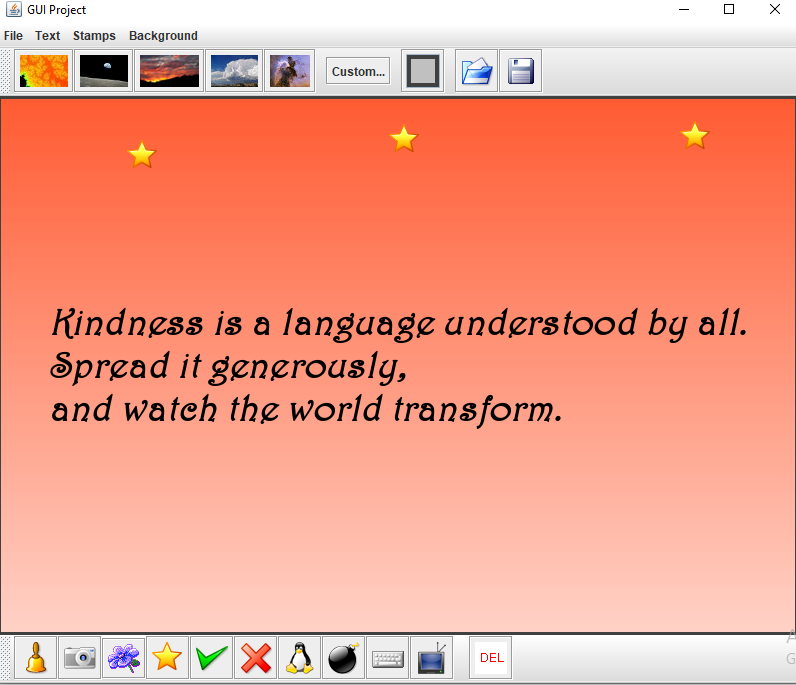
}

**8. Output**









**The End**