Comments from the Instructor

Here are four good portfolios. They have all provided a good overview.
Group A and B have done a real good job mapping their work to the three objectives that I had listed:
1) Interaction with materials
2) Working with more complex issues
3) Working at higher levels of Blooms taxonomy

The table of contents of these teams are really helpful- so please include it in your overview.

It is important to provide your SOURCE code with your submission. I should be able to run it AND also view the code. i.e. submit a zip file with your src code AND your overview file.

I am redacting any student info from the following portfolios (for student privacy reasons). If I missed any, please excuse me!

Do note that each team has explored topics that were not discussed in class.
Also, note that there are other teams (and individuals) that have done really well - but I am just including a few here.

Many Thanks,
Simanta

PORTFOLIO I

ComS 319 – Fall 2014

[redacted]

Porfolio I

Table of Contents

Introduction

This portfolio demonstrates the learning expectations of Java Swing through a program called JSweeper, Minesweeper in Java. JSweeper provides new insight and answers questions about Java Swing. The basic subjects include layout managers, event handling, and rendering elements. JSweeper also contains many complexities including thread handling, observer framework, and MVC framework with JTable. The higher levels of Bloom's Taxonomy are demonstrated by JSweeper though its creation, analysis, and evaluation.

Interaction with Materials

Insights & Questions

How do I send events between a JTable and JButton (Observer Pattern)?

How do I receive multiple inputs from a dialog window?

How do I format cells in a JTable (Custom Cell Renderer)?

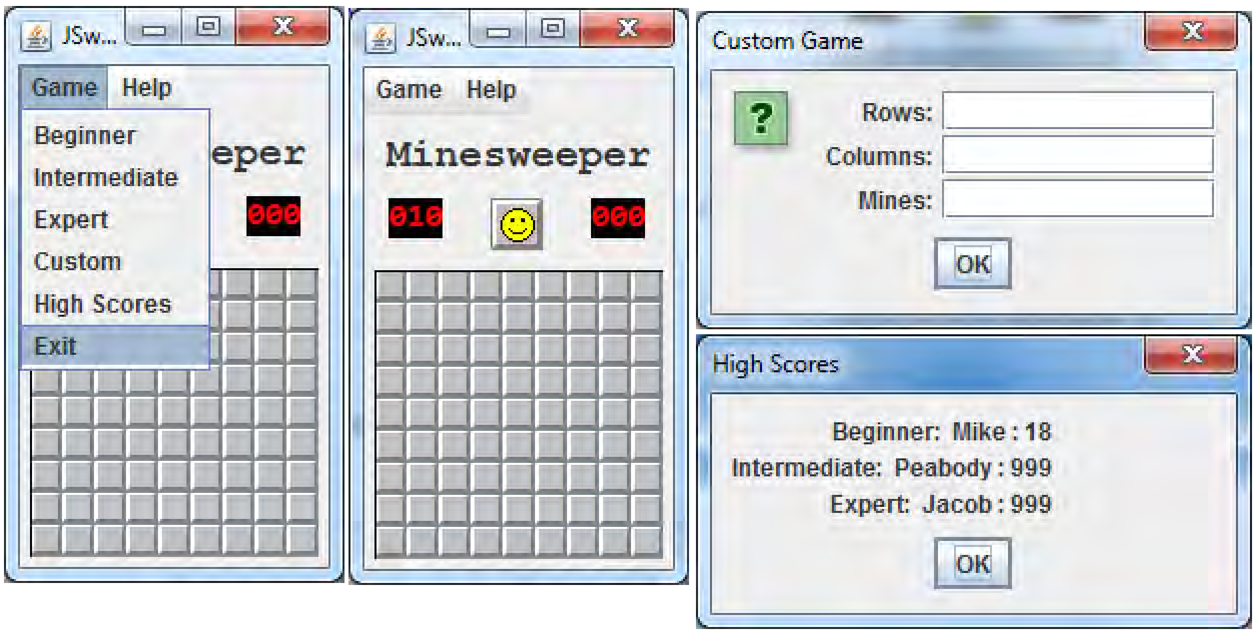
How do save custom data in a JTable (Custom Table Model)?

Where do I put my business logic (MVC)?

How do I reinitialize base JFrame?

How do I use the Swing Timer element?

Figure 1: Game Menu Screens



```
Beginner:  Mike : 18
Intermediate:  Peabody : 999
Expert:  Jacob : 999
```

User Interface Elements

JSweeper demonstrates a profound amount of interaction with the Java Swing user interface (UI) elements. It uses many different layouts, element types, and panels which make up our view. There are three main parts to the view which are the game menu, status elements, and minefield.

Game Menu

The game menu uses JMenuBar, JMenu, JMenuItem, and JOptionPane. The game difficulty buttons each have action listeners that fire to update the difficulty setting in the model. The custom difficulty option shows a message dialog that has inputs for field size and number of mines. The high scores menu item uses an action listener to read a text file containing the quickest match for each level of difficulty. The file is written to whenever a game is successfully completed using any game mode except custom.

Figure 2: Difficulty Button (Beginner)

```java
JMenu gameMenu = new JMenu("Game");
JMenuItem beginnerItem = new JMenuItem("Beginner");
beginnerItem.addActionListener(new ActionListener() {
	@Override
	public void actionPerformed(ActionEvent arg0) {
		model.getMinefieldModel().setDifficulty(Difficulty.Beginner);
	}
});
gameMenu.add(beginnerItem);
```

Figure 3: Difficulty Button (Custom)

```java
JMenuItem customItem = new JMenuItem("Custom");
customItem.addActionListener(new ActionListener() {
	@Override
	public void actionPerformed(ActionEvent arg0) {
		JPanel custom = new JPanel(new BorderLayout(5, 5));

		JPanel labels = new JPanel(new GridLayout(0, 1, 2, 2));
		labels.add(new JLabel("Rows:", SwingConstants.RIGHT));
		labels.add(new JLabel("Columns:", SwingConstants.RIGHT));
		labels.add(new JLabel("Mines:", SwingConstants.RIGHT));
		custom.add(labels, BorderLayout.WEST);

		JPanel controls = new JPanel(new GridLayout(0, 1, 2, 2));

		JTextField rows = new JTextField();
		controls.add(rows);

		JTextField cols = new JTextField();
		controls.add(cols);

		JTextField mines = new JTextField();
		controls.add(mines);

		custom.add(controls, BorderLayout.CENTER);

		JOptionPane.showMessageDialog(null, custom, "Custom Game",
				JOptionPane.QUESTION_MESSAGE);

		model.getMinefieldModel().setDifficulty(Difficulty.Custom,
				Integer.parseInt(rows.getText()),
				Integer.parseInt(cols.getText()),
				Integer.parseInt(mines.getText()));
	}
});
gameMenu.add(customItem);
```

Status Elements

The status elements interact with the minefield to help give information about the current game to the user. The game control button, the middle element, is a JFaceButton, child of JButton, which implements JFaceButtonListener. Using the observer pattern, it listens for four types of events, lose, win, mouse-down, and reset. The button has two separate pressed icons for when the user presses the field and when the user presses the button itself. The two events mouse-down and reset are generated by the minefield when the user clicks down and releases click from the minefield.

Both elements to the left and right of the JFaceButton extend JSweeperLabel which is a customized JLabel. The flag counter, placed to the left, displays the number of mines remaining to be flagged. The element is a JFlagCounter which implements FlagCounterListener to know when tiles are flagged on the minefield. The timer, placed to the right, displays a timer that counts up.

Figure 4: Game Control Button

```
/**
 * JFaceButton displays game status via its icon and can reset the game if
 * clicked on.
 *
 * @author [redacted]
 */
public class JFaceButton extends JButton implements JFaceButtonListener {
	/**
	 * Constructs the JFaceButton.
	 */
	public JFaceButton() {
		super();
		setAlignmentX(Component.CENTER_ALIGNMENT);
		setBorderPainted(false);
		setBorder(null);
		setMargin(new Insets(0, 0, 0, 0));
		setContentAreaFilled(false);
		setIcon(JFaceButtonIcon.Default.getIcon());
		setPressedIcon(JFaceButtonIcon.Pressed.getIcon());
	}

	@Override
	public void gameEnded() {
		setIcon(JFaceButtonIcon.Lose.getIcon());
	}

	@Override
	public void mouseDown() {
		setIcon(JFaceButtonIcon.OFace.getIcon());
	}

	@Override
	public void reset() {
		setIcon(JFaceButtonIcon.Default.getIcon());
	}

	@Override
	public void gameWon() {
		setIcon(JFaceButtonIcon.Win.getIcon());
	}
}
```

Minefield

The minefield is a JMinefield, child of JTable. To create the minefield, the default cell renderer is replaced with JMinefieldCellRenderer. The renderer gets the game tile at the cell and returns a JLabel with the icon set as the game tile's icon. The table cells each have a width and height of 16 pixels as well. It uses a JMinefieldModel, which extends AbstractTableModel, as its model. The model stores the grid of game tile objects, number of mines, number of flags placed, time and difficulty. The model is constructed with a difficulty parameter which determines the size of the field along with the amount of mines. The model has GameSettingsListeners, implemented using the observer pattern, which send alerts when settings like difficulty change. Since the model extends AbstractTableModel, it also has many inherited events such as table changed or cell updated. These events are handled by the JMinefieldController.

Figure 5: JMinefield (extended JTable)

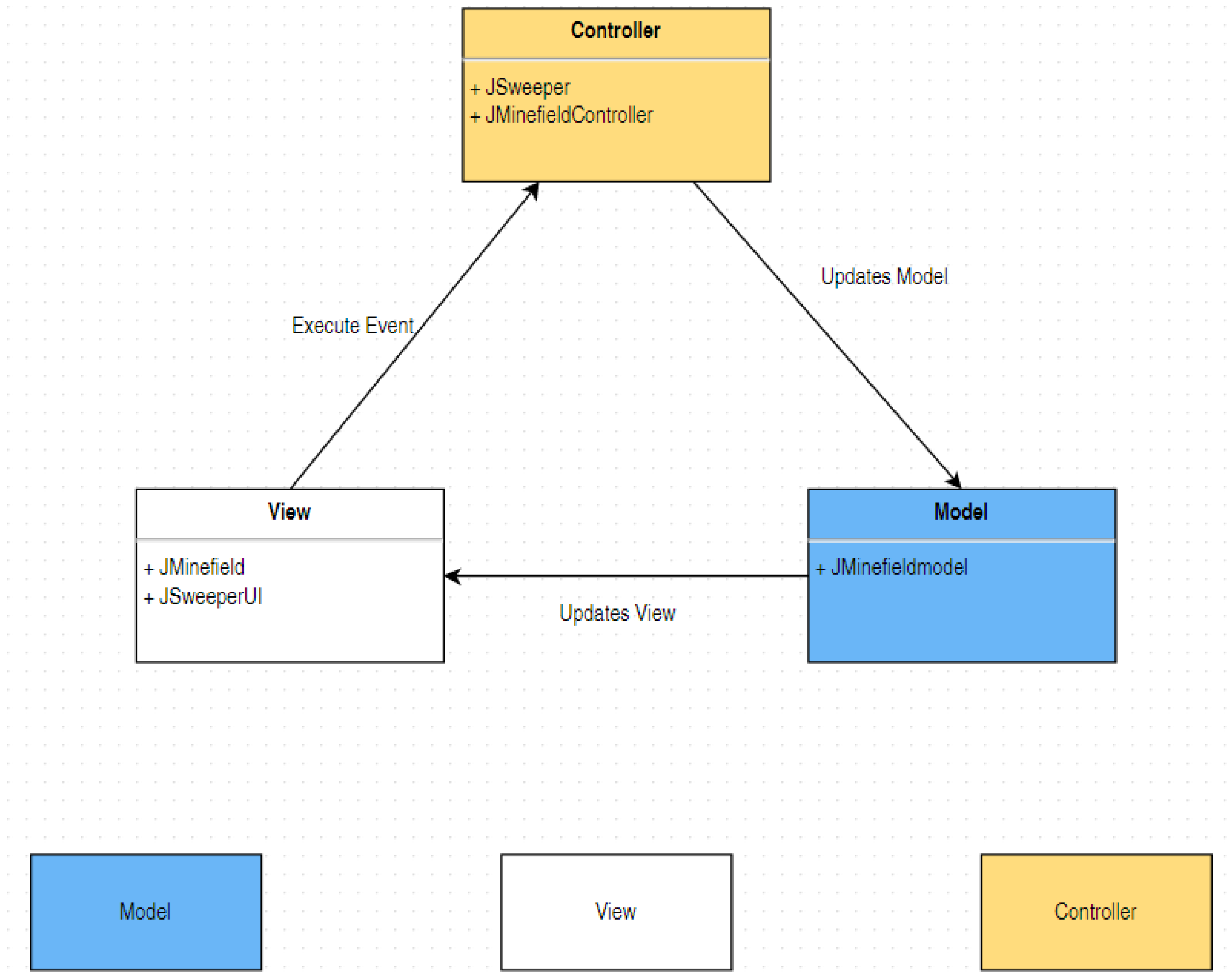
```java
/**
 * This represents the view of the minefield that the main game is played on.
 *
 * @author [redacted]
*/
public class JMinefield extends JTable {
   /**
    * Constructs the view of the minefield, given a JMinefieldModel. Formats
    * the table so it can look and feel like a Minesweeper field.
    *
    * @param model
    */
   public JMinefield(JMinefieldModel model) {
      super(model);
      setBorder(new MatteBorder(1, 1, 0, 0, Color.BLACK));
      setRowHeight(16);
      setFillsViewportHeight(true);
      setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
      selectionModel.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
      rowSelectionAllowed = false;

      for (int x = 0; x < getColumnCount(); x++) {
            getColumnModel().getColumn(x).setMaxWidth(16);
            getColumnModel().getColumn(x).setCellRenderer(
                        new JMinefieldCellRenderer());
      }
   }

   // overrides the DefaultTableCellRenderer so that it displays the icons
   // within JTiles
   private class JMinefieldCellRenderer extends DefaultTableCellRenderer {
      public Component getTableCellRendererComponent(JTable table,
                  Object value, boolean isSelected, boolean hasFocus, int row,
                  int col) {
            JTile tile = (JTile) value;
            JLabel label = new JLabel();
            label.setIcon(tile.getIcon());
            return label;
      }
   }
}
```

Complex Topics

Model-View-Controller (MVC) Pattern

Developing JSweeper with the MVC pattern was very difficult and involved multiple rounds of refactoring. The JSweeper class is the main controller which spawns the system frame into a new thread, letting the main thread die. It then creates the JMinefieldController with is responsible for controlling the JMinefield (extends JTable). The JMinefieldController instantiates a JMinefieldModel and JMinefield. Lastly the main controller creates the main view and renders it into the system frame. The main view contains all of the panels and layouts for organzing the user-interface. It also places the JMinefield into the UI along with the menu and status elements.

Figure 6: MVC Flow Diagram



Bloom’s Taxonomy

Analysis

Distinguishing between model-view-controller was major part of analysis. We needed to differentiate controller logic from the model and view. Examining what code controlled data and visuals and separating them into respective classes was critical. We experimented with threads to figure out how to reset the game. The end solution was to add an event the event handler thread to reset the game in the main controller.

Evaluation

Evaluating when to implement the observer pattern was a major part of evaluation. We decided that we needed to implement GameSettingsListener even though the AbstractTableModel already has listeners because we wanted to separate the controller events from the view events. The AbstractTableModel events are used in by the controller while the GameSettingsListener events are used in the view.

When creating the mouse events for the JMinefieldController, we had to judge what actions needed to be called on each click. In minesweeper, almost all of the interface is done through types of clicks which created a complex click event handler.

Creation

- JSweeper itself is a complete Java Swing application created from scratch
- Categorized minesweeper operations as controller actions, view visuals, and model data
- Identified objects to store game elements including icons, tiles, and buttons
- Formulated observer patterns to relay information between the controllers and views
- Generated threads to handle timer events and the system frame
- Designed user interface of the view to look similar to the classic Microsoft Minesweeper
- Planned out how to combine all mouse events into one set of succinct logic
- Composed a model to store all data for the game that only stores data (no business logic)
- Implemented many event listeners for the field, button, timer, game status, etc.

Examples

Figure 7: Timer Thread

```
JPanel timerPanel = new JPanel(new FlowLayout());
JTimerLabel timerLabel = new JTimerLabel();
timerPanel.add(timerLabel);
Thread timerThread = new Thread(new Runnable() {
   public void run() {
      while (true) {
            timerLabel.updateTextNumeric(model.getTime());
            try {
                  Thread.sleep(500);
            } catch (InterruptedException e) {
                  // Should never happen
                  System.out.println("Timer update thread interrupted. Error!");
                  System.exit(-1);
            }
      }
   }
});
timerThread.start();
bodyPanel.add(timerPanel);
```

Figure 8: MinefieldMouseAdapter – Mouse is released on the JMinefield

```
private class MinefieldMouseAdapter extends MouseAdapter {
   // Handles what happens when the mouse is pressed on the JMinefield
   public void mousePressed(MouseEvent e) {

      // Don't do anything if the game is over.
      if (gameOver) {
            return;
      }

      // get which button was pressed
      switch (e.getButton()) {
      case 1:
            // left click
            leftClickDown = true;
            for (JFaceButtonListener btn : faceButtonListeners)
                  btn.mouseDown();
            break;
      case 3:
            // right click
            rightClickDown = true;
            break;
      default:
            break;
      }
   }

   // Handles what happens when the mouse is released on the JMinefield.
   public void mouseReleased(MouseEvent e) {
      // Don't do anything if the game is over.
      if (gameOver) {
            return;
      }

      // get the row and column where the mouse was released
      int row = view.rowAtPoint(e.getPoint());
      int col = view.columnAtPoint(e.getPoint());

      // Simultaneous click
      if (leftClickDown && rightClickDown) {
            leftClickDown = false;
            rightClickDown = false;

            // call the twoClicks helper function
            try {
                  twoClicks(row, col);
            } catch (GameOverException e1) {
                  return;
            }

            // reset the JFaceButton to the default smiley face
            for (JFaceButtonListener btn : faceButtonListeners)
                  btn.reset();
            return;
      }
```

```
        // get which button was pressed
        switch (e.getButton()) {
        case 1:
                // left click
                leftClickDown = false;

                // call the left click helper function
                leftClick(row, col);
                break;
        case 3:
                // right click
                rightClickDown = false;

                // call the right click helper function
                rightClick(row, col);
                break;
        default:
                break;
        }

        // update the view
        model.fireTableCellUpdated(row, col);
    }
}
```

Portfolio I: [redacted]

By [redacted]

Overview

The goal of this first portfolio was to gain a better understanding of Java Swing. For everyone in our group, we had never used Swing before this class and it would be a learning experience for all of us. Despite having never used Swing before, we were able to easily learn the syntax and patterns of Swing from various tutorials. In order to fully pierce the surface of what Swing could do, we experimented in coding various simple applications. After two weeks of practice, we had a strong understanding of the basic components of Swing. The next step was to try to work with the more complex features of Swing that were not covered in class. Our goal was create some cool applications using the advanced features of Swing.

Table of Contents

Section I:
Evidence of Interaction with Materials

In labs 1 and 2, we were exposed to labels, frames, various button types, input fields, tables, and many other Swing components. The applications we created in labs using these features were simple but a great learning experience for us too. The labs gave us a good understanding of what we could design given the tools provided by Swing. For our portfolio, we used what we learned in lab and built on that by adding more complex components to our applications. A few specific goals we wanted to accomplish were:

1. Learn to implement and use the JSpinner and JSlidder components.
2. Create an application that effectively uses the JMenuBar.
3. Have a pop-up window that displayed a functioning progress bar.
4. Differentiate between when to use the DefaultTableModel and when to implement AbstractTableModel.
5. Create a table with another component inside of it.
6. Incorporate non-Swing libraries or features that interact with our applications.
7. Add HTML to Java Swing for formatting

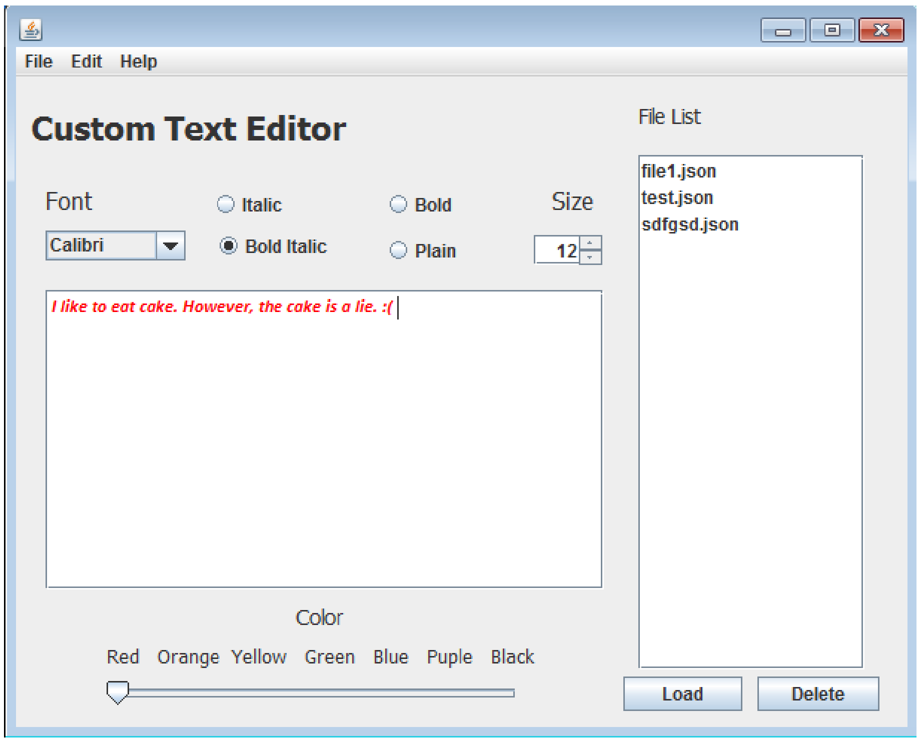
Section II:
Exploration of More Complex Issues

Experiment 1: Text Editor Application

A. Goals of Text Editor Application:

One application that we created was a custom text editor, similar to Notepad or Notepad++. We mostly wanted to experiment with various components of Swing and figure out what we could do. There were no immediate questions we wanted to answer when creating the application, but rather, we wanted to the figure out: “what can we do?” A simple acceptance criteria was devised for the application which would have the following features:

1. Ability to change the font, color, style, and size of the text on screen.
2. Contain and editable text field that a user can type into.
3. Save the text on screen into a file, along with the correct text formatting.
4. Load a saved file that the user created.
5. Delete a file that the user created.
6. Perform a replace or replace all on the text.
7. Perform a find operation on the text



Font: Calibri

Italic [✓ Bold Italic] Bold Plain

Size: 12

I like to eat cake. However, the cake is a lie. :(

file1.json
test.json
sdfgsd.json

We were able to successfully design and create the application given the desired features. Below is a screenshot of the completed application while running:

B. Description of Functionality of Text Editor Application:
The main part of the application is an editable text box where a user can type any text. A JSlidder was used to change the color of the text and a JSpinner to change the text's size.

The available fonts of the text are shown in a combobox. The list of fonts is imported from a text file and the combobox fields are generated from it. Code from the first lab exercise was reused to create the radio buttons that change the style of the text.

The "File" button on the top menu bar gives the user the options of saving the file or exiting the application. If the save option is chosen, a pop-up window will appear with a prompt to type a name. If the name already exists or has an error with it, the prompt will reappear. Saving the file creates a JSON object file to store the relevant data of the text on screen. It saves the font, color, style, size, and text on screen into a .json file. The file will then appear in the File List on the main page of the application.

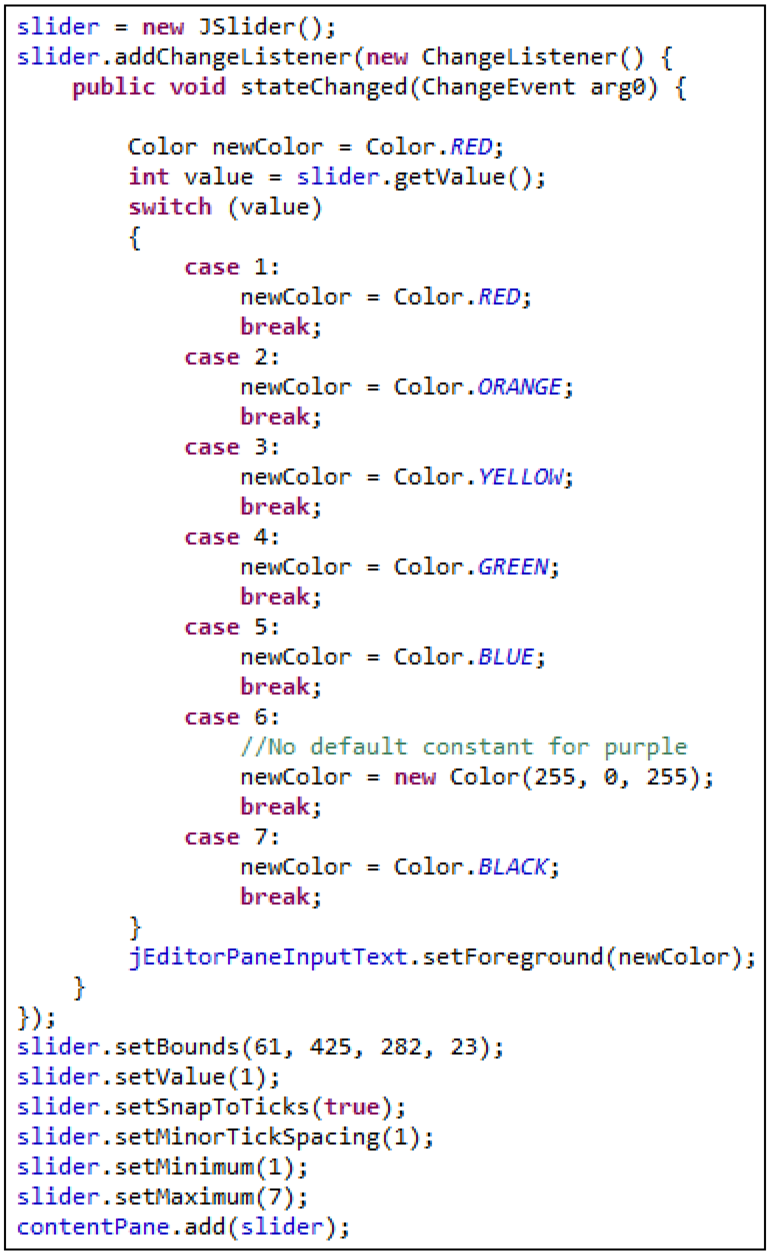
The "Edit" button on the top menu bar gives the user the options of finding or replacing the text on screen. If the replace option is chosen, a pop-up window will appear with a prompt to type "Text To Replace" and "Replacement Text." After the replace completes, it will notify the user how many occurrences were replaced. If the find option is chosen, a pop-up window will appear with a prompt to type a word to find. After the find completes, each instance of the found word will be highlighted with grey. In Java terms, the foreground of text was set to the highlighted color.

The "Help" button on the top menu bar opens a scrollable pop-up window. The window contains detailed explanations about the functionality of the application. It also provides comments on how to use the various features along with their limitations.

The File List is a single selection list of Files created by the application. The load button will get the selected file and load its contents onto the screen. The current text on the screen will be deleted and replaced with the contents of the loaded file. All of the various text modifier buttons, spinner, slider, and etc. will be set according to the parameters of the loaded file. Any initial text or formats that existed before the load will be lost if they were not saved. The delete button will delete a file from the list. If that file's text was being displayed on the editor, that text will persist along with its formatting.

C. Code Segments of Text Editor Application:

Below is code segment for the JSlider that was created for the application. It has a minimum value of 1 and a max value of 7. By setting the minor tick spacing to 1, it forces the component to have 7 possible positions for the slider to be located at. A simple switch and case statement is used to appropriately set the text color based on the current value of the slider. The method setSnapToTicks(true) ensures that the slider will be at one of those values and it improves the aesthetic value of the component too.



```java
slider = new JSlider();
slider.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent arg0) {

        Color newColor = Color.RED;
        int value = slider.getValue();
        switch (value)
        {
            case 1:
                newColor = Color.RED;
                break;
            case 2:
                newColor = Color.ORANGE;
                break;
            case 3:
                newColor = Color.YELLOW;
                break;
            case 4:
                newColor = Color.GREEN;
                break;
            case 5:
                newColor = Color.BLUE;
                break;
            case 6:
                //No default constant for purple
                newColor = new Color(255, 0, 255);
                break;
            case 7:
                newColor = Color.BLACK;
                break;
        }
        jEditorPaneInputText.setForeground(newColor);
    }
});
slider.setBounds(61, 425, 282, 23);
slider.setValue(1);
slider.setSnapToTicks(true);
slider.setMinorTickSpacing(1);
slider.setMinimum(1);
slider.setMaximum(7);
contentPane.add(slider);
```

Experiment 2: Advanced Tables

When we started the portfolio project, we had several questions about tables:

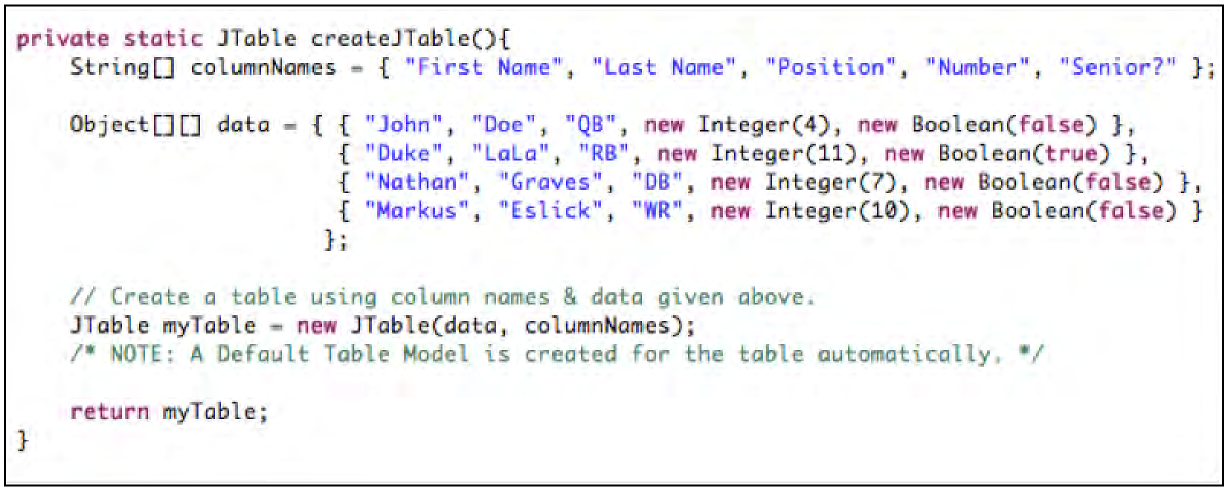
- Can we make tables without a table model?
- What are the advantages of table models?
- What is the difference between a DefaultTableModel and implementing the AbstractTableModel?
- Which type of model is better?
- Is it possible to put other components inside of the table?

We also wanted to improve our understanding of JTables. We did this using a series of JTable applications that demonstrate various aspects of tables in Swing.

A. DefaultTableModel

As we started investigating, we discovered that every JTable requires a table model. If a TableModel is not specified, the DefaultTableModel is automatically used. Table models store the content of the cells as well as other pertinent information about the table, such as column names and how to render each cell.

The method below demonstrates how to create a JTable using a string array of the column names and an array of Object arrays (2D array) that contains the contents of the table. When the new JTable is constructed, a DefaultTableModel is automatically created for the table.

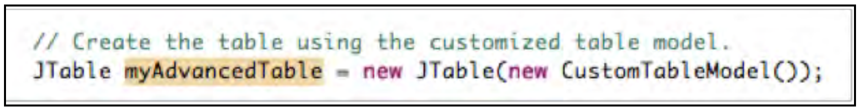


```java
private static JTable createJTable(){
    String[] columnNames = { "First Name", "Last Name", "Position", "Number", "Senior?" };

    Object[][] data = { { "John", "Doe", "QB", new Integer(4), new Boolean(false) },
                        { "Duke", "LaLa", "RB", new Integer(11), new Boolean(true) },
                        { "Nathan", "Graves", "DB", new Integer(7), new Boolean(false) },
                        { "Markus", "Eslick", "WR", new Integer(10), new Boolean(false) }
                      };

    // Create a table using column names & data given above.
    JTable myTable = new JTable(data, columnNames);
    /* NOTE: A Default Table Model is created for the table automatically. */

    return myTable;
}
```

B. AbstractTableModel

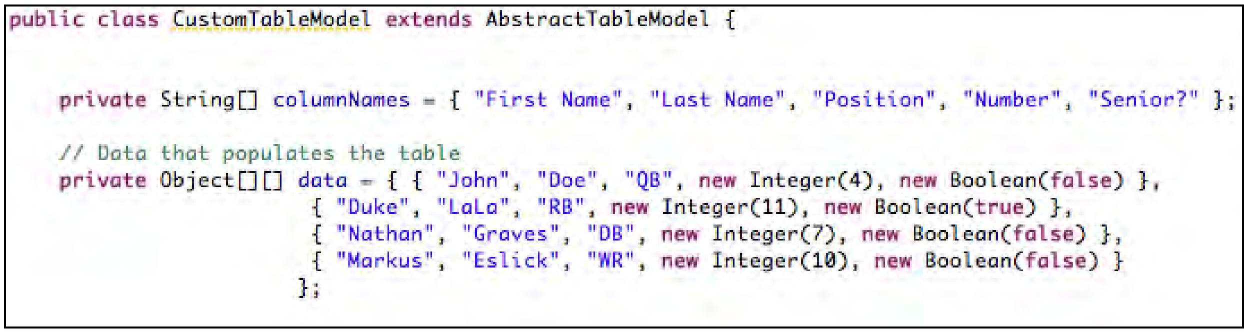
The DefaultTableModel is convenient for quickly creating tables, but doesn't allow for much flexibility. Cell contents are displayed as text and every cell is editable. Fortunately, Java Swing provides us with a way of creating more customizable tables. Instead of relying on the DefaultTableModel, we can create a custom table model that implements the AbstractTableModel class.

The code snippet below creates a table that uses CustomTableModel, a table model that extends AbstractTableModel.



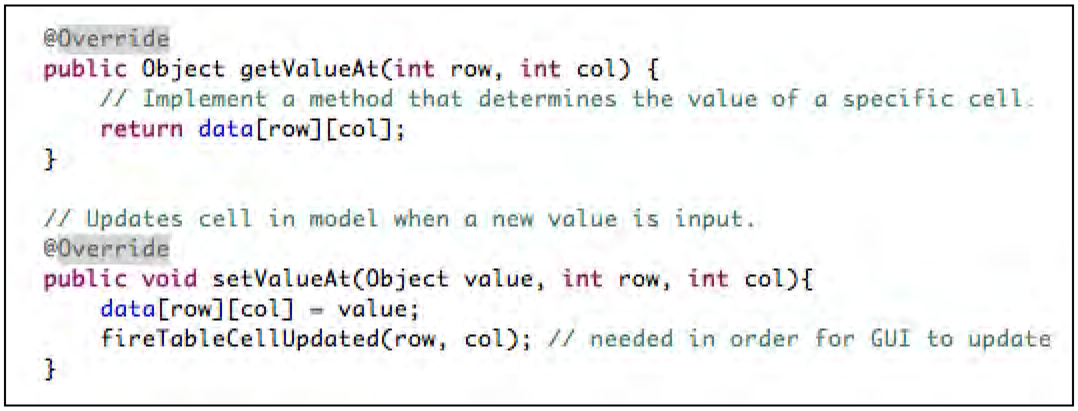
```
// Create the table using the customized table model.
JTable myAdvancedTable = new JTable(new CustomTableModel());
```

Below is the beginning of the CustomTableModel class, an example implementation of AbstractTableModel. Notice that the data is stored in a 2D array for this particular class, but how it is stored is completely up to the programmer. For example, the data could be retrieved from a file or a database instead.



```
public class CustomTableModel extends AbstractTableModel {


    private String[] columnNames = { "First Name", "Last Name", "Position", "Number", "Senior?" };

    // Data that populates the table
    private Object[][] data = { { "John", "Doe", "QB", new Integer(4), new Boolean(false) },
                                { "Duke", "LaLa", "RB", new Integer(11), new Boolean(true) },
                                { "Nathan", "Graves", "DB", new Integer(7), new Boolean(false) },
                                { "Markus", "Eslick", "WR", new Integer(10), new Boolean(false) }
                              };
```

The *getValueAt* and *setValueAt* methods both need to be implemented so that the data is stored or retrieved from the correct location. When a new value is set, you must also fire an event to let the GUI know that the model has been updated.



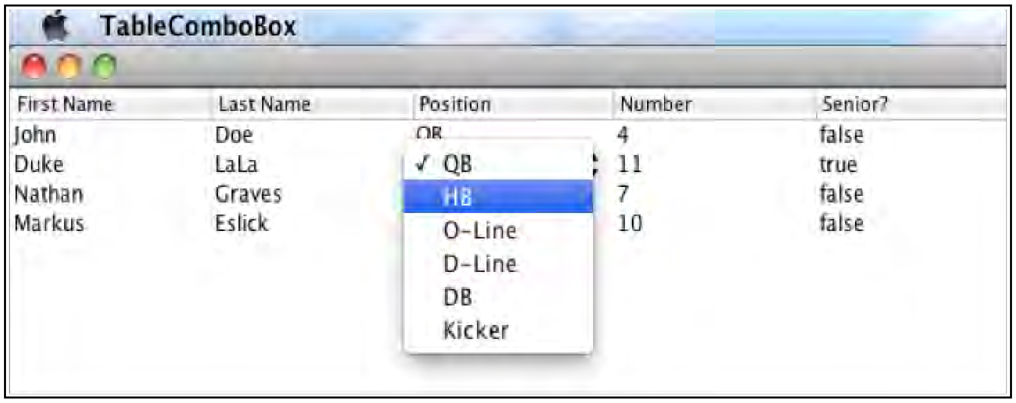
```
@Override
public Object getValueAt(int row, int col) {
    // Implement a method that determines the value of a specific cell.
    return data[row][col];
}

// Updates cell in model when a new value is input.
@Override
public void setValueAt(Object value, int row, int col){
    data[row][col] = value;
    fireTableCellUpdated(row, col); // needed in order for GUI to update
}
```

Other methods such as *getColumnCount, getRowCount, getColumnName,* and *getColumnClass* also need to be overridden so they return accurate information. Which cells the user can edit is controlled by the *isCellEditable* method. If you want all cells to be editable (like the DefaultTableModel) simply return true. In the example code below, all cells besides the first two columns are editable.

```
// Indicates which cells are editable.
@Override
public boolean isCellEditable(int row, int col){
    // Allow any fields besides first & last name to be edited.
    if (col < 2){
        return false;
    } else { // Any row besides row 0 and row 1.
        return true;
    }
}
```

C. Using a ComboBox in a Table

Adding components inside of a table can make the GUI more user-friendly. We chose to demonstrate how to insert a ComboBox into a table because it is common for data cells to have a limited number of choices. For our example, the user selects a person's position from a list. The resulting GUI table is pictured below.



First Name	Last Name	Position	Number	Senior?
John	Doe	QB	4	false
Duke	LaLa		11	true
Nathan	Graves		7	false
Markus	Eslick		10	false

The code below demonstrates how to add a JComboBox into a table. First, create a ComboBox and add all of the choices that can be selected. Next, set the ComboBox as the editor for the desired column (column 2 in this example).

```
// Create combo box and add selection choices.
JComboBox posComboBox = new JComboBox();
posComboBox.addItem("QB");
posComboBox.addItem("HB");
posComboBox.addItem("O-Line");
posComboBox.addItem("D-Line");
posComboBox.addItem("DB");
posComboBox.addItem("Kicker");

// Specify the position column
TableColumn positionCol = myTable.getColumnModel().getColumn(2);

// Switch the editor to be the newly created combo box.
positionCol.setCellEditor(new DefaultCellEditor(posComboBox));
```

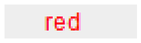
Experiment 3: Implementing HTML into Java Swing for formatting

When considering formatting in swing a few questions came up:

- How do you use more than one color?
- How do you use more than one font?
- Does all the text inside a button or a jLabel have to be one specific formatting option?
- Can you add images in line with your text

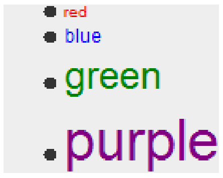
As it turns out, many swing components use text strings as a part of their GUI, and by default a components text is displayed in a single font and color, all on one line. If you want to mix and match fonts or colors, along with extra formatting options you will need to use html. There are a few ways to do this, the first way is to simply put the html right into the component. As an example here is a JLabel html declaration along with its output.

```
theLabel = new JLabel("<html><font color=red>red</font></html>");
```


red

Now changing the color of the font to red is not that impressive of a feat, but the possibilities are as large as the limitations of html itself. As another example, if you wanted to make a label with bullets of all different colors and sizes you could do this.

```
theLabel = new JLabel("<html><ul><li><font color=red><font size =-2>red</font></font>"
        + "<li><font color=blue><font size =-1>blue</font></font>"
        + "<li><font color=green><font size =6>green</font></font>"
        + "<li><font color=purple><font size =10>purple</font></font></ul></html>");
```


- red
- blue
- green
- purple

As a second way to use an html, you can import an html file whether it be from a website or one that you generate yourself. As an example, we placed an html file into a java url and then used it as a declaration for an editor pane. Using the following code editor pane would look just like the "website" that is htmlfile.html.

```
java.net.URL helpURL = TextSamplerDemo.class.getResource(
                                "htmlfile.html");
editorPane.setPage(helpURL);
```

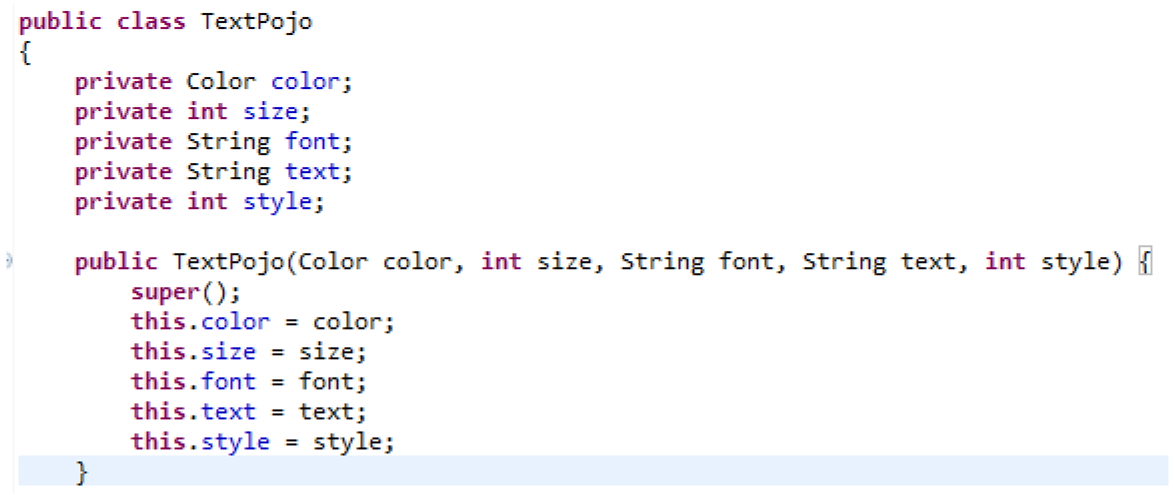
Using html can further advance the look and feel of your java GUI, whether it is to help with formatting your tables as in experiment 2, or to add an image into a button you only need to type the html code that will do it.

Section III:
Evidence of Working at Higher Levels of Bloom’s Taxonomy

Experiment 1: Complex Design Issues of Text Editor Application:

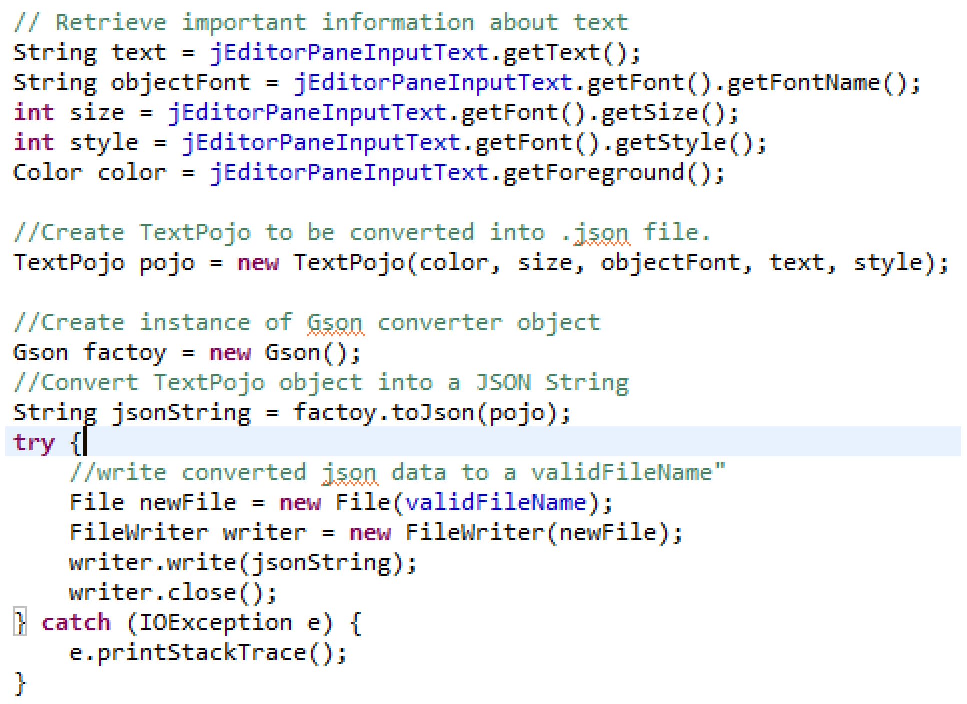
One major design issue that we came across was saving the font, style, size, and color of the text. The original idea was to save the information in a “.txt” file and put the font information either at the bottom or top of the file. A scanner object would then iterate through the File and parse out the relevant information. The scanner would need to be smart enough to differentiate which text was user typed and which text segments contained the formatting information. This would have been difficult to implement and may not have worked 100% of the time. The second option was to use HTML to save the formatting. This approach was nicer but still required some complex ideas and was not right for this scenario. This is the point where we decided to do a third experiment and just focus on HTML and see how it would work.

We decided to go with a third approach of saving the information in a “.json” file which was a simple solution. A helper object (TextPojo) was created to store the relevant information about the text.



```java
public class TextPojo
{
    private Color color;
    private int size;
    private String font;
    private String text;
    private int style;

    public TextPojo(Color color, int size, String font, String text, int style) {
        super();
        this.color = color;
        this.size = size;
        this.font = font;
        this.text = text;
        this.style = style;
    }
```

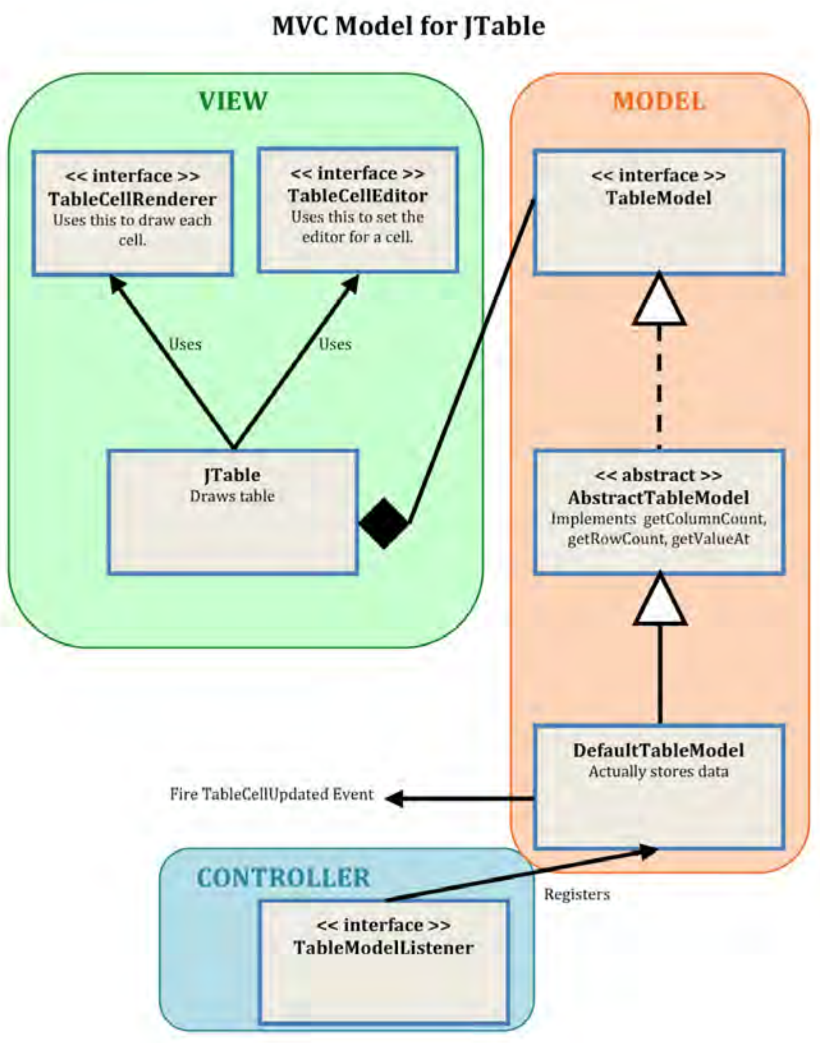
The class was a basic Java object with variables to store the data and had getter/setter methods. We used an external library, GSON, which is a Google library for parsing JSON objects. It is able to convert the TextPojo object into a .json file, along with converting a .json file into a TextPojo object. The data from the TextPojo object is used to set and retrieve the necessary text and font information. Below is part of the code for the action after the save button is pressed



```
// Retrieve important information about text
String text = jEditorPaneInputText.getText();
String objectFont = jEditorPaneInputText.getFont().getFontName();
int size = jEditorPaneInputText.getFont().getSize();
int style = jEditorPaneInputText.getFont().getStyle();
Color color = jEditorPaneInputText.getForeground();

//Create TextPojo to be converted into .json file.
TextPojo pojo = new TextPojo(color, size, objectFont, text, style);

//Create instance of Gson converter object
Gson factoy = new Gson();
//Convert TextPojo object into a JSON String
String jsonString = factoy.toJson(pojo);
try {
    //write converted json data to a validFileName"
    File newFile = new File(validFileName);
    FileWriter writer = new FileWriter(newFile);
    writer.write(jsonString);
    writer.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

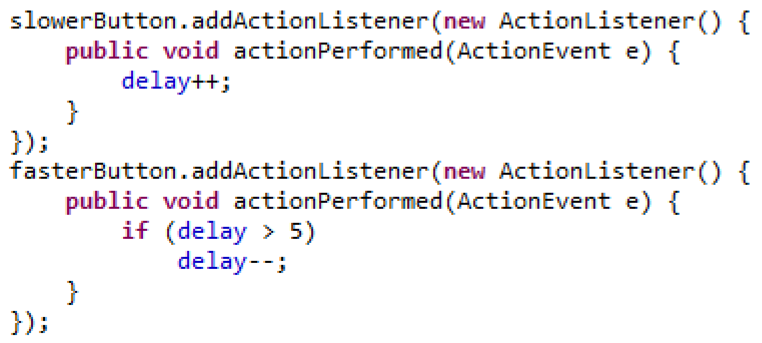
Experiment 2: Advanced Table Models

Our JTable experiments were designed to evaluate the differences between the DefaultTableModel and AbstractTableModel. In order to compare the two, you must first understand how TableModels interact with the table and how that table is displayed in the GUI. We created the Model-View-Controller Diagram below to illustrate how the different parts interact. The View is made up of the JTable itself, which uses a TableCellRenderer to draw each cell on the GUI and TableCellEditors to edit cells. The Model stores the table data. Either the DefaultTableModel (picture in diagram) or a custom class implements AbstractTableModel, an abstract class that implements some of the methods required by the TableModel Interface. The TableModel interface allows the Model to communicate with the View. The Controller is a Listener that registers to receive events from the model. The concrete model class still needs to fire an event in order for the Controller to realize it needs to update.



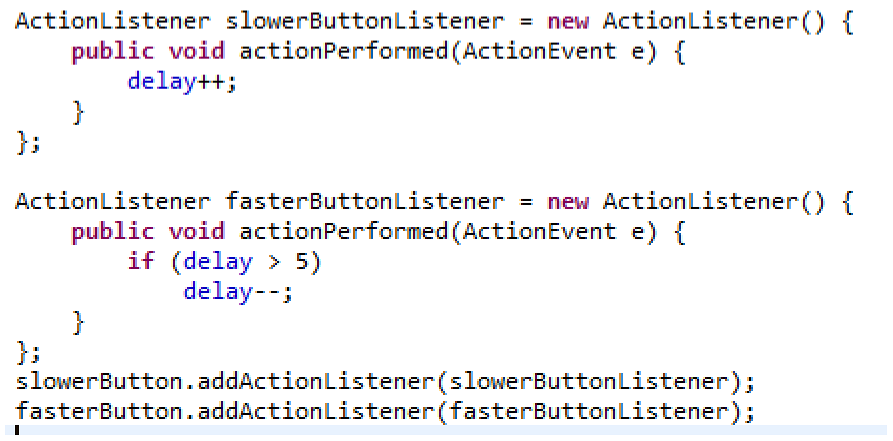
Experiment 3: Memory leaks

As with any program you need to worry about memory leaks and java swing is no different. One thing we were curious about was how listeners ended up closing. In many of the examples given the classes use anonymous inner classes for the listeners, but as it turns out doing as such can leave the threads in memory, even after they stop running. For an example code we have



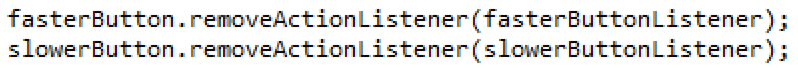
```
slowerButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        delay++;
    }
});
fasterButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (delay > 5)
            delay--;
    }
});
```

In order to make sure that we do not have any memory leak we must name them



```
ActionListener slowerButtonListener = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        delay++;
    }
};

ActionListener fasterButtonListener = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (delay > 5)
            delay--;
    }
};
slowerButton.addActionListener(slowerButtonListener);
fasterButton.addActionListener(fasterButtonListener);
```

Then after the thread is closing we need to make sure to remove them.



```
fasterButton.removeActionListener(fasterButtonListener);
slowerButton.removeActionListener(slowerButtonListener);
```

This will insure that the code does not cause any memory leaks. It is important when learning a new form of coding to not learn any bad forms, as memory leaks in large amounts of code can take an unbelievable amount of time to find and correct.

Portfolio 1
CS 319

[redacted]

Introduction

Java Swing is very useful for its portability, simplicity, and adaptability, making it an extremely interesting unit to study. In this segment, we learned about a variety of issues and concepts with Java Swing, from the underlying logic to how to create new applications the basic concepts. Throughout this unit, we explored some of the interesting facets of Java Swing, complex concepts related to Swing, evaluated Java Swing, and created new applications.

One of the more interesting things in Java Swing is its radio buttons. Radio buttons can initially be set to true, meaning they appear selected when the GUI opens. Experimenting with a few of the examples provided showed that when multiple radio buttons were set to true, the order in which they were added determined which one was initially set to true. This led us to further investigate exactly how Java actually determines which button will be set as true.

Underneath Java Swing’s elegant API, there are difficult issues that are used heavily: event handling and the model-view-controller design pattern. Java Swing makes use of a dedicated event handling thread to handle events and make corresponding updates to a user interface. We initially looked into the possibility of other approaches to handling events. However further exploration demonstrated the difficulties of polling and multithreaded approaches to event handling. We also explored the MVC or model-view-controller design pattern. This pattern allows function-specific code to be divided up and independently developed and tested, yet it can over-complicate simple programs.

While Java Swing provides a simple framework for Java desktop applications, it can be difficult to use. In order to decrease the difficulty of constructing layouts, many different strategies have been employed. WindowsBuilder is a popular plugin which allows designers to use a drag-and-drop GUI to design layout. Android development is based on Java, yet Android largely avoids the use of the Swing Library, though some of the core concepts are similar. Android user interface design includes different layouts including relative layouts which allows a designer to easily place components relative to each other in a view. In addition, the Model-View-Controller design pattern which is ubiquitous in Swing also has a few disadvantages.

Java Swing is used by many developers, so there are many external libraries that can be used to extend its capabilities. Two useful libraries are FEST and JGraphX which allow user interface testing and a graph component respectively. FEST allows end-to-end testing of user interfaces. For example, you can mimic when the user clicks on a button or types text into a field and then check to make sure that the observers of that button have done their correct actions. In the example under “Testing the User Interface”, we check to make sure the text in a text label is changed correctly when a person clicks on the “Italic” radio button. On the other hand, JGraphX provides a graph component for displaying things such as networks and webs. Under the section “JGraphX” there is a code snippet and screenshot of the basic capabilities of this library. FEST and JGraphX allow you to create more complex applications that are stable and useful.

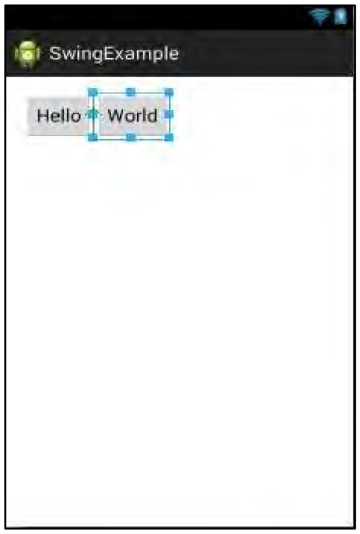
Java Swing is a fascinating framework, with complex topics, interesting quirks, and useful extensions. While it has some defects, such as the difficulty of layouts and over-complication, the advantages make it a common UI framework. The external libraries make Java Swing especially useful by eliminating the need to write new components or test frameworks. Radio buttons can have some quirks which are fascinating to delve into while event handling and the model-view-controller design pattern are used in a large number of applications. Below are deeper explanations of the concepts listed above including diagrams, concept maps, code snippets, and screenshots.

Comparison between Android and Java Swing

Developing for Android and Java are very similar in nature given that both are based upon the Java API. When it comes to user interfaces, Android appears to have drawn heavily upon the design of Java's Swing package. In particular, Android uses a single event handling thread to process events from an event queue, and a special thread used for making updates to the user interface. However, when it comes to layouts, there is some difference in the syntax of how to layout components (buttons, labels, etc…) within a view. The primary difference between the two is that Android makes use of separate XML files to declare view components (called widgets in android) and the components' basic properties (dimensions, background, text size, etc…). This may have been influenced by MVC and other design patterns which encourage the separation of the code for the user interface and the background business logic.

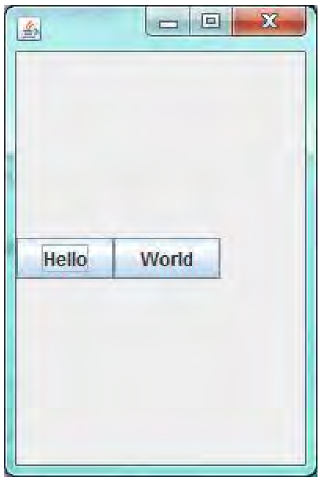
While the primary method of handling events is the same between Android and Java, the types of layout managers are very different between Java Swing and Android user interface design. Whereas Android primarily uses RelativeLayouts and LinearLayouts (though it has many other layouts); Java Swing relies on its own set of layout managers: BorderLayout, BoxLayout, CardLayout, FlowLayout, GridBagLayout, GridLayout, GroupLayout, SpringLayout. The following is an example of how to layout two buttons horizontally and vertically in both Android and Java Swing:

Buttons Horizontally in Android



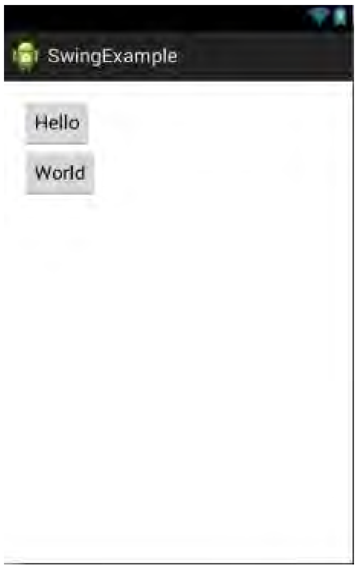
```
<RelativeLayout
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     <Button
     android:id="@+id/helloButton"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="@string/hello" />
     <Button
     android:id="@+id/worldButton"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_toRightOf="@+id/helloButton"
     android:text="@string/World" />
</RelativeLayout>
```

Buttons Horizontally in Swing



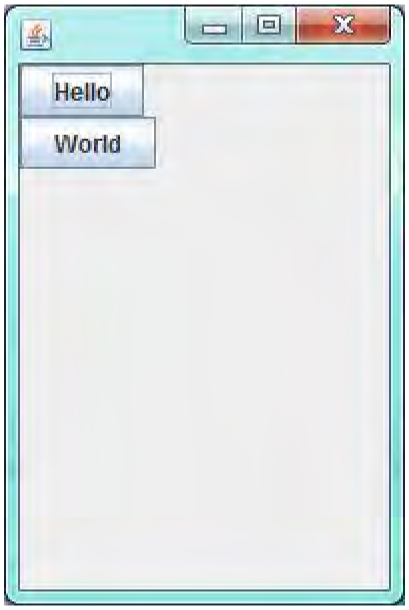
```
import java.awt.*;
import javax.swing.*;
public class HelloWorld extends JFrame {
     private static final long serialVersionUID = 1L;
     static JPanel panel;
     public HelloWorld(){

     panel = new JPanel();
     panel.setLayout(new BoxLayout(panel,
                         BoxLayout.X_AXIS));

     JButton helloButton = new JButton("Hello");
     panel.add(helloButton);

     JButton worldButton = new JButton("World");
     panel.add(worldButton);
}
     public static void main(String[]args){
     HelloWorld frame = new HelloWorld();
     frame.add(panel);
     frame.setSize(200, 300);
     frame.setDefaultCloseOperation(
                    WindowConstants.EXIT_ON_CLOSE);

     frame.setVisible(true);
     }
}
```

Buttons Vertically in Android



```
<RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
    <Button
        android:id="@+id/helloButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello" />
     <Button
        android:id="@+id/worldButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/World"
        android:layout_below="@+id/helloButton"/>
</RelativeLayout>
```

Buttons Vertically in Swing



```
import java.awt.*;
import javax.swing.*;

public class HelloWorld extends JFrame {
        private static final long serialVersionUID = 1L;
        static JPanel panel;
        public HelloWorld(){

        panel = new JPanel();
        //the BoxLayout is Y_AXIS as opposed to X_AXIS
        panel.setLayout(new BoxLayout(panel,
                                BoxLayout.Y_AXIS));

        JButton helloButton = new JButton("Hello");
        panel.add(helloButton);

        JButton worldButton = new JButton("World");
        panel.add(worldButton);
}
        public static void main(String[]args){
        HelloWorld frame = new HelloWorld();
        frame.add(panel);
        frame.setSize(200, 300);
        frame.setDefaultCloseOperation(
                        WindowConstants.EXIT_ON_CLOSE);
        frame.setVisible(true);
        }
}
```

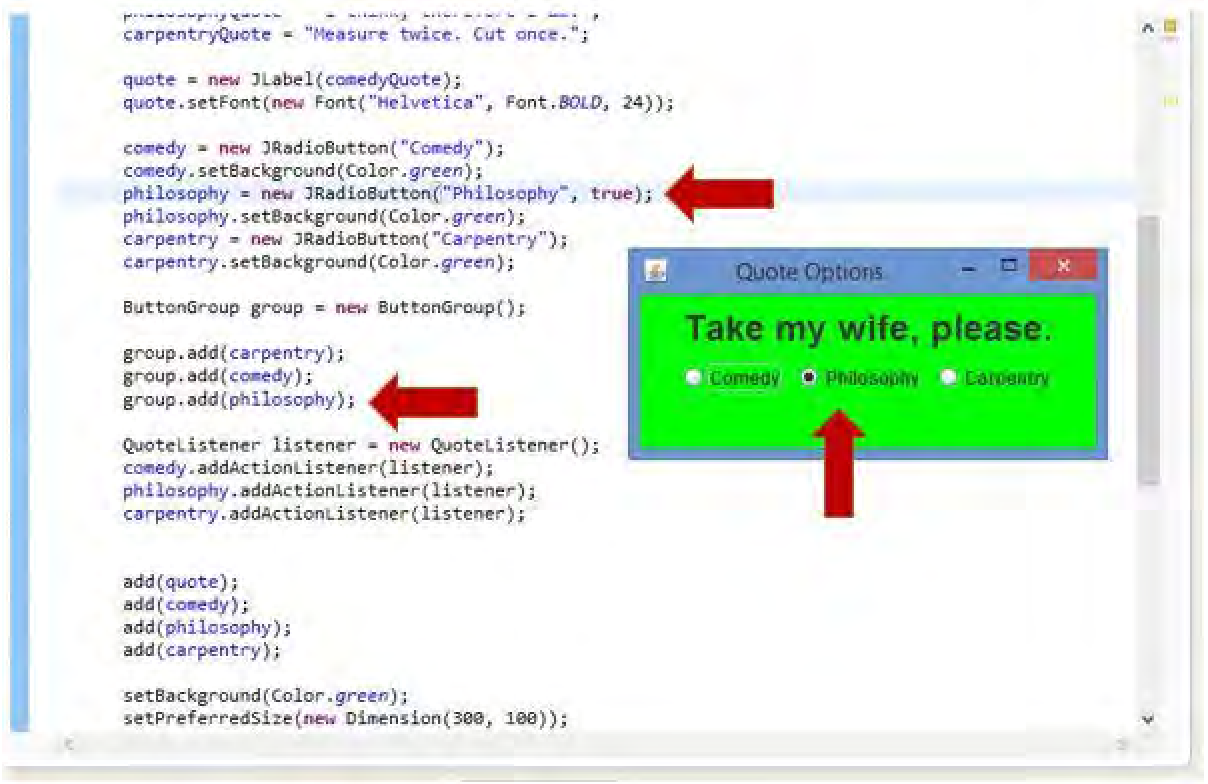
While the number of lines of code is approximately the same, the ability to align elements relative to each other allows a programmer to rapidly add items to the layout and place them on the screen where they need to be. Because the RelativeLayout from android simplifies the process of designing GUIs, many designers prefer it to the built-in layouts present in the Swing package. In response, multiple 3rd party libraries for swing have been created which emulate the RelativeLayout present in android, the most popular of which is open-source and easy to add to an existing java project: https://code.google.com/p/relativelayout/

Using Windows Builder

Given that many layout managers do not produce an exact representation of what a designer may want, the designer may be inclined to specify the exact location of each component within a view. However, manually typing the exact location of each component can be tedious and require careful precision to ensure no incorrect values are specified for the location or bounds of a component. The difficulties with manually designing absolute layouts led to the creation of "drag-and-drop" GUI designers. Windows Builder is an eclipse plug-in which allows a designer to simply "drag-and-drop" new components onto a Java Swing container.

Radio Buttons in Java Swing

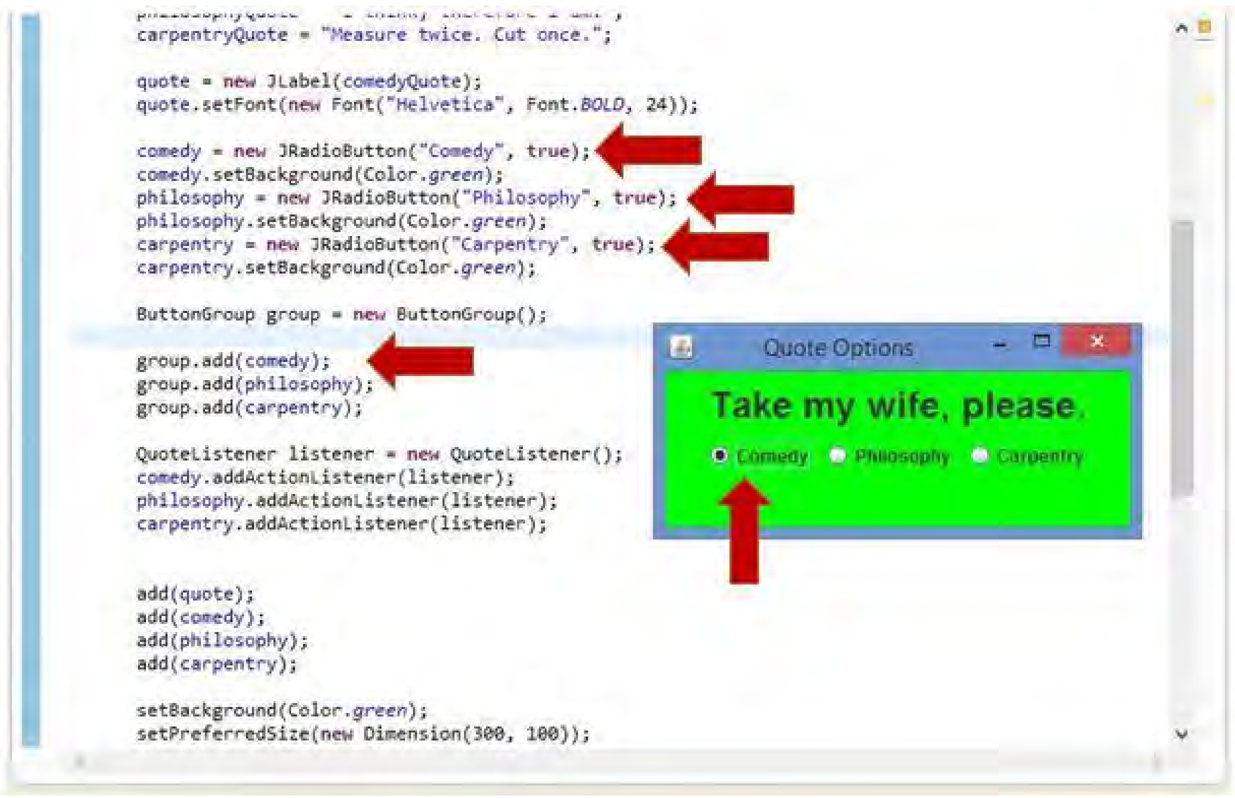
There are some interesting characteristics when setting certain radio buttons to true in a button group. When multiple JRadioButtons are created then added to a Button Group, and only one of them is set to true, that one will be selected when the application opens. This can be seen in the code example below.



```
carpentryQuote = "Measure twice. Cut once.";

quote = new JLabel(comedyQuote);
quote.setFont(new Font("Helvetica", Font.BOLD, 24));

comedy = new JRadioButton("Comedy");
comedy.setBackground(Color.green);
philosophy = new JRadioButton("Philosophy", true);
philosophy.setBackground(Color.green);
carpentry = new JRadioButton("Carpentry");
carpentry.setBackground(Color.green);

ButtonGroup group = new ButtonGroup();

group.add(carpentry);
group.add(comedy);
group.add(philosophy);

QuoteListener listener = new QuoteListener();
comedy.addActionListener(listener);
philosophy.addActionListener(listener);
carpentry.addActionListener(listener);


add(quote);
add(comedy);
add(philosophy);
add(carpentry);

setBackground(Color.green);
setPreferredSize(new Dimension(300, 100));
```

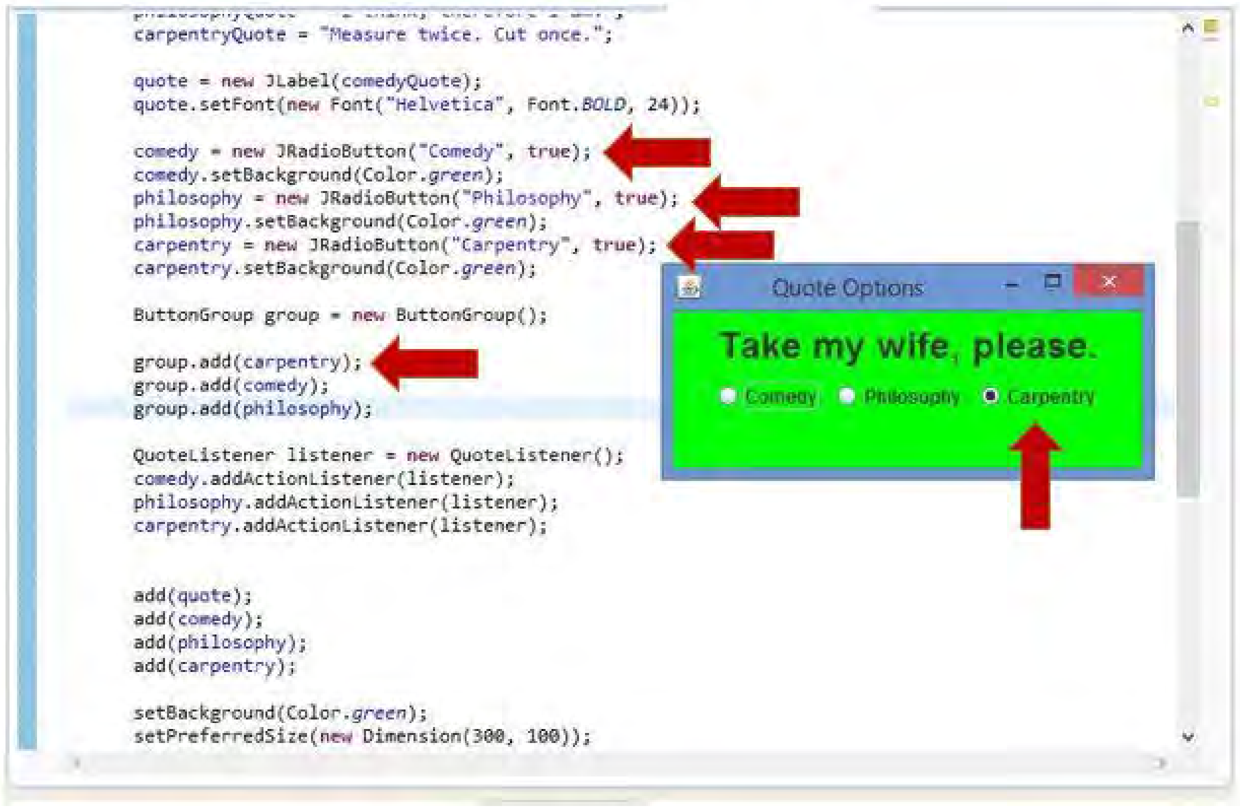
The red arrows point out the places in the code where Philosophy is set to true and then where philosophy is added to the Button Group. Even through philosophy is added last to the button group, and on the GUI application it is listed second, it is still selected as true. After testing out multiple orders for adding the JRadioButtons to the Button Group, it was determined that if only one button is set to true, then the order they are added in does not matter. This is not the case when there are multiple JRadioButtons set to true.

Now, if multiple items in a Button Group are selected as true, not all of the buttons appear selected upon running the application. The order in which the buttons are added to the group determines which button will be selected as true. The two pictures below clearly shows this.



```
carpentryQuote = "Measure twice. Cut once.";

quote = new JLabel(comedyQuote);
quote.setFont(new Font("Helvetica", Font.BOLD, 24));

comedy = new JRadioButton("Comedy", true);
comedy.setBackground(Color.green);
philosophy = new JRadioButton("Philosophy", true);
philosophy.setBackground(Color.green);
carpentry = new JRadioButton("Carpentry", true);
carpentry.setBackground(Color.green);

ButtonGroup group = new ButtonGroup();

group.add(comedy);
group.add(philosophy);
group.add(carpentry);

QuoteListener listener = new QuoteListener();
comedy.addActionListener(listener);
philosophy.addActionListener(listener);
carpentry.addActionListener(listener);


add(quote);
add(comedy);
add(philosophy);
add(carpentry);

setBackground(Color.green);
setPreferredSize(new Dimension(300, 100));
```

Take my wife, please.

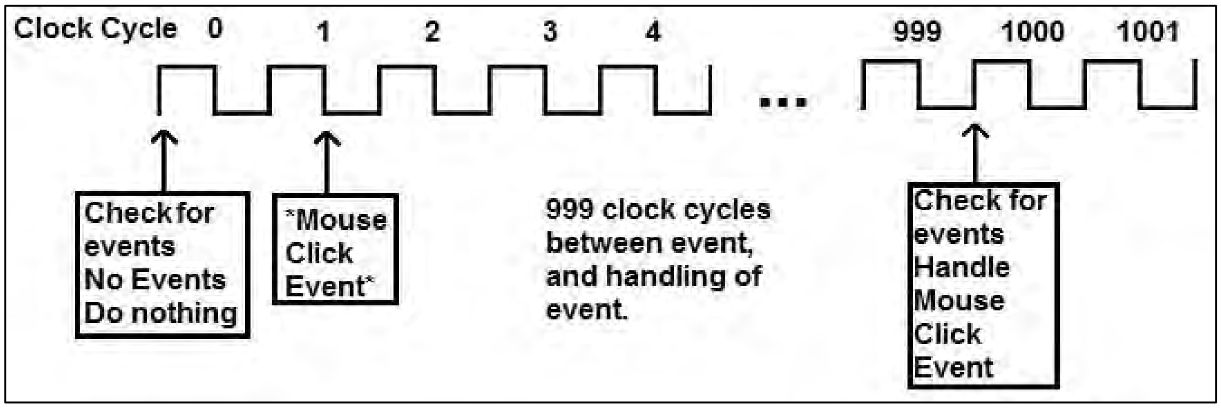


```
carpentryQuote = "Measure twice. Cut once.";

quote = new JLabel(comedyQuote);
quote.setFont(new Font("Helvetica", Font.BOLD, 24));

comedy = new JRadioButton("Comedy", true);
comedy.setBackground(Color.green);
philosophy = new JRadioButton("Philosophy", true);
philosophy.setBackground(Color.green);
carpentry = new JRadioButton("Carpentry", true);
carpentry.setBackground(Color.green);

ButtonGroup group = new ButtonGroup();

group.add(carpentry);
group.add(comedy);
group.add(philosophy);

QuoteListener listener = new QuoteListener();
comedy.addActionListener(listener);
philosophy.addActionListener(listener);
carpentry.addActionListener(listener);


add(quote);
add(comedy);
add(philosophy);
add(carpentry);

setBackground(Color.green);
setPreferredSize(new Dimension(300, 100));
```

The red arrows point out that each of the three JRadioButtons which are a part of the Button Group are all selected as true. Only one line of code changes between the two picture, this line is the one adding carpentry to the Button Group. Carpentry is originally added last in the first picture, and in the second picture it is added first. This shows that when the buttons are added to the Button Group, only the first button added is actually set as true. This means that if multiple Radio Buttons that are added to the same Button Group, only one of them can be set to true.

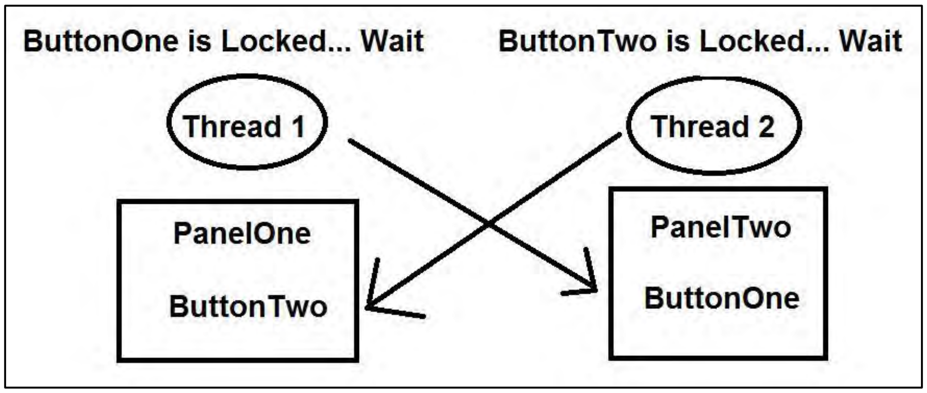
Handling Events inside a GUI: Why use a single event handling thread?

When considering the best way to handle events (including, but not limited to, button clicks, keyboard presses, and mouse movements) there are many different potential strategies that could be used. Three potential methods include: polling, single thread event queue handling, and multithreaded event queue handling. Polling is a particularly bad implementation of handling events as it will waste CPU cycles checking for events when no events have occurred, and is relatively slow in responding to events. For example if a program checks for events every 1000 clock cycles, there may be a large latency between the firing of the event and the handling of the event. The picture on the next page shows one of the most glaring problems with polling:


Clock Cycle 0 1 2 3 4 ... 999 1000 1001
Check for events No Events Do nothing
Mouse Click Event
999 clock cycles between event, and handling of event.
Check for events Handle Mouse Click Event

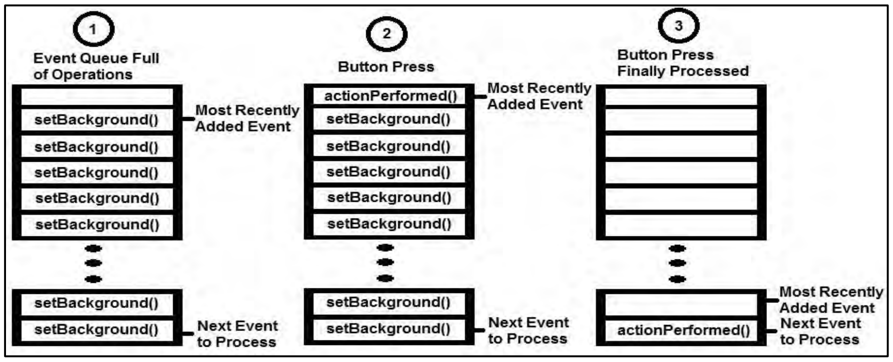
As you can see, polling may be one of the least responsive ways to handle events. Because polling is considered to be a bad idea, this may lead one to believe a multithreaded event queue would then be the best option. However, many problems associated with multithreaded programming are compounded by the updating of components present in a graphical user interface.

A deadlock is likely to occur if long-running updates to the GUI run on different threads simultaneously. For example, say there are two threads which are attempting to perform their own operations simultaneously. The thread “thread1” needs panelOne, buttonOne, and buttonTwo to perform its operation, and the thread “thread2” needs panelTwo, buttonTwo, and buttonOne to perform its operation. In order to prevent a race condition, the threads place a lock on the memory resources they currently have, and wait for the other necessary resources to become available. In the picture shown on the next page, the threads will become deadlocked as neither can run until the necessary resources have been freed by the other thread.



To combat this problem, one possibility would be to transfer control of setting locks on components to the programmer, thus removing the possibility that the background thread handlers would not run into problems. However, with most GUIs containing many components, it would greatly increase the complexity of the code necessary to rapidly produce bug-free GUIs. Swing's designers chose simplicity and reliability over the potential for a small, but unreliable performance increase. This decision for a single event handling thread over a multithreaded approach has been repeated in most modern-day user interface design packages (including Android).

Outside of the aforementioned small decrease in performance when compared to a multithreaded approach, there is one other notable downside to a queue-style of handling events. The major downside to a queue-style of handling events is that if individual events take a long time to complete, then the queue may backfill and the GUI may become unresponsive. For example, if there are many events containing panel.setBackground() operations for a large JPanel() queued in quick succession - it may take a while before all of the operations complete. The growth of the queue may result in other components on the GUI (such as buttons) becoming unresponsive to user input. See below for an illustration of the potential problem:


1
Event Queue Full of Operations
setBackground()
Most Recently Added Event
Next Event to Process
2
Button Press
actionPerformed()
3
Button Press Finally Processed

Note that the button press's actionPerformed() method will not be handled until all previously queued events have been handled by the event handling thread. Though this appears to be a fundamental problem with the queue event handling model – this problem will continue to be less and less noticeable as modern-day processors continue to improve in performance. This problem can also be mitigated by the software developer if they use good programming practices, and ensure the code in event actions take a short amount of time to execute.

Testing the User Interface

Testing the user interface of a product can be one of the most challenging and difficult tasks involved in its creation. The challenges arise when trying to mimic the user's actions, such as clicking on a button or entering text into the field.

One simple solution that quickly runs into difficulties is just firing the necessary events from your JUnit tests and then checking to make sure the values change correctly. However, this approach requires making components public or package private so that JUnit tests can access them.

There are several testing frameworks available to test Swing user interfaces, though. One of these is called FEST. FEST allows you to search for components and access them without making them public, getting around the problem of scope. FEST is also very easy to use. Below is a JUnit test that tests the feature from Exercise 1 where the text style should change when the user selects the italic radio button.

```
public void BasicTest() {
	window.label("Hello World!").font().requireBold();
	window.label("Hello World!").font().requireItalic();
	window.radioButton("Italic").click();
	window.label("Hello World!").font().requireItalic();
	window.label("Hello World!").font().requireNotBold();
}
```

Integrating FEST into the project is very simple; you simply add their external JARs to the build path. Using the code above only required a few small changes to the original source code we wrote such as assigning names to the radio button and label to make searching from them easier. One disadvantage of FEST is that it was very slow to run - the small test above took nearly four seconds because it actually brought up the JFrame and moved the pointer to click on the radio button.

While end to end tests such as the one above are useful and important, their use can be minimized by using a modular programing style. In this style, there are many small functions rather than a few large functions. These smaller methods are easier to test and write. While there may be a greater number of tests, each test is simpler and there are fewer end-to-end tests.

Another challenge is testing to make sure that the user interface is being rendered properly. This can be checked to some degree by using API calls such as getWidth and getHeight but such tests are easily flaky or do not test the GUI sufficiently which lessens their usefulness.

Differences between JDialog, JWindow, and JFrame

JFrame: This is the main window that an application typically runs in. It can contain one or more buttons, text input fields, text labels and other elements inside it. The window also allows a user to maximize, minimize, and close it.

JDialog: This is typically a box that pops up from a program running in a JFrame window. The JDialog box does not allow a user to continue using the main window until the JDialog box has been closed. This is typically used in programs to make sure a user reads over something and understands what it says. It is also used to request some form of input from the user. Many times, programs will ask a user to input some sort of text, such as their name. After the user inputs this information the JDialog box can be closed and the user can continue using the main program. The JDialog box also does not have a minimize or maximize button, but simply an exit button to close the window.

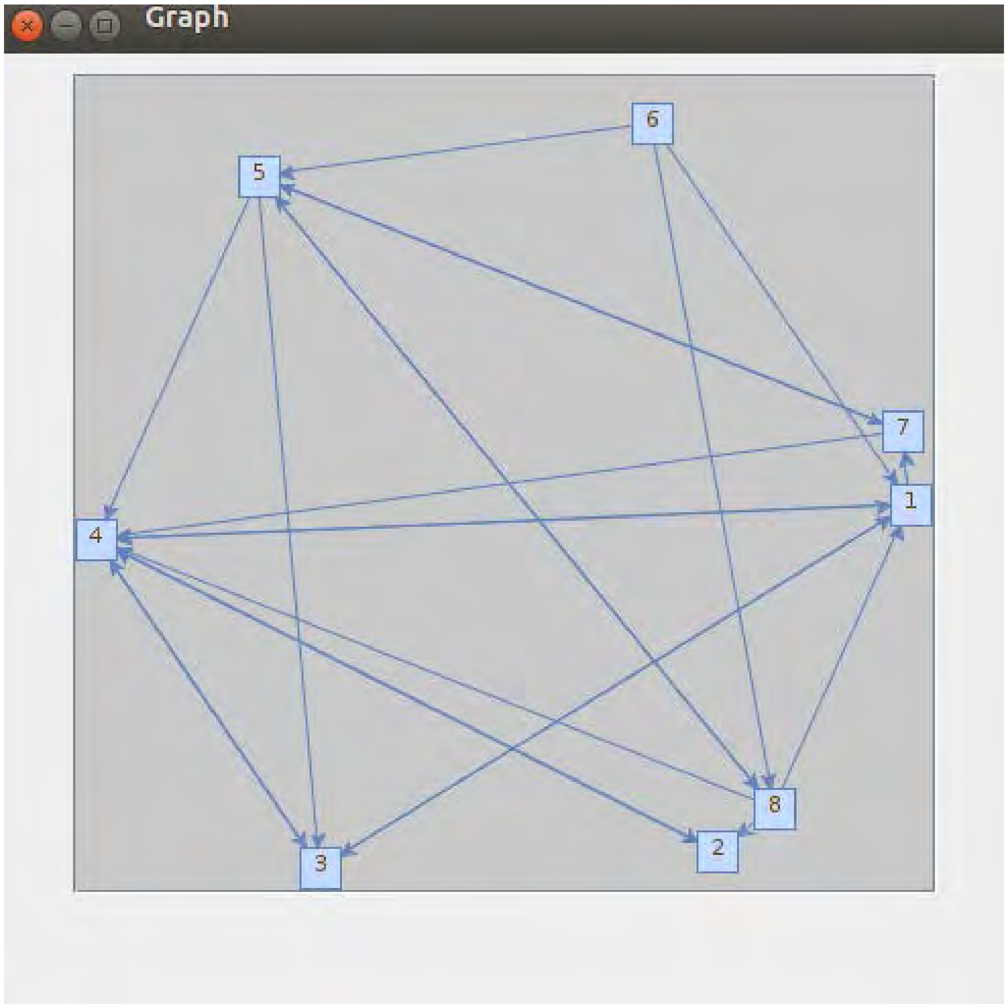
JWindow: This is a window that can live outside of the JFrame and completely on its own. It is different than a JDailog box because the user does not have to close this box before they can continue to use the main application. A JWindow also can be moved freely around the desktop by the user and is not constrained to the JFrame. This allows a program to open two or more separate windows on a desktop so the user can see multiple information. It also allows the user to move these windows around as they please on the desktop.

Advantages and disadvantages of Model-View-Controller

A Model-View-Controller pattern or MVC pattern breaks apart the way a program manages and displays data, as well as how a user interacts with that data. The view handles how all data is displayed, the model manages and stores all that data, and the controller handles what happens when a user interacts with the system, as well as controlling the model and view operations. This pattern allows code to be developed and tested independently of each other. The model portion, for example, can be developed by one programmer, while the view is developed by a different programmer. The two do not have to wait for one another to finish parts of their code. They are able to develop independently, and then integrate their code once everything is done. This pattern also helps reduce redundant code by allowing code to be called on as needed instead of having to rewrite some code each time. For example, if a certain view needs to be displayed multiple times, one view can be written and called upon multiple times to display an array of different items.

Although this pattern helps to reduce redundant code and make it easier to separate parts of the code, it does have some drawbacks. This pattern can be slower especially for programs running on low-end hardware. Using this pattern, an application must reference several different parts of the code several different times. Smaller applications that do not need to make references to say the views multiple times, could be better off not using the MVC pattern. The other main disadvantage to this pattern is that it is complicated. For larger programs, this MVC may be needed to organize code, but for smaller programs, the MVC pattern can be more confusing and complicated than needed.

JGraph



Besides the components available in the standard Java library, there are open-source components that can add further functionality to a Swing User Interface. JGraphX is a very useful component that is used to display graphs. It is the open-source version of the proprietary library mxGraph. The basic structure of the JGraphX is that mxGraphComponent extends JComponents and takes care of drawing the vertices and edges. mxGraphComponent has a mxGraph which is a model and stores the vertex and edge information. mxGraphComponent also supports changing the appearance of the vertices and edges through styles. Below is the snippet of source code that is used to create the graph shown above.

```
String[] vertices = {"1", "2", "3", "4", "5", "6", "7", "8"};
boolean[][] edges = new boolean[vertices.length][vertices.length];
....
mxGraph graph = new mxGraph();
graph.getModel().beginUpdate();
Object parent = graph.getDefaultParent();
try {
	Object[] v = new Object[vertices.length];
	for(int i = 0; i < vertices.length; i++) {
		v[i] = graph.insertVertex(parent, null, vertices[i],
				(int) (Math.cos(360.0/vertices.length*i)*200+200),
				(int) (Math.sin(360.0/vertices.length*i)*200+200), 20,
20);
	}
	Object[][] e = new Object[vertices.length][vertices.length];
	for(int i = 0; i < vertices.length; i++) {
		for (int j = 0; j < vertices.length; j++) {
			if (edges[i][j]) {
				graph.insertEdge(parent, null, "", v[i], v[j]);
			}
		}
	}
} finally {
	graph.getModel().endUpdate();
}
mxGraphComponent graphComponent = new mxGraphComponent(graph);
graphComponent.getViewport().setBackground(new Color(200, 200, 200));
```

Conclusion

Clearly Java Swing is a fascinating framework, whose popular and relatively easy-to-use GUI design is widely distributed. Because GUIs are so important to making software more user friendly, it is important to have a strong understanding of the underlying nature of the user interface construction and design. Many of the challenges associated with the construction of GUIs and event handling are simplified by Swing. Although there are many different approaches to dealing with the challenges associated with GUI design, other frameworks and applications draw upon the logic used within Java Swing. Overall this unit showed there is much to learn in order to gain a better understanding of swing, but also much to gain from this increased understanding of user interfaces.

Threads in Java Swing

Exploring, experimenting, and testing thread usage within Java Swing

COMPUTER SCIENCE 319

September 21, 2014

Authored by: [redacted]

Threads in Java Swing

Exploring, experimenting, and testing thread usage within Java Swing

Introduction

Although the use of multiple threads can be beneficial in certain situations and applications, it is generally not recommended when using Java Swing. The interface components constructed by Java Swing are not immutable, thus if multiple threads are used and its possible the schedule of those threads will not yield a desirable result. Such complications would include inconsistencies in memory, deadlock, or a frozen interface. Updates to Java Swing components and models such as JLabel, JPanel, ListModel, etc are all executed through the single EventDispatcherThread. There are instances, though, where multiple threads are safe or more practical. Our objective is to explore these situations as well as take a closer look at the EventDispatcherThread and how it benefits applications.

Expected Outcomes

Upon completion of this portfolio, we hope to gain a better understanding of the framework behind how Java Swing is executed. We want to be able to identify situations where creating our own threads would be safe and beneficial and then feel comfortable programming within those situations.

Our Reflection

In analyzing our own work for this portfolio, the first step we had to take was a lot of research. Many hours prior to writing any part of this portfolio or beginning on any piece of code for this portfolio, there was a lot of research to be done. We made it a point to explore the unique threading concepts of Java Swing in the most thorough manner we could in order to be able to begin our write ups. Once we obtained enough information, we constructed the outline for this project. Once we did this, we could pick a topic from our outline and dive further into it. For all of the topics, we made sure to elaborate on our understanding of the topic and then for most we created a program demonstrating our understanding. Before we reached the final program for a specific topic, we first had to experiment with other programs. We analyzed multiple source codes to help determine the program we should write. We then explained our program – how it worked and our insights. For example, on the topic of the Event Dispatch Thread we first had to explore all the relevant research we could. Once we obtained a comfortable understanding, we were able to study and compare source code to analyze how the EDT works. From there, we could begin writing our own code interacting and experimenting with the EDT. Finally, we developed a program which would demonstrate the functionality we aimed to present and discuss. We worked to attempt to do this with all of the topics we showcased. In the end, we think we showed a lot of work, a lot of understanding of that work, and a better overall comprehension of the topic of threads in Java Swing.

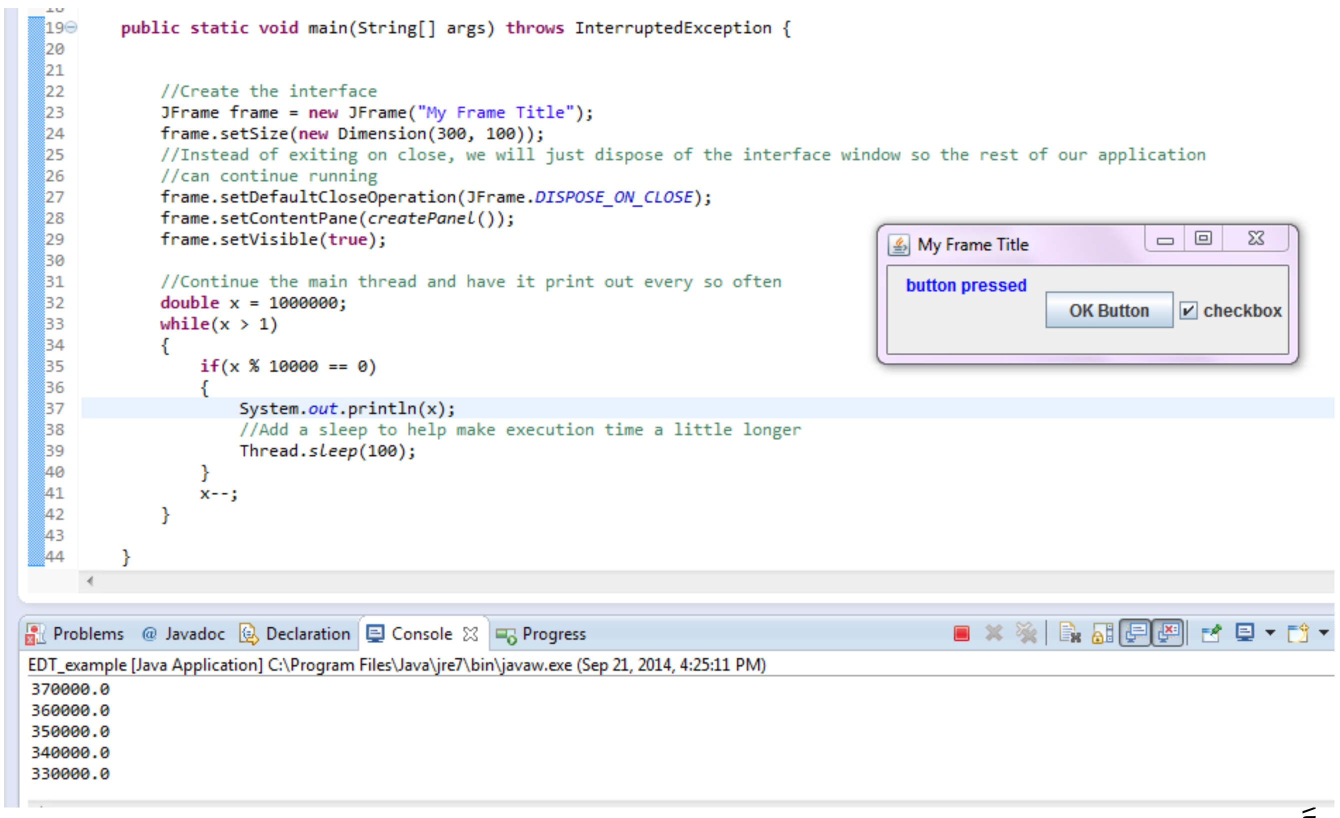
Table of Contents

Event Dispatch Thread

As stated before, the use of threads when constructing a Java Swing application is generally not recommended. Java Swing uses a single thread called the Event Dispatch Thread to execute all its tasks and updates. When one creates a main() method which sets up a GUI application, the event dispatch thread is started to control updates and interactions for the GUI. This thread is separate from the main thread initialized at the start of the program. By having this event dispatch thread, it allows the user to have a specific thread for the GUI and thus other portions of the main thread can run at the same time.

Example: EDT_example

An easy example to show this is the following program called "EDT_example". The main method creates a GUI and then continues to do some computations and print statements.



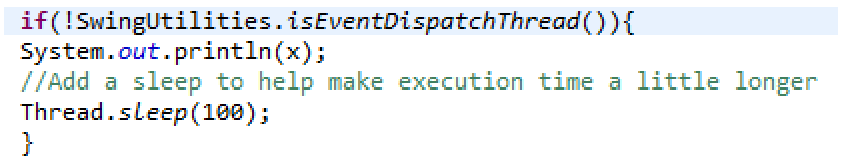
```java
    public static void main(String[] args) throws InterruptedException {


        //Create the interface
        JFrame frame = new JFrame("My Frame Title");
        frame.setSize(new Dimension(300, 100));
        //Instead of exiting on close, we will just dispose of the interface window so the rest of our application
        //can continue running
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        frame.setContentPane(createPanel());
        frame.setVisible(true);

        //Continue the main thread and have it print out every so often
        double x = 1000000;
        while(x > 1)
        {
            if(x % 10000 == 0)
            {
                System.out.println(x);
                //Add a sleep to help make execution time a little longer
                Thread.sleep(100);
            }
            x--;
        }

    }
```

```
EDT_example [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Sep 21, 2014, 4:25:11 PM)
370000.0
360000.0
350000.0
340000.0
330000.0
```

Here is a snippet of the important parts of the code as well as the execution of the program. As you can see, I was able to interact with the GUI while the main thread continued its execution and printing. I feel this example demonstrates the usefulness of the event dispatch thread. There may be many circumstances where there is more to a program than only displaying and updating a GUI. The EDT makes it possible to do more with your program.

Recognizing the event dispatch thread

To expand a little further on the previous example, we can explicitly tell if we are in the event dispatch thread by using the Swing utilities method isEventDispatchThread(), which returns true or false depending on what thread is executing that portion of the process. So if I were to add a simple conditional using this method, we would be able to see it is indeed the main thread executing the print statements.



```
if(!SwingUtilities.isEventDispatchThread()){
System.out.println(x);
//Add a sleep to help make execution time a little longer
Thread.sleep(100);
}
```

When this conditional is added, the program runs just like it did before. If one were to take out the “not”, then the program no longer prints to the screen because this part of the program does not get executed by the event dispatch thread.

USING THREADS IN JAVA SWING

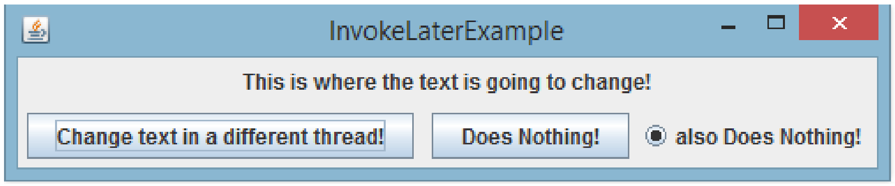
As we have talked about using a thread to change the GUI should not be done because most methods are not thread safe. We have also talked about how if you put all of tasks on the EDT that the GUI can become unresponsive. So how do you work around making the GUI responsive but not changing it in any other threads. The answer is held with the methods invokeLater and invokeAndWait. These methods make it possible for a thread to tell the EDT to make a change to the GUI.

The method invokeLater belongs to the SwingUtilites class and is used to update or perform any task on the EDT asynchronously. It works by adding a runnable object to the EDT queue.

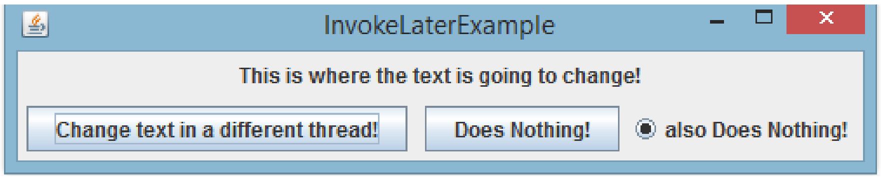
The method invokeAndWait also belongs to the SwingUtilites class and is used to update or perform any task on the EDT but it performs those tasks synchronously. InvokeAndWait is a blocking method that will block until the task is complete.

Example: InvokeLaterExample

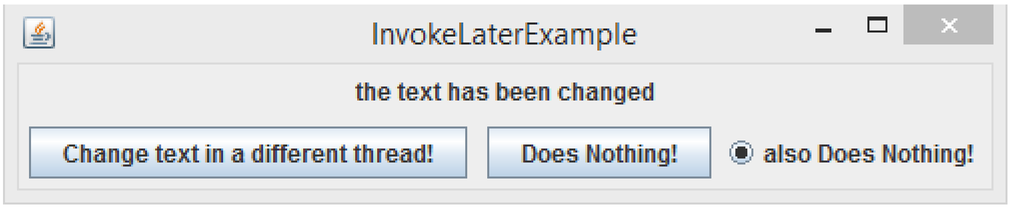
For an example we chose to use invokeLater to update the GUI from a SwingWorker thread. The idea behind the example is that we want to do a time-consuming task before changing the GUI. We want to use a worker thread so that the GUI is still responsive till the task is completed. We need to use the method invokeLater in order to tell the EDT to change the GUI.


InvokeLaterExample
This is where the text is going to change!
Change text in a different thread!
Does Nothing!
also Does Nothing!

This is a picture of the GUI. Pressing the first button will start the thread that does a time-consuming task then asks the EDT to change the text on the top. The other buttons do nothing(you can use these to make sure the GUI has not frozen because of the time-consuming task).


InvokeLaterExample
This is where the text is going to change!
Change text in a different thread!
Does Nothing!
also Does Nothing!

After the first button is pressed the GUI does not change but is still responsive.

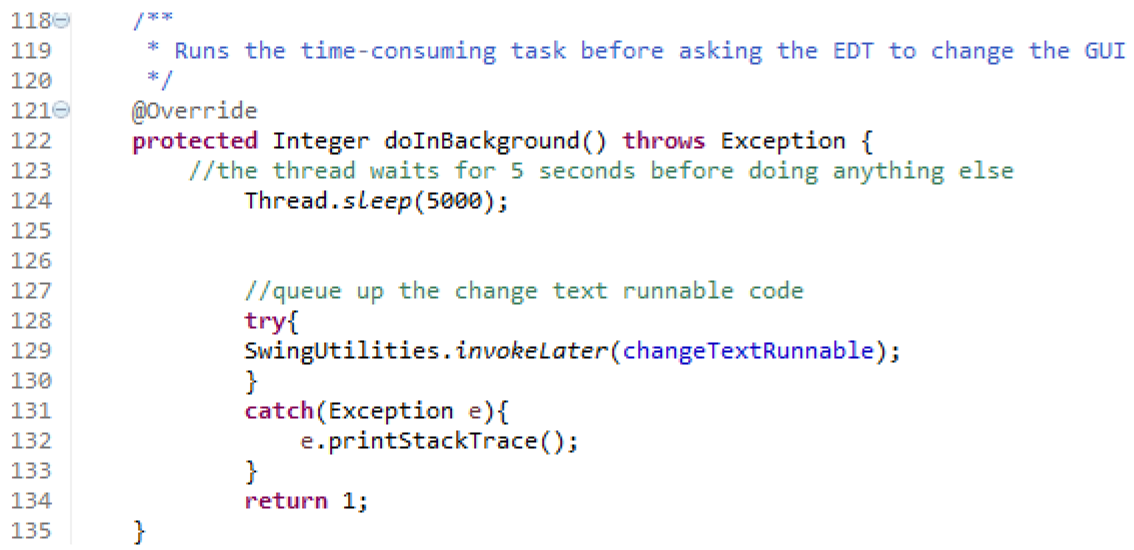


After 5 seconds the text at the top of the frame changes because the worker thread asks the EDT to change it.

CODE:

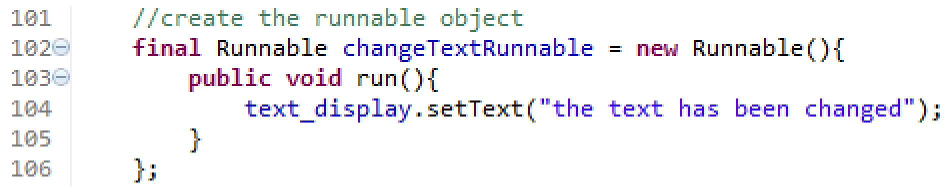
```java
    /**
     * Creates the button that creates and starts the worker thread when pressed
     * @param text_display the JLable to be changed when the button is pressed
     * @return the JPanel containing the button
     */
    private JPanel createControllerButtonsPanel(JLabel text_display) {
        this.text_display = text_display;
        JPanel controllerButtonsPanel = new JPanel();
        JButton changeText = new JButton("Change text in a different thread!");
        controllerButtonsPanel.add(changeText);

        //Action listener for the change text button. creates and executes the worker thread
        changeText.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent evt){
                //make a new worker thread that runs for 10 second
                ChangeTextSwingWorker worker = new ChangeTextSwingWorker(text_display);
                //start the worker thread
                worker.execute();
            }
        });
        return controllerButtonsPanel;
    }

}
```

This section of code is where the button that creates the worker thread and starts it is created. You can see in the action listener that the worker thread is executed when the button is pressed.



```java
    /**
     * Runs the time-consuming task before asking the EDT to change the GUI
     */
    @Override
    protected Integer doInBackground() throws Exception {
        //the thread waits for 5 seconds before doing anything else
            Thread.sleep(5000);


            //queue up the change text runnable code
            try{
            SwingUtilities.invokeLater(changeTextRunnable);
            }
            catch(Exception e){
                e.printStackTrace();
            }
            return 1;
    }
```

This section of code shows the doInBackground method of the worker thread. This is the code that is executed when the execute command is given. the thread sleeps for 5 seconds and then asks the EDT to run the changeTextRunnable object.



```java
    //create the runnable object
    final Runnable changeTextRunnable = new Runnable(){
        public void run(){
            text_display.setText("the text has been changed");
        }
    };
```

This section of code shows how the runnable object is created. the code within the run method is what is run by the EDT when the invokeLater is called.

THREAD-SAFE EXCEPTIONS

Time consuming tasks

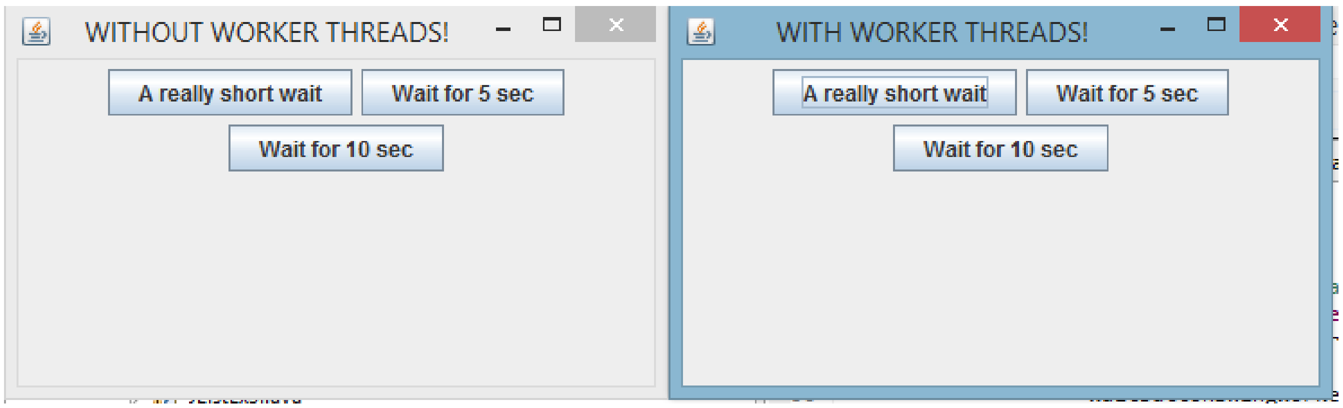
A GUI user is very sensitive to time delays in the GUI and when a Swing program needs to run a time-consuming task it can “freeze” the GUI if run on the main thread. Even a fraction of a second can be noticed by the user and they don’t like to wait. In this instance a programer would want to create a worker or background thread which does the time-consuming task while updating the GUI as it goes. Swing has an abstract class called StringWorker for this exact purpose. SwingWorker allows the programer to create a worker thread that is thread safe, does not freeze the GUI and is capable of updating the GUI and informing the user how long it is going to take.

Example: SwingWorkerExample

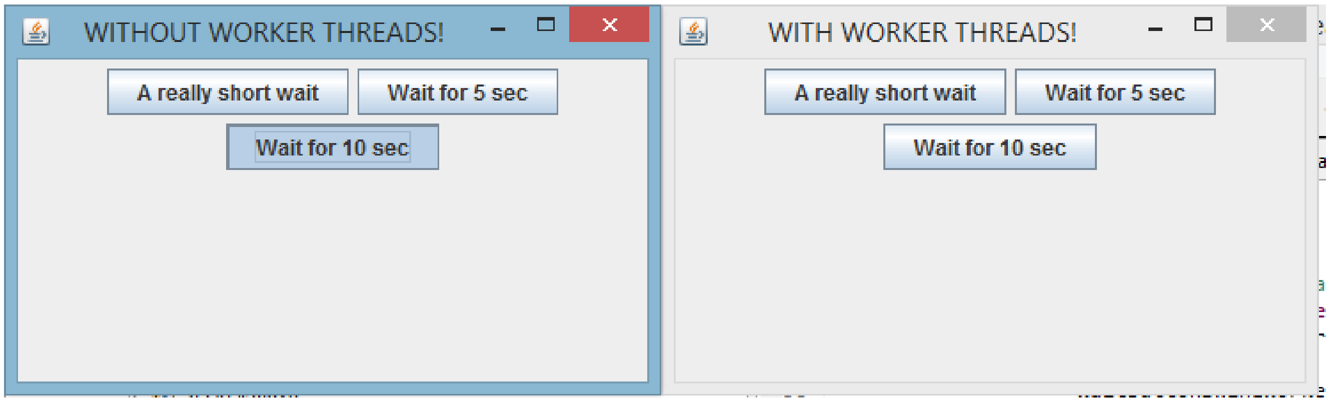
We created a simple program called “SwingWorkerExample” that shows what a Swing program that shows what a Swing GUI looks like when running a time consuming task on the main thread compared to using a SwingWorker thread.

The Method without using SwingWorker just waits for the given amount of time before it displays that it has completed the task simulating a task that would be time-consuming. If you rn the program and click any of the buttons you will notice the time delay before you can click anything again. It also does not let you close the program and the GUI completely freezes.

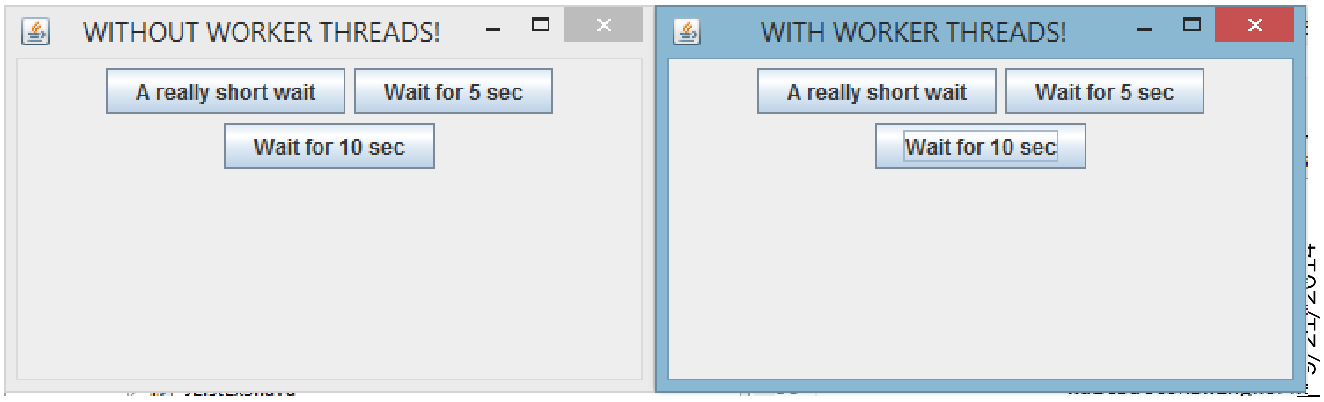
The method using the SwingWorker does the same thing regarding the wait but instead does the waiting on the worker thread rather than the main thread. When running the program and clicking the buttons the GUI does not freeze and the user can actually run multiple instances of each thread. The only way to know that the tread has done its computation for this example is by the tread bringing up a dialog box saying that it has completed. There are other ways of informing the user what the thread is doing. One way would be outputting strings updating the user as to the exact step that the thread is on or by creating a status bar that fills up to 100%. Both of these are easy to implement when using StringWorker.



The GUIs next to each other the example running the time-consuming task in the main thread on the left and the example using SwingWorker on the right.



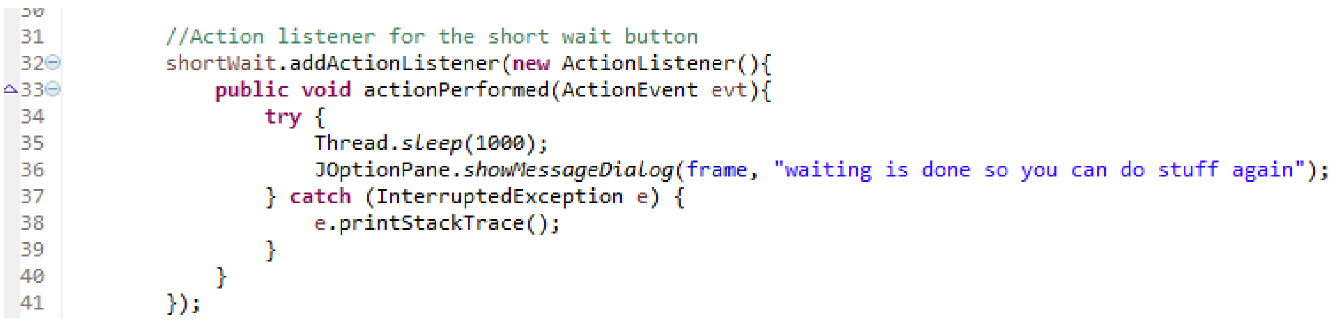
after clicking one of the buttons on the example that does not use SwingWorker the GUI freezes showing the pressed button still pressed until the task is complete.



This shows the example that uses SwingWorker right after the wait for 10 sec button was pressed. Because the time-consuming task is run on the worker thread the user is able to continue to use the GUI.

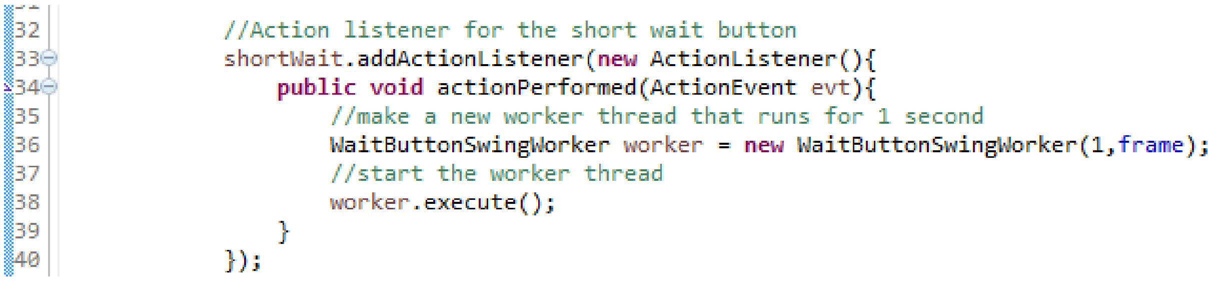
CODE:

There are a few portions of code that are interesting for each portion of this example.



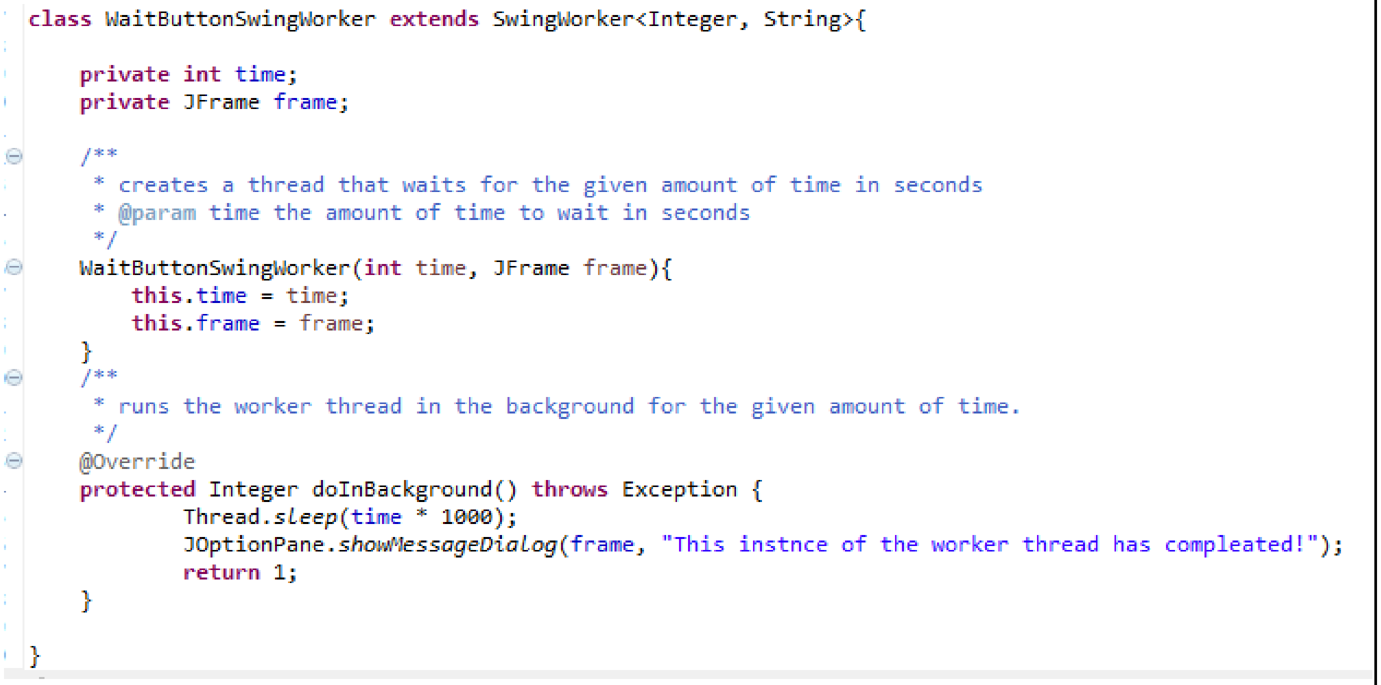
```java
        //Action listener for the short wait button
        shortWait.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent evt){
                try {
                    Thread.sleep(1000);
                    JOptionPane.showMessageDialog(frame, "waiting is done so you can do stuff again");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
```

This section of code is from the example of code that does not use SwingWorker. This shows how the program starts executing the time-consuming task within the main thread.



```java
            //Action listener for the short wait button
            shortWait.addActionListener(new ActionListener(){
                public void actionPerformed(ActionEvent evt){
                    //make a new worker thread that runs for 1 second
                    WaitButtonSwingWorker worker = new WaitButtonSwingWorker(1,frame);
                    //start the worker thread
                    worker.execute();
                }
            });
```

This section of code is from the example that uses the SwingWorker. This shows us that after the button is pressed an instance of the SwingWorker is created and then is told to execute from the main thread.



```java
class WaitButtonSwingWorker extends SwingWorker<Integer, String>{

    private int time;
    private JFrame frame;

    /**
     * creates a thread that waits for the given amount of time in seconds
     * @param time the amount of time to wait in seconds
     */
    WaitButtonSwingWorker(int time, JFrame frame){
        this.time = time;
        this.frame = frame;
    }
    /**
     * runs the worker thread in the background for the given amount of time.
     */
    @Override
    protected Integer doInBackground() throws Exception {
            Thread.sleep(time * 1000);
            JOptionPane.showMessageDialog(frame, "This instnce of the worker thread has compleated!");
            return 1;
    }

}
```

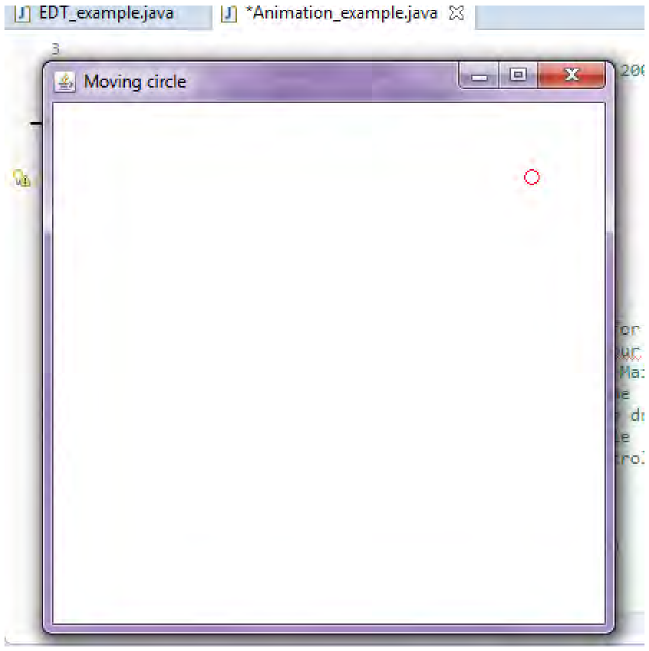
This is the SwingWorker class that was extended for this example. As you can see there is not much code that needs to be written. In main thread the worker.execute() starts the doInBackground() method and runs the code. In this method a programer can add lines that tell the main thread how how far it has executed and can even send objects.

Animation

Now that we have discussed how one can use worker threads to accomplish tasks one would not want to be run on the event dispatcher thread, we can look at the practicality of this. It is not recommended to put the EDT to sleep() or wait(), but such things can be useful when wanting to implement animation and moving objects. One way to achieve a moving effect is to create a separate thread for the moving object which sleeps for a little bit and then changes the object's x and y coordinates. If one wanted to have multiple moving objects, one could invoke more than one of these threads.

Example: Animation_example

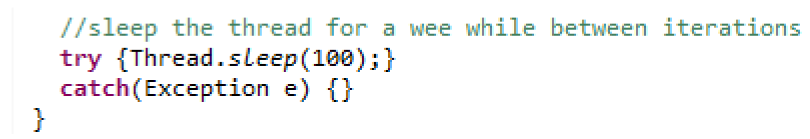
I created a simple example using Java Swing's 2D graphics to make a circle bounce around a screen. It uses a thread separate than that of the EDT to update the movement of the circle. Here is a capture of the program running:



To make a separate thread for the moving circle object we just extend it from Thread:

```
class Circle extends Thread {
```

It then has a run() method where we can update the drawing and then put the thread to sleep. Sleep will be put at the end of the run() method:



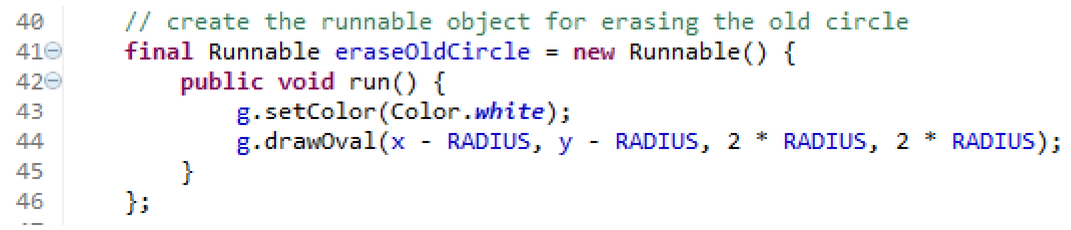
```
    //sleep the thread for a wee while between iterations
    try {Thread.sleep(100);}
    catch(Exception e) {}
  }
```

This is just one way to implement animations. Using a thread in this particular instance is safe due to few updates going on as well as no user input. As the application becomes more complex, having separate threads manually updating the GUI may cause problems. In this case, we can use what we have talked about before- Java Swing Utilities InvokeAndWait() and InvokeLater() methods. The purpose the example we just demonstrated was to explore an application of using a custom built thread rather than a Java Swing worker thread.

Application of Swing Utilities

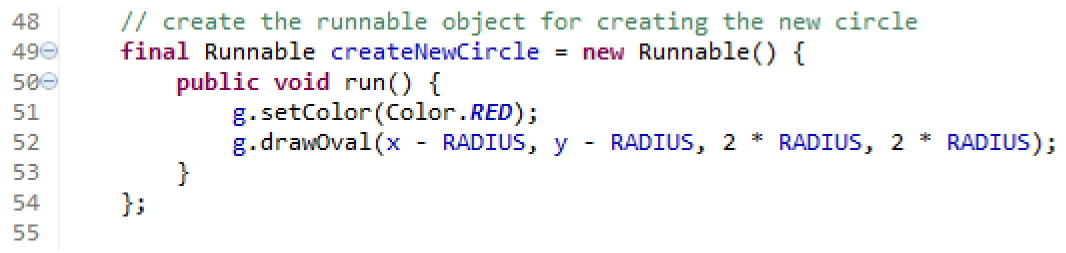
As we said before we can change the above example to include the Swing utilities that we talked about earlier. In this case we can use the invokeAndWait method in order to tell the event dispatch thread to erase the old Circle and add the new one. This creates a much more reliable Swing program and also lets us add more functionality. Because the animations Have to take place in order We would want to make sure that the invokeAndWait method is used so that the event dispatch thread executes the erase before it adds the new circle back.

Example: Animation_utilities



```java
    // create the runnable object for erasing the old circle
    final Runnable eraseOldCircle = new Runnable() {
        public void run() {
            g.setColor(Color.white);
            g.drawOval(x - RADIUS, y - RADIUS, 2 * RADIUS, 2 * RADIUS);
        }
    };
```

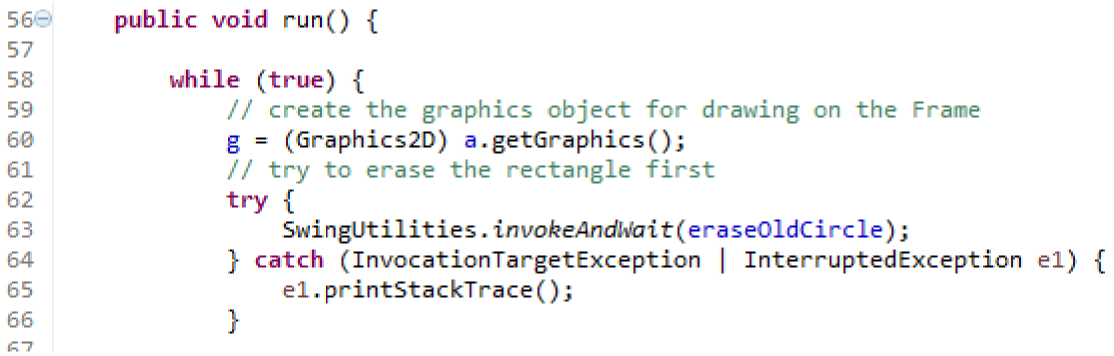
Here is the runnable object for erasing the old circle



```java
    // create the runnable object for creating the new circle
    final Runnable createNewCircle = new Runnable() {
        public void run() {
            g.setColor(Color.RED);
            g.drawOval(x - RADIUS, y - RADIUS, 2 * RADIUS, 2 * RADIUS);
        }
    };

```

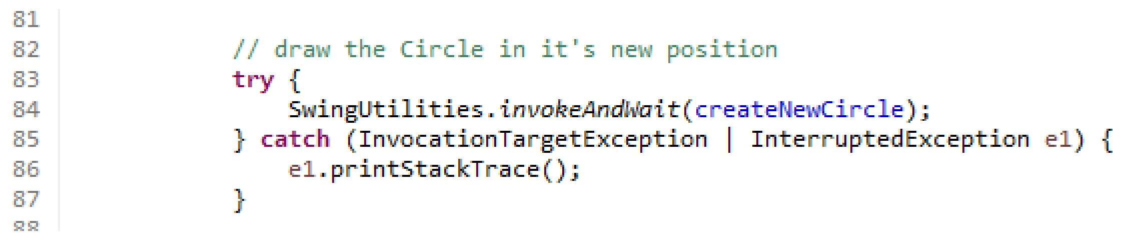
Here is the runnable object for creating the new cercle

These two objects can be run by the EDT when the thread calls SwingUtilites.invokeAndWait.



```java
    public void run() {

        while (true) {
            // create the graphics object for drawing on the Frame
            g = (Graphics2D) a.getGraphics();
            // try to erase the rectangle first
            try {
                SwingUtilities.invokeAndWait(eraseOldCircle);
            } catch (InvocationTargetException | InterruptedException e1) {
                e1.printStackTrace();
            }
```

This portion of the code shows where the first invokeAndWait method is called in order to erase the old circle.



```java

            // draw the Circle in it's new position
            try {
                SwingUtilities.invokeAndWait(createNewCircle);
            } catch (InvocationTargetException | InterruptedException e1) {
                e1.printStackTrace();
            }
```

After doing some calculations on where the next circle will be placed this code is run, requesting the EDT to update the GUI to include the new circle.

Explicit Thread-Safe Methods

Many of the methods within java swing that are thread safe deal with different aspects of a text editor. There are methods withing the different document classes that allow thread safe addition and removal of content, as well as the UndoManager, which is the only class in java swing that is completely thread safe, that gives the ability to undo and redo significant changes. Other thread safe methods include a set of methods that allow the addition and removal of different attributes to the style of the text.

One of the first uses which comes to mind, knowing this is a system to allow multiple writers, would be to use the same editor at once. Another use would be if a thread was creating a message to be displayed, different parts could be printed while the thread is still working.

Thread Safe Methods:

- javax.swing.text.DefaultStyledDocument
  - insert() - Adds new Elements in bulk. Uses a write lock.
  - SetLogicalStyle() - Sets the logical style to use for the paragraph at the given position.
  - SetCharacterAttributes() - Sets attributes for a part of the document, uses a write lock.
  - SetParagraphAttributes() - Sets attributes for a paragraph.
- javax.swing.undo.UndoManager
  - All methods – An object that creates a list of undoable edits and provides methods for Undoing and Redoing edits.
- javax.swing.text.AbstractDocument
  - render() - Safely renders the model in the presence of currency, asynchronous updates are supported.
  - Remove() - Uses a write lock to remove content.
  - insertString() - Uses a write lock to add content.
  - CreatePosition() - Returns a position that will change as the document is altered.
- javax.swing.text.JtextComponent
  - print() - Prints content of the JTextComponent
  - getPrintable() - Gets the printable content of the Component

- javax.swing.text.StyleContext
  - addAttribute() - Adds an attribute to a set.
  - addAttributes() - Adds attributes to a set.
  - removeAttribute() - Removes and attribute from a set.
  - removeAttributes() - Removes attributes from a set.
  - reclaim() - returns a set no longer needed by the implmentation.
- javax.swing.text.DefauoltCaret
  - repaint() - repaints the Caret object.
- javax.swing.text.PlainDocument
  - insertString() - Inserts content into the document, uses a write lock.
- javax.swing.text.html.HTMLDocument
  - setParagraphAttributes() - Sets attributes for a paragraph.

Conclusion

After exploring these threading topics in Java Swing, we all learned a lot and would definitely feel comfortable explaining how threads relate to Java Swing. It wasn't until many hours later and a lot of research and testing until we truly understood the importance of thread safety in Swing. In the end, we came to the conclusion that although it is possible to use multithreading in Java Swing, it is best to avoid it if one can. And if one finds an application where multiple threads would be beneficial, using the built in Java Swing Utility methods are the best way to execute it. Also the use of the Swing worker threads is safe and more beneficial when the process must wait for something.

Appendix

EDT_example.java:

```
package thread_portfolio;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class EDT_example {

	public static void main(String[] args) throws InterruptedException {


		  //Create the interface
		 JFrame frame = new JFrame("My Frame Title");
		frame.setSize(new Dimension(300, 100));
		//Instead of exiting on close, we will just dispose of the interface
		//window so the rest of our application
		//can continue running
		frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
		frame.setContentPane(createPanel());
		frame.setVisible(true);

		//Continue the main thread and have it print out every so often
		double x = 1000000;
		while(x > 1)
		{
		   if(x % 10000 == 0)
		   {
			   if(!SwingUtilities.isEventDispatchThread()){
			   System.out.println(x);
			   //Add a sleep to help make execution time a little longer
			   Thread.sleep(100);
			   }
		   }
		   x--;
		}
```

```
        }

    static JPanel createPanel() {

            JPanel mainPanel = new JPanel();
        mainPanel.setLayout(new BoxLayout (mainPanel, BoxLayout.X_AXIS));


            JPanel statusPanel = new JPanel();
            final JLabel statusMessage = new JLabel("Nothing yet");
            statusMessage.setForeground(Color.BLUE);
            statusPanel.add(statusMessage);
            mainPanel.add(statusPanel);


        JButton button1 = new JButton("OK Button");

        //Listener for button
        button1.addActionListener(
                new ActionListener() {
                        @Override
                        public void actionPerformed(ActionEvent e) {
                                statusMessage.setText("button pressed");
                        }
                }
            );
        mainPanel.add(button1);

            // Create a Checkbox and add it to the panel
        final JCheckBox chk = new JCheckBox("checkbox");
        chk.addChangeListener(

                new ChangeListener() {
                        @Override
                        public void stateChanged(ChangeEvent e) {
                          if(chk.isSelected()) {
                                  statusMessage.setText("checked");
                          } else {
                                statusMessage.setText("not checked");
                          }
                        }
                }

            );
        mainPanel.add(chk);


            return mainPanel;

        }
    }
```

InvokeLaterExample:

```
package thread_portfolio.src;


import java.awt.Dimension;
import javax.swing.JFrame;

/**
 * Class that contains the main method. This class creates the frame and puts the
InvokeLaterPanel into it.
 * @author [redacted]
 *
 */
public class InvokeLaterExample extends JFrame{

	static JFrame frame;
	public static void main(String[] args){
		//create the frame and sets its properties
		frame = new JFrame("InvokeLaterExample");
		frame.setSize(new Dimension(370, 220));
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		//adds the InvokeLaterPanel to the frame and packs it
		frame.getContentPane().add(new InvokeLaterPanel());
		frame.pack();
		frame.setVisible(true);
	}

}
package thread_portfolio.src;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.SwingUtilities;
import javax.swing.SwingWorker;

/**
 *
 * @author [redacted]
 *
 */
public class InvokeLaterPanel extends JPanel {

	private JLabel text_display;
/**
 * Constructor for the InvokeLaterPanel
 */
	InvokeLaterPanel(){
		//set the layout to borderlayout
		setLayout(new BorderLayout());
		//make the text panel
		JPanel textPanel = new JPanel();
		textPanel.setMaximumSize(new Dimension(10, 100));

		//make the text box for the panel
		text_display = new JLabel("This is where the text is going to change!");
		textPanel.add(text_display);

		//make the button that changes the text after waiting for some time in a
different thread
		JPanel controllerButtonsPanel =
createControllerButtonsPanel(text_display);


		//make buttons that you can play with while you wait for the text to
change
		JPanel uselessButtonsPanel = createUselessButtonsPanel();

		//add the panels to the main panel
		add(textPanel,BorderLayout.NORTH);
		add(uselessButtonsPanel,BorderLayout.EAST);
		add(controllerButtonsPanel, BorderLayout.WEST);

	}
	/**
	 * creates a few useless buttons so we can tell the GUI is still responsive
while the time-consuming class is running
	 * @return the button pannel that contains useless buttons
	 */
	private JPanel createUselessButtonsPanel() {
		//create the JPanel to be return
		JPanel UselessButtonsPanel = new JPanel();
		//A JButton that does nothing
		JButton button1 = new JButton("Does Nothing!");
		//A JRadioButton that does nothing
		JRadioButton radio1 = new JRadioButton("also Does Nothing!", true);

		//add buttons to the panel
		UselessButtonsPanel.add(button1);
		UselessButtonsPanel.add(radio1);

		return UselessButtonsPanel;
	}
	/**
	 * Creates the button that creates and starts the worker thread when pressed
	 * @param text_display the JLable to be changed when the button is pressed
	 * @return the JPanel containing the button
	 */
	private JPanel createControllerButtonsPanel(final JLabel text_display) {
		this.text_display = text_display;
		JPanel controllerButtonsPanel = new JPanel();
		JButton changeText = new JButton("Change text in a different thread!");
		controllerButtonsPanel.add(changeText);
```

```
            //Action listener for the change text button. creates and executes the
worker thread
            changeText.addActionListener(new ActionListener(){
                  public void actionPerformed(ActionEvent evt){
                        //make a new worker thread that runs for 10 second
                        ChangeTextSwingWorker worker = new
ChangeTextSwingWorker(text_display);
                        //start the worker thread
                        worker.execute();
                  }
            });
            return controllerButtonsPanel;
      }

}
/**
 * Thread that runs a time-consuming task before asking the EDT to change the given
JLable on the GUI
 * @author [redacted]
 *
 */
class ChangeTextSwingWorker extends SwingWorker<Integer, String>{

      //create the runnable object
      final Runnable changeTextRunnable = new Runnable(){
            public void run(){
                  text_display.setText("the text has been changed");
            }
      };

      //the JLable object to be changed
      private JLabel text_display;

      /**
       * creates a thread that waits for the given amount of time in seconds
       * @param time the amount of time to wait in seconds
       */
      ChangeTextSwingWorker(JLabel text_display){
            this.text_display = text_display;
      }
      /**
       * Runs the time-consuming task before asking the EDT to change the GUI
       */
      @Override
      protected Integer doInBackground() throws Exception {
            //the thread waits for 5 seconds before doing anything else
                  Thread.sleep(5000);


                  //queue up the change text runnable code
                  try{
                  SwingUtilities.invokeLater(changeTextRunnable);
                  }
                  catch(Exception e){
                        e.printStackTrace();
```

```
                              }
                              return 1;
              }

      }
```

Time Consuming Task:

```
package thread_portfolio.src;


import java.awt.Dimension;
import javax.swing.JFrame;

/**
 * The class containing the main method for the example of a Swing GUI with and
 * without a worker thread
 *
 * @author [redacted]
 *
 */
public class SwingWorkerExample {
	public String input;
	public int fontSize;
	public String fontType;
	public String color;

	/**
	 * main method that creates the frame, sets paramiters and adds the panels
	 * for both of the frames.
	 *
	 * @param args
	 */
	public static void main(String[] args) {

		// makes a new JFrame with with the "without worker threads panel" in it
		// THIS FRAME WILL FREEZE IF YOU CLICK A BUTTON!!!!!!
		// CAUTION!!!!!!!!! THIS WILL ALSO FREEZE THE OTHER FRAME BECAUSE I
		// OPENED THEM AT THE SAME TIME
		JFrame frame = new JFrame("WITHOUT WORKER THREADS!");
		frame.setSize(new Dimension(370, 220));
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

		frame.getContentPane().add(new WithoutWorkerThreadPanel(frame));
		frame.setVisible(true);

		// makes a new JFrame with the "with worker threads panel" in it
		// THIS FRAME WILL NOT FREEZE IF YOU CLICK A BUTTON!!!!
		JFrame frame2 = new JFrame("WITH WORKER THREADS!");
		frame2.setSize(new Dimension(370, 220));
		frame2.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

		frame2.getContentPane().add(new WithWorkerThreadPanel(frame2));
		frame2.setVisible(true);

	}

}


package thread_portfolio.src;
```

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.SwingWorker;

/**
 * class containing the constructor for creating the panel that uses a worker thread
to run a time-consuming process
 * @author [redacted]
 *
 */
public class WithWorkerThreadPanel extends JPanel {

	/**
	 * Local JFrame variable for holding the frame that this panel will be put in.
	 * Needed for displaying the dialog box
	 */
	private JFrame frame;

	/**
	 *Constructor for making the panel that uses a Swing Worker thread to do a
time consuming process
	 * @param frame- the frame that this panel is put in
	 */
	WithWorkerThreadPanel(JFrame frame) {
		//store the frame in the local variable
		this.frame = frame;

		//set the layout to borderLayout
		setLayout(new BorderLayout());
		//adds the waitbuttons to the center of the button layouts
		add(waitButtons(), BorderLayout.CENTER);
	}

	/**
	 * creates the panel for adding the wait buttons without using
	 * @return- the panel containing the buttons.
	 */
	private JPanel waitButtons() {

		//Create the panel containing the buttons
		JPanel waitButtons = new JPanel();

		//create the buttons
		JButton shortWait = new JButton("A really short wait");
		JButton waitFive = new JButton("Wait for 5 sec");
		JButton waitTen = new JButton("Wait for 10 sec");

		//add the buttons to the panel
```

```
             waitButtons.add(shortWait);
             waitButtons.add(waitFive);
             waitButtons.add(waitTen);

             // Action listener for the short wait button
             shortWait.addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent evt) {
                           // make a new worker thread that runs for 1 second
                           WaitButtonSwingWorker worker = new
WaitButtonSwingWorker(1,
                                         frame);
                           // start the worker thread
                           worker.execute();
                    }
             });

             // Action listener for the wait5 button
             waitFive.addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent evt) {
                           // make a new worker thread that runs for 5 second
                           WaitButtonSwingWorker worker = new
WaitButtonSwingWorker(5,
                                         frame);
                           // start the worker thread
                           worker.execute();
                    }
             });

             // Action listener for the wait10 button
             waitTen.addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent evt) {
                           // make a new worker thread that runs for 10 second
                           WaitButtonSwingWorker worker = new
WaitButtonSwingWorker(10,
                                         frame);
                           // start the worker thread
                           worker.execute();
                    }
             });

             // return the button panel
             return waitButtons;
      }
}

/**
 * Class that contains the constructor for the Swing Worker that does a time
consuming process then opens a dialog box saying it has compleated
 * @author [redacted]
 *
 */
class WaitButtonSwingWorker extends SwingWorker<Integer, String> {

      /**
       *Local time variable to hold integer amount of time this thread will wait
```

```
	 */
	private int time;
	/**
	 * Local frame variable to hold the frame that the dialog box will originate
from
	 */
	private JFrame frame;

	/**
	 * creates a thread that waits for the given amount of time in seconds
	 * 
	 * @param time
	 *            the amount of time to wait in seconds
	 */
	WaitButtonSwingWorker(int time, JFrame frame) {
		//set the local time variable
		this.time = time;
		//set the local frame variable
		this.frame = frame;
	}

	/**
	 * runs the worker thread in the background for the given amount of time.
	 */
	@Override
	protected Integer doInBackground() throws Exception {
		//sleep for time seconds
		Thread.sleep(time * 1000);
		//open the dialog box sayng that the task has been compleated.
		JOptionPane.showMessageDialog(frame,
				"This instnce of the worker thread has compleated!");
		return 1;
	}

}


package thread_portfolio.src;


import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;

/**
 * Class containing the constructor making the panel for a GUI without the
 * worker thread
 * 
 * @author [redacted]
 *
 */
public class WithoutWorkerThreadPanel extends JPanel {

	/**
	 * private JFrame for holding the frame that contains this panel. The frame
	 * is needed for displaying a dialog box
	 */
	private JFrame frame;

	/**
	 * Constructor for making the panel that does not use the worker thread
	 * example.
	 *
	 * @param frame
	 *           the frame containing this panel
	 */
	public WithoutWorkerThreadPanel(JFrame frame) {
		// set the local frame veriable
		this.frame = frame;

		setLayout(new BorderLayout());

		add(waitButtons(), BorderLayout.CENTER);
	}

	// creates the panel for adding the wait buttons without using
	private JPanel waitButtons() {
		//create the JPanel for the wait buttons
		JPanel waitButtons = new JPanel();
		//Creates the buttons
		JButton shortWait = new JButton("A really short wait");
		JButton waitFive = new JButton("Wait for 5 sec");
		JButton waitTen = new JButton("Wait for 10 sec");
		//adds the buttons to the waitbutton panel
		waitButtons.add(shortWait);
		waitButtons.add(waitFive);
		waitButtons.add(waitTen);

		// Action listener for the short wait button
		shortWait.addActionListener(new ActionListener() {
			//after pressing the button the Thread sleeps for one second then
opens a dialog box saying it has compleated the process
			public void actionPerformed(ActionEvent evt) {
				try {
					Thread.sleep(1000);
					JOptionPane.showMessageDialog(frame,
							"waiting is done so you can do stuff
again");
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
			}
		});
		// Action listener for the wait5 button
		waitFive.addActionListener(new ActionListener() {
			//after pressing the button the Thread sleeps for five second
then opens a dialog box saying it has compleated the process
			public void actionPerformed(ActionEvent evt) {
				try {
					Thread.sleep(5000);
					JOptionPane.showMessageDialog(frame,
							"waiting is done so you can do stuff
again");
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
			}
		});

		// Action listener for the wait10 button
		waitTen.addActionListener(new ActionListener() {
			//after pressing the button the Thread sleeps for ten second then
opens a dialog box saying it has compleated the process
			public void actionPerformed(ActionEvent evt) {
				try {
					Thread.sleep(10000);
					JOptionPane.showMessageDialog(frame,
							"waiting is done so you can do stuff
again");
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
			}
		});

		// return the button panel
		return waitButtons;
	}
}
```

Animation_example:

```java
package thread_portfolio;


/* AniMain.java - animation demo - Java 1.5 version - 2006 11 22
   revised version adds rules dialogue */

import java.awt.*; import java.awt.event.*;

import javax.swing.*;

import java.util.*;

class Circle extends Thread {

  //Properties of a Circle Object
      Graphics2D g;                       //graphics object for drawing
      Color c;                            //the Circle's colour
      Animation_example a;                //reference to the Main Frame
      int width,height;                 //size of Main Frame
      final int RADIUS=5;               //size of Circle to draw
      int    x,y;                          //location of Circle
      int xspeed=4, yspeed=4; //variables to control movement

//Circle constructor
      Circle(Animation_example a) {
   //assign the colour and main reference passed in
   this.a=a;
   //start the Circle Thread
   this.start();
   //obtain the width and height of Main Frame
   width=a.getWidth(); height=a.getHeight();
   //generate random location to display Circle for the first time
         x=(int) (Math.random()*(width-2*RADIUS))+RADIUS;
         y=(int) (Math.random()*(height-2*RADIUS))+RADIUS;
      }


public void run() {

      while(true) {
   //create the graphics object for drawing on the Frame
   g=(Graphics2D) a.getGraphics();
   //erase the rectangle first
   g.setColor(Color.white);
   g.drawOval(x-RADIUS,y-RADIUS,2*RADIUS,2*RADIUS);

   //Change the direction it's going based on its current position
         if (x <= RADIUS){
                xspeed =4;
         }
         else if (x >= width-RADIUS){
                xspeed = -4;
```

```
            }
            if (y <= RADIUS){
                   yspeed =4;
            }
            else if (y >= height-RADIUS){
                  yspeed=-4;
            }
            x += xspeed;
            y += yspeed;


            //draw the Circle in it's new position
            g.setColor(Color.RED);
            g.drawOval(x-RADIUS,y-RADIUS,2*RADIUS,2*RADIUS);

          //sleep the thread for a wee while between iterations
          try {Thread.sleep(100);}
          catch(Exception e) {}
        }
      }
    }

    public class Animation_example extends Frame{
      //the main controlling Thread object creates all Circle Threads
      Thread t;
      //the x & y coordinates of current mouse location
      //all Circle threads need to see these variables
      int x,y;
      //an ArrayList to hold all Circle Thread Objects
      ArrayList<Circle> threads=new ArrayList<Circle>();


      //Frame constructor
      public Animation_example() {

           //build GUI
        super("Moving circle");
        this.setBounds(100,100,400,400);
        this.setBackground(Color.white);

        //register events
        this.addWindowListener(new Window());
        this.setVisible(true);


        Graphics2D g=(Graphics2D) Animation_example.this.getGraphics();
        //set the color to white-same as Frame's background
        //draw a rectangle the size of the Frame
        g.setColor(Color.white);

    g.fillRect(0,0,Animation_example.this.getWidth(),Animation_example.this.getHeight());

        //construct and start the circle thread
        threads.add(new Circle(this));
```

```
 }


  //end the program when the user hits the Frame's close button
  class Window extends WindowAdapter
  { public void windowClosing(WindowEvent e) {System.exit(1);} }



  //main starts the program
  public static void main() { //create a graphics object to clear the frame
         new Animation_example();
}

}
```

Animation_utilities:

```
package thread_portfolio;

/* AniMain.java - animation demo - Java 1.5 version - 2006 11 22

 revised version adds rules dialogue */
import java.awt.*;
import java.awt.event.*;

import javax.swing.*;

import java.lang.reflect.InvocationTargetException;
import java.util.*;

class Circle extends Thread {

      // Properties of a Circle Object
      Graphics2D g; // graphics object for drawing
      Color c; // the Circle's color
      Animation_utilities a; // reference to the Main Frame
      int width, height; // size of Main Frame
      final int RADIUS = 5; // size of Circle to draw
      int x, y; // location of Circle
      int xspeed = 4, yspeed = 4; // variables to control movement

      // Circle constructor
      Circle(Animation_utilities a) {
            // assign the color and main reference passed in
            this.a = a;
            // start the Circle Thread
            this.start();
            // obtain the width and height of Main Frame
            width = a.getWidth();
            height = a.getHeight();
            // generate random location to display Circle for the first time
            x = (int) (Math.random() * (width - 2 * RADIUS)) + RADIUS;
            y = (int) (Math.random() * (height - 2 * RADIUS)) + RADIUS;
```

```
        }

        // create the runnable object for erasing the old circle
        final Runnable eraseOldCircle = new Runnable() {
                public void run() {
                        g.setColor(Color.white);
                        g.drawOval(x - RADIUS, y - RADIUS, 2 * RADIUS, 2 * RADIUS);
                }
        };

        // create the runnable object for creating the new circle
        final Runnable createNewCircle = new Runnable() {
                public void run() {
                        g.setColor(Color.RED);
                        g.drawOval(x - RADIUS, y - RADIUS, 2 * RADIUS, 2 * RADIUS);
                }
        };

        public void run() {

                while (true) {
                        // create the graphics object for drawing on the Frame
                        g = (Graphics2D) a.getGraphics();
                        // try to erase the rectangle first
                        try {
                                SwingUtilities.invokeAndWait(eraseOldCircle);
                        } catch (InvocationTargetException | InterruptedException e1) {
                                e1.printStackTrace();
                        }

                        // Change the direction it's going based on its current position
                        if (x <= RADIUS) {
                                xspeed = 4;
                        } else if (x >= width - RADIUS) {
                                xspeed = -4;
                        }
                        if (y <= RADIUS) {
                                yspeed = 4;
                        } else if (y >= height - RADIUS) {
                                yspeed = -4;
                        }
                        x += xspeed;
                        y += yspeed;

                        // draw the Circle in it's new position
                        try {
                                SwingUtilities.invokeAndWait(createNewCircle);
                        } catch (InvocationTargetException | InterruptedException e1) {
                                e1.printStackTrace();
                        }

                        // sleep the thread for a wee while between iterations
                        try {
                                Thread.sleep(100);
                        } catch (Exception e) {
                        }
                }
        }
}

public class Animation_utilities extends Frame {
        // the main controlling Thread object creates all Circle Threads
        Thread t;
        // the x & y coordinates of current mouse location
        // all Circle threads need to see these variables
        int x, y;
        // an ArrayList to hold all Circle Thread Objects
        ArrayList<Circle> threads = new ArrayList<Circle>();

        // Frame constructor
        public Animation_utilities() { // build GUI
                super("Moving circle");
                this.setBounds(100, 100, 400, 400);
                this.setBackground(Color.white);

                // register events
                this.addWindowListener(new Window());
                this.setVisible(true);

                Graphics2D g = (Graphics2D) Animation_utilities.this.getGraphics();
                // set the color to white-same as Frame's background
                // draw a rectangle the size of the Frame
                g.setColor(Color.white);
                g.fillRect(0, 0, Animation_utilities.this.getWidth(),
Animation_utilities.this.getHeight());

                // construct and start the main thread
                threads.add(new Circle(this));

        }

        // end the program when the user hits the Frame's close button
        class Window extends WindowAdapter {
                public void windowClosing(WindowEvent e) {
                        System.exit(1);
                }
        }

        // main starts the program
        public static void main(String[] args) { // create a graphics object to
                                                                               //
clear the frame
                new Animation_utilities();
        }
}
```