Contents

0 Case Study: NitPad: A Text Editor

Now we shall look at a simple, full-featured application that will be quite similar to the Notepad application you see on certain dark side machines.

We are going to think about this program from the standpoint of the user. We must be vigilant and protect the user from data loss. Since he is a paying customer, we must see carefully to his needs.

Let us begin by creating a graphical shell

```
import javax.swing.JFrame;
public class Nitpad extends JFrame implements Runnable
{
    public Nitpad()
    {
        super("Nitpad: Unsaved *");
    }
    public void run()
    {
        setSize(600,600);
        setVisible(true);
    }
    public static void main(String[] args)
    {
        Nitpad np = new Nitpad();
        javax.swing.SwingUtilities.invokeLater(np);
    }
}
```

Run this code and a blank window should appear on your screen with the string `"Nitpad: Unsaved *"` emblazoned on the title bar. Let us begin by making some design decisions. For the main text area, we will use a `JEditorPane`; this

is a flexible widget in which you may type text. We will put this inside of a `JScrollPane` and add it to the content pane. Placing it in a `JScrollPane` causes scrollbars to materialize when the text is too large to display in the content pane. Let us begin by doing that.

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JEditorPane;
public class Nitpad extends JFrame implements Runnable
{
    JEditorPane jep;
    public Nitpad()
    {
        super("Nitpad: Unsaved *");
        jep = new JEditorPane();
    }
    public void run()
    {
        setSize(600,600);
        getContentPane().add(new JScrollPane(jep));
        setVisible(true);
    }
    public static void main(String[] args)
    {
        Nitpad np = new Nitpad();
        javax.swing.SwingUtilities.invokeLater(np);
    }
}
```

Why is the `JEditorPane` a state variable? Open Notepad and look at it. You can see that its file and edit operations are driven by menus. Since this is to happen, it will be helpful to be able to communicate with `jep` throughout the entire program. Notice that we initialized this state variable in the second line of the constructor.

Run the program; observe that you can type characters into the `JEditorPane`.

0.1 Laying out Menus

Our program will have two menus, `File` and `Edit`. The file menu should have the standard list of menu items: New, Open, Save, Save As, and Quit. The Edit menu will feature Cut, Copy, Paste and Select All. Let us now create the menus and menu items so we can see them. First, remember we need these includes.

```
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
```

We then remember to do the following.

1. Nail in the menu bar with this code.

   ```
   JMenuBar mbar = new JMenuBar();
   setJMenuBar(mbar);
   ```

2. Create and add in the two menus.

   ```
   JMenu fileMenu = new JMenu("File");
   JMenu editMenu = new JMenu("Edit");
   mbar.add(fileMenu);
   mbar.add(editMenu);
   ```

3. Populate the menus.

   ```
   JMenuItem newItem = new JMenuItem("New");
   JMenuItem openItem = new JMenuItem("Open");
   JMenuItem saveItem = new JMenuItem("Save");
   JMenuItem saveAsItem = new JMenuItem("Save As...");
   JMenuItem printItem = new JMenuItem("Print");
   JMenuItem quitItem = new JMenuItem("Quit");
   fileMenu.add(newItem);
   fileMenu.add(openItem);
   fileMenu.add(saveItem);
   fileMenu.add(saveAsItem);
   fileMenu.add(printItem);
   fileMenu.add(quitItem);
   JMenuItem copyItem = new JMenuItem("Copy");
   JMenuItem pasteItem = new JMenuItem("Paste");
   JMenuItem cutItem = new JMenuItem("Cut");
   JMenuItem selectAllItem = new JMenuItem("Select All");
   editMenu.add(copyItem);
   editMenu.add(pasteItem);
   editMenu.add(cutItem);
   editMenu.add(selectAllItem);
   ```

Here is our latest result. Run it and see all of the menus and menu items being displayed. We have created much of the view for this application and a very little of the model. However, we should be pleased by the app's appearance. We also added here code to set the font in the constructor. Notice the appearance of another import.

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
```

```
import javax.swing.JEditorPane;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import java.awt.Font;
public class Nitpad extends JFrame implements Runnable
{
    JEditorPane jep;
    public Nitpad()
    {
        super("Nitpad: Unsaved *");
        jep = new JEditorPane();
        jep.setFont(new Font("Monospaced", Font.PLAIN, 12));
    }
    public void run()
    {
        setSize(600,600);
        getContentPane().add(new JScrollPane(jep));
        makeMenus();
        setVisible(true);
    }
    public void makeMenus()
    {
        JMenuBar mbar = new JMenuBar();
        setJMenuBar(mbar);
        JMenu fileMenu = new JMenu("File");
        JMenu editMenu = new JMenu("Edit");
        mbar.add(fileMenu);
        mbar.add(editMenu);
        JMenuItem newItem = new JMenuItem("New");
        JMenuItem openItem = new JMenuItem("Open");
        JMenuItem saveItem = new JMenuItem("Save");
        JMenuItem saveAsItem = new JMenuItem("Save As...");
        JMenuItem printItem = new JMenuItem("Print");
        JMenuItem quitItem = new JMenuItem("Quit");
        fileMenu.add(newItem);
        fileMenu.add(openItem);
        fileMenu.add(saveItem);
        fileMenu.add(saveAsItem);
        fileMenu.add(printItem);
        fileMenu.add(quitItem);
        JMenuItem copyItem = new JMenuItem("Copy");
        JMenuItem pasteItem = new JMenuItem("Paste");
        JMenuItem cutItem = new JMenuItem("Cut");
        JMenuItem selectAllItem = new JMenuItem("Select All");
        editMenu.add(copyItem);
```

```
        editMenu.add(pasteItem);
        editMenu.add(cutItem);
        editMenu.add(selectAllItem);
    }
    public static void main(String[] args)
    {
        Nitpad np = new Nitpad();
        javax.swing.SwingUtilities.invokeLater(np);
    }
}
```

0.2 Getting a File to Save via Menus

We begin by attaching empty action listeners to each `File` menu item. For example, in the `New` menu item, we proceed as follows.

```
newItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
    }
});
```

Do not forget to add these imports.

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

Proceed to attach an empty action listener to each `Edit` menu item.

Before going any further, we must make a plan for how these items are to work. *We must think about the user experience and design appropriately.* It is likely and desirable that we will develop some private methods that will be used by the listeners to do their jobs. Once we make a plan, it will be clear what functions will be executed by more than one listener. Said function should be coded as private methods.

Now we must think first about the state of our program. So far, it has a `JEditorPane` which can accept typed text. However, this pane does not know what to do with the text. What other state variables do we need?

Open Notepad on a Windoze machine. Any instance of Notepad can open exactly one file. Either we are typing in an unsaved window or a window pointing at a file. This tells us we need a `File` object representing our current file. When Nitpad is first started, we will make it a `null` object. Add a state variable as follows.

```
private File currentFile;
```

Add this line to the constructor.

```
currentFile = null;
```

Let us begin by trying to save a file. We need to plan the action of the listener attached to the `Save` menu item. Let us now write a plan in comments in its code.

```
saveItem.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            //if currentFile is null, prompt to select
            //a file as the current file

            //write to the current file

            //place the absolute path of the current file
            //in the title bar.
        }
    });
```

Now let us look at `Save As` before moving ahead.

```
saveAsItem.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            //prompt to select
            //a file as the current file to save to.

            //write to the current file

            //place the absolute path of the current file
            //in the title bar.
            }

        }
    });
```

You can see some commonalities in the functions of the two menu item's action listeners. Both prompt to select a current file. Both write to the current file. Both update the title bar. This points up the need for three private methods. Notice how we pass the buck with the `IOException` in the first method. You can add these method stubs to your class. Make sure you import the class

`java.io.IOException`. Note we also devleop a function to select a currentFile, because it will be a common operation

```
private void  writeToCurrentFile() throws IOException{}
private void fixTitleBar(){}
private void promptToSave() {}
private void selectCurrentFile() {}
```

Let us try coding up these methods, beginning with `writeToCurrentFile()`. rere are the basic steps.

1. Create a buffered writer linked to the current file.
2. Extract the text from the editor pane.
3. Write the text to the file
4. Close the file connection so the file is properly saved.

Now we code it up.

```
public void  writeToCurrentFile() throws IOException
{
   //Create a buffered write linked to the current file.
    BufferedWriter bw = new BufferedWriter(
        new FileWriter(currentFile));
    //Extract the text from the editor pane.
    String blob = jep.getText();
    //Write the text to a file.
    bw.write(blob);
    //Close the file coonection.
    bw.close();
}
```

You might properly ask: *What if the current file is null?* We have blithely ignored that possibility. Coördinating that problems is best left to the action listener. We are adhering to the design prinicple of *atomicity of purpose*; i.e. our function does exactly one thing. Other objects will orchestrate its behavior. It is a precondition of this function that a valid current file be selected.

It is now time to test this out. We are going to show a primitive version of our app that just saves to a file named `hammer.txt`. Here is the file with the additions we have made. We must now place code in the `Save` action listener to get things going.

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
```

```
import javax.swing.JEditorPane;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import java.io.File;
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Nitpad extends JFrame implements Runnable
{
    JEditorPane jep;
    private File currentFile;
    public Nitpad()
    {
        super("Nitpad: Unsaved *");
        jep = new JEditorPane();
        jep.setFont(new Font("Monospaced", Font.PLAIN, 12));
        currentFile = new File("hammer.txt");  //temporary code to
        //test the writeToCurrentFile method.  TODO: Get rid of this.
    }
    public void run()
    {
        setSize(600,600);
        getContentPane().add(new JScrollPane(jep));
        makeMenus();
        setVisible(true);
    }
    public void makeMenus()
    {

        JMenuBar mbar = new JMenuBar();
        setJMenuBar(mbar);
        JMenu fileMenu = new JMenu("File");
        JMenu editMenu = new JMenu("Edit");
        mbar.add(fileMenu);
        mbar.add(editMenu);
        JMenuItem newItem = new JMenuItem("New");
        JMenuItem openItem = new JMenuItem("Open");
        JMenuItem saveItem = new JMenuItem("Save");
        JMenuItem saveAsItem = new JMenuItem("Save As...");
        JMenuItem quitItem = new JMenuItem("Quit");
        fileMenu.add(newItem);
```

```
fileMenu.add(openItem);
fileMenu.add(saveItem);
fileMenu.add(saveAsItem);
fileMenu.add(quitItem);
JMenuItem copyItem = new JMenuItem("Copy");
JMenuItem pasteItem = new JMenuItem("Paste");
JMenuItem cutItem = new JMenuItem("Cut");
JMenuItem selectAllItem = new JMenuItem("Select All");
editMenu.add(copyItem);
editMenu.add(pasteItem);
editMenu.add(cutItem);
editMenu.add(selectAllItem);
newItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
    }
});
openItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
    }
});
saveItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        //if currentFile is null, prompt to select
        //a file as the current file

        //write to the current file

        //place the absolute path of the current file
        //in the title bar.
    }
});
saveAsItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        //prompt to select
        //a file as the current file to save to.

        //write to the current file

        //place the absolute path of the current file
        //in the title bar.
    }
});
```

```
        quitItem.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e)
            {
            }
        });

    }
    private void fixTitleBar(){}
    private void promptToSave(){}
    private void selectCurrentFile(){}

    private void writeToCurrentFile() throws IOException
    {
       //Create a buffered write linked to the current file.
        BufferedWriter bw = new BufferedWriter(
            new FileWriter(currentFile));
        //Extract the text from the editor pane.
        String blob = jep.getText();
        //Write the text to a file.
        bw.write(blob);
        //Close the file connection.
        bw.close();
    }
    public static void main(String[] args)
    {
        Nitpad np = new Nitpad();
        javax.swing.SwingUtilities.invokeLater(np);
    }
}
```

Let us now code this action listener. This is very minimal, but all we want to do is to test and see if the current file is getting written to `hammer.txt`. Add this to your class.

```
saveItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        //if currentFile is null, prompt to select
        //a file as the current file

        //write to the current file
        try
        {
            writeToCurrentFile();
        }
        catch(IOException ex)
```

```
        {
            System.err.println("Not happy");
        }

        //place the absolute path of the current file
        //in the title bar.
    }
});
```

Now place text in the `JEditorPane` and select `Save` from the `File` menu. Close the application. You should have a file `hammer.txt` with the text you typed in it.

0.3 Is the Window Saved?

If we quit or application or navigate away from a file, we need to know if the the contents of the window are saved. This is necessary becasue we must protect the user from inadvertant data loss. Hence, we will add the state variable

```
private boolean saved;
```

In the constructor, we initialize as follows.

```
saved = false;
```

This brings up a whole set of problems we have to deal with.

- The `saved` state variable must be updated whenever the contents of the window change. How do we do that?
- As part of our interface, we will put a * at the end of the title bar when the contents of the window are not saved. How do we make this happen?
- Where are the perils of data loss? How do we prevent them? We must think that whenever we display a new file in the window, the changes to the old one must be saved.
- If the current file is null, we must act as if the window is unsaved.

Let us begin by getting up a mechanism to point to a current file. For now, you will need to add this empty methd which we will fill soon.

```
private void writeToCurrentFile() throws IOException(){}
```

Next, let us dispose of the matter of selecting the current file. Let us return `true` if a file is selected and `false` otherwise.

```
private boolean selectCurrentFile()
{
    boolean chosen = false;
    JFileChooser jfc = new JFileChooser();
    int willSave = jfc.showSaveDialog(Nitpad.this);
    if(willSave == JFileChooser.APPROVE_OPTION)
    {
        currentFile = jfc.setSelectedFile();
        chosen = true;
    }
    return chosen;
}
```

Note that we have changed the return type of this function, since we have seen a need to do so. The caller will need to know if a file is selected and saved to so it can act accordingly. Note the use of `JOptionPane`'s static constants to do the job. These are the integers that are returned by this method.

```
private int promptToSaveWindow()
{
    int choice = JOptionPane.showConfirmDialog(Nitpad.this,
        "Save document in window?");
    boolean selected = false;
    if(choice == JOptionPane.YES_OPTION)
    {
        selected = selectCurrentFile();
    }
    try
    {
        if(selected)
        {
            writeToCurrentFile();
        }
        else
        {
            choice = JOptionPane.CANCEL_OPTION;
        }

    }
    catch(Exception ex)
    {
        //bail out conservatively
        choice = JOptionPane.CANCEL_OPTION;
    }
    return choice;
}
```

Now we shold get rid of the temporay refrence to `hammer.txt` at the beginning and alter the constructor to read

```
currentFile = null;
```

Another matter of housekeeping is the title bar. Let's get it at least partially working.

```
    private void fixTitleBar()
    {
        //TODO  fix star
        String currentFileString = (currentFile == null)? "Unsaved":
            currentFile.getAbsolutePath();
        setTitle("Nitpad:  " + currentFileString);
    }
```

0.4 Getting Save and Save As to Work

Let us now turn our attention to the `actionPerformed` method in the `saveItem` action listener. Here is what we have so far.

```
saveItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        //if currentFile is null, prompt to select
        //a file as the current file

        //write to the current file
        try
        {
            writeToCurrentFile();
        }
        catch(IOException ex)
        {
            System.err.println("Not happy");
        }

        //place the absolute path of the current file
        //in the title bar.
    }
});
```

If the current file is null we can select as follows

```
boolean saveWanted = false;
if(currentFile == null)
    saveWanted = selectCurrentFile();
```

Now we can make writing to the current file conditional on the save being wanted.

```
if(saveWanted)
{
    writeToCurrentFile();
    fixTitleBar();
    saved = true;
}
```

Now the method looks like this.

```
saveItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        //if currentFile is null, prompt to select
        //a file as the current file
        boolean saveWanted = false;
        if(currentFile == null)
            saveWanted = selectCurrentFile();
        //write to the current file
        try
        {
            if(saveWanted)
            {
                writeToCurrentFile();
                //place the absolute path of the current file
                //in the title bar.
                fixTitleBar();
                saved = true;
            }
        }
        catch(IOException ex)
        {
            System.err.printf("Could not open %s\n",
                currentFile.getAbsolutePath());
        }
    }
});
```

Now we can write the `Save As` action listener pretty easily.

```
saveAsItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        //prompt to select
        //a file as the current file to save to.
        boolean saveWanted = false;
        saveWanted = selectCurrentFile();
        //write to the current file and
        //place the absolute path of the current file
        //in the title bar.
        try
        {
            if(saveWanted)
            {
                writeToCurrentFile();
                fixTitleBar();
                saved = true;
            }
        }
        catch(IOException ex)
        {
            System.err.printf("Could not open %s\n",
                currentFile.getAbsolutePath());
        }
    }
});
```

0.5 Getting the `File` Menu in Order

Now we will turn to writing the rest of the action listeners for the `File` menu. This will require some careful design on our part to prevent data loss for the user. Let us begin with `New`.

If the current file is unsaved and `New` is selected, we must offer the user the opportunity to save his file before he navigates away from the window and loses his work. We force the user to deliberately abandon the contents of the window.

Once this is done, we set `currentFile` to `null` and we clear all text from the `JEditorPane`.

Be warned, we do not have the feature in place that monitors whether the text in the window is saved in full working order. We will do that after we get the menu items working. Let us plunge ahead recklessly, assuming that this desired feature is working. We begin with `New`; first we show the outline in comments.

```
newItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        //If the contents of the window are unsaved,
        //prompt the user to save them.
        //Then, clear the JEditorPane and set the current
        //file to null.
    }
});
```

We now insert the code to make this work. Notice how we protect the user from losing data in the window and how we give lots of opportunity to back out whilst doing nothing.

```
newItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        //If the contents of the window are unsaved,
        //prompt the user to save them.
        int saveWindowChoice = -1;
        int saveFileChoice = -1;
        if(!saved)
        {
           saveWindowChoice = promptToSave();
            if(saveWindowChoice == JOptionPane.CANCEL_OPTION)
                ;//do nothing
            else if(saveWindowChoice == JOptionPane.NO_OPTION)
            {
                currentFile = null;
                jep.setText("");  //abandon changes
            }
            if(saveWindowChoice == JOptionPane.YES_OPTION)
            {
                if(currentFile == null)
                {
                    saveFileChoice = selectCurrentFile();
                    if(saveFileChoice)
                    {
                        writeToCurrentFile();
                    }
                }
            }
        }
        //Then, clear the JEditorPane and set the current
        //file to null.
    }
```

```
});
```