POP MINI PROJECT REPORT

DEPARTMENT OF INFORMATION TECHNOLOGY

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA

COURSE TITLE

Paradigms Of Programming(IT206)

SUBMITTED TO :

Mr Pradeep Jagannath

SUBMITTED BY :

1.AKRITI KUMARI (15IT107)
2.ANKITA BHALAVI (15IT108)
3.RENU CHOUDHARY (15IT136)
4.SUPRIYA SAHOO (15IT145)

SUBMISSION DATE :

18 NOVEMBER 2016

ACKNOWLEDGEMENT

The success and final outcome of this project required a lot of guidance and assistance from many people and we are extremely fortunate to have got this all along the completion of our project work. We would like to express special gratitude to our course instructor Mr.Pradeep Jagannath for giving us this opportunity to do this project in Paradigms of Programming . Your support, motivation and advice helped us to finish the project in the prescribed time. We owe our profound gratitude to our seniors, who guided us all along till the completion of our project work by providing all the necessary information for developing the project.

PROJECT ANALYSIS

PACKAGES REQUIRED :

- import java.awt.*;
- import java.awt.event.*;
- import java.io.*;
- import javax.swing.*;
- import javax.swing.text.*;

The **java**.**awt**.**event package** defines classes and interfaces used for **event** handling in the **AWT** and Swing. The members of this **package** fall into three categories: **Events**. The classes with names ending in "**Event**" represent specific types of **events**, generated by the **AWT** or by one of the **AWT** or Swing components.

Java.io package provides classes for system input and output through data streams, serialization and the file system.This reference will take you through simple and practical methods available in java.io package.

The javax.swing.text package contains the powerful `JTextComponent` text editor and all of its supporting infrastructure. The `JTextField`, `JTextArea`, `JEditorPane`, and other text input components of the `javax.swing` package all subclass `JTextComponent` and rely on the other classes and interfaces of this package.

The `Document` interface defines the data model for the `JTextComponent`. It is the basic abstraction for documents that can be displayed and edited. The `AbstractDocument` class implements this interface and provides a number of useful features and extensions. `StyledDocument` extends `Document` to define support for documents that have styles associated with their content. `DefaultStyledDocument` is a concrete implementation based on `AbstractDocument`. Other important classes and interfaces in this package include: `EditorKit`, `Element`, `View`, `AbstractDocument.Content`, `Caret`, and `Highlighter`.

The class

```
class TextEditor extends JFrame {
```

Base variables.

```
private JTextArea area = new JTextArea(20,120);

    private JFileChooser dialog = new JFileChooser(System.getProperty("user.dir"));

    private String currentFile = "Untitled";

    private boolean changed = false;
```

Our JTextArea will be the area where you can write documents and etc...

Constructor Part #1

```
public TextEditor() {

        area.setFont(new Font("Monospaced",Font.PLAIN,12));

        JScrollPane scroll = new
JScrollPane(area,JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,JScrollPane.HORIZONTAL
_SCROLLBAR_ALWAYS);

        add(scroll,BorderLayout.CENTER);


        JMenuBar JMB = new JMenuBar();

        setJMenuBar(JMB);

        JMenu file = new JMenu("File");

        JMenu edit = new JMenu("Edit");

        JMB.add(file); JMB.add(edit);
```

So here we are our area and the toolbar we will use... with JScrollPane we will be able to view a bit more of our text and area.

Constructor Part #2

```
file.add(New);file.add(Open);file.add(Save);
        file.add(Quit);file.add(SaveAs);
        file.addSeparator();

        for(int i=0; i<4; i++)
            file.getItem(i).setIcon(null);

        edit.add(Cut);edit.add(Copy);edit.add(Paste);

        edit.getItem(0).setText("Cut out");
        edit.getItem(1).setText("Copy");
        edit.getItem(2).setText("Paste");
```

Now by adding methods and arguments we haven't implemented from the beginning will of course complain in an active IDE.
While adding we also position them in a correct order with our for loop and set every icon to *null*.

Constructor Part #3

```
JToolBar tool = new JToolBar();
        add(tool,BorderLayout.NORTH);
        tool.add(New);tool.add(Open);tool.add(Save);
        tool.addSeparator();

        JButton cut = tool.add(Cut), cop = tool.add(Copy),pas = tool.add(Paste);

        cut.setText(null); cut.setIcon(new ImageIcon("cut.gif"));
```

```
        cop.setText(null); cop.setIcon(new ImageIcon("copy.gif"));
        pas.setText(null); pas.setIcon(new ImageIcon("paste.gif"));

        Save.setEnabled(false);
        SaveAs.setEnabled(false);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        area.addKeyListener(k1);
        setTitle(currentFile);
        setVisible(true);
    }
```

We are currently adding our functions and giving some proper icons for our toolbar items ! Note we are adding in icons and giving null text to each item.
Our Save and SaveAs functions will represent the standard every time we launch the application...hence it's false...

Listener

```
private KeyListener k1 = new KeyAdapter() {
        public void keyPressed(KeyEvent e) {
            changed = true;
            Save.setEnabled(true);
            SaveAs.setEnabled(true);
        }
    };
```

Now here we have our listener which will listen when any key is pressed thus making the changed statement true and allowing us to save or save as.

Action #1

```
Action Open = new AbstractAction("Open", new ImageIcon("open.gif")) {
        public void actionPerformed(ActionEvent e) {
            saveOld();
            if(dialog.showOpenDialog(null)==JFileChooser.APPROVE_OPTION) {
                readInFile(dialog.getSelectedFile().getAbsolutePath());
            }
            SaveAs.setEnabled(true);
        }
    };
```

By pressing on our ItemIcon we will trigger an Action(Abstract one). Also by triggering the Icon action we will be able to preserve the moment of our text. Resulting us to be able choose where to save and use it later :>

Action #2

```
Action Save = new AbstractAction("Save", new ImageIcon("save.gif")) {
        public void actionPerformed(ActionEvent e) {
            if(!currentFile.equals("Untitled"))
                saveFile(currentFile);
            else
                saveFileAs();
        }
    };
```

So here is our save function. We will be able to save our current file. So by playing along I thought like this. If the currentFile doesn't equal to Untitled then just save it as it's called... else save as thus allowing the sure to give it a title

Action #3

```
Action SaveAs = new AbstractAction("Save as...") {
        public void actionPerformed(ActionEvent e) {
            saveFileAs();
        }
    };
```

Nothing much with this action. We are just allowing the user to saveAs...thus giving it a title...

Action #4

```
Action Quit = new AbstractAction("Quit") {
        public void actionPerformed(ActionEvent e) {
            saveOld();
            System.exit(0);
        }
    };
```

A quit function is needed also we are saving the old one. Which means... we will see later on

Action Maping

```
ActionMap m = area.getActionMap();
    Action Cut = m.get(DefaultEditorKit.cutAction);
    Action Copy = m.get(DefaultEditorKit.copyAction);
    Action Paste = m.get(DefaultEditorKit.pasteAction);
```

By allowing us, to give absolute correct power we are using the DefaultEditorKit in the Swing package...(IO too..)

Methods #1

```
private void saveFileAs() {
        if(dialog.showSaveDialog(null)==JFileChooser.APPROVE_OPTION)
```

```
            saveFile(dialog.getSelectedFile().getAbsolutePath());
    }
```

So now we have defined what saveFileAs will result to. However what will saveFile do, is something I will return to.

Methods #2

```
private void saveOld() {
        if(changed) {
            if(JOptionPane.showConfirmDialog(this, "Would you like to save "+ currentFile +" ?","Save",JOptionPane.YES_NO_OPTION)== JOptionPane.YES_OPTION)
                saveFile(currentFile);
        }
    }
```

By using a saveOld giving us an option to save the changed state thus naming it old.

Method #3

```
private void readInFile(String fileName) {
        try {
            FileReader r = new FileReader(fileName);
            area.read(r,null);
            r.close();
            currentFile = fileName;
            setTitle(currentFile);
            changed = false;
        }
        catch(IOException e) {
            Toolkit.getDefaultToolkit().beep();
```

```
                JOptionPane.showMessageDialog(this,"Editor can't find the file called
"+fileName);
            }
        }
```

As we are allowed to save, how do we open a file? Easy making a filereader method.

Method #4

```
private void saveFile(String fileName) {
            try {
                FileWriter w = new FileWriter(fileName);
                area.write(w);
                w.close();
                currentFile = fileName;
                setTitle(currentFile);
                changed = false;
                Save.setEnabled(false);
            }
            catch(IOException e) {
            }
        }
```

Our savefile method. Allowing the user to define a name for the title as well to his/her document.

The main

```
public  static void main(String[] arg) {
            new TextEditor();
        }
}
```