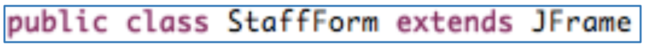
Techniques Used:

- Structural Hierarchy of classes
- GUI Features and Swing
- Database Complexity and Features
- Inheritance
- Polymorphism
- Encapsulation
- Creating and Parsing a Text File
- Abstract Data Structures
- Additional Libraries

Structural Hierarchy of Classes

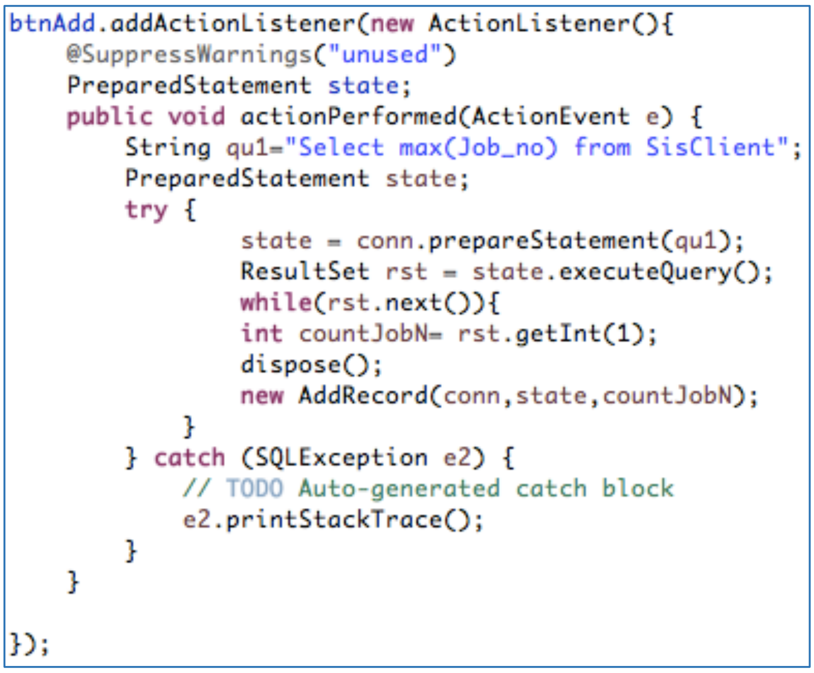
The Model-View-Controller Framework has been used to relate classes and maintain data validity and consistency. The classes *StaffForm.java, AddRecord.java, AddRecord_1.java, AddRecord_2.java, AddRecord_3.java* make up this structure.



```java
public class StaffForm extends JFrame
```

Figure 1A – StaffForm is the Model class

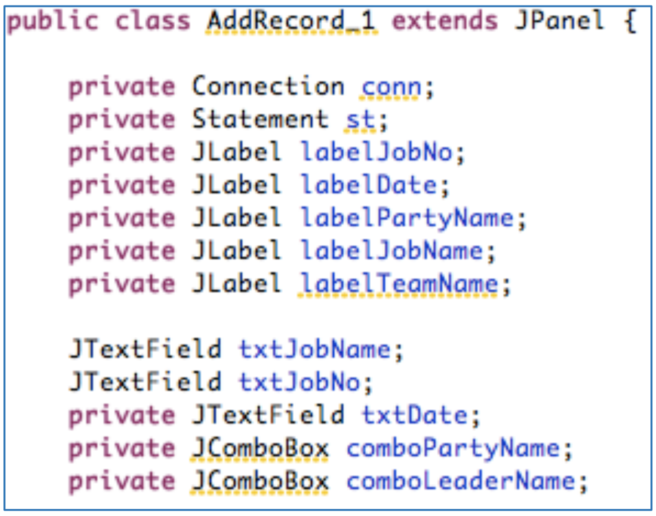
The above class holds the Constructor *StaffForm(Connection conn, Statement st)* which puts an action-listener to *btnAdd (Figure 1B)*, when pressed, opens the form *AddRecord.java,* while fetching data from the database and passing it-to *AddRecord.java*.



```java
btnAdd.addActionListener(new ActionListener(){
    @SuppressWarnings("unused")
    PreparedStatement state;
    public void actionPerformed(ActionEvent e) {
        String qu1="Select max(Job_no) from SisClient";
        PreparedStatement state;
        try {
                state = conn.prepareStatement(qu1);
                ResultSet rst = state.executeQuery();
                while(rst.next()){
                int countJobN= rst.getInt(1);
                dispose();
                new AddRecord(conn,state,countJobN);
            }
        } catch (SQLException e2) {
            // TODO Auto-generated catch block
            e2.printStackTrace();
        }
    }

});
```

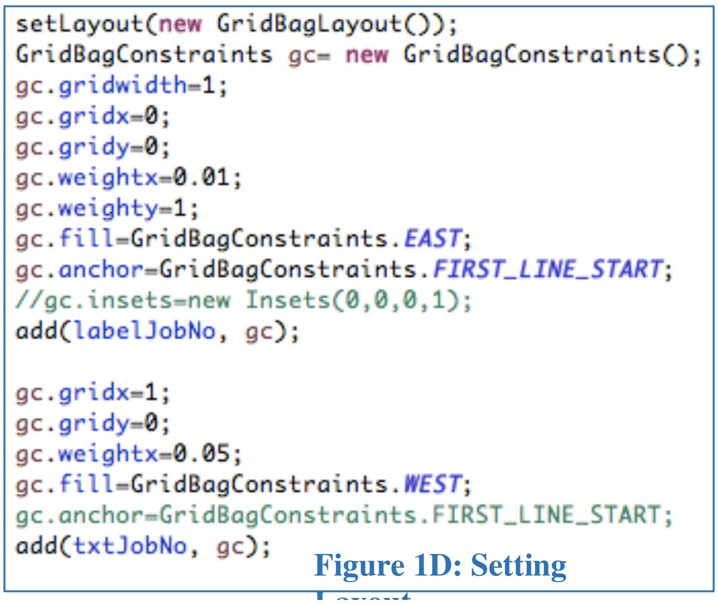
Figure 1B: Action Listener in StaffForm.java used to pass connection of the DB to AddRecord class

StaffForm.java acts as the model class as it provides data to *AddRecord*.java, by passing the connection of the database and running a query which allows the *Job-Number* (primary-key in the database) to be updated in *AddRecord*.java.



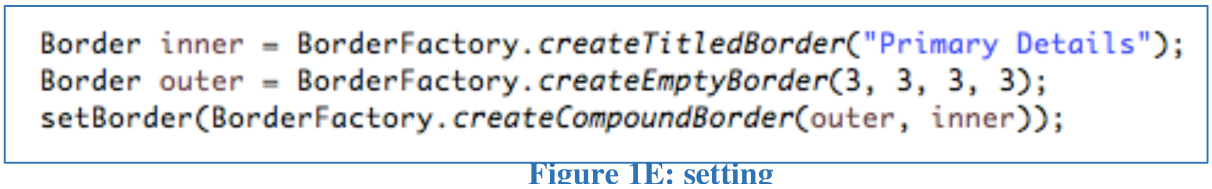
```
public class AddRecord_1 extends JPanel {

    private Connection conn;
    private Statement st;
    private JLabel labelJobNo;
    private JLabel labelDate;
    private JLabel labelPartyName;
    private JLabel labelJobName;
    private JLabel labelTeamName;

    JTextField txtJobName;
    JTextField txtJobNo;
    private JTextField txtDate;
    private JComboBox comboPartyName;
    private JComboBox comboLeaderName;
```

Figure 1C



```
setLayout(new GridBagLayout());
GridBagConstraints gc= new GridBagConstraints();
gc.gridwidth=1;
gc.gridx=0;
gc.gridy=0;
gc.weightx=0.01;
gc.weighty=1;
gc.fill=GridBagConstraints.EAST;
gc.anchor=GridBagConstraints.FIRST_LINE_START;
//gc.insets=new Insets(0,0,0,1);
add(labelJobNo, gc);

gc.gridx=1;
gc.gridy=0;
gc.weightx=0.05;
gc.fill=GridBagConstraints.WEST;
gc.anchor=GridBagConstraints.FIRST_LINE_START;
add(txtJobNo, gc);
```

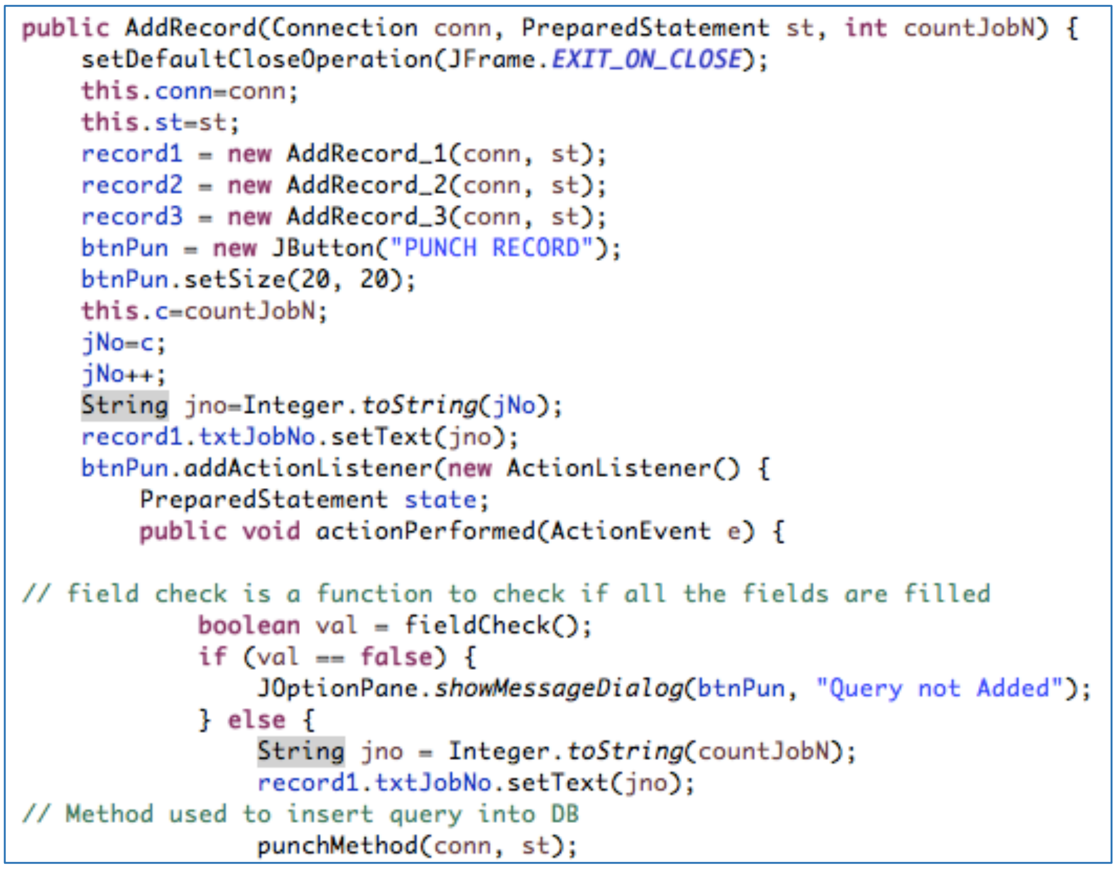
Figure 1D: Setting



```
Border inner = BorderFactory.createTitledBorder("Primary Details");
Border outer = BorderFactory.createEmptyBorder(3, 3, 3, 3);
setBorder(BorderFactory.createCompoundBorder(outer, inner));
```

Figure 1E: setting

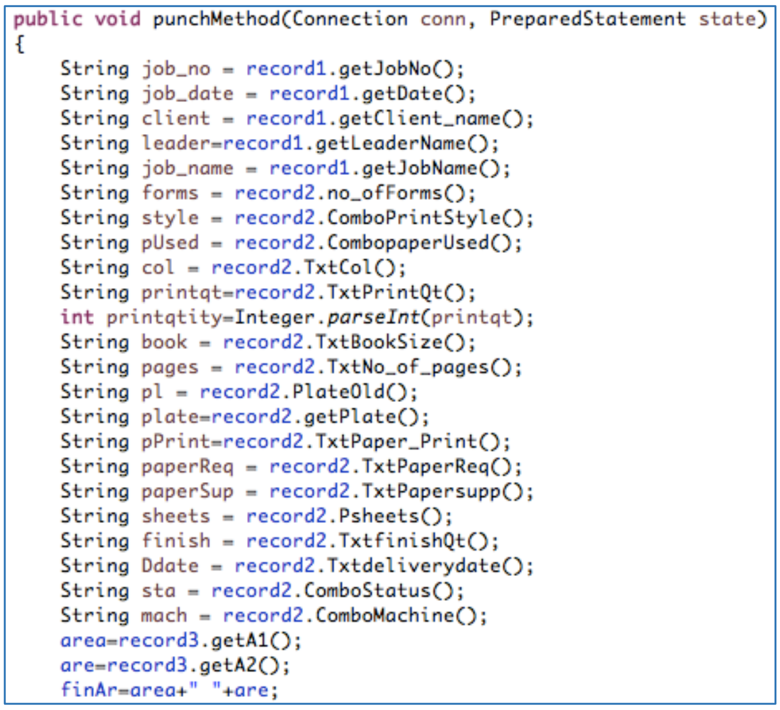
The above code snapshots are from the *AddRecord_1* class, which makes it the View-Class in the MVC framework as it contains the elements of the graphical-user-interface such as text-fields, combo-boxes, labels etc. that allow the user to interact with the form. The classes *AddRecord_2.java* and *AddRecord_3.java* have similarly been coded to become the View-Classes. (Refer to page 5, Appendix, to see the code).



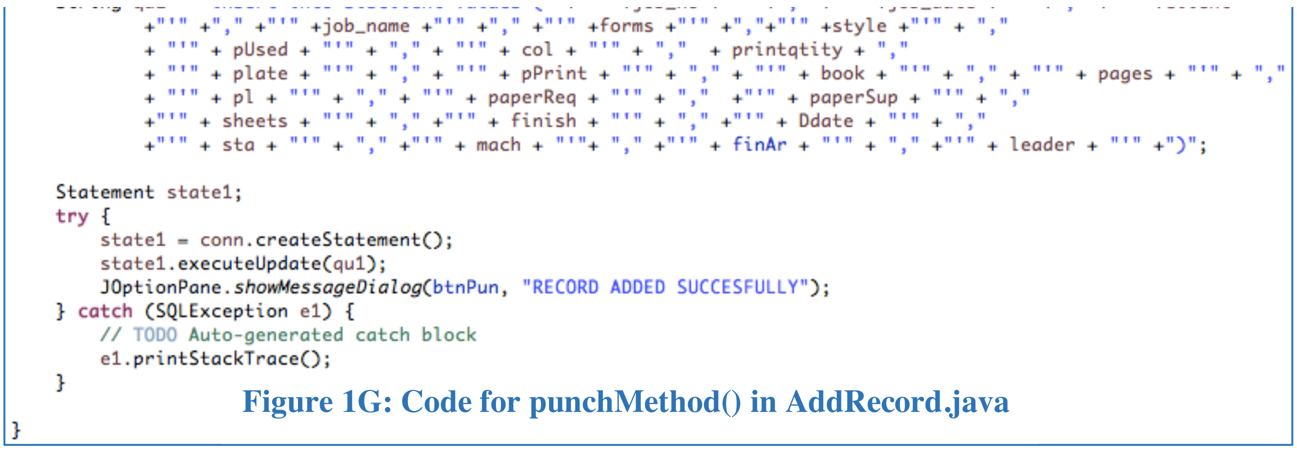
```
public AddRecord(Connection conn, PreparedStatement st, int countJobN) {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.conn=conn;
    this.st=st;
    record1 = new AddRecord_1(conn, st);
    record2 = new AddRecord_2(conn, st);
    record3 = new AddRecord_3(conn, st);
    btnPun = new JButton("PUNCH RECORD");
    btnPun.setSize(20, 20);
    this.c=countJobN;
    jNo=c;
    jNo++;
    String jno=Integer.toString(jNo);
    record1.txtJobNo.setText(jno);
    btnPun.addActionListener(new ActionListener() {
        PreparedStatement state;
        public void actionPerformed(ActionEvent e) {

// field check is a function to check if all the fields are filled
            boolean val = fieldCheck();
            if (val == false) {
                JOptionPane.showMessageDialog(btnPun, "Query not Added");
            } else {
                String jno = Integer.toString(countJobN);
                record1.txtJobNo.setText(jno);
// Method used to insert query into DB
                punchMethod(conn, st);
```

Figure 1F: fieldCheck() and punchMethod() in AddRecord.java

The following snapshot is from *AddRecord* class, which acts as the Controller-class, because, it controls the data coming from the user, using the function *punchMethod*(), which uses SQL-query to insert user data into the database. It also controls the data communicated to the user from the model by either showing failure messages when the *punchMethod*() query fails, or, if any fields in the *StaffForm* are unfilled by the user. The Method *fieldCheck*() is called before *punchmethod()* to ensure all the fields are given a value in the *StaffForm*.



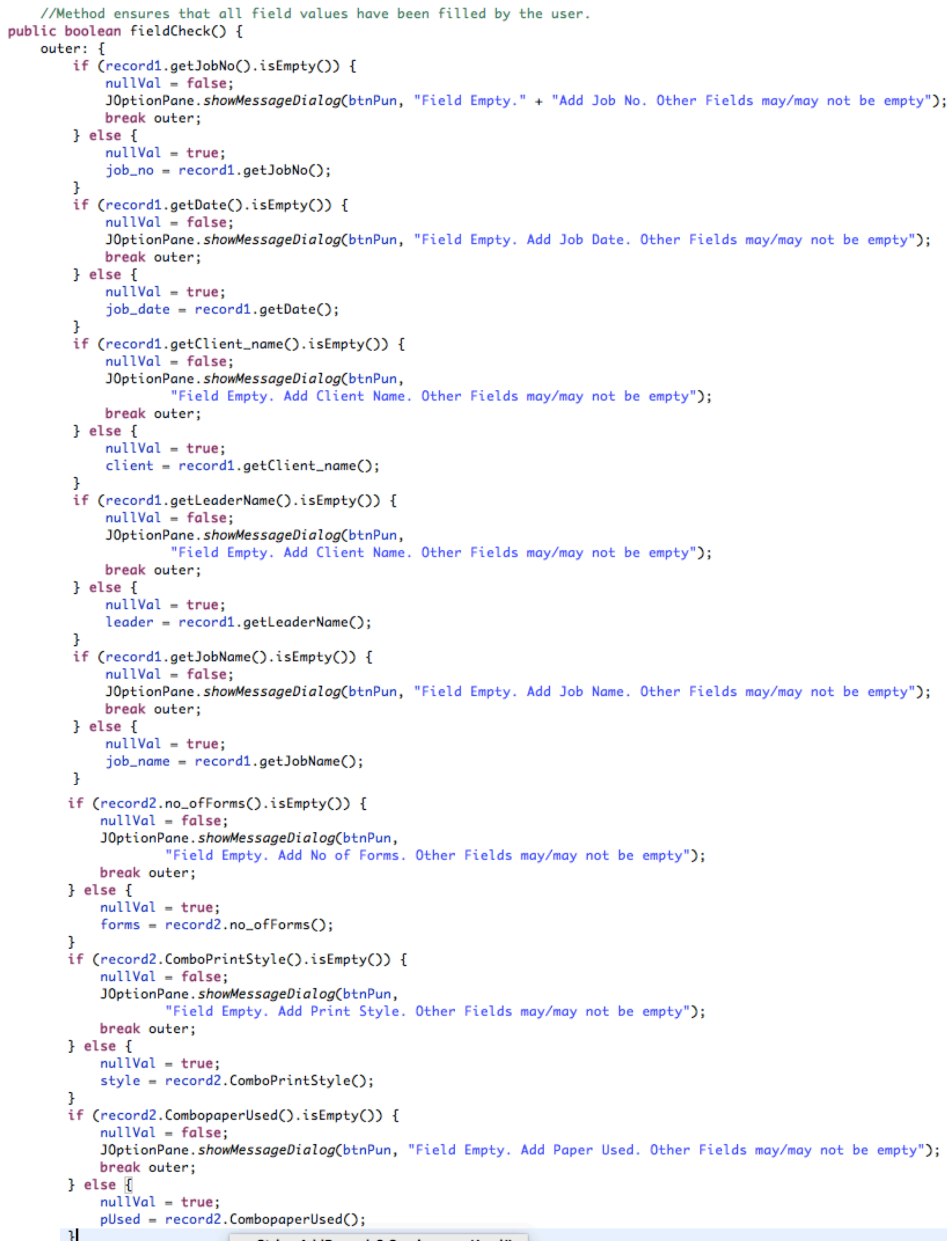
```
public void punchMethod(Connection conn, PreparedStatement state)
{
    String job_no = record1.getJobNo();
    String job_date = record1.getDate();
    String client = record1.getClient_name();
    String leader=record1.getLeaderName();
    String job_name = record1.getJobName();
    String forms = record2.no_ofForms();
    String style = record2.ComboPrintStyle();
    String pUsed = record2.CombopaperUsed();
    String col = record2.TxtCol();
    String printqt=record2.TxtPrintQt();
    int printqtity=Integer.parseInt(printqt);
    String book = record2.TxtBookSize();
    String pages = record2.TxtNo_of_pages();
    String pl = record2.PlateOld();
    String plate=record2.getPlate();
    String pPrint=record2.TxtPaper_Print();
    String paperReq = record2.TxtPaperReq();
    String paperSup = record2.TxtPapersupp();
    String sheets = record2.Psheets();
    String finish = record2.TxtfinishQt();
    String Ddate = record2.Txtdeliverydate();
    String sta = record2.ComboStatus();
    String mach = record2.ComboMachine();
    area=record3.getA1();
    are=record3.getA2();
    finAr=area+" "+are;
```



```
            +"'" +"," +"'" +job_name +"'" +"," +"'" +forms +"'" +","+"'" +style +"'" + ","
            + "'" + pUsed + "'" + "," + "'" + col + "'" + ","  + printqtity + ","
            + "'" + plate + "'" + "," + "'" + pPrint + "'" + "," + "'" + book + "'" + "," + "'" + pages + "'" + ","
            + "'" + pl + "'" + "," + "'" + paperReq + "'" + ","  +"'" + paperSup + "'" + ","
            +"'" + sheets + "'" + "," +"'" + finish + "'" + "," +"'" + Ddate + "'" + ","
            +"'" + sta + "'" + "," +"'" + mach + "'"+ "," +"'" + finAr + "'" + "," +"'" + leader + "'" +")";

    Statement state1;
    try {
        state1 = conn.createStatement();
        state1.executeUpdate(qu1);
        JOptionPane.showMessageDialog(btnPun, "RECORD ADDED SUCCESFULLY");
    } catch (SQLException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }
}
```

Figure 1G: Code for punchMethod() in AddRecord.java

The above code is of the *punchMethod()* showing the SQL query and the instance variables created in figure 1F that were used to get the data from respected classes.



```
    //Method ensures that all field values have been filled by the user.
public boolean fieldCheck() {
    outer: {
        if (record1.getJobNo().isEmpty()) {
            nullVal = false;
            JOptionPane.showMessageDialog(btnPun, "Field Empty." + "Add Job No. Other Fields may/may not be empty");
            break outer;
        } else {
            nullVal = true;
            job_no = record1.getJobNo();
        }
        if (record1.getDate().isEmpty()) {
            nullVal = false;
            JOptionPane.showMessageDialog(btnPun, "Field Empty. Add Job Date. Other Fields may/may not be empty");
            break outer;
        } else {
            nullVal = true;
            job_date = record1.getDate();
        }
        if (record1.getClient_name().isEmpty()) {
            nullVal = false;
            JOptionPane.showMessageDialog(btnPun,
                    "Field Empty. Add Client Name. Other Fields may/may not be empty");
            break outer;
        } else {
            nullVal = true;
            client = record1.getClient_name();
        }
        if (record1.getLeaderName().isEmpty()) {
            nullVal = false;
            JOptionPane.showMessageDialog(btnPun,
                    "Field Empty. Add Client Name. Other Fields may/may not be empty");
            break outer;
        } else {
            nullVal = true;
            leader = record1.getLeaderName();
        }
        if (record1.getJobName().isEmpty()) {
            nullVal = false;
            JOptionPane.showMessageDialog(btnPun, "Field Empty. Add Job Name. Other Fields may/may not be empty");
            break outer;
        } else {
            nullVal = true;
            job_name = record1.getJobName();
        }
       if (record2.no_ofForms().isEmpty()) {
           nullVal = false;
           JOptionPane.showMessageDialog(btnPun,
                   "Field Empty. Add No of Forms. Other Fields may/may not be empty");
           break outer;
       } else {
           nullVal = true;
           forms = record2.no_ofForms();
       }
       if (record2.ComboPrintStyle().isEmpty()) {
           nullVal = false;
           JOptionPane.showMessageDialog(btnPun,
                   "Field Empty. Add Print Style. Other Fields may/may not be empty");
           break outer;
       } else {
           nullVal = true;
           style = record2.ComboPrintStyle();
       }
       if (record2.CombopaperUsed().isEmpty()) {
           nullVal = false;
           JOptionPane.showMessageDialog(btnPun, "Field Empty. Add Paper Used. Other Fields may/may not be empty");
           break outer;
       } else {
           nullVal = true;
           pUsed = record2.CombopaperUsed();
       }
```

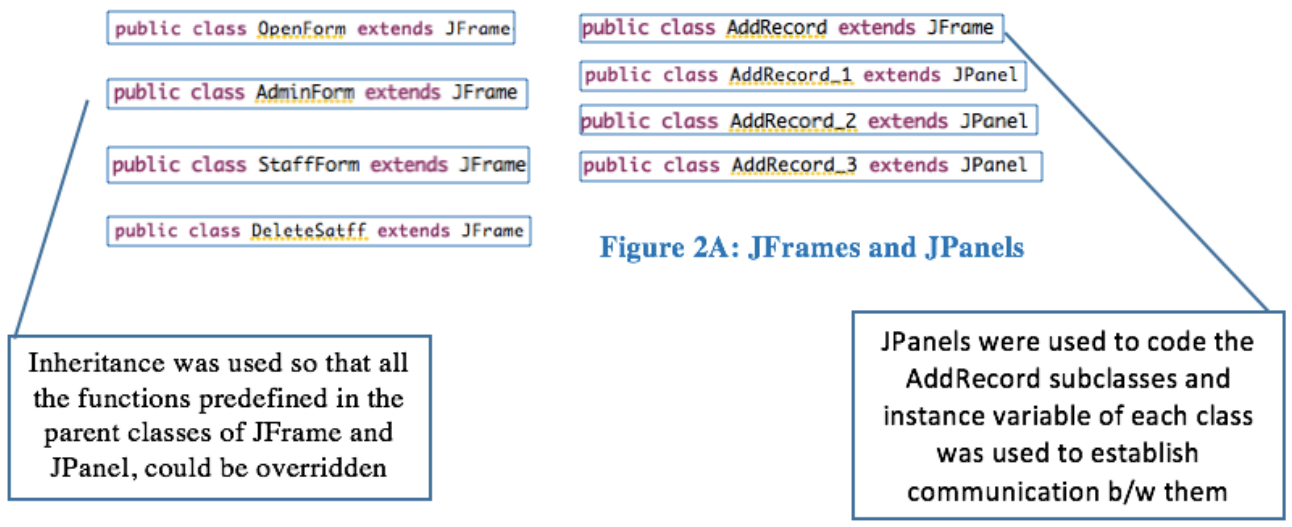
```
        if (record2.Psheets().isEmpty()) {
            nullVal = false;
            JOptionPane.showMessageDialog(btnPun,
                    "Field Empty. Add No of Sheets. Other Fields may/may not be empty");
            break outer;
        } else {
            nullVal = true;
            sheets = record2.Psheets();
        }
        if (record2.ComboStatus().isEmpty()) {
            nullVal = false;
            JOptionPane.showMessageDialog(btnPun, "Field Empty. Add Status. Other Fields may/may not be empty");
            break outer;
        } else {
            nullVal = true;
            sta = record2.ComboStatus();
        }
        if (record2.ComboMachine().isEmpty()) {
            nullVal = false;
            JOptionPane.showMessageDialog(btnPun, "Field Empty. Add Machine. Other Fields may/may not be empty");
            break outer;
        } else {
            nullVal = true;
            mach = record2.ComboMachine();
        }
        if (record2.Txtdeliverydate().isEmpty()) {
            nullVal = false;
            JOptionPane.showMessageDialog(btnPun,
                    "Field Empty. Add Delivery Date. Other Fields may/may not be empty");
            break outer;
        } else {
            nullVal = true;
            Ddate = record2.Txtdeliverydate();
        }
        if (record2.TxtfinishQt().isEmpty()) {
            nullVal = false;
            JOptionPane.showMessageDialog(btnPun,
                    "Field Empty. Add Finish Quantity. Other Fields may/may not be empty");
            break outer;
        } else {
            nullVal = true;
            finish = record2.TxtfinishQt();
        }
    }
    return nullVal;
}
```

Figure 1H: Code for fieldCheck()in AddRecord.java

The above code shows *fieldCheck*() method, which uses the instance-variables created in figure 1F to get the data input by the user and check if the value of it is null/empty. This method uses label (*outer loop:*) to break out from the conditional-statements if any value is left empty and returns false.

GUI Features and Swing

A Combination of JFrames and JPanels were used.

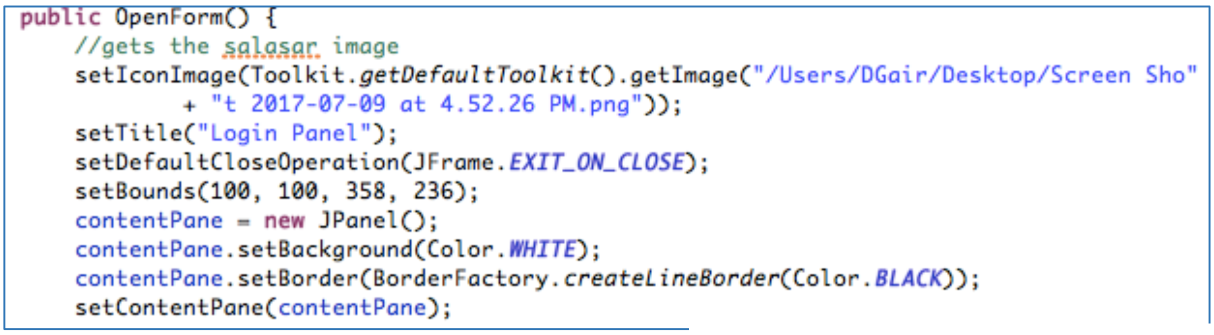


```
public class OpenForm extends JFrame
public class AdminForm extends JFrame
public class StaffForm extends JFrame
public class DeleteSatff extends JFrame

public class AddRecord extends JFrame
public class AddRecord_1 extends JPanel
public class AddRecord_2 extends JPanel
public class AddRecord_3 extends JPanel
```

Inheritance was used so that all the functions predefined in the parent classes of JFrame and JPanel, could be overridden

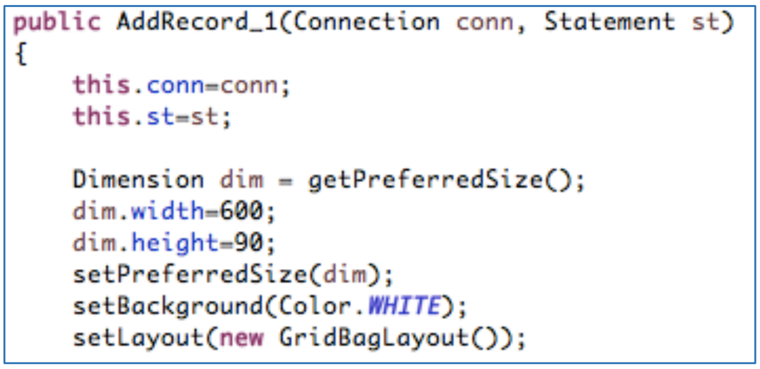
JPanels were used to code the AddRecord subclasses and instance variable of each class was used to establish communication b/w them

Figure 2A: JFrames and JPanels



```
public OpenForm() {
    //gets the salasar image
    setIconImage(Toolkit.getDefaultToolkit().getImage("/Users/DGair/Desktop/Screen Sho"
            + "t 2017-07-09 at 4.52.26 PM.png"));
    setTitle("Login Panel");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setBounds(100, 100, 358, 236);
    contentPane = new JPanel();
    contentPane.setBackground(Color.WHITE);
    contentPane.setBorder(BorderFactory.createLineBorder(Color.BLACK));
    setContentPane(contentPane);
```

Figure 2B: Use of predefined JFrame Methods



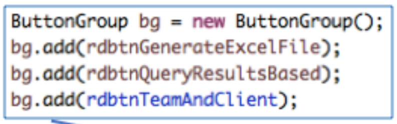
```
public AddRecord_1(Connection conn, Statement st)
{
    this.conn=conn;
    this.st=st;

    Dimension dim = getPreferredSize();
    dim.width=600;
    dim.height=90;
    setPreferredSize(dim);
    setBackground(Color.WHITE);
    setLayout(new GridBagLayout());
```

Figure 2C: Use of Predefined JPanel Methods

The above code snapshots show the overridden-methods used from the parent *JFrame* class and *JPanel* class. *OpenForm*.java extends *JFrame* and uses methods like *setTitle*(), *setBounds*() etc., while *AddRecord_1.java* extends *JPanel* and uses methods like *setLayout*(), *setBackground*() etc. (Evidence of Inheritance)

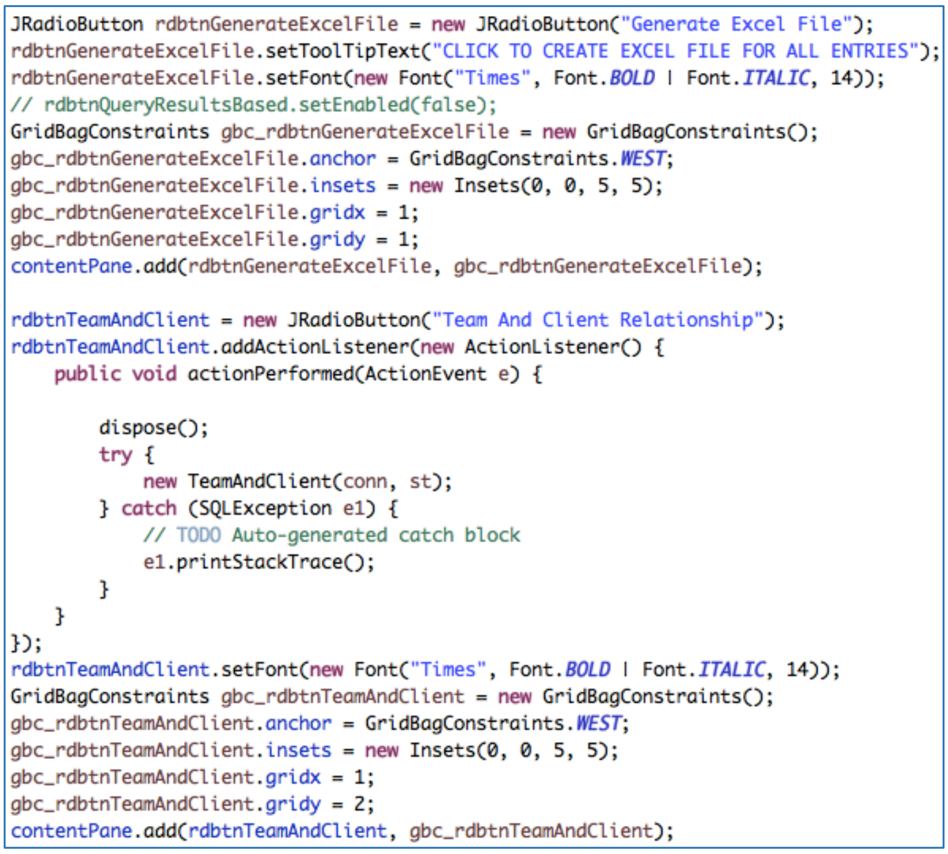
The method setDefaultCloseOperation() is used to terminate the program. ToolKit, an AWT component was used to set the image in the Login form (Figure 2B) and the Admin form.

GUI Component: RadioButtons

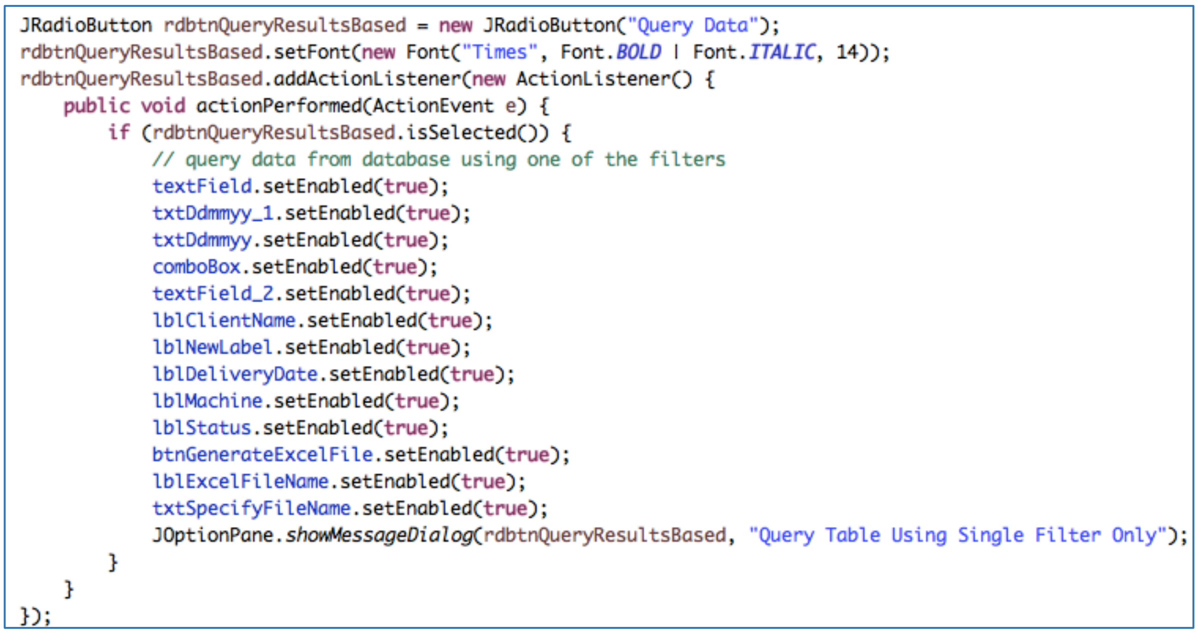


```
ButtonGroup bg = new ButtonGroup();
bg.add(rdbtnGenerateExcelFile);
bg.add(rdbtnQueryResultsBased);
bg.add(rdbtnTeamAndClient);
```

JRadioButtons were used to when the input was specific or predefined, so that the user chooses a single option out of the list of options. Adding the Radio Buttons to ButtonGroup allowed this



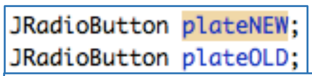
```java
JRadioButton rdbtnGenerateExcelFile = new JRadioButton("Generate Excel File");
rdbtnGenerateExcelFile.setToolTipText("CLICK TO CREATE EXCEL FILE FOR ALL ENTRIES");
rdbtnGenerateExcelFile.setFont(new Font("Times", Font.BOLD | Font.ITALIC, 14));
// rdbtnQueryResultsBased.setEnabled(false);
GridBagConstraints gbc_rdbtnGenerateExcelFile = new GridBagConstraints();
gbc_rdbtnGenerateExcelFile.anchor = GridBagConstraints.WEST;
gbc_rdbtnGenerateExcelFile.insets = new Insets(0, 0, 5, 5);
gbc_rdbtnGenerateExcelFile.gridx = 1;
gbc_rdbtnGenerateExcelFile.gridy = 1;
contentPane.add(rdbtnGenerateExcelFile, gbc_rdbtnGenerateExcelFile);

rdbtnTeamAndClient = new JRadioButton("Team And Client Relationship");
rdbtnTeamAndClient.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {

        dispose();
        try {
            new TeamAndClient(conn, st);
        } catch (SQLException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }
    }
});
rdbtnTeamAndClient.setFont(new Font("Times", Font.BOLD | Font.ITALIC, 14));
GridBagConstraints gbc_rdbtnTeamAndClient = new GridBagConstraints();
gbc_rdbtnTeamAndClient.anchor = GridBagConstraints.WEST;
gbc_rdbtnTeamAndClient.insets = new Insets(0, 0, 5, 5);
gbc_rdbtnTeamAndClient.gridx = 1;
gbc_rdbtnTeamAndClient.gridy = 2;
contentPane.add(rdbtnTeamAndClient, gbc_rdbtnTeamAndClient);
```



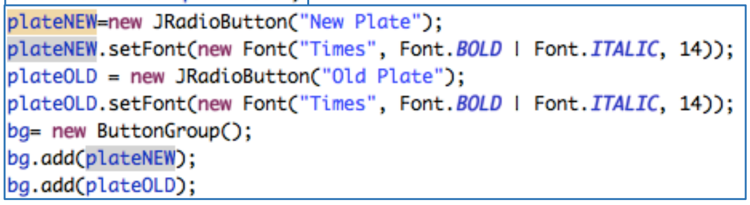
```java
JRadioButton rdbtnQueryResultsBased = new JRadioButton("Query Data");
rdbtnQueryResultsBased.setFont(new Font("Times", Font.BOLD | Font.ITALIC, 14));
rdbtnQueryResultsBased.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (rdbtnQueryResultsBased.isSelected()) {
            // query data from database using one of the filters
            textField.setEnabled(true);
            txtDdmmyy_1.setEnabled(true);
            txtDdmmyy.setEnabled(true);
            comboBox.setEnabled(true);
            textField_2.setEnabled(true);
            lblClientName.setEnabled(true);
            lblNewLabel.setEnabled(true);
            lblDeliveryDate.setEnabled(true);
            lblMachine.setEnabled(true);
            lblStatus.setEnabled(true);
            btnGenerateExcelFile.setEnabled(true);
            lblExcelFileName.setEnabled(true);
            txtSpecifyFileName.setEnabled(true);
            JOptionPane.showMessageDialog(rdbtnQueryResultsBased, "Query Table Using Single Filter Only");
        }
    }
});
```

Figure 2E: Use of JRadioButton in AdminForm.java

The above is the implementation of radio-buttons in the Admin-Form. All fields in the form are enabled when the Radio Button – 'Query Data' is clicked and a message is shown using Message-Dialog.



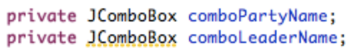
```java
JRadioButton plateNEW;
JRadioButton plateOLD;
```



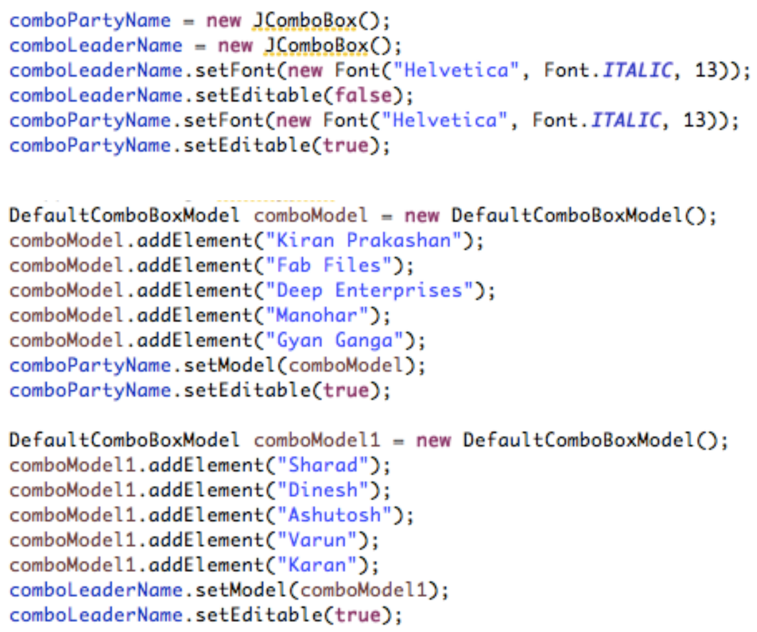
```java
plateNEW=new JRadioButton("New Plate");
plateNEW.setFont(new Font("Times", Font.BOLD | Font.ITALIC, 14));
plateOLD = new JRadioButton("Old Plate");
plateOLD.setFont(new Font("Times", Font.BOLD | Font.ITALIC, 14));
bg= new ButtonGroup();
bg.add(plateNEW);
bg.add(plateOLD);
```

Figure 2D: Use of Button Group

GUI Component: Drop Down List



```java
private JComboBox comboPartyName;
private JComboBox comboLeaderName;
```

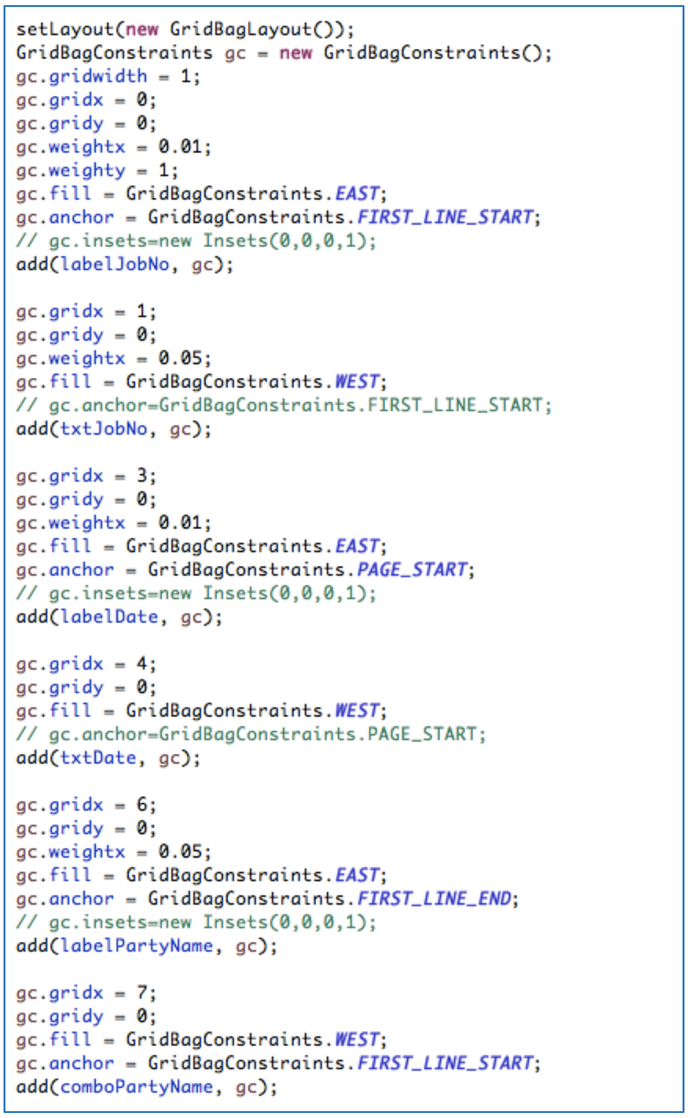


```java
comboPartyName = new JComboBox();
comboLeaderName = new JComboBox();
comboLeaderName.setFont(new Font("Helvetica", Font.ITALIC, 13));
comboLeaderName.setEditable(false);
comboPartyName.setFont(new Font("Helvetica", Font.ITALIC, 13));
comboPartyName.setEditable(true);


DefaultComboBoxModel comboModel = new DefaultComboBoxModel();
comboModel.addElement("Kiran Prakashan");
comboModel.addElement("Fab Files");
comboModel.addElement("Deep Enterprises");
comboModel.addElement("Manohar");
comboModel.addElement("Gyan Ganga");
comboPartyName.setModel(comboModel);
comboPartyName.setEditable(true);

DefaultComboBoxModel comboModel1 = new DefaultComboBoxModel();
comboModel1.addElement("Sharad");
comboModel1.addElement("Dinesh");
comboModel1.addElement("Ashutosh");
comboModel1.addElement("Varun");
comboModel1.addElement("Karan");
comboLeaderName.setModel(comboModel1);
comboLeaderName.setEditable(true);
```

Figure 2E: Use of JComboBox and DefaultListModel in AddRecord_2.java

The following code is the implementation of JComboBox and Default-List-Model for adding a dynamic-selection list, so that the user can add new, or choose existing data from the list.
.

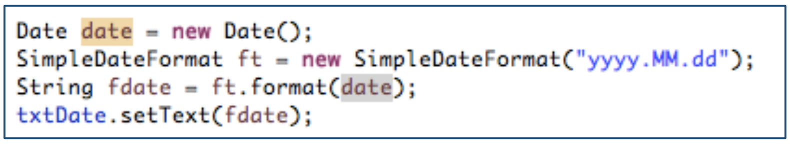
GUI Component: GridBagLayout.



```
setLayout(new GridBagLayout());
GridBagConstraints gc = new GridBagConstraints();
gc.gridwidth = 1;
gc.gridx = 0;
gc.gridy = 0;
gc.weightx = 0.01;
gc.weighty = 1;
gc.fill = GridBagConstraints.EAST;
gc.anchor = GridBagConstraints.FIRST_LINE_START;
// gc.insets=new Insets(0,0,0,1);
add(labelJobNo, gc);

gc.gridx = 1;
gc.gridy = 0;
gc.weightx = 0.05;
gc.fill = GridBagConstraints.WEST;
// gc.anchor=GridBagConstraints.FIRST_LINE_START;
add(txtJobNo, gc);

gc.gridx = 3;
gc.gridy = 0;
gc.weightx = 0.01;
gc.fill = GridBagConstraints.EAST;
gc.anchor = GridBagConstraints.PAGE_START;
// gc.insets=new Insets(0,0,0,1);
add(labelDate, gc);

gc.gridx = 4;
gc.gridy = 0;
gc.fill = GridBagConstraints.WEST;
// gc.anchor=GridBagConstraints.PAGE_START;
add(txtDate, gc);

gc.gridx = 6;
gc.gridy = 0;
gc.weightx = 0.05;
gc.fill = GridBagConstraints.EAST;
gc.anchor = GridBagConstraints.FIRST_LINE_END;
// gc.insets=new Insets(0,0,0,1);
add(labelPartyName, gc);

gc.gridx = 7;
gc.gridy = 0;
gc.fill = GridBagConstraints.WEST;
gc.anchor = GridBagConstraints.FIRST_LINE_START;
add(comboPartyName, gc);
```

Figure 2H: Use of GridBagLayout UpdateForm.java

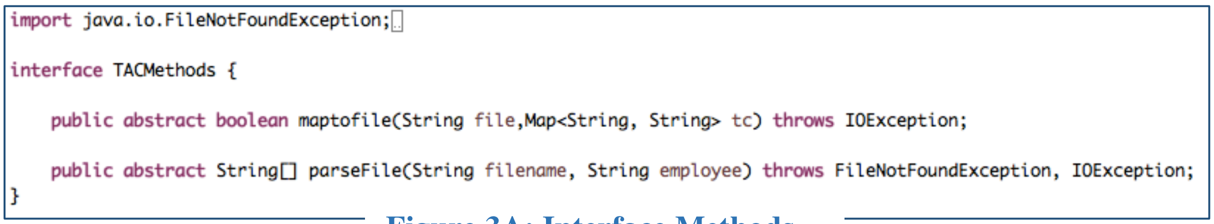
Layout of the forms was handled manually using the grid-bag-layout, because of the ease by which components can be adjusted on the x-y plane



```
Date date = new Date();
SimpleDateFormat ft = new SimpleDateFormat("yyyy.MM.dd");
String fdate = ft.format(date);
txtDate.setText(fdate);
```

Figure 2I: The above code gets the current date and sets it in the StaffForm before the user opens it.

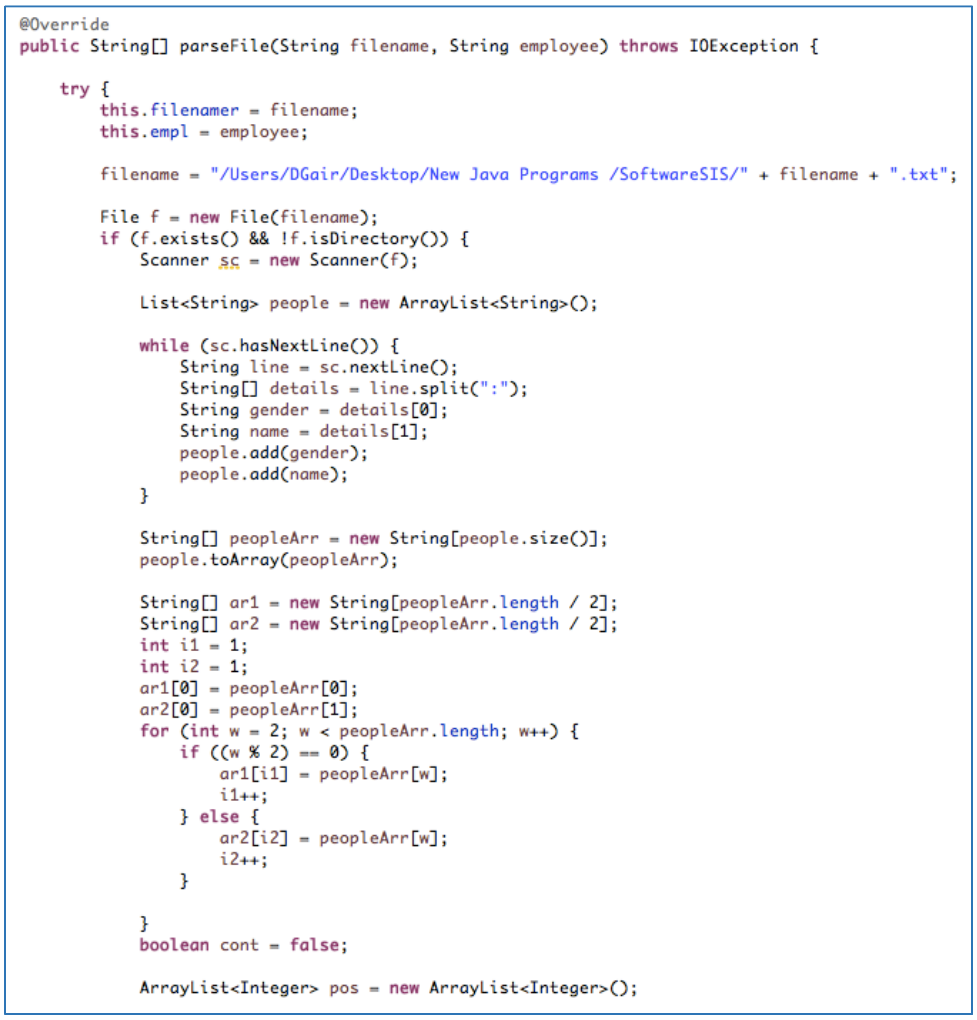
Interfaces



```
import java.io.FileNotFoundException;

interface TACMethods {

    public abstract boolean maptofile(String file,Map<String, String> tc) throws IOException;

    public abstract String[] parseFile(String filename, String employee) throws FileNotFoundException, IOException;
}
```

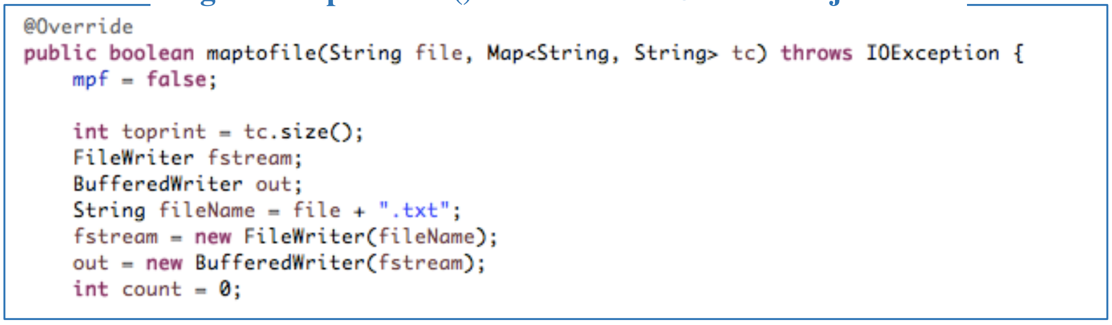
Figure 3A: Interface Methods

TACMethods, the interface used to achieve abstraction, so that the two methods defined in it can be used by different classes and they can define their own implementation of the methods.

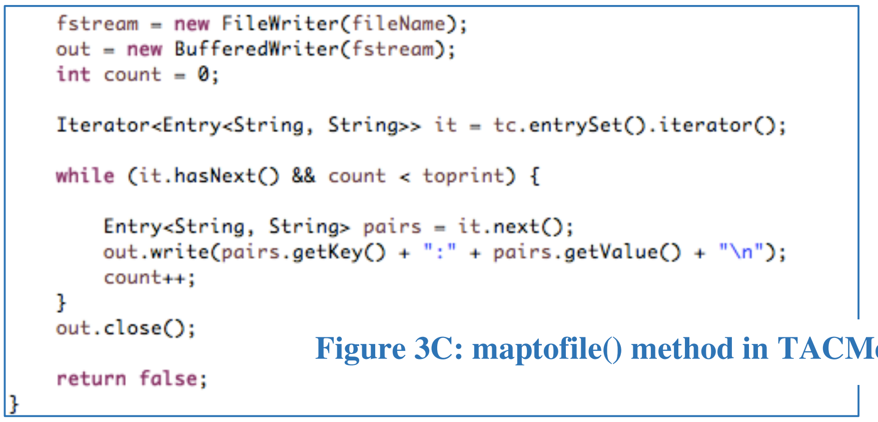


```
@Override
public String[] parseFile(String filename, String employee) throws IOException {

    try {
        this.filenamer = filename;
        this.empl = employee;

        filename = "/Users/DGair/Desktop/New Java Programs /SoftwareSIS/" + filename + ".txt";

        File f = new File(filename);
        if (f.exists() && !f.isDirectory()) {
            Scanner sc = new Scanner(f);

            List<String> people = new ArrayList<String>();

            while (sc.hasNextLine()) {
                String line = sc.nextLine();
                String[] details = line.split(":");
                String gender = details[0];
                String name = details[1];
                people.add(gender);
                people.add(name);
            }

            String[] peopleArr = new String[people.size()];
            people.toArray(peopleArr);

            String[] ar1 = new String[peopleArr.length / 2];
            String[] ar2 = new String[peopleArr.length / 2];
            int i1 = 1;
            int i2 = 1;
            ar1[0] = peopleArr[0];
            ar2[0] = peopleArr[1];
            for (int w = 2; w < peopleArr.length; w++) {
                if ((w % 2) == 0) {
                    ar1[i1] = peopleArr[w];
                    i1++;
                } else {
                    ar2[i2] = peopleArr[w];
                    i2++;
                }

            }
            boolean cont = false;

            ArrayList<Integer> pos = new ArrayList<Integer>();
```

Figure 3B: parseFile() method in TACMethodE.java



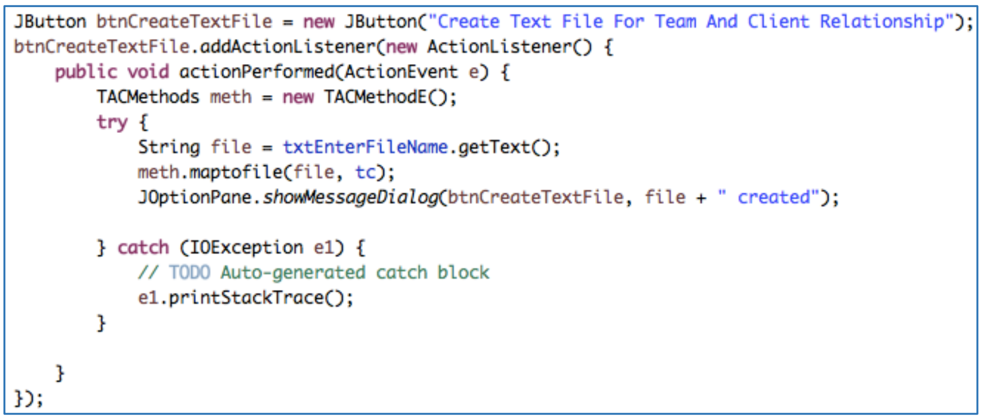
```
@Override
public boolean maptofile(String file, Map<String, String> tc) throws IOException {
    mpf = false;

    int toprint = tc.size();
    FileWriter fstream;
    BufferedWriter out;
    String fileName = file + ".txt";
    fstream = new FileWriter(fileName);
    out = new BufferedWriter(fstream);
    int count = 0;
```



```
    fstream = new FileWriter(fileName);
    out = new BufferedWriter(fstream);
    int count = 0;

    Iterator<Entry<String, String>> it = tc.entrySet().iterator();

    while (it.hasNext() && count < toprint) {

        Entry<String, String> pairs = it.next();
        out.write(pairs.getKey() + ":" + pairs.getValue() + "\n");
        count++;
    }
    out.close();

    return false;
}
```

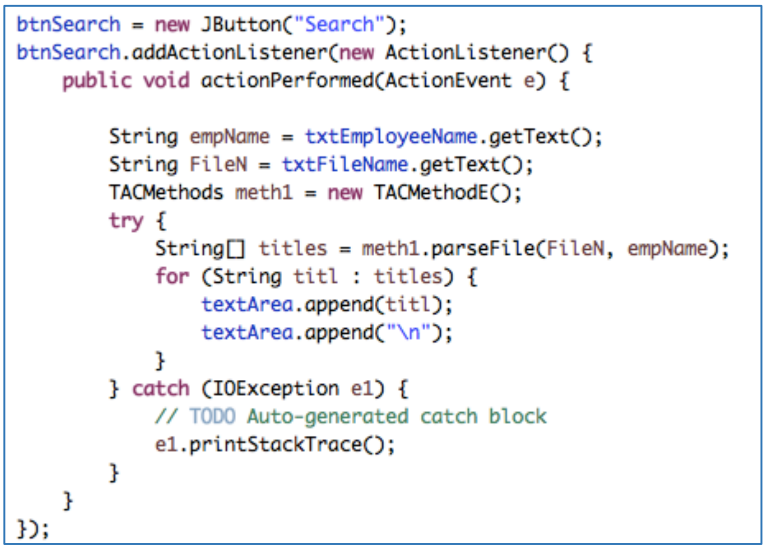
Figure 3C: maptofile() method in TACMethodE.java

The above code is the definition of the two methods in the TACMethodsE.java class that were defined in the Interface.



```
JButton btnCreateTextFile = new JButton("Create Text File For Team And Client Relationship");
btnCreateTextFile.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        TACMethods meth = new TACMethodE();
        try {
            String file = txtEnterFileName.getText();
            meth.maptofile(file, tc);
            JOptionPane.showMessageDialog(btnCreateTextFile, file + " created");

        } catch (IOException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }

    }
});
```

Figure 3D: Instance variable ‘meth’ used to call maptofile(), created in the interface, defined in TACMethodE.java from TACMethod.java



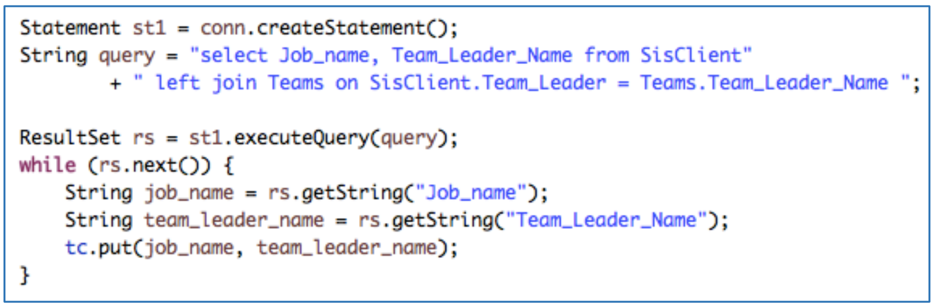
```
btnSearch = new JButton("Search");
btnSearch.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {

        String empName = txtEmployeeName.getText();
        String FileN = txtFileName.getText();
        TACMethods meth1 = new TACMethodE();
        try {
            String[] titles = meth1.parseFile(FileN, empName);
            for (String titl : titles) {
                textArea.append(titl);
                textArea.append("\n");
            }
        } catch (IOException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }
    }
});
```

Figure 3E: Instance variable ‘meth1’ used to call parseFile(), created in the interface, defined in TACMethodE.java from TACMethod.java

The following code shows how dynamic binding was used to communicate between the interface and the class that holds the definition of the two methods defined in the interface. ‘meth’ and ‘meth1’ are used to call the two methods. (Evidence of Polymorphism)

Database Complexity and Features:

Establishing relationship between Teams Table and SisClient Table

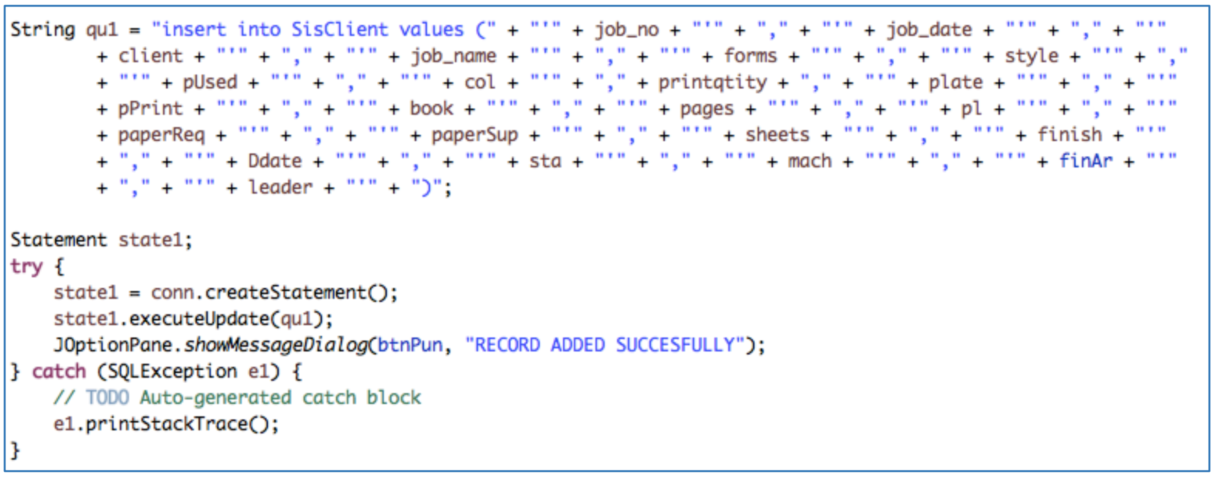


```
Statement st1 = conn.createStatement();
String query = "select Job_name, Team_Leader_Name from SisClient"
        + " left join Teams on SisClient.Team_Leader = Teams.Team_Leader_Name ";

ResultSet rs = st1.executeQuery(query);
while (rs.next()) {
    String job_name = rs.getString("Job_name");
    String team_leader_name = rs.getString("Team_Leader_Name");
    tc.put(job_name, team_leader_name);
}
```

Figure 4A: SQL query to establish Relationship

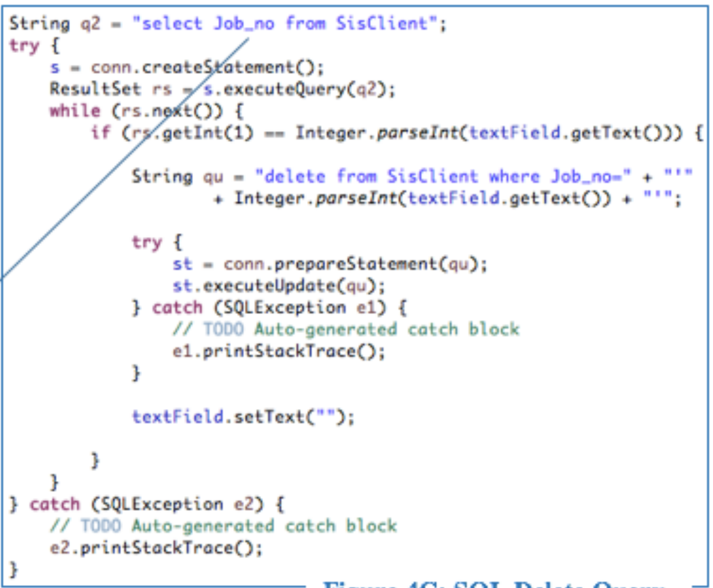
The above code shows established a left-join using the fields Team_Leader from SisClient and Team_Leader_Name from Teams-and-Client-Table.

Inserting/Deleting/Updating/Searching using SQL



```
String qu1 = "insert into SisClient values (" + "'" + job_no + "'" + "," + "'" + job_date + "'" + "," + "'"
        + client + "'" + "," + "'" + job_name + "'" + "," + "'" + forms + "'" + "," + "'" + style + "'" + ","
        + "'" + pUsed + "'" + "," + "'" + col + "'" + "," + printqtity + "," + "'" + plate + "'" + "," + "'"
        + pPrint + "'" + "," + "'" + book + "'" + "," + "'" + pages + "'" + "," + "'" + pl + "'" + "," + "'"
        + paperReq + "'" + "," + "'" + paperSup + "'" + "," + "'" + sheets + "'" + "," + "'" + finish + "'"
        + "," + "'" + Ddate + "'" + "," + "'" + sta + "'" + "," + "'" + mach + "'" + "," + "'" + finAr + "'"
        + "," + "'" + leader + "'" + ")";

Statement state1;
try {
    state1 = conn.createStatement();
    state1.executeUpdate(qu1);
    JOptionPane.showMessageDialog(btnPun, "RECORD ADDED SUCCESFULLY");
} catch (SQLException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
}
```

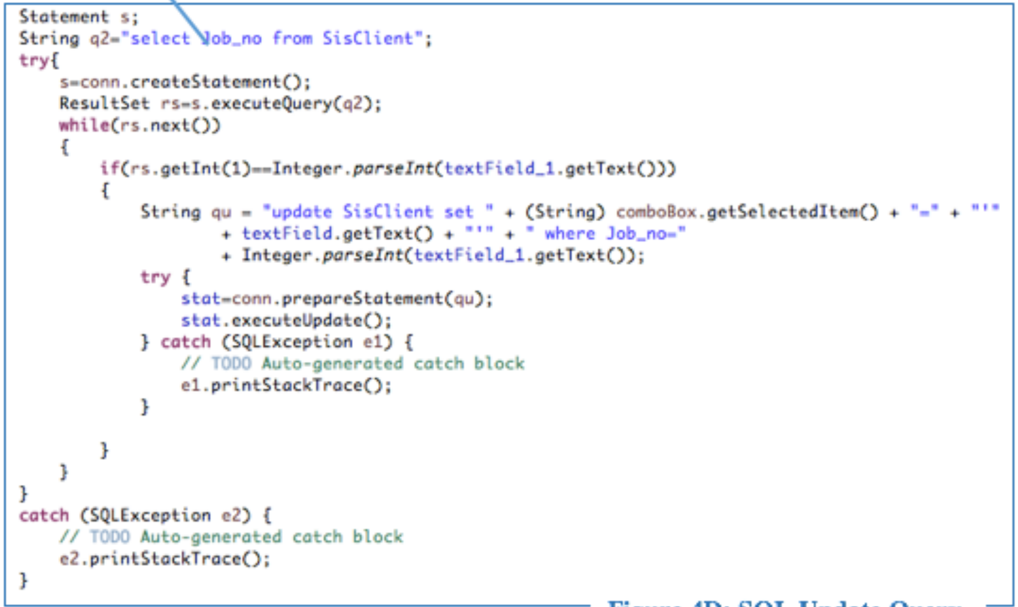
Figure 4B: SQL Insert Query



```
String q2 = "select Job_no from SisClient";
try {
    s = conn.createStatement();
    ResultSet rs = s.executeQuery(q2);
    while (rs.next()) {
        if (rs.getInt(1) == Integer.parseInt(textField.getText())) {

            String qu = "delete from SisClient where Job_no=" + "'"
                    + Integer.parseInt(textField.getText()) + "'";

            try {
                st = conn.prepareStatement(qu);
                st.executeUpdate(qu);
            } catch (SQLException e1) {
                // TODO Auto-generated catch block
                e1.printStackTrace();
            }

            textField.setText("");

        }
    }
} catch (SQLException e2) {
    // TODO Auto-generated catch block
    e2.printStackTrace();
}
```

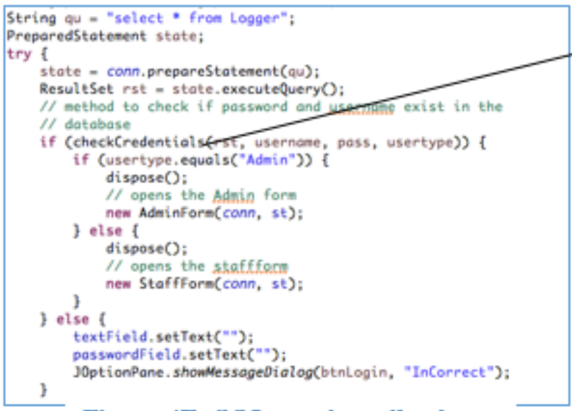
Figure 4C: SQL Delete Query

To Update Or Delete any item from the table, the user needs to input the job_no first, as it is the primary key of the table and is used to uniquely identify the record that user wants to delete. Therefore, both queries search for the job_no first, and if found, they then run the update/delete query.



```
Statement s;
String q2="select Job_no from SisClient";
try{
    s=conn.createStatement();
    ResultSet rs=s.executeQuery(q2);
    while(rs.next())
    {
        if(rs.getInt(1)==Integer.parseInt(textField_1.getText()))
        {
            String qu = "update SisClient set " + (String) comboBox.getSelectedItem() + "=" + "'"
                    + textField.getText() + "'" + " where Job_no="
                    + Integer.parseInt(textField_1.getText());
            try {
                stat=conn.prepareStatement(qu);
                stat.executeUpdate();
            } catch (SQLException e1) {
                // TODO Auto-generated catch block
                e1.printStackTrace();
            }

        }
    }
}
catch (SQLException e2) {
    // TODO Auto-generated catch block
    e2.printStackTrace();
}
```

Figure 4D: SQL Update Query

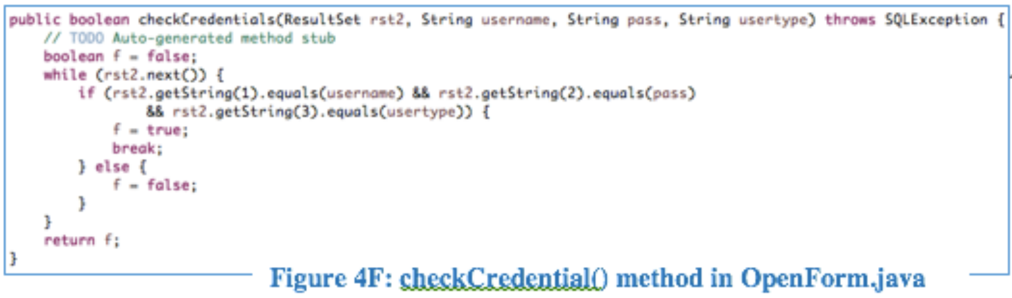


```
String qu = "select * from Logger";
PreparedStatement state;
try {
    state = conn.prepareStatement(qu);
    ResultSet rst = state.executeQuery();
    // method to check if password and username exist in the
    // database
    if (checkCredentials(rst, username, pass, usertype)) {
        if (usertype.equals("Admin")) {
            dispose();
            // opens the Admin form
            new AdminForm(conn, st);
        } else {
            dispose();
            // opens the staffform
            new StaffForm(conn, st);
        }
    } else {
        textField.setText("");
        passwordField.setText("");
        JOptionPane.showMessageDialog(btnLogin, "InCorrect");
    }
```

Figure 4E: SQL to select all values

Figure 4E shows the query that checks if the login details input by the user match the ones stored in the Database. This is done using *checkCredentials() method*

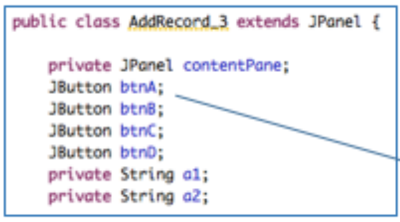
Figure 4F shows this method which compares the two values, the one that is input by the user and the one stored in the database, and returns true if the values exist in the database.



```
public boolean checkCredentials(ResultSet rst2, String username, String pass, String usertype) throws SQLException {
    // TODO Auto-generated method stub
    boolean f = false;
    while (rst2.next()) {
        if (rst2.getString(1).equals(username) && rst2.getString(2).equals(pass)
                && rst2.getString(3).equals(usertype)) {
            f = true;
            break;
        } else {
            f = false;
        }
    }
    return f;
}
```

Figure 4F: checkCredential() method in OpenForm.java

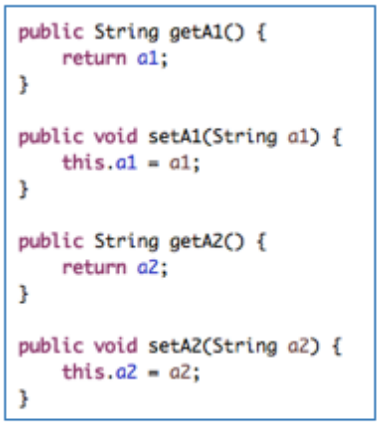
Encapsulation



```
public class AddRecord_3 extends JPanel {

    private JPanel contentPane;
    JButton btnA;
    JButton btnB;
    JButton btnC;
    JButton btnD;
    private String a1;
    private String a2;
```

Figure 5A: Use of private in AddRecord_3.java

All program variables have similarly been initialized as private The only way to access the fields is using getter and setter methods, so any other class cannot call the fields and cannot change the data that resides in them.



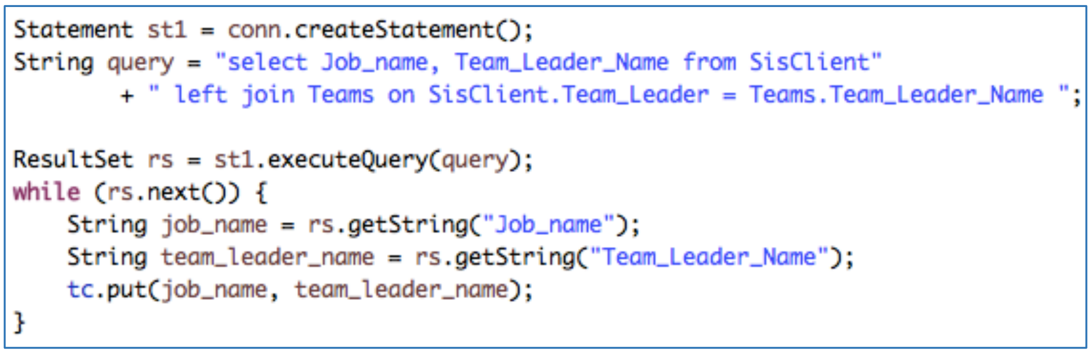
```
public String getA1() {
    return a1;
}

public void setA1(String a1) {
    this.a1 = a1;
}

public String getA2() {
    return a2;
}

public void setA2(String a2) {
    this.a2 = a2;
}
```

This technique particularly helps when classes need to communicate with other classes to fetch data such as AddRecord_3.java and AddRecord.java. Figure 5B shows the getter and setter method used to fetch

Figure 5B: getter and setter methods in OpenForm.java

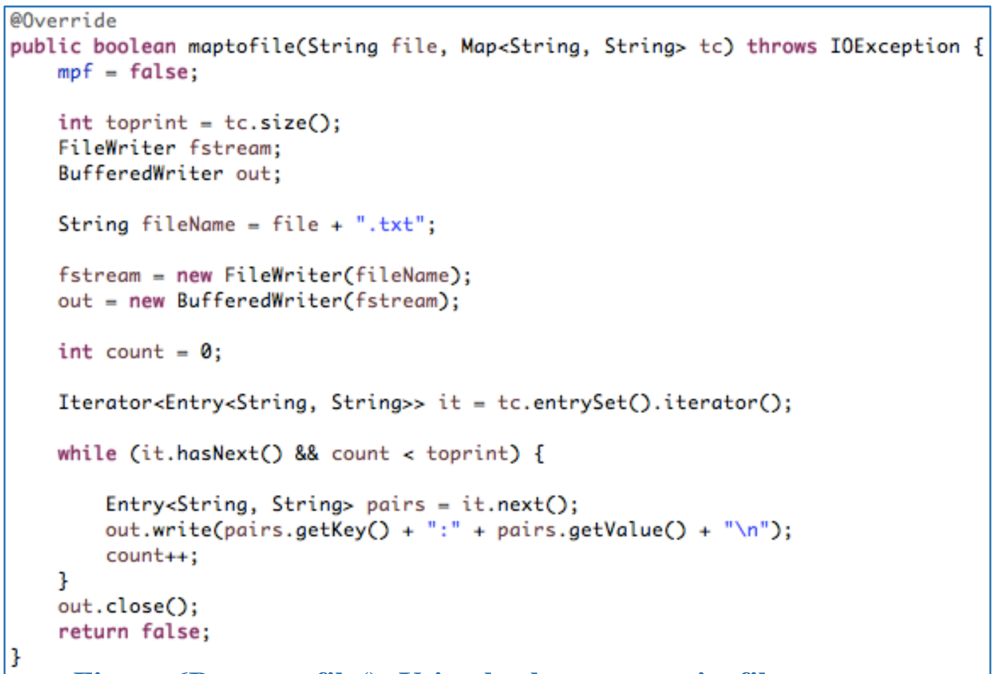
Parsing a Text File and ADS

The abstract data structure of hash maps which stores key value pairs has been used to create a text file.



```
Statement st1 = conn.createStatement();
String query = "select Job_name, Team_Leader_Name from SisClient"
        + " left join Teams on SisClient.Team_Leader = Teams.Team_Leader_Name ";

ResultSet rs = st1.executeQuery(query);
while (rs.next()) {
    String job_name = rs.getString("Job_name");
    String team_leader_name = rs.getString("Team_Leader_Name");
    tc.put(job_name, team_leader_name);
}
```

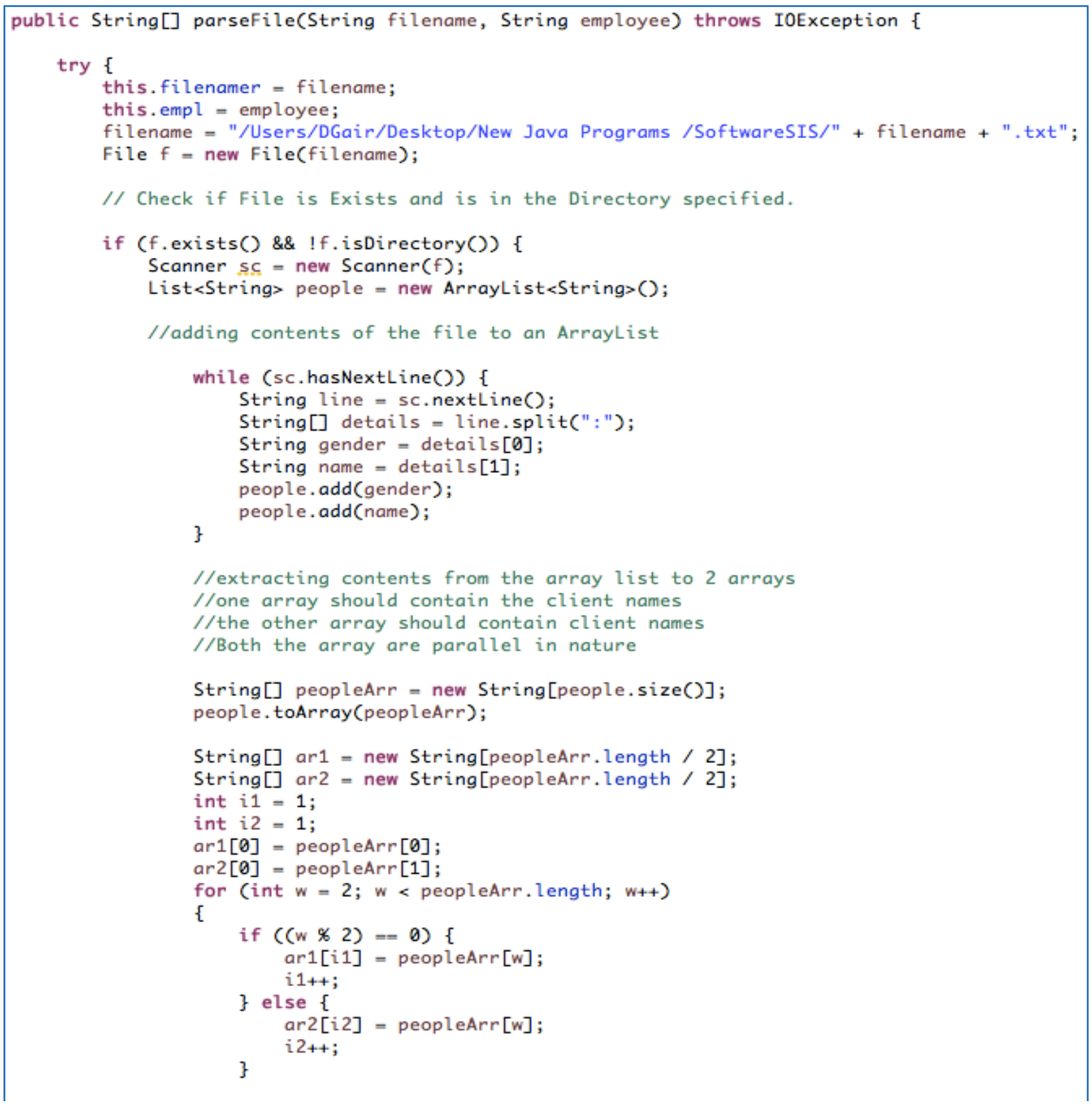
Figure 6A: Inserting Values from a Query to a hash map in AdminRecord.java



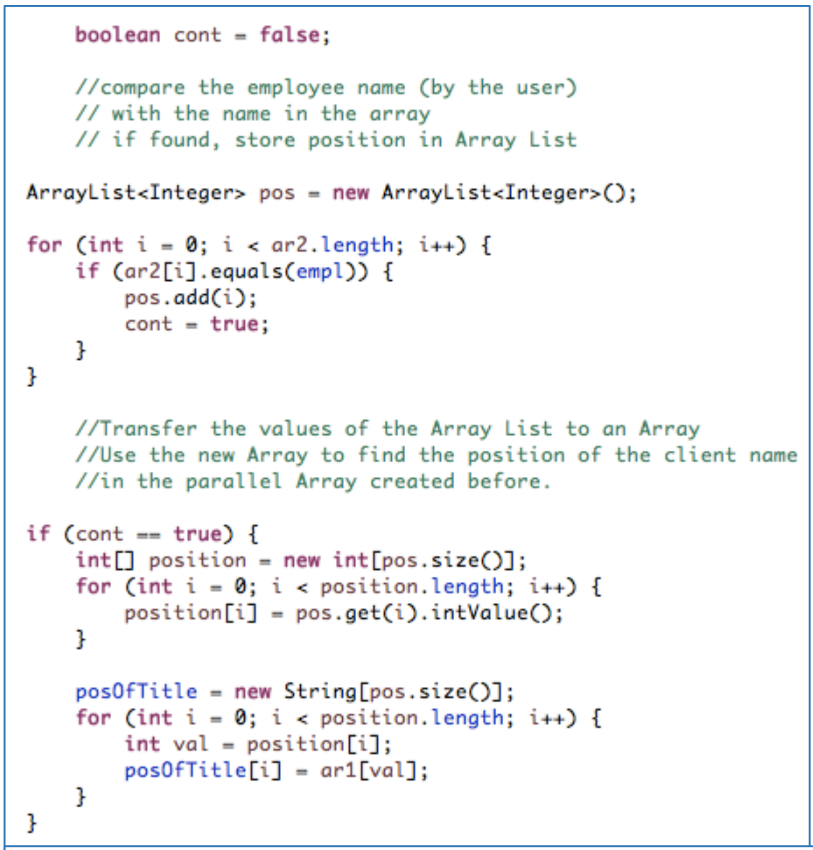
```java
@Override
public boolean maptofile(String file, Map<String, String> tc) throws IOException {
    mpf = false;

    int toprint = tc.size();
    FileWriter fstream;
    BufferedWriter out;

    String fileName = file + ".txt";

    fstream = new FileWriter(fileName);
    out = new BufferedWriter(fstream);

    int count = 0;

    Iterator<Entry<String, String>> it = tc.entrySet().iterator();

    while (it.hasNext() && count < toprint) {

        Entry<String, String> pairs = it.next();
        out.write(pairs.getKey() + ":" + pairs.getValue() + "\n");
        count++;
    }
    out.close();
    return false;
}
```

Figure 6B: maptofile()- Using hash map to write file

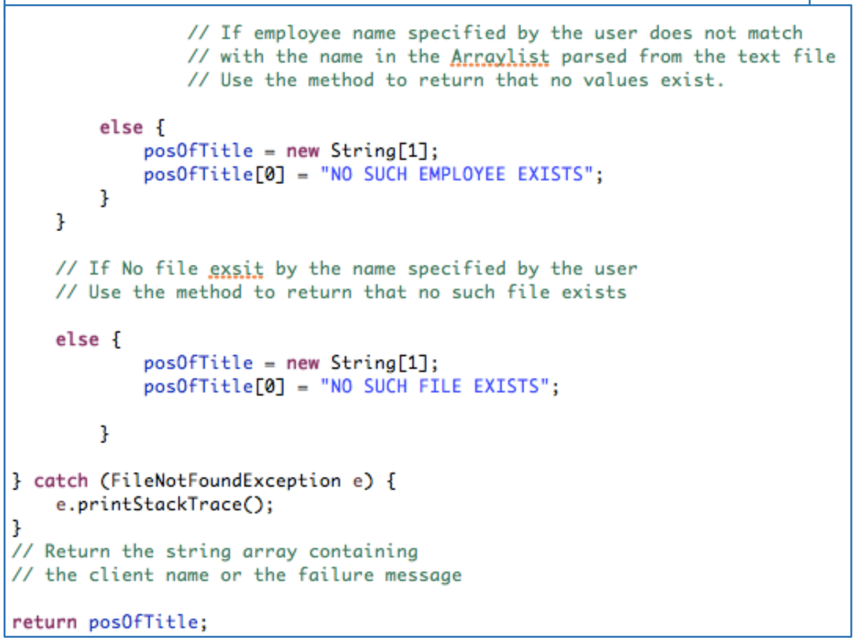
'tc' is the instance variable of the hashmap which is created by the content of the query shown in 6A. This is passed to the method *maptofile*() in the *TACMethodsE* class, which uses this map to write a file using FileWriter and BufferedWriter. The name of this file is specified by the user. The content of the hashmap includes the name of the manager and the name of the client's order (job_name).
The program further allows to search for all the orders (by their job_names) that a manager from the company handles, from a file that was created by the user. This requires parsing a text file and has been achieved using coding shown by 6C.



```java
public String[] parseFile(String filename, String employee) throws IOException {

    try {
        this.filenamer = filename;
        this.empl = employee;
        filename = "/Users/DGair/Desktop/New Java Programs /SoftwareSIS/" + filename + ".txt";
        File f = new File(filename);

        // Check if File is Exists and is in the Directory specified.

        if (f.exists() && !f.isDirectory()) {
            Scanner sc = new Scanner(f);
            List<String> people = new ArrayList<String>();

            //adding contents of the file to an ArrayList

                while (sc.hasNextLine()) {
                    String line = sc.nextLine();
                    String[] details = line.split(":");
                    String gender = details[0];
                    String name = details[1];
                    people.add(gender);
                    people.add(name);
                }

                //extracting contents from the array list to 2 arrays
                //one array should contain the client names
                //the other array should contain client names
                //Both the array are parallel in nature

                String[] peopleArr = new String[people.size()];
                people.toArray(peopleArr);

                String[] ar1 = new String[peopleArr.length / 2];
                String[] ar2 = new String[peopleArr.length / 2];
                int i1 = 1;
                int i2 = 1;
                ar1[0] = peopleArr[0];
                ar2[0] = peopleArr[1];
                for (int w = 2; w < peopleArr.length; w++)
                {
                    if ((w % 2) == 0) {
                        ar1[i1] = peopleArr[w];
                        i1++;
                    } else {
                        ar2[i2] = peopleArr[w];
                        i2++;
                    }
```



```
    boolean cont = false;

    //compare the employee name (by the user)
    // with the name in the array
    // if found, store position in Array List

ArrayList<Integer> pos = new ArrayList<Integer>();

for (int i = 0; i < ar2.length; i++) {
    if (ar2[i].equals(empl)) {
        pos.add(i);
        cont = true;
    }
}

    //Transfer the values of the Array List to an Array
    //Use the new Array to find the position of the client name
    //in the parallel Array created before.

if (cont == true) {
    int[] position = new int[pos.size()];
    for (int i = 0; i < position.length; i++) {
        position[i] = pos.get(i).intValue();
    }

    posOfTitle = new String[pos.size()];
    for (int i = 0; i < position.length; i++) {
        int val = position[i];
        posOfTitle[i] = ar1[val];
    }
}
```



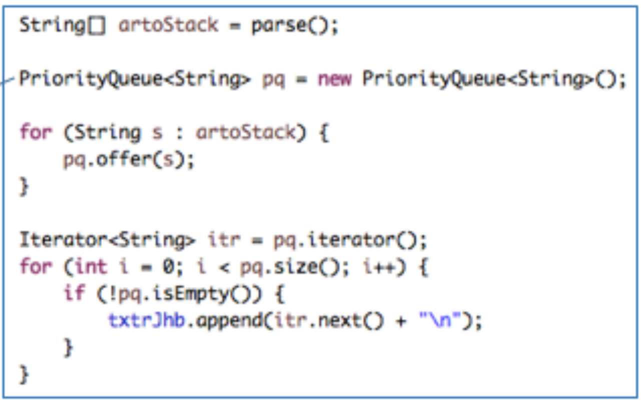
```
                // If employee name specified by the user does not match
                // with the name in the Arraylist parsed from the text file
                // Use the method to return that no values exist.

        else {
            posOfTitle = new String[1];
            posOfTitle[0] = "NO SUCH EMPLOYEE EXISTS";
        }
    }

    // If No file exsit by the name specified by the user
    // Use the method to return that no such file exists

    else {
            posOfTitle = new String[1];
            posOfTitle[0] = "NO SUCH FILE EXISTS";

        }

} catch (FileNotFoundException e) {
    e.printStackTrace();
}
// Return the string array containing
// the client name or the failure message

return posOfTitle;
```

Figure 6C: parseFile() method- parse Comments.txt to find client names for a single manager

The above code shows the method used to parse a text file to find all the client names for a single Team-Leader. ArrayList was used because of its dynamic property i.e. wherever the size of an array or a list was unknown an ArrayList was used. The predefined function .toArray() was used to convert an Arraylist to an array.

Parsing a text file was also needed in the AdminForm, where the user is given functionality to either add a new comment or display all the previous comments. I created a file called ‘Comments.txt’ where all the comments where written to, so when the user clicks the button to display all the comments, the file needed to be parsed and the last comment added by the user had to appear first on the screen first. The following code shows the method used to achieve this.

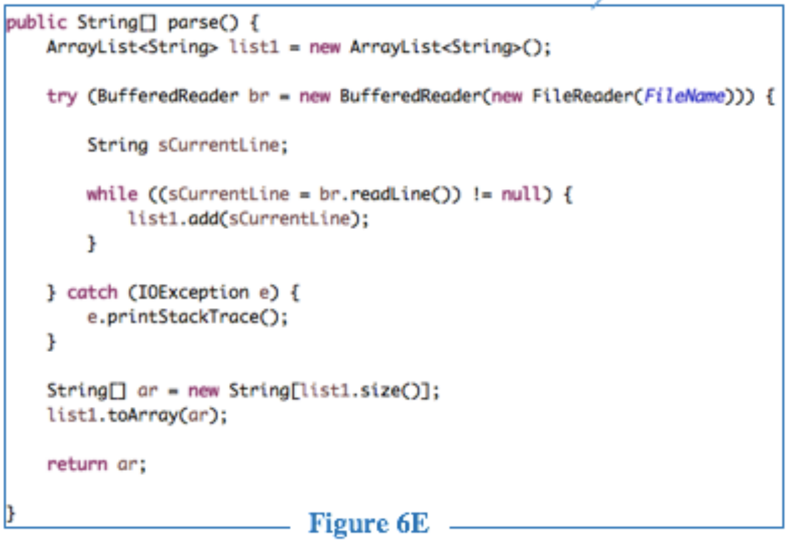
. Because ques in java follow the first-in-first-out procedure, whatever value that is first read by the line reader (latest comment of the user) is stored in the que first, which means that it is the first to come out and first to be displayed on the screen.



```
String[] artoStack = parse();

PriorityQueue<String> pq = new PriorityQueue<String>();

for (String s : artoStack) {
    pq.offer(s);
}

Iterator<String> itr = pq.iterator();
for (int i = 0; i < pq.size(); i++) {
    if (!pq.isEmpty()) {
        txtrJhb.append(itr.next() + "\n");
    }
}
```

Figure 6D: use of Ques to display Client Names on Screen

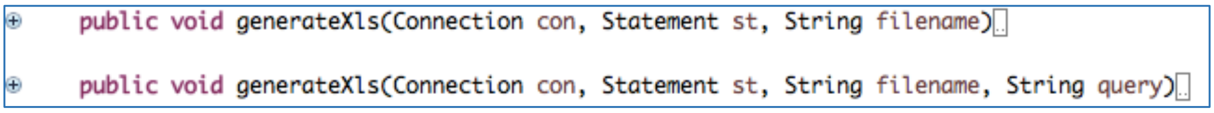
Figure 6E shows the code to show all comments that exists in a test file, Method parse() is called first, which return a string-type array containing all comments from the text



```
public String[] parse() {
    ArrayList<String> list1 = new ArrayList<String>();

    try (BufferedReader br = new BufferedReader(new FileReader(FileName))) {

        String sCurrentLine;

        while ((sCurrentLine = br.readLine()) != null) {
            list1.add(sCurrentLine);
        }

    } catch (IOException e) {
        e.printStackTrace();
    }

    String[] ar = new String[list1.size()];
    list1.toArray(ar);

    return ar;

}
```

Figure 6E

Polymorphism



```
⊕   public void generateXls(Connection con, Statement st, String filename)

⊕   public void generateXls(Connection con, Statement st, String filename, String query)
```

Figure 7

The above code shows overloading, where two methods have similar names but different parameters. The first method generates an Excel file an excel file of the all data in SisClient table is generated. The second method creates an excel file based on queries run by the user in the admin form (refer to appendix page 6 to see code for the two methods)

Word Count without heading, footnotes, captions - 1043

Code Sources:
Java Swing (GUI) Programming: From Beginner to Expert by John Purcell On Udemy

Complete Java Masterclass by Tim Bachulka on Udemy