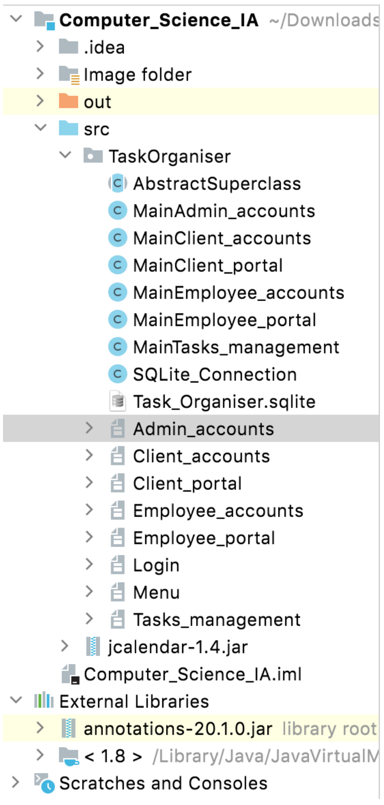
Criterion C

All classes:



```
Computer_Science_IA  ~/Downloads
  .idea
  Image folder
  out
  src
    TaskOrganiser
      AbstractSuperclass
      MainAdmin_accounts
      MainClient_accounts
      MainClient_portal
      MainEmployee_accounts
      MainEmployee_portal
      MainTasks_management
      SQLite_Connection
      Task_Organiser.sqlite
      Admin_accounts
      Client_accounts
      Client_portal
      Employee_accounts
      Employee_portal
      Login
      Menu
      Tasks_management
    jcalendar-1.4.jar
  Computer_Science_IA.iml
External Libraries
  annotations-20.1.0.jar  library root
  < 1.8 >  /Library/Java/JavaVirtualM
Scratches and Consoles
```

List of techniques used:

- Array list
- Hash table
- Static methods and variables
- Parameter passing
- Usage of a StringBuilder to enable a random password generator according to the role of the user.
- Simple and compound selection
- While loop
- For loop
- Method returning a value
- Nested loops
- Nested conditioning
- Filtering information from the database.
- Hierarchical inheritance from an abstract superclass.
- Polymorphism of base structure methods.

- Use of JAR file to implement calendars in the system[1][2]

<u>Structure of the program:</u>

The program follows a hierarchical inheritance-based structure with polymorphism-overloaded methods. This means, that an abstract superclass contains all the main methods of the system (in general terms), and by the use of parameters, they can be called from any other subclass to employ it in a different way by changing the arguments of the function. These methods were standardized and generalized so that they could suit every class and work flawlessly. Hence, this structure enabled the reusability of important methods and also an effective error handling and update approach; the logical and syntax errors were fixed only once in the abstract superclass. Furthermore, all of the classes that send data to the database were encapsulated to avoid misuse of important variables and protect the data of objects. Hence, these could only be accessed by accessors and changed with mutators.

<u>Generate a random password:</u>

I generated a random password by selecting characters from a fixed string and appending them to a StringBuilder. After the random characters were selected by means of an instance of the Random class in Java, a prefix (depending on what type of user it was) was added to the final password string that would be displayed.

[1] Words, G., Idea, W., asktejas, B., Flojea, M., freaks, U., 问说网, J., Ethics, I., softwareengineeringcodefoethic, S., Website, J., Aplikasi, D., JCalender &#8211; Tech Blicks | Tips, V., JAVA, K. and java.lang.IllegalStateException: Attempt to mutate in notification (JDateChooser, J. -. J. D. D. -. P.
Words, Google et al. "Jcalendar – Toedter.Com". Toedter.Com, 2023, https://toedter.com/jcalendar/. Accessed 20 Sept 2023.

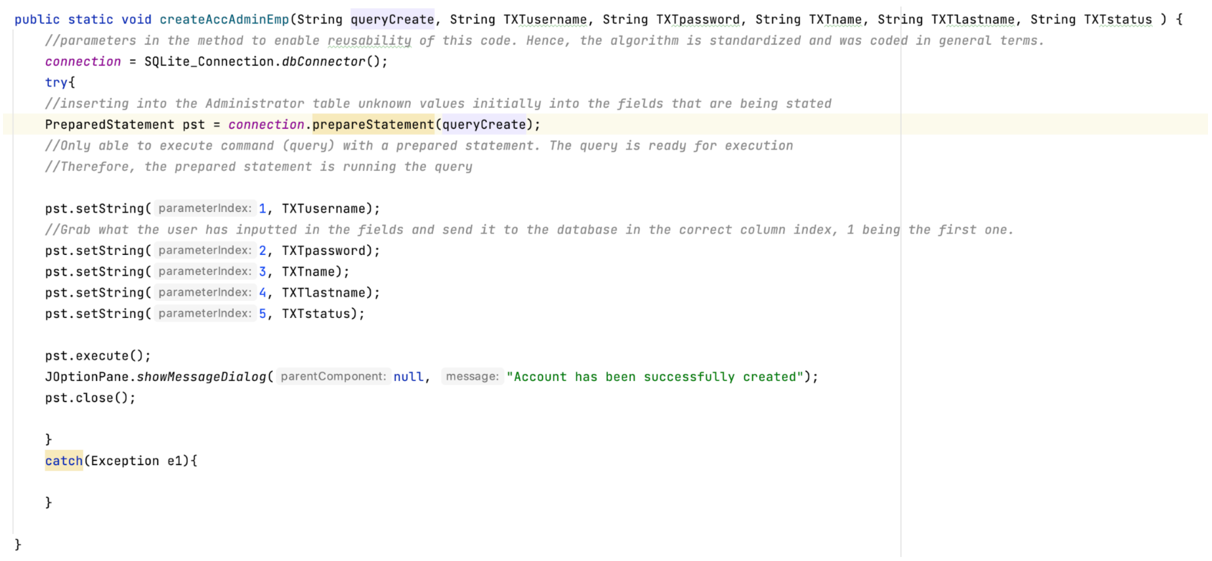
[2] Downloaded from: http://www.java2s.com/Code/Jar/j/Downloadjcalendar14jar.htm

```java
public static @NotNull String generatePassword(String prefix) {

    //creating a string of all characters
    String gen = "abcdefghijklmnopqrstuvwxyz123456789!·$%/()=¿?*^¨ç><'¡#";
    //creating a random string builder
    StringBuilder code = new StringBuilder();
    StringBuilder codef = new StringBuilder();
    // create an object of the random class
    Random random = new Random();
    // create a variable to specify the length of the randomly picked
    int length = 3;
    // for loop to loop through the selected characters and keep generating different strings
    // looping from 1-38, not through the characters.
    // Getting a random index and then use the random index to choose the character from gen
    for (int i = 0; i < length; i++) {
        // generate random index number
        int index = random.nextInt(gen.length());
        //generating character from string by specified index
        char randomChar = gen.charAt(index);
        // append the character to a string builder
        code.append(randomChar);
        //Join the prefix string to the randomly selected character to have the password
    }
    codef.append(prefix + code);

    //Turning the final code into a string using the toString method
    String FinalRandomString = codef.toString();
    return FinalRandomString;

}
```

Create an administrator or employee account:

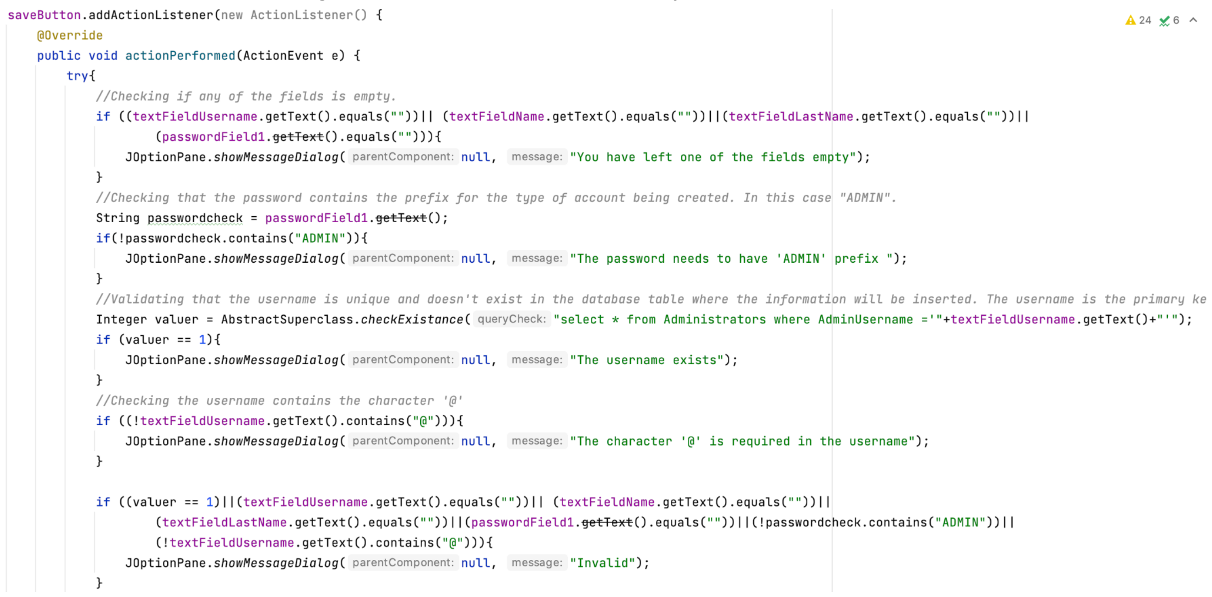
Since both the administrators and the employees share the same fields when one of their accounts is created, one method was created for both. The information is validated and then inserted into the database table by the use of a preparedStatement and a query that is specified within the class where the function is called. The query is sent as an argument to enable the reusability and versatility of this code: the query to create an employee account will be different from the one to create an administrator account, however, the body of the algorithm is the same[3].

[3] See appendix 1 for a sample query used.



```
public static void createAccAdminEmp(String queryCreate, String TXTusername, String TXTpassword, String TXTname, String TXTlastname, String TXTstatus ) {
    //parameters in the method to enable reusability of this code. Hence, the algorithm is standardized and was coded in general terms.
    connection = SQLite_Connection.dbConnector();
    try{
    //inserting into the Administrator table unknown values initially into the fields that are being stated
    PreparedStatement pst = connection.prepareStatement(queryCreate);
    //Only able to execute command (query) with a prepared statement. The query is ready for execution
    //Therefore, the prepared statement is running the query

    pst.setString( parameterIndex: 1, TXTusername);
    //Grab what the user has inputted in the fields and send it to the database in the correct column index, 1 being the first one.
    pst.setString( parameterIndex: 2, TXTpassword);
    pst.setString( parameterIndex: 3, TXTname);
    pst.setString( parameterIndex: 4, TXTlastname);
    pst.setString( parameterIndex: 5, TXTstatus);

    pst.execute();
    JOptionPane.showMessageDialog( parentComponent: null,  message: "Account has been successfully created");
    pst.close();

    }
    catch(Exception e1){

    }

}
```

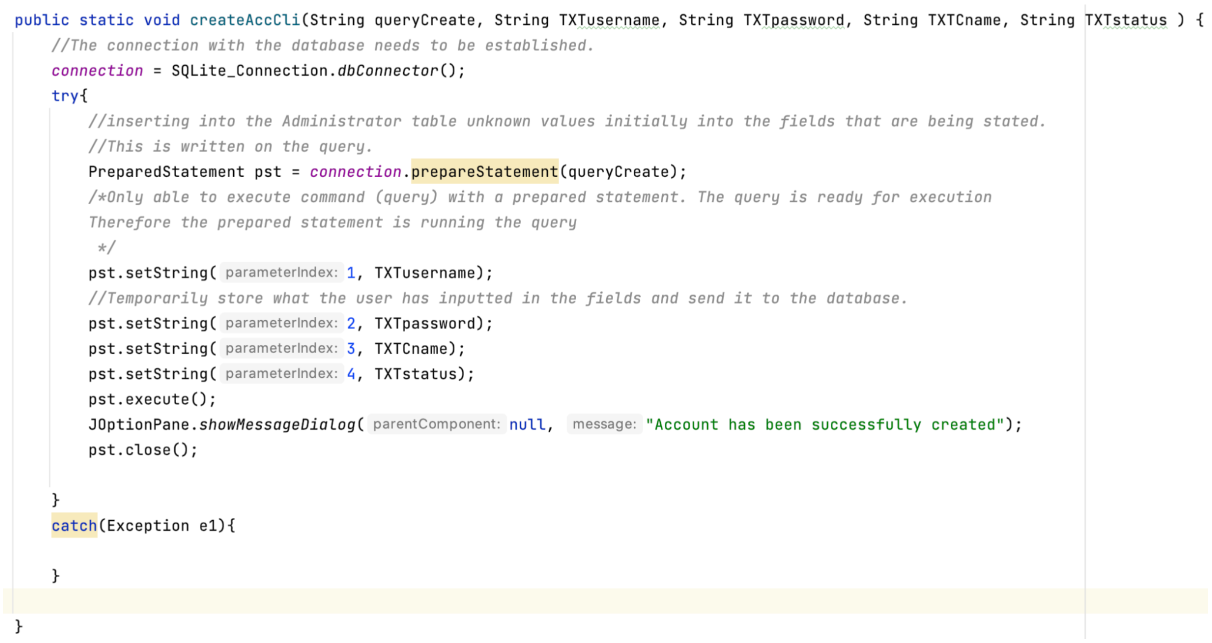
Validations when creating an administrator or employee account:



```
saveButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        try{
            //Checking if any of the fields is empty.
            if ((textFieldUsername.getText().equals(""))|| (textFieldName.getText().equals(""))||(textFieldLastName.getText().equals(""))||
                    (passwordField1.~~getText~~().equals(""))){
                JOptionPane.showMessageDialog( parentComponent: null,  message: "You have left one of the fields empty");
            }
            //Checking that the password contains the prefix for the type of account being created. In this case "ADMIN".
            String passwordcheck = passwordField1.~~getText~~();
            if(!passwordcheck.contains("ADMIN")){
                JOptionPane.showMessageDialog( parentComponent: null,  message: "The password needs to have 'ADMIN' prefix ");
            }
            //Validating that the username is unique and doesn't exist in the database table where the information will be inserted. The username is the primary ke
            Integer valuer = AbstractSuperclass.checkExistance( queryCheck: "select * from Administrators where AdminUsername ='"+textFieldUsername.getText()+"'");
            if (valuer == 1){
                JOptionPane.showMessageDialog( parentComponent: null,  message: "The username exists");
            }
            //Checking the username contains the character '@'
            if ((!textFieldUsername.getText().contains("@"))){
                JOptionPane.showMessageDialog( parentComponent: null,  message: "The character '@' is required in the username");
            }

            if ((valuer == 1)||(textFieldUsername.getText().equals(""))|| (textFieldName.getText().equals(""))||
                    (textFieldLastName.getText().equals(""))||(passwordField1.~~getText~~().equals(""))||(!passwordcheck.contains("ADMIN"))||
                    (!textFieldUsername.getText().contains("@"))){
                JOptionPane.showMessageDialog( parentComponent: null,  message: "Invalid");
            }
```

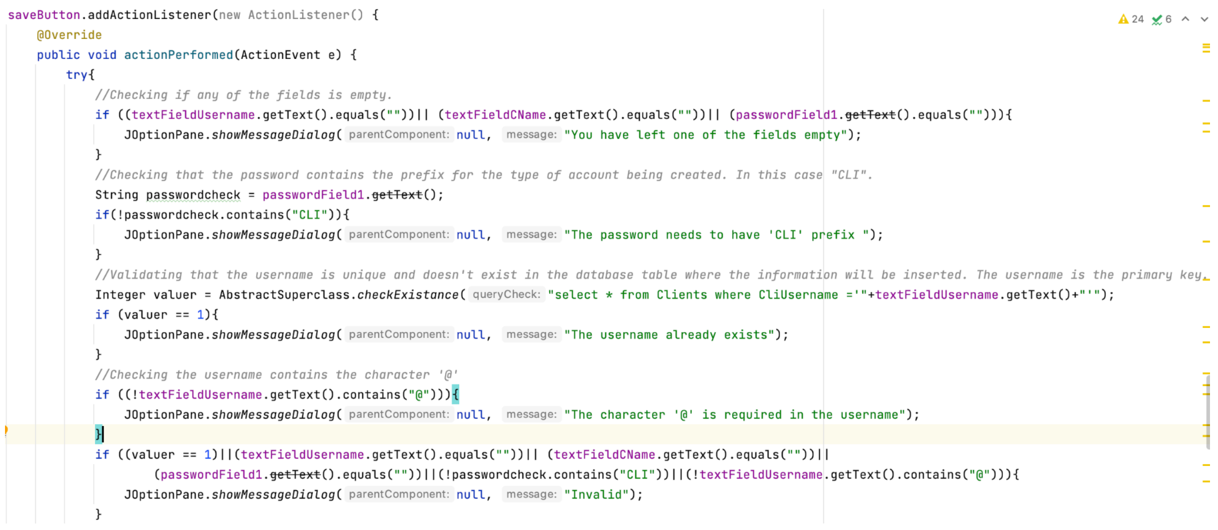
Creating a client account

The algorithm is very similar to the one that creates an administrator or employee account; the logical commands are the same. The only change is the data required to create an account. This means that the column indexes, the information to insert, and hence, the method parameters are different.



```java
public static void createAccCli(String queryCreate, String TXTusername, String TXTpassword, String TXTCname, String TXTstatus ) {
    //The connection with the database needs to be established.
    connection = SQLite_Connection.dbConnector();
    try{
        //inserting into the Administrator table unknown values initially into the fields that are being stated.
        //This is written on the query.
        PreparedStatement pst = connection.prepareStatement(queryCreate);
        /*Only able to execute command (query) with a prepared statement. The query is ready for execution
        Therefore the prepared statement is running the query
         */
        pst.setString( parameterIndex: 1, TXTusername);
        //Temporarily store what the user has inputted in the fields and send it to the database.
        pst.setString( parameterIndex: 2, TXTpassword);
        pst.setString( parameterIndex: 3, TXTCname);
        pst.setString( parameterIndex: 4, TXTstatus);
        pst.execute();
        JOptionPane.showMessageDialog( parentComponent: null,  message: "Account has been successfully created");
        pst.close();

    }
    catch(Exception e1){

    }

}
```

Validations when creating a client account:

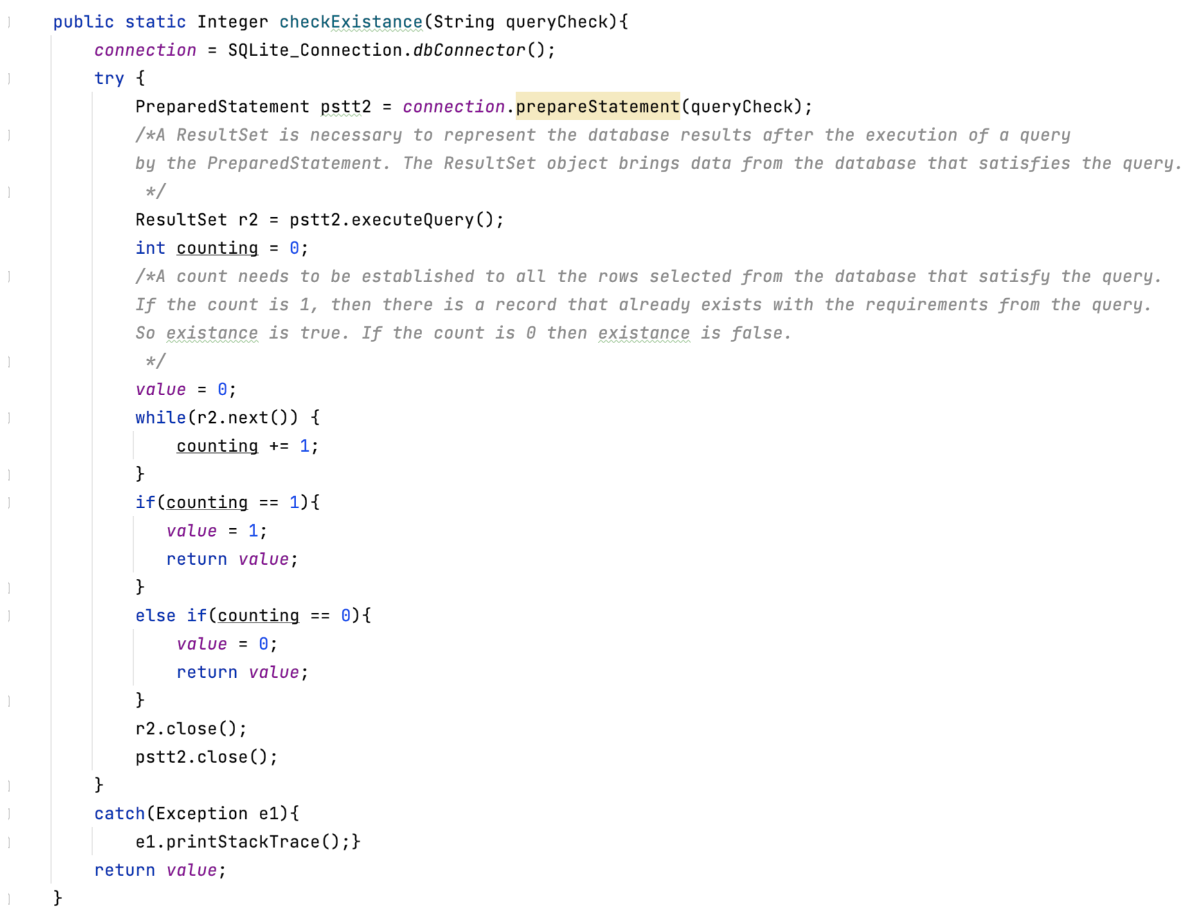


```java
saveButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        try{
            //Checking if any of the fields is empty.
            if ((textFieldUsername.getText().equals(""))|| (textFieldCName.getText().equals(""))|| (passwordField1.getText().equals(""))){
                JOptionPane.showMessageDialog( parentComponent: null,  message: "You have left one of the fields empty");
            }
            //Checking that the password contains the prefix for the type of account being created. In this case "CLI".
            String passwordcheck = passwordField1.getText();
            if(!passwordcheck.contains("CLI")){
                JOptionPane.showMessageDialog( parentComponent: null,  message: "The password needs to have 'CLI' prefix ");
            }
            //Validating that the username is unique and doesn't exist in the database table where the information will be inserted. The username is the primary key.
            Integer valuer = AbstractSuperclass.checkExistance( queryCheck: "select * from Clients where CliUsername ='"+textFieldUsername.getText()+"'");
            if (valuer == 1){
                JOptionPane.showMessageDialog( parentComponent: null,  message: "The username already exists");
            }
            //Checking the username contains the character '@'
            if ((!textFieldUsername.getText().contains("@"))){
                JOptionPane.showMessageDialog( parentComponent: null,  message: "The character '@' is required in the username");
            }
            if ((valuer == 1)||(textFieldUsername.getText().equals(""))|| (textFieldCName.getText().equals(""))||
                    (passwordField1.getText().equals(""))||(!passwordcheck.contains("CLI"))||(!textFieldUsername.getText().contains("@"))){
                JOptionPane.showMessageDialog( parentComponent: null,  message: "Invalid");
            }
```

Checking the existence of accounts:

The following method is utilized in the login and in the validation of the creation of accounts. Both the ResultSet and the PreparedStatement are being used to execute the query and obtained a filtered set of data according to the requirements of the query[4].

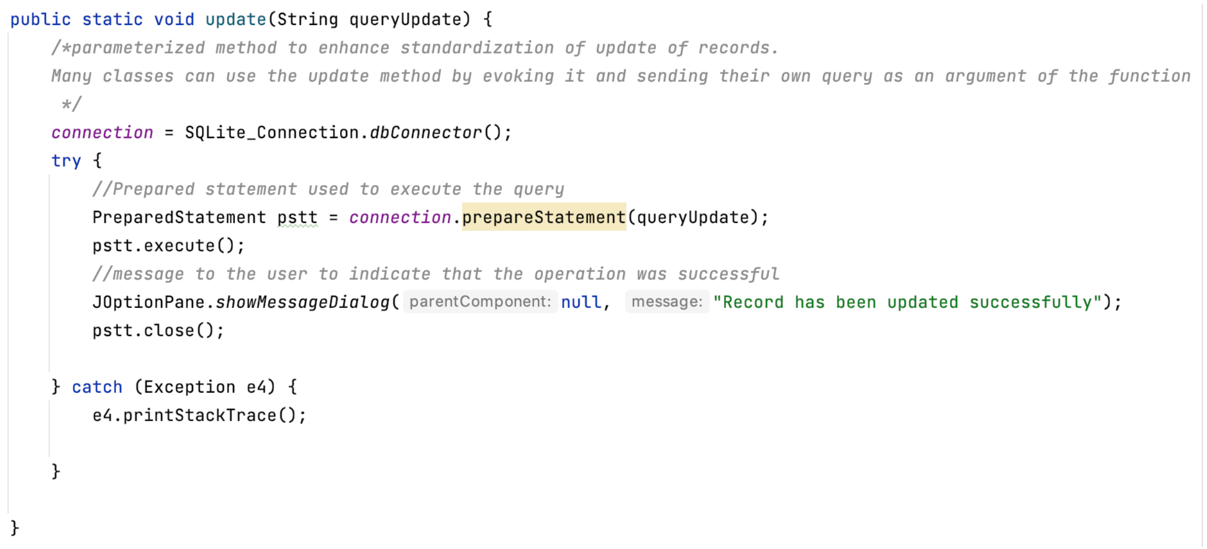
[4] See appendix 2 for a sample query used.



```java
public static Integer checkExistance(String queryCheck){
    connection = SQLite_Connection.dbConnector();
    try {
        PreparedStatement pstt2 = connection.prepareStatement(queryCheck);
        /*A ResultSet is necessary to represent the database results after the execution of a query
        by the PreparedStatement. The ResultSet object brings data from the database that satisfies the query.
         */
        ResultSet r2 = pstt2.executeQuery();
        int counting = 0;
        /*A count needs to be established to all the rows selected from the database that satisfy the query.
        If the count is 1, then there is a record that already exists with the requirements from the query.
        So existance is true. If the count is 0 then existance is false.
         */
        value = 0;
        while(r2.next()) {
            counting += 1;
        }
        if(counting == 1){
           value = 1;
           return value;
        }
        else if(counting == 0){
            value = 0;
            return value;
        }
        r2.close();
        pstt2.close();
    }
    catch(Exception e1){
        e1.printStackTrace();}
    return value;
}
```

Update of account and task information:

Again, the prepared statement is used to execute the update query being sent by the classes. Basically, the query is the one that does the update.[5]



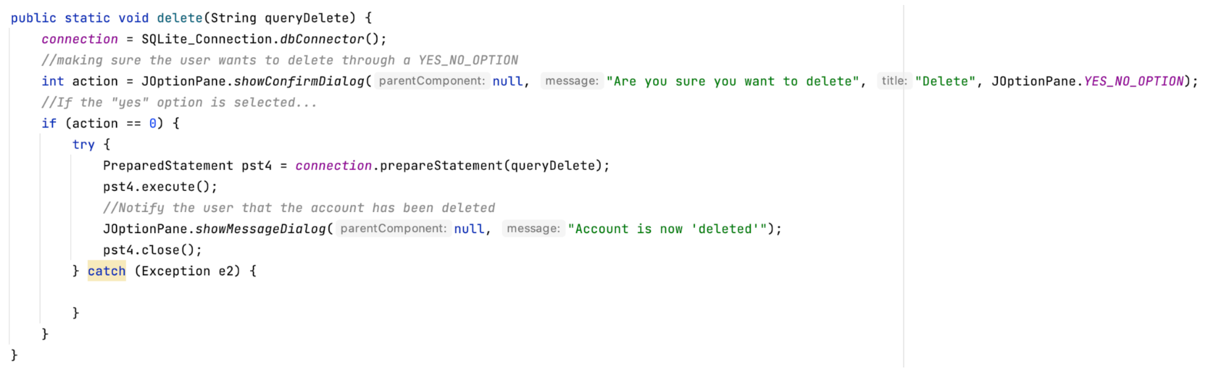
```java
public static void update(String queryUpdate) {
    /*parameterized method to enhance standardization of update of records.
    Many classes can use the update method by evoking it and sending their own query as an argument of the function
     */
    connection = SQLite_Connection.dbConnector();
    try {
        //Prepared statement used to execute the query
        PreparedStatement pstt = connection.prepareStatement(queryUpdate);
        pstt.execute();
        //message to the user to indicate that the operation was successful
        JOptionPane.showMessageDialog( parentComponent: null,  message: "Record has been updated successfully");
        pstt.close();

    } catch (Exception e4) {
        e4.printStackTrace();

    }

}
```

Logical removal of an account or task:

Since the client company wanted to keep track of the history of all the tasks and accounts created, a normal removal of records cannot take place. Therefore, this algorithm carries out

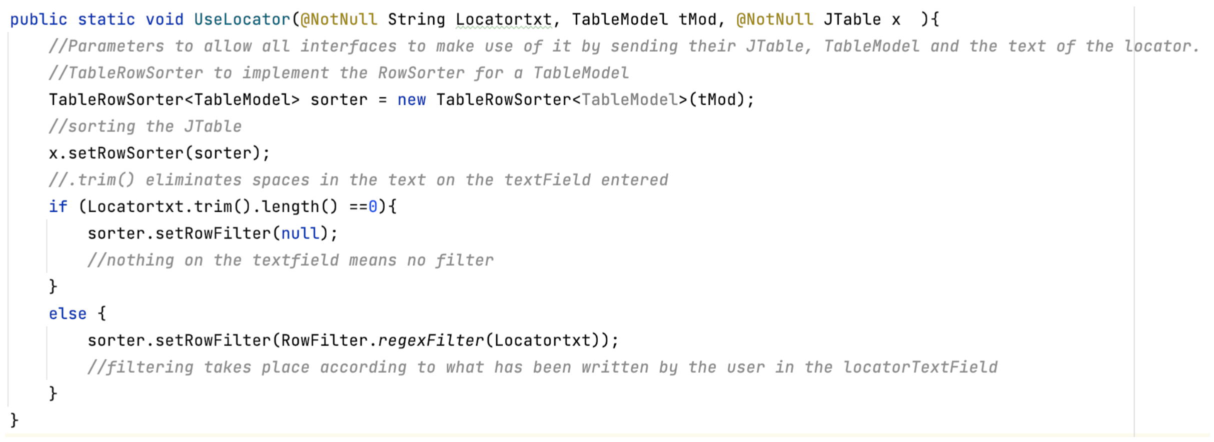
[5] See appendix 3 for a sample query used.

a logical deletion of the record, meaning that its status changes to “deleted”. The query specifies this action and the prepared statement executes it [6]. Furthermore, a “YES_NO_OPTION” is indispensable to avoid mistakes.



```
public static void delete(String queryDelete) {
    connection = SQLite_Connection.dbConnector();
    //making sure the user wants to delete through a YES_NO_OPTION
    int action = JOptionPane.showConfirmDialog( parentComponent: null,  message: "Are you sure you want to delete",  title: "Delete", JOptionPane.YES_NO_OPTION);
    //If the "yes" option is selected...
    if (action == 0) {
        try {
            PreparedStatement pst4 = connection.prepareStatement(queryDelete);
            pst4.execute();
            //Notify the user that the account has been deleted
            JOptionPane.showMessageDialog( parentComponent: null,  message: "Account is now 'deleted'");
            pst4.close();
        } catch (Exception e2) {

        }
    }
}
```

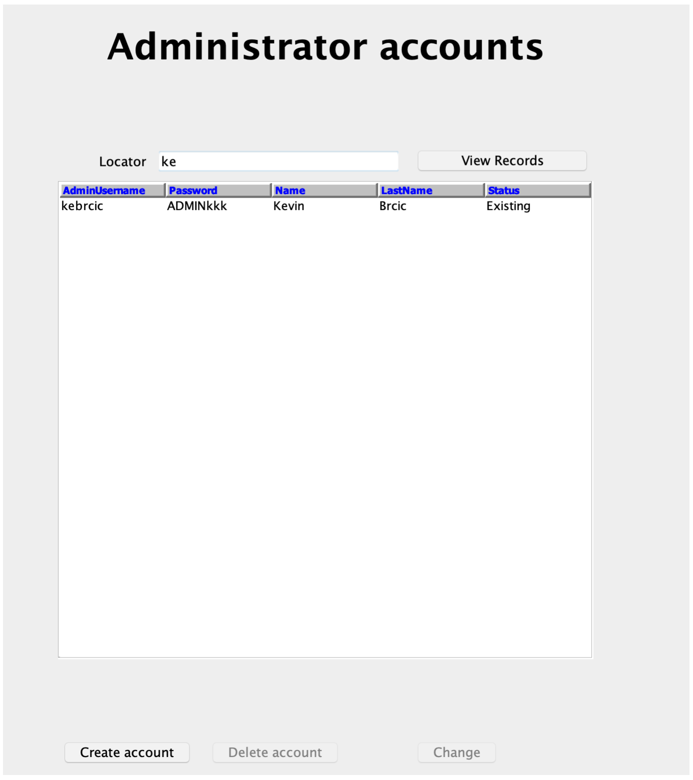
<u>Filtering by the use of a locator:</u>
In order to sort the data displayed on the JTable, I designed an algorithm that filters information according to the characters written over a textField. A sorter and a regexFilter was employed to filter the data based on the information written in a locator textField. The .trim() method ensures that there is data on the textField to perform the filtering.



```
public static void UseLocator(@NotNull String Locatortxt, TableModel tMod, @NotNull JTable x  ){
    //Parameters to allow all interfaces to make use of it by sending their JTable, TableModel and the text of the locator.
    //TableRowSorter to implement the RowSorter for a TableModel
    TableRowSorter<TableModel> sorter = new TableRowSorter<TableModel>(tMod);
    //sorting the JTable
    x.setRowSorter(sorter);
    //.trim() eliminates spaces in the text on the textField entered
    if (Locatortxt.trim().length() ==0){
        sorter.setRowFilter(null);
        //nothing on the textfield means no filter
    }
    else {
        sorter.setRowFilter(RowFilter.regexFilter(Locatortxt));
        //filtering takes place according to what has been written by the user in the locatorTextField
    }
}
```

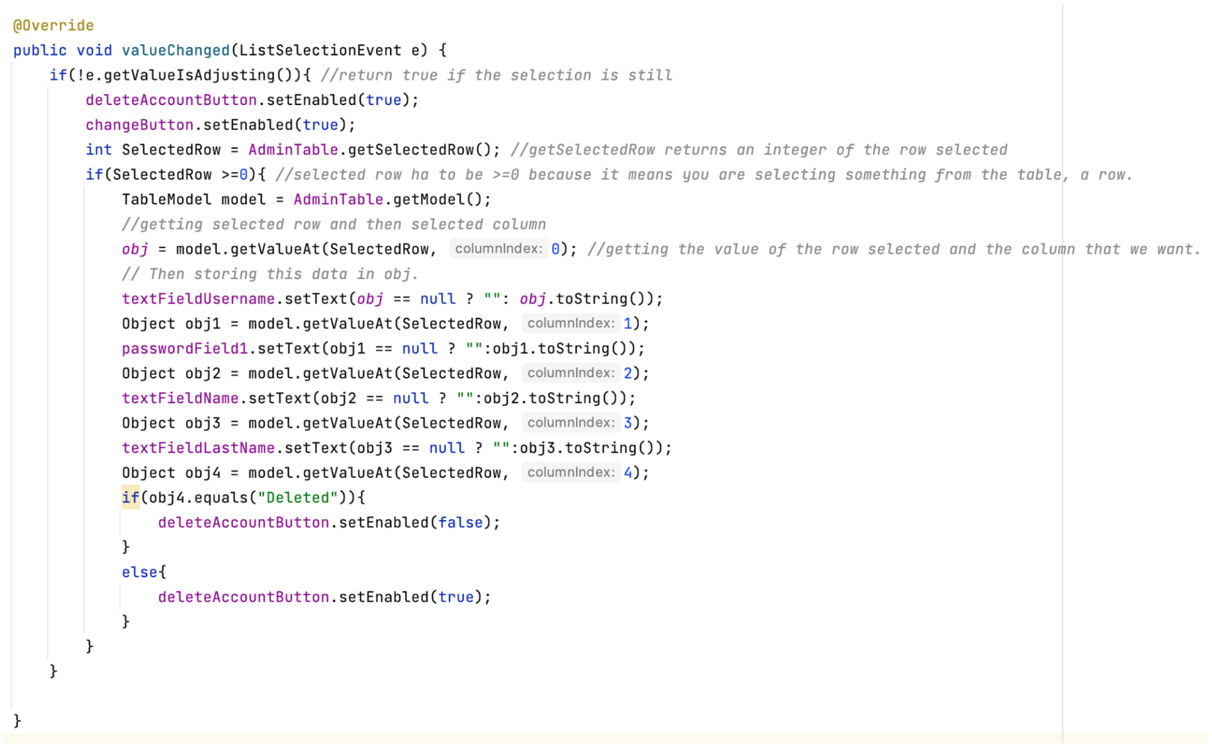
Sample output of the use of the algorithm:

[6] See appendix 4 for a sample query used.



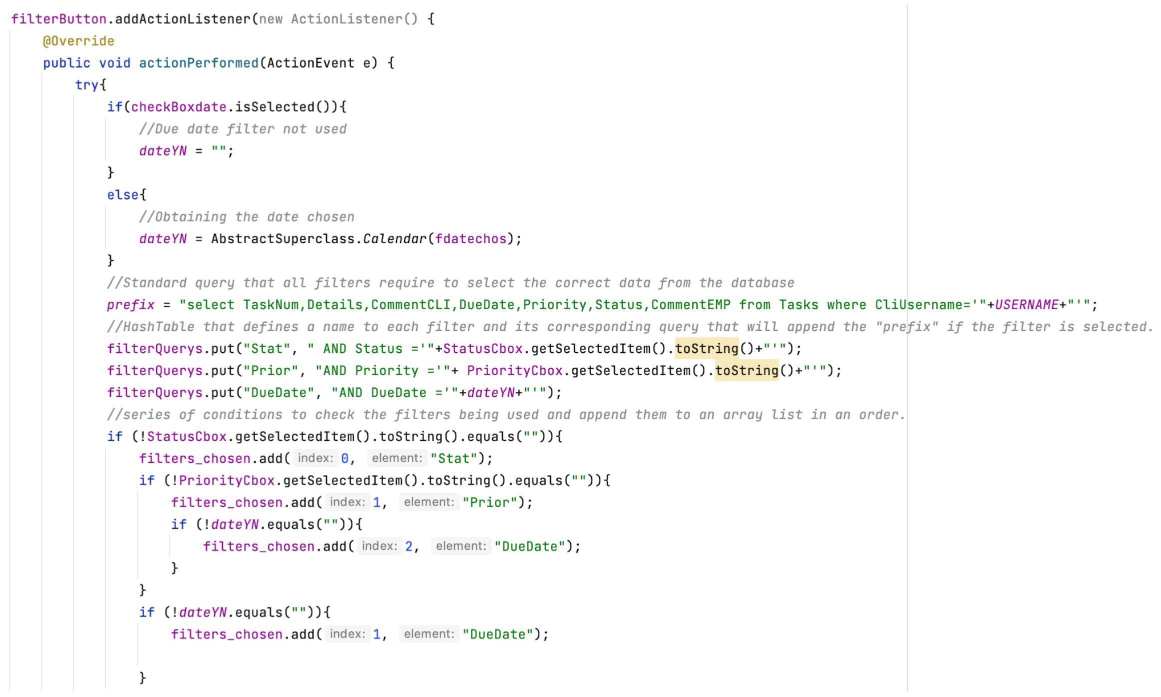
AdminUsername	Password	Name	LastName	Status
kebrcic	ADMINkkk	Kevin	Brcic	Existing

<u>Identification of selected rows from JTable:</u>
By using the Java listener valueChanged, the user interactions with the JTable (selection of rows) were identified as this selection triggered the method. The value of each field of the selected row was stored in an object as an index (row index, column index) and then converted to a string in order to display it in a textField to the user. This method enabled all users to select a row from the JTable and modify, delete, or make changes to its fields without having to manually specify the primary key of the record to update it; it makes the interfaces user-friendly and more automatized.

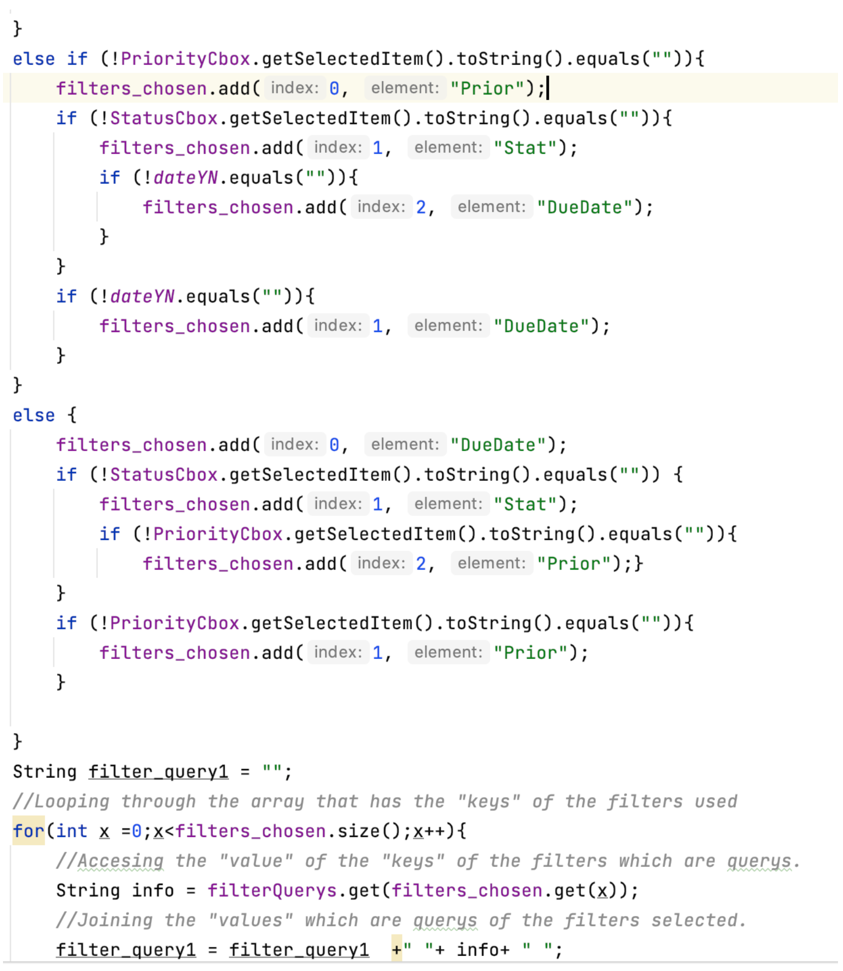


```java
@Override
public void valueChanged(ListSelectionEvent e) {
    if(!e.getValueIsAdjusting()){ //return true if the selection is still
        deleteAccountButton.setEnabled(true);
        changeButton.setEnabled(true);
        int SelectedRow = AdminTable.getSelectedRow(); //getSelectedRow returns an integer of the row selected
        if(SelectedRow >=0){ //selected row ha to be >=0 because it means you are selecting something from the table, a row.
            TableModel model = AdminTable.getModel();
            //getting selected row and then selected column
            obj = model.getValueAt(SelectedRow,  columnIndex: 0); //getting the value of the row selected and the column that we want.
            // Then storing this data in obj.
            textFieldUsername.setText(obj == null ? "": obj.toString());
            Object obj1 = model.getValueAt(SelectedRow,  columnIndex: 1);
            passwordField1.setText(obj1 == null ? "":obj1.toString());
            Object obj2 = model.getValueAt(SelectedRow,  columnIndex: 2);
            textFieldName.setText(obj2 == null ? "":obj2.toString());
            Object obj3 = model.getValueAt(SelectedRow,  columnIndex: 3);
            textFieldLastName.setText(obj3 == null ? "":obj3.toString());
            Object obj4 = model.getValueAt(SelectedRow,  columnIndex: 4);
            if(obj4.equals("Deleted")){
                deleteAccountButton.setEnabled(false);
            }
            else{
                deleteAccountButton.setEnabled(true);
            }
        }
    }

}
```

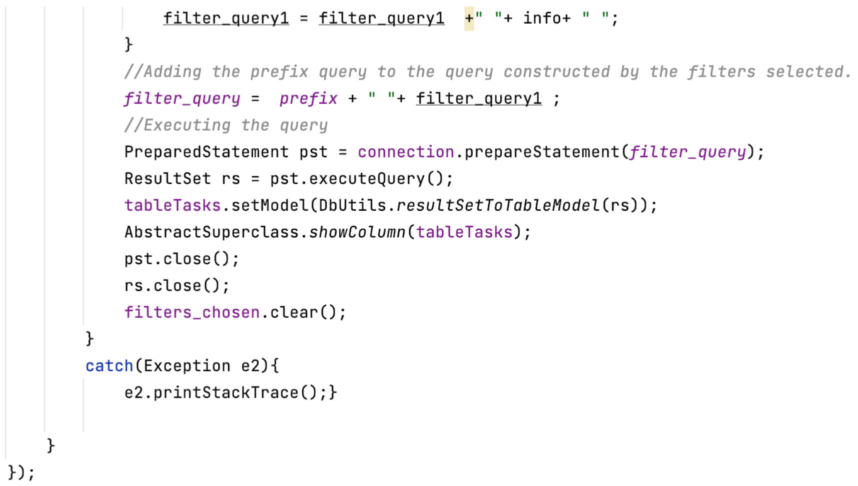
<u>Flexible filtering:</u>
This algorithm provides the user with the flexibility to filter the information from the database table in any way by using the filters in the interface. This means that the filtering is undergone with the filters that the user selects.
By the use of a hashtable, the algorithm defines the query for each filter that could append the prefix query to undergo correct filtering. Through a series of nested conditions, the filters selected are identified and the “keys” of these filters are stored in an arraylist. The for loop enables the “values” to be accessed from the “keys” of the hashtable, which are querys, to then construct a complete query that will be executed with the PreparedStatement.



```java
filterButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        try{
            if(checkBoxdate.isSelected()){
                //Due date filter not used
                dateYN = "";
            }
            else{
                //Obtaining the date chosen
                dateYN = AbstractSuperclass.Calendar(fdatechos);
            }
            //Standard query that all filters require to select the correct data from the database
            prefix = "select TaskNum,Details,CommentCLI,DueDate,Priority,Status,CommentEMP from Tasks where CliUsername='"+USERNAME+"'";
            //HashTable that defines a name to each filter and its corresponding query that will append the "prefix" if the filter is selected.
            filterQuerys.put("Stat", " AND Status ='"+StatusCbox.getSelectedItem().toString()+"'");
            filterQuerys.put("Prior", "AND Priority ='"+ PriorityCbox.getSelectedItem().toString()+"'");
            filterQuerys.put("DueDate", "AND DueDate ='"+dateYN+"'");
            //series of conditions to check the filters being used and append them to an array list in an order.
            if (!StatusCbox.getSelectedItem().toString().equals("")){
                filters_chosen.add( index: 0,  element: "Stat");
                if (!PriorityCbox.getSelectedItem().toString().equals("")){
                    filters_chosen.add( index: 1,  element: "Prior");
                    if (!dateYN.equals("")){
                        filters_chosen.add( index: 2,  element: "DueDate");
                    }
                }
                if (!dateYN.equals("")){
                    filters_chosen.add( index: 1,  element: "DueDate");

                }
```

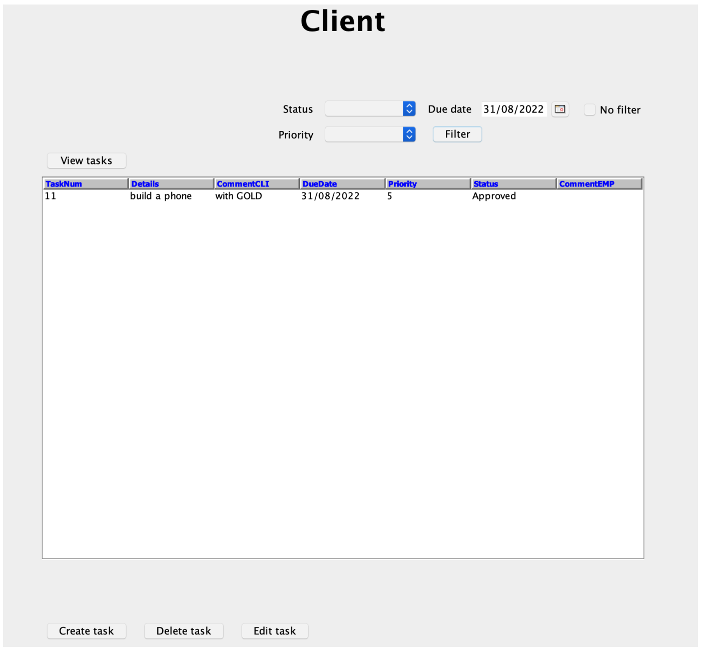


```java
}
else if (!PriorityCbox.getSelectedItem().toString().equals("")){
    filters_chosen.add( index: 0,  element: "Prior");
    if (!StatusCbox.getSelectedItem().toString().equals("")){
        filters_chosen.add( index: 1,  element: "Stat");
        if (!dateYN.equals("")){
            filters_chosen.add( index: 2,  element: "DueDate");
        }
    }
    if (!dateYN.equals("")){
        filters_chosen.add( index: 1,  element: "DueDate");
    }
}
else {
    filters_chosen.add( index: 0,  element: "DueDate");
    if (!StatusCbox.getSelectedItem().toString().equals("")) {
        filters_chosen.add( index: 1,  element: "Stat");
        if (!PriorityCbox.getSelectedItem().toString().equals("")){
            filters_chosen.add( index: 2,  element: "Prior");}
    }
    if (!PriorityCbox.getSelectedItem().toString().equals("")){
        filters_chosen.add( index: 1,  element: "Prior");
    }

}
String filter_query1 = "";
//Looping through the array that has the "keys" of the filters used
for(int x =0;x<filters_chosen.size();x++){
    //Accesing the "value" of the "keys" of the filters which are querys.
    String info = filterQuerys.get(filters_chosen.get(x));
    //Joining the "values" which are querys of the filters selected.
    filter_query1 = filter_query1  +" "+ info+ " ";
```



```java
                filter_query1 = filter_query1  +" "+ info+ " ";
            }
            //Adding the prefix query to the query constructed by the filters selected.
            filter_query =  prefix + " "+ filter_query1 ;
            //Executing the query
            PreparedStatement pst = connection.prepareStatement(filter_query);
            ResultSet rs = pst.executeQuery();
            tableTasks.setModel(DbUtils.resultSetToTableModel(rs));
            AbstractSuperclass.showColumn(tableTasks);
            pst.close();
            rs.close();
            filters_chosen.clear();
        }
        catch(Exception e2){
            e2.printStackTrace();}

    }
});
```

Sample output of the use of the algorithm:

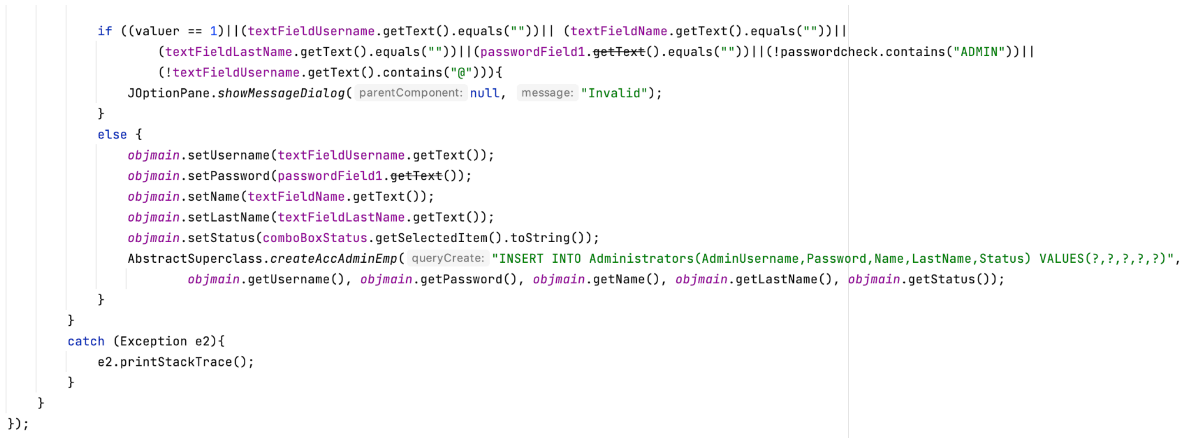


31/08/2022

TaskNum	Details	CommentCLI	DueDate	Priority	Status	CommentEMP
11	build a phone	with GOLD	31/08/2022	5	Approved	

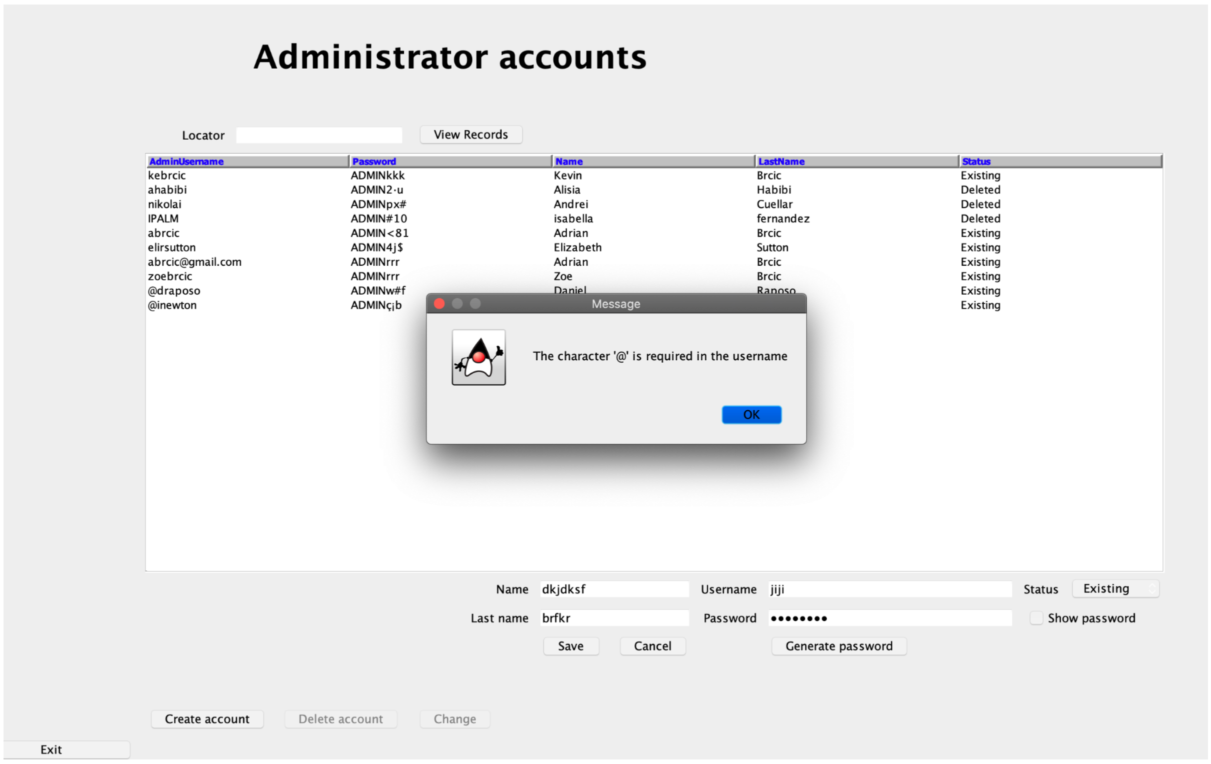
Error Handling:

To inform the user of any error I used try-catch blocks and also the class library JOptionPane to provide an information message with the specific error that followed a set of validations within the algorithms.



```
            if ((valuer == 1)||(textFieldUsername.getText().equals(""))|| (textFieldName.getText().equals(""))||
                    (textFieldLastName.getText().equals(""))||(passwordField1.~~getText~~().equals(""))||(!passwordcheck.contains("ADMIN"))||
                    (!textFieldUsername.getText().contains("@"))){
                JOptionPane.showMessageDialog( parentComponent: null,  message: "Invalid");
            }
            else {
                objmain.setUsername(textFieldUsername.getText());
                objmain.setPassword(passwordField1.~~getText~~());
                objmain.setName(textFieldName.getText());
                objmain.setLastName(textFieldLastName.getText());
                objmain.setStatus(comboBoxStatus.getSelectedItem().toString());
                AbstractSuperclass.createAccAdminEmp( queryCreate: "INSERT INTO Administrators(AdminUsername,Password,Name,LastName,Status) VALUES(?,?,?,?,?)",
                        objmain.getUsername(), objmain.getPassword(), objmain.getName(), objmain.getLastName(), objmain.getStatus());
            }
        }
        catch (Exception e2){
            e2.printStackTrace();
        }
    }
});
```

Sample output:



Administrator accounts

Locator

AdminUsername	Password	Name	LastName	Status
kebrcic	ADMINkkk	Kevin	Brcic	Existing
ahabibi	ADMIN2·u	Alisia	Habibi	Deleted
nikolai	ADMINpx#	Andrei	Cuellar	Deleted
IPALM	ADMIN#10	isabella	fernandez	Deleted
abrcic	ADMIN<81	Adrian	Brcic	Existing
elirsutton	ADMIN4j$	Elizabeth	Sutton	Existing
abrcic@gmail.com	ADMINrrr	Adrian	Brcic	Existing
zoebrcic	ADMINrrr	Zoe	Brcic	Existing
@draposo	ADMINw#f	Daniel	Raposo	Existing
@inewton	ADMINç¡b			Existing

Message

The character '@' is required in the username

Name	dkjdksf	Username	jiji	Status	Existing
Last name	brfkr	Password	••••••••		Show password

Word Count: 953