BAY HOUSE SCHOOL AND SIXTH FORM

Chair of Governors Training Tracker

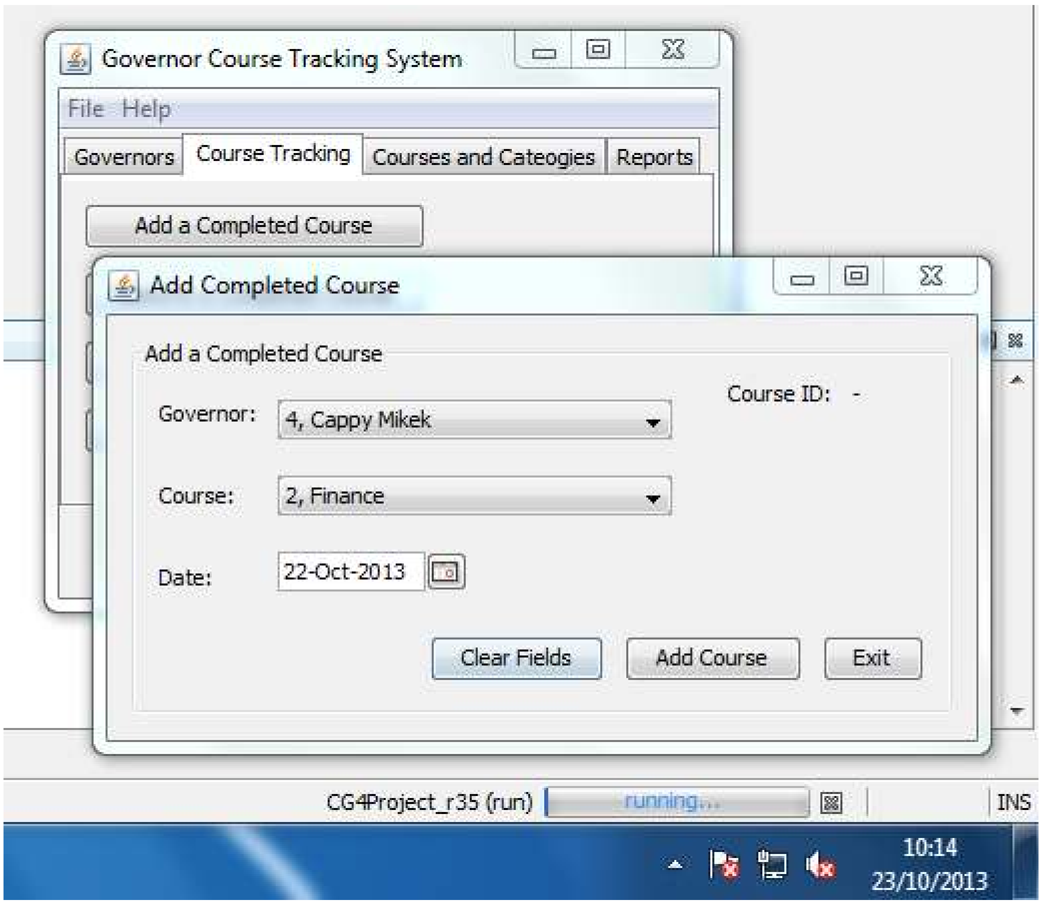
CG4.3 – Software Development

John Robinson

09/01/2014

Contents

Adding



Governor: 4, Cappy Mikek

Course ID: -

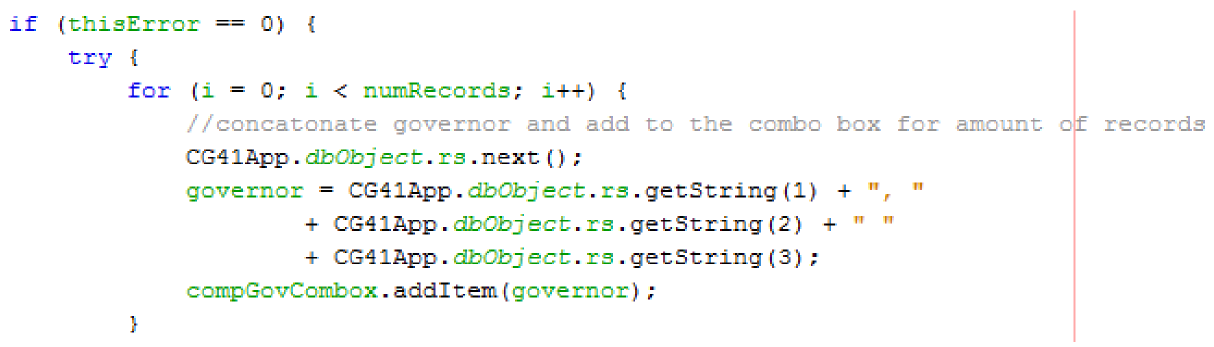
Course: 2, Finance

Date: 22-Oct-2013

Above we have the form for adding a completed course. The user will select a Governor Already in the Database from a drop down combo box. The user will also do the same for the course. Finally, the user will select a date from the calendar box.

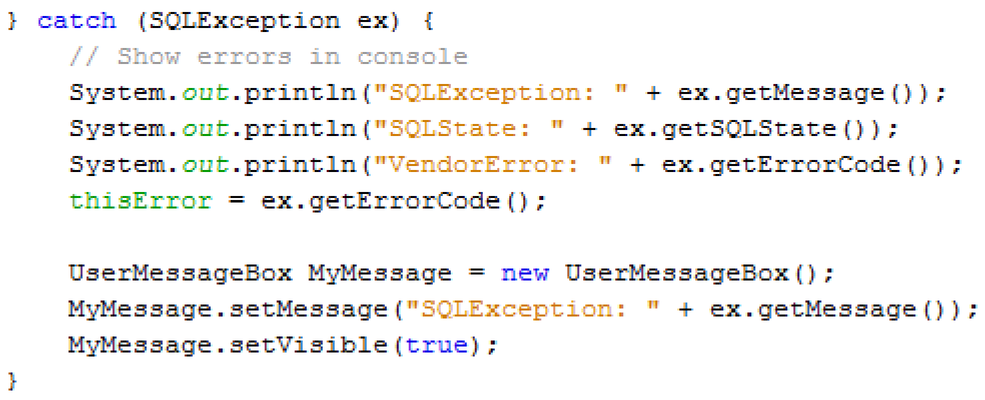
```
/**
 * This method loads the governors into the combo box
 */
private void loadGovernors() {
    CG41App.dbObject.sqlString = "select COUNT(*) from governor"; //get count
    thisError = CG41App.dbObject.getCountBySelect(); //execute count query
    numRecords = CG41App.dbObject.NumberOfRecords; //get number of records
    CG41App.dbObject.sqlString = "select GovID, Firstname, Surname from governor";
    thisError = CG41App.dbObject.getRecordSetBySelect(); //execute query
```

Above is the start of the code for adding the governors to the combo box. Here I interface with a database object given to me by my teacher. I count the amount of records and store it in an integer for later. Then I select all the governors to use in the next part of the code.



```
if (thisError == 0) {
    try {
        for (i = 0; i < numRecords; i++) {
            //concatonate governor and add to the combo box for amount of records
            CG41App.dbObject.rs.next();
            governor = CG41App.dbObject.rs.getString(1) + ", "
                    + CG41App.dbObject.rs.getString(2) + " "
                    + CG41App.dbObject.rs.getString(3);
            compGovCombox.addItem(governor);
        }
```

The code above will execute if there's no error accessing the database. I use a FOR loop to add the governors to the combo box. From the dbObject, I can pull each part of the governor from the Record Set to create a string to add to the database.

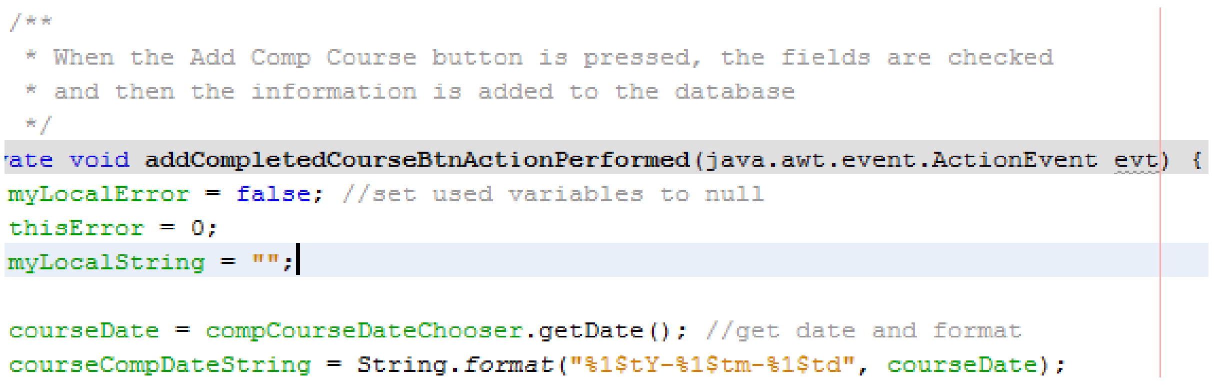


```
} catch (SQLException ex) {
    // Show errors in console
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
    thisError = ex.getErrorCode();

    UserMessageBox MyMessage = new UserMessageBox();
    MyMessage.setMessage("SQLException: " + ex.getMessage());
    MyMessage.setVisible(true);
}
```

As there was a TRY statement, I need to use a CATCH statement to catch any errors that appear from trying to retrieve from the record set. This is then displayed to the user using a message box while also being written to the console for debugging in the IDE.

While the above code is used for adding the governors to the combo box, a similar method is used for adding the courses their combo box.

The calendar piece is from an external library that I didn't write but have been able to exploit its features to benefit my code and system. It allows me to ensure the user can only insert a correctly formatted date.



```
/**
 * When the Add Comp Course button is pressed, the fields are checked
 * and then the information is added to the database
 */
ate void addCompletedCourseBtnActionPerformed(java.awt.event.ActionEvent evt) {
myLocalError = false; //set used variables to null
thisError = 0;
myLocalString = "";

courseDate = compCourseDateChooser.getDate(); //get date and format
courseCompDateString = String.format("%1$tY-%1$tm-%1$td", courseDate);
```

The above code shows the beginning of what happens when the Add Course button is pressed. The variables are reset to ensure that any left over use of the variables don't conflict with the rest of the code.

The date has be taken from the date chooser and converted into a useful format. The Date class has a useful function called “format” which allows me to format the date however I want by using a pattern.

```
//This if statement will check to make sure the user has selected everything
if(courseCompDateString.equals("null-null-null")
        || compGovCombox.getSelectedItem().equals("-")
        || compCourseCombox.getSelectedItem().equals("-")){

    myLocalError = true;
    //error message for the user
    UserMessageBox MyMessage = new UserMessageBox();
    MyMessage.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    MyMessage.setMessage("Error adding completed course.\n"
            + "Please make sure you have selected something\nin all fields.\n"
            + "Please try again.");
    MyMessage.setTitle("Error Adding Completed Course");
    MyMessage.setVisible(true);

}
```

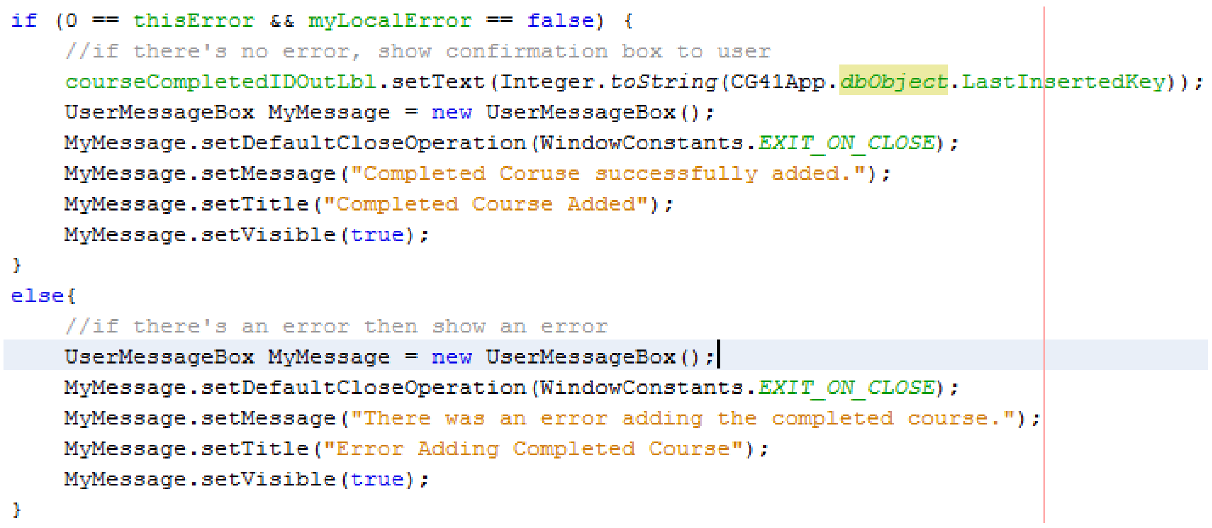
Above is a check to make sure the fields have all been filled out before the program continues to add it to the database. If there’s an issue, display the user an error message which explains where to fix the problem.

```
else { //if all is okay
    compGov = compGovCombox.getSelectedItem().toString(); //get governor
    compGovIDArray = compGov.split(",", 2); //split string
    govID = Integer.parseInt(compGovIDArray[0]); //get the gov ID
    //same as with the gov but with the course
    completedCourse = compCourseCombox.getSelectedItem().toString();
    compCourseIDArray = completedCourse.split(",", 2);
    courseID = Integer.parseInt(compCourseIDArray[0]);
}
```

If there’s no problem, we can continue. I can use a method called toString to convert the selected items in the combo boxes to strings to use later. Integer also has a similar function where I can parse Integers from strings to store in an Integer variable. I use an array here to split the contents of the selected item because I store both the ID and name in the combo box.

```
if(myLocalError == false){ //add the comp course if there's no error
    myLocalString = "insert into coursetaken ("
            + "GovID, CourseID, DateTaken)"
            + "values ('" + govID + "', "
            + "'" + courseID + "', "
            + "'" + courseCompDateString + "')";//concatonate string
    CG41App.dbObject.sqlString = myLocalString; //set string
    thisError = CG41App.dbObject.insertRecord(); //execute query
}
```

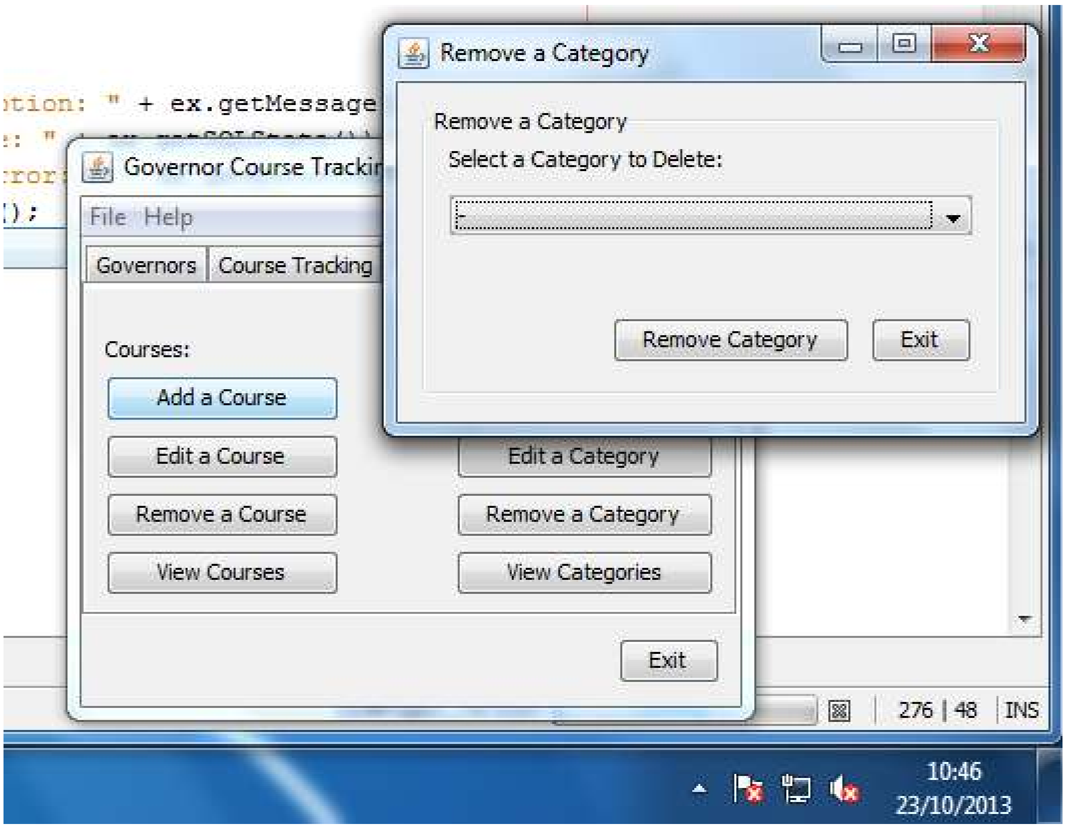
If there’s no error, we can add to the database. As I’m using MySQL, I have to create the query string then set it in the database object. I can then execute this string. Then I can receive an error code to be used later on.



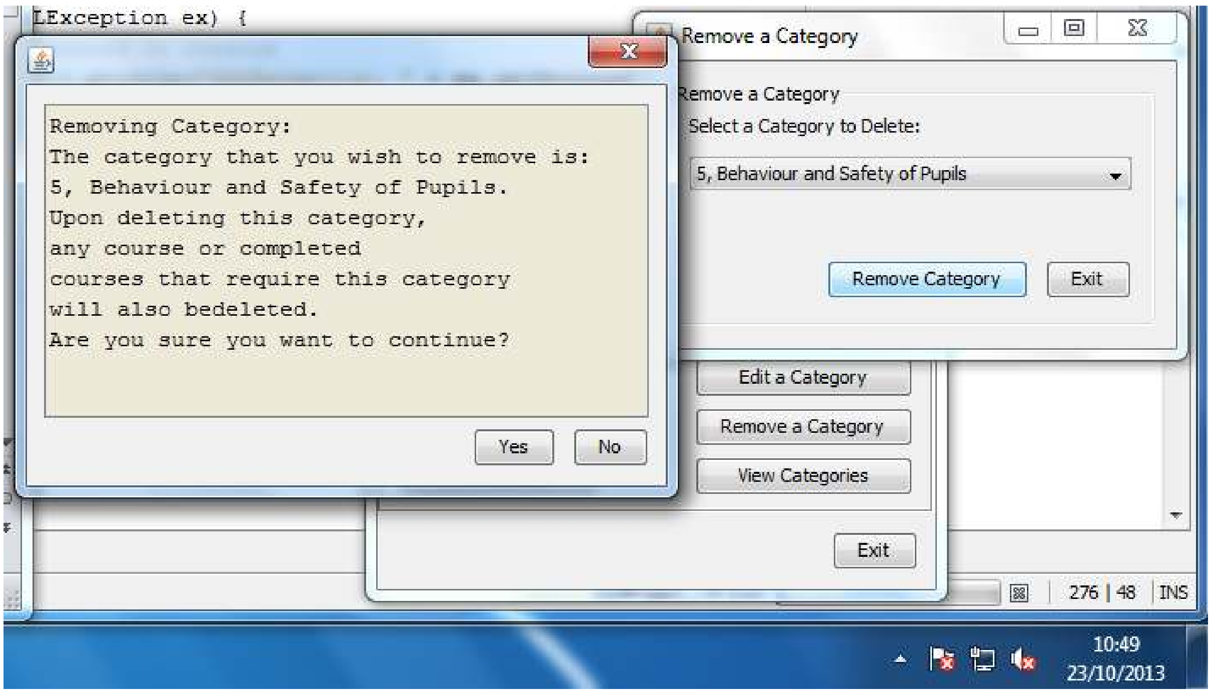
```
if (0 == thisError && myLocalError == false) {
    //if there's no error, show confirmation box to user
    courseCompletedIDOutLbl.setText(Integer.toString(CG41App.dbObject.LastInsertedKey));
    UserMessageBox MyMessage = new UserMessageBox();
    MyMessage.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    MyMessage.setMessage("Completed Coruse successfully added.");
    MyMessage.setTitle("Completed Course Added");
    MyMessage.setVisible(true);
}
else{
    //if there's an error then show an error
    UserMessageBox MyMessage = new UserMessageBox();
    MyMessage.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    MyMessage.setMessage("There was an error adding the completed course.");
    MyMessage.setTitle("Error Adding Completed Course");
    MyMessage.setVisible(true);
}
```

If there’s no error anywhere, then set the Completed Course ID and show a success message. Else, show an error message.

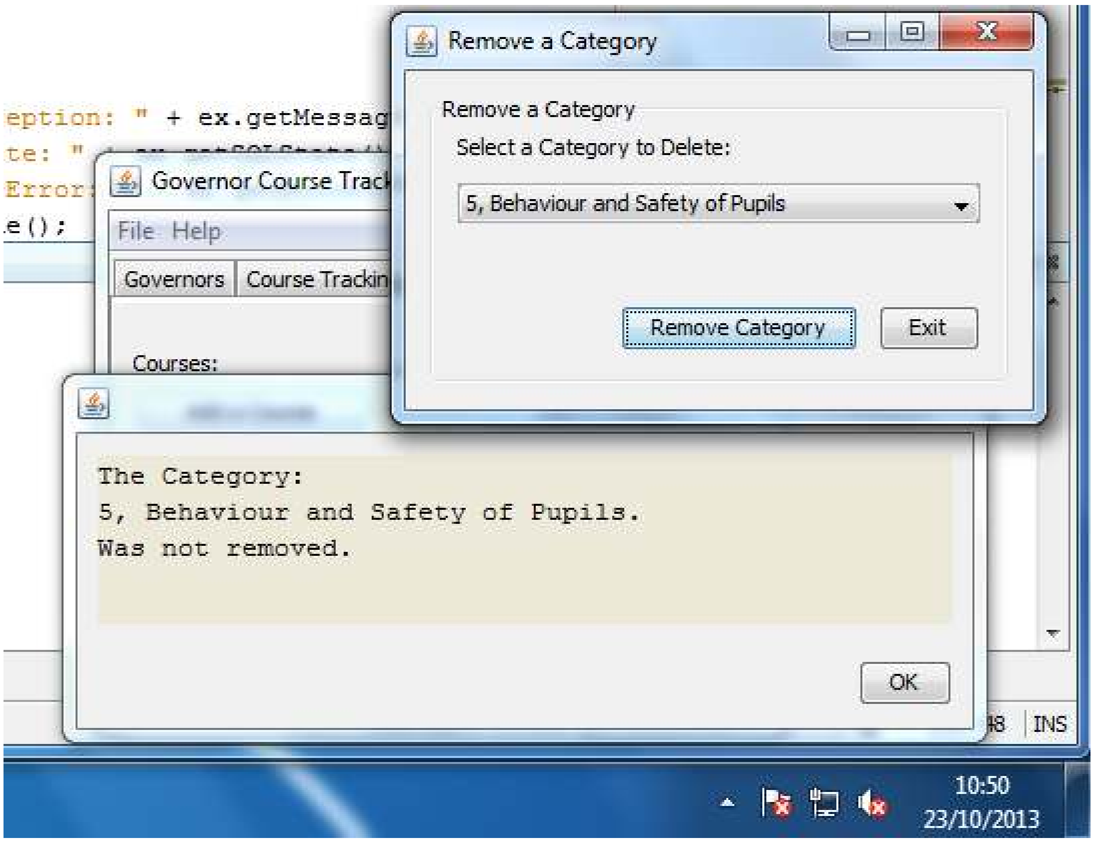
Removing



For an example on removing items, I will use the class that handles removing categories. Above is the form the user sees when they want to remove a category. They select a category from a combo box and then press the button to remove it. After that, a confirmation box pops up to ensure they want to remove it.


```
Removing Category:
The category that you wish to remove is:
5, Behaviour and Safety of Pupils.
Upon deleting this category,
any course or completed
courses that require this category
will also bedeleted.
Are you sure you want to continue?
```

Dependant on whether they say yes or no, another box pops to tell the user of their decision and the action it had. The pop up blocks interaction with the previous window to ensure that they can't change the selection and remove a different category that originally selected. This is a feature of the Swing Library from Java. After pressing the no button, this was the result:



```
The Category:
5, Behaviour and Safety of Pupils.
Was not removed.
```

The code that loads the categories into the combo box is similar to the code previously shown for getting the governors into the combo box.

When the remove button is pressed, a method executes which begins the process of removing the selected item.

```
* Removes the selected governor from the database on press of button
*/
rivate void removeCategoryBtnActionPerformed(java.awt.event.ActionEvent evt) {
    myLocalString = ""; //reset used variables
    myCat = "";
    thisError = 0;
    removeCategory = false;

    myCat = categoryCombox.getSelectedItem().toString(); //get category info
    if (myCat.equals("-")) { //check to make sure the user has selected a category
        removeCategory = false;
        //show an error to the user
        UserMessageBox MyMessage = new UserMessageBox();
        MyMessage.setMessage("Please select a category to remove.");
        MyMessage.setVisible(true);
```

The variables are assigned blank values. Then the selected item from the combo box is retrieved. If the category string equals a dash (the initial item in the combo box) an error message box pops up for the user.

```
} else { //if there's no error
    UserConfirmationBox userConfirm = new UserConfirmationBox(this, true);
    //make sure the user knows what they're removing
    userConfirm.setMessage("Removing Category:\n"
            + "The category that you wish to remove is:\n"
            + myCat
            + ".\nUpon deleting this category,\nany course or completed"
            + "\ncourses that require this category\nwill also be"
            + "deleted.\nAre you sure you want to continue?");
    userConfirm.setVisible(true);
    removeCategory = userConfirm.removeConfirmation; //take the result fron
    userConfirm.dispose(); //make sure we close it
```

Next we have to make sure they want to remove the selected category. The above code creates a new confirmation box which will allow me to ask the user if they want to remove the selected category. An example of the confirmation window can be seen a few images above.

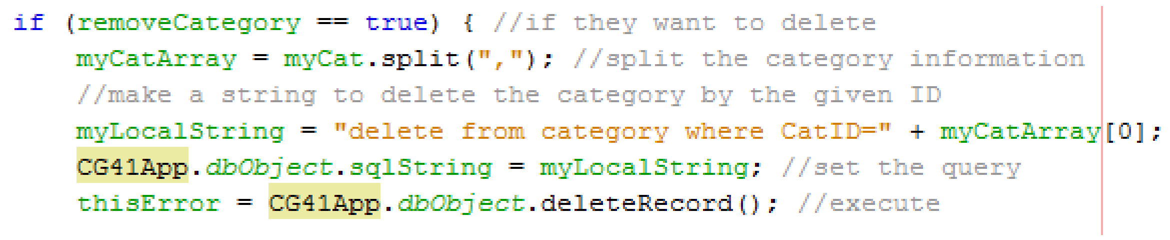
The user confirmation box class only has two methods and one global variable. Dependant on what button is pressed the state of the Boolean variable will change:

```
 * When the yes button is pressed, this method will set the public
 * remove confirmation variable to true. It will also hide this window and
 * allow the you to retrieve the variable before closing this object
 */
private void yesBtnActionPerformed(java.awt.event.ActionEvent evt) {
    removeConfirmation = true;
    setVisible(false);
}

/**
 * When the no button is pressed, this method will set the public
 * remove confirmation variable to false. It will also hide this window and
 * allow the you to retrieve the variable before closing this object
 */
private void noBtnActionPerformed(java.awt.event.ActionEvent evt) {
    removeConfirmation = false;
    setVisible(false);
}
```

The changed variable is retrieved from the confirmation box and its logic state is used to determine how the rest of the code runs. As this class doesn’t close on its own when a button is pressed, make we make sure we close the object.

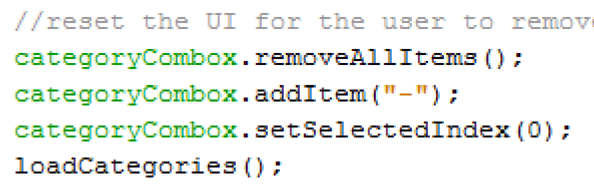
```
if (removeCategory == false) { //if they don't want to delete
    UserMessageBox MyMessage = new UserMessageBox();
    MyMessage.setMessage("The Category:\n"
            + myCat
            + ".\nWas not removed.");
    MyMessage.setVisible(true);
}
```

If the user selects no, we don’t need to remove the category. Tell the user this has happened.



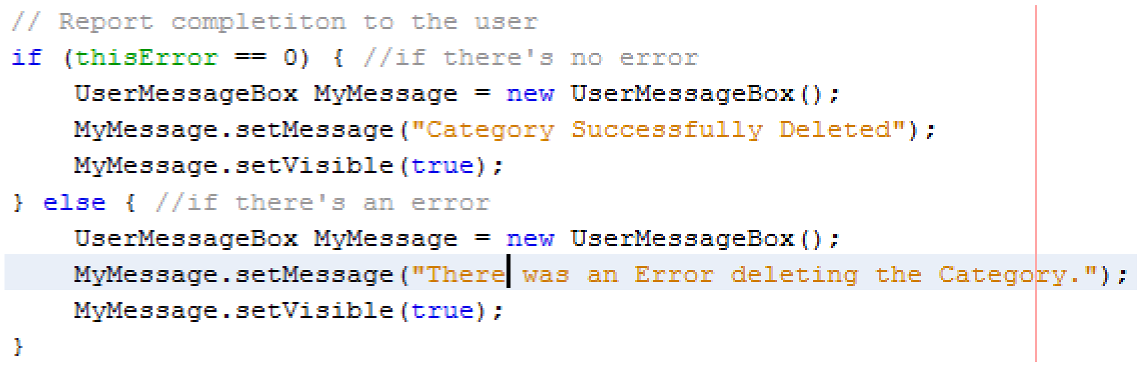
```
if (removeCategory == true) { //if they want to delete
    myCatArray = myCat.split(","); //split the category information
    //make a string to delete the category by the given ID
    myLocalString = "delete from category where CatID=" + myCatArray[0];
    CG41App.dbObject.sqlString = myLocalString; //set the query
    thisError = CG41App.dbObject.deleteRecord(); //execute
```

If they want to continue removing the category, split the category into an array and get the ID. Remove the category by the ID. As I have set my database to Cascade records when they are modified, any other records that are related to the record removed will also be removed. This is a feature available in MySQL that I have used to my advantage.



```
//reset the UI for the user to remove another category
categoryCombox.removeAllItems();
categoryCombox.addItem("-");
categoryCombox.setSelectedIndex(0);
loadCategories();
```

Above we have code that executes after we've removed the category. It will reset the Swing components and recall the loadCategories method. This will retrieve all the remaining categories left in the database after we have removed one.

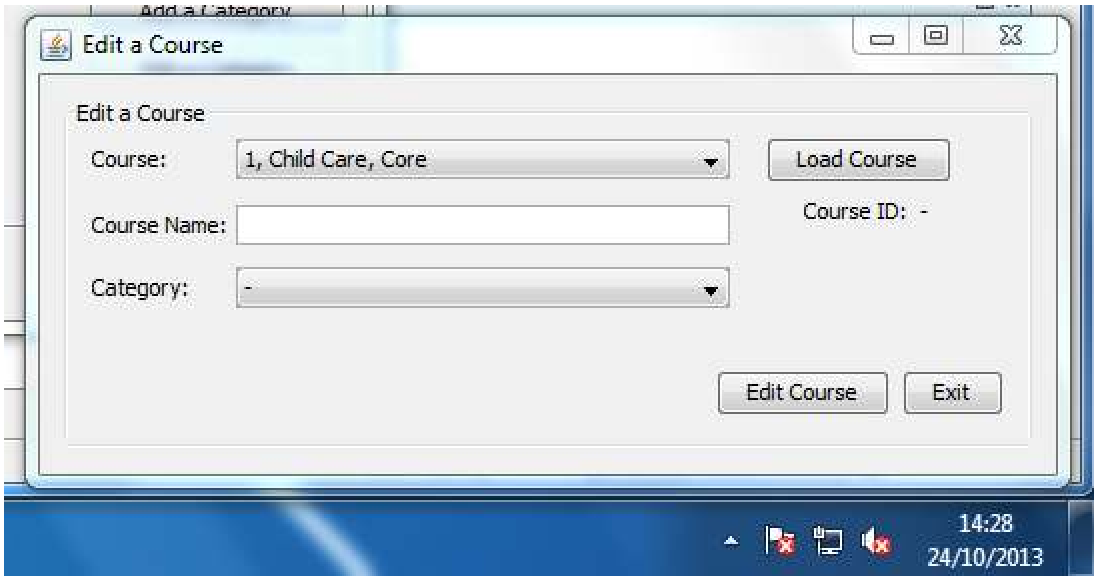


```
// Report completiton to the user
if (thisError == 0) { //if there's no error
    UserMessageBox MyMessage = new UserMessageBox();
    MyMessage.setMessage("Category Successfully Deleted");
    MyMessage.setVisible(true);
} else { //if there's an error
    UserMessageBox MyMessage = new UserMessageBox();
    MyMessage.setMessage("There was an Error deleting the Category.");
    MyMessage.setVisible(true);
}
```

Similar to other parts of my code, I have a simple IF and ELSE statement to handle showing the completion and failure of removing the record.

Editing

For an example on editing, I will be using the Edit a Course window and class.



Edit a Course

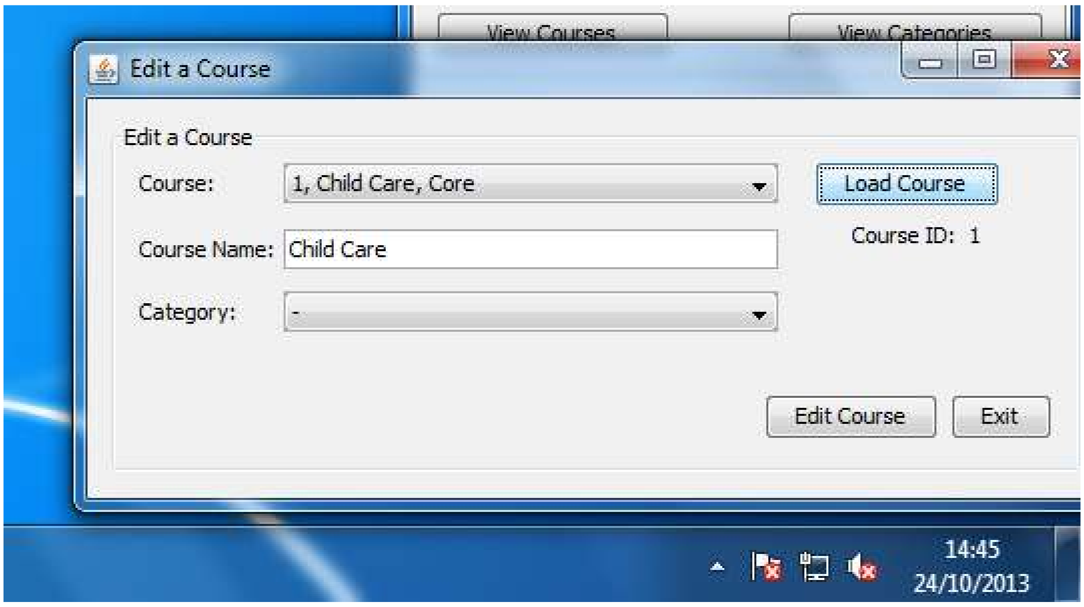
Course: 1, Child Care, Core

Course Name:

Category: -

Course ID: -

Above is an image of what the form looks like before a course is loaded to be edited. The first combo box holds the ID, name and category of the course.



Edit a Course

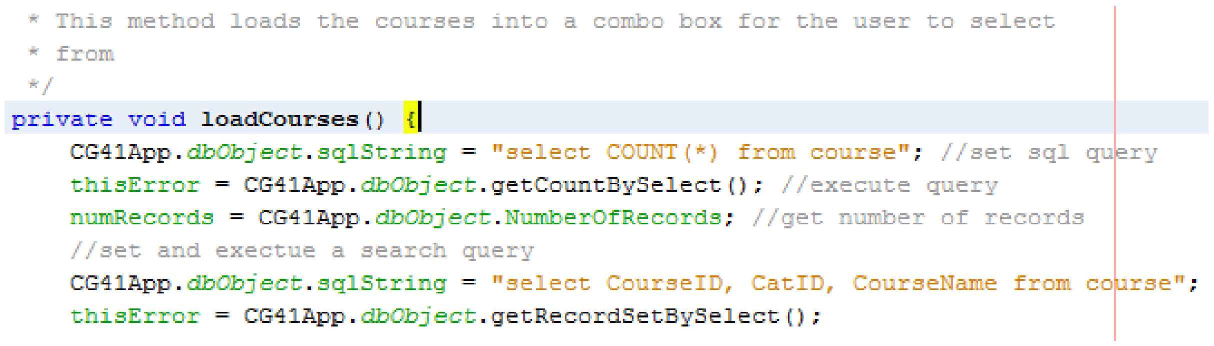
Course: 1, Child Care, Core

Course Name: Child Care

Category: -

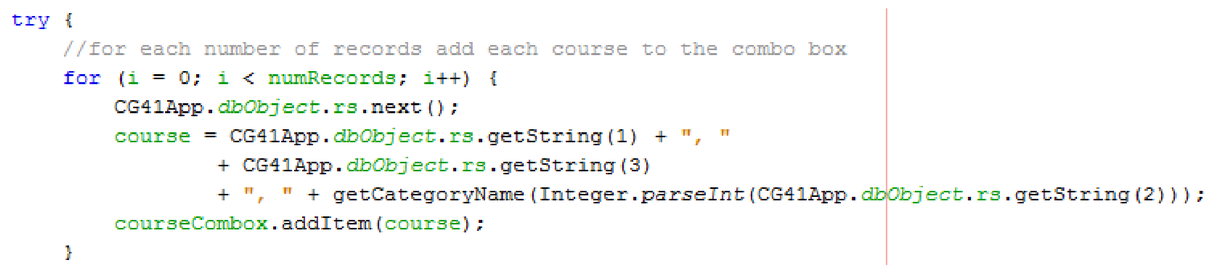
Course ID: 1

After the user has pressed Load Course, the ID is moved to the ID label, the name is moved to the name field and the categories are loaded.



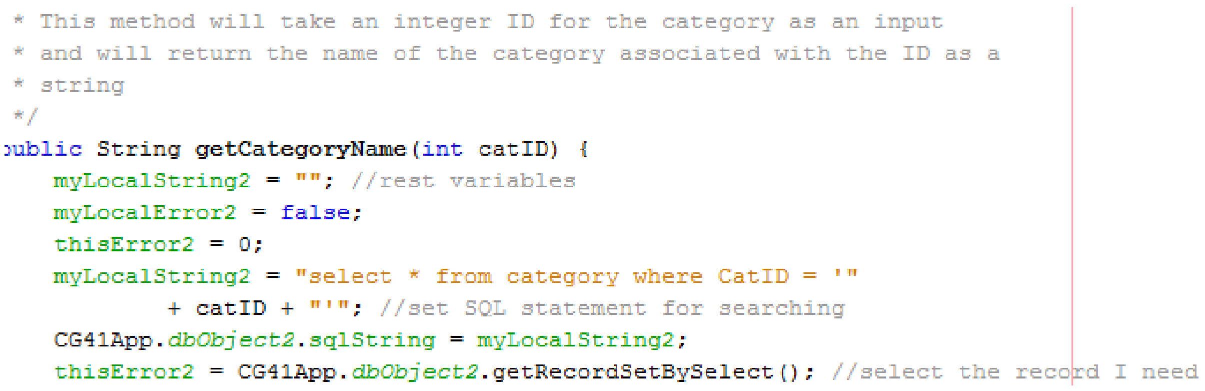
```
 * This method loads the courses into a combo box for the user to select
 * from
 */
private void loadCourses() {
    CG41App.dbObject.sqlString = "select COUNT(*) from course"; //set sql query
    thisError = CG41App.dbObject.getCountBySelect(); //execute query
    numRecords = CG41App.dbObject.NumberOfRecords; //get number of records
    //set and exectue a search query
    CG41App.dbObject.sqlString = "select CourseID, CatID, CourseName from course";
    thisError = CG41App.dbObject.getRecordSetBySelect();
```

Courses are loaded into the combo box a similar way other combo boxes are populated. However, this combo box has an exception.



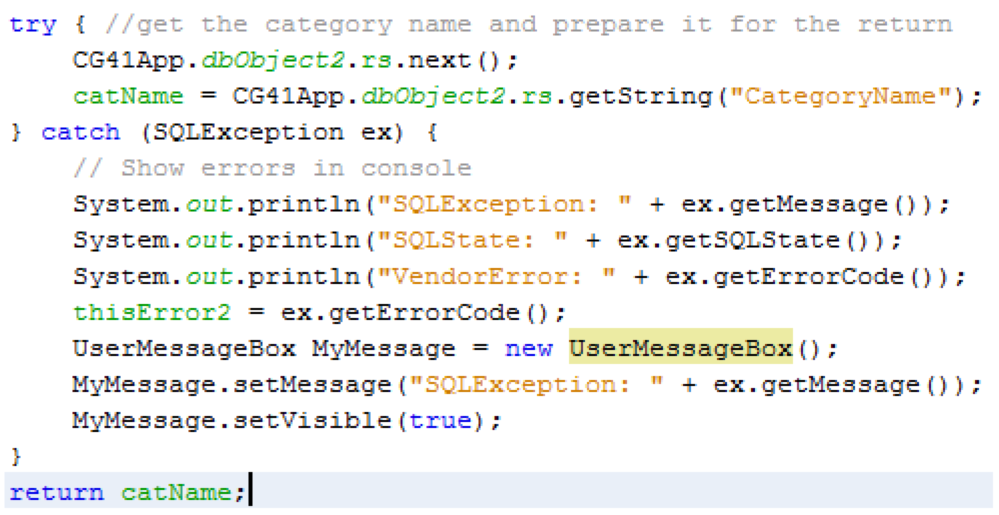
```
try {
    //for each number of records add each course to the combo box
    for (i = 0; i < numRecords; i++) {
        CG41App.dbObject.rs.next();
        course = CG41App.dbObject.rs.getString(1) + ", "
                + CG41App.dbObject.rs.getString(3)
                + ", " + getCategoryName(Integer.parseInt(CG41App.dbObject.rs.getString(2)));
        courseCombox.addItem(course);
    }
```

I have written a method to obtain the category name to go into the combo box as well. The reason I had to do this is because the table “course” only has the category ID.



```
 * This method will take an integer ID for the category as an input
 * and will return the name of the category associated with the ID as a
 * string
 */
ublic String getCategoryName(int catID) {
   myLocalString2 = ""; //rest variables
   myLocalError2 = false;
   thisError2 = 0;
   myLocalString2 = "select * from category where CatID = '"
           + catID + "'"; //set SQL statement for searching
   CG41App.dbObject2.sqlString = myLocalString2;
   thisError2 = CG41App.dbObject2.getRecordSetBySelect(); //select the record I need
```

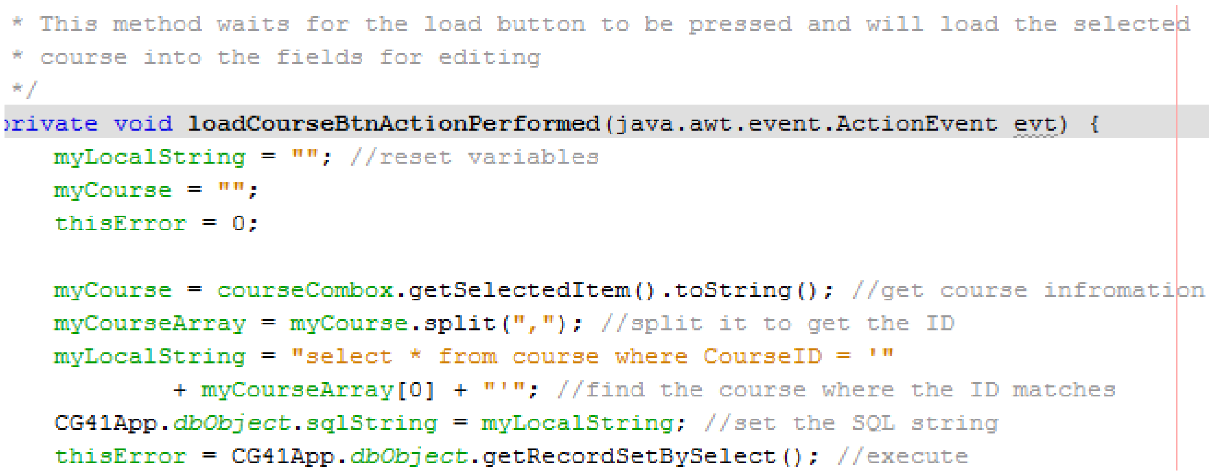
This method will take a category ID and return a category name. Above we see the code that selects the category record where the category ID matches the records in the database.



```
try { //get the category name and prepare it for the return
    CG41App.dbObject2.rs.next();
    catName = CG41App.dbObject2.rs.getString("CategoryName");
} catch (SQLException ex) {
    // Show errors in console
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
    thisError2 = ex.getErrorCode();
    UserMessageBox MyMessage = new UserMessageBox();
    MyMessage.setMessage("SQLException: " + ex.getMessage());
    MyMessage.setVisible(true);
}
return catName;
```

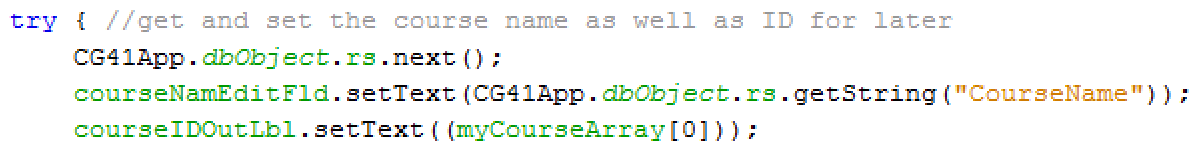
On the previous page we have the code that takes the category name out of the record set and stores it in a variable. The catch statement catches any errors in the try statement and as before displays them. The category name is then returned.

After the combo box has been populated, we need to wait until the user presses the Load Course button to load the selected course.



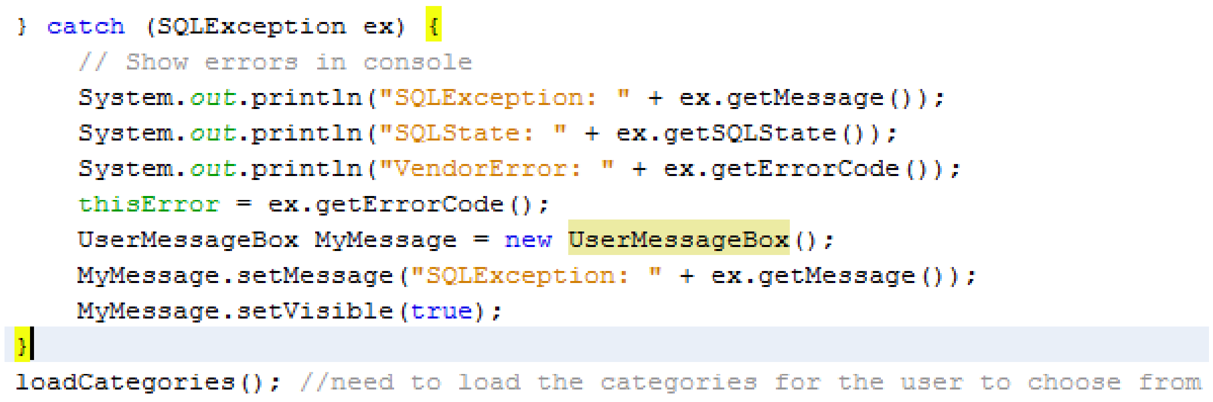
```
 * This method waits for the load button to be pressed and will load the selected
 * course into the fields for editing
 */
rivate void loadCourseBtnActionPerformed(java.awt.event.ActionEvent evt) {
    myLocalString = ""; //reset variables
    myCourse = "";
    thisError = 0;

    myCourse = courseCombox.getSelectedItem().toString(); //get course infromation
    myCourseArray = myCourse.split(","); //split it to get the ID
    myLocalString = "select * from course where CourseID = '"
            + myCourseArray[0] + "'"; //find the course where the ID matches
    CG41App.dbObject.sqlString = myLocalString; //set the SQL string
    thisError = CG41App.dbObject.getRecordSetBySelect(); //execute
```

The above code shows what happens when the user presses the button. A query is made to select the record where the course ID matches the course ID of the records in the database.



```
try { //get and set the course name as well as ID for later
    CG41App.dbObject.rs.next();
    courseNamEditFld.setText(CG41App.dbObject.rs.getString("CourseName"));
    courseIDOutLbl.setText((myCourseArray[0]));
```

Then we have to try and get the course name and course ID out of the Record Set in the dbObject.



```
} catch (SQLException ex) {
    // Show errors in console
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
    thisError = ex.getErrorCode();
    UserMessageBox MyMessage = new UserMessageBox();
    MyMessage.setMessage("SQLException: " + ex.getMessage());
    MyMessage.setVisible(true);
}
loadCategories(); //need to load the categories for the user to choose from
```

Then we catch any errors and print them. Finally we need to load the categories into the combo box. This is done the same way any other combo box in this program is populated.

After loading the course the user is free to edit the course and the category the course belongs to in the form window. Once completed editing the course, the user then presses the “Edit Course” button. http://gyazo.com/3db47c335069a7cfae7fd5302f442d49.png

```
  * This method will wait for the edit button to be pressed and then
  * will collect all the information needed for the edit
  */
vate void editCourseBtnActionPerformed(java.awt.event.ActionEvent evt) {
 myLocalError = false; //rest the variables
 thisError = 0;
 myLocalString = "";

 // Check if all fields are filled in.  If not, then print an error
 // message and don't do any more processing
 if (courseNamEditFld.getText().equals("")) {
     myLocalError = true;

     UserMessageBox MyMessage = new UserMessageBox();
     MyMessage.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
     MyMessage.setMessage("Please Enter All Fields.");
     MyMessage.setTitle("Error Editing Course");
     MyMessage.setVisible(true);
 }
```

First we have to make sure that the user hasn't just left the Course name field blank. If they have, show an error and set error to true to prevent editing the course.

```
 if (catEditCombox.getSelectedItem().toString().equals("-")) {
     myLocalError = true;

     UserMessageBox MyMessage = new UserMessageBox();
     MyMessage.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
     MyMessage.setMessage("Please Select a Category.");
     MyMessage.setTitle("Error Editing Course");
     MyMessage.setVisible(true);
 }
```

Next we need to see if the user has selected a category to assign the course. If there's no category, set error to true and display an error message.

```
 errorMessage = "Please check the following: \n\n";
 if (!myLocalError) {
     //validation of course name
     if (!CG41App.validateObject.validateCourseName(courseNamEditFld.getText())) {
         myLocalError = true;
         errorMessage = errorMessage + "The course name should begin with"
                 + "a letter and be followed by up to 20 letters, numbers"
                 + "apostophes, spaces and dashes.";
     }
     if (myLocalError) {
     //One of the validation checks failed
     UserMessageBox MyMessage = new UserMessageBox();
     MyMessage.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
     MyMessage.setMessage(errorMessage);
     MyMessage.setTitle("Error Editing Course");
     MyMessage.setVisible(true);
     }
 }
```

Next we need to continue checking the validation of the course name. The validation technique is the same here but with a different pattern.

```
 * This method requires a string input for the course name and will return a
 * boolean value as to whether or not it matches the validation pattern
 */
public boolean validateCourseName(String inCourseName) {
    return inCourseName.matches("([a-zA-z]{1,2})([a-zA-Z0-9 '-]{1,20})");
}
```

For more information about my validation techniques, see the validation section of this document.

If the validation fails, create an error message to display to the user. Then proceed to display this error message. Also set the error variable to true to prevent the code from continuing.

```
 if (!myLocalError) { //this will collect the information and amend the course
     myCat = catEditCombox.getSelectedItem().toString();
     myCategoryArray = myCat.split(","); //get category name and split it
     catID = Integer.parseInt(myCategoryArray[0]); //get category ID
     myLocalString = "update course set " + "CourseName= '" //create string
             + courseNamEditFld.getText().toString() + "', CatID = '" + catID
             + "' where CourseID= " + courseIDOutLbl.getText();
     CG41App.dbObject.sqlString = myLocalString; //set SQL string
     thisError = CG41App.dbObject.insertRecord();//execute query
 }
```

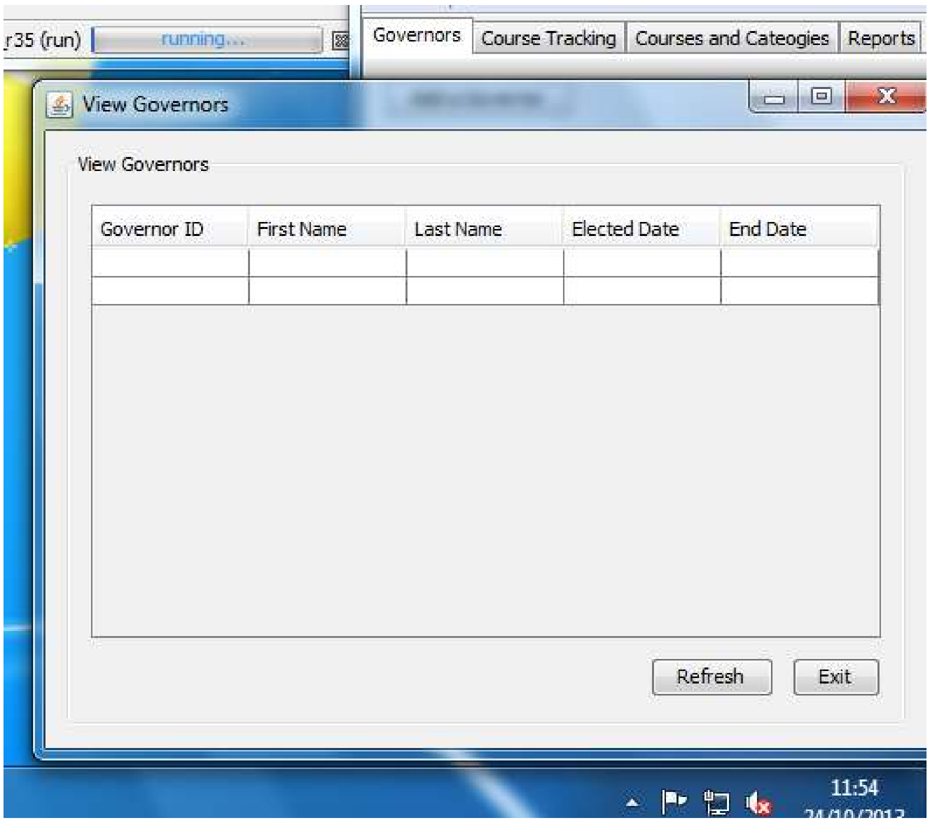
Here we have the code for actually amending the record. Each of the new parts of the record are collected and concatenated into one string and made into a query. This query is then executed. I have utilised several key functions of the language here to pull selected items from combo boxes or to get information from a text field and turn it into a string for processing.

```
if (0 == thisError && !myLocalError) {
    //if edit is successful
    UserMessageBox MyMessage = new UserMessageBox();
    MyMessage.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    MyMessage.setMessage("Course edited successfully");
    MyMessage.setTitle("Edit Successful");
    MyMessage.setVisible(true);
}
```

Again this is some error checking to make sure all went well and to display the success to the user.

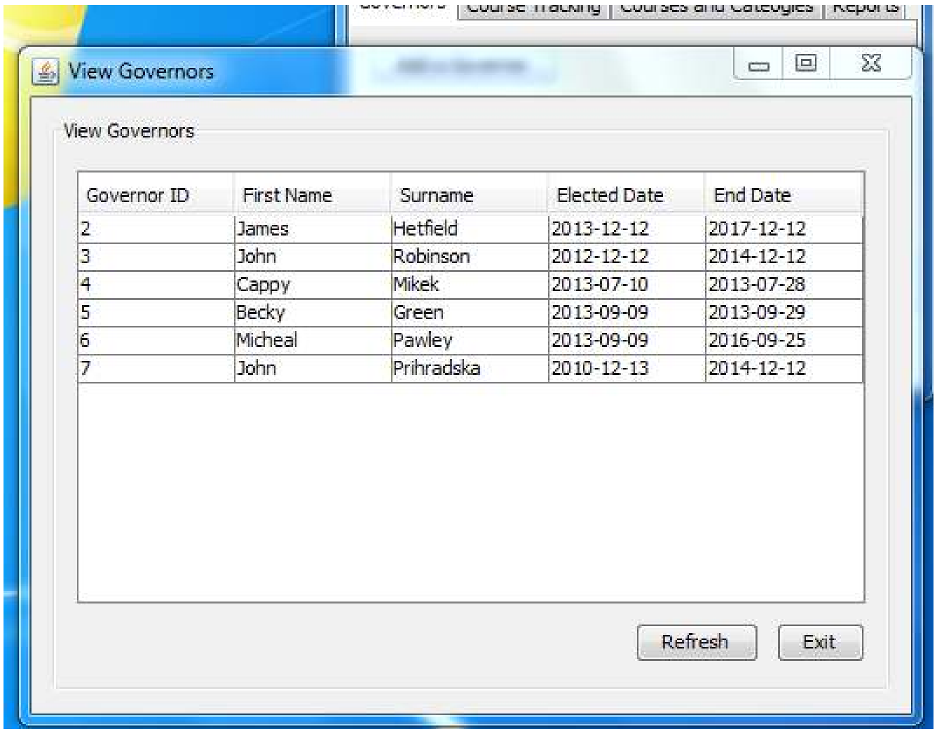
Viewing

Above we have a before shot of what one of the viewing forms looks like. In this example I have used the View Governors class to demonstrate how I allow the user to view records.



Governor ID	First Name	Last Name	Elected Date	End Date

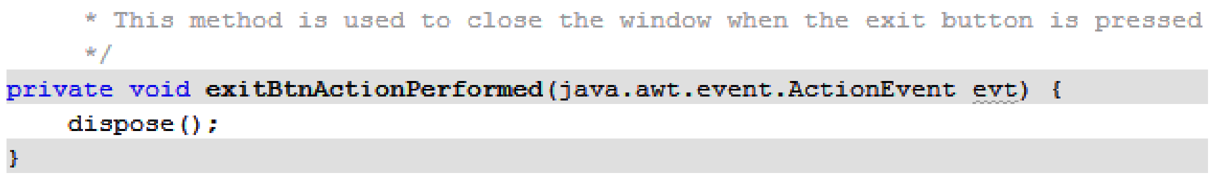
Below we have the after shot for when the user has pressed the Refresh button.



Governor ID	First Name	Surname	Elected Date	End Date
2	James	Hetfield	2013-12-12	2017-12-12
3	John	Robinson	2012-12-12	2014-12-12
4	Cappy	Mikek	2013-07-10	2013-07-28
5	Becky	Green	2013-09-09	2013-09-29
6	Micheal	Pawley	2013-09-09	2016-09-25
7	John	Prihradska	2010-12-13	2014-12-12

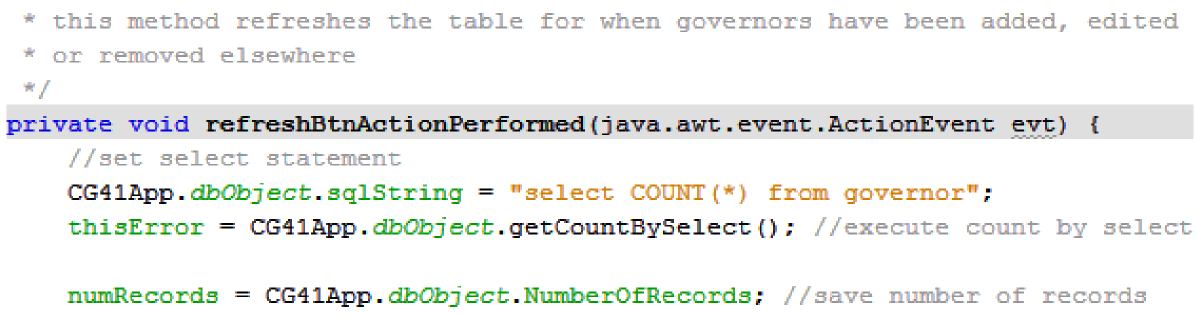
Here we see a table which splits up the information with sensibly named columns.

The only real piece to this class is the refresh button. Below is a quick look at the code for the exit button. This button appears on every window and allows the user to close down the window being used without closing down the whole program.



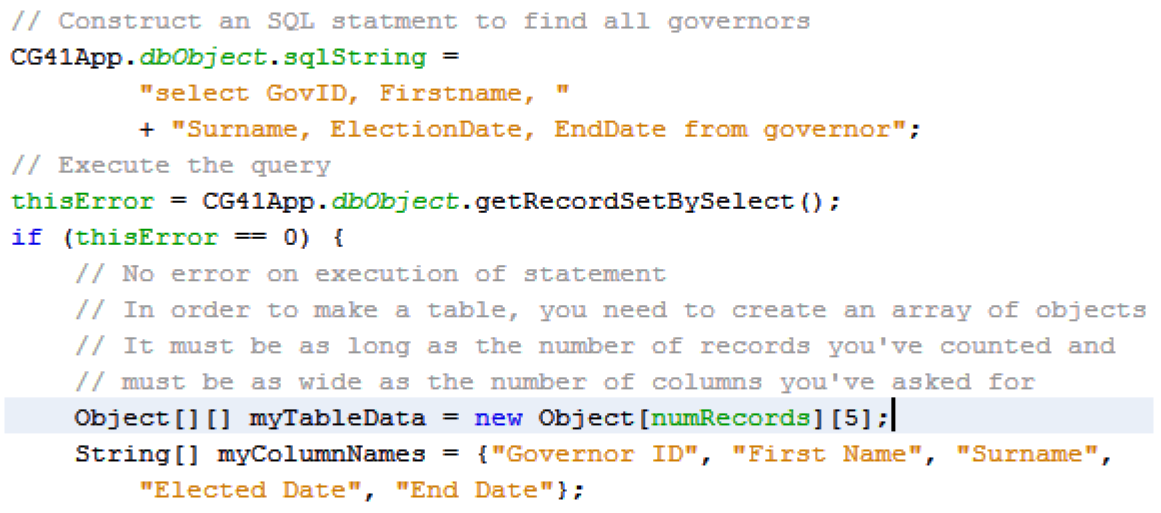
```
     * This method is used to close the window when the exit button is pressed
     */
private void exitBtnActionPerformed(java.awt.event.ActionEvent evt) {
    dispose();
}
```

Otherwise, we have the code for the refresh button:



```
   * this method refreshes the table for when governors have been added, edited
   * or removed elsewhere
   */
 private void refreshBtnActionPerformed(java.awt.event.ActionEvent evt) {
     //set select statement
     CG41App.dbObject.sqlString = "select COUNT(*) from governor";
     thisError = CG41App.dbObject.getCountBySelect(); //execute count by select

     numRecords = CG41App.dbObject.NumberOfRecords; //save number of records
```

The first part of this code sets an SQL statement to count the amount of governor records. This is set in the dbObject then executed and the result in stored.



```
// Construct an SQL statment to find all governors
CG41App.dbObject.sqlString =
        "select GovID, Firstname, "
        + "Surname, ElectionDate, EndDate from governor";
// Execute the query
thisError = CG41App.dbObject.getRecordSetBySelect();
if (thisError == 0) {
    // No error on execution of statement
    // In order to make a table, you need to create an array of objects
    // It must be as long as the number of records you've counted and
    // must be as wide as the number of columns you've asked for
    Object[][] myTableData = new Object[numRecords][5];
    String[] myColumnNames = {"Governor ID", "First Name", "Surname",
        "Elected Date", "End Date"};
```

Next we need to get all the records from the database. So we set a query to get all the records. Then we need to set up the table but only if there's no error from accessing the database. I've used an array for type Object to store my records in. Then I need another array to set up the column names.

```
try {
    int row = 0;
    while (CG41App.dbObject.rs.next()) // For each record
    {
        //get each part of the governor from the Record Set
        govID = CG41App.dbObject.rs.getString("GovID");
        firstname = CG41App.dbObject.rs.getString("Firstname");
        surname = CG41App.dbObject.rs.getString("Surname");
        elecDate = CG41App.dbObject.rs.getDate("ElectionDate");
        endDate = CG41App.dbObject.rs.getDate("EndDate");
        myTableData[row][0] = govID;
        myTableData[row][1] = firstname;
        myTableData[row][2] = surname;
        myTableData[row][3] = String.valueOf(elecDate);
        myTableData[row][4] = String.valueOf(endDate);
        row = row + 1;            // Next row in table
    }
```

Here I've used a while loop to ensure I put all the records into the table. Each part of the governor record is taken from the record set in the dbObject. Then row by row the information is added to the table.

```
    // Have the table data, so make a new table with it
    JTable jtblAllGovernors = new JTable(myTableData, myColumnNames);
    jtblAllGovernors.setFillsViewportHeight(true);
    governorScrlPane.getViewport().add(jtblAllGovernors);
    governorScrlPane.repaint();
```

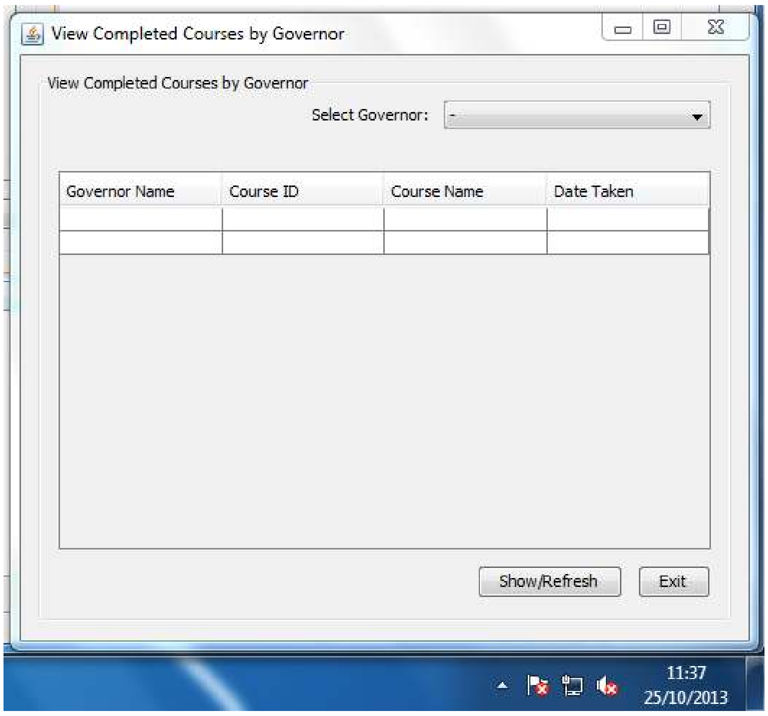
Now that we have all the information in one place, we need to display it. Make a new JTable and display it in the window.

```
} catch (SQLException ex) {
    // Show errors in console
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
    thisError = ex.getErrorCode();

    UserMessageBox MyMessage = new UserMessageBox();
    MyMessage.setMessage("SQLException: " + ex.getMessage());
    MyMessage.setVisible(true);
}
```

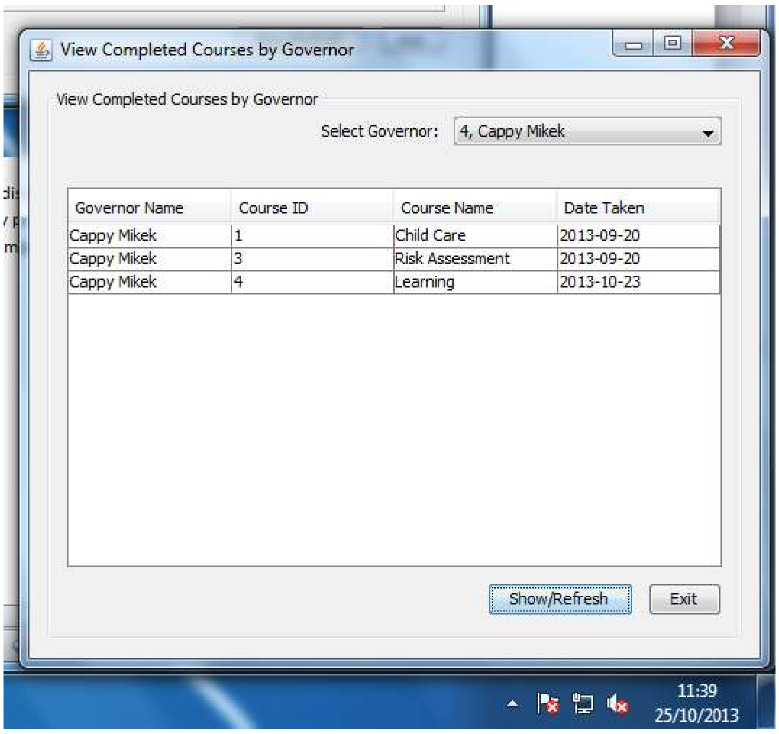
This is the catch statement for the try. It will catch any errors that happen in the try then print them to console and create a window with the error in it.

Viewing by Given Criteria

There are two ways to view the completed courses table in the program. One is by a given Course and one is by a given Governor.



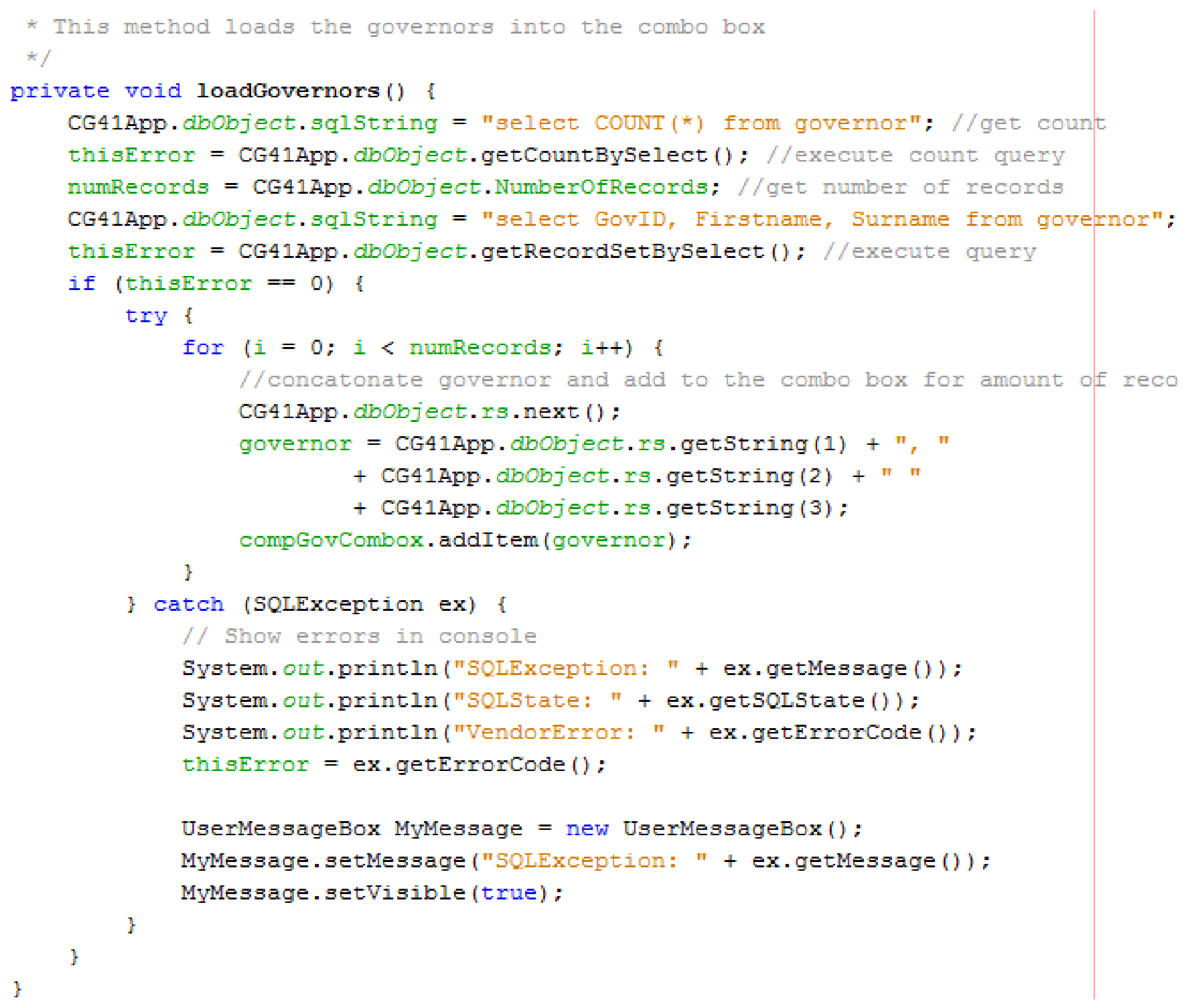
In this example, I will be displaying the Governor version of this form. At the top, the user selects a governor. After that, they press the "Show/Refresh" button. This will display all the courses that the selected governor has completed.


View Completed Courses by Governor
View Completed Courses by Governor
Show/Refresh
Exit
11:39
25/10/2013

Select Governor: 4, Cappy Mikek

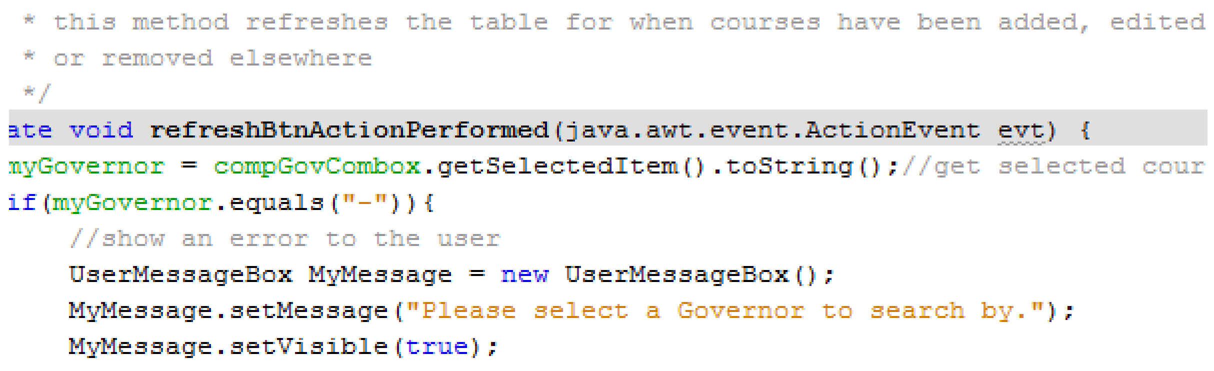
Governor Name	Course ID	Course Name	Date Taken
Cappy Mikek	1	Child Care	2013-09-20
Cappy Mikek	3	Risk Assessment	2013-09-20
Cappy Mikek	4	Learning	2013-10-23

Above, we can see a table of all the courses "Cappy Mikek" has completed.



```
 * This method loads the governors into the combo box
 */
private void loadGovernors() {
    CG41App.dbObject.sqlString = "select COUNT(*) from governor"; //get count
    thisError = CG41App.dbObject.getCountBySelect(); //execute count query
    numRecords = CG41App.dbObject.NumberOfRecords; //get number of records
    CG41App.dbObject.sqlString = "select GovID, Firstname, Surname from governor";
    thisError = CG41App.dbObject.getRecordSetBySelect(); //execute query
    if (thisError == 0) {
        try {
            for (i = 0; i < numRecords; i++) {
                //concatonate governor and add to the combo box for amount of reco
                CG41App.dbObject.rs.next();
                governor = CG41App.dbObject.rs.getString(1) + ", "
                        + CG41App.dbObject.rs.getString(2) + " "
                        + CG41App.dbObject.rs.getString(3);
                compGovCombox.addItem(governor);
            }
        } catch (SQLException ex) {
            // Show errors in console
            System.out.println("SQLException: " + ex.getMessage());
            System.out.println("SQLState: " + ex.getSQLState());
            System.out.println("VendorError: " + ex.getErrorCode());
            thisError = ex.getErrorCode();

            UserMessageBox MyMessage = new UserMessageBox();
            MyMessage.setMessage("SQLException: " + ex.getMessage());
            MyMessage.setVisible(true);
        }
    }
}
```

Above is the method I use to load the governors into the combo box. This is similar to other methods of loading records from the database into a combo box.



```
 * this method refreshes the table for when courses have been added, edited
 * or removed elsewhere
 */
ate void refreshBtnActionPerformed(java.awt.event.ActionEvent evt) {
myGovernor = compGovCombox.getSelectedItem().toString();//get selected cour
if(myGovernor.equals("-")){
    //show an error to the user
    UserMessageBox MyMessage = new UserMessageBox();
    MyMessage.setMessage("Please select a Governor to search by.");
    MyMessage.setVisible(true);
```

Here we have the beginning of the code for the refresh button. We want to make sure that a governor has been selected else we can’t continue. Show an error to the user if they haven’t selected a governor.

```
myGovernorArray = myGovernor.split(","); //get the ID out
myGovernorID = Integer.parseInt(myGovernorArray[0]); //store the ID

//count the amount of records where the selected ID is concerned
myLocalString = "select COUNT(*) from coursetaken where GovID = "
        + myGovernorID;
CG41App.dbObject.sqlString = myLocalString;
thisError = CG41App.dbObject.getCountBySelect();//execute

numRecords = CG41App.dbObject.NumberOfRecords; //save number of records
```

Next we need to get the Govenror ID so split it into an array and take the ID. Then select all the governors from the table where the ID matches the selected ID. This allows us to count the amount of records we're going to display.

```
// Construct an SQL statment to find all the information we need:
//The information that we're pulling here is the course name, datetaken,
//course ID, Governor ID and Governor First/Last names but only where the
//governor ID selected and coursetaken gocernor ID match.
//The joins allow us to pull the information from the three tables
myLocalString = "SELECT course.CourseName, coursetaken.DateTaken, "
        + "coursetaken.GovID, governor.Firstname, governor.Surname, "
        + "coursetaken.CourseID FROM (coursetaken "
        + "INNER JOIN course ON course.CourseID = coursetaken.CourseID) "
        + "INNER JOIN governor ON governor.GovID = coursetaken.GovID "
        + "WHERE (coursetaken.GovID = " + myGovernorID + ")";
CG41App.dbObject.sqlString = myLocalString;
```

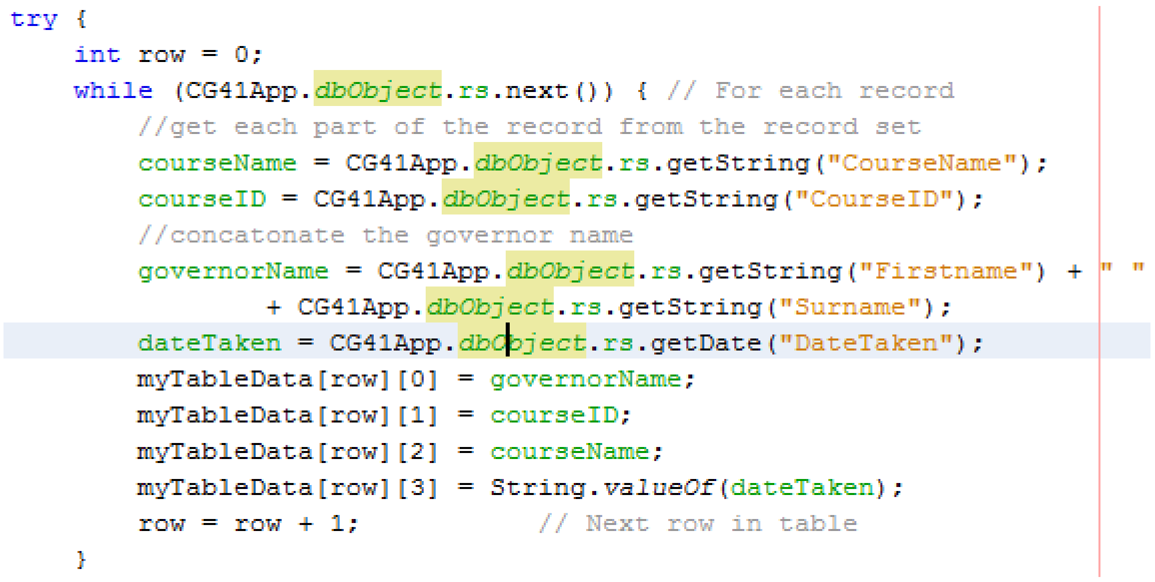
Next we have to select the information from the database that we want to display. We do this by using a JOIN statement from the MySQL query language. This allows me to declare how the tables are joined and what columns of information I want to retrieve. For instance, I want to select the information to display in the table but only where the Governor ID matches the selected one. This way I can display almost anything from the database without having to do more than one query.

```
// Execute the query
thisError = CG41App.dbObject.getRecordSetBySelect();
```

Then we just execute it.

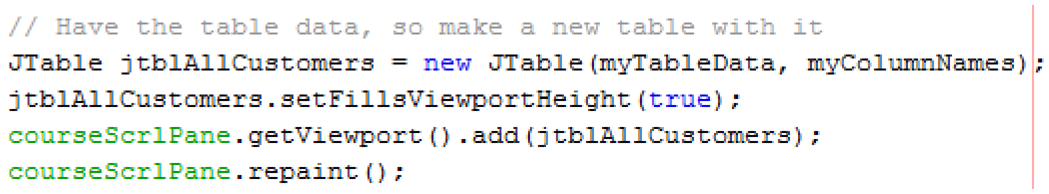
```
if (thisError == 0) {
    // No error on execution of statement
    // In order to make a table, you need to create an array of objects
    // It must be as long as the number of records you've counted and
    // must be as wide as the number of columns you've asked for
    Object[][] myTableData = new Object[numRecords][4];
    String[] myColumnNames = {"Governor Name", "Course ID",
        "Course Name", "Date Taken"};
```

This sets up the table just like the table in the viewing section of this document.



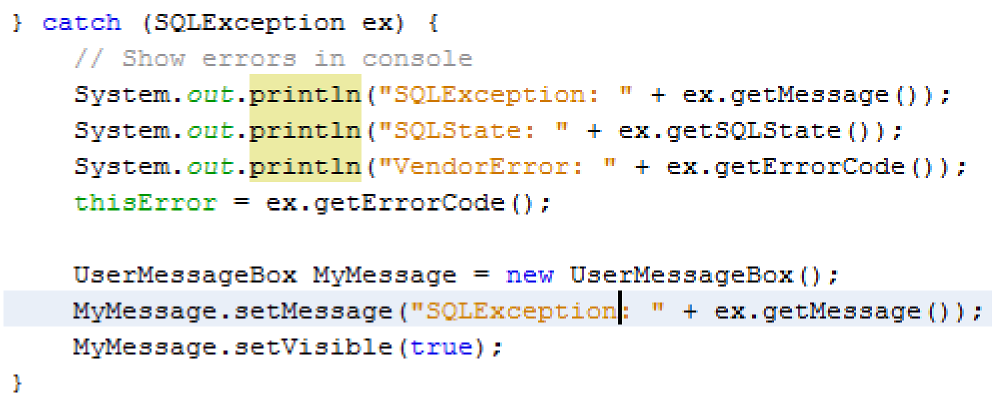
```
try {
    int row = 0;
    while (CG41App.dbObject.rs.next()) { // For each record
        //get each part of the record from the record set
        courseName = CG41App.dbObject.rs.getString("CourseName");
        courseID = CG41App.dbObject.rs.getString("CourseID");
        //concatonate the governor name
        governorName = CG41App.dbObject.rs.getString("Firstname") + " "
                + CG41App.dbObject.rs.getString("Surname");
        dateTaken = CG41App.dbObject.rs.getDate("DateTaken");
        myTableData[row][0] = governorName;
        myTableData[row][1] = courseID;
        myTableData[row][2] = courseName;
        myTableData[row][3] = String.valueOf(dateTaken);
        row = row + 1;             // Next row in table
    }
```

Populate the table ready for displaying.



```
// Have the table data, so make a new table with it
JTable jtblAllCustomers = new JTable(myTableData, myColumnNames);
jtblAllCustomers.setFillsViewportHeight(true);
courseScrlPane.getViewport().add(jtblAllCustomers);
courseScrlPane.repaint();
```

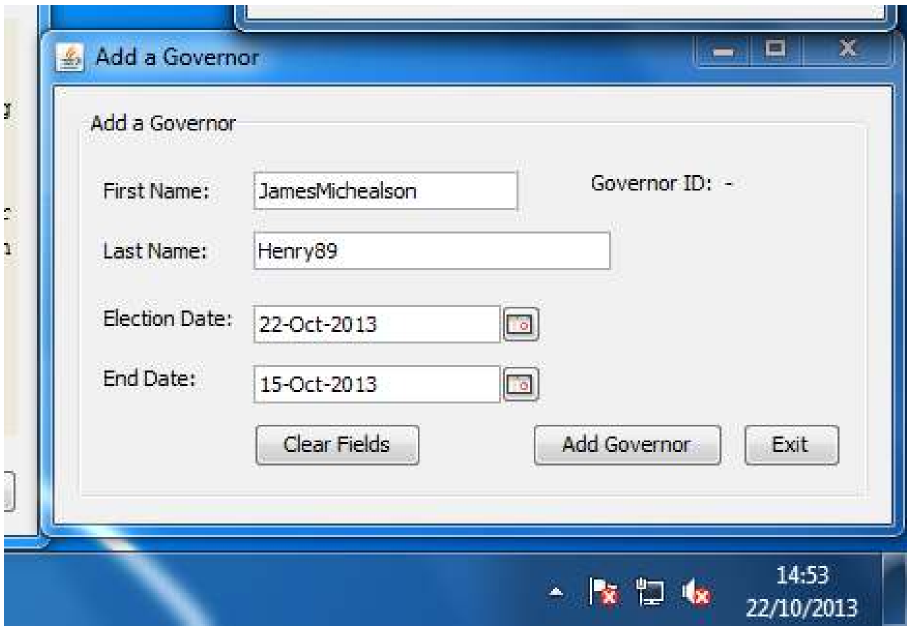
Display the table.



```
} catch (SQLException ex) {
    // Show errors in console
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
    thisError = ex.getErrorCode();

    UserMessageBox MyMessage = new UserMessageBox();
    MyMessage.setMessage("SQLException: " + ex.getMessage());
    MyMessage.setVisible(true);
}
```

Make sure we catch all the errors.

Validation


Add a Governor

Add a Governor

First Name: JamesMichealson

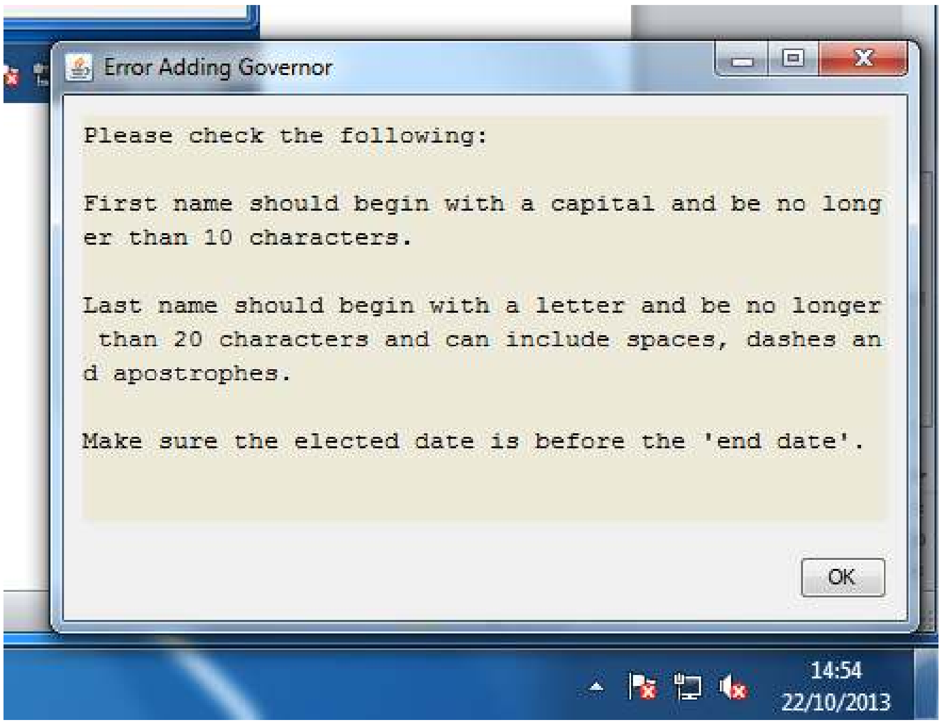
Governor ID: -

Last Name: Henry89

Election Date: 22-Oct-2013

End Date: 15-Oct-2013

Above we have an image of the “Add a Governor” form. This is the form that the user would fill out should they want to add a new governor. I have already filled out the forms in anticipation of triggering the validation checks and popping up an error message to show the user where they went wrong.


Error Adding Governor

```
Please check the following:

First name should begin with a capital and be no long
er than 10 characters.

Last name should begin with a letter and be no longer
 than 20 characters and can include spaces, dashes an
d apostrophes.

Make sure the elected date is before the 'end date'.
```

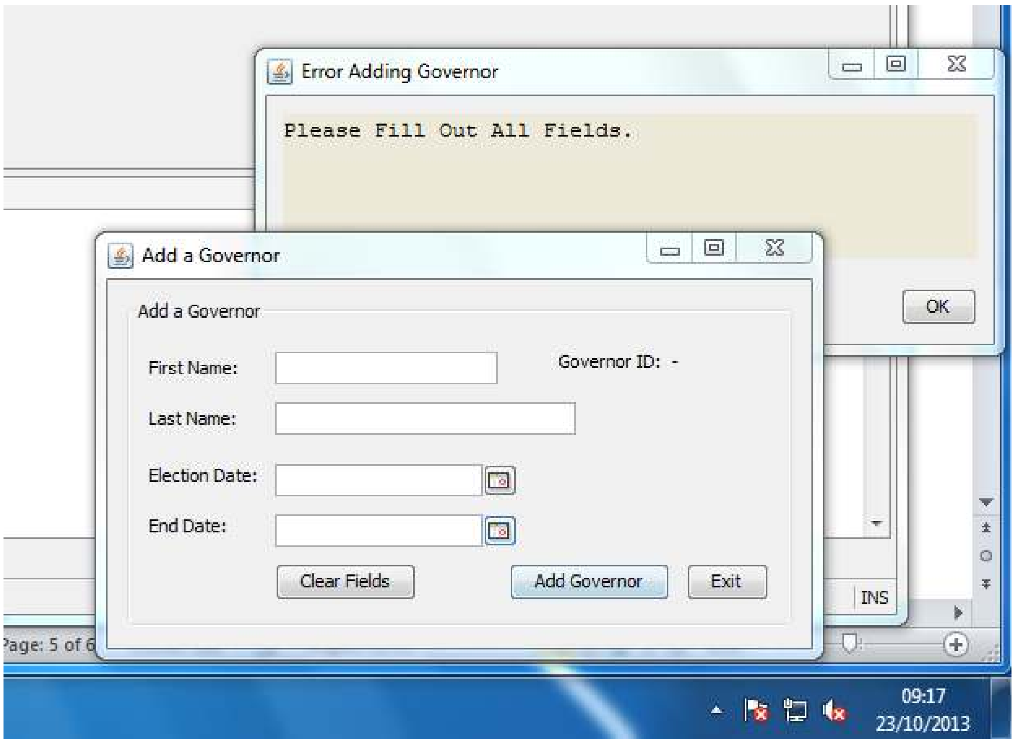
Above again, we can see that the first name, last name and the order of the dates failed to pass the validation. Below, we can the validation code that executes when the user presses the “Add Governor” button.

```
//Quick check to see if the fields have been filled in, else print an error
if (frstNamFld.getText().equals("")
        || lstNamFld.getText().equals("")
        || elecDateString.equals("null-null-null")
        || endDateString.equals("null-null-null")) {
    myLocalError = true;
}
if (myLocalError) {
    // One field must be blank, therefore display an error
    UserMessageBox MyMessage = new UserMessageBox();
    MyMessage.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    MyMessage.setMessage("Please Fill Out All Fields.");
    MyMessage.setTitle("Error Adding Governor");
    MyMessage.setVisible(true);
}
```

The first check is to make sure all the fields are filled in. The first IF statement checks the first name field and last name field to see if the user has entered anything into them. Then we have to make sure the user has actually selected two dates. Further up in the code, the dates are taken from the date boxes and converted to strings to be used later on in the add governor process. Each field to be checked is separated by the OR logic operator.

If there's an error with any one of the four fields being blank, then a Boolean value (myLocalError) is set to true to prevent the rest of the code from running. I'll explain how this works a bit more with the next section below.

The second IF statement above will only trigger if the Boolean value myLocalError is set to true, then a new user message box is created which tells the user that they need to fill out the entire form.


Error Adding Governor

Please Fill Out All Fields.

Add a Governor

Add a Governor

First Name:

Governor ID: -

Last Name:

Election Date:

End Date:

Above is an example of what happens when at least one of the fields is left blank. In this example, I left all the fields blank.

The next part of the validation is to ensure that the information entered into the fields is formatted correctly to meet my pre-defined patterns.

```
//Error checking for validation of each field where an error message will be
//created for the user's viewing for any errors
errorMessage = "Please check the following: \n\n";
if (!myLocalError) {
    //validation of first name
    if(!CG41App.validateObject.validateFirstName(frstNamFld.getText())){
        myLocalError = true;
        errorMessage = errorMessage + "First name should begin with a capital"
                + " and be no longer than 10 characters. \n\n";
    }
```

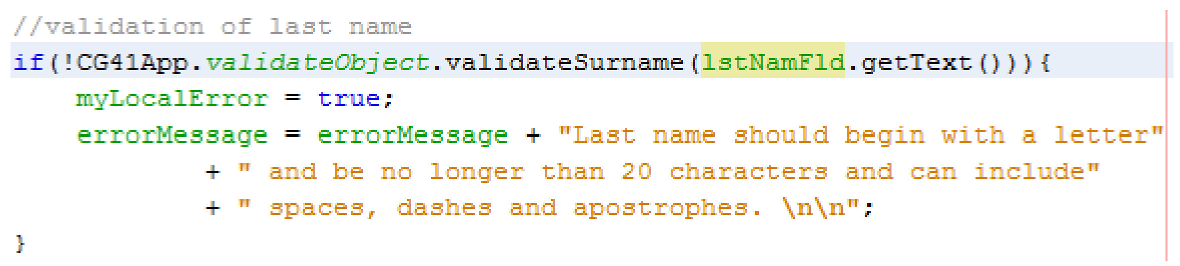
In the image above, the first thing we see is a String variable being set to contain the beginning of an error message. Throughout the following code, more text will be added to this variable to create an appropriate error message to display to the user for where the validation failed to pass.

Next we see an IF statement that will only trigger if the Boolean variable myLocalError is false. Earlier, if when checking to see if the fields were blank this variable was set to true, this IF statement wouldn't trigger and the code would skip over this section. So the validation checks on making sure what the user has entered is formatted correctly will only execute if something has been entered into all fields. This is to make sure that the user doesn't see multiple error messages such as getting an error message that says they need to fill out all the fields but getting an error message at the same time that says all of the fields failed to pass validation.

The second IF statement above checks to see if the entered first name passes the predefined validation pattern. The validation object was created in the class called CG41App and the method I am calling from the ValidateObject is validateFirstName. The string I'm passing to this method is the text from the First Name field on the form. If the validation comes back false, the IF statement code executes then the Boolean value myLocalError is set true and more information is added to the errorMessage string.

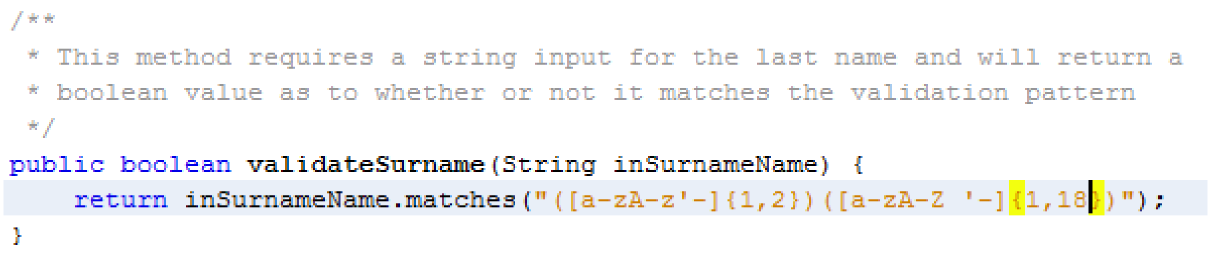
```
/**
 * This method requires a string input for the first name and will return a
 * boolean value as to whether or not it matches the validation pattern
 */
public boolean validateFirstName(String inFirstName) {
    return inFirstName.matches("[A-Z][a-zA-Z]{1,9}");
}
```

As I keep mentioning a "predefined validation pattern" I would like to show an example and explain what this means. I have been able to exploit a library or function in the programming language called "Regular Expressions". These expressions, in my case, allow me to test strings against a string that I have defined. In the example above, the string that I am testing is the firstname which is received from where the method is called. In this example, the first name sent to this method would be the name in the first name field in the Add Govenor form. I am comparing this name against the pattern "[A-Z] [a-zA-Z]{1,9}". This means that the first letter has to be a capital, but only a letter. The next at least 1, but up to 9 letters must be letters, but can be a mixture of upper or lower case. This allows for a total of 10 letters in the first name. If the validation passes, a true boolean result is returned to the method that called it. Else, it returns false.



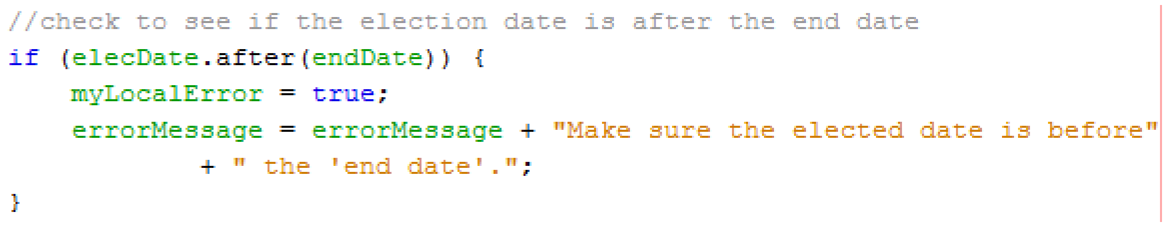
```
 //validation of last name
 if(!CG41App.validateObject.validateSurname(lstNamFld.getText())){
     myLocalError = true;
     errorMessage = errorMessage + "Last name should begin with a letter"
             + " and be no longer than 20 characters and can include"
             + " spaces, dashes and apostrophes. \n\n";
 }
```

The validation IF statement above works almost like previous validation test, but the pattern to compare by and the error message if the validation fails are slightly different. For instance, the above error message explains that the last name must be no longer than 20 characters but can include spaces, dashes and apostphes. Same as before, if the validation passes we move on. Else, add to the error message then move on.



```
/**
 * This method requires a string input for the last name and will return a
 * boolean value as to whether or not it matches the validation pattern
 */
public boolean validateSurname(String inSurnameName) {
    return inSurnameName.matches("([a-zA-z'-]{1,2})([a-zA-Z '-]{1,18})");
}
```

Above is the validation pattern that is used to check the last name. Instead of requiring a capital letter first, the last name allows the user to input up to 2 lower case, upper case, dashes or apostrophes followed by up to 18 upper or lower case letters as well as dashes, apostrophes or spaces. This should allow for people with last names that begin with lower case letters, or include spaces, dashes or apostrophes to be entered into the system.



```
//check to see if the election date is after the end date
if (elecDate.after(endDate)) {
    myLocalError = true;
    errorMessage = errorMessage + "Make sure the elected date is before"
            + " the 'end date'.";
}
```

The above validation code is unique in that it doesn't call for an external function from another class that I've made to work. There is a function in the Date class that Java provides called "after" that allows me to test two Date variables against each other to see if one date is after another. We want to make sure that the date the governor was elected was before the date their election term ends else this wouldn't make sense. So if the election date IS before the end date, then we don't need to do anything. Else, set the Boolean value to true to prevent adding the governor and add to the error message.