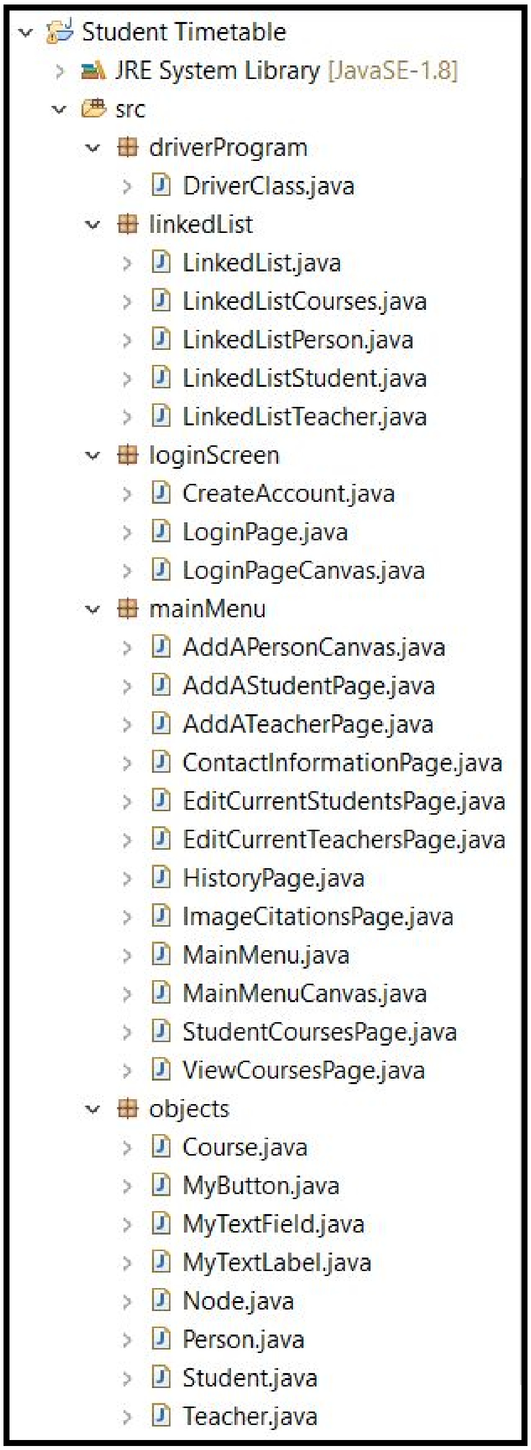
Criterion C: Development

My high school student timetable program is written in Java. The purpose of this program is to provide the guidance counselors in our schools with a database to manage timetables. This program is a user-friendly program and is simple to use. Figure 1 below shows all of the classes involved in my program.

Figure 1 - All classes in my program



In figure 1, the “objects” package contains classes that can be instantiated more than once during the execution of my program. For example, a “Student” object is created every time a student is added to the database.

The “MyButton,” “MyTextField,” and “MyTextLabel” classes from the “objects” package inherit properties from the “JButton,” “JTextField,” and “JTextLabel” classes respectively. Figures 2, 3 and 4 show the code for these classes.

Figure 2 - MyButton class

```java
package objects;

import javax.swing.*;

// Custom class for allowing easier creation of buttons
public class MyButton extends JButton {
    private static final long serialVersionUID = 1L;

    public MyButton (String text) {
        super (text, null);
    }

    public void setButton(int width, int height, int x, int y) {
        setSize(width, height);
        setLocation(x, y);
        setOpaque(true);
    }
}
```

Figure 3 - MyTextField class

```java
package objects;

import javax.swing.*;

// Custom class for allowing easier creation of text fields
public class MyTextField extends JTextField {
    private static final long serialVersionUID = 1L;

    public void setTextField(int width, int height, int x, int y) {
        setSize(width, height);
        setLocation(x, y);
    }
}
```

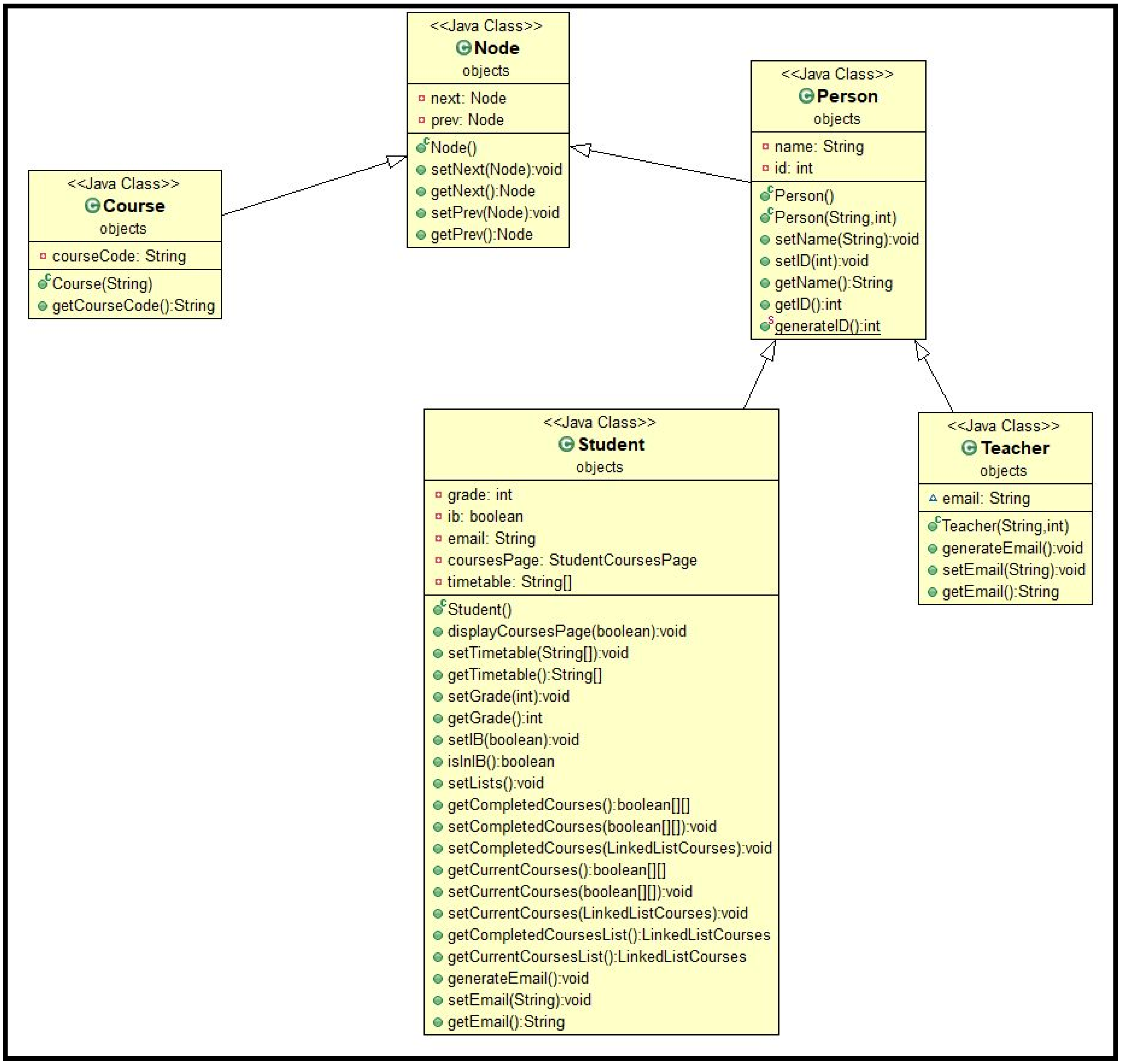
Figure 4 - MyTextLabel class

```java
package objects;

import java.awt.*;

// Custom class for allowing easier creation of labels
public class MyTextLabel extends JLabel {
    private static final long serialVersionUID = 1L;

    public MyTextLabel(String text, int center) {
        super (text, null, center);
    }

    public void setTextLabel(int width, int height, int x, int y, Color foregroundColour, Color backgroundColour, Font f) {
        setSize(width, height);
        setLocation(x, y);
        setForeground(foregroundColour);
        setBackground(backgroundColour);
        setFont(f);
        setOpaque(true);
    }
}
```

The reason I used inheritance with the “MyButton,” “MyTextField,” and “MyTextLabel” classes is to allow for easier instantiation of buttons, text fields, and text labels in my program.

In the “objects” package, I also have the “Node,” “Course,” “Person,” “Student,” and “Teacher” classes. These classes follow the inheritance hierarchy outlined in figure 5 below.

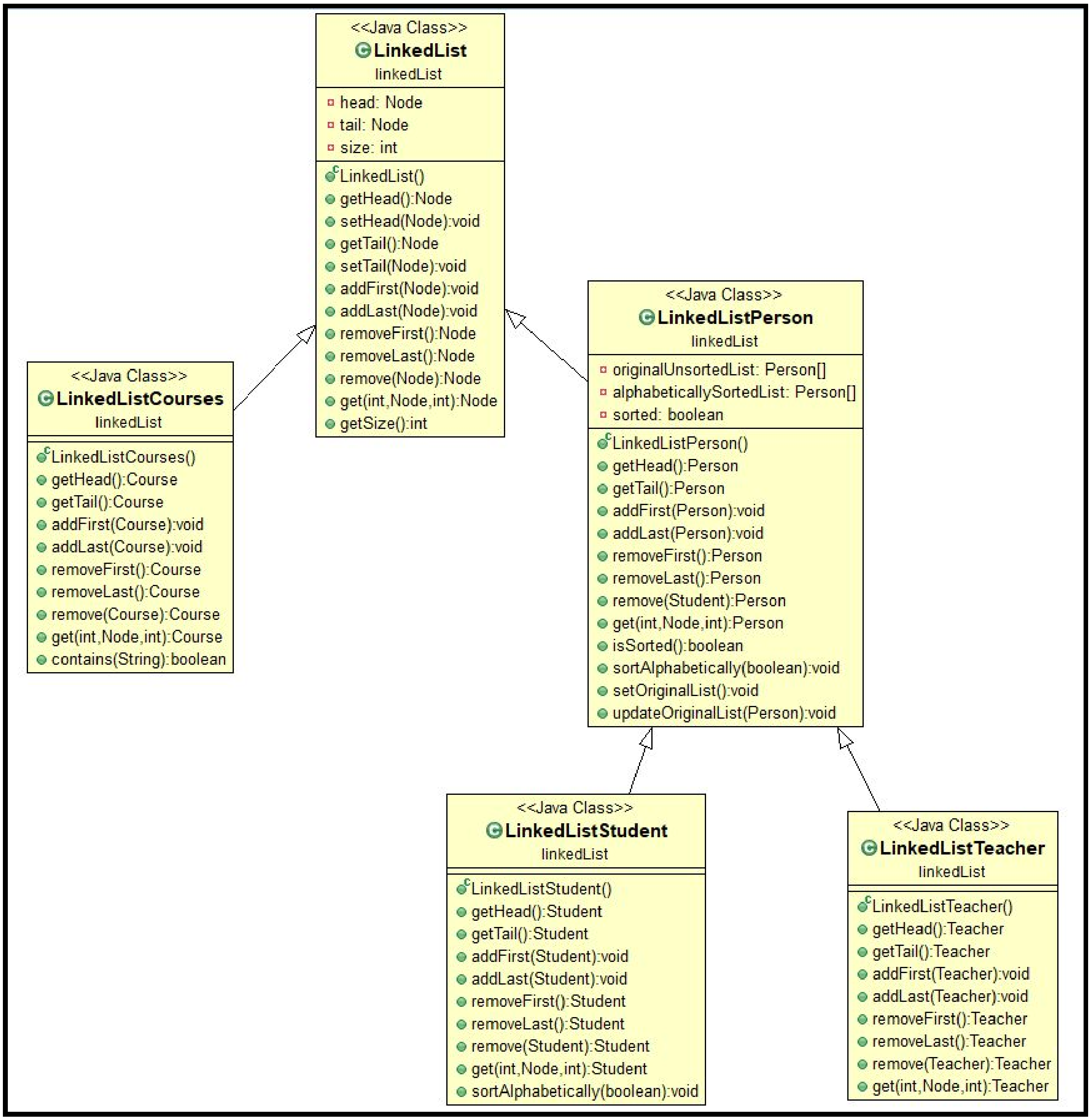
Figure 5 - UML Diagram showing the hierarchy of the “Node,” “Course,” “Person,” “Student,” and “Teacher” classes



In this case, inheritance is used to reuse lines of code, which helps in keeping the code more organized and allowing for easier code modifications.

By referring back to figure 1, notice the “linkedlist” package. This package contains my custom “LinkedList” class, which is a doubly linked list class that allows for effective insertion and deletion. Figure 6 shows the inheritance hierarchy of classes in this “linkedlist” package.

Figure 6 - UML Diagram showing the hierarchy of classes inside of the “linkedlist” package



Similarly to figure 5, inheritance is used here for code reusability and easier code modifications. The three most important methods of my “LinkedList” class are “addLast()”, “remove()”, and “get()”, which are displayed in figures 7, 8, and 9 below.

Figure 7 - “addLast()” method in “LinkedList” class

```
public void addLast(Node n) {
    if (tail == null) {
        head = tail = n;
        n.setNext(null);
        n.setPrev(null);
    } else {
        tail.setNext(n);
        n.setPrev(tail);
        n.setNext(null);
        tail = n;
    }
    size ++;
}
```

Figure 8 - “remove()” method in “LinkedList” class

```
public Node removeFirst() {
    Node n = head;
    head = head.getNext();
    n.setNext(null);
    head.setPrev(null);
    size --;
    return n;
}

public Node removeLast() {
    Node n = tail;
    tail = tail.getPrev();
    n.setPrev(null);
    tail.setNext(null);
    size --;
    return n;
}

public Node remove(Node n) {
    if (n.getPrev() == null && n.getNext() == null) {
        head = tail = null;
        size --;
        return n;
    }
    else if (n.getPrev() == null) return(removeFirst());
    else if (n.getNext() == null) return(removeLast());
    else {
        Node prev = n.getPrev(), next = n.getNext();
        n.setPrev(null);
        n.setNext(null);
        prev.setNext(next);
        next.setPrev(prev);
        size --;
        return n;
    }
}
```

The “remove()” method has O(1) time complexity because each node has a pointer to its previous and next nodes. This is the main reason why a doubly linked list data structure is used to store the list of student and teacher objects in my program.

Figure 9 - "get()" method in "LinkedList" class

```java
public Node get(int curIdx, Node curNode, int target) {
    if (curIdx == target) return curNode;
    return get(curIdx + 1, curNode.getNext(), target);
}
```

From figure 9 above, a recursive algorithm is used for the "get()" method instead of an iterative algorithm because although both of these algorithms have the same time complexity of O(n), where n is the size of the list, a recursive algorithm is used here because it requires less lines of code to implement.

Figure 10 below shows the process of dynamic polymorphism when a subclass of my "LinkedList" class uses some methods.

Figure 10 - Dynamic polymorphism in the "LinkedListCourses" class achieved through method overriding

```java
package linkedList;

import objects.*;

public class LinkedListCourses extends LinkedList {
    public LinkedListCourses() {
        super();
    }

    public Course getHead() {
        return (Course) super.getHead();
    }

    public Course getTail() {
        return (Course) super.getTail();
    }

    public void addFirst(Course st) {
        super.addFirst(st);
    }

    public void addLast(Course st) {
        super.addLast(st);
    }

    public Course removeFirst() {
        return (Course) super.removeFirst();
    }

    public Course removeLast() {
        return (Course) super.removeLast();
    }

    public Course remove(Course st) {
        return (Course) super.remove(st);
    }

    public Course get(int curIdx, Node curNode, int target) {
        return (Course) super.get(curIdx, curNode, target);
    }
```

The lines of code highlighted in red prove that the superclass contains some of the exact same methods as the subclass. Therefore, dynamic polymorphism occurs.

This dynamic polymorphism element allow the reusability of method names and therefore makes method-calling less confusing

The “DriverClass” is the class which contains the main method. The declaration and instantiation of the objects in the main class are displayed in figure 11 below.

Figure 11 - Declaration and Instantiation of objects in the “DriverClass”

Declaration of objects

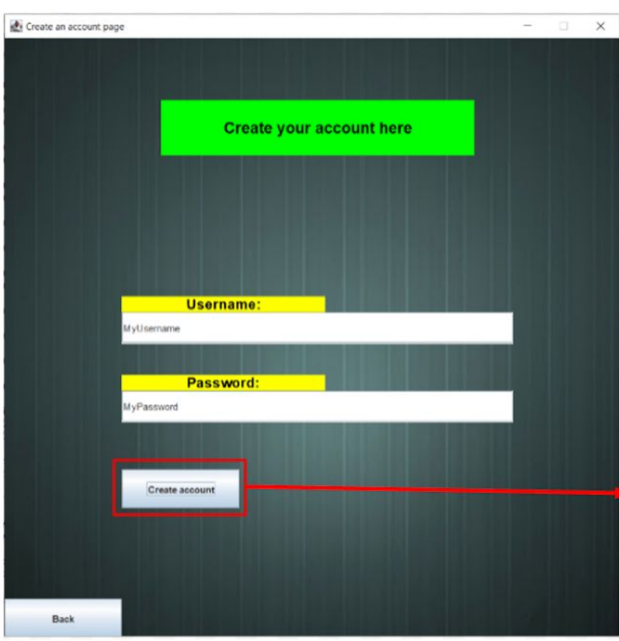
```java
public class DriverClass {
    private static LoginPage loginPage;
    private static LoginPageCanvas loginCanvas, createAccountCanvas;
    private static CreateAccount createAccountPage;

    private static MainMenu mainMenuPage;
    private static MainMenuCanvas mainMenuCanvas;

    private static AddAStudentPage addAStudent;
    private static AddAPersonCanvas addAStudentCanvas;

    private static EditCurrentStudentsPage editCurrentStudents;

    private static ViewCoursesPage viewCourses;

    private static AddATeacherPage addATeacher;
    private static AddAPersonCanvas addATeacherCanvas;

    private static EditCurrentTeachersPage editCurrentTeachers;

    private static ContactInformationPage contactInformationPage;

    private static HistoryPage historyPage;

    private static ImageCitationsPage imageCitationsPage;
```

Instantiation of objects

```java
public static void main(String[] args) throws IOException {
    loginCanvas = new LoginPageCanvas();
    loginPage = new LoginPage(loginCanvas);

    createAccountCanvas = new LoginPageCanvas();
    createAccountPage = new CreateAccount(createAccountCanvas);

    historyPage = new HistoryPage();

    mainMenuCanvas = new MainMenuCanvas();
    mainMenuPage = new MainMenu(mainMenuCanvas);

    addAStudentCanvas = new AddAPersonCanvas();
    addAStudent = new AddAStudentPage(addAStudentCanvas);

    editCurrentStudents = new EditCurrentStudentsPage();

    viewCourses = new ViewCoursesPage();

    addATeacherCanvas = new AddAPersonCanvas();
    addATeacher = new AddATeacherPage(addATeacherCanvas);

    editCurrentTeachers = new EditCurrentTeachersPage();

    contactInformationPage = new ContactInformationPage();

    imageCitationsPage = new ImageCitationsPage();
}
```

The login screen appears after running the main method. An important step of logging in is checking whether the user enters the correct username and password. Hashing is used here to generate hash codes for the user’s username and password. Figures 12 and 13 explain this hashing process in greater detail.

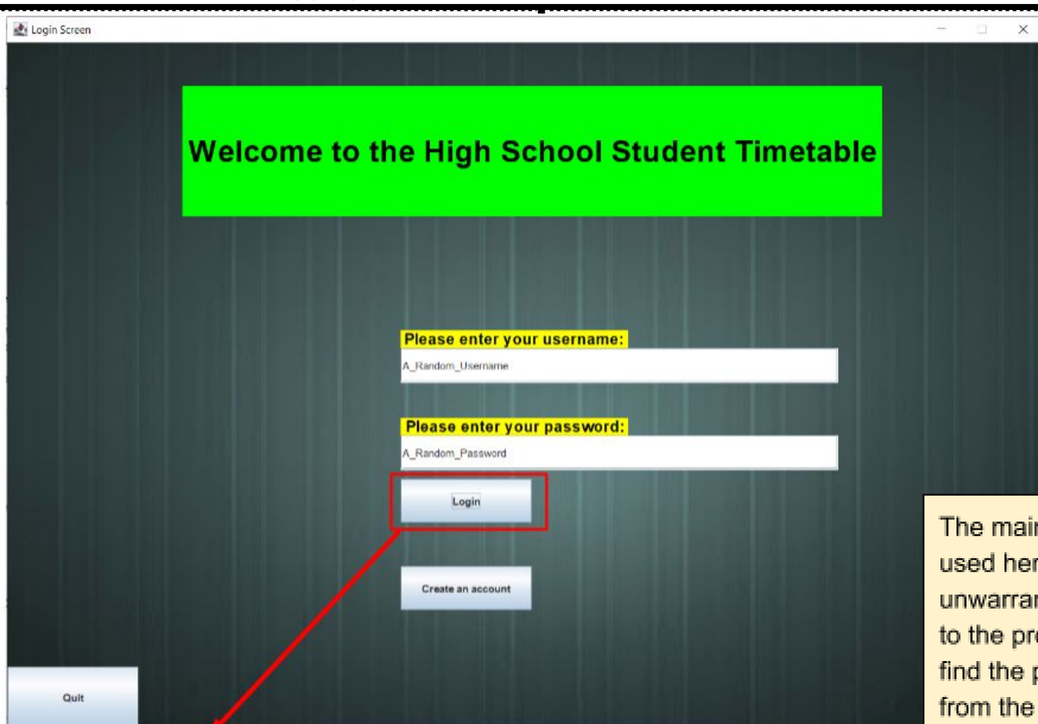
Figure 12 - The hashing involved in the process of logging in




```java
// Method for generating the hashcode of a string
public long generateHashcode(String str) {
    int n = str.length();
    long hshCode = 0;

    /*
     * A prime number is chosen as the seed because a prime number is coprime to all other numbers.
     * When overflow happens, since 131 is coprime to Long.MAX_VALUE,
     * the generated hashcode will be secure by evenly distributing the numbers.
     */
    int seed = 131;

    for (int i = 1; i <= n; i ++) {
        hshCode = hshCode * seed + str.charAt(i - 1);
    }

    return hshCode;
}
```

```java
// Immediately after the user has created an account, this method is called
public void setUsernameAndPassword(String username, String password) {
    usernameHashcode = generateHashcode(username);
    passwordHashcode = generateHashcode(password);

    // Saving the user's login info
    try {
        loginInfo.seek(0);
        loginInfo.writeBytes(usernameHashcode + "\n" + passwordHashcode);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figure 13 - The process of logging in once an account has been created



The main reason hashing is used here is for security, so unwarranted users with access to the program cannot easily find the program's login info from the "LoginInfo" text file.

```
} else if (button.equals(loginButton)) {  // User has chosen to try logging in
    String tempUsername = usernameTextField.getText().toLowerCase().trim();
    String tempPassword = passwordTextField.getText().toLowerCase().trim();

    long tempUsernameHashcode = generateHashcode(tempUsername);
    long tempPasswordHashcode = generateHashcode(tempPassword);

    if (tempUsernameHashcode != 0 && tempPasswordHashcode != 0 && tempUsernameHashcode == usernameHashcode && tempPasswordHashcode == passwordHashcode) {
        setVisible(false);
        DriverClass.displayMainMenu();
    } else {
        JOptionPane.showMessageDialog(this, "The username or password that you entered is incorrect.", "Warning", JOptionPane.WARNING_MESSAGE);
    }
}
```

Immediately upon logging in, a main menu screen is presented and past info entered into the database by the user is recalled. This data recollection is done through the parsing of text files. The parsing process for reloading the teacher's data into the database is shown in figure 14.

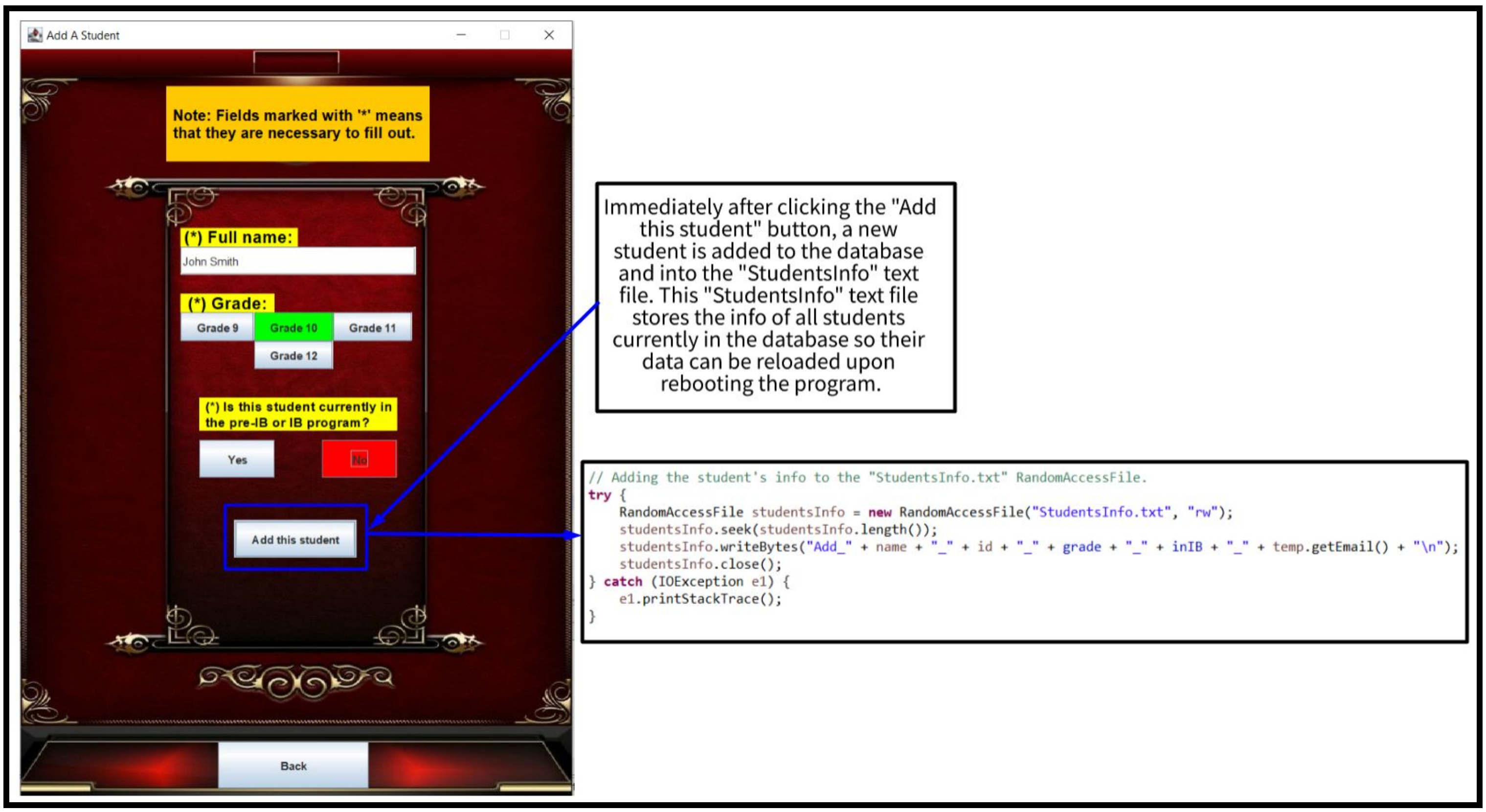
Figure 14 - Parsing a text file to reload past info about teachers added into the database by user

```
// Reloading the past information uploaded by the user about the teachers in the database
RandomAccessFile teachersInfo = new RandomAccessFile("TeachersInfo.txt", "rw");
teachersInfo.seek(0);
line = "x";
while (true) {
    line = teachersInfo.readLine();
    if (line == null) break;

    String[] parts = line.split("_");

    if (parts[0].equals("Add")) {
        Teacher temp = new Teacher(parts[1], Integer.parseInt(parts[2]));
        temp.setEmail(parts[3]);

        teachers.addLast(temp);
    } else if (parts[0].equals("Remove")) {
        Teacher tch = findTeacher(Integer.parseInt(parts[1]));
        teachers.remove(tch);
    }
}
teachersInfo.close();
```

In figure 14, the “String[] parts = line.split(“_”);” is a critical piece of code and the “_” symbol is an important symbol used to separate pieces of information about the teacher from a particular line in the “TeachersInfo” text file.

Adding data to an instance of the RandomAccessFile class is used when adding a student or teacher to my database. The algorithm used to add this data requires direct manipulation of the file pointer using the “seek()” method. The process of this algorithm for adding a student is displayed in figure 15.

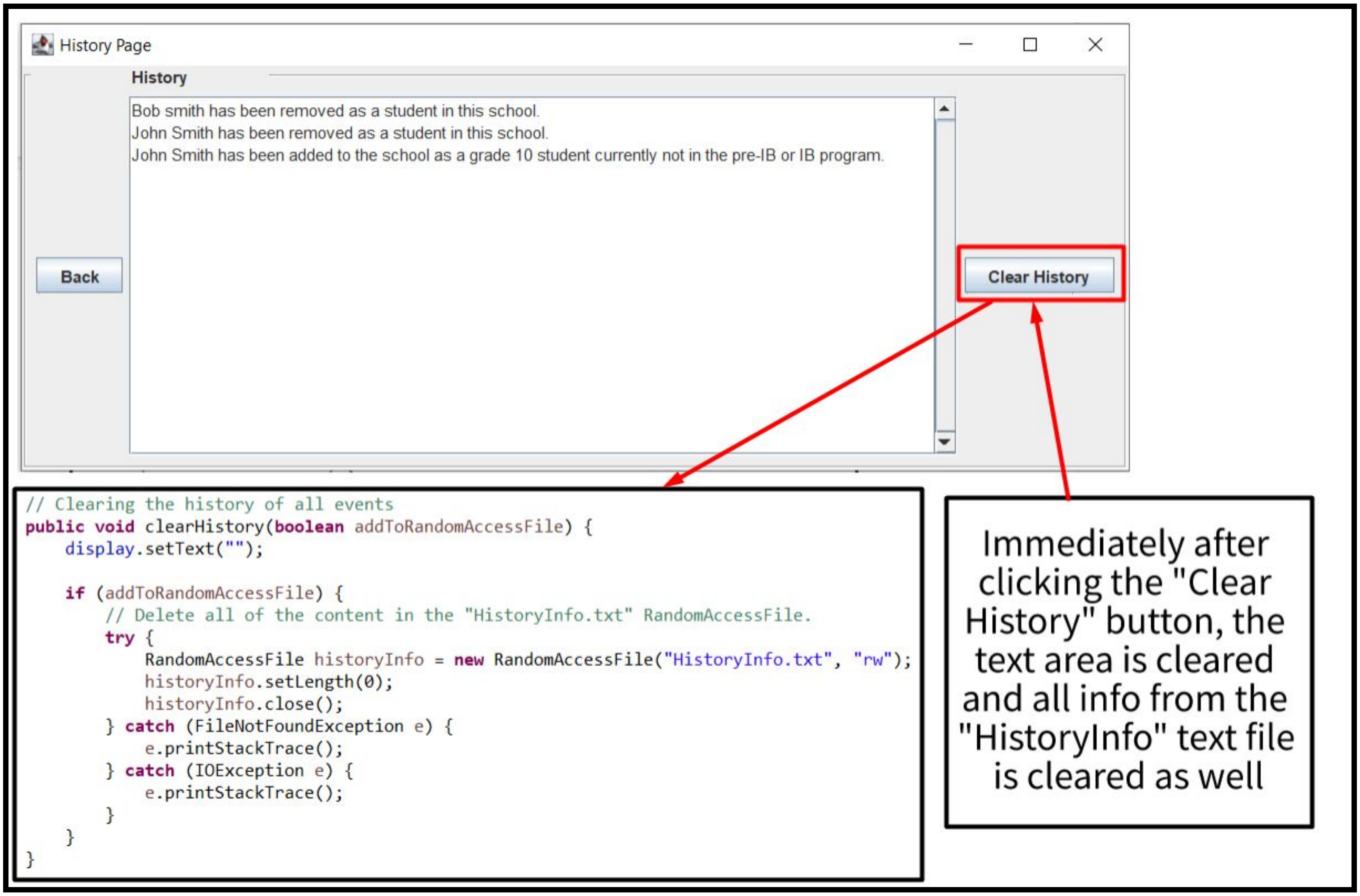
Figure 15 - Algorithm for adding a student’s data to an instance of the RandomAccessFile class



In the code for figure 15 above, the “_” character is used to allow parsing of data in the “StudentsInfo” text file later on. Also, notice the “try” and “catch” keywords in the code of figure 15. These keywords handle exception handling, allowing my program to be robust.

Deletion of data from an instance of the RandomAccessFile class is used when the “Clear History” button is clicked in the history page of my program. This process is outlined in figure 16.

Figure 16 - Clearing all events from the history page and from an instance of the RandomAccessFile class




```
// Clearing the history of all events
public void clearHistory(boolean addToRandomAccessFile) {
    display.setText("");

    if (addToRandomAccessFile) {
        // Delete all of the content in the "HistoryInfo.txt" RandomAccessFile.
        try {
            RandomAccessFile historyInfo = new RandomAccessFile("HistoryInfo.txt", "rw");
            historyInfo.setLength(0);
            historyInfo.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

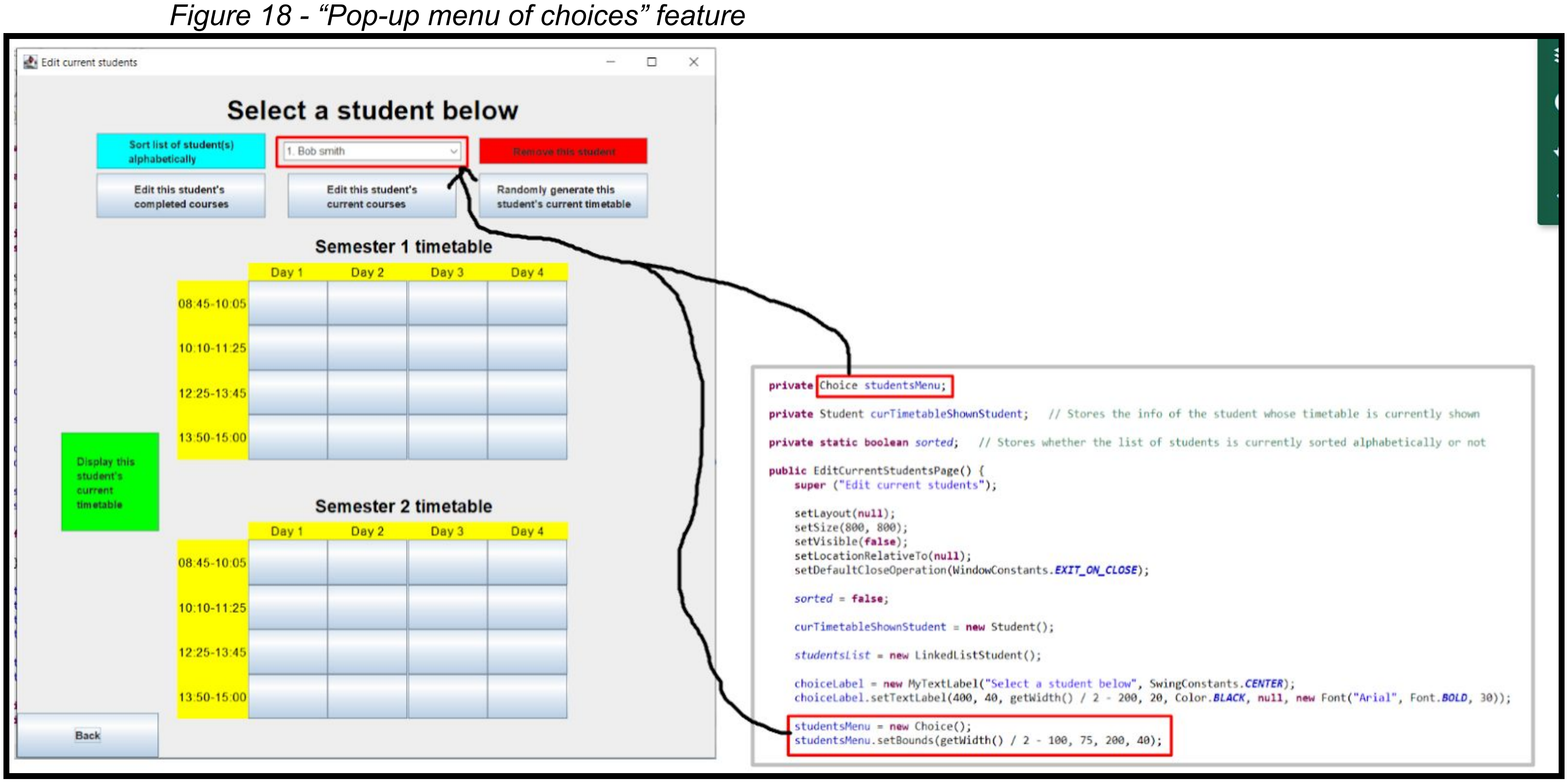
In figure 16 above, the “historyInfo.setLength(0)” in the code is the method which clears all text from the “HistoryInfo” text file.

Overall, the addition, deletion, and parsing of data from instances of the RandomAccessFile class help my program save and keep track of all data added by the user into the database.

Apart from using buttons, text fields, and text labels, I also included the “text area” feature, the “scrollbar” feature, and the “pop-up menu of choices” feature. The “text area” and “scrollbar” features are in the history page, as shown in figure 17. The “pop-up menu of choices” feature can be seen on the page for editing current students, as shown in figure 18.

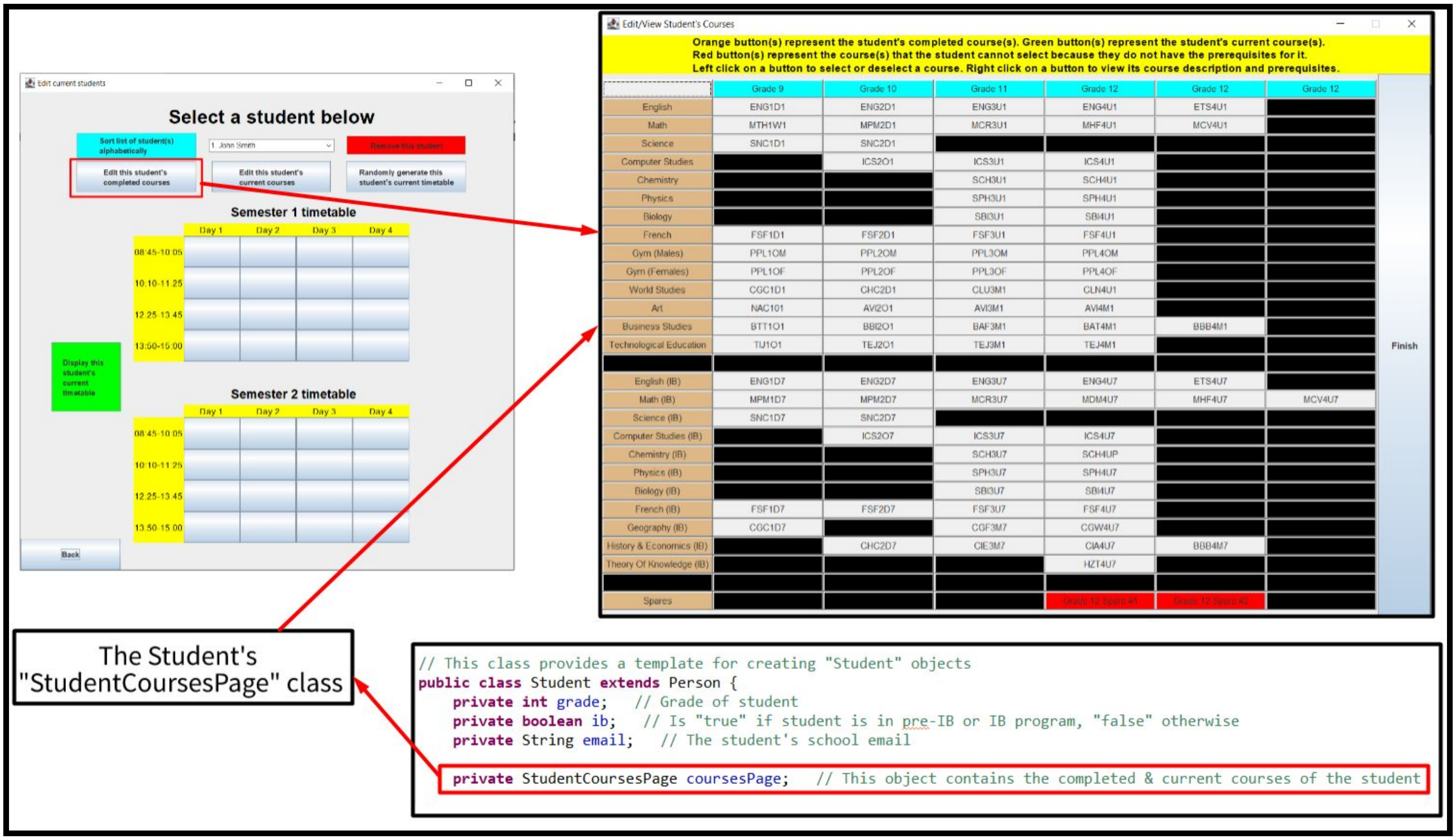
Figure 17 - “Text area” feature and “scrollbar” feature

Figure 18 - “Pop-up menu of choices” feature



When editing a student's courses, the "StudentCoursesPage" class for that student pops up and many 2D arrays of different types are utilized. Figures 19, 20, and 21 show the process of editing a student's completed courses.

Figure 19 - Editing a student's completed courses (Part 1)



In figure 20 below, the “completedCourses” 2D boolean array keeps track of all buttons on the “Edit/View Student’s Courses” page. When a white button is clicked, it is highlighted as orange and “completedCourses[row_of_button_clicked][column_of_button_clicked]” is true.

Figure 20 - Editing a student’s completed courses (Part 2)

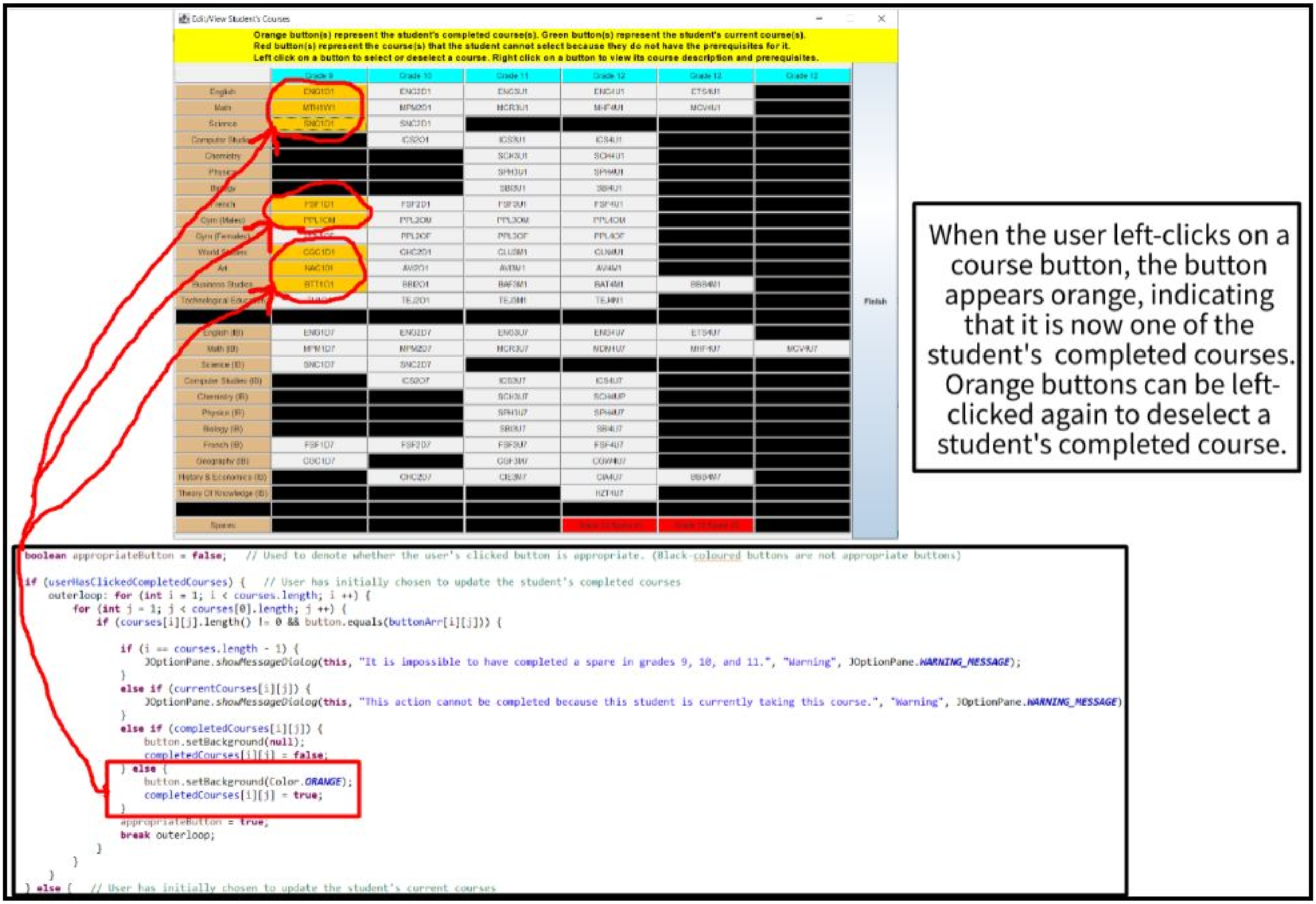
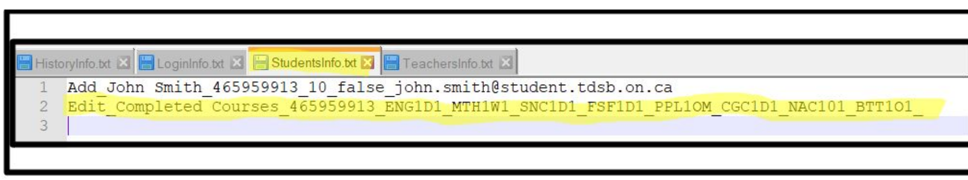


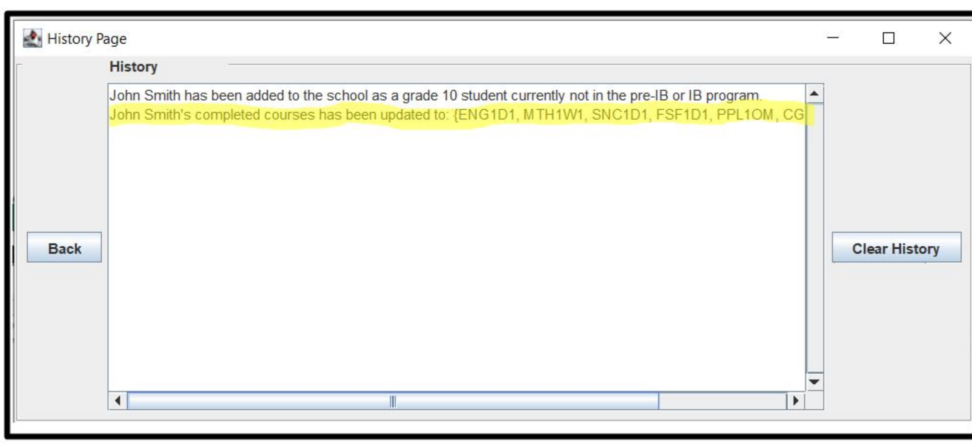
Figure 21 shows important functions that occur when the user clicks the “finish” button after updating the student’s completed courses.

Figure 21 - Editing a student's completed courses (Part 3)


```java
// Adding the event of updating the student's completed courses to the history page.
String str = stu.getName() + "'s completed courses has been updated to: {";
for (int i = 0; i < completedCoursesList.getSize(); i ++) {
    str += completedCoursesList.get(0, completedCoursesList.getHead(), i).getCourseCode();
    if (i != completedCoursesList.getSize() - 1) str += ", ";
}
str += "}.";
DriverClass.addEventToHistory(str, true);

// Adding the event of updating the student's completed courses to the "StudentsInfo.txt" RandomAccessFile.
try {
    RandomAccessFile studentsInfo = new RandomAccessFile("StudentsInfo.txt", "rw");
    studentsInfo.seek(studentsInfo.length());

    str = "Edit_Completed Courses_" + stu.getID() + "_";
    for (int i = 0; i < completedCoursesList.getSize(); i ++) {
        str += completedCoursesList.get(0, completedCoursesList.getHead(), i).getCourseCode() + "_";
    }
    studentsInfo.writeBytes(str + "\n");
    studentsInfo.close();
} catch (FileNotFoundException e1) {
    e1.printStackTrace();
} catch (IOException e1) {
    e1.printStackTrace();
}
```



Adding the student's completed courses info to an instance of the RandomAccessFile class. This data is saved in the "StudentsInfo" text file.



Updating the history page of the program to show the event of updating the student's completed courses

When the program is closed and opened again, the data about the students and their courses need to be reloaded. To set up the student's completed courses, a triple nested "for" loop algorithm is used, as shown in figure 22.

Figure 22 - Algorithm used in setting a student's completed courses

```java
public void setCompletedCourses(LinkedListCourses c) {  // 'c' is a doubly linked list of courses
    completedCourses = new boolean[courses.length][courses[0].length];  // Each cell in this 2D array relates to a specific available or non-available course

    for (int a = 0; a < c.getSize(); a ++) {  // Looping through the elements of linked list 'c'
        String courseCode = c.get(0, c.getHead(), a).getCourseCode();  // Getting the course code from linked list 'c' at index 'a'

        // Nested 'for' loop that loops through all cells of the "completedCourses" 2D boolean array to find which cell is related to the "courseCode" String
        outerloop:
        for (int i = 1; i < courses.length; i ++) {
            for (int j = 1; j < courses[0].length; j ++) {
                if (courses[i][j].length() == 0) continue;  // If there is no text in a particular cell, then this cell is empty and does not contain a course

                String check;
                // Finding the course code associated with a particular cell and assigning it to the "check" variable
                if (i == courses.length - 1) check = courses[i][j];
                else check = courses[i][j].substring(6, 12);

                // If the course code of linked list 'c' matches the cell's course code with row 'i' and column 'j', then mark this cell as a completed course and exit "outerloop"
                if (courseCode.equals(check)) {
                    completedCourses[i][j] = true;
                    break outerloop;
                }
            }
        }
    }

    completedCoursesList = c;  // Along with setting the "completedCourses" 2D boolean array, set the "completedCoursesList" as well
}
```

The algorithm for setting a student's completed courses is the exact same algorithm as the one used for setting the student's current courses

An important feature in the student's course selection process is that right-clicking on a course code shows its course description and prerequisites. An example of this is in figure 23.

Figure 23 - The feature of right-clicking a course while selecting the student's current courses

This button was right-clicked

An example of dynamic polymorphism is provided here with method overriding

```java
// This method is so if the user right clicks on a course button, they can view its course description and prerequisites needed for the course.
@Override
public void mouseClicked(MouseEvent e) {
    Button button = (Button) e.getSource();
    if(SwingUtilities.isRightMouseButton(e)) {
        String str = findButtonString(button);
        if (str.length() == 0) JOptionPane.showMessageDialog(this, "That is not an appropriate course.", "Warning", JOptionPane.WARNING_MESSAGE);
        else JOptionPane.showMessageDialog(this, str);
    }
}
```

The algorithm used to sort a list of students/teachers in my program is my own sorting algorithm which is pretty similar to selection sort. The reason I cannot use selection sort here is because I am dealing with objects instead of primitive data types. The code for my sorting algorithm is shown in figure 24 below.

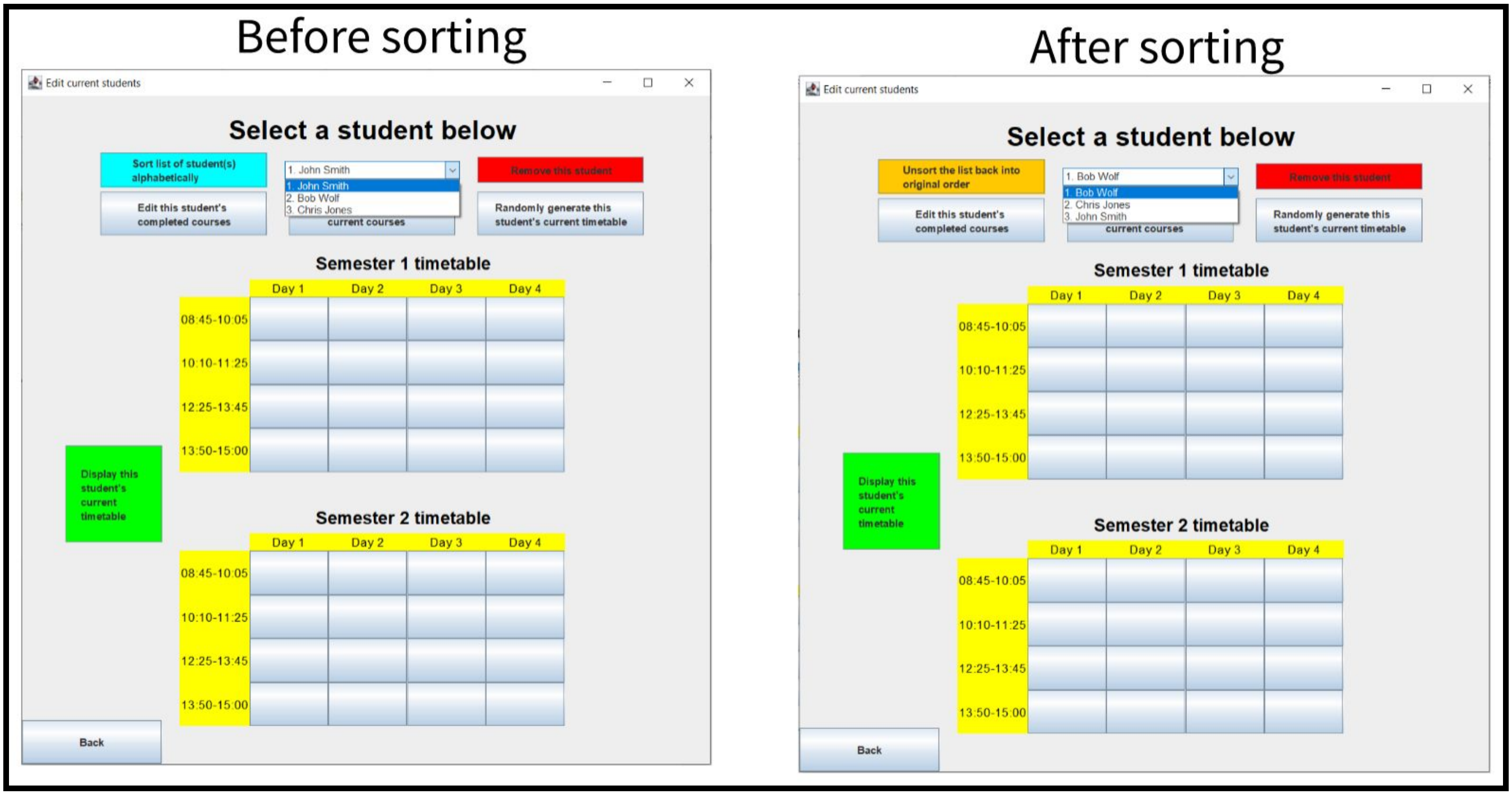
Figure 24 - My sorting algorithm used to sort a list of people alphabetically

```
// If the user has chosen to sort the list alphabetically
if (flag) {
    alphabeticallySortedList = new Person[super.getSize()];   // Array used to store the alphabetically sorted list of people

    boolean[] vis = new boolean[size];  // Stores which indices of "originalUnsortedList" have been assigned to the "alphabeticallySortedList"

    // My sorting algorithm for sorting the list of people alphabetically.
    for (int i = 0; i < size; i ++) {
        int minIdx = -1;  // "minIdx" stores the index of an element in "originalUnsortedList" which has "!vis[index]" and is the smallest lexicographically

        for (int j = 0; j < size; j ++) {  // Looping from the indices of "originalUnsortedList"
            if (!vis[j]) {  // "vis[j]" is "false" if "originalUnsortedList[j]" is not in the "alphabeticallySortedList" yet
                if (minIdx == -1) {  // If the default value for "minIdx" is used, update it to 'j'
                    minIdx = j;
                } else {  // Compare the alphabetic values of "originalUnsortedList[minIdx]" with "originalUnsortedList[j]"
                    String minName = originalUnsortedList[minIdx].getName();
                    String currentName = originalUnsortedList[j].getName();

                    // If "originalUnsortedList[j]" is lexicographically smaller than "originalUnsortedList[minIdx]", update the value of "minIdx"
                    if (currentName.compareTo(minName) < 0) minIdx = j;
                }
            }
        }

        alphabeticallySortedList[i] = originalUnsortedList[minIdx];  // Set the 'i-th' smallest element in "alphabeticallySortedList"
        vis[minIdx] = true;  // Mark "vis[minIdx]" as true, so "minIdx" can no longer be analyzed from the "originalUnsortedList" in future iterations
    }

    // Since this sorting algorithm deals with objects, it is necessary to reset the pointers of the "originalUnsortedList"
    for (int i = 0; i < size; i ++) {
        originalUnsortedList[i].setPrev(null);
        originalUnsortedList[i].setNext(null);
    }

    // Setting up the list based on the sorted "alphabeticallySortedList" array
    for (int i = 0; i < size; i ++) {
        if (size == 1) {
            super.setHead(alphabeticallySortedList[i]);
            super.setTail(alphabeticallySortedList[i]);
        } else if (i == 0) {
            super.setHead(alphabeticallySortedList[i]);
        } else if (i == size - 1) {
            super.setTail(alphabeticallySortedList[i]);
            alphabeticallySortedList[i - 1].setNext(alphabeticallySortedList[i]);
            alphabeticallySortedList[i].setPrev(alphabeticallySortedList[i - 1]);
        } else {
            alphabeticallySortedList[i].setPrev(alphabeticallySortedList[i - 1]);
            alphabeticallySortedList[i - 1].setNext(alphabeticallySortedList[i]);
        }
    }

    sorted = true;   // Indicates that the list is currently sorted alphabetically
}
```

The “originalUnsortedList” array is an array of type “Person” which stores the order of the unsorted list of people

An example of how this algorithm works is displayed in figure 25.

Figure 25 - An execution of the algorithm in figure 24 for sorting a list of students

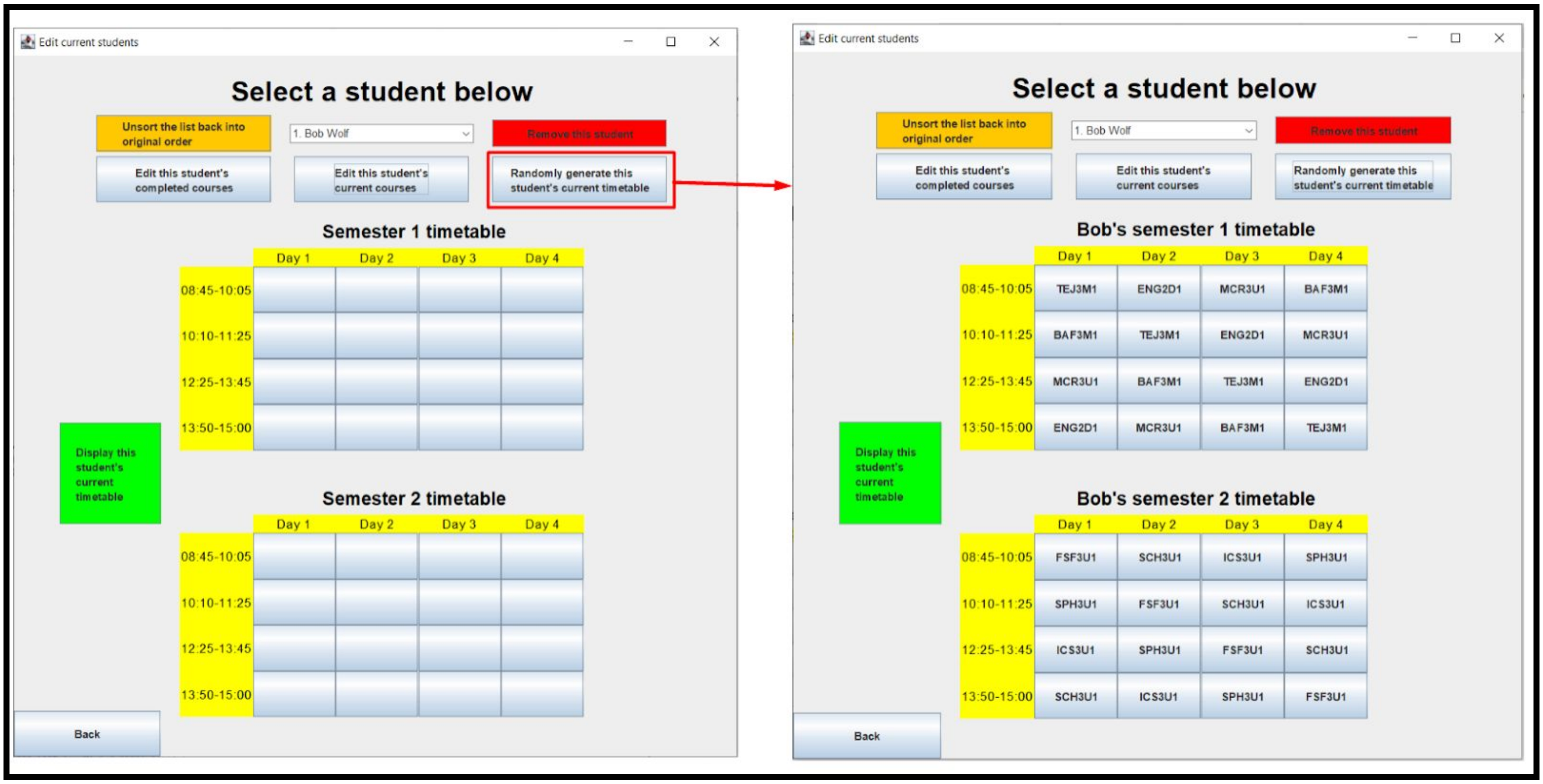


My algorithm to randomly generate and store a student's timetable uses Java's "Math.random()" method. The code for this algorithm is displayed in figure 26 and an example of how it works is displayed in figure 27.

Figure 26 - Algorithm for randomly generating a student's timetable and storing it in an array

```java
// Generating a random timetable for the selected student and displaying it
public void generateTimetable(Student stu) {
    LinkedListCourses list = stu.getCurrentCoursesList();  // List of the student's current courses

    boolean[] visited = new boolean[8];  // Maximum of 8 courses in a timetable

    /* Each cell in this "timetable" array represents a unique course.
     * The ordering of these courses represent the ordering of the student's timetable. */
    String[] timetable = new String[8];  // Maximum of 8 courses in a timetable

    // Algorithm used to randomly generate the student's timetable and store it in the "timetable" array
    for (int i = 0; i < 8; i ++) {  // 8 Courses to be randomly assigned to student's timetable
        while (true) {  // This "while" statement will continue to be "true" until an empty spot is found in "list"
            int x = (int)(Math.random() * 8);  // Generating a random number from 0 to 7 (representing all possible indices in "list")
            if (!visited[x]) {  // "visited[x]" is "false" when "list.get(x)" has not yet been assigned to "timetable" array
                visited[x] = true;  // Mark index 'x' of "list" as occupied in the "timetable" array

                // Getting the course code associated with index 'x' from "list" and assigning it to "timetable[i]"
                timetable[i] = list.get(0, list.getHead(), x).getCourseCode();

                if (timetable[i].equals("Grade 12 Spare #1") || timetable[i].equals("Grade 12 Spare #2")) { // Special case
                    timetable[i] = "Spare";
                }
                break;
            }
        }
    }

    stu.setTimetable(timetable);  // Set this student's current timetable

    displayTimetable(timetable);  // Display the student's current timetable
```

Figure 27 - An execution of the algorithm in figure 26 for generating a student's timetable (Note: The selected student must have 8 current courses in order to generate their timetable)



The algorithms for adding, editing, and sorting teachers are identical to that of the students.

All throughout my code, encapsulation is used to increase the privacy and security of variables in my code. An example of this encapsulation technique is shown in figure 28.

Figure 28 - Encapsulation of data in the "Student" class

```
// This class provides a template for creating "Student" objects
public class Student extends Person {
    private int grade;    // Grade of student
    private boolean ib;    // Is "true" if student is in pre-IB or IB program, "false" otherwise
    private String email;    // The student's school email

    private StudentCoursesPage coursesPage;    // This object contains the completed & current courses of the student

    private String[] timetable;    // Timetable of the student
```

Encapsulation

If these private variables need to be retrieved or set, "getter" and "setter" methods are implemented for them

Word Count: 1088