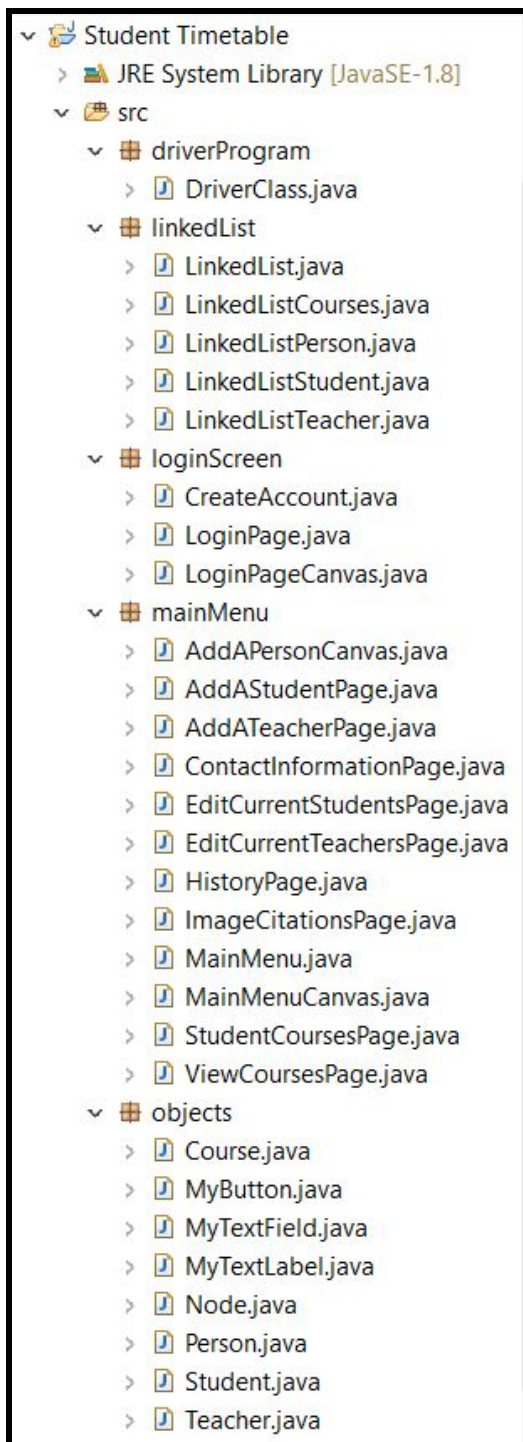


Criterion C: Development

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

```
1 package objects;
2
3 import javax.swing.*;
4
5 // Custom class for allowing easier creation of buttons
6 public class MyButton extends JButton {
7     private static final long serialVersionUID = 1L;
8
9     public MyButton (String text) {
10         super (text, null);
11     }
12
13     public void setButton(int width, int height, int x, int y) {
14         setSize(width, height);
15         setLocation(x, y);
16         setOpaque(true);
17     }
18 }
```

Figure 3 - MyTextField class

```
1 package objects;
2
3 import javax.swing.*;
4
5 // Custom class for allowing easier creation of text fields
6 public class MyTextField extends JTextField {
7     private static final long serialVersionUID = 1L;
8
9     public void setTextField(int width, int height, int x, int y) {
10         setSize(width, height);
11         setLocation(x, y);
12     }
13 }
```

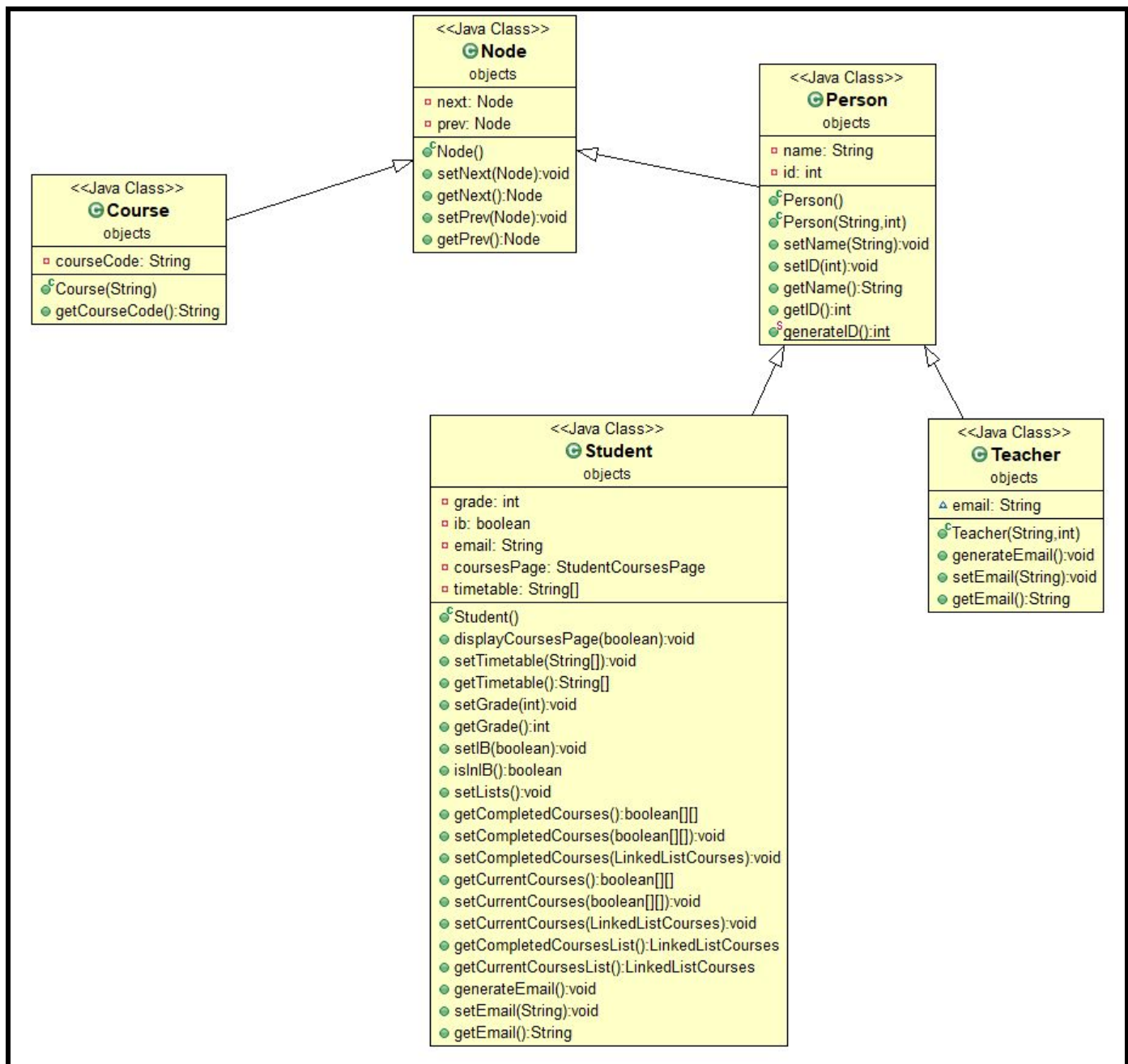
Figure 4 - MyTextLabel class

```
1 package objects;
2
3 import java.awt.*;
4
5 // Custom class for allowing easier creation of labels
6 public class MyTextLabel extends JLabel {
7     private static final long serialVersionUID = 1L;
8
9     public MyTextLabel(String text, int center) {
10         super (text, null, center);
11     }
12
13     public void setTextLabel(int width, int height, int x, int y, Color foregroundColour, Color backgroundColour, Font f) {
14         setSize(width, height);
15         setLocation(x, y);
16         setForeground(foregroundColour);
17         setBackground(backgroundColour);
18         setFont(f);
19         setOpaque(true);
20     }
21 }
22 }
```

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.

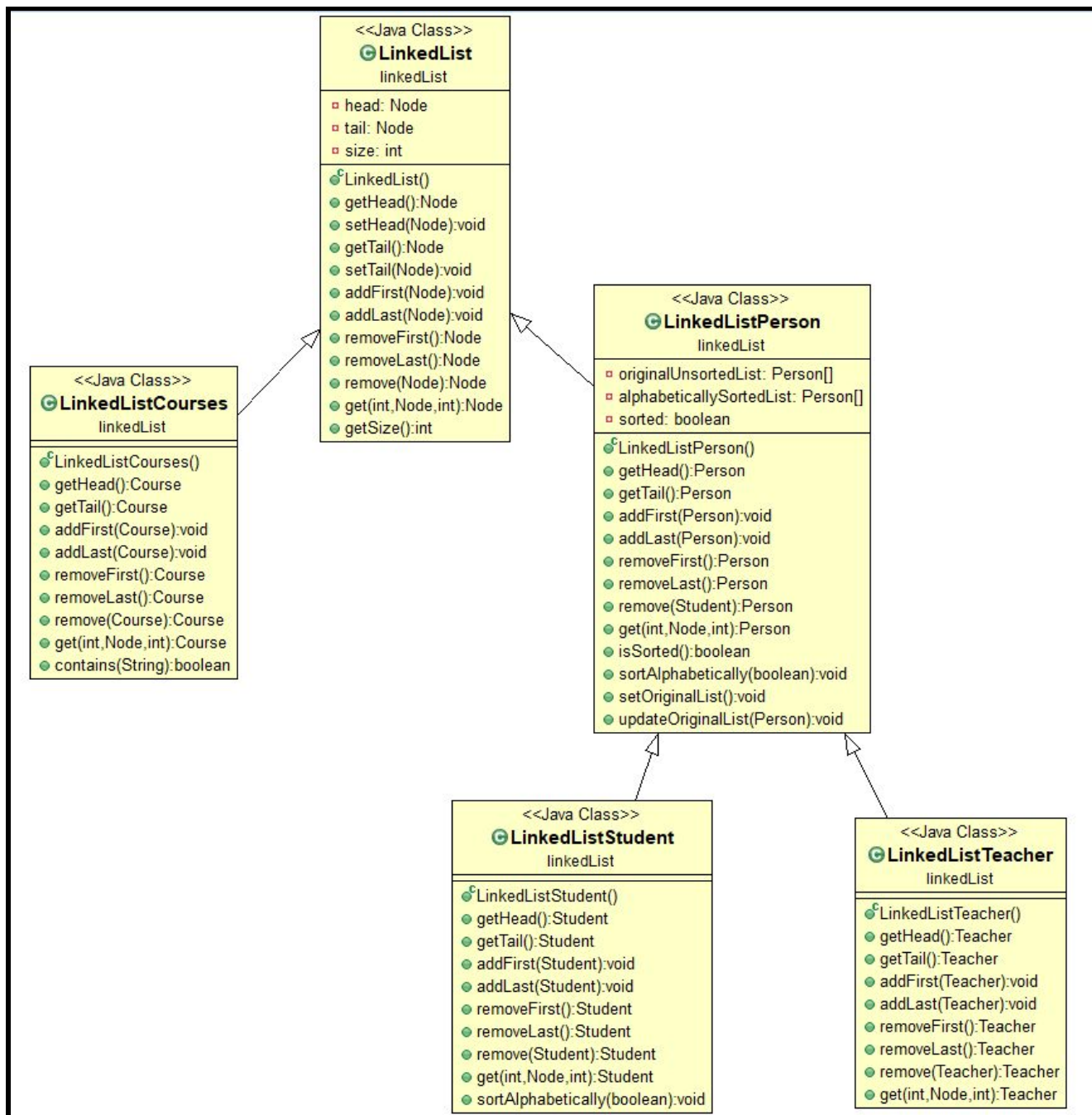
Figure 5 - UML Diagram showing the hierarchy of the “Node,” “Course,” “Person,” “Student,” and “Teacher” classes



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

```
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

```
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”

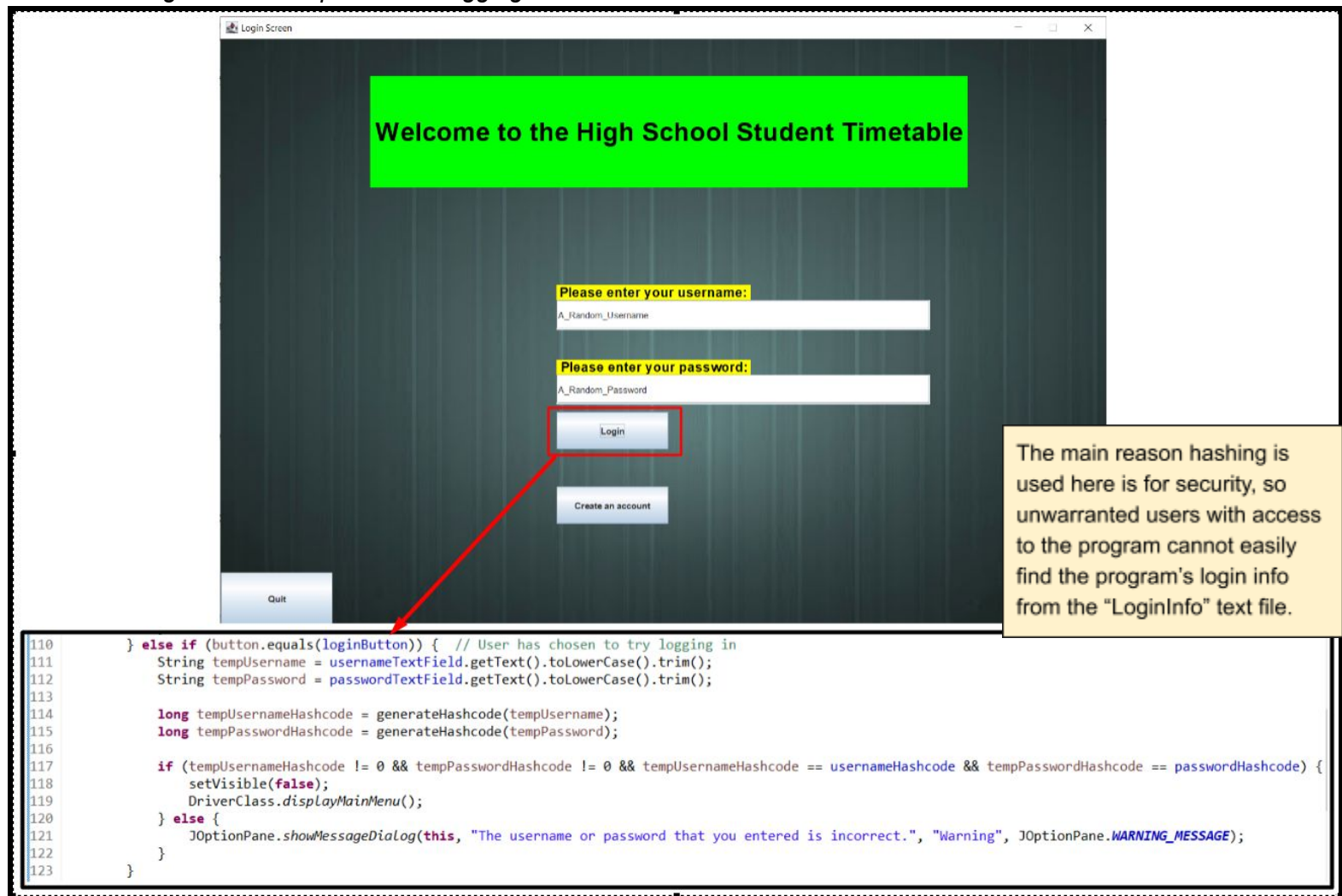
Declaration of objects	Instantiation of objects
<pre>public class DriverClass { private static LoginPage loginPage; private static LoginPageCanvas loginCanvas, createAccountCanvas; private static CreateAccount createAccountPage; private static MainMenu mainMenuPage; private static MainMenuCanvas mainMenuCanvas; 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; }</pre>	<pre>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(); }</pre>

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

	<pre>// 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; } // 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(); } }</pre>
--	---

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



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]);

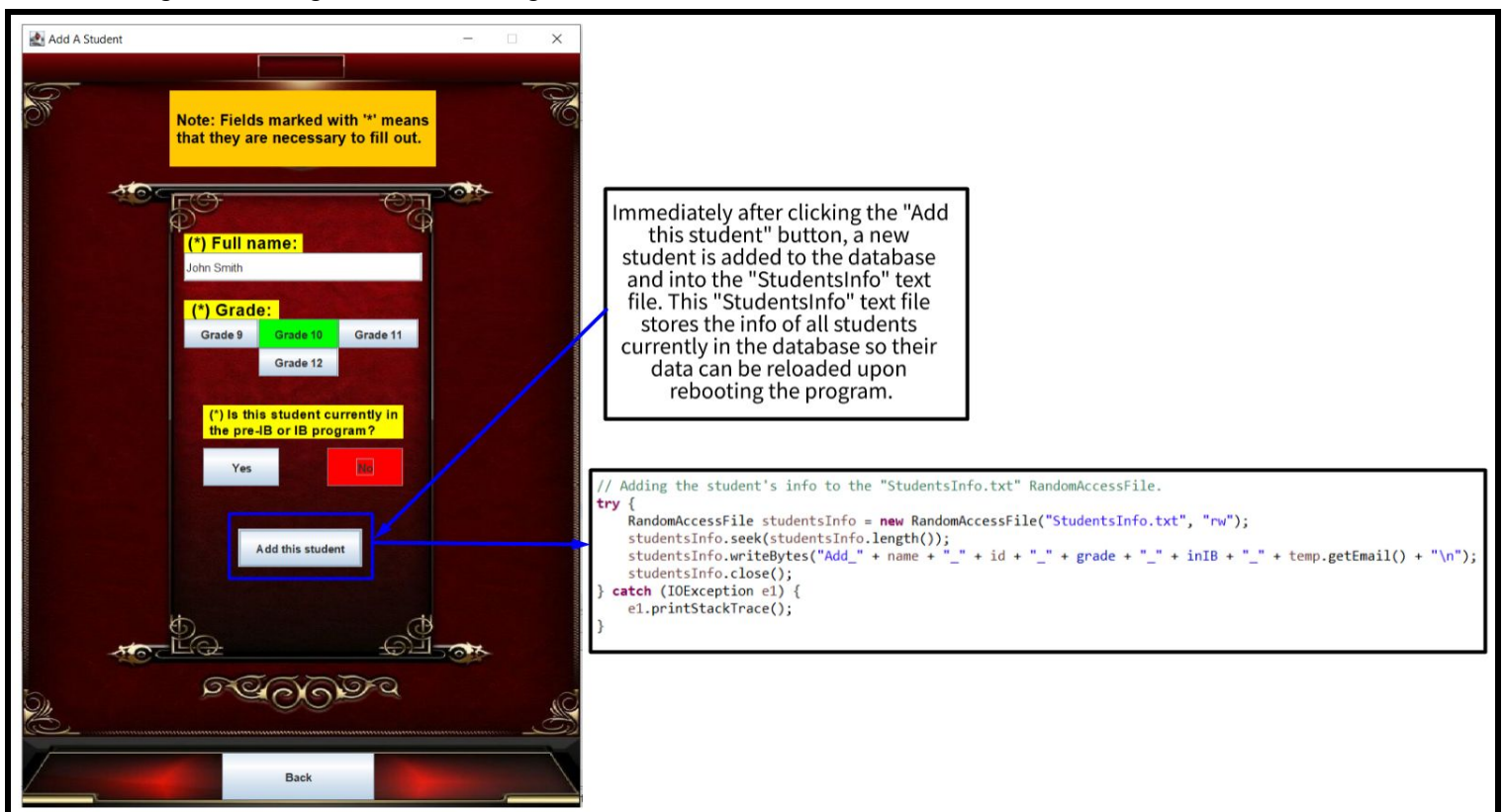
        teachers.addLast(temp);
    } else if (parts[0].equals("Remove")) {
        Teacher tch = findTeacher(Integer.parseInt(parts[1]));
        teachers.remove(tch);
    }
}
teachersInfo.close();

```


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



The screenshot shows a window titled "Add A Student" with a red background and gold borders. It contains a form with the following fields and buttons:

- A yellow note box: "Note: Fields marked with '*' means that they are necessary to fill out."
- A text field for "Full name:" containing "John Smith".
- Radio buttons for "Grade:" with options "Grade 9", "Grade 10" (selected), "Grade 11", and "Grade 12".
- A text field for "Is this student currently in the pre-IB or IB program?" with "Yes" and "No" buttons.
- An "Add this student" button.
- A "Back" button at the bottom.

Annotations include:

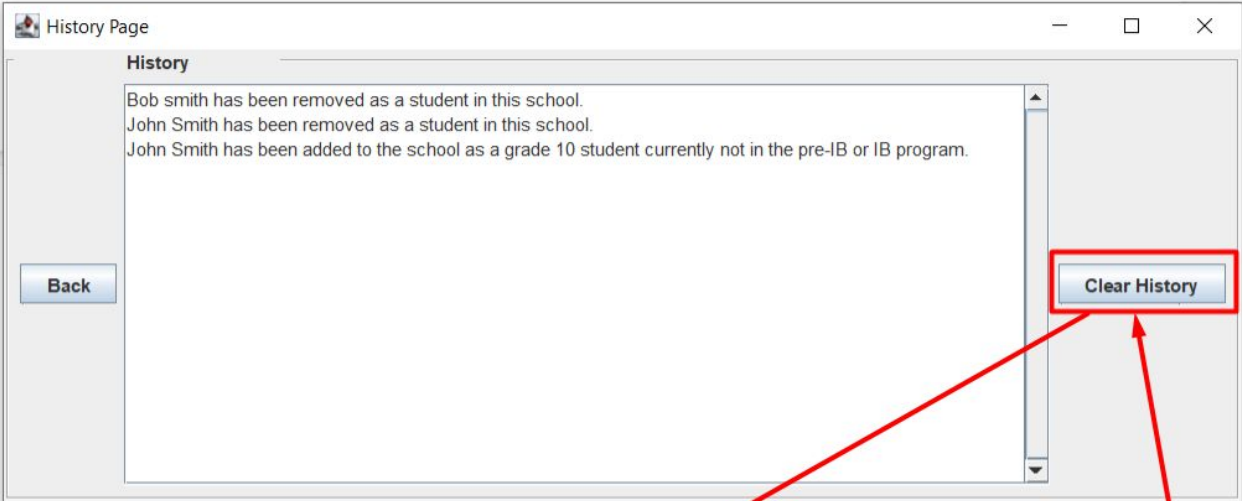
- A blue arrow pointing from the "Add this student" button to a text box: "Immediately after clicking the 'Add this student' button, a new student is added to the database and into the 'StudentsInfo' text file. This 'StudentsInfo' text file stores the info of all students currently in the database so their data can be reloaded upon rebooting the program."
- A blue arrow pointing from the "Add this student" button to a code block.

```
// Adding the student's info to the "StudentsInfo.txt" RandomAccessFile.
try {
    RandomAccessFile studentsInfo = new RandomAccessFile("StudentsInfo.txt", "rw");
    studentsInfo.seek(studentsInfo.length());
    studentsInfo.writeBytes("Add_" + name + "_" + id + "_" + grade + "_" + inIB + "_" + temp.getEmail() + "\n");
    studentsInfo.close();
} catch (IOException e1) {
    e1.printStackTrace();
}
```

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();
        }
    }
}
```

Immediately after clicking the "Clear History" button, the text area is cleared and all info from the "HistoryInfo" text file is cleared as well

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

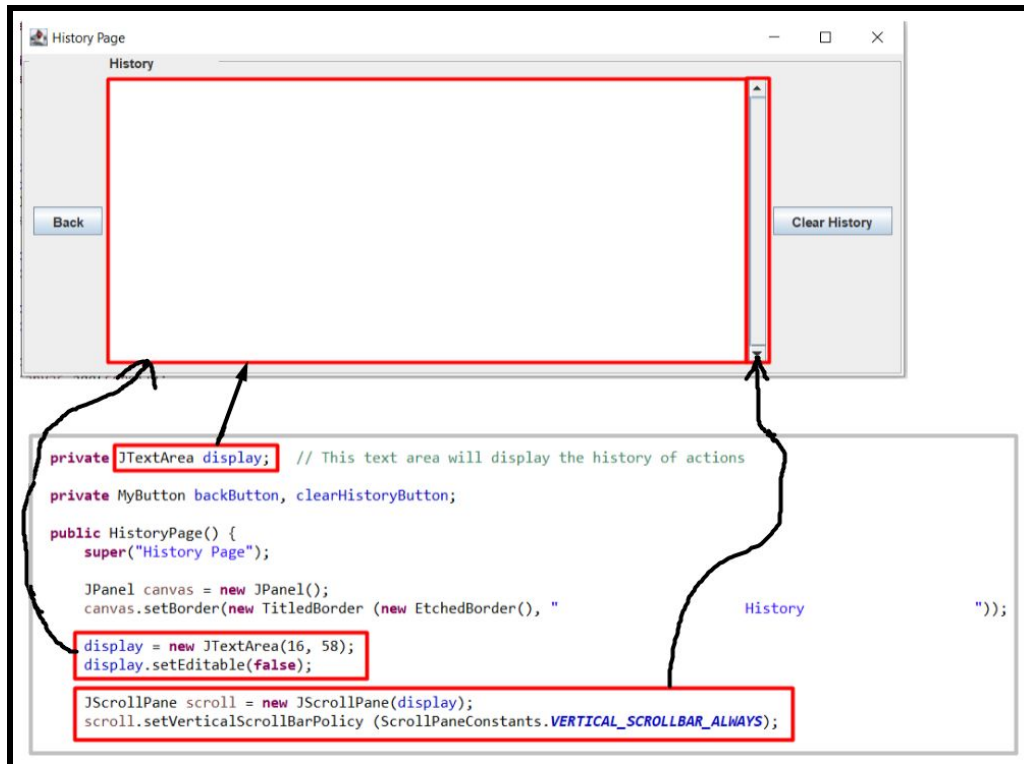
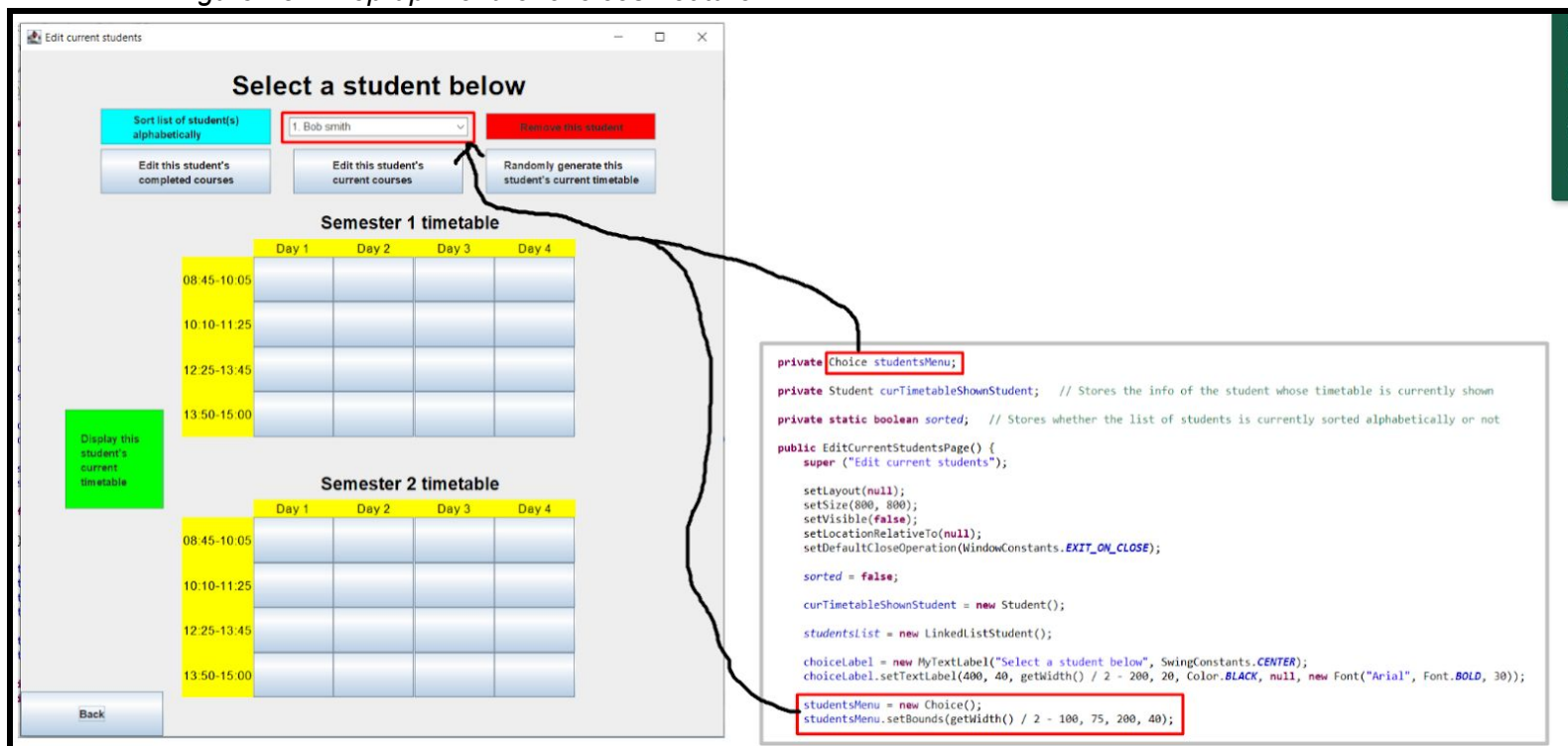


Figure 18 - “Pop-up menu of choices” feature



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)

The figure illustrates the process of editing a student's completed courses. It consists of three main parts:

- Left Panel (User Interface):** A window titled "Edit current students" showing a "Select a student below" section with a dropdown menu for "1 John Smith". Below this are two buttons: "Edit this student's completed courses" (highlighted with a red box) and "Edit this student's current courses". To the right of these buttons is a "Randomly generate this student's current timetable" button. Below the buttons are two timetable sections: "Semester 1 timetable" and "Semester 2 timetable", each with a grid for days 1 through 4 and time slots from 08:45 to 13:00. A green button labeled "Display this student's current timetable" is located to the left of the timetables. A "Back" button is at the bottom left.
- Right Panel (Course Selection Table):** A window titled "Edit/View Student's Courses" showing a table of courses for a student. The table has columns for Grade 9, Grade 10, Grade 11, Grade 12, Grade 12, and Grade 12. The rows list various subjects like English, Math, Science, Computer Studies, Chemistry, Physics, Biology, French, Gym (Males), Gym (Females), World Studies, Art, Business Studies, and Technological Education. The table is color-coded: orange buttons represent completed courses, green buttons represent current courses, and red buttons represent courses that cannot be selected due to prerequisites. A legend at the top explains this color-coding. A "Finish" button is on the right side of the table.
- Bottom Panel (Code Snippet):** A code editor showing the Java code for the "StudentCoursesPage" class. The code defines a class that extends "Person" and includes private attributes for grade, IB status, email, and a reference to the "StudentCoursesPage" object. The "coursesPage" attribute is highlighted with a red box.

The Student's "StudentCoursesPage" 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
}
```


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)



Orange button(s) represent the student's completed course(s). Green button(s) represent the student's current course(s). Red button(s) represent the course(s) that the student cannot select because they do not have the prerequisites for it. Left click on a button to select or deselect a course. Right click on a button to view its course description and prerequisites.

	Grade 9	Grade 10	Grade 11	Grade 12	Grade 13	Grade 14
English	ENG101	ENG201	ENG301	ENG401	ENG501	
Math	MTH101	MTH201	MTH301	MTH401	MTH501	
Science	SCI101	SCI201				
Computer Studies		CS201	CS301	CS401		
Chemistry			CH301	CH401		
Physics			PH301	PH401		
Biology			BI301	BI401		
French	FR101	FR201	FR301	FR401		
Spanish	SP101	SP201	SP301	SP401		
World Studies	WST101	WST201	WST301	WST401		
Art	ART101	ART201	ART301	ART401		
Business Studies	BS101	BS201	BS301	BS401		
Technology	TE101	TE201	TE301	TE401		
English (IB)	ENG101	ENG201	ENG301	ENG401	ENG501	
Math (IB)	MTH101	MTH201	MTH301	MTH401	MTH501	
Science (IB)	SCI101	SCI201				
Computer Studies (IB)		CS201	CS301	CS401		
Chemistry (IB)			CH301	CH401		
Physics (IB)			PH301	PH401		
Biology (IB)			BI301	BI401		
French (IB)	FR101	FR201	FR301	FR401		
Spanish (IB)	SP101	SP201	SP301	SP401		
World Studies (IB)	WST101	WST201	WST301	WST401		
Art (IB)	ART101	ART201	ART301	ART401		
Business Studies (IB)	BS101	BS201	BS301	BS401		
Technology (IB)	TE101	TE201	TE301	TE401		
Spanish						

When the user left-clicks on a course button, the button appears orange, indicating that it is now one of the student's completed courses. Orange buttons can be left-clicked again to deselect a student's completed course.

```

boolean appropriateButton = false; // Used to denote whether the user's clicked button is appropriate. (Black-colored buttons are not appropriate buttons)
if (userHasClickedCompletedCourses) { // User has initially chosen to update the student's completed courses
    outerloop: for (int i = 1; i < courses.length; i++) {
        for (int j = 1; j < courses[0].length; j++) {
            if (courses[i][j].length() != 0 && button.equals(buttonArr[i][j])) {
                if (i == courses.length - 1) {
                    JOptionPane.showMessageDialog(this, "It is impossible to have completed a spare in grades 9, 10, and 11.", "Warning", JOptionPane.WARNING_MESSAGE);
                }
                else if (currentCourses[i][j]) {
                    JOptionPane.showMessageDialog(this, "This action cannot be completed because this student is currently taking this course.", "Warning", JOptionPane.WARNING_MESSAGE);
                }
                else if (completedCourses[i][j]) {
                    button.setBackground(null);
                    completedCourses[i][j] = false;
                }
                else {
                    button.setBackground(Color.ORANGE);
                    completedCourses[i][j] = true;
                }
                appropriateButton = true;
                break outerloop;
            }
        }
    }
} else { // User has initially chosen to update the student's current courses

```

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)

Orange button(s) represent the student's completed course(s). Green button(s) represent the student's current course(s).
 Red button(s) represent the course(s) that the student cannot select because they do not have the prerequisites for it.
 Left click on a button to select or deselect a course. Right click on a button to view its course description and prerequisites.

	Grade 9	Grade 10	Grade 11	Grade 12	Grade 12	Grade 12
English	ENG1D1	ENG2D1	ENG3U1	ENG4U1	ETS4U1	
Math	MTH1W1	MTH2P1	MCR3U1	MFC4U1	MCV4U1	
Science	SC1D1	SC2D1				
Computer Studies		ICS2D1	ICS3U1	ICS4U1		
Chemistry			SCH3U1	SCH4U1		
Physics			SPH3U1	SPH4U1		
Biology			SD0U1	SD0U1		
French	FSL1D1	FSL2U1	FSL3U1	FSL4U1		
Gen. Math	PPL1D1	PPL2D1	PPL3D1	PPL4D1		
Open (Planning)	PP11OP	PP12OP	PP13OP	PP14OP		
World Studies	GC1D1	GC2D1	GL0U1	GLN4U1		
Art	NAC1D1	AV0D1	AV0U1	AV0U1		
Business Studies	BT1D1	BB2D1	BAP3U1	BAT4U1	BBB4U1	
Technological Education	TE1D1	TE2D1	TE3U1	TE4U1		
English (E)	ENG1D1	ENG2D1	ENG3U1	ENG4U1	ETS4U1	
Math (M)	MTH1D1	MTH2D1	MCR3U1	MM4U1	MMF4U1	MCV4U1
Science (S)	SC1D1	SC2D1				
Computer Studies (C)		ICS2D1	ICS3U1	ICS4U1		
Chemistry (Ch)			SCH3U1	SCH4U1		
Physics (Ph)			SPH3U1	SPH4U1		
Biology (B)			SD0U1	SD0U1		
French (F)	FSL1D1	FSL2D1	FSL3U1	FSL4U1		
Geography (G)	GC1D1		GL03U1	GL04U1		
History & Economics (H)		CHC2D1	CHC3U1	CHC4U1	BBB4U1	
Theory Of Knowledge (T)						
Options				Open To Select	Open To Select	

```
// 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();
}
```

```
1 Add John Smith_465959913_10_false_john.smith@student.tdsb.on.ca
2 Edit_Completed_Courses_465959913_ENGL1D1_MTH1W1_SNC1D1_FSF1D1_PPL1OM_CGC1D1_NAC101_BTT101_
3
```

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

History Page

History

- John Smith has been added to the school as a grade 10 student currently not in the pre-IB or IB program.
- John Smith's completed courses has been updated to: {ENG1D1, MTH1W1, SNC1D1, FSF1D1, PPL1OM, CG

Back

Clear History

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

```
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

Orange button(s) represent the student's completed course(s). Green button(s) represent the student's current course(s). Red button(s) represent the course(s) that the student cannot select because they do not have the prerequisites for it. Left click on a button to select or deselect a course. Right click on a button to view its course description and prerequisites.

This button was right-clicked

An example of dynamic polymorphism is provided here with method overriding

```
// 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

Before sorting

After sorting

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

```
// 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 left screenshot shows the 'Edit current students' page. It has a dropdown menu with '1. Bob Wolf' selected. Below it are three buttons: 'Unsort the list back into original order', 'Edit this student's completed courses', and 'Randomly generate this student's current timetable' (highlighted with a red box). There are also two empty buttons: 'Edit this student's current courses' and 'Remove this student'. Below these are two empty timetable grids for 'Semester 1' and 'Semester 2'. A green button 'Display this student's current timetable' is on the left. A 'Back' button is at the bottom left.

The right screenshot shows the same page after clicking the highlighted button. The title is 'Select a student below'. The dropdown menu still shows '1. Bob Wolf'. The buttons are the same. Below them are two filled timetable grids. The first is 'Bob's semester 1 timetable' and the second is 'Bob's semester 2 timetable'. Both have 4 columns (Day 1 to Day 4) and 4 rows (08:45-10:05 to 13:50-15:00). The cells contain course codes. A green button 'Display this student's current timetable' is on the left. A 'Back' button is at the bottom left.

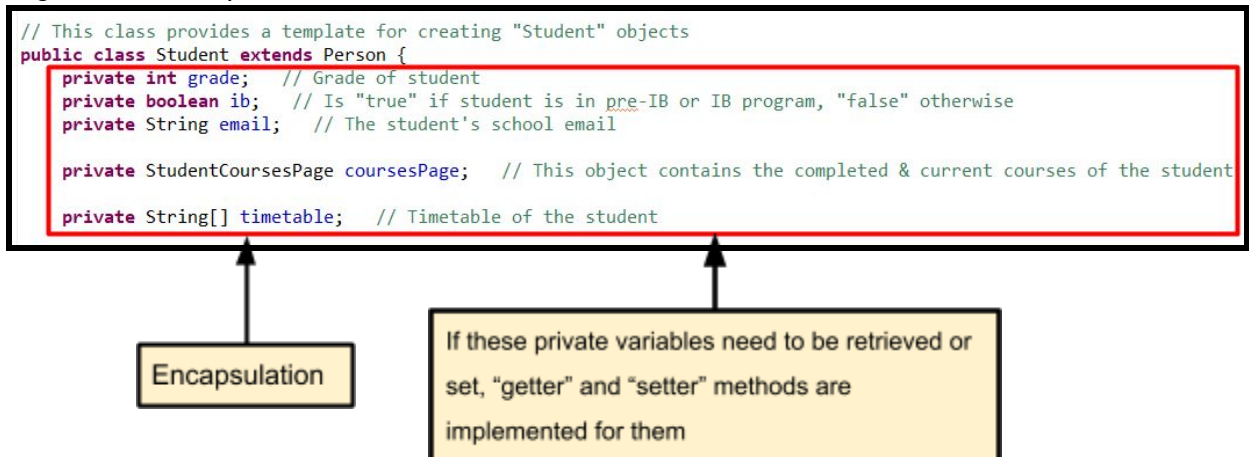
	Day 1	Day 2	Day 3	Day 4
08:45-10:05	TEJ3M1	ENG2D1	MCR3U1	BAF3M1
10:10-11:25	BAF3M1	TEJ3M1	ENG2D1	MCR3U1
12:25-13:45	MCR3U1	BAF3M1	TEJ3M1	ENG2D1
13:50-15:00	ENG2D1	MCR3U1	BAF3M1	TEJ3M1

	Day 1	Day 2	Day 3	Day 4
08:45-10:05	FSF3U1	SCH3U1	ICS3U1	SPH3U1
10:10-11:25	SPH3U1	FSF3U1	SCH3U1	ICS3U1
12:25-13:45	ICS3U1	SPH3U1	FSF3U1	SCH3U1
13:50-15:00	SCH3U1	ICS3U1	SPH3U1	FSF3U1

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



Word Count: 1088