```
JOptionPane.showMessageDialog(this, "Please enter valid
grades", "Error",
JOptionPane.ERROR_MESSAGE);
return;
}
```

In this piece of code, the program tries to parse the text inside two fields as integers, if one of them is not an integer, the error happens and the code inside the Catch part is execute, which is a message window popping up saying that the user needs to enter a valid grade. Another example:

```
try {
            File file = new File(fileName);
            if (!file.exists()) {
                file.createNewFile();
            }
            // write to file
            FileWriter fw = new FileWriter(file, true);
            fw.write(studentInfo + "\n");
            fw.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
```

In this code, the program attempts to open a file, if a file does not exist, it creates the file before writing anything onto it. If anything happens during the execution of the code, the code inside Catch would be executed, printing out what error happens into the console log for the developer to debug.

## V. Test

### 1. Test plan

| No | Description | Data | Expected result | Actual result | Status |
|---|---|---|---|---|---|
| **1** | Register with valid data | username: admin<br>password: admin | A success message display, user is returned to login screen, and "admin,admin" is added into user.csv | As expected | Pass |
| | Register with invalid data by input the same information as last step | username: admin<br>password: admin | A fail message displays, no new data is written on the file user.csv | As expected | Pass |
| **2** | Login with valid data | username: admin<br>password: admin | A success message display, user is directed to student index window | As expected | Pass |

| | | | | | |
|---|---|---|---|---|---|
| | Login with invalid data | username: asasasasasadmin<br>password: admin | A fail message display, user stays on the login window | As expected | Pass |
| **3** | Add a student with valid data | ID:1<br>name: A<br>date of birth: 12/12/2000<br>phone number: 1234567890<br>email address: A@gmail.com<br>address: A street<br>class: 12D<br>english: 40<br>math: 50 | A success message display, user is directed to student index window, the new student is added into the table and student.csv | As expected | Pass |
| | Add a student with duplicated ID | ID:1<br>name: B<br>date of birth: 12/12/2000<br>phone number: 1234567890<br>email address: B@gmail.com<br>address: A street<br>class: 10A<br>english: 90<br>math: 32 | A failure message display, user is directed to student index window, no data is added to table or student.csv | As expected | Pass |
| | Add a student with wrong email format | ID:12<br>name: B<br>date of birth: 12/12/2000<br>phone number: 1234567890<br>email address: Bgmailcom<br>address: A street<br>class: 10A<br>english: 90<br>math: 32 | A failure message display, user is directed to student index window, no data is added to table or student.csv | As expected | Pass |

| | | | | | |
|---|---|---|---|---|---|
| | Add a student with wrong phone number format | ID:12<br>name: B<br>date of birth: 12/12/2000<br>phone number: 1234asffe<br>email address: B@gmail.com<br>address: A street<br>class: 10A<br>english: 90<br>math: 32 | A failure message display, user is directed to student index window, no data is added to table or student.csv | As expected | Pass |
| | Add a student with wrong date format | ID:12<br>name: B<br>date of birth: 12122000<br>phone number: 1234567890<br>email address: B@gmail.com<br>address: A street<br>class: 10A<br>english: 90<br>math: 32 | A failure message display, user is directed to student index window, no data is added to table or student.csv | As expected | Pass |
| | Add a student with wrong grade format | ID:12<br>name: B<br>date of birth: 12/12/2000<br>phone number: 1234567890<br>email address: B@gmail.com<br>address: A street<br>class: 10A<br>english: 90e<br>math: 32 | A failure message display, no data is added to table or student.csv | As expected | Pass |

| | | | | | |
|---|---|---|---|---|---|
| | | ID:12<br>name: B<br>date of birth: 12/12/2000<br>phone number: 1234567890<br>email address: B@gmail.com<br>address: A street<br>class: 10A<br>english: 90<br>math: 32e | A failure message display, no data is added to table or student.csv | As expected | Pass |
| **4** | Edit a student with valid data | ID:1<br>name: A edited<br>date of birth: 12/12/2000<br>phone number: 1234567890<br>email address: A@gmail.com<br>address: A street<br>class: 12D<br>english: 40<br>math: 50 | Success message display, the user is directed back to the index menu and the data of student whose ID is 1 is updated | As expected | Pass |
| | Edit a student with wrong email format | ID:1<br>name: A edited<br>date of birth: 12/12/2000<br>phone number: 1234567890<br>email address: Agmailcom<br>address: A street<br>class: 12D<br>english: 40<br>math: 50 | A failure message display, user is directed to student index window, no data is added to table or student.csv | As expected | Pass |

| | | | | | |
|---|---|---|---|---|---|
| | Edit a student with wrong phone number format | ID:1<br>name: A edited<br>date of birth: 12/12/2000<br>phone number: 12345asd<br>email address: A@gmail.com<br>address: A street<br>class: 12D<br>english: 40<br>math: 50 | A failure message display, user is directed to student index window, no data is added to table or student.csv | As expected | Pass |
| | Edit a student with wrong date format | ID:1<br>name: A edited<br>date of birth: 12122000<br>phone number: 1234567890<br>email address: A@gmail.com<br>address: A street<br>class: 12D<br>english: 40<br>math: 50 | A failure message display, user is directed to student index window, no data is added to table or student.csv | As expected | Pass |
| | Edit a student with wrong grade format | ID:1<br>name: A edited<br>date of birth: 12/12/2000<br>phone number: 1234567890<br>email address: A@gmail.com<br>address: A street<br>class: 12D<br>english: 40e<br>math: 50 | A failure message display, no data is added to table or student.csv | As expected | Pass |

| | | | | | |
|---|---|---|---|---|---|
| | | ID:1<br>name: A edited<br>date of birth: 12/12/2000<br>phone number: 1234567890<br>email address: A@gmail.com<br>address: A street<br>class: 12D<br>english: 40<br>math: 50e | A failure message display, no data is added to table or student.csv | As expected | Pass |
| **5** | Search for student by name | name: A | All students whose name contains "A" is displayed on the table | As expected | Pass |
| **6** | Search for students by ID | ID:1 | The student whose ID is 1 is displayed | As expected | Pass |
| **7** | Delete all student | | All students are deleted from student.csv | As expected | Pass |
| **8** | Delete student | ID: 1 | The student whose ID is 1 is deleted | As expected | Pass |

TABLE 7 TEST PLAN

## 2. JUnit test

To make the testing done more easily and conveniently, Junit has been implemented to perform an array of multiple tests simultaneously.

To aid the test, some generator has been programmed to generate random data for each column.

```
// random string generator
    public String randomString(int length) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < length; i++) {
            sb.append((char) ((int) (Math.random() * 26)
+ 97));
        }
        return sb.toString();
    }

    // random email generator
    public String randomEmail(int length) {
        StringBuilder sb = new StringBuilder();
```

```
        for (int i = 0; i < length; i++) {
            sb.append((char) ((int) (Math.random() * 26)
+ 97));
        }
        return sb.toString() + "@gmail.com";
    }

    // random phone number generator that consists of
all integer
    public String randomPhone(int length) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < length; i++) {
            sb.append((int) (Math.random() * 10));
        }
        return sb.toString();
    }

    // random date of birth generator, date, month, year
are separated by "/"
    public String randomDob() {
        StringBuilder sb = new StringBuilder();
        int date = (int) (Math.random() * 30) + 1;
        int month = (int) (Math.random() * 12) + 1;
        int year = (int) (Math.random() * 100) + 1900;
        sb.append(date);
        sb.append("/");
        sb.append(month);
        sb.append("/");
        sb.append(year);
        return sb.toString();

    }
```

```
    // random grade generator
    public int randomGrade() {
        return (int) (Math.random() * 100) + 1;
    }
```

### 2.1. testRegisterStudent

The first implemented test is testRegisterStudent which constructs a new Student object. When a student is successfully registered, the Controller assigns the value “true” to the variable “isRegister”. The test read this variable and determine if the test passes.

```
@Test
    public void testRegisterStudent() {
        System.out.println("registerStudent");
        // if a message dialog saying "Register
successfully" is shown, then this test
        // is passed
        // student information include name, id, email,
phone number, address, dob,
        // mathGrade, englishGrade, classId
        String name = randomString(20);
        String id = randomString(20);
        String email = randomEmail(20);
        String phone = randomPhone(10);
        String address = randomString(20);
        String dob = randomDob();
        int mathGrade = randomGrade();
        int englishGrade = randomGrade();
        String classId = randomString(20);
        studentController instance = new
studentController();
        boolean expResult = true;
        instance.registerStudent(name, id, email, phone,
address, dob, mathGrade, englishGrade, classId);
```

```
        boolean result = instance.isRegister;
        assertEquals(expResult, result);
        // TODO review the generated test code and
remove the default call to fail.
    }
```

### 2.2. testGetStudentList

The second test is the test of the feature that reads a file and pass the data into a list of Students. The test then read the list, if the list is not empty, then it passes.

```
@Test
    public void testGetStudentList() {
        System.out.println("getStudentList");
        studentController instance = new
studentController();
        // student list is returned
        // ArrayList<Student> expResult = null;
        studentController.studentList.clear();
        ArrayList<StudentModel> result = null;
        instance.getStudentList();
        // if result is not null, then this test is
passed
        assertNotNull(studentController.studentList);

        // TODO review the generated test code and
remove the default call to fail.
        // fail("The test case is a prototype.");
    }
```

### 2.3. testCheckEmail

The next test is to see if the checkEmail function can properly validate an email formatted String. The test generate a random String followed by "@gmail.com". If the String follows the conventional email format of email@domain.domain2, then it returns a boolean value "true" and the test passes.

```
@Test
    public void testCheckEmail() {
        // check email with a random email
```

```
        System.out.println("checkEmail");
        String email = randomEmail(20);
        studentController instance = new
studentController();
        boolean expResult = true;
        boolean result = instance.checkEmail(email);
        assertEquals(expResult, result);
        // TODO review the generated test code and
remove the default call to fail.
    }
```

### 2.4. testCheckDate

The next test is to see if the checkDate function can properly validate a date formatted String. The test generate a random String in the format of dd/mm/yyyy. If the String follows the conventional format, then it returns a boolean value “true” and the test passes.

```
@Test
    public void testCheckDate() {
        // check date in the dd/mm/yyyy
        System.out.println("checkDate");
        String date = randomDob();
        studentController instance = new
studentController();
        boolean expResult = true;
        boolean result = instance.checkDate(date);
        assertEquals(expResult, result);
        // TODO review the generated test code and
remove the default call to fail.

    }
```

### 2.5. testCheckPhone

The next test is to see if the checkPhone function can properly validate a phone formatted String. The test generate a random String in the format of 10 integer digits. If the String follows the conventional format, then it returns a boolean value “true” and the test passes.

```
@Test
    public void testCheckPhone() {
        // check phone number with a random phone number
        System.out.println("checkPhone");
        String phone = randomPhone(10);
        studentController instance = new
studentController();
        boolean expResult = true;
        boolean result = instance.checkPhone(phone);
        assertEquals(expResult, result);
        // TODO review the generated test code and
remove the default call to fail.
    }
```

### 2.6. testCheckStudent

The next test is to see if the program can properly return a student when a user search for an ID. The test read the file student.csv, make an ArrayList of Student out of it and check the id column of the first row and calls for the function checkStudent(). If a student is found, then the function returns a “true” boolean value.

```
    @Test
    public void testCheckStudent() {
        // read the file student.csv
        // read the second column of the file as id
        // check if the id is in the student list
        // if the id is in the student list, then this
test is passed
        System.out.println("checkStudent");
        studentController instance = new
studentController();
        instance.getStudentList();
        String id =
studentController.studentList.get(1).getId();
        boolean expResult = true;
        boolean result = instance.checkStudent(id);
```

```
        assertEquals(expResult, result);
        // TODO review the generated test code and
remove the default call to fail.

    }
```

### 2.7. testReadStudentFromFile

The test testReadStudentFromFile() is to test the ability to read from a file and make a List of Student out of it. If the list is no longer null after reading the file, then the test passes

```
@Test
    public void testReadStudentFromFile() {
        // read the file student.csv
        // if student list is not null, then this test
is passed
        System.out.println("readStudentFromFile");
        studentController instance = new
studentController();
        instance.readStudentFromFile("student.csv",
instance.studentList);
        assertNotNull(studentController.studentList);
        // TODO review the generated test code and
remove the default call to fail.
    }
```

### 2.8. testDeleteStudent

This test uses deleteStudent() to delete a student then use findStudent to check for it in the file, if the deleted student is not found anymore, the function returns false and the test passes.

```
@Test
    public void testDeleteStudent() {
        // read the file student.csv, get an id from the
student list
        // delete the student with the id
```

```
        // if the student is deleted, then the student
when finding with id is null
        // and this test is passed
        System.out.println("deleteStudent");
        studentController instance = new
studentController();
        instance.getStudentList();
        String id =
studentController.studentList.get(1).getId();
        instance.deleteStudent(id);
        StudentModel result = instance.findStudent(id);
        assertNull(result);
        // TODO review the generated test code and
remove the default call to fail.
    }
```

### 2.9. testDeleteAllStudent

This test uses deleteAllStudent() to delete all data on the file and then compare the file to an empty list, if they are equal, then the test passes.

```
@Test
    public void testDeleteAllStudent() {
        // delete all student in the student list
        // if the student list is empty, then this test
is passed
        System.out.println("deleteAllStudent");
        studentController instance = new
studentController();
        instance.getStudentList();
        instance.deleteAllStudent();
        // expected result = studentList
        // studentList.clear
        ArrayList<StudentModel> expResult =
instance.studentList;
```

```
        expResult.clear();
        assertEquals(expResult,
studentController.studentList);
        // TODO review the generated test code and
remove the default call to fail.
    }
```

### 2.10. testSortStudentByGpaAscending

This test uses sortStudentByGpaAscending() to sort the list of students in the order of GPA ascending and then goes through the list comparing each pair of student's gpa. If the GPA of the next student is greater than the current one, the loop continues. If it keeps happening until the loop ends, it means that the list is in the correct order and the test passes.

```
@Test
    public void testSortStudentByGpaAscending() {
        // sort the student list by the order of gpa
ascending
        // check the gpa of student from first to last,
if it's in
        // ascending order, then this test is passed
        System.out.println("sortStudentByGpaAscending");
        studentController instance = new
studentController();
        instance.getStudentList();
        instance.sortStudentByGpaAscending();
        ArrayList<StudentModel> result =
instance.studentList;
        for (int i = 0; i < result.size() - 1; i++) {
            assertTrue(result.get(i).getGpa() <=
result.get(i + 1).getGpa());
        }
        // TODO review the generated test code and
remove the default call to fail.
    }
```

### 2.11. testSortStudentByGpaDescending

This test uses sortStudentByGpaDescending() to sort the list of students in the order of GPA ascending and then goes through the list comparing each pair of student's gpa. If the GPA of the next student is smaller than the current one, the loop continues. If it keeps happening until the loop ends, it means that the list is in the correct order and the test passes.

```
@Test
    public void testSortStudentByGpaDescending() {
        // sort the student list by the order of gpa
descending
        // check the gpa of student from first to last,
if it's in
        // descending order, then this test is passed
        System.out.println("sortStudentByGpaDescending")
;
        studentController instance = new
studentController();
        instance.getStudentList();
        instance.sortStudentByGpaDescending();
        ArrayList<StudentModel> result =
instance.studentList;
        for (int i = 0; i < result.size() - 1; i++) {
            assertTrue(result.get(i).getGpa() >=
result.get(i + 1).getGpa());
        }
        // TODO review the generated test code and
remove the default call to fail.
    }
```

## Junit tests results

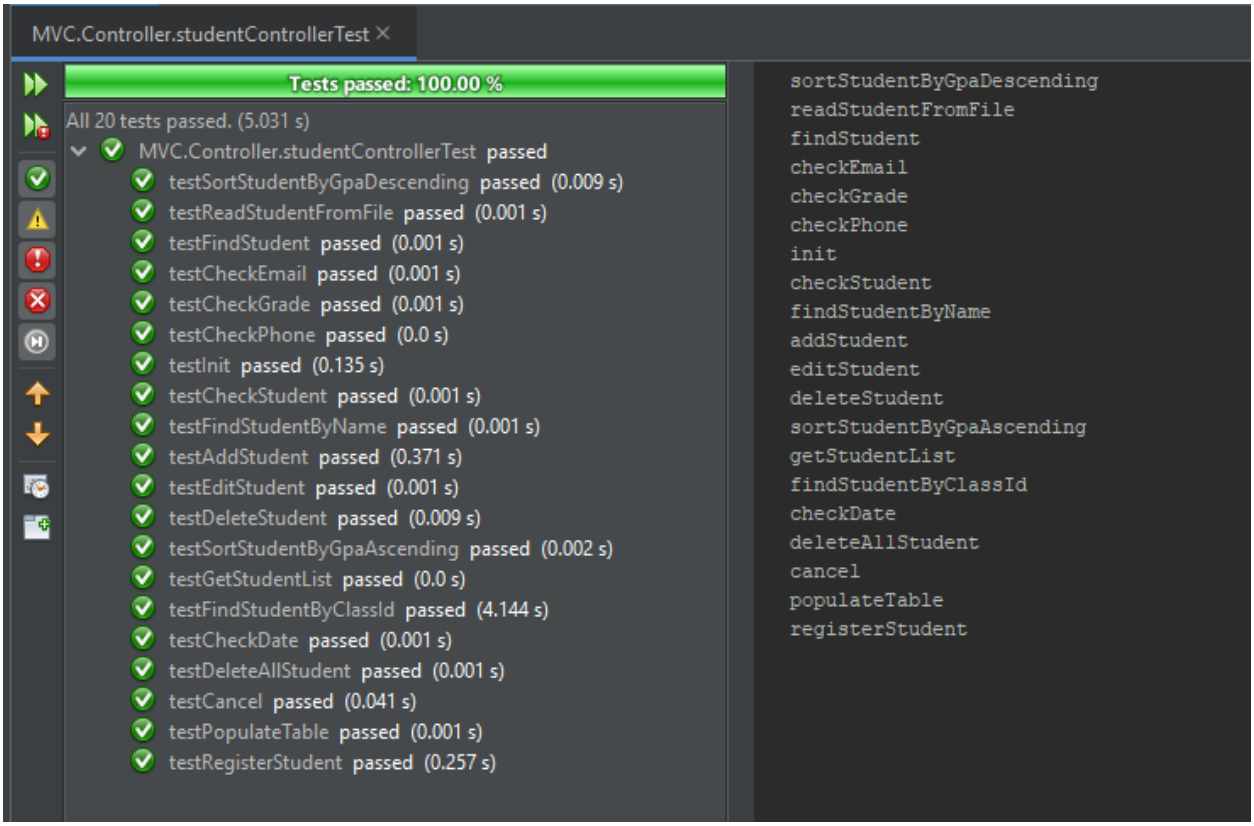


**FIGURE 8 THE TESTS RAN SUCCESSFULLY**