Project 1 - Testing Document

This document was made using the markdown editor, *Obsidian*. Along with a submitted testing file named "ProjectOneTests.java", there will be *several* code snippets illustrating the tests before a brief, *English* explanation. Unfortunately, obsidian lacks a time effective way to convert markdown files into 'pretty-looking' PDFs, so I apologize in advance for slightly strange formatting.

> **A Note About my Testing**
>
> The *entirety* of this project was done using Visual Studio Code. As a result, all the tests I had to conduct were done in a separate document that made use of JUnit and Microsoft's VS Code Extension called "Test Runner for Java". This extension is very well known, but it can be accessed through Test Runner for Java - Visual Studio Marketplace. Because I am using JUnit instead of an interactions pane, **there is no output for successful tests**. that being said, I will note exactly what the test is doing and what I am using for the assert... commands. I thoroughly tested *all* of my methods, and I will be sure to indicate where and how multiple methods are being tested in the same test.

Because the code tests are done in their own file, I had to import the correct packages to fully make use of this VS Code extension.

The code at the beginning of my document for imports looks like the following, where the expression .* means import all methods from package.

```
import static org.junit.Assert.*;
import org.junit.*;
```

"import static" is used to prevent repetitive static naming of my methods in the code. It allows me to use phrases like assertEquals and assertTrue/False.

> **Delta Values for 'double' Arguments in Assert**

All of my delta values in this document are equal to 0. This is because I do not want to allow any error in my tests, and this value is perfect for ensuring exact results.

Fields and Their Definitions

The following fields were created for ease of use and comparison in the upcoming tests:

```
private String accountInput = "12345";
private double currentBalance = 27.0;
private int balLimit = 104;
private double inputBookFine = 20.0;
private double inputReserveFine = 10.0;
private double inputRate = 0.1;
private double inputMonthly = 46.0;
private double inputPaid = 35.6;
private double emptyBal = 0.0;
private String studentName = "John";
private String studentAddress = "11447 Juniper";
```

- "accountInput" stores a sample account number for the student
- "currentBalance" stores a model balance for the student
- "balLimit" stores a balance limit for the student to have to start
- "inputBookFine" is a model amount a student might be charged per overdue book
- "inputReserveFine" is a model amount a student might be charged per overdue reserved item
- "inputRate" is a default amount for the interest rate
- "inputMonthly" is the default monthly payment amount
- "inputPaid" is the amount paid this month by the user
- "emptyBal" is a parameter I used to have a 0 balance field
- "studentName" default name for any student
- "studentAddress" default address for any student

Commenting my Fields

In the testing file, I omitted comments on all of my field variables to save time for myself as I planned on placing them in this document all along. In the actual Class files, however, you can see comments over each field explaining what they store.

Also, I made all fields private but access them *without* getter/setter methods in my Testing Document in order to save time!

Necessity of Java 8

As proof of the Java version I used, below you can find the Path to JAVA_HOME on my laptop as given through the command line.

```
$ java -version
openjdk version "1.8.0_392"
OpenJDK Runtime Environment Corretto-8.392.08.1 (build 1.8.0_392-b08)
OpenJDK 64-Bit Server VM Corretto-8.392.08.1 (build 25.392-b08, mixed
mode)
```

This also handles the heavy lifting of all the testing done below, if I'm not mistaken.

Account Class Tests

↖ Account Instance

Account account = new Account(accountInput, balLimit);

The above code was used to create an instance of the Account class in my testing file. It takes in two preassigned fields, where accountInput is a String storing "12345" and balLimit is an int storing 104. Only one of the constructors is tested here because this constructor calls the previous constructor to work properly, as it contains the phrase "this(...)".

Testing Account's dual-input constructor among other methods

```
@Test
public void accConstructorTest() {
        String num = account.getAccountNumber();
        assertEquals(accountInput, num);
        int predLimit = account.getBalanceLimit();
        assertEquals(balLimit, predLimit);
}
```

- This tests the "getAccountNumber" by comparing the inputted number that was in the constructor to the field that was sent through it. It is predicted that the getter method should yield the same number as the inputted number, "12345", and that is exactly the case.
- This tests the "setBalanceLimit" and "getBalanceLimit" methods in conjunction by comparing the inputted limit in the constructor to that returned by the "getBalanceLimit" method. The value of 104 was held by the 'inputLimit' field which was used for the "setBalanceLimit" method. Since that is the value received by the getter method, therefore this test was also successful.

Testing the "getBalance" and "setBalance" methods

```
@Test
public void balanceTest() {
        account.setBalance(currentBalance);
        double bal = account.getBalance();
        assertEquals(currentBalance, bal, 0);
}
```

- This test the "getBalance" method in conjunction with the "setBalance" method because I set the account's balance using the declared field in my testing document. Because I then got that value with the "getBalance" method, the value returned should be equal to the field in my tests. This was the case, as indicate by the lack of output from assertEquals.

Testing the "charge" method

```
@Test
public void chargeTest() {
```

```
        double actualCharge = account.getBalance() + 4;
        account.charge(4);
        double predCharge = account.getBalance();
        assertEquals(actualCharge, predCharge, 0);
}
```

- This tests the "charge" method by using the "getBalance" method, as I have already confirmed that it works as intended. I first found what the actual charge should be by adding 4 to the accounts balance and storing it in a variable called 'actualCharge'. Then, I used "account.charge(4)" to hopefully add 4 to the accounts balance. I checked if it worked by using the "getBalance" method once again, and knew the test was successful when assertEquals gave no output.

Testing the "credit" method

```
@Test
public void creditTest() {
        double actualCredit = account.getBalance() - 3.6;
        account.credit(3.6);
        double predCredit = account.getBalance();
        assertEquals(actualCredit, predCredit, 0);
}
```

- This tests the "credit" method in the exact same way as the "charge" method is tested. The only difference here is that I subtracted 3.6 from the accounts balance to ensure that the method works with Double values. After gathering my variable in the exact same way as the "charge" method, I then ran them both through assertEquals, which indicated a successful test run by giving no output.

LibraryAccount Class Tests

↖ **LibraryAccount Instances**

LibraryAccount simpleLibrary = new LibraryAccount(accountInput);
LibraryAccount libAccount = new LibraryAccount(accountInput, balLimit, inputBookFine, inputReserveFine);

The two instances above were created in the testing file in order to fully and thoroughly test the two constructors in this class. Because these constructors use the "super(...)" command to work properly, I chose to test both of them to ensure the correctness of my code.

Testing LibraryAccount's first constructor

```
@Test
public void libSimpleTest() {
        String libAccountNum = simpleLibrary.getAccountNumber();
        assertEquals(accountInput, libAccountNum);
}
```

- This tests the constructor with 2 inputs for the LibraryAccount Class. It makes use of the "getAccountNumber" method from the Account class by calling the method to store the account number in a variable named 'libAccountNum'. I then compared this value with the inputted value of "12345" and knew the test was successful by the lack of output from assertEquals.

Testing the large constructor as well as other methods in the class

```
@Test
public void libSignatureTest() {
        String libAccountNumber = libAccount.getAccountNumber();
        double bookFineAmount = libAccount.getBookFine();
        double reserveFineAmount = libAccount.getReserveFine();
        assertEquals(accountInput, libAccountNumber);
        assertEquals(balLimit, libAccount.getBalanceLimit());
        assertEquals(inputBookFine, bookFineAmount, 0);
        assertEquals(inputReserveFine, reserveFineAmount, 0);
}
```

- This tests the "getAccountMethod" once again by using the same procedure as the previous test. The way I tested this is nearly identical, with the only change change being the instance of the Class, which is called 'libAccount' here. The assertEquals call for these variables gave no output indicating a successful test.

- This tests the "getBalanceLimit" once again by comparing my inputted field containing the balance limit with the one returned by the method. Since the field has a value of 104, the "getBalanceLimit" should yield the same result. This is the case, as indicated by the lack of output from assertEquals.
- This tests the "getBookFine" as well as the "setBookFine" methods together as well. Since the "setBookFine" method is used when calling the constructor, we can predict that the book fine amount should be equal to the inputted amount of 20.0. Since the "getBookFine" method's assigned variable yielded no output when put through assertEquals, I can safely assume that the methods work.
- This tests the "getReserveFine" and "setReserveFine" methods together. The "setReserveFine" method is called when using the constructor. I predicted that the reserve fine should be equal to my inputted value of 10.0, and that is confirmed by assertEquals when it is given the value that contains the return from "getReserveFine".

Testing the "decrement" method branch 1

```
@Test
public void overdueDecrementTest() {
        libAccount.decrementOverdueBooks();
        int amountBook = libAccount.getNumberOverdueBooks();
        assertEquals(0, amountBook);
        libAccount.decrementOverdueReserve();
        int amountReserve = libAccount.getNumberOverdueReserve();
        assertEquals(0, amountReserve);
}
```

- This tests the first branch of the "decrementOverdueBooks" method, where I try to subtract from 0 books. This shouldn't be allowed, so the number of overdue books in the account should stay at 0. This is confirmed by using the "getNumberOverdueBooks" method and placing the outputted value in an assertEquals expression with the expected value being 0. The lack of output from assertEquals shows a successful test.
- This tests the first branch of the "decrementOverdueReserve" method in an identical way. Here, I used the "getNumberOverdueReserve" method to get the result of subtracting 1 from 0. Since this isn't allowed, the value should stay at 0, which is confirmed by the assertEquals method not giving an output.

Testing the 'else' branch of the "decrement" method along with the "increment" methods

```
@Test
public void overdueAccountTests() {
        int finalIncrease = (int)(Math.random() + 6);
        for (int i = 0; i < finalIncrease; i++) {
                libAccount.incrementOverdueBooks();
                libAccount.incrementOverdueReserve();
        }

        libAccount.decrementOverdueBooks();
        libAccount.decrementOverdueReserve();
        assertEquals(finalIncrease - 1, libAccount.getNumberOverdueBooks());
        assertEquals(finalIncrease - 1, libAccount.getNumberOverdueReserve());
}
```

- This tests the "increment" methods in conjunction by running them a predetermined, random amount of times. This is accomplished by using a for loop that runs 'finalIncrease' amount of times. 'finalIncrease' holds the random value between 1 and 6 generated by the 'Math.random() + 6', which is a built in class. I then tested the decrement methods by using them on both the overdue books and overdue reserve, where I then compared the "getNumberOverdueBooks" and "getNumberOverdueReserve" returned values with one less than the final increase. This works because I only ran the decrement methods once. The lack of output from assertEquals indicates a successful test.

Testing the "setter" methods for the overdue books and reserved items

```
@Test
public void setterOverdueTests() {
        libAccount.setNumberOverdue(29);
        assertEquals(29, libAccount.getNumberOverdueBooks());
        libAccount.setNumberOverdueReserve(113);
        assertEquals(113, libAccount.getNumberOverdueReserve());
}
```

- This tests the setter methods by using the predetermined accurate getter methods for the class. I set the amount of Overdue Books to 29 and used the getter method in conjunction with the assertEquals method to confirm its correctness. I did the same for the number of Overdue Reserved Items, except with the number 113. The lack of output from this JUnit test indicates a successful running of the code.

Testing the 'canBorrow' method

```
@Test
public void canBorrowTest() {
        assertTrue(libAccount.canBorrow());
        libAccount.charge(libAccount.getBalanceLimit() + 1);
        assertFalse(libAccount.canBorrow());
}
```

- This tests the 'true' branch of this method by using assertTrue. Because the initial value of the balance is less than the balance limit, the student should be able to borrow. This result is confirmed by the lack of output from assertTrue.
- This tests the 'false' branch of the method by using assertFalse. I accomplished this by charging the account a value 1 greater than the accounts limit. Since this would put the student's balance above the balance limit, I predicted a false output. This was confirmed by the lack of output from assertFalse.

Testing the "endOfDay" method

```
@Test
public void endOfDayTest() {
        for (int i = 0; i < 5; i++) {
                libAccount.incrementOverdueBooks();
                libAccount.incrementOverdueReserve();
        }
        double bookFine = libAccount.getNumberOverdueBooks() *
libAccount.getBookFine();
        double reserveFine = libAccount.getNumberOverdueReserve() *
libAccount.getReserveFine();
        double balanceTotal = bookFine + reserveFine + libAccount.getBalance();
        libAccount.endOfDay();
```

```
        assertEquals(balanceTotal, libAccount.getBalance(), 0);
}
```

- This test builds off of all of my previous testing techniques. I first increment the number of overdue books and overdue reserved items by 5 using a for loop. I then broke up the components of the actual "endOfDay" method into its individual components. As you can see above, I multiplied each of the fines by their respective amounts of the items, and added it to the current balance in an attempt to nullify any potential errors. I then got the new balance from the "getBalance" method and compared it to the manually calculated EOD total. I would expect these values to be equal, and that is confirmed by the lack of output from assertEquals.

CreditAccount Class Tests

↖ CreditAccount Instance

CreditAccount credAccount = new CreditAccount(accountInput, inputRate);

This creates a Creedit Account to be used through these tests with a given account number and interest rate.

Testing Interest Rate Getter-Setters and Constructor

```
@Test
public void interestRateTest() {
        credAccount.setInterestRate(inputRate);
        assertEquals(inputRate, credAccount.getInterestRate(), 0);
}
```

- Checks to see if the inputted interest rate in the constructor call was properly assigned with the setInterestRate. This is confirmed by the lack of output from assertEquals when comparing the input to the getInterestRate method.

Testing the montly payment Getter-Setters

```
@Test
public void monthlyPaymentTest() {
        credAccount.setMonthlyPayment(inputMonthly);
        assertEquals(inputMonthly, credAccount.getMonthlyPayment(), 0);
}
```

- This test sends the inputMonthly field through the setter method, which is why that field is the expected value when using the getter method. No output from code indicating a successful test run and proper methods.

Testing the amount paid getter setters

```
@Test
public void amountPaidTest() {
        credAccount.setAmountPaidThisMonth(inputPaid);
        assertEquals(inputPaid, credAccount.getAmountPaidThisMonth(), 0);
}
```

- This sends the inputPaid field through the setter method, making it the expected value in my JUnit test. This value is recieved by the getter method correctly, indicating correct code and a successful test.

Testing the overridden credit method

```
@Test
public void overCredit() {
        double predBal = credAccount.getBalance() - 10;
        double predAmountPaid = credAccount.getAmountPaidThisMonth() + 10;
        credAccount.credit(10);
        assertEquals(predBal, credAccount.getBalance(), 0);
        assertEquals(predAmountPaid, credAccount.getAmountPaidThisMonth(), 0);
}
```

- I first determined the expected output from the method using the pred... variables. Because I am crediting $10, the balance should decrease by 10 and the amount paid should increase by 10. This is exactly what happens as seen by the lack of error from assert.

Testing an unpaid endOfMonth method

```
@Test
public void endOfMonthIntTest() {
        credAccount.setMonthlyPayment(inputMonthly);
        credAccount.setAmountPaidThisMonth(inputMonthly - 1);
        double oldBal = credAccount.getBalance();
        double predBal = (credAccount.getInterestRate() *
credAccount.getBalance()) / 12 + oldBal;
        credAccount.endOfMonth();

        assertEquals(predBal, credAccount.getBalance(), 0);
        assertEquals(emptyBal, credAccount.getAmountPaidThisMonth(), 0);
        assertEquals(credAccount.getBalance(), credAccount.getMonthlyPayment(),
0);
}
```

- I first set the amount paid by the account to be one less tan the required monthly payment so that interest would be charged. Using the inputted interest rate and balance of the account before any amount was charged, I manually calculated the new amount that should be in the account. This amount was stored in the 'predBal' variable to be the value expected. I then used three assertions to check to see that the balance was charged the right amount, the amount paid this month was reset, and that the new monthly amount was set to the increased balance. All of these were correct and the tests ran without error.

Testing a paid endOfMonth method

```
@Test
public void endOfMonthTest() {
        credAccount.setMonthlyPayment(inputMonthly);
        credAccount.setAmountPaidThisMonth(inputMonthly);
        double oldBal = credAccount.getBalance();
        credAccount.endOfMonth();

        assertEquals(oldBal, credAccount.getBalance(), 0);
        assertEquals(emptyBal, credAccount.getAmountPaidThisMonth(), 0);
        assertEquals(credAccount.getBalance(), credAccount.getMonthlyPayment(),
```

```
0);
}
```

- I first set the amount paid by the account to be equal to the amount owed. Because this is the case, no interest should be charged and the balance should not change. Instead, the balance should become the new monthly payment amount, the amount paid should be zero, and the current balance should equal the balance before the method was used. This is the case as proved by the assertions with the aforementioned expected values.

StudentAccount Class Tests

↖ **StudentAccount Class Tests**

StudentAccount studentAccount = new StudentAccount(accountInput, studentName);
LibraryAccount testLibraryAccount = new LibraryAccount(accountInput, balLimit, inputBookFine, inputReserveFine);
CreditAccount testTuitionAccount = new CreditAccount(accountInput, inputRate);
CreditAccount testDiningAccount = new CreditAccount(accountInput, inputRate);

Using the fields mentioned at the beginning of this document, I created the main instance of the class I would be using for these tests called studentAccount. I then created the other test instances that I would use on and off throughout the following tests.

In the first half of these tests, I am very detailed about my creation and storage of local instances of the classes in my tests. In the latter half, however, I omitted these steps. This was mainly for my own sanity, as I was retyping the same thing too much when the same thing is happening every time.

Testing the Constructor and name getter-setters

```
@Test
public void constructorTest() {
```

```
        assertEquals(accountInput, studentAccount.getAccountNumber());
        assertEquals(studentName, studentAccount.getName());
}
```

- Like the other Classes, I tested the constructor first in order to ensure my code works perfectly with the initialized values. I knew I fed the studentName and accountInput in to the constructor, so those were my expected values. If the constructor works properly, it should also make use of the setName method. It does this correctly, and so does the getName method as seen by the lack of output from the Test case. The same can be said about the account number's test.

Testing the address getter-setters

```
@Test
public void addressTest() {
        studentAccount.setAddress(studentAddress);
        assertEquals(studentAddress, studentAccount.getAddress());
}
```

- This asserts that the expected/inputted address I used with the set method is the same one retrieved by the get method. These methods are confirmed to work because of the lack of failure in the test.

Testing the LibraryAccount getter-setters

```
@Test
public void libraryGetSet() {
        // This code runs before the rest in order to test the null
initialization and comparison technique I used
        LibraryAccount nullLibraryAccount = studentAccount.getLibraryAccount();
        assertTrue(nullLibraryAccount.getAccountNumber() == null);

        // This sets the class field to the correct value for testing
        studentAccount.setLibraryAccount(testLibraryAccount);

        // this creates a model library account to extract information to test
        LibraryAccount modelLibraryAccount =
studentAccount.getLibraryAccount();
```

```
        assertEquals(accountInput, modelLibraryAccount.getAccountNumber());
        assertEquals(balLimit, modelLibraryAccount.getBalanceLimit(), 0);
        assertEquals(inputBookFine, modelLibraryAccount.getBookFine(), 0);
        assertEquals(inputReserveFine, modelLibraryAccount.getReserveFine(),
0);
}
```

- I wanted to make sure that the null initialization works properly in the parent class, so I used assertTrue to confirm that the account name was equal to the null qualifier. I only made the original account number null because this account may or may not exist.
- I then set the library account in the class to the one I declared in this testing document section, and retrieved it with the getLibraryAccount method, saving it to a LibraryAccount variable. I then tested all of the values that were sent through the method and made sure they were equal to the original value I used to declare the field. The assertions here all ran without output, indicating a successful test and correct code.

Testing the TuitionAccount getter-setters

```
@Test
public void tuitionGetSet() {
        // This code runs before the rest in order to test the null
initialization and comparison technique I used
        CreditAccount nullTuitionAccount = studentAccount.getTuitionAccount();
        assertTrue(nullTuitionAccount.getAccountNumber() == null);

        // This sets the class field to the correct value for testing
        studentAccount.setTuitionAccount(testTuitionAccount);

        // This creates a model TuitionAccount to extract information and test
        CreditAccount modelTuitionAccount = studentAccount.getTuitionAccount();
        assertEquals(accountInput, modelTuitionAccount.getAccountNumber());
        assertEquals(inputRate, modelTuitionAccount.getInterestRate(), 0);
}
```

- I wanted to make sure that the null initialization works properly in the parent class, so I used assertTrue to confirm that the account name was equal to the null qualifier. I only made the original account number null because this account may or may not exist.

- I then set the tuition account using the corresponding method and saving it to a test variable using the get method. I tested to make sure these methods worked by comparing the expected value, the fields I used to create the instance fields, with the values received by the getter methods for those specific values. No output from assertions meant correct code.

Testing the DiningAccount getter-setters

```
@Test
public void diningGetSet() {
        // This code runs before the rest in order to test the null
initialization and comparison technique I used
        CreditAccount nullDiningAccount = studentAccount.getDiningAccount();
        assertTrue(nullDiningAccount.getAccountNumber() == null);

        // This sets the class field to the correct value for testing
        studentAccount.setDiningAccount(testDiningAccount);

        // This creates a model DiningAccount to extract information and test
        CreditAccount modelDiningAccount = studentAccount.getDiningAccount();
        assertEquals(accountInput, modelDiningAccount.getAccountNumber());
        assertEquals(inputRate, modelDiningAccount.getInterestRate(), 0);
}
```

- I wanted to make sure that the null initialization works properly in the parent class, so I used assertTrue to confirm that the account name was equal to the null qualifier. I only made the original account number null because this account may or may not exist.
- I then set the tuition account to the instance field I had in the testing file, and then used the get method to save it to a local variable. I then used this local variable and the associated getter methods and compared those with my expected field values that I originally used to create the instance field. No output from the assertions indicated correctness.

Testing the overridden getBalance method

```
@Test
public void balanceOverride() {
        // This sets all of the manipulatable instances of classes in the
parent class to usable values
        studentAccount.setLibraryAccount(testLibraryAccount);
```

```
        studentAccount.setTuitionAccount(testTuitionAccount);
        studentAccount.setDiningAccount(testDiningAccount);

        // This retrieves all of the previously mentioned instances
        LibraryAccount balanceLibraryAccount =
studentAccount.getLibraryAccount();
        CreditAccount balanceTuitionAccount =
studentAccount.getTuitionAccount();
        CreditAccount balanceDiningAccount = studentAccount.getDiningAccount();

        double balance = 20;
        balanceLibraryAccount.setBalance(balance);
        balanceDiningAccount.setBalance(balance);
        balanceTuitionAccount.setBalance(balance);

        // This prepares variables for the actual JUnit test
        double actualBalanceSum = studentAccount.getBalance();
        double expectedBalanceSum = balance * 3 - 20;

        assertEquals(expectedBalanceSum, actualBalanceSum, 0);
}
```

Note: because I already tested the way I compared and created null accounts, I did not test all 5 (?) possible combinations of accounts existing or not, as it would be redundant and a waste of time

- First I created 3 different local instances of the different account types using the getter-setter technique I used the the previous tests. I then used the setBalance method to set each account to 20. Because I know the default balance for any instance of the account class is 20 and this method subtracts this number from the sum of the sub-accounts balances, I subtracted 20 from the sum of all 3 accounts (3×20) to get the expected balance. I then retrieved the actual balance by running the overridden getBalance method and saved it to a local variable. I compared this result with the expected balance. The assertion here had no output, indicated that the code runs as expected.

Testing the if branch of the overridden charge method

```
@Test
public void chargeOverrideIf() {
```

```
        studentAccount.setTuitionAccount(testTuitionAccount);
        CreditAccount chargeTuitionAccount =
studentAccount.getTuitionAccount();

        chargeTuitionAccount.charge(10);
        double expectedTuition = 30;
        assertEquals(expectedTuition, chargeTuitionAccount.getBalance(), 0);
}
```

*Note: The charge method in this class is said to **only** apply to the Tuition Account*

- I first tested the possibility of a Tuition Account existing and the summed balance (overridden getBalance method) - the amount to be charged being positive. If this is the case, the charge should increase the accounts balance by 10, from the original 20. This gave me the expected value of 30, which was correct as shown by the lack of output from the assertion on the last line.

Testing the else branch of the overridden charge method

```
@Test
public void chargeOverrideElse() {
        double chargeReal = 120;
        double expectedBalance = 20;
        studentAccount.setBalance(expectedBalance);
        assertEquals(-20, studentAccount.getBalance(), 0);
        studentAccount.charge(chargeReal);
        assertEquals(-140,studentAccount.getBalance(),0);
}
```

- If either a Tuition Account does not exist or the difference mentioned above is zero, then the second branch of the conditional is used. I only tested the possibility of a tuition account not existing, because I knew that that would also mean that there was a 0 balance in all of the accounts, as well as guaranteeing that the else branch would run.
- I tested an intermediate getBalance too make sure the latter half of this test would be successful. Because the default balance of the parent account is 20 and no other accounts exist, the getBalance method should result in a negative balance, in this case -20, as seen by the first assertion.

- I then tested the charge override by charging 120 dollars to the account, which should result in the negation of the difference between the balance found in the bullet point above this one and the amount wished to be charged. This sets the balance of the parent account to 140 (the refund balance), but the getBalance() command negates it for the reason mentioned above. That is why the expected value is -140, which is confirmed by the lack of output from from the assertion.

Below are all of the tests related to the most complicated method of the entire project, the overridden credit method

---

About these tests...

1. The Tuition Tests do not have a no credit statement becuase there is no possibility where a non-positive value is accepted
2. I added an error message in a conditional to account for the point mentioned above
3. While these may be repetitive in their wording and code, it is what I felt was necessary to cover as many possibilities as possible, as the specific branches of the method impact one another, unlike the ones in getBalance()
4. I did not test the overflow of the spare credit until the very end as to not make my life more comflicated
5. I did not test the possibilities of having combinations of null fields because there would be way too many tests I would have to run, and I am already certain that my teqchnique for checking to see if an account exists is correct, as seen in my previous tests.

Testing the error message of the method

```
@Test
public void creditError() {
        studentAccount.credit(-10);
}
```

- I confirmed this test works by running it in VS Code and checking the debug console, where I was met with the error message I set in the parent class using System.err.println(...).

Testing the If branch of the tuition branch only

```
@Test
public void tuitionTestIf() {
        studentAccount.setTuitionAccount(testTuitionAccount);
        CreditAccount creditTuitionAccount =
studentAccount.getTuitionAccount();
        creditTuitionAccount.setAmountPaidThisMonth(10);
        creditTuitionAccount.setMonthlyPayment(20);

        studentAccount.credit(30);
        assertEquals(20, creditTuitionAccount.getAmountPaidThisMonth(), 0);
}
```

- I first set a created TuitionAccount to have a monthly payment of 20 while only having paid 10 of it. This meant that a credit of 30 would result in 10 of the credit being used to increase the amount paid by 10, meeting the required monthly payment amount. This was confirmed using the assertion seen in the final line of the code above, where I made sure the amount paid this month was equal to my expected value of 20. This was the case.

Testing the else if branch of the tuition branch only

```
@Test
public void tuitionTestElseIf() {
        studentAccount.setTuitionAccount(testTuitionAccount);
        CreditAccount creditTuitionAccount =
studentAccount.getTuitionAccount();
        creditTuitionAccount.setAmountPaidThisMonth(10);
        creditTuitionAccount.setMonthlyPayment(20);

        studentAccount.credit(5);
        assertEquals(15, creditTuitionAccount.getAmountPaidThisMonth(), 0);
}
```

- The other branch of this conditional is the possibility of not crediting enough to pay the monthly payment fully. In this case, the full credit should be used to add to the amount paid towards the monthly payment. I used the same monthly payment and amount paid as before,

but this time I only credited 5. This should mean that the amount paid this month should be exactly 15, no more no less. This was confirmed by the assertion not throwing an error.

Testing the If branch of the dining branch only

```
@Test
public void diningAccountIf() {
        studentAccount.setDiningAccount(testDiningAccount);
        CreditAccount creditDiningAccount = studentAccount.getDiningAccount();
        creditDiningAccount.setAmountPaidThisMonth(10);
        creditDiningAccount.setMonthlyPayment(20);

        studentAccount.credit(30);
        assertEquals(20, creditDiningAccount.getAmountPaidThisMonth(), 0);
}
```

- I first set a created DiningAccount to have a monthly payment of 20 while only having paid 10 of it. This meant that a credit of 30 would result in 10 of the credit being used to increase the amount paid by 10, meeting the required monthly payment amount. This was confirmed using the assertion seen in the final line of the code above, where I made sure the amount paid this month was equal to my expected value of exactly 20. This was the case.

Testing the else if branch of the dining branch only

```
@Test
public void diningAccountElseIf() {
        studentAccount.setDiningAccount(testDiningAccount);
        CreditAccount creditDiningAccount = studentAccount.getDiningAccount();
        creditDiningAccount.setAmountPaidThisMonth(10);
        creditDiningAccount.setMonthlyPayment(20);

        studentAccount.credit(5);
        assertEquals(15, creditDiningAccount.getAmountPaidThisMonth(), 0);
}
```

- The other branch of this conditional is the possibility of not crediting enough to pay the monthly payment fully. In this case, the full credit should be used to add to the amount paid towards the monthly payment. I used the same monthly payment and amount paid as before,

but this time I only credited 5. This should mean that the amount paid this month should be exactly 15, no more no less. This was confirmed by the assertion not throwing an error.

Testing the 'no credit' branch of the dining branch only

```
@Test
public void diningAccountNone() {
        studentAccount.setDiningAccount(testDiningAccount);
        CreditAccount creditDiningAccount = studentAccount.getDiningAccount();
        creditDiningAccount.setAmountPaidThisMonth(10);
        creditDiningAccount.setMonthlyPayment(20);

        studentAccount.credit(0);
        assertEquals(10, creditDiningAccount.getAmountPaidThisMonth(), 0);
}
```

- Lastly, there is a possibility that there is not available credit to pay off anything in the dining account. If that is the case, 0 would technically be credited and the amount paid this month should not change at all. This is confirmed by the assertion at the end throwing an error when I expected the same amount paid as the value I gave the instance initially.

Testing the if branch of the library branch only

```
@Test
public void libraryAccountIf() {
        studentAccount.setLibraryAccount(testLibraryAccount);
        LibraryAccount creditLibraryAccount =
studentAccount.getLibraryAccount();
        creditLibraryAccount.setBalance(20);

        studentAccount.credit(30);
        assertEquals(0, creditLibraryAccount.getBalance(), 0);
}
```

- For the library account, there is no amount to be paid in a month, just a balance value. To test the credit method on this account specifically, I set the balance to 20 and credited 30. This made sure that the credit would not reduce the balance below zero. Because I credited more

than enough, the balance was set to zero as confirmed by the assertion with 0 as my expected value.

Testing the else if branch of library dining branch only

```
@Test
public void libraryAccountElseIf() {
        studentAccount.setLibraryAccount(testLibraryAccount);
        LibraryAccount creditLibraryAccount =
studentAccount.getLibraryAccount();
        creditLibraryAccount.setBalance(20);

        studentAccount.credit(15);
        assertEquals(5, creditLibraryAccount.getBalance(), 0);
}
```

- The other branch of this conditional is the possibility of not crediting enough to pay the balance fully. In this case, the full credit should be used to subtract from the balance. I used the same balance as before, but this time I only credited 15. This should mean that the remaining balance in the account should be exactly 5. This was confirmed by the assertion not throwing an error.

Testing the 'no credit' branch of the library branch only

```
@Test
public void libraryAccountNone() {
        studentAccount.setLibraryAccount(testLibraryAccount);
        LibraryAccount creditLibraryAccount =
studentAccount.getLibraryAccount();
        creditLibraryAccount.setBalance(20);

        studentAccount.credit(0);
        assertEquals(20, creditLibraryAccount.getBalance(), 0);
}
```

- Lastly, there is the possibility of not having enough credit to impact the balance of the library account. If this is the case, it is practically like crediting 0, where the balance will not change

at all. This meant that the balance should remain at 20, as shown by the assertion not throwing an error upon running of the test.

Testing Possibility of all accounts existing and having spare credit

```
@Test
public void testAllBranches() {
        // This preps the original balance for the final test
        double originalBalance = studentAccount.getBalance();

        // This sets a TuitionAccount with reasonable values
        studentAccount.setTuitionAccount(testTuitionAccount);
        CreditAccount creditTuitionAccount =
studentAccount.getTuitionAccount();
        creditTuitionAccount.setAmountPaidThisMonth(10);
        creditTuitionAccount.setMonthlyPayment(20);

        // This sets a DiningAccount with reasonable values
        studentAccount.setDiningAccount(testDiningAccount);
        CreditAccount creditDiningAccount = studentAccount.getDiningAccount();
        creditDiningAccount.setAmountPaidThisMonth(10);
        creditDiningAccount.setMonthlyPayment(20);

        // This sets a LibraryAccount with reasonable values
        studentAccount.setLibraryAccount(testLibraryAccount);
        LibraryAccount creditLibraryAccount =
studentAccount.getLibraryAccount();
        creditLibraryAccount.setBalance(20);

        studentAccount.credit(1000);

        // This tests to see if the money want through all accounts
        assertEquals(20, creditTuitionAccount.getAmountPaidThisMonth(), 0);
        assertEquals(20, creditDiningAccount.getAmountPaidThisMonth(), 0);
        assertEquals(0, creditLibraryAccount.getBalance(), 0);

        // This tests the final conditional that adds the remaining credit to
the account itself
```

```
        double remainingCredit = 1000 - (10 + 10 + 20);

        assertEquals(-(remainingCredit + originalBalance),
studentAccount.getBalance(), 0);
}
```

*Note: the value in the expected position is negated because the instructions say that the overridden "getBalance" method should subtract the current refund amount, or the parent class's balance, from the summation of the balances in the rest of the class instances. Because the initial value is 0 and I am crediting all of the money I possibly can the ending balance in the class instances will be 0, leaving me with the negated remainder of the credit plug the original balance of the account itself. **This value is not representative of the actual field value, just the getBalance output***

- The first half of this test is the combination of all of the initializations from the above tests in this section. Although there is a lot of text, it gets the job done.
- After all of the instances have been created with appropriate amounts needing to be paid off, I credited more than enough money (1000). This would mean the total amount paid this month should be equal to the monthly payment (20) for the Tuition and Dining Accounts. It also meant the balance of the library account should be 0. These were all the case as shown by the block of assertions that ran without throwing an error.
- Finally, I tested to make sure the remaining credit was actually going into the accounts balance. The logistics of this can be found italicized under the code block here. I used an assert statement here to compare the original balance, as found by the variable at the top of this test, with the getBalance output. The lack of output from the assertion here indicated proper code.

That concludes the testing portion of this project

```
Again, I appologize for funky formatting and length
```