CPSC 2151

Lab 6

Due: Tuesday, Oct. 31st && Thursday, Nov. 2nd **BEFORE LAB**

In this lab, you will work with Test Driven Development to complete a mortgage application system. You are provided with a `TestMortgage` class, which will provide several JUnit test cases for the `Mortgage` class. We do not have test cases for each method, just the "difficult" methods. We will discuss JUnit and how to create JUnit test cases in more detail in class, but this will be a good introduction. Every test case in JUnit is a method. Running the JUnit class will run all the test cases and report which one failed. If you have any failed test cases, you can go to that specific method and see what input was used to see why your program failed.

This process is called Test Driven Development because we develop our test cases first, and we can tell when we have completed the code correctly by whether or not all of our test cases pass. If you think you correctly completed the method, you can run the test suite and see if the test cases pass as you work on the code. **Note:** To test some of these methods, it is assumed that you have completed simpler methods such as the constructor and the `get` methods. Those methods will be called, as well as the method being tested. Also, JUnit does not allow us to call private methods directly, so we can only test those indirectly through publicly available methods. Luckily, all the test cases have been provided for you in this lab.

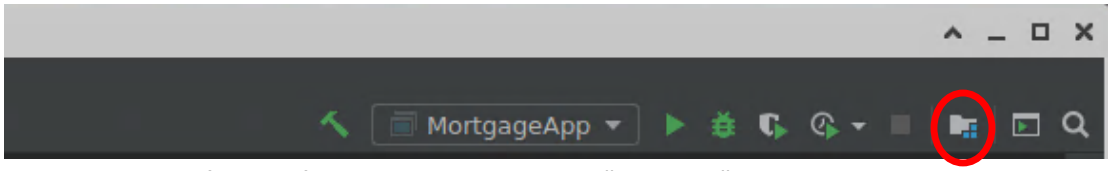
Provided Files:

- `IMortgage.java` – The interface for our `Mortgage` object. This file is completed.
- `AbsMortgage.java` – An abstract class that implements `IMortgage`. It provides an override to `toString()`. This `toString()` method is used for some comparisons in our JUnit test cases. Do not make any changes to this class.
- `ICustomer.java` – The interface for our `Customer` object. This file is completed.
- `AbsCustomer.java` – An abstract class that implements `ICustomer`. It provides an override to `toString()`. Additionally, it provides one `protected` field called `loan`. `loan` is an `IMortgage` object to hold the mortgage the customer is applying for. It is `protected` and not `private` because a `private` field is not accessible to our subclasses. While we often will not put any data fields in an abstract class, in this case, we know that any customer will need an `IMortgage` object. Do not make any changes to this class.
- `Customer.java` – A class that extends `AbsCustomer` and implements `ICustomer`. This class is completed for you and does not require any changes.
- `TestMortgage.java` – This is our JUnit test file for the `IMortgage` interface. It will use the `ICustomer` interface as well but will not test it. You should not make any changes to this file.
- `MortgageApp.java` – A simple driver for our `IMortgage` and `ICustomer` interfaces. This is provided just to have a simple `main` method to run your code when not using the JUnit test cases. You do not need to make any changes to this file.
- `Makefile` – the `Makefile` to run this JUnit code on the SoC Unix machines. **Note**: this will only work on an SoC Unix machine since it refers to a specific version and location for JUnit. You should only edit this file if you run into issues with tab and end-of-line encoding.

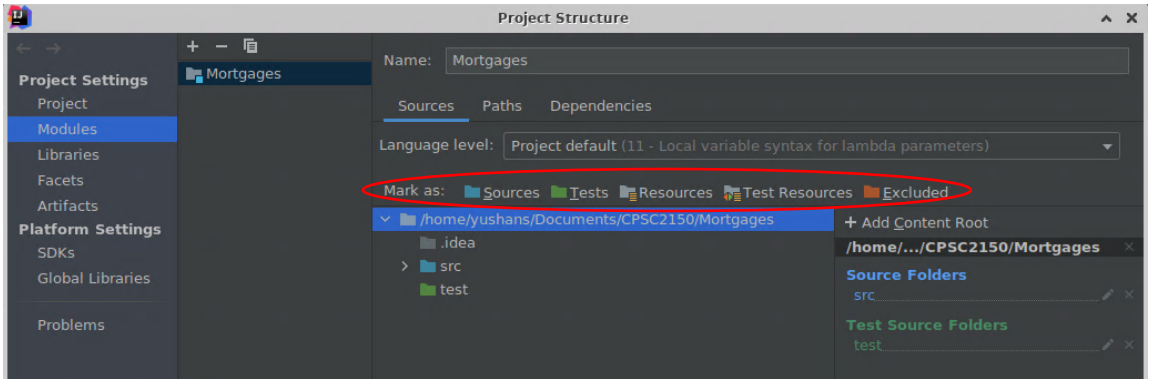
Setup:

Please follow the setup instructions carefully to ensure that JUnit is set up correctly. You will need to perform some setup to use JUnit in your IntelliJ project:

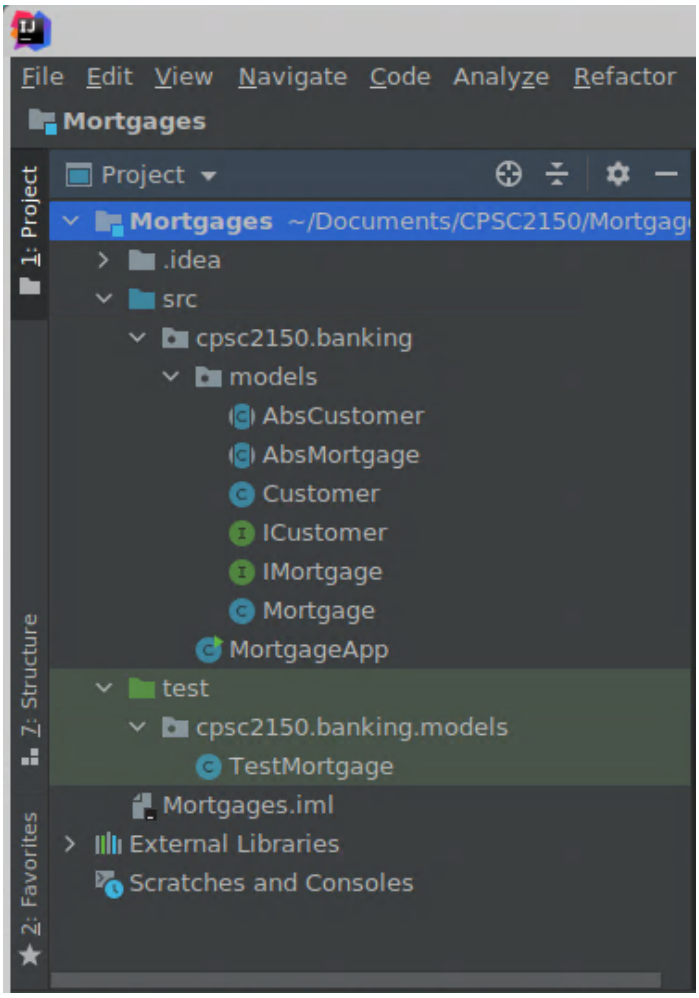
1. Start by creating a new project called `Mortgages` and then add a package called `cpsc2150.banking.models`.
2. Add your `IMortgage`, `AbsMortgage`, `ICustomer`, `AbsCustomer`, and `Customer` code files to the `cpsc2150.banking.models` package. Add your `MortgageApp` code file to the `cpsc2150.banking` package, so it is just outside of the `models` folder.
3. Create a new build that points to the `main` function in `MortgageApp`.
4. In your navigation panel on the left side of the screen, right-click on the project folder. This should be the highest level folder you can see, and it should be called `Mortgages`. Right-click on it and select **New->Directory**. Name your new directory "`test`". This directory should appear in your navigation pane below your source folder.
5. Now click on the project structure button on the top right corner of the Intellij window. It should be right next to your search icon.



   a. On the left side of the new window, select "Modules"
   b. If it is not already on the "Sources" tab, select that tab. You should now see a list of your folders, such as `src` and `test`.
   c. Select the test folder. Above the list of folders is a "Mark as" option. Select the "Tests" option to mark your `test` folder as a repository for tests. Your `test` folder should turn green.

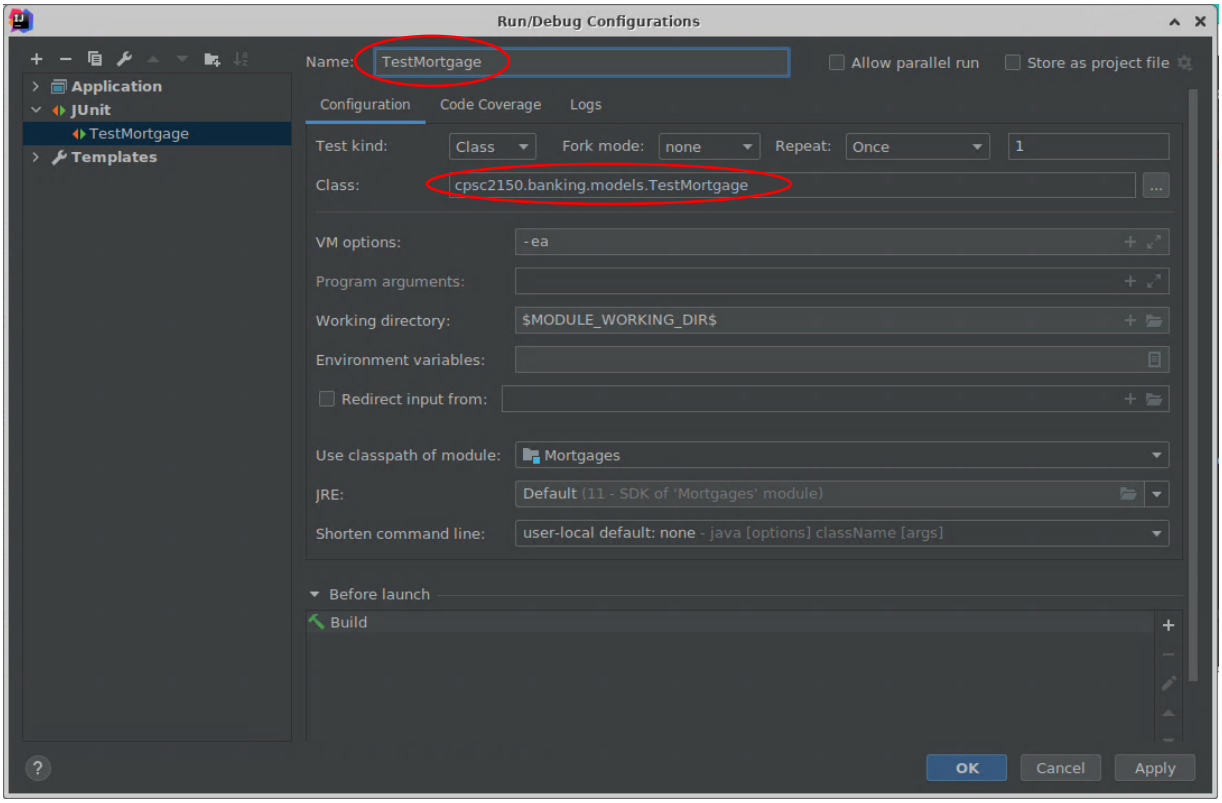


   d. Hit OK to exit the Project Structure window
6. In the navigation pane, right-click on your test folder, and add a package to it. Use the same package name as the `src` folder (`cpsc2150.banking.models`)
7. Add the `TestMortgage` class to the new package in the `test` folder. It's a good practice to name your JUnit classes `Test<CLASSNAME>` to keep our naming consistent. Your file structure should look like this:



8. IntelliJ will likely have an issue with the `@Test` annotation (among other things) in `TestMortgage`. This is because it doesn't know we are trying to use JUnit. Click on the `@Test` annotation and hit alt + enter. If one of the options is to add JUnit4 to the classpath, congratulations! You can select that option and skip ahead to step 10. I suspect that won't happen.
9. Assuming you were not able to add JUnit in the previous step, follow these instructions:
    a. Open the Project Structure window again
    b. Select "Libraries" on the left
    c. Click on the green plus sign to add a new library, select "From Maven"
    d. A search box will appear. Type in "junit:junit:4.12" and hit search
    e. One of the options from the drop-down should be junit:junit:4.12. Select that one and hit OK
        i. It's OK if you use any number after the 4. It just needs to be JUnit 4, not 3 or 5. However, the SoC Unix has 4.12 installed.
    f. Select to add it to the current project
    g. Hit OK at the bottom of the Project Structure window
    h. It should now recognize your JUnit statements
    i. If it does not, select `@Test` with your cursor, and alt + enter should now add the JUnit statements.

10. We should now have our test cases set up, but we still need to add a configuration to run the JUnit tests
11. Click on your configuration selector (top right corner of the screen, next to the run button) and select Edit Configuration
12. Click on the plus icon to add a new configuration, and select the type as JUnit
13. In the menu that pops up, calls your new configuration `TestMortgage`
14. In the class option, use the selector (button with three dots) to select the `TestMortgage` class



15. Hit OK to save this new configuration. You can't run your new configuration yet, because you have not written the code.
16. Create a new class (in the package in the `src` directory) called `Mortgage.java` that extends `AbsMortgage` and implements `IMortgage`. Complete the class as specified in `IMortgage.java`
17. Once your `Mortgage` class is complete, you can now run your JUnit configuration. You hope to see a green bar saying all of your tests have passed. If not, it will tell you the name of the tests that failed. Those failed test cases should give you an indication of where the fault is in your code. When you have the green bar saying all test cases have passed, you have completed the code.

Mortgage.java

`Mortgage.java` is the class you need to add to complete this assignment. `Mortgage.java` needs to fully implement the `IMortgage` interface and extend the `AbsMortgage` class. The `IMortgage` interface specification should give you some idea of how to implement the class, but we have some more information here. **Remember to add in your invariants and correspondences.**

- You need to provide a constructor for `Mortgage` that takes in the cost of the home (`double`), the down payment (`double`), the number of years (`int`), and the customer (`ICustomer`). `TestMortgage` calls the constructor, so the arguments need to be in that order. Use these values to compute the concepts in the **Defines** clause of `IMortgage`. **Remember to write contracts for this constructor!**
- The Annual Percentage Rate (APR), which is different from our concept of `Rate`, is calculated as follows:
    - Start with the Base APR of 2.5%
    - If the loan is for less than 30 years, add 0.5%; otherwise, add 1%
    - If the percent down is not at least 20%, add 0.5% to the APR
    - The customer's credit score also adds to the rate according to the following table:

Credit is:	Credit score range	Add to the APR
Very Bad	< 500	10 %
Bad	500 <= score < 600	5%
Fair	600 <= score < 700	1%
Good	700 <= score < 750	0.5%
Great	750 <= score <= 850	0%

    - So a customer with Fair credit applying for a 20 year mortgage with at 10% down will have an APR of: 2.5% + 1% + 0.5% + 0.5% = 4.5% APR
- Our Principal amount for the loan is the cost of the house minus the down payment.
- The Debt to income ratio is the debt payments (over a period of time) divided by the income (over the same period of time). The debt payments should include the payments for the mortgage itself.
- The Percent Down is the percentage of the house's cost that is covered by the down payment for the home.
- The monthly payments for the loan are = $\frac{monthly\ interest\ rate * principal}{1 - (1 + monthly\ interest\ rate)^{-(total\ number\ of\ payments)}}$
- A loan will be rejected if:
    - The APR is greater than or equal to 10% OR
    - The Percent Down is less than 3.5% OR
    - The Debt to Income Ratio is greater than 40%
- You may write private helper functions inside the `Mortgage` class if you want, but you should not add any public methods not specified in the interface.

General requirements and tips

- Remember our best practices we've discussed in class
- No magic numbers, use `public static final` variables
- Make sure you follow the contracts provided
- Remember to comment your code. Javadoc comments and contracts are a good start and maybe enough for more straightforward methods but comment more complicated parts when needed.

Running JUnit on Unix Machines

Set up the correct directory structures on one of the School of Computing Unix machines (`cpsc2150/banking/models`). Upload all your code files into the proper directories. Remember to

go into the "`test`" directory that IntelliJ created and add the `TestMortgage.java` file as well. While IntelliJ kept these in separate directories, we will store them in one directory on Unix (to keep our `makefile` more straightforward). Place the `makefile` outside of the package directory. You should be able to run the `MortgageApp` program by using the `make` and `make run` commands. You can run the JUnit code by using the `make test` and make `runtest` commands. The `make clean` command should delete any `.class` files.

Partners

You are required to work in teams of 3-4 students on this lab assignment. Make sure you include all partners' names on the submission. You only need to submit one copy. Remember that working with a partner means working *with* a partner, not dividing up the work. You will need this code for a later lab, so make sure both partners have a copy of it.

Before Submitting

You should make sure your `MortgageApp` program and your JUnit code will run on SoC Unix before you submit. Make sure you correctly set up the directory structure to match the package name. You are provided with a `makefile` that should be included.

Submitting to Gradescope

Upload your `Lab6.zip` to Gradescope. It should automatically check if you have submitted everything correctly and verify that it was able to compile and test your code. While not all test cases are visible, you should make sure that all visible test cases pass. The visible test cases should be a good starting point for checking whether your solution is correct or not.

Only one member per group needs to submit the assignment, but you need to make sure to select your partner's name when creating a submission on Gradescope.

NOTE: Make sure you zipped up your files correctly and didn't forget something! Always check your submissions on Gradescope to ensure you uploaded the correct zip file and it is in the proper format that it expects. Notify your course instructor immediately of any Gradescope issues.