

CPSC 2151

Lab 7

Due: Friday, October 9th at 10:00 pm

In this lab you will be working 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 function. When you run the JUnit class, it will run all the test cases, and report which one failed. If you have any failed test cases, you can go to that specific function, and see what input was used to see why your program failed.

This is what’s called Test Driven Development because we developed 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. As you work on the code, if you think you have the function correctly completed, you can run the test suite and see if the test cases for that function pass. **Note:** in order to test some of these functions, it is assumed that you have completed simpler functions such as the constructor and the `get` functions. Those functions will be called as well as the function being tested. Also, JUnit does not allow us to directly call private functions, so we can only test those indirectly through publicly available functions. Luckily, all the test cases have been provided for you in this lab.

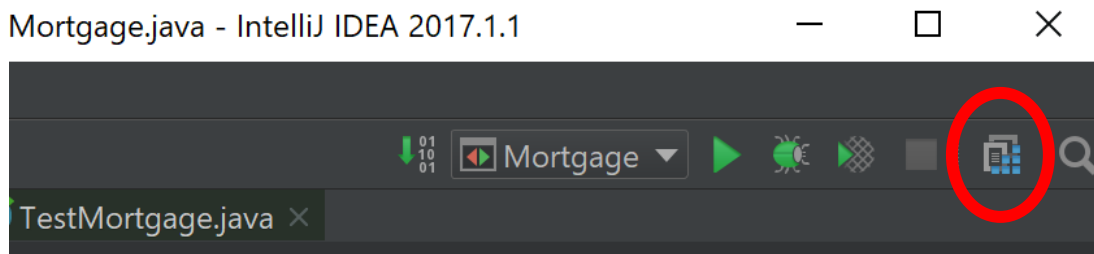
Provided Files:

- `IMortgage.java` – The interface for our `Mortgage` object. This file is completed.
- `AbsMortgage.java` – An Abstract class for 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 you can use 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 a SoC Unix machine, since it is referring to a specific version and location on for JUnit. You should only edit this files if you run into issues with tab and end of line encoding.

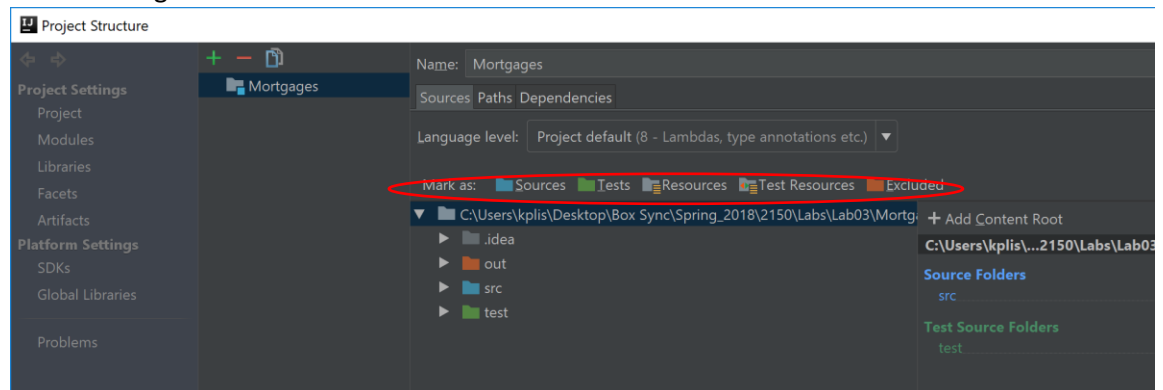
Setup:

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

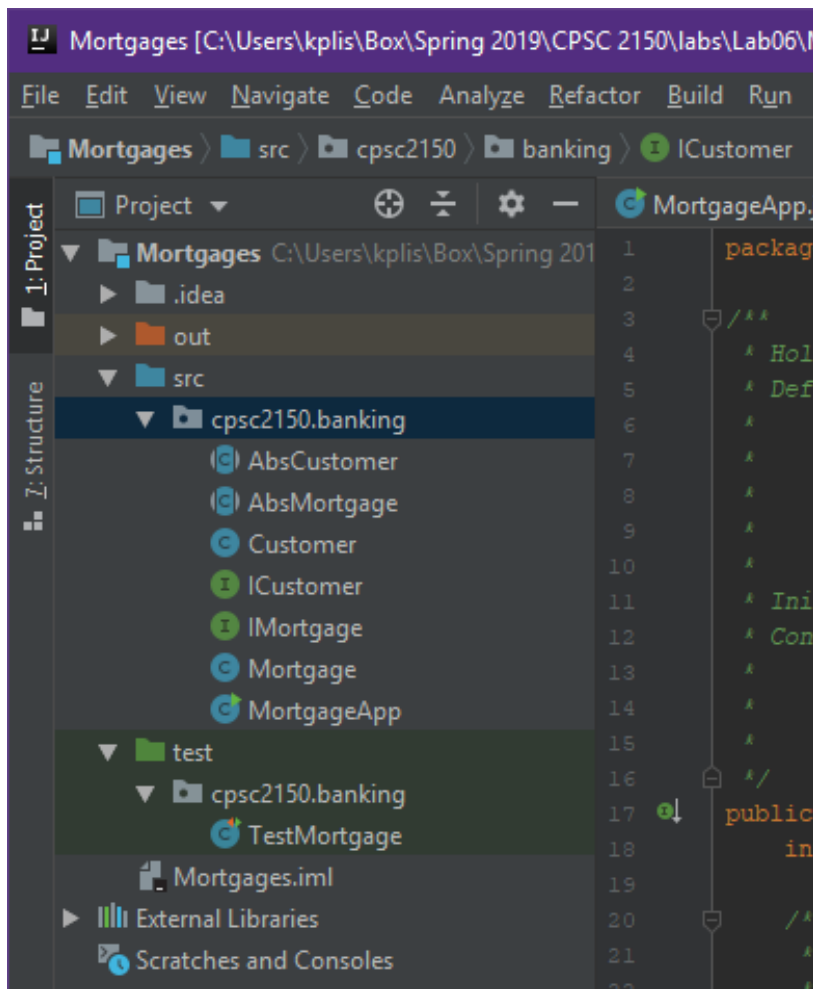
1. Start by creating a new project called `Mortgages`, and adding a package called `cpssc2150.banking` to it.
2. Add your `IMortgage`, `AbsMortgage`, `ICustomer`, `AbsCustomer`, `Customer`, and `MortgageApp` code files to the `cpssc2150.banking` project.
3. Create a new build that points to the main 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 Source As” option. Select the “Tests” option to mark your `test` folder as a repository for tests. Your `test` folder should turn green.

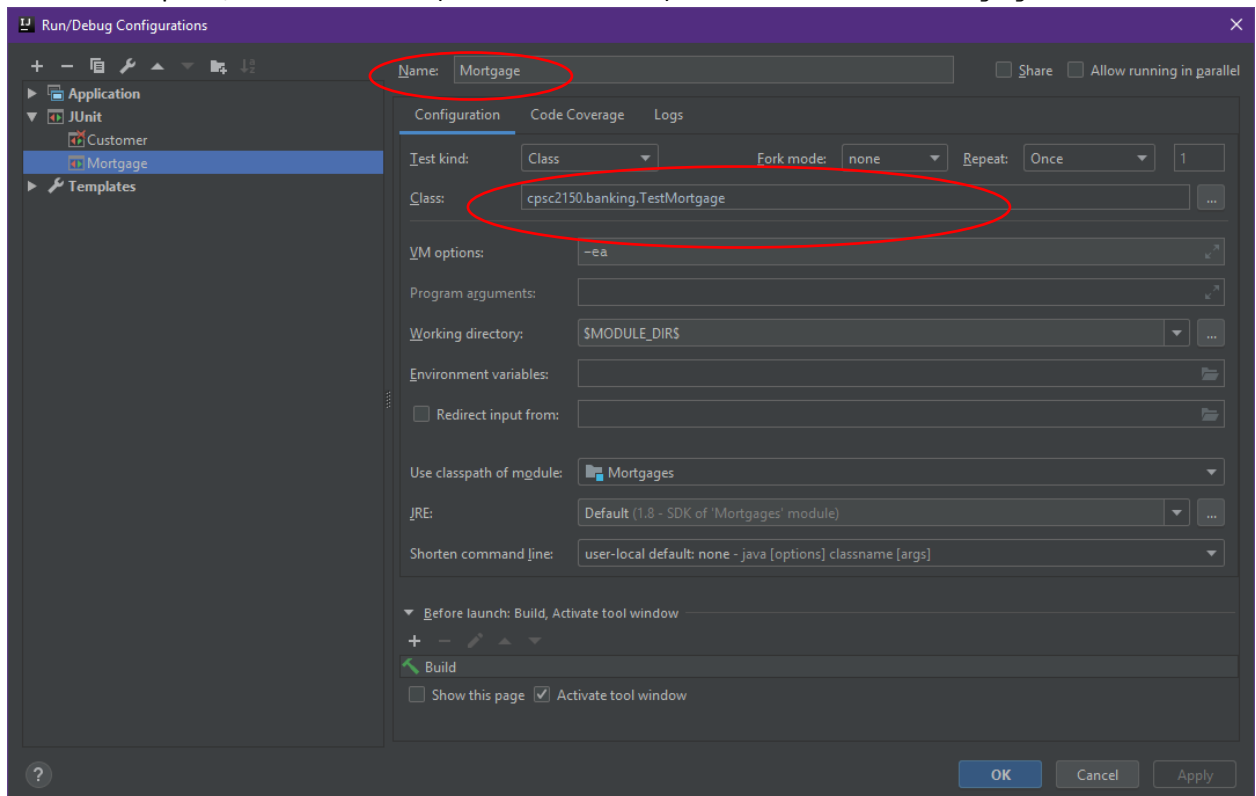


- d. Hit OK to exit the Project Structure window
6. In the navigation pane, right click on your test folder, and add a package to it. Use the same package name as the `src` folder (`cpssc2150.banking`)
 7. Add the `TestMortgage` class to the new package in the `test` folder. It's a good practice to name your JUnit classes `Test<CLASSNAME>` to keep our naming consistent. Your file structure should look like:

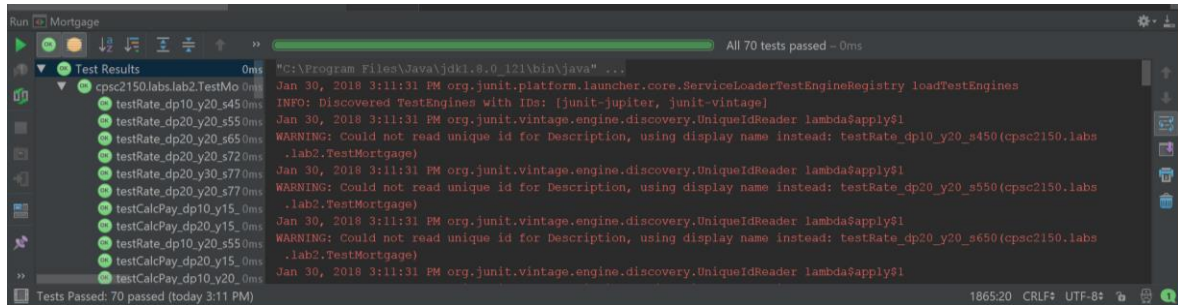


8. Most likely IntelliJ will have an issue with the `@Test` annotation (among other things) in `TestMortgage`. This is because it doesn't know we are trying to use JUnit. Click on the `@Test` annotation and hit `alt + enter`. If one of the options is to add JUnit4 to the classpath, congratulations! You 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" and hit search
 - e. One of the options that pops up should be JUnit:JUnit:4.10. Select that one and hit OK
 - i. It's OK if the 4.10 is 4.11 or 4.12 or any number after the 4. It just needs to be JUnit 4, not 3 or 5.
 - 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, call your new configuration `TestMortgage`
14. In the class option, use the selector (button with 3 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 completed, you can now your 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 on where the fault is in your code. When you have the green bar saying all test cases have passed, then you have completed the code. You'll get a lot or warnings. Ignore those.



Mortgage.java

Mortgage.java is the class you need to add in order 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 on how to implement the class, but we have some more information here as well. Remember to complete your correspondences and your invariants.

- 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). The constructor is called by TestMortgage, 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:

- o Start with the Base APR of 2.5%
- o If the loan is for less than 30 years, add 0.5%, otherwise add 1%
- o If the percent down is not at least 20%, add 0.5% to the APR
- o 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%

- o 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 cost of the house that is covered by the down payment for the house.
- The monthly payments for the loan is =
$$\frac{\text{monthly interest rate} * \text{principal}}{1 - (1 + \text{monthly interest rate})^{-(\text{total number of payments})}}$$
- A loan will be rejected if:
 - o The APR is greater than or equal to 10% OR
 - o 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 may be enough for simpler functions, but also comment your code as well when needed.

Running JUnit on Unix Machines

Set up the correct directory structure on one of the School of Computing Unix machines (`cpssc2150/banking`). Upload all your code files into that directory. Remember to go into the "Test" directory that IntelliJ created and upload 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` simpler). 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.

Groups

You may, but are not required to, work with a partner on this lab. Your partner must be in the same lab section as you, not just the same lecture section. If you work with a partner, only one person should submit the assignment. You should put the names of both partners in a comment at the top of the `Mortgage.java` file in order for both partners to get credit. This assignment may take more than just the lab time. Make sure you are able to meet outside of class to work on the assignment before you decide to work with someone else. Remember to actively collaborate and communicate with your partner. Trying to just divide up the work evenly will be problematic.

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. No late submissions will be accepted.

Submitting your file

You will submit your files using handin in the lab section you are enrolled in. If you are unfamiliar with handin, more information is available at <https://handin.cs.clemson.edu/help/students/>. You should submit a zipped directory with your package directory and your `makefile`. The TA should be able to unzip your directory and use any of the targets as specified above.

NOTE: Make sure you zipped up your files correctly and didn't forget something! Always check your submissions on handin to ensure you uploaded the correct zip file.