

CS160 Assignment #4 - Testing

Table of Contents

Page 2	Project and Title Authors
Page 3	Preface
Page 4	Instruction
Page 5-15	Black-Box Tests
Page 16	Coverage Criteria For White-Box Testing
Pages 17-31	White-Box Tests

Project and Title Authors

Title: ParkHere

Team Name: BlueJ Boys

Team Members:

- Kevin Vu (008861554)
- Stanley Plagata (009072700)
- Ricky Reyes (010674794)
- Nelson Nguyen (009233250)

Preface

Expected readership is any customers and stakeholders.

Version 1.0 on September 26, 2017: Initial Document Creation

We are creating the requirement document for the *ParkHere* application to describe the needed actions for the project.

Version 1.1 on October 7, 2017: Design Document

We are creating the design document for the *ParkHere* application to describe the mobile application. For developers and architecture design.

Version 1.2 on November 5, 2017: Implementation Document

We are creating the implementation document for the *ParkHere* application to describe the implementation changes for the mobile application.

Version 1.3 on November 9, 2017: Testing Document

We are creating the testing document for the *ParkHere* application to describe the testing performed on the implementation of the application.

Instruction

Black-Box Testing Instructions

Black-Box Tests reside within the directory:

ParkHere>app>src>androidTest>java>com.parkhere.android

Right-click and run each test class for executing the Black-Box Tests.

White-Box Testing Instructions

White-Box Tests reside within the directory:

ParkHere>app>src>test>java>com.parkhere.android

Right-click and run each test class for executing the White-Box Tests.

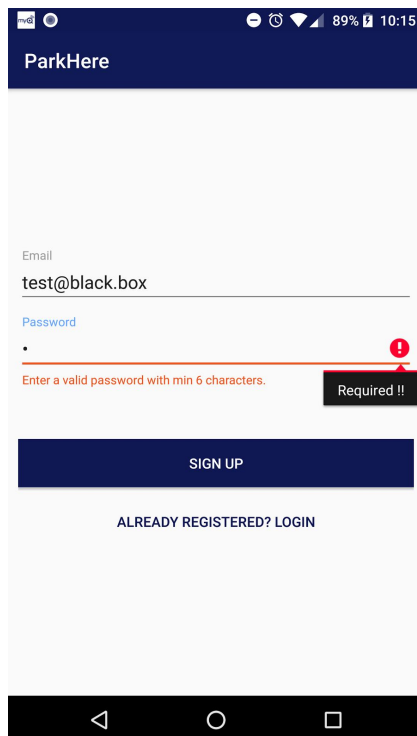
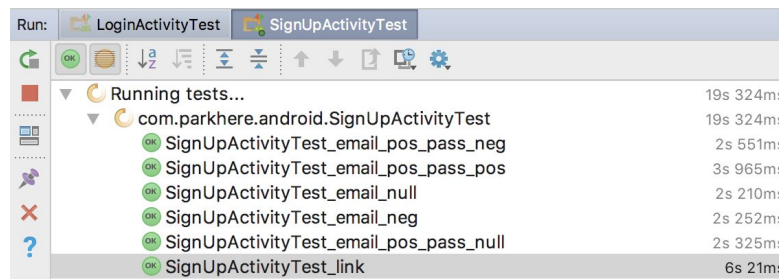
Black-Box Test: SignUpActivityTest

What is it: Tests the different scenarios a user would run into during the sign-up process.

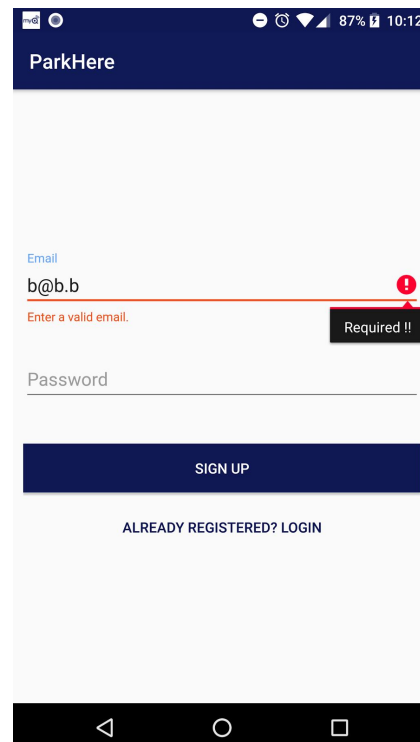
What it does: This covers the combinations of invalid and valid email and password, as well as the link button to going to and from *LoginActivity*.

Reason for test cases: This makes sure user is met with proper validity statements determined by their inputs.

Test Results: Tests were designed to look for specific error message strings that surface below the text field. Test cases found that our email/password validation was somewhat weak, allowing simplistic and some edge case invalid inputs. We strengthened the requirements, although we have email-confirmation functionality as a pending new feature. After updating our requirements, tests pass.



email positive, password negative



email negative

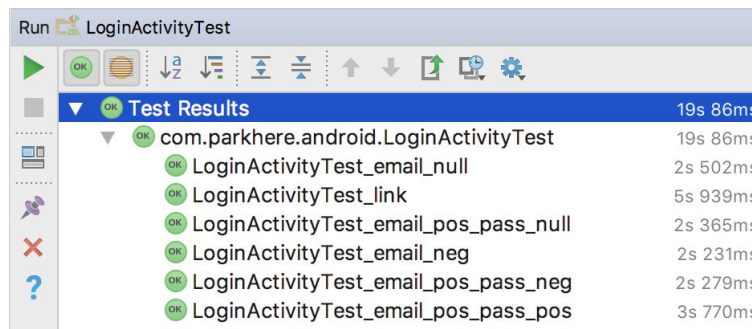
Black-Box Test: LoginActivityTest

What is it: Tests the different scenarios a user would run into during the login process.

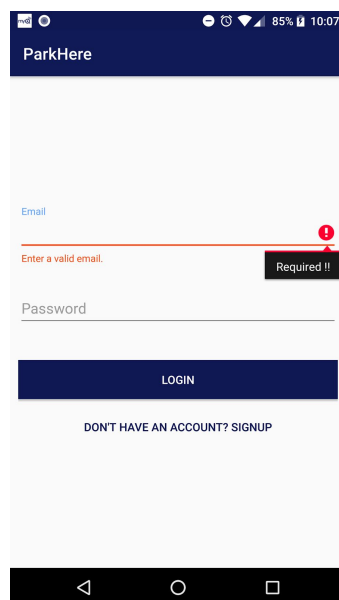
What it does: This covers the combinations of invalid and valid email and password, as well as the link button to going to and from *SignupActivity*.

Reason for test cases: This makes sure user is met with proper validity statements determined by their inputs.

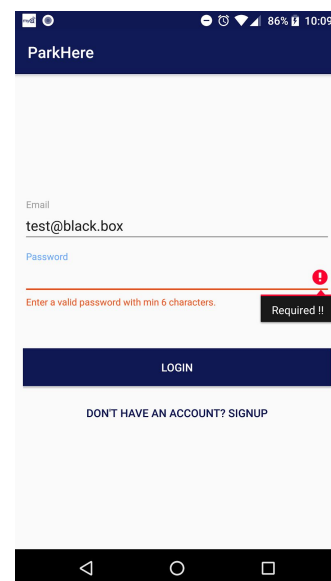
Test Results: Tests were designed to look for specific error message strings that surface below the text field. After strengthening our sign up requirements, we need to make sure the login follows the same rules. The tests pass.



Run LoginActivityTest	
Test Results	19s 86ms
com.parkhere.android.LoginActivityTest	19s 86ms
LoginActivityTest_email_null	2s 502ms
LoginActivityTest_link	5s 939ms
LoginActivityTest_email_pos_pass_null	2s 365ms
LoginActivityTest_email_neg	2s 231ms
LoginActivityTest_email_pos_pass_neg	2s 279ms
LoginActivityTest_email_pos_pass_pos	3s 770ms



email null



email positive, password null

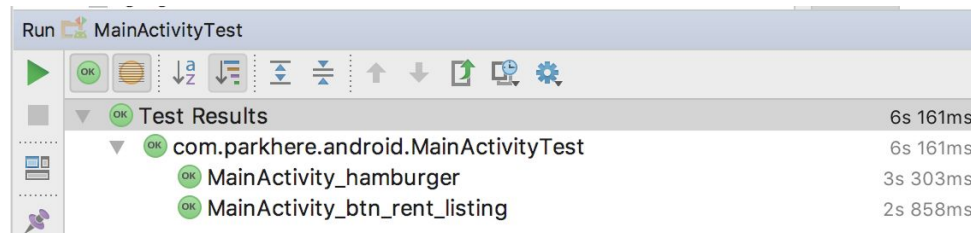
Black-Box Test: MainActivityTest

What is it: *MainActivity* is the page that users are met with upon successful login. Here a user may browse listings in their area, begin the process of reserving a listing, and access the hamburger menu.

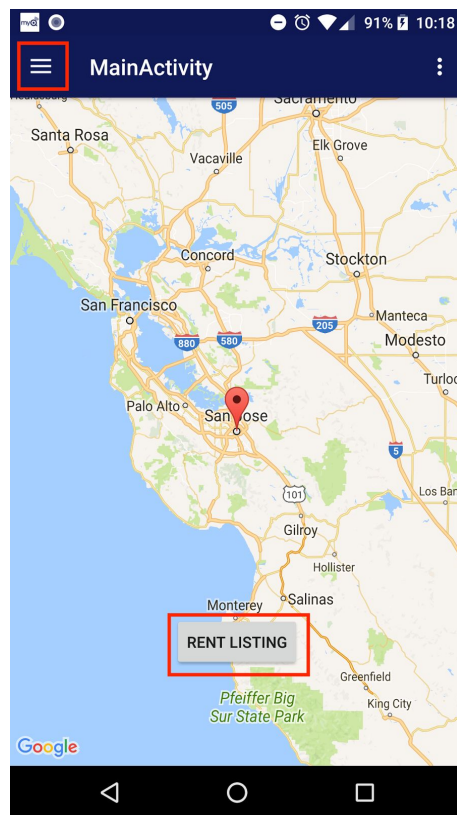
What it does: *MainActivityTest* aims to verify core buttons work and display properly.

Reason for test cases: Since *MainActivity* is the “home” of our app, it is vital that the buttons used to navigate to the other features of the app are working as expected.

Test Results: Here we test if the hamburger menu button is clickable, as well as the *RentListing* button. Tests pass. Issues we ran into on this page was access to the firebase server, displaying the map activity in home. After many failures in being able to run the *MainActivityTest*, we were able to fix the issues. The tests pass.



Run MainActivityTest	
Test Results	6s 161ms
com.parkhere.android.MainActivityTest	6s 161ms
MainActivity_hamburger	3s 303ms
MainActivity_btn_rent_listing	2s 858ms



the hamburger and rent listing buttons tested boxed in red

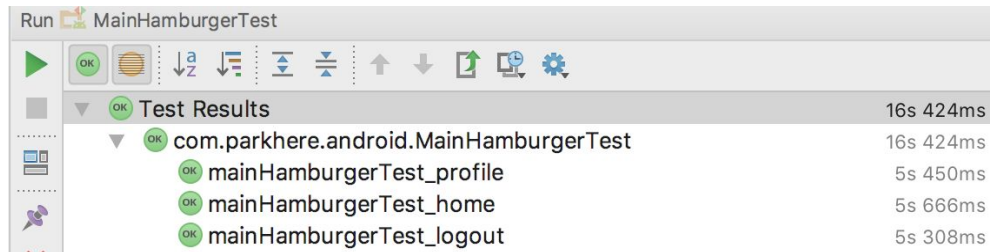
Black-Box Test: MainHamburgerTest

What is it: Tests surrounding the the buttons within the hamburger menu.

What it does: Verifies that the following buttons exist, are clickable, and reach the proper state.

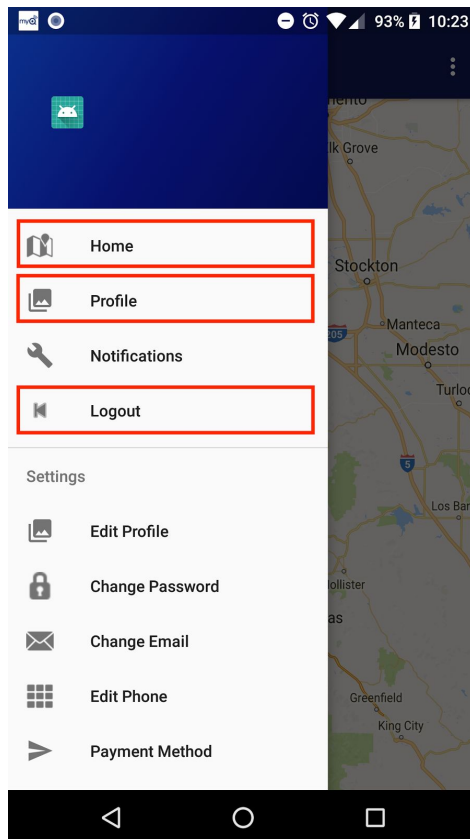
Reason for test cases: The hamburger menu is the apps main point of navigation, therefore the buttons need to be tested to make sure they are working as expected.

Test Results: Tests pass.



The screenshot shows the 'Run' tab in Android Studio. At the top, there's a toolbar with various icons. Below it, the 'Test Results' section is expanded, showing a list of test cases. The first test case is 'com.parkhere.android.MainHamburgerTest' with a duration of 16s 424ms. It has four sub-items, all marked with a green 'OK' icon: 'mainHamburgerTest_profile' (5s 450ms), 'mainHamburgerTest_home' (5s 666ms), and 'mainHamburgerTest_logout' (5s 308ms).

Test Case	Duration
com.parkhere.android.MainHamburgerTest	16s 424ms
mainHamburgerTest_profile	5s 450ms
mainHamburgerTest_home	5s 666ms
mainHamburgerTest_logout	5s 308ms



Home, Profile, and Logout buttons tested boxed in red

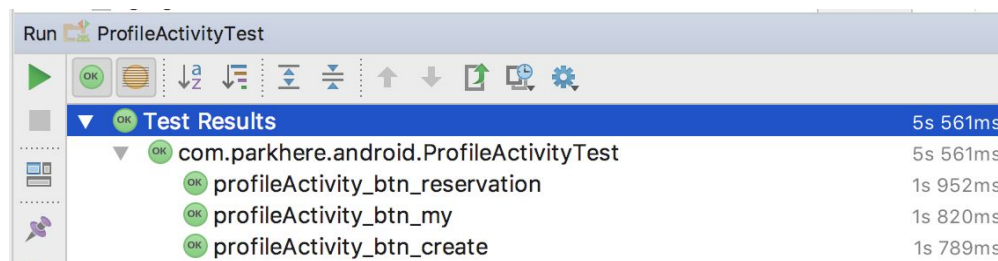
Black-Box Test: ProfileActivityTest

What is it: *ProfileActivity* is where the user is able to make new listings, view their listings, and their reservations

What it does: This *ProfileActivityTest* class verifies that each button exists and is clickable, reaching the proper state. The user must be logged in before runtime.

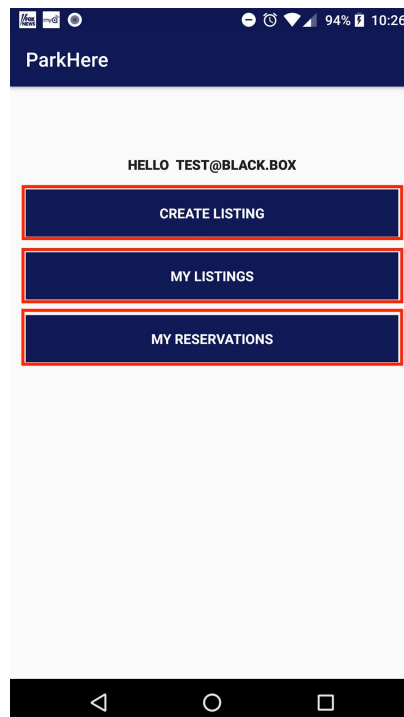
Reason for test cases: This makes sure that these three core user functions are accessible to a user of the app.

Test Results: Ran into some issues starting this test from the *ProfileActivity*, having a dependency on logging in first. We are able to bypass this blocker by logging into the app before running the test class. When doing so, tests pass as buttons are clickable and reach proper state.



The screenshot shows the 'Run' window in Android Studio with the 'Test Results' tab selected. The test class 'com.parkhere.android.ProfileActivityTest' has passed, and its three sub-tests have also passed. The execution times are listed in milliseconds.

Test Name	Duration
com.parkhere.android.ProfileActivityTest	5s 561ms
profileActivity_btn_reservation	1s 952ms
profileActivity_btn_my	1s 820ms
profileActivity_btn_create	1s 789ms



Create Listing, My Listings, and My Reservations buttons boxed in red

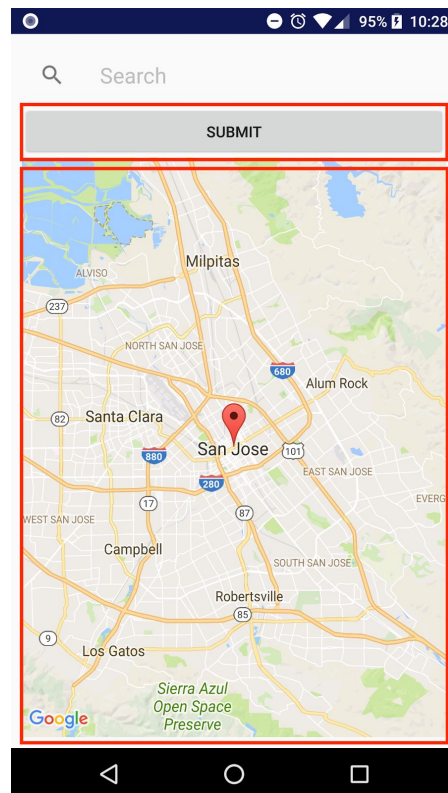
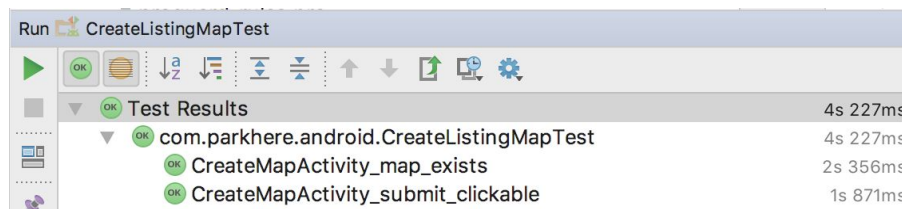
Black-Box Test: CreateListingMapTest

What is it: *CreateListingMap* is the first step in our *CreateListing* flow. This page is for users to enter in an address for creating a listing.

What it does: This test case validates that the map is properly being shown in the activity, and also verifies the *Submit* button is clickable.

Reason for test cases: As this page is important in assisting users to find their address quickly, it is important to make sure the functionality is working as expected.

Test Results: Tests pass. The map exists within the page, and the *Submit* button is clickable.



Map and Submit button tested are boxed in red

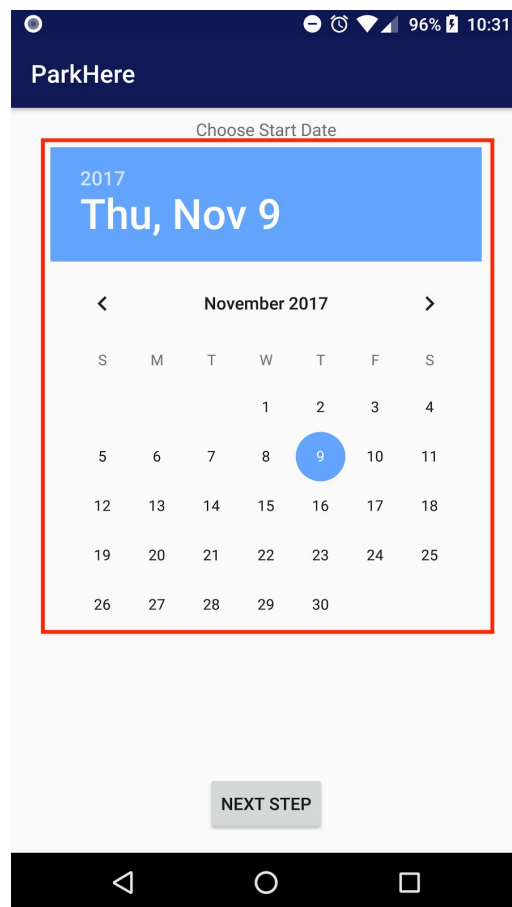
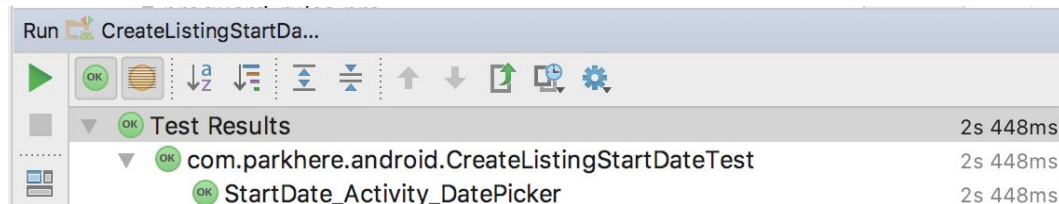
Black-Box Test: CreateListingStartDateTest

What is it: Test that covers the *CreateListingStartDate* Activity.

What it does: This test confirms that the *DatePicker* exists within the activity.

Reason for test cases: The main reason for this test case is to verify the existence of the *DatePicker*, or *Calendar* that we are using to select the date. This is important to make sure our create flow is usable and is presenting the proper resources.

Test Results: Test passes.



DatePicker is boxed in red

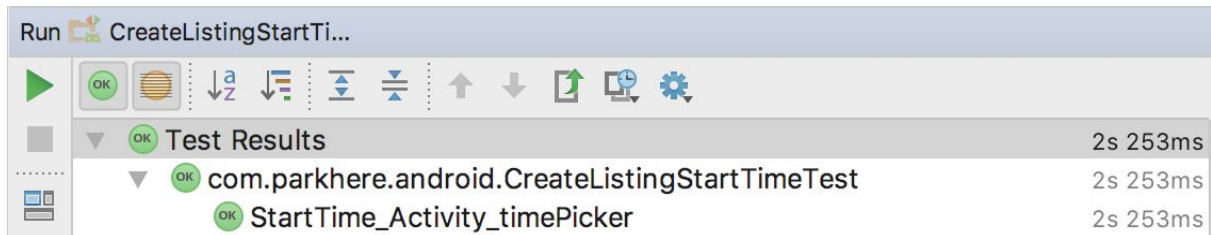
Black-Box Test: CreateListingStartTimeTest

What is it: Test that covers the *CreateListingStartTime* Activity.

What it does: This test confirms that the *TimePicker* exists within the activity.

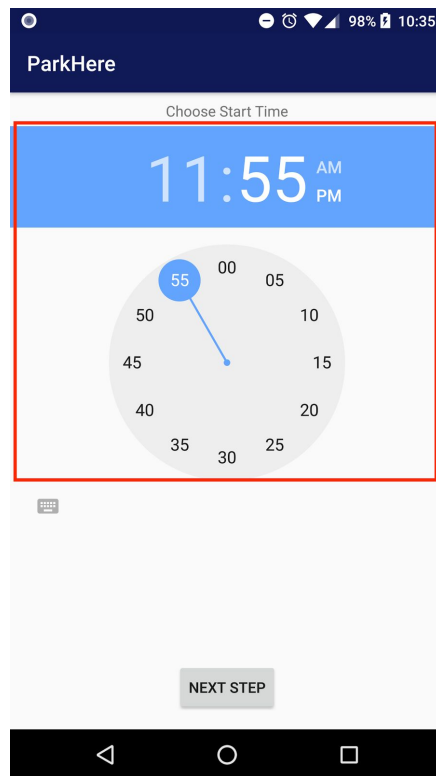
Reason for test cases: The main reason for this test case is to verify the existence of the *TimePicker*, or clock that we are using to select the date. This is important to make sure our create flow is usable and is presenting the proper resources.

Test Results: Test passes.



The screenshot shows the 'Run' tab in Android Studio with the test results for 'CreateListingStartTi...'. The 'Test Results' section is expanded, showing a successful test run for 'com.parkhere.android.CreateListingStartTimeTest' and its sub-test 'StartTime_Activity_timePicker'. Both tests passed, each taking 2s 253ms.

Test Name	Duration
Test Results	2s 253ms
com.parkhere.android.CreateListingStartTimeTest	2s 253ms
StartTime_Activity_timePicker	2s 253ms



Timepicker is boxed in red

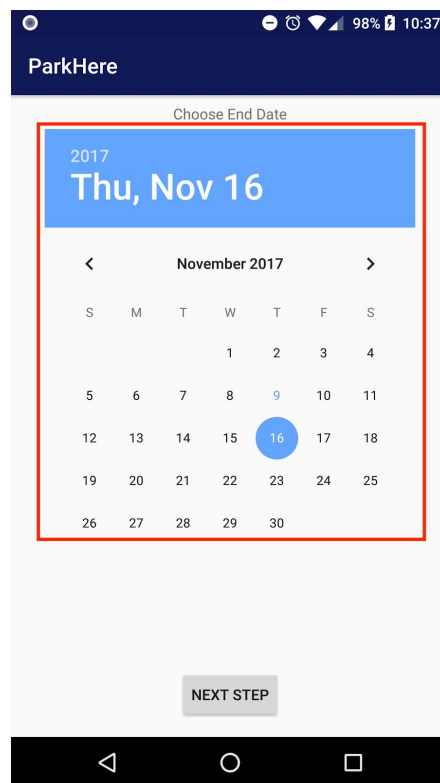
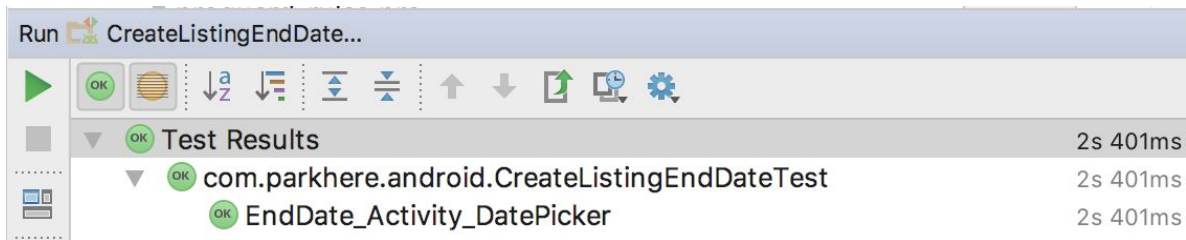
Black-Box Test: CreateListingEndDateTest

What is it: Test that covers the *CreateListingStartDate* Activity.

What it does: This test confirms that the *DatePicker* exists within the activity.

Reason for test cases: The main reason for this test case is to verify the existence of the *DatePicker*, or *Calendar* that we are using to select the date. This is important to make sure our create flow is usable and is presenting the proper resources. As this is similar to *StartDate* Activity, we need to make sure all the steps have the proper resources.

Test Results: Test passes.



Datepicker is boxed in red

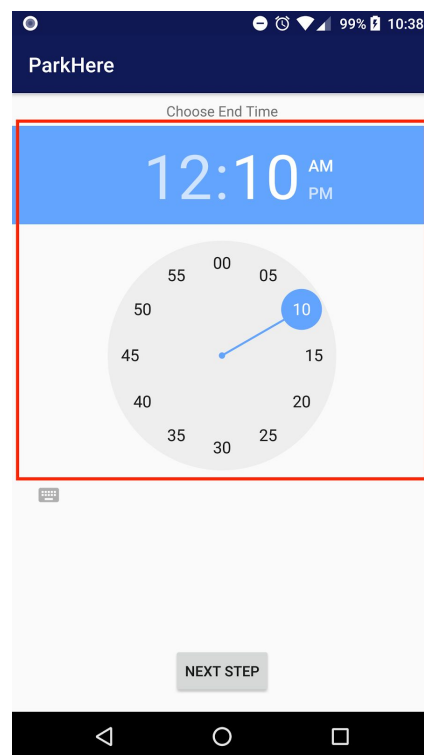
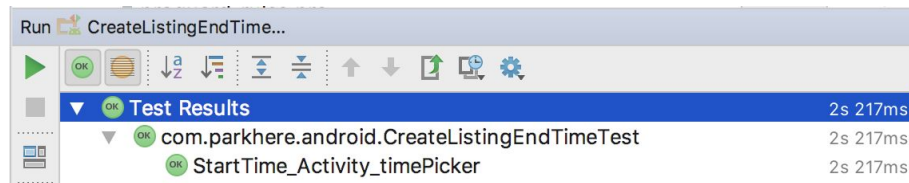
Black-Box Test: CreateListingEndTimeTest

What is it: Test that covers the *CreateListingStartTime* Activity.

What it does: This test confirms that the *TimePicker* exists within the activity.

Reason for test cases: The main reason for this test case is to verify the existence of the *TimePicker*, or clock that we are using to select the date. This ending time is important to make sure our create flow is usable and is presenting the proper resources.

Test Results: Test passes.



Timepicker is boxed in red

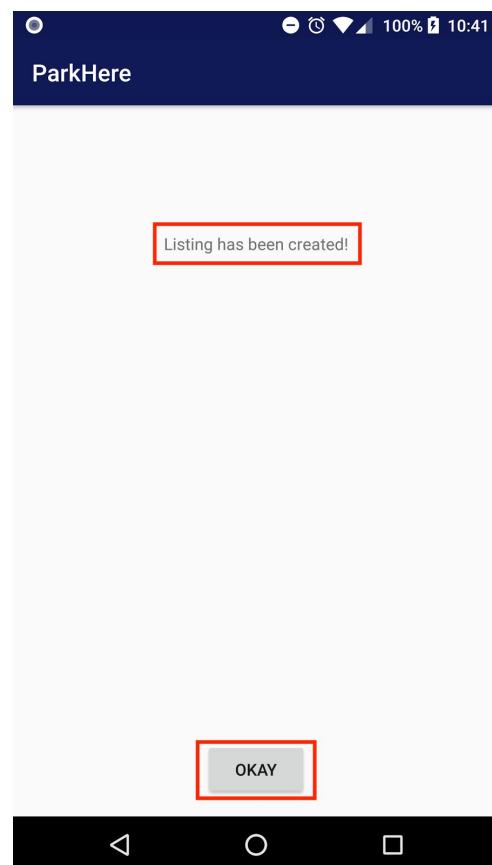
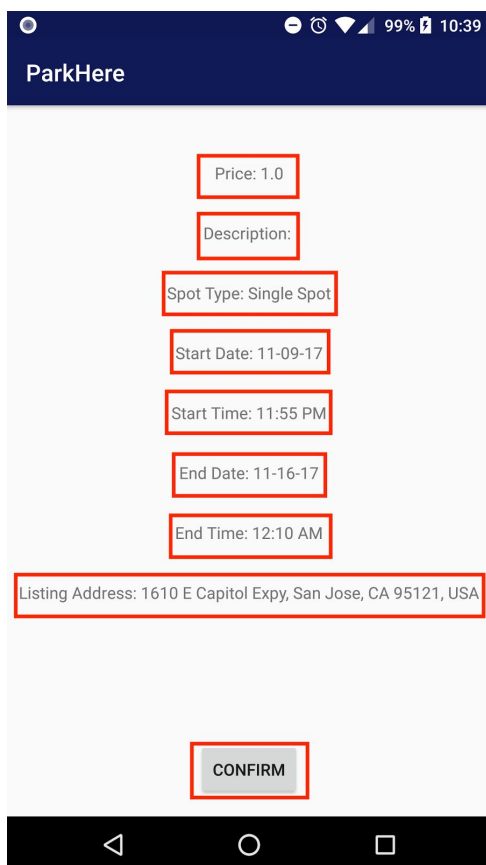
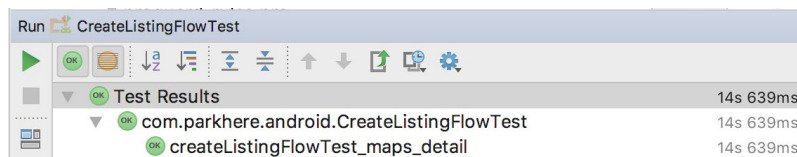
Black-Box Test: CreateListingFlowTest

What is it: This test case runs through a positive case of the entire *CreateListing* flow.

What it does: verifies that the activities are properly passing their elements to the end of the activity. Goes through each of the steps with valid data.

Reason for test cases: This test case makes sure that our *CreateListing* flow is working properly. At any time this test fails, we know that there is a major issue in our system.

Test Results: This test arose many issues with how we were validating our results. As this test runs through multiple activities, it proved useful in validating our internal functions that validate the data. After fixing those issues, the test passes. For scalability, we could automate input sets to test a larger amount of listings in the system.



Final text and okay button outlined in red

Each required input and *confirm* button
outlined in red

Coverage Criteria For White-Box Testing

Explanation for the use of control flow and conditionals in our testing:

The way the behavior of our white-box testing was handled was through using control flows and conditionals. This decision was made because most of the functions are checking for invalid inputs within the activities. We ensured that we met our coverage criterion by providing test inputs that would check the true path on the conditional and the false path on the conditional. For instance, there would be a good or valid inputs that would assert true and bad or invalid inputs that would assert false for the functions we were testing. The conditionals were used to guide the control flow into different conditional statements in order to narrow down the possible reasons for failure. For example, a test will be executed on a value; the following tests executed on the value will differ based on the result of the first conditional statement.

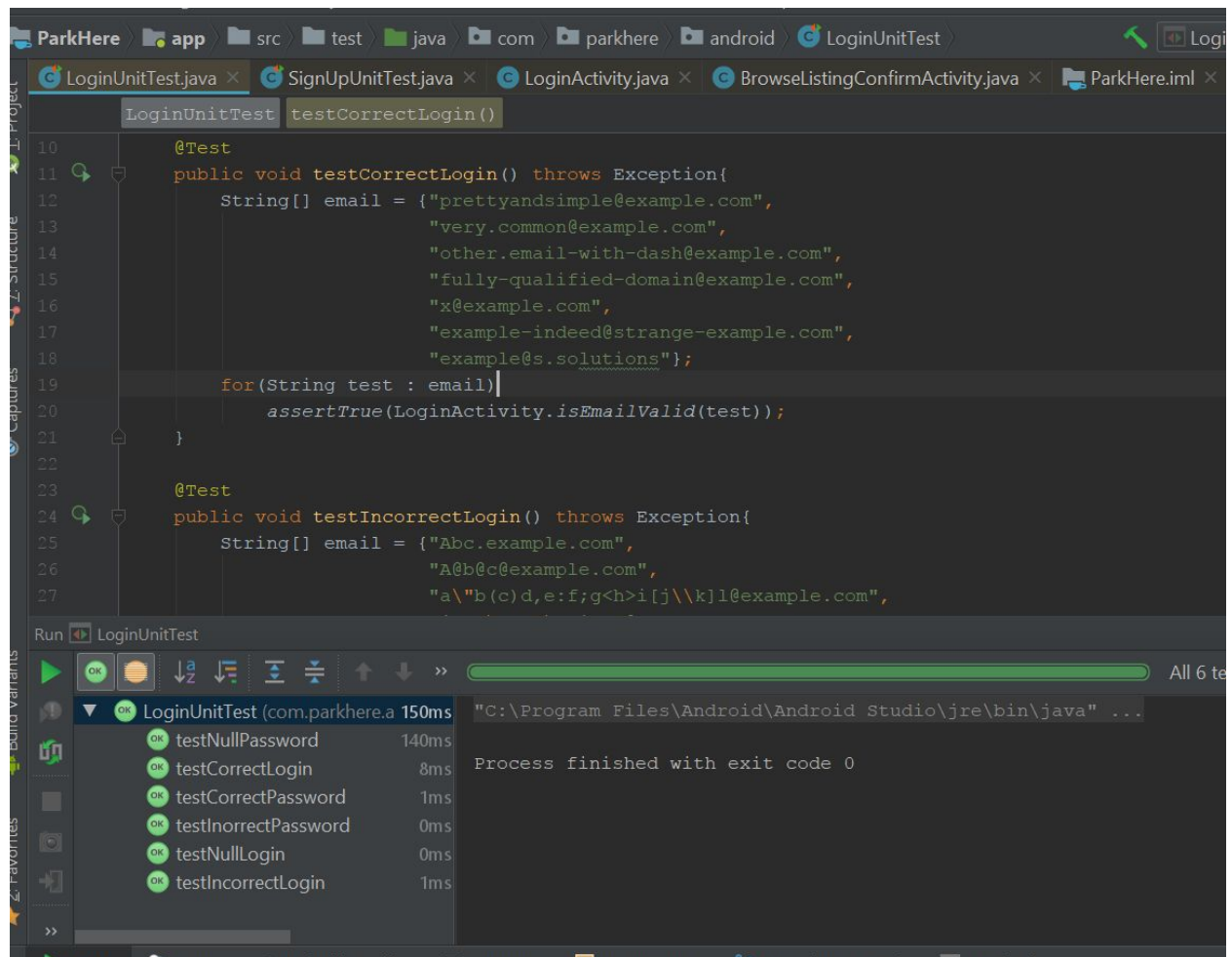
White-Box Test: LoginUnitTest1

What is it: This tests the user's input in the *LoginActivity* class to examine the email validation.

What it does: The test iterates through an array of strings that contain important cases for the function *isEmailValid(String s)* in the *LoginActivity* class.

Reason for test cases: The *LoginUnitTest* class tested three core functions which are correct emails, incorrect emails, and a null email. The cases that should pass include dot emails, dot-dash emails, dash-dash emails, single-character emails, dash-at-dash emails, and name-at-dot emails. The cases that should fail include emails without ats, emails with multiple ats, emails with special characters, emails with quoted strings, emails with double dots, emails with white spaces, nested emails, and the null case.

Test Results: All test cases passed.



The screenshot displays the Android Studio IDE with the `LoginUnitTest.java` file open. The file contains two test methods: `testCorrectLogin()` and `testIncorrectLogin()`. The `testCorrectLogin()` method iterates through an array of valid email addresses and asserts that `LoginActivity.isEmailValid()` returns `true` for each. The `testIncorrectLogin()` method iterates through an array of invalid email addresses and asserts that `LoginActivity.isEmailValid()` returns `false` for each. Below the code editor, the `Run` tab shows the test results for `LoginUnitTest`. The tests passed, and the process finished with exit code 0.

```
10  @Test
11  public void testCorrectLogin() throws Exception{
12      String[] email = {"prettyandsimple@example.com",
13                       "very.common@example.com",
14                       "other.email-with-dash@example.com",
15                       "fully-qualified-domain@example.com",
16                       "x@example.com",
17                       "example-indeed@strange-example.com",
18                       "example@s.solutions"};
19      for(String test : email){
20          assertTrue(LoginActivity.isEmailValid(test));
21      }
22
23  @Test
24  public void testIncorrectLogin() throws Exception{
25      String[] email = {"Abc.example.com",
26                       "A@b@c@example.com",
27                       "a\"b(c)d,e:f;g<h>i[j\\k]l@example.com",
```

Run LoginUnitTest

Test Method	Duration
testNullPassword	140ms
testCorrectLogin	8ms
testCorrectPassword	1ms
testIncorrectPassword	0ms
testNullLogin	0ms
testIncorrectLogin	1ms

Process finished with exit code 0

Impact: There are many different edge cases for testing email validation. Some of the tests failed at first, but a few changes to the *LoginActivity* class solved these different cases. All tests passed.

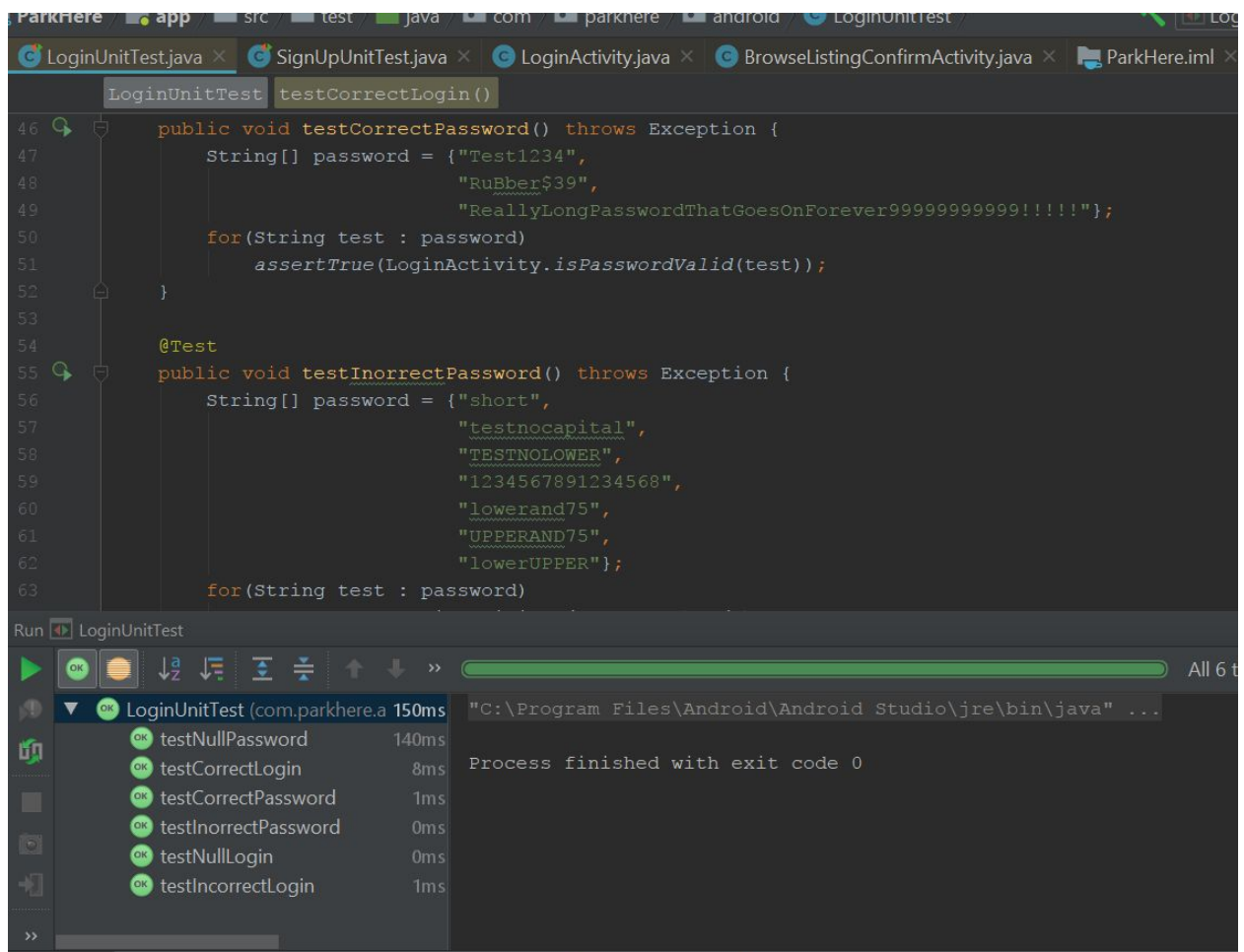
White-Box Test: LoginUnitTest2

What is it: Tests the user's input in the *LoginActivity* to check if the password is a valid or invalid.

What it does: The test checks for password validation.

Reason for test cases: The *LoginUnitTest* class tested three core functions which are correct passwords, incorrect passwords, and a null password. The cases that should pass include password with special characters, and a long password. The cases that should fail include passwords under eight characters, passwords without uppercase letters, passwords with only uppercase, passwords with only numbers, passwords with only lowercase and number, passwords with uppercase and number, lower with upper, and the null case.

Test Results:



The screenshot displays an IDE with the `LoginUnitTest.java` file open. The code defines two test methods: `testCorrectPassword()` and `testIncorrectPassword()`. Below the code, the 'Run' tab shows the test results for `LoginUnitTest`.

```
public void testCorrectPassword() throws Exception {
    String[] password = {"Test1234",
                        "RuBber$39",
                        "ReallyLongPasswordThatGoesOnForever9999999999!!!!!!"};
    for(String test : password)
        assertTrue(LoginActivity.isPasswordValid(test));
}

@Test
public void testIncorrectPassword() throws Exception {
    String[] password = {"short",
                        "testnocapital",
                        "TESTNOLOWER",
                        "1234567891234568",
                        "lowerand75",
                        "UPPERAND75",
                        "lowerUPPER"};
    for(String test : password)
```

Run LoginUnitTest

Test Case	Duration
testNullPassword	140ms
testCorrectLogin	8ms
testCorrectPassword	1ms
testIncorrectPassword	0ms
testNullLogin	0ms
testIncorrectLogin	1ms

Process finished with exit code 0

Impact: There are a few test cases for password validation. We require a string greater than 8 characters and at least one uppercase, lowercase, and a digit. We changed the password requirements for stronger security. All tests passed.

White-Box Test: SignUpUnitTest1

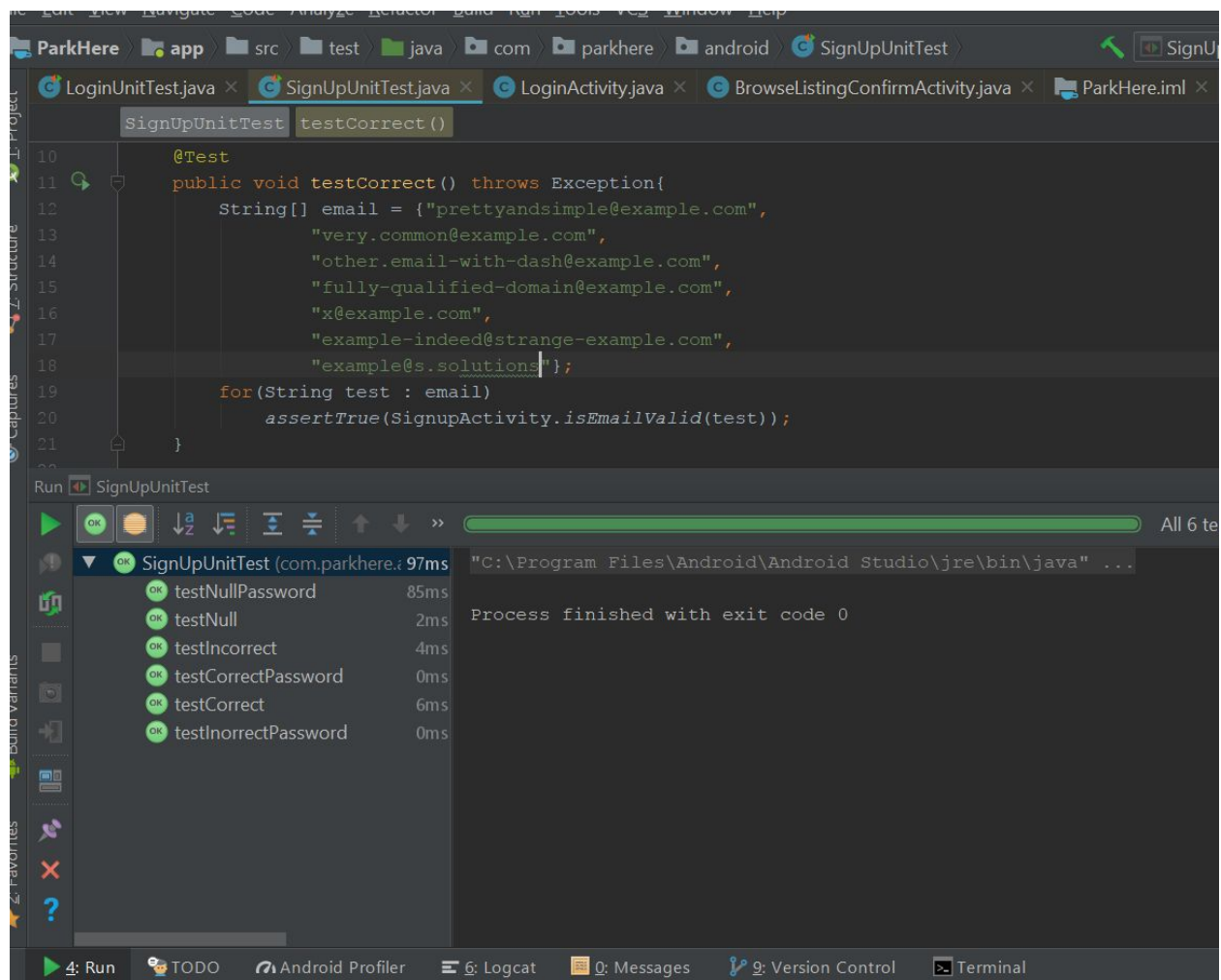
What is it: Tests the user's input in the *SignupActivity* to examine the email validation.

What it does: The test checks for email validation.

Reason for test cases: The *SignUpUnitTest* class tested three core functions which are

correct emails, incorrect emails, and a null email. The cases that should pass include dot emails, dot-dash emails, dash-dash email, single character emails, dash-at-dash emails, and name-at-dot emails. The cases that should fail include emails without ats, emails with multiple ats, emails with special characters, emails with quoted strings, emails with double dots, emails with white spaces, nested emails, emails with special characters, and the null case.

Test Results:



The screenshot displays an IDE window with the `SignUpUnitTest.java` file open. The code defines a `testCorrect()` method that tests various email addresses. Below the code, the `Run` tab shows the test results for `SignUpUnitTest`.

```
10      @Test
11      public void testCorrect() throws Exception{
12          String[] email = {"prettyandsimple@example.com",
13                          "very.common@example.com",
14                          "other.email-with-dash@example.com",
15                          "fully-qualified-domain@example.com",
16                          "x@example.com",
17                          "example-indeed@strange-example.com",
18                          "example@s.solutions"};
19          for(String test : email)
20              assertTrue(SignupActivity.isEmailValid(test));
21      }
```

Run Results:

Test Case	Duration	Status
testNullPassword	85ms	OK
testNull	2ms	OK
testIncorrect	4ms	OK
testCorrectPassword	0ms	OK
testCorrect	6ms	OK
testIncorrectPassword	0ms	OK

The terminal output shows the command `"C:\Program Files\Android\Android Studio\jre\bin\java" ...` and the message `Process finished with exit code 0`.

Impact: There are many different edge cases for testing email validation. Some of the tests failed at first, but a few changes to the *SignupActivity* solved these different cases. All tests passed.

White-Box Test: SignUpUnitTest2

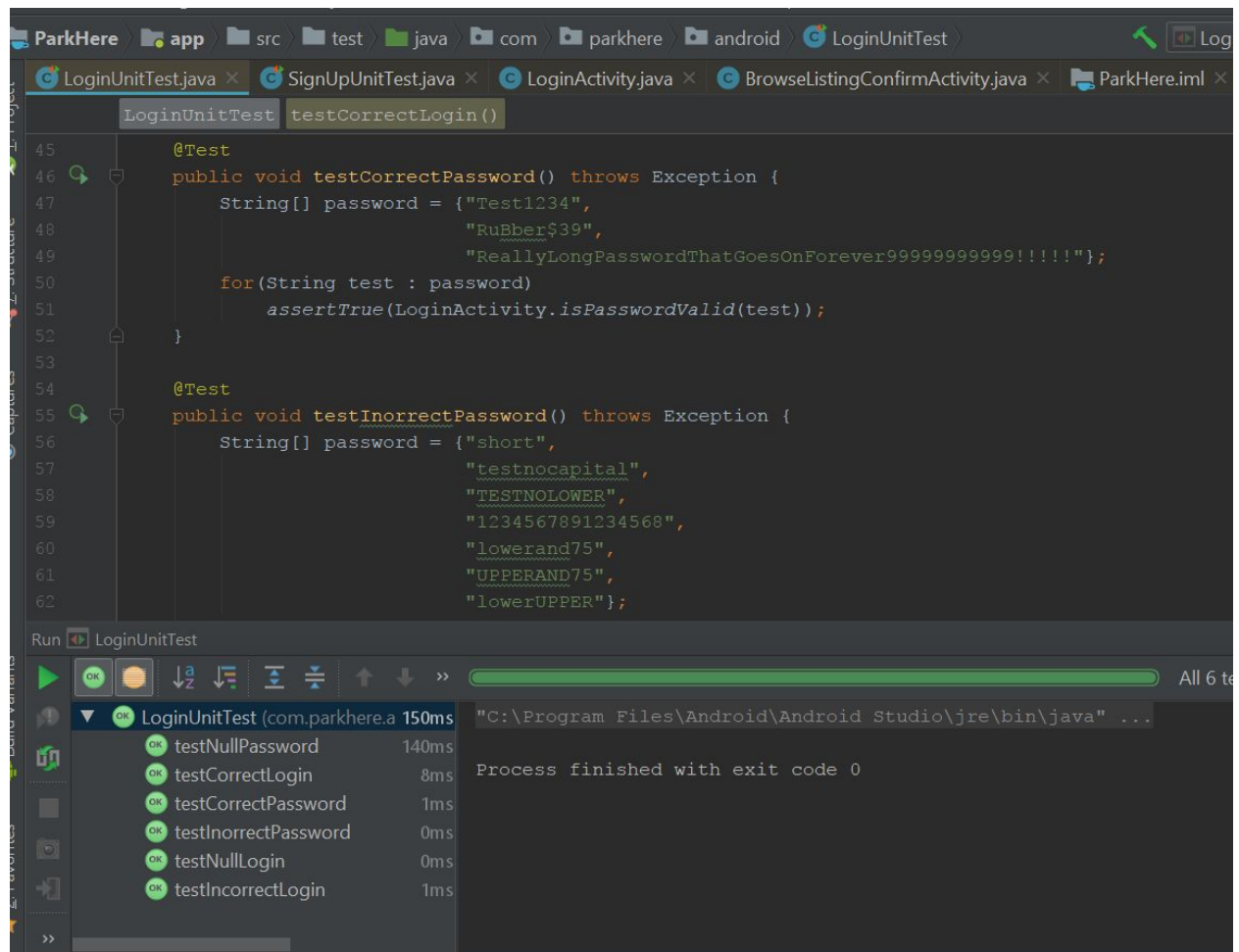
What is it: Tests the user's input in the *SignupActivity* to check if the password is a valid or invalid.

What it does: The test checks for password validation.

Reason for test cases: The *SignUpUnitTest* class tested three core functions which are

correct passwords, incorrect passwords, and a null password. The cases that should pass include passwords normal passwords, passwords with special character, and a long password. The cases that should fail are passwords under eight characters, passwords with no uppercase, passwords with only uppercase, passwords with only numbers, passwords with only lowercase and number, passwords with uppercase and number, lower with upper, and the null case.

Test Results:



The screenshot displays an IDE window with the following components:

- File Explorer:** Shows the project structure: ParkHere > app > src > test > java > com > parkhere > android > LoginUnitTest.
- Code Editor:** Contains the `LoginUnitTest` class with two test methods:

```
45     @Test
46     public void testCorrectPassword() throws Exception {
47         String[] password = {"Test1234",
48                             "RuBber$39",
49                             "ReallyLongPasswordThatGoesOnForever9999999999!!!!!!"};
50         for(String test : password)
51             assertTrue(LoginActivity.isPasswordValid(test));
52     }
53
54     @Test
55     public void testIncorrectPassword() throws Exception {
56         String[] password = {"short",
57                             "testnocapital",
58                             "TESTNOLOWER",
59                             "1234567891234568",
60                             "lowerand75",
61                             "UPPERAND75",
62                             "lowerUPPER"};
```
- Run Console:** Shows the execution of the tests:

Test Method	Duration
testNullPassword	140ms
testCorrectLogin	8ms
testCorrectPassword	1ms
testIncorrectPassword	0ms
testNullLogin	0ms
testIncorrectLogin	1ms
- Logcat:** Shows the process finished with exit code 0.

Impact: There are a few test cases for password validation. We require a string greater than eight characters and at least one uppercase, lowercase, and a digit. We changed the password requirements for stronger security. All tests passed.

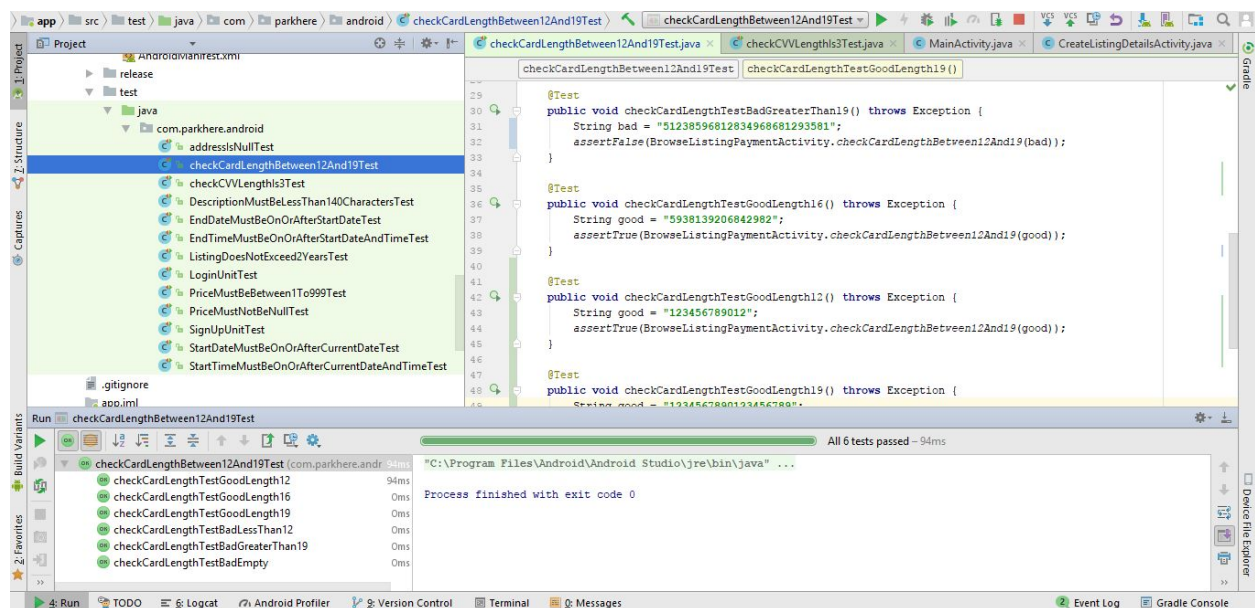
White-Box Test: checkCardLengthBetween12And19Test

What is it: This test is for testing if the input length for the card number within the *BrowseListingPayment* activity is valid.

What it does: The test checks if the card number length is between twelve and nineteen, because valid credit cards and debit cards don't exceed nineteen digits in length and are not less than twelve digits in length. To execute the test case, right click the *checkCardLengthBetween12And19Test* class within the *test/java/com/parkhere/android* directory and click run.

Reason for test cases: There are six test cases within this test unit. The six test cases test an input that is less than twelve digits in length, an input that is greater than nineteen digits in length, empty input and an valid inputs that are twelve, sixteen and nineteen in length. The reason I tested an empty input because it's possible the user can enter no input. I tested inputs greater than nineteen and less than twelve to see if the function returned false to indicate that it is not possible to input anything else than a card number input that has a length within twelve and nineteen digits. Finally I used a good input of length twelve, sixteen, and nineteen to see if it returns true and makes sure that the card number input is valid before moving on to the next activity.

Test Results: All test cases passed.



Impact: Because all of the test cases passed, the function was working as intended and no bugs were found.

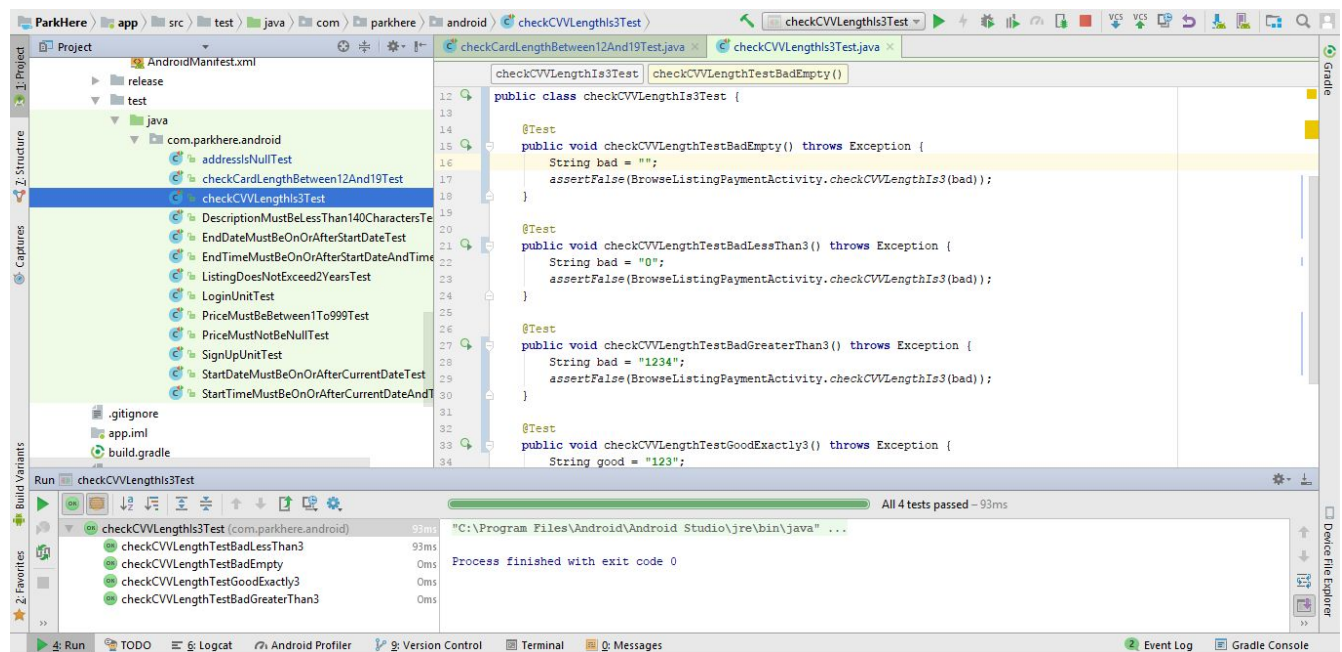
White-Box Test: checkCVVLengthIs3Test

What is it: This test is for testing the if the CVV input within the *BrowseListingPayment* activity is valid.

What it does: The test checks if the CVV input is exactly three digits in length because all credit cards and debit cards have exactly three digits for their CVV. To execute the test case, right click the *checkCVVLengthIs3Test* class within the `test/java/com/parkhere/android` directory and click run.

Reason for test cases: There are four test cases for this test unit. The four test cases tests an empty input, an input of exactly three in length, an input greater than three and an input less than three. The reason I tested an empty input because it's possible the user can enter no input. I tested inputs greater than three and less than three to see if the function returned false to indicate that it is not possible to input anything else than a string that has a length of three. Finally I used a good input of exactly length three to see if it returns true and makes sure that the CVV input is valid before moving on to the next activity.

Test Results: All test cases passed.



Impact: Because all of the test cases passed, the function was working as intended and no bugs were found.

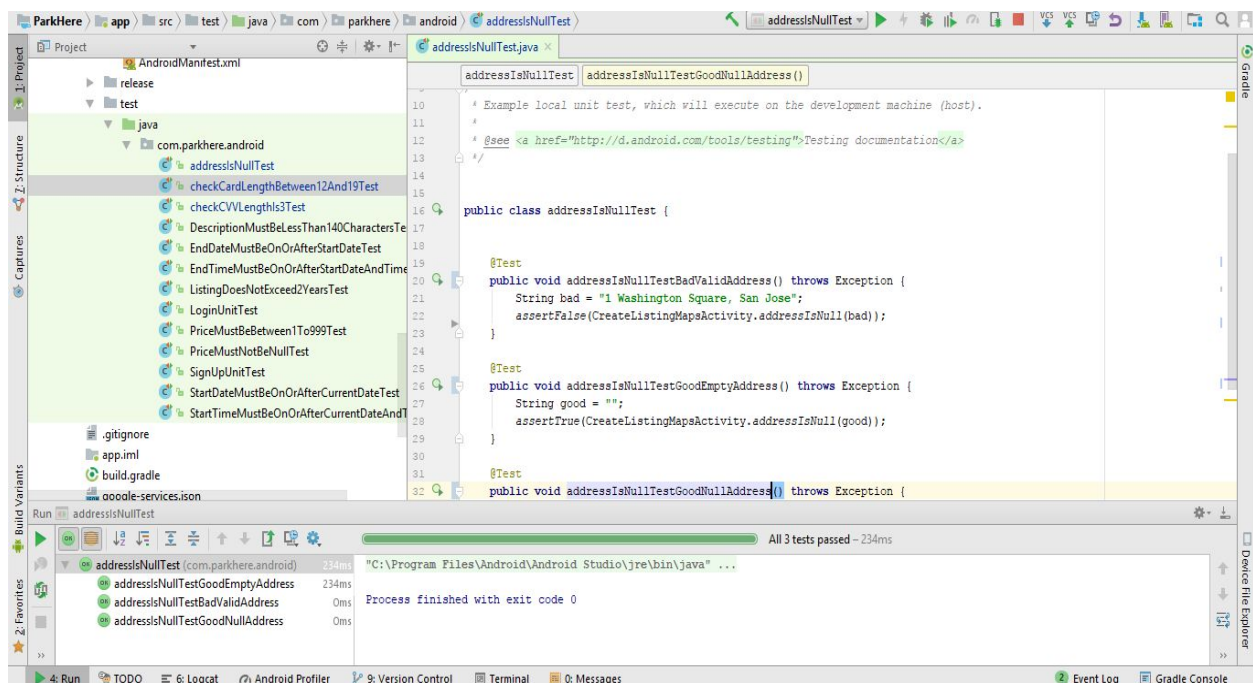
White-Box Test: addressIsNullTest

What is it: This test is for testing if the address is null or empty within the *CreateListingMaps* activity before sending it to the next activity.

What it does: This test checks if the address given from the Place Autocomplete Text box is null or empty. This is because if a null or empty address is sent to the next activity, it will cause crashes within the main application because an invalid address can't be shown on a marker on the google map. To execute the test case, right click the *isAddressNullTest* class within the *test/java/com/parkhere/android* directory and click run.

Reason for test cases: There are three test cases for this test unit. The three test cases tests a null value, an empty string and a passing value that is a valid address. The reason I used a null value input is to test if the *addressIsNull* function returns true. I also used an empty string to see if the function also returned true. If this function returns true on a null address or empty string, then we can prevent the activity from going forward unless the input is valid. I included a bad input which is a valid address to see if the function returns false. I didn't include any other bad inputs because as long as the input isn't null or empty, any other input would return false.

Test Results: All three test cases passed.



The screenshot displays the Android Studio interface. On the left, the Project Explorer shows the directory structure with the file `addressIsNullTest` selected under `com.parkhere.android.test`. The main editor shows the `addressIsNullTest.java` file with the following code:

```
10  addressIsNullTest addressIsNullTestGoodNullAddress()
11
12  * Example local unit test, which will execute on the development machine (host).
13  *
14  * @see <a href="http://d.android.com/tools/testing">Testing documentation</a>
15  */
16
17  public class addressIsNullTest {
18
19      @Test
20      public void addressIsNullTestBadValidAddress() throws Exception {
21          String bad = "1 Washington Square, San Jose";
22          assertFalse(CreateListingMapsActivity.addressIsNull(bad));
23      }
24
25      @Test
26      public void addressIsNullTestGoodEmptyAddress() throws Exception {
27          String good = "";
28          assertTrue(CreateListingMapsActivity.addressIsNull(good));
29      }
30
31      @Test
32      public void addressIsNullTestGoodNullAddress() throws Exception {
```

At the bottom, the Run tab shows the test results for `addressIsNullTest` (com.parkhere.android). The results indicate that all three tests passed:

- `addressIsNullTestGoodEmptyAddress` (234ms)
- `addressIsNullTestBadValidAddress` (0ms)
- `addressIsNullTestGoodNullAddress` (0ms)

The overall status is "All 3 tests passed - 234ms". The console output shows "Process finished with exit code 0".

Impact: Because all of the test cases passed, the function was working as intended and no bugs were found.

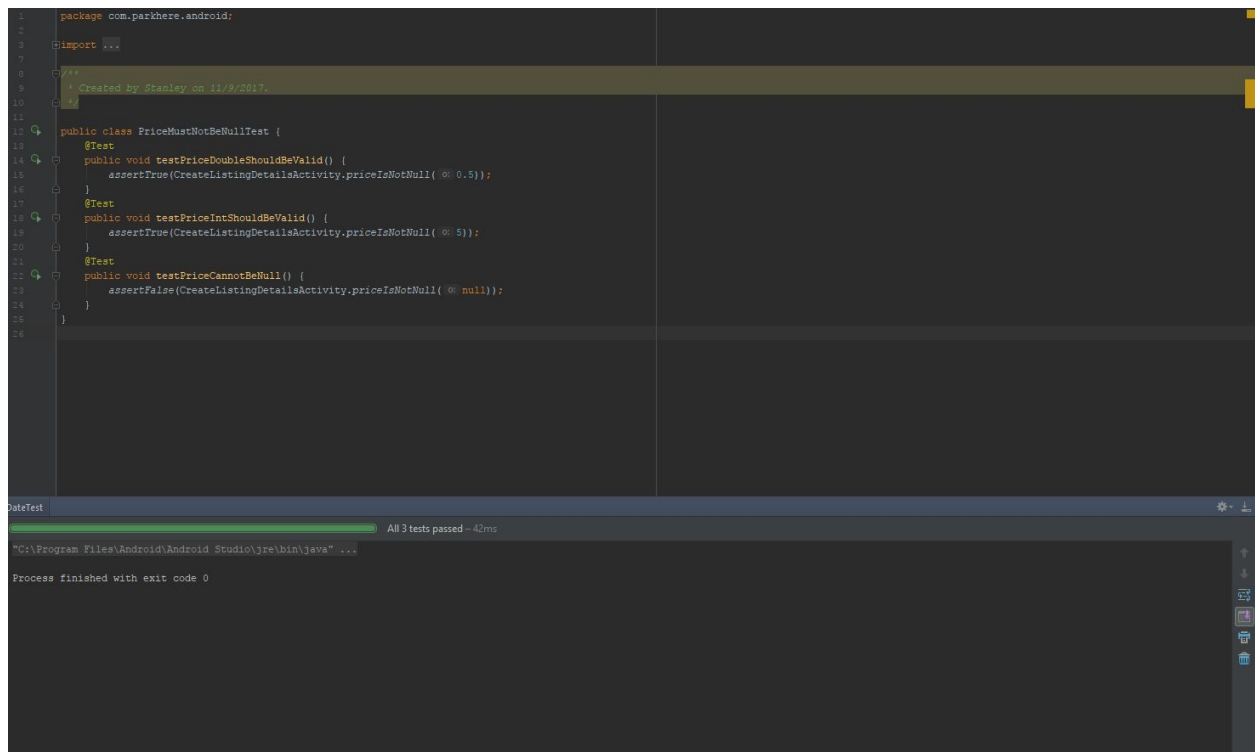
White-Box Test: PriceMustNotBeNull

What it is: This test was included in order to test for null values for the *Price* variable.

What it does: This test checks for cases where the *Price* variable would be null and cases where the *Price* variable wouldn't be null. Execute the test by right clicking on the class and clicking "Run 'PriceMustNotBeNullTest'".

Reason for test cases: In the code, the *Price* variable is used as a parameter for certain methods which would crash if there was a *null* value for *Price*. The *Price* variable was tested for *int*, *Double*, and *null* because these are the three possible data types for *Price*.

Test Results: The test passes all three cases. Initially, there were errors caught by another testing class. Later, those errors were adapted into this test class in order to handle its own errors.



```
1 package com.parkhere.android;
2
3 import ...
4
5 /**
6  * Created by Stanley on 11/9/2017.
7  */
8
9
10
11 public class PriceMustNotBeNullTest {
12     @Test
13     public void testPriceDoubleShouldBeValid() {
14         assertTrue(CreateListingDetailsActivity.priceIsNotNull(0.5));
15     }
16     @Test
17     public void testPriceIntShouldBeValid() {
18         assertTrue(CreateListingDetailsActivity.priceIsNotNull(5));
19     }
20     @Test
21     public void testPriceCannotBeNull() {
22         assertFalse(CreateListingDetailsActivity.priceIsNotNull(null));
23     }
24 }
25
```

TestResults

All 3 tests passed - 42ms

"C:\Program Files\Android\Android Studio\jre\bin\java" ...

Process finished with exit code 0

Impact: At first, this test was not included. However, after errors occurred, this test was made in order to locate the errors. After running this test, the error was concluded to be *null* values for *Price*.

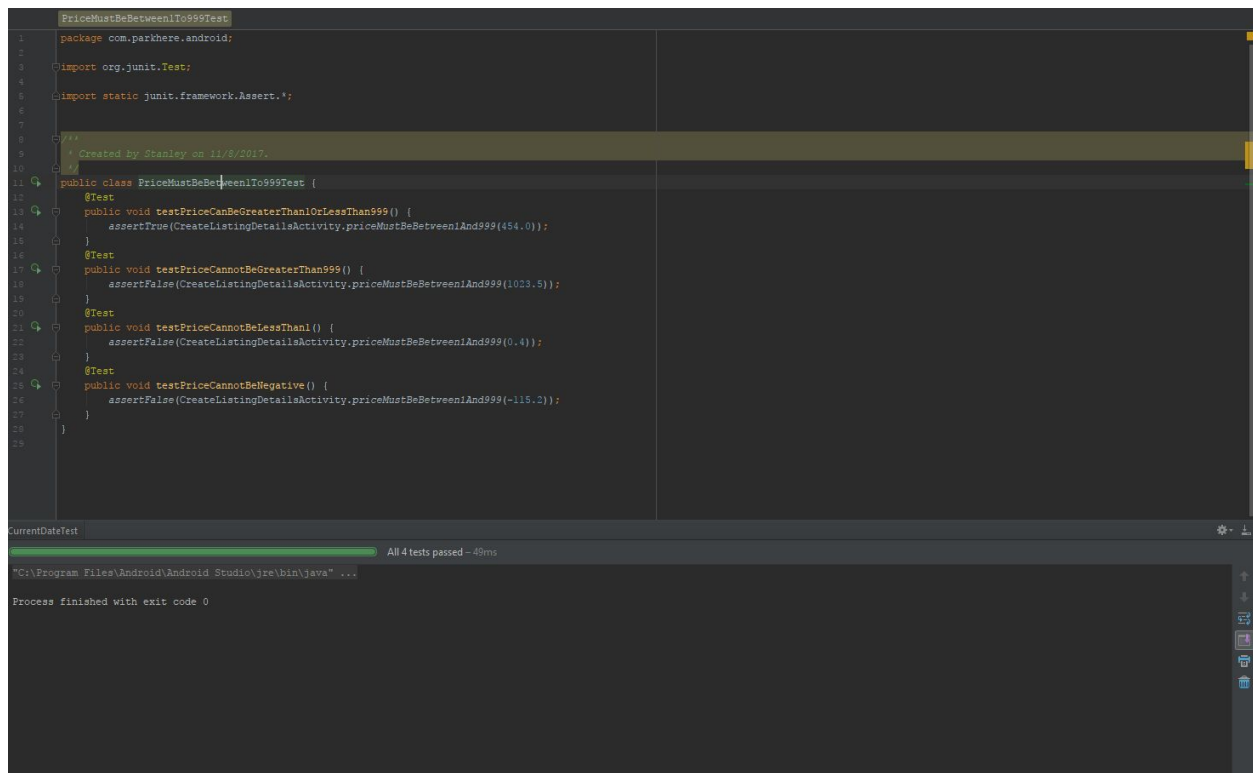
White-Box Testing: PriceMustBeBetween1To999

What is it: This test was included in order to ensure the values for the *Price* variable were not too low or too high.

What it does: This test checks for cases where the *Price* variable is above, below, and inside of its constraints as well as negative values. Execute the test by right clicking on the class and clicking “Run ‘PriceMustBeBetween1To999Test’”.

Reason for test cases: This function sets the price per hour for the listing, so there needed to be a floor and a ceiling for input prices in order to protect the integrity of the application.

Test Results: At first, the test would fail for non-decimal values. After realizing this, the textbox was changed to only accept decimal numbers. The test also caught an error where the greater-than or less-than symbols were reversed. Eventually, the test would pass all its cases.



```
1 package com.parkhere.android;
2
3 import org.junit.Test;
4
5 import static junit.framework.Assert.*;
6
7
8 /**
9  * Created by Stanley on 11/8/2017.
10  */
11 public class PriceMustBeBetween1To999Test {
12     @Test
13     public void testPriceCanBeGreaterThanOrEqualTo999() {
14         assertTrue(CreateListingDetailsActivity.priceMustBeBetween1And999(454.0));
15     }
16     @Test
17     public void testPriceCannotBeGreaterThanOrEqualTo999() {
18         assertFalse(CreateListingDetailsActivity.priceMustBeBetween1And999(1023.5));
19     }
20     @Test
21     public void testPriceCannotBeLessThan1() {
22         assertFalse(CreateListingDetailsActivity.priceMustBeBetween1And999(0.4));
23     }
24     @Test
25     public void testPriceCannotBeNegative() {
26         assertFalse(CreateListingDetailsActivity.priceMustBeBetween1And999(-115.2));
27     }
28 }
```

CurrentDateTest All 4 tests passed - 49ms

"C:\Program Files\Android\Android Studio\jre\bin\java" ...

Process finished with exit code 0

Impact: This test case ensures that the application will only be able to accept *Price* values between 1 and 999 and will not crash if given other inputs.

White-Box Testing: DescriptionMustBeLessThan140Characters

What is it: This test was included in order to ensure that amount of characters in a spot description is properly restricted.

What it does: This test ensures that the string in the *Description* variable is length 140 or less as well as if it is not over length 140. Execute the test by right clicking on the class and clicking “Run ‘DescriptionMustBeLessThan 140CharactersTest’”.

Reason for test cases: Storing numerous strings with extremely long lengths can begin to take up a lot of space in the database with little to no benefit. This test ensures that users are not storing overly excessive amounts of data.

Test Results: After testing this function, it was obvious that there needed to be constraints on the *Description* variable. There are potential errors that will be prevented with the completion of this test case.

The screenshot displays the Android Studio interface. The main editor shows a Java file named `DescriptionMustBeLessThan140CharactersTest`. The code defines a class `DescriptionMustBeLessThan140CharactersTest` with two test methods:

```
package com.parkhere.android;

import ...

/**
 * Created by Stanley on 11/9/2017.
 */

public class DescriptionMustBeLessThan140CharactersTest {

    @Test
    public void testDescriptionCanBeLessThanOrEqualTo140Characters() {
        String string = "aaaaabbbbccccddddddeeeefffffgggghhhhhiiiijjjjkkkkllllmmmmnnnnoooooppppqqqqrrrrssssttttwwwwwvvvvvxxxxxxxxxxxxx";
        assertTrue(CreateListingDetailsActivity.descriptionMustBeLessThan140Characters(string));
    }

    @Test
    public void testDescriptionCannotBeMoreThan140Characters() {
        String string = "aaaaabbbbccccddddddeeeefffffgggghhhhhiiiijjjjkkkkllllmmmmnnnnoooooppppqqqqrrrrssssttttwwwwwvvvvvxxxxxxxxxxxxxzzzzz1234567890123";
        assertFalse(CreateListingDetailsActivity.descriptionMustBeLessThan140Characters(string));
    }
}
```

Below the editor, the Run tab shows the execution results. A green progress bar indicates that all tests passed. The output text reads: "All 2 tests passed - 39ms". At the bottom, the console shows the path `"C:\Program Files\Android\Android Studio\jre\bin\java" ...` and confirms that the process finished with exit code 0.

Impact: This test ensured that the database is not being filled with unnecessarily long string values. As a result, this is no longer an issue.

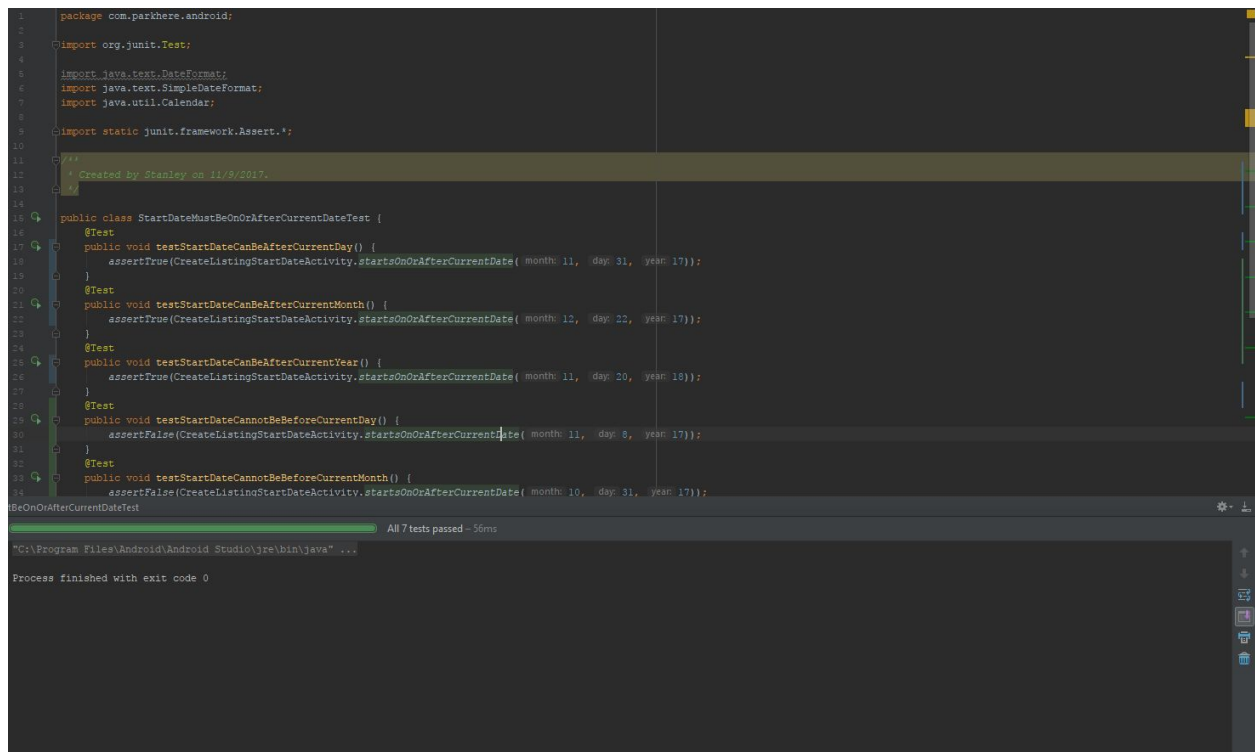
White-Box Testing: StartDateMustBeOnOrAfterCurrentDate

What is it: This test was included in order to ensure that the start date of the listing begins on or after the current date.

What it does: This test ensures that the days, months, and years before the current date are invalid; it also ensures that the current day, month, and year and later are all valid. Execute the test by right clicking on the class and clicking “Run ‘StartDateMustBeOnOrAfterCurrentDateTest’”.

Reason for test cases: In order to keep the listing dates valid, the listings needed constraint that limited them to being posted only on the current day and later.

Test Results: There were repeated errors with the parsing of the dates and times, the proper formatting conventions were adapted. After fixing the errors with the structure of the dates and times, the test passed all seven cases.



```
1 package com.parkhere.android;
2
3 import org.junit.Test;
4
5 import java.text.DateFormat;
6 import java.text.SimpleDateFormat;
7 import java.util.Calendar;
8
9 import static junit.framework.Assert.*;
10
11 /**
12  * Created by Stanley on 11/9/2017.
13  */
14
15 public class StartDateMustBeOnOrAfterCurrentDateTest {
16     @Test
17     public void testStartDateCanBeAfterCurrentDay() {
18         assertTrue(CreateListingStartDateActivity.startsOnOrAfterCurrentDate( month: 11, day: 31, year: 17));
19     }
20     @Test
21     public void testStartDateCanBeAfterCurrentMonth() {
22         assertTrue(CreateListingStartDateActivity.startsOnOrAfterCurrentDate( month: 12, day: 22, year: 17));
23     }
24     @Test
25     public void testStartDateCanBeAfterCurrentYear() {
26         assertTrue(CreateListingStartDateActivity.startsOnOrAfterCurrentDate( month: 11, day: 20, year: 18));
27     }
28     @Test
29     public void testStartDateCannotBeBeforeCurrentDay() {
30         assertFalse(CreateListingStartDateActivity.startsOnOrAfterCurrentDate( month: 11, day: 8, year: 17));
31     }
32     @Test
33     public void testStartDateCannotBeBeforeCurrentMonth() {
34         assertFalse(CreateListingStartDateActivity.startsOnOrAfterCurrentDate( month: 10, day: 31, year: 17));
35     }
36 }
37
38 BeOnOrAfterCurrentDateTest
39
40 All 7 tests passed - 56ms
41
42 "C:\Program Files\Android\Android Studio\jre\bin\java" ...
43
44 Process finished with exit code 0
```

Impact: After running this test, many errors were corrected. Listings were no longer able to be started before the current time. The success of this unit test also allowed for the use of comparing dates and times for the other test cases as well.

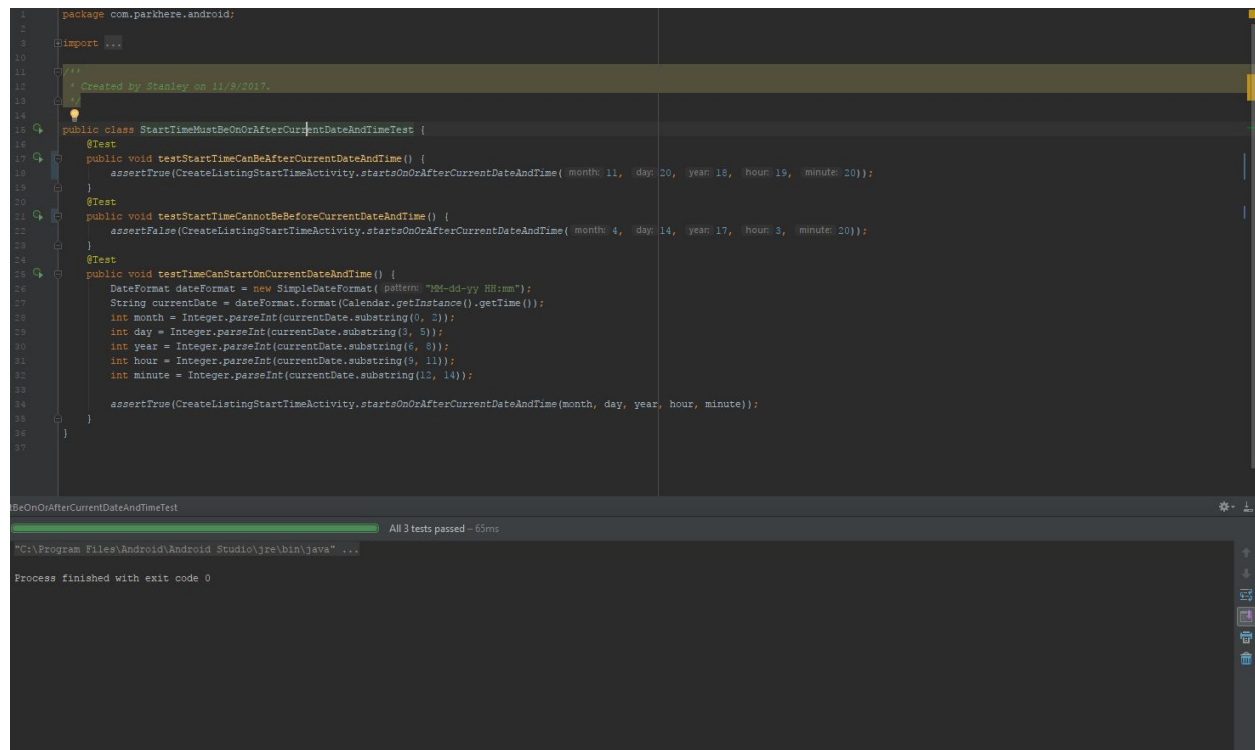
White-Box Test: StartTimeMustBeOnOrAfterCurrentDateAndTime

What is it: This test was included in order to ensure that the start time begins on or after the current date and time.

What it does: This test ensures that the days, months, years, hours, and minutes before the current date are invalid; it also ensures that the current days, months, years, hours and minutes and later are all valid. Execute the test by right clicking on the class and clicking “Run ‘StartTimeMustBeOnOrAfterCurrentDateAndTimeTest’”.

Reason for test cases: In order to keep the listing dates valid, the listings needed constraint that limited them to being posted only on the current day and time and later.

Test Results: Initially, the code would allow for start times before the current time. This was an error with how the times were being parsed. There were also errors with the method of obtaining the dates and times. After seeing these errors, the formatting conventions were changed. The test passes all three cases in the end.



```
1 package com.parkhere.android;
2
3 import ...
4
5 /**
6  * Created by Stanley on 11/9/2017.
7  */
8
9 public class StartTimeMustBeOnOrAfterCurrentDateAndTimeTest {
10     @Test
11     public void testStartTimeCanBeAfterCurrentDateAndTime() {
12         assertTrue(CreateListingStartTimeActivity.startsOnOrAfterCurrentDateAndTime( month 11, day 20, year 18, hour 19, minute 20));
13     }
14     @Test
15     public void testStartTimeCannotBeBeforeCurrentDateAndTime() {
16         assertFalse(CreateListingStartTimeActivity.startsOnOrAfterCurrentDateAndTime( month 4, day 14, year 17, hour 3, minute 20));
17     }
18     @Test
19     public void testTimeCanStartOnCurrentDateAndTime() {
20         DateFormat dateFormat = new SimpleDateFormat( pattern: "MM-dd-yy HH:mm");
21         String currentDate = dateFormat.format(Calendar.getInstance().getTime());
22         int month = Integer.parseInt(currentDate.substring(0, 2));
23         int day = Integer.parseInt(currentDate.substring(3, 5));
24         int year = Integer.parseInt(currentDate.substring(6, 8));
25         int hour = Integer.parseInt(currentDate.substring(9, 11));
26         int minute = Integer.parseInt(currentDate.substring(12, 14));
27
28         assertTrue(CreateListingStartTimeActivity.startsOnOrAfterCurrentDateAndTime(month, day, year, hour, minute));
29     }
30 }
31
32
```

StartTimeMustBeOnOrAfterCurrentDateAndTimeTest

All 3 tests passed - 65ms

"C:\Program Files\Android\Android Studio\jre\bin\java" ...

Process finished with exit code 0

Impact: This was a great result to receive during testing because it ensured that the time constraints were being properly coded since they were implemented similarly to the date constraints. Overall, the testing helped immensely in being able to write future functions and test cases.

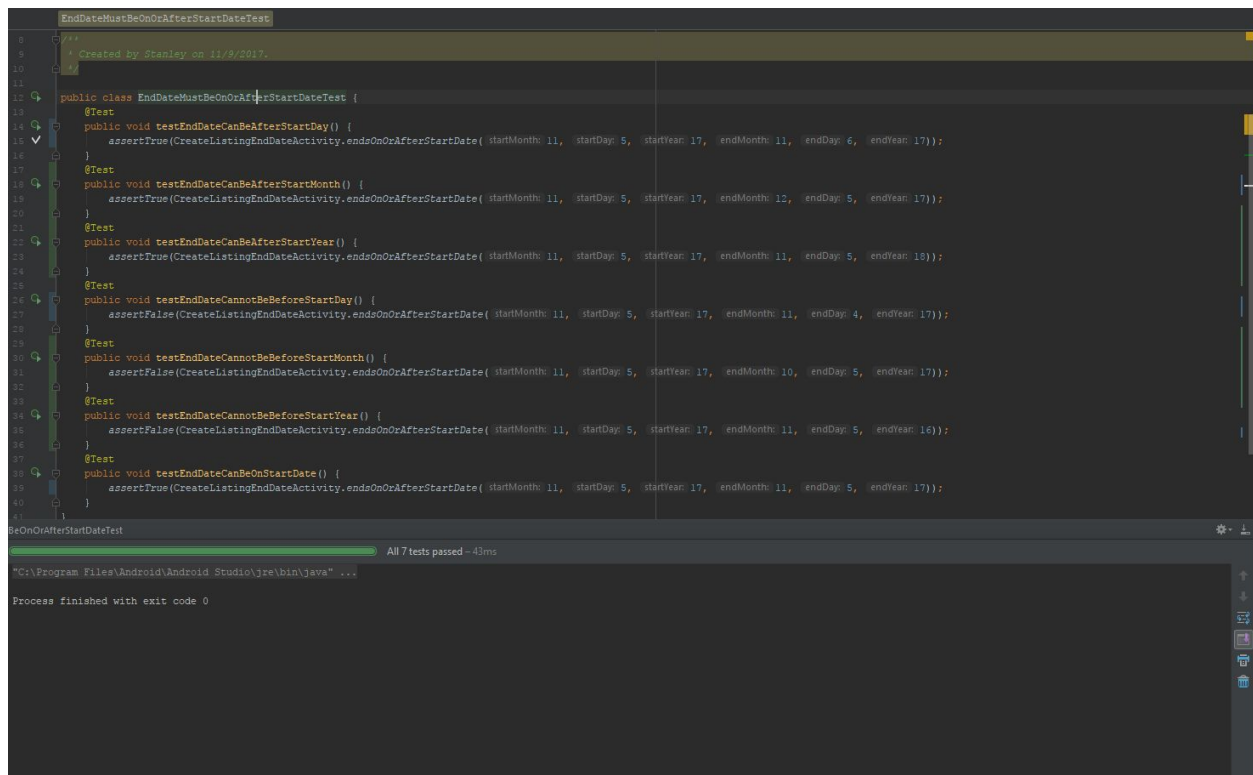
White-Box Testing: EndDateMustBeOnOrAfterStartDate

What is it: This test was included in order to ensure that the end date of the listing finishes on or after the start date.

What it does: This test ensures that the days, months, and years before the start date are invalid; it also ensures that the start day, month, and year and later are all valid. Execute the test by right clicking on the class and clicking “Run ‘EndDateMustBeOnOrAfterStartDateTest’”.

Reason for test cases: Similarly to the StartDate constraints, the *EndDate* constraints needed to restrict the listings so that they can only finish on the start date or later.

Test Results: At this point in testing, the majority of the errors caught by these tests were syntax and formatting errors, which were noticed by examining the flow of the code. After fixing the errors, the test passed all seven cases.



```
1  EndDateMustBeOnOrAfterStartDateTest
2  //
3  Created by Stanley on 11/9/2017.
4
5
6  public class EndDateMustBeOnOrAfterStartDateTest {
7      @Test
8      public void testEndDateCanBeAfterStartDate() {
9          assertTrue(CreateListingEndDateActivity.endsOnOrAfterStartDate( startMonth: 11, startDay: 5, startYear: 17, endMonth: 11, endDay: 6, endYear: 17));
10     }
11     @Test
12     public void testEndDateCanBeAfterStartMonth() {
13         assertTrue(CreateListingEndDateActivity.endsOnOrAfterStartDate( startMonth: 11, startDay: 5, startYear: 17, endMonth: 12, endDay: 5, endYear: 17));
14     }
15     @Test
16     public void testEndDateCanBeAfterStartYear() {
17         assertTrue(CreateListingEndDateActivity.endsOnOrAfterStartDate( startMonth: 11, startDay: 5, startYear: 17, endMonth: 11, endDay: 5, endYear: 18));
18     }
19     @Test
20     public void testEndDateCannotBeBeforeStartDate() {
21         assertFalse(CreateListingEndDateActivity.endsOnOrAfterStartDate( startMonth: 11, startDay: 5, startYear: 17, endMonth: 11, endDay: 4, endYear: 17));
22     }
23     @Test
24     public void testEndDateCannotBeBeforeStartMonth() {
25         assertFalse(CreateListingEndDateActivity.endsOnOrAfterStartDate( startMonth: 11, startDay: 5, startYear: 17, endMonth: 10, endDay: 5, endYear: 17));
26     }
27     @Test
28     public void testEndDateCannotBeBeforeStartYear() {
29         assertFalse(CreateListingEndDateActivity.endsOnOrAfterStartDate( startMonth: 11, startDay: 5, startYear: 17, endMonth: 11, endDay: 5, endYear: 16));
30     }
31     @Test
32     public void testEndDateCanBeOnStartDate() {
33         assertTrue(CreateListingEndDateActivity.endsOnOrAfterStartDate( startMonth: 11, startDay: 5, startYear: 17, endMonth: 11, endDay: 5, endYear: 17));
34     }
35 }
36
37 BeOnOrAfterStartDateTest
38
39 All 7 tests passed - 43ms
40
41 "C:\Program Files\Android\Android Studio\jre\bin\java" ...
42
43 Process finished with exit code 0
```

Impact: This test class was more difficult to create because of the large amount of parameters required to construct each test method. These test cases allow for more confident usage and comparison of the formatted times and dates.

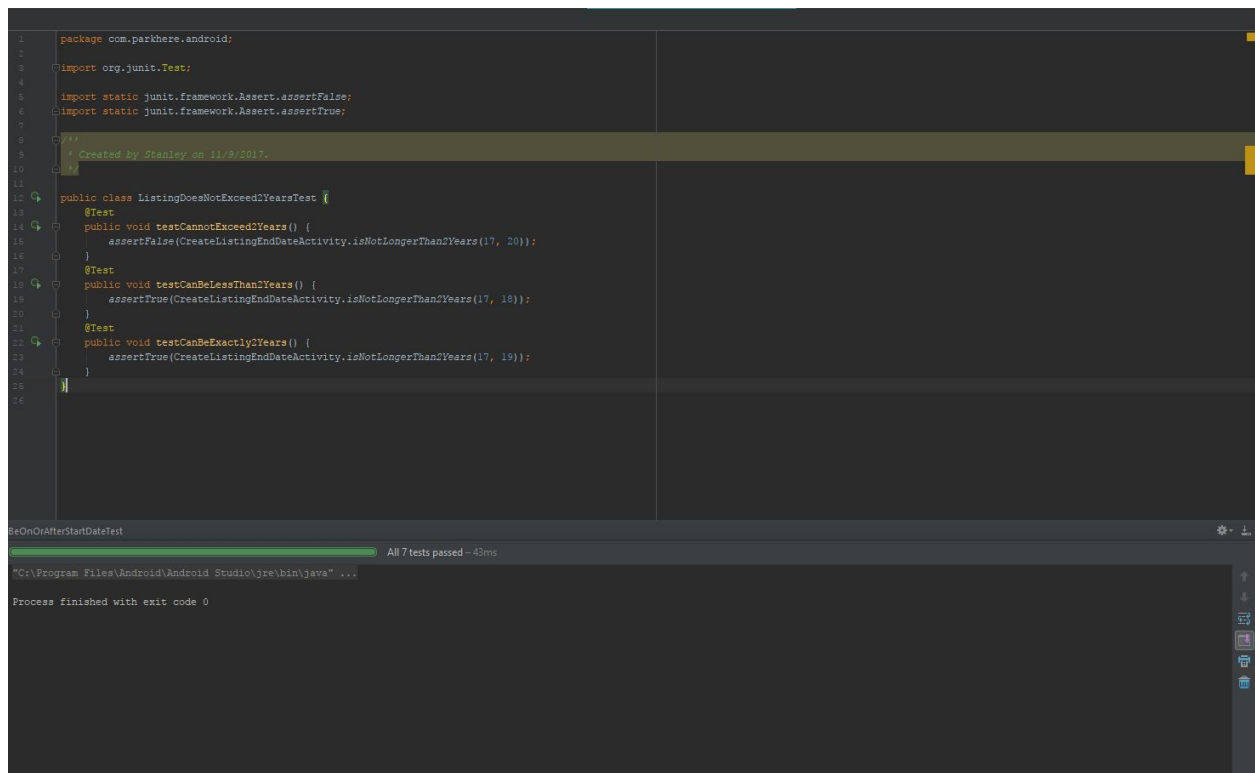
White-Box Testing: ListingMustNotExceed2Years

What is it: This test was included in order to ensure that the listings will be posted with reasonable durations.

What it does: This test ensures that there are no listings which can exceed two years for their duration. It tests for listings exceeding two years, listings less than two years, and listings for exactly two years. Execute the test by right clicking on the class and clicking “Run ‘ListingMustNotExceed2YearsTest’”.

Reason for test cases: The reasoning for this test case is a similar reasoning for testing against unreasonably large strings: preventing excess data storage. With the prevention of users posting indefinite listings, the load on the database will be lightened in the future.

Test Results: After testing this function, there was another issue similar to a previous test class. It was necessary to restrict the length of the postings in order to prevent indefinite listings..



```
1 package com.parkhere.android;
2
3 import org.junit.Test;
4
5 import static junit.framework.Assert.assertFalse;
6 import static junit.framework.Assert.assertTrue;
7
8 /**
9  * Created by Stanley on 11/9/2017.
10  */
11
12 public class ListingDoesNotExceed2YearsTest {
13     @Test
14     public void testCannotExceed2Years() {
15         assertFalse(CreateListingEndDateActivity.isNotLongerThan2Years(17, 20));
16     }
17     @Test
18     public void testCanBeLessThan2Years() {
19         assertTrue(CreateListingEndDateActivity.isNotLongerThan2Years(17, 18));
20     }
21     @Test
22     public void testCanBeExactly2Years() {
23         assertTrue(CreateListingEndDateActivity.isNotLongerThan2Years(17, 19));
24     }
25 }
26
```

BeOnOrAfterStartDateTest All 7 tests passed - 43ms

"C:\Program Files\Android\Android Studio\jre\bin\java" ...

Process finished with exit code 0

Impact: With the addition of the constraints of the listing duration length, the database will be protected from overly excessive amounts of unneeded data.

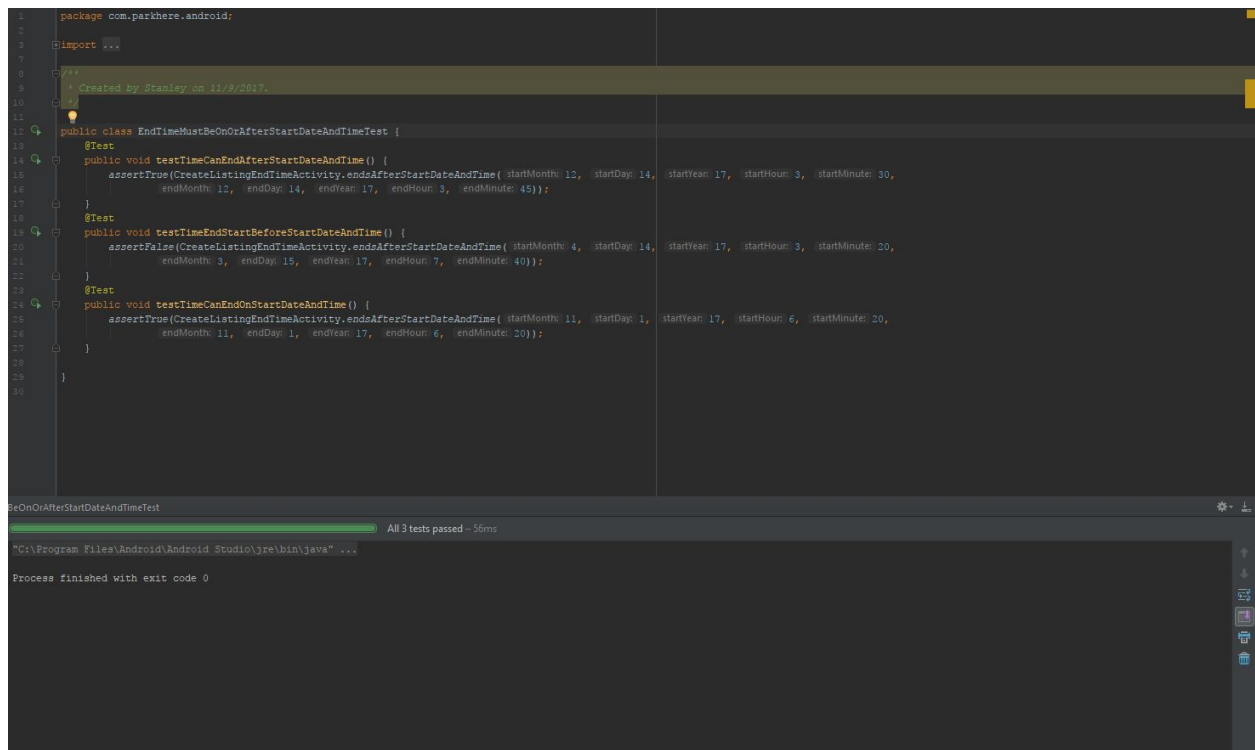
White-Box Testing: EndTimeMustBeOnOrAfterStartDateAndTime

What is it: This test was included in order to ensure that the end time of the listing can only finish after the start date and time.

What it does: This test ensures that the listing end time is valid only after the start date and time. The test also checks for if the listing is invalid if it does have an end time before the start date and time. Execute the test by right clicking on the class and clicking “Run ‘EndTimeMustBeOnOrAfterStartDateAndTimeTest’”.

Reason for test cases: Like the previous test cases, it was important to ensure that the end time had similar constraints as the start date, start time, and end date.

Test Results: The test case for the final activity in the *CreateListing* activity sequence created a lot of errors because it had the biggest amount of variables being passed through to it. As a result of this, there were many errors where the data was being passed through incorrectly. At the end, the test passed all its cases.



```
1 package com.parkhere.android;
2
3 import ...
4
5 /**
6  * Created by Stanley on 11/9/2017.
7  */
8
9 public class EndTimeMustBeOnOrAfterStartDateAndTimeTest {
10     @Test
11     public void testTimeCanEndAfterStartDateAndTime() {
12         assertTrue(CreateListingEndTimeActivity.endsAfterStartDateAndTime( startMonth: 12, startDay: 14, startYear: 17, startHour: 3, startMinute: 30,
13             endMonth: 12, endDay: 14, endYear: 17, endHour: 3, endMinute: 45));
14     }
15     @Test
16     public void testTimeEndStartBeforeStartDateAndTime() {
17         assertFalse(CreateListingEndTimeActivity.endsAfterStartDateAndTime( startMonth: 4, startDay: 14, startYear: 17, startHour: 3, startMinute: 20,
18             endMonth: 3, endDay: 15, endYear: 17, endHour: 7, endMinute: 40));
19     }
20     @Test
21     public void testTimeCanEndOnStartDateAndTime() {
22         assertTrue(CreateListingEndTimeActivity.endsAfterStartDateAndTime( startMonth: 11, startDay: 1, startYear: 17, startHour: 6, startMinute: 20,
23             endMonth: 11, endDay: 1, endYear: 17, endHour: 6, endMinute: 20));
24     }
25 }
```

EndTimeMustBeOnOrAfterStartDateAndTimeTest All 3 tests passed - 56ms

"C:\Program Files\Android\Android Studio\jre\bin\java" ...

Process finished with exit code 0

Impact: After being able to test and solve the last part of this activity sequence, the *CreateListing* activity sequence was far more viable for use.