1. Follow the lab lecture about JavaDoc and unit tests then continue from step 2.
2. Create a new project in Android Studio named **ListyCity**.
3. Create a new java class named City **com.example.listycity.** This class shall have two variables **city** and **province**. Create getters of these variables.

```
public class City  {
   private String city;
   private String province;

   City(String city, String province){
      this.city = city;
      this.province = province;
   }

   String getCityName(){
      return this.city;
   }

   String getProvinceName(){
      return this.province;
   }
}
```

4. Add Javadoc comments for City, its variables and functions. e.g.

```
/**
* This is a class that defines a City.
*/
public class City {....}
```

5. Create a new class CityList under **com.example.listycity.** Now we will write JavaDoc comments for this class and also test this class by writing unit tests.

```
/**
* This is a class that keeps a list of city objects
*/
public class CityList {
}
```

6. Declare a list to hold the city objects.

```
private List<City> cities = new ArrayList<>();
```

7. Implement a method to add city objects to this list. If a city already exists then throw Exception. Here we also have written JavaDoc comments with ***@param*** tag.

```
/**
* This adds a city to the list if the city does not exist
* @param city
*      This is a candidate city to add
*/
public void add(City city) {
  if (cities.contains(city)) {
     throw new IllegalArgumentException();
  }
  cities.add(city);
  }
```

8. Create another method to get a list of city objects sorted according to the city name. Here we also have written JavaDoc comments with **@*return*** tag.

```
/**
* This returns a sorted list of cities
* @return
*      Return the sorted list
*/
public List<City> getCities() {
  List<City> list = cities;
  Collections.sort(list);
  return list;
}
```

9. In the **CityList** class the **Collections.*sort*(list);** line shows an error. We want to sort the name of the cities but we are trying to sort the city objects. To sort an Object by its property, we have to make the Object ***implement*** the ***Comparable*** interface and ***override*** the ***compareTo()*** method. *Lists* (and arrays) of objects that implement *Comparable* interface can be sorted automatically by ***Collections.sort(),*** therefore we also need to implement the *compareTo() method*. All wrapper classes and *String* class in java implement *Comparable* interface. *Wrapper* classes are compared by their values, and strings are compared lexicographically. To know more visit the link below: https://howtodoinjava.com/java/collections/java-comparable-interface/

10. Go to the City class and make sure it looks like the following with ***Comparable<city>*** interface and ***compareTo()*** method implementation:

```
  @Override
  public int compareTo(Object o) {
     City city = (City) o;
     return this.city.compareTo(city.getCityName());          // this.city refers to the city name
  }
}
```

11. The `compareTo` method compares the current city's name (`this.city`) with the name of the other city (`city.getCityName()`). Now the error disappears and you can sort the city objects according to the city names. If two cities are equal then *compareTo()* method returns 0.

12. Finally, the CityList class:

```
package com.example.listycity;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/**
* This is a class that keeps track of a list of city objects
*/
public class CityList {
  private List<City> cities = new ArrayList<>();

  /**
   * This adds a city to the list if the city does not exist
   * @param city
   *      This is a candidate city to add
   */
  public void add(City city) {
      if (cities.contains(city)) {
          throw new IllegalArgumentException();
      }
      cities.add(city);
  }

  /**
   * This returns a sorted list of cities
   * @return
   *      Return the sorted list
   */
  public List<City> getCities() {
      List<City> list = cities;
      Collections.sort(list);
      return list;
  }

}
```

13. Select ***Tools -> Generate*** JavaDoc to create HTML java documentation from your JavaDoc comments. Select the ***"Module app"*** and select the ***output directory***.

14. If you encounter an error while generating javadoc, locate android.jar and add it to your gradle dependencies as follows:

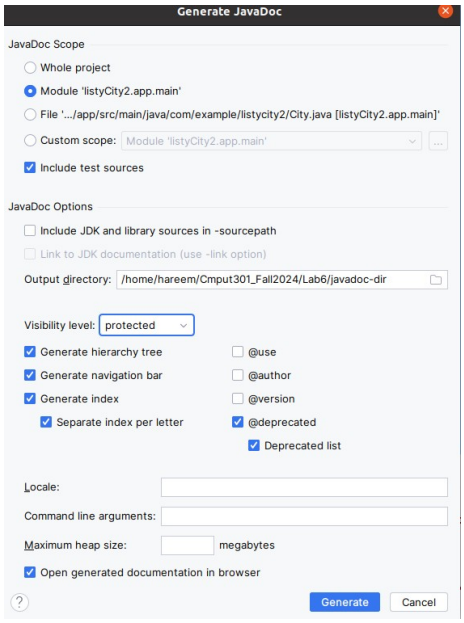
    implementation(files("/home/hareem/Android/Sdk/platforms/android-34/android.jar"))

15. To find the android.jar

- ✘ First, shift from Android view to the Project view
- ✘ Go to **External Libraries** and click to expand
- ✘ You will see **android.jar** under `Android API XX` where XX is the API level (like `android-34` for API level 34). Example path below: `/home/yourusername/Android/Sdk/platforms/android-34/android.jar`
- ✘ Now right click and copy the absolute path.
  This `android.jar` file contains the Android classes needed for your Javadoc generation.

  https://stackoverflow.com/questions/71721535/android-studio-fails-to-generate-javadocs-due-to-packages-not-existing/73102343#73102343

16. Remove the (implementation files...) line once you've generated your javadoc.
17. **Note:** By default private methods may not appear in the generated Javadoc unless you include them by setting the right visibility at the time of generating javadocs.



18. Now let's move to writing **unit tests.** We will use **junit 5** for this lab because it has more modern testing features. By default junit 4 is included in every project. To use junit 5 you'll have to include

    ```
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.0.1'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.0.1'
    ```

    under *dependencies* on app Gradle (*build.gradle(Module:app)*) file. After adding sync project.

    From official docs:

    `Junit-jupiter-api` JUnit Jupiter API for writing tests and extensions.

    `Junit-jupiter-engine` JUnit Jupiter test engine implementation, only required at runtime.

19. Under **com.example.listycity(test)** folder, create a new class **CityListTest** to test the functionalities of our **CityList** class.

20. Create two private methods for creating a mock city object and adding to the cityList.

    ```
    package com.example.listycity;
    import org.junit.jupiter.api.Test;
    import static org.junit.jupiter.api.Assertions.*;

    class CityListTest {
      private CityList mockCityList() {
        CityList cityList = new CityList();
        cityList.add(mockCity());
        return cityList;
      }

      private City mockCity() {
        return new City("Edmonton", "Alberta");
      }
    }
    ```

21. Write a test for the add() method which is in CityList class. Write it under the *mockCity()* method. Add city objects using *add()* method and check if the addition is successful by

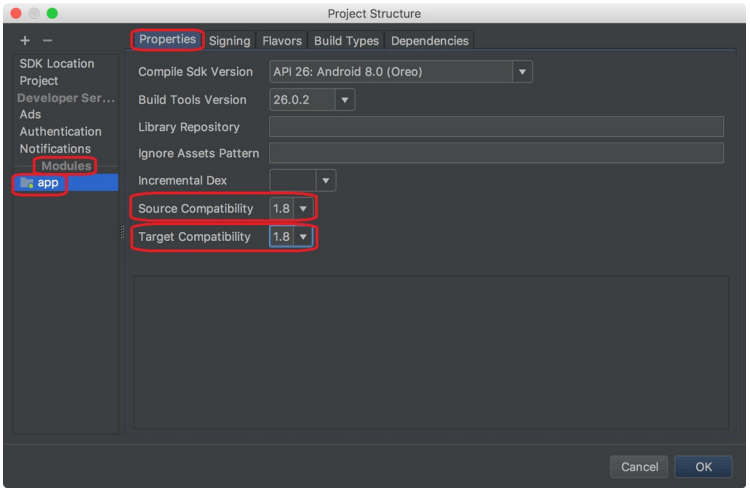
assertEquals() and *assertTrue()*. We need to add ***@Test*** before any test method to identify it as a junit test.

```
@Test
void testAdd() {
   CityList cityList = mockCityList();
   assertEquals(1, cityList.getCities().size());
   City city = new City("Regina", "Saskatchewan");
   cityList.add(city);
   assertEquals(2, cityList.getCities().size());
   assertTrue(cityList.getCities().contains(city));
}
```

22. Write another test method to check if an IllegalArgumentException is thrown when adding a city that already exists in the list.

```
@Test
void testAddException() {
   CityList cityList = mockCityList();
   City city = new City("Yellowknife", "Northwest Territories");
   cityList.add(city);
   assertThrows(IllegalArgumentException.class, () -> {
      cityList.add(city);
   });
}
```

23. If you get an error like "Lambda expressions not supported at this language level" Then the solution: Change the java language to version 1.8 (Go to **File → Project Structure**)

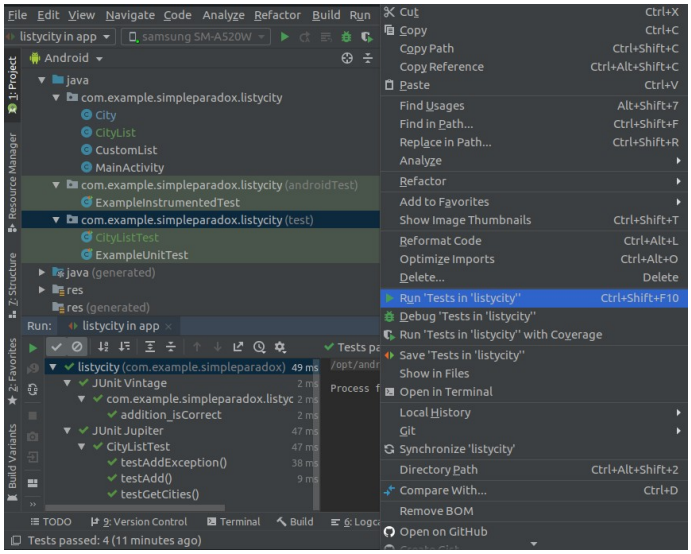


24. Write another test method to test the behavior of *getCities()* method in the *CityList* class.

```
@Test
void testGetCities() {
   CityList cityList = mockCityList();
// This line checks if the first city in the cityList (retrieved by cityList.getCities().get(0))
// is the same as the city returned by mockCity()
```

```
        assertEquals(0, mockCity().compareTo(cityList.getCities().get(0)));
    // This pushes down the original city
        City city = new City("Charlottetown", "Prince Edward Island");
        cityList.add(city);
    // Now the original city should be at position 1
        assertEquals(0, city.compareTo(cityList.getCities().get(0)));
        assertEquals(0, mockCity().compareTo(cityList.getCities().get(1)));
    }
```

25. Run the tests (right-click on the test folder) and see the test output as following:



26. If you get this error “Test events not received”, then add following lines to **android section** of gradle

```
    tasks.withType<Test>{
            useJUnitPlatform()
    }
```

Right click and run tests again and the error should be gone.

27. The tests can be written before the real implementation of the *CityList* class. We can implement all the tests first and they must fail as there is no implementation of *CityList* class and its methods. Then we can give implementation of the class and methods. Then our tests would pass. This is called test-driven development. You can also follow this method.

28. Complete *CityListTest* class

app × CityListTest.java ×

```
1   package com.example.simpleparadox.listycity;
2
3   import org.junit.jupiter.api.Test;
4
5
6   import static org.junit.jupiter.api.Assertions.*;
7
8   class CityListTest {
9
10      private CityList mockCityList() {
11          CityList cityList = new CityList();
12          cityList.add(mockCity());
13          return cityList;
14      }
15
16      private City mockCity() {
17          return new City( city: "Edmonton",  province: "Alberta");
18      }
19
20      @Test
21      void testAdd() {
22          CityList cityList = mockCityList();
23
24          assertEquals( expected: 1, cityList.getCities().size());
25
26          City city = new City( city: "Regina",  province: "Saskatchewan");
27          cityList.add(city);
28
29          assertEquals( expected: 2, cityList.getCities().size());
30          assertTrue(cityList.getCities().contains(city));
31      }
32
33      @Test
34      void testAddException() {
35          CityList cityList = mockCityList();
36
37          City city = new City( city: "Yellowknife",  province: "Northwest Territories");
38          cityList.add(city);
39
40          assertThrows(IllegalArgumentException.class, () -> {
41              cityList.add(city);
42          });
43      }
44
45      @Test
46      void testGetCities() {
47          CityList cityList = mockCityList();
48
49          assertEquals( expected: 0, mockCity().compareTo(cityList.getCities().get(0)));
50
51          City city = new City( city: "Charlottetown",  province: "Prince Edward Island");
52          cityList.add(city);
53
54          assertEquals( expected: 0, city.compareTo(cityList.getCities().get(0)));
55          assertEquals( expected: 0, mockCity().compareTo(cityList.getCities().get(1)));
56      }
57
58  }
```

CityListTest

29. Here is the gradle level dependencies

```
dependencies { this: DependencyHandlerScope
    implementation(files( ...paths: "/home/hareem/Android/Sdk/platforms/android-34/android.jar"))
    implementation("androidx.appcompat:appcompat:1.6.1")
    implementation("com.google.android.material:material:1.11.0")
    implementation("androidx.constraintlayout:constraintlayout:2.1.4")
    //testImplementation("junit:junit:4.13.2")
    testImplementation ("org.junit.jupiter:junit-jupiter-api:5.0.1")
    testRuntimeOnly ("org.junit.jupiter:junit-jupiter-engine:5.0.1")
    androidTestImplementation("androidx.test.ext:junit:1.1.5")
    androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")
}
```