Assignment 4: Android Development

In this assignment you will build a simple Android app similar in spirit to the one shown at the end of the demo for the Android lesson. The app is a cost-of-living wizard that:

1. Takes as input:
    a. A *Base Salary* to be converted.
        i. This input should be an **integer** and **non negative**.
        ii. This input should be provided to the app through an EditText widget initialized to “0”.
    b. A *Current City*, which provides the baseline for the cost-of-living computation.
        i. This input should be one of the cities listed in the cost of living table provided below.
        ii. This input should be provided to the app through a Spinner widget initialized to “Atlanta, GA”.
    c. A *New City*, whose cost of living will be used to convert the base salary.
        i. This input should be one of the cities listed in the cost of living table provided below.
        ii. This input should be provided to the app through a Spinner widget initialized to “Atlanta, GA”.
2. Produces as output
    a. The *Target Salary*, which is the salary required to have a standard of living in the New City that is equivalent to the standard of living that the Base Salary provides in the Current City.
        i. This output should be an **integer** calculated by (1) multiplying the Base Salary by the New City’s cost of living index, (2) dividing the resulting number by the Current City’s cost of living index, and (3) rounding the result of the division to the **nearest whole dollar**. Important: **do not round or truncate intermediate results** in the calculation.
        ii. This output should be shown using a non editable text field initialized to “0” and (re)computed whenever any of the input fields is changed. If the inputs are invalid, the output should be set to “” (i.e., the empty string).

Cost of Living Table

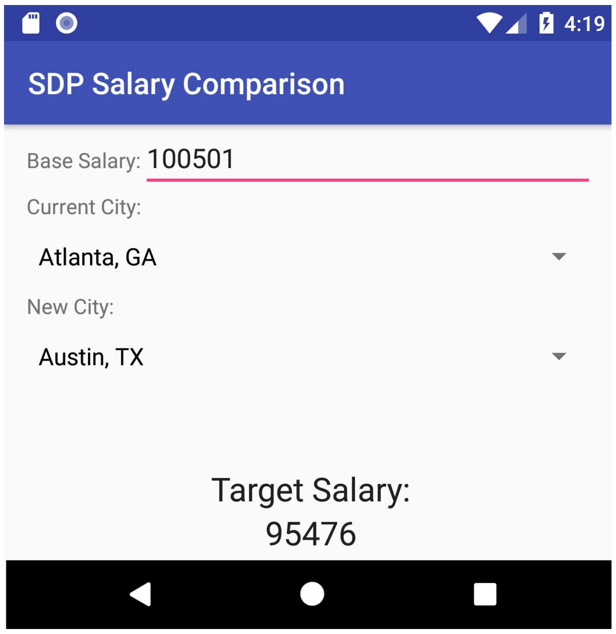
City	**Cost of Living Index**
Atlanta, GA	160
Austin, TX	152
Boston, MA	197
Honolulu, HI	201
Las Vegas, NV	153
Mountain View, CA	244

New York City, NY	232
San Francisco, CA	241
Seattle, WA	198
Springfield, MO	114
Tampa, FL	139
Washington D.C.	217

Error Messages

If the user provides an **invalid** input (i.e., an invalid Base Salary), the app should display an error message associated to the Base Salary entry using the capabilities provided (or better, inherited) by the EditText widget. The error message should be **exactly** "Invalid salary". If you do this correctly, the results should be a floating text "Invalid salary", with error mark "❗", right next to Base Salary widget, as shown in Screenshot 4 below.

Below are five (slightly cropped) screenshots of the app that provide examples of normal behavior (Screenshots 1, 2, 3, and 5) and errors (Screenshot 4). You can also treat the screenshots as example test results and use them to check that your app behaves as intended.

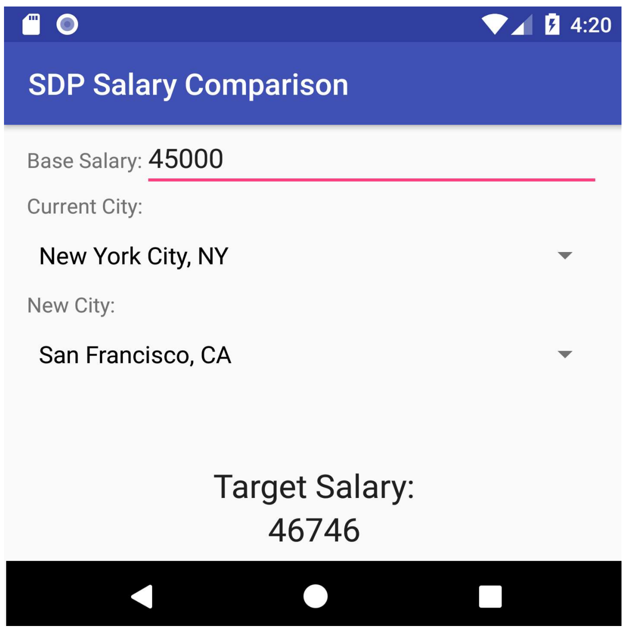

SDP Salary Comparison

Base Salary: 100501

Current City: Atlanta, GA

New City: Austin, TX

Target Salary:
95476

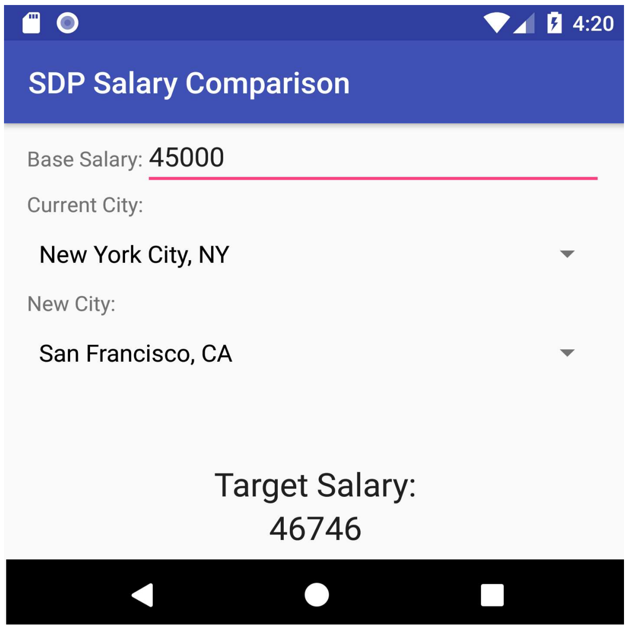

SDP Salary Comparison

Base Salary: 45000

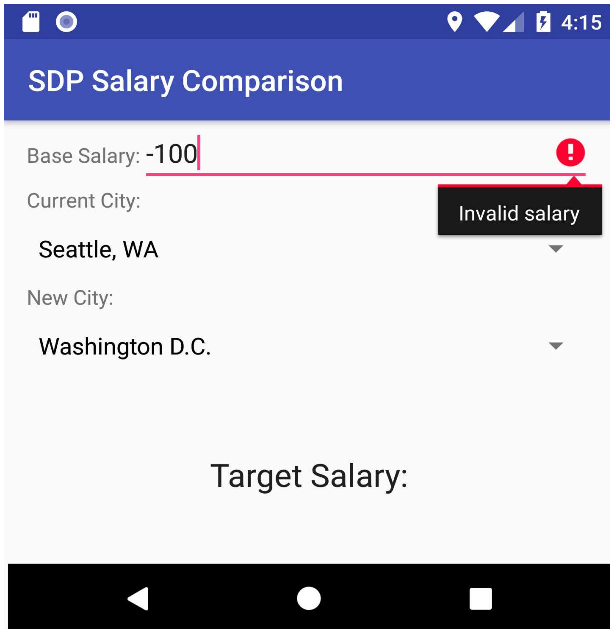
Current City: New York City, NY

New City: San Francisco, CA

Target Salary:
46746

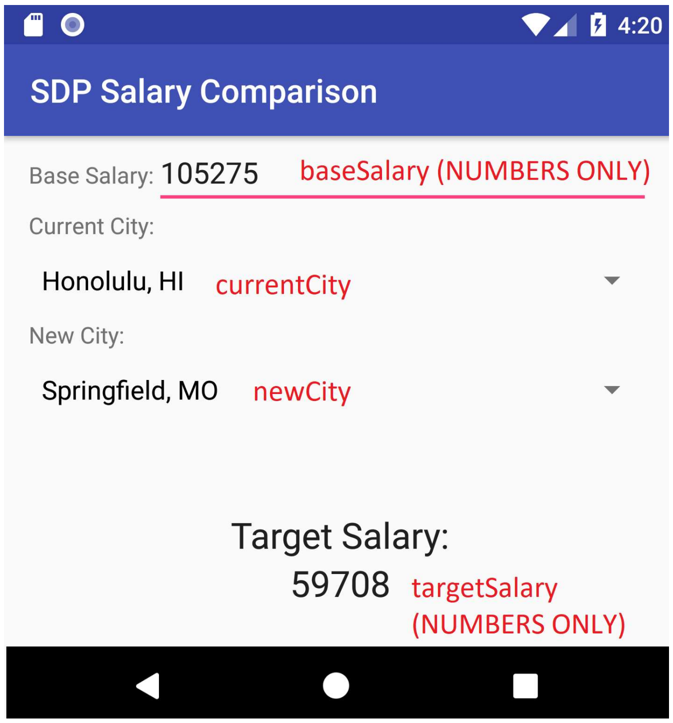


Base Salary: 45000
Current City: New York City, NY
New City: San Francisco, CA
Target Salary: 46746



Base Salary: -100
Current City: Seattle, WA
New City: Washington D.C.
Target Salary:

We suggest that you try to keep your user interface (UI) similar to the one shown, but you don't have to. However, **you must make sure to use exactly the same identifiers that we show in the next figure for the key widgets in the UI**. This is very important, as we will use these identifiers to automatically test your app. The identifiers are also listed next to the figure for your (copy-and-paste) convenience.



Base Salary: 105275
Current City: Honolulu, HI
New City: Springfield, MO
Target Salary: 59708

Identifiers:

- `"baseSalary"`
- `“currentCity”`
- `“newCity”`
- `"targetSalary"`

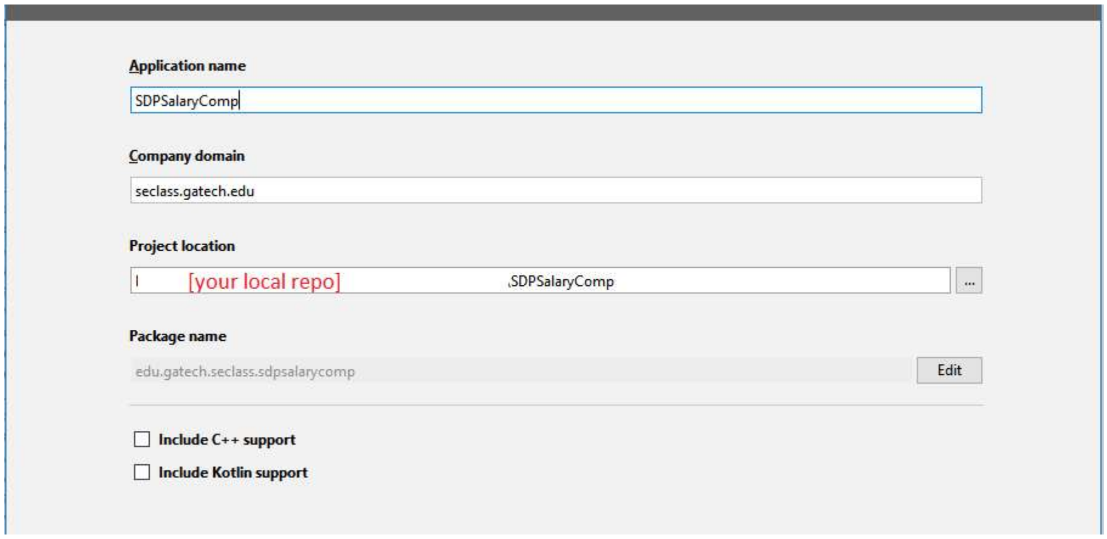
For example, in the XML layout file for your app, the entry for the text field used to input the *Salary* should have the following ID: `android:id="@+id/baseSalary"`.

Please note that, as described in the detailed instructions below, we took some steps to help you make sure that you used the right identifiers and strings.

Detailed Instructions

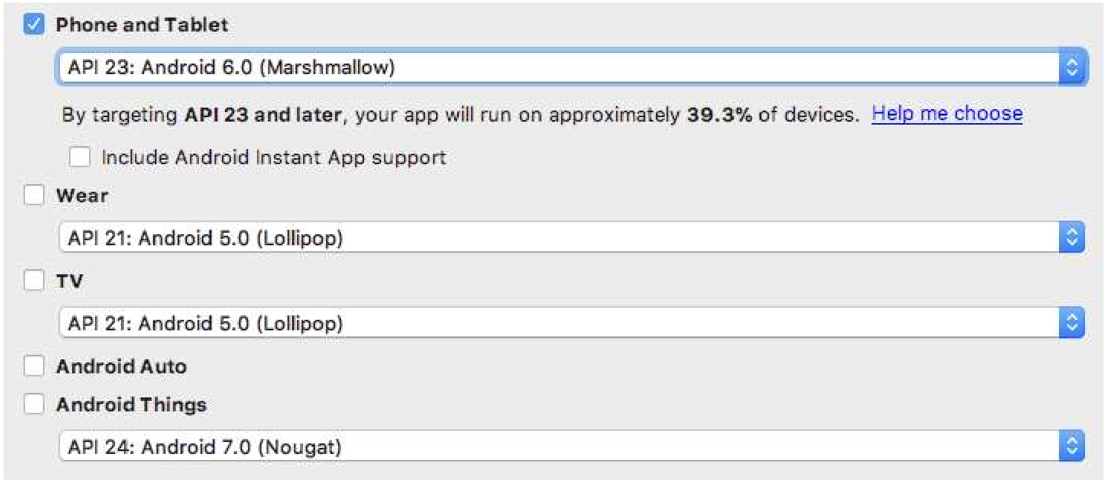
To complete the assignment you must perform the following tasks:

1. In the root of **your assigned individual GitHub repository**, create a directory called `Assignment4`. Hereafter, we will refer to this directory in your local repo as `<dir>`.
2. Using Android Studio, create an Android app project called “SDPSalaryComp” in `<dir>`. Check that this generates a directory SDPSalaryComp under `<dir>`.



Application name
SDPSalaryComp
Company domain
seclass.gatech.edu
Project location
[your local repo] \SDPSalaryComp
Package name
edu.gatech.seclass.sdpsalarycomp

   Ensure company domain is entered as “seclass.gatech.edu”, which will result in the package **`edu.gatech.seclass.sdpsalarycomp`.**
3. Select “**API 23: Android 6.0 (Marshmallow)**” as the **minimum sdk** for your app.



Phone and Tablet ✓
API 23: Android 6.0 (Marshmallow)
By targeting **API 23 and later**, your app will run on approximately **39.3%** of devices. Help me choose
Include Android Instant App support
Wear
API 21: Android 5.0 (Lollipop)
TV
API 21: Android 5.0 (Lollipop)
Android Auto
Android Things
API 24: Android 7.0 (Nougat)

   This `minSdkVersion` should be reflected in your build.grade file in the project.
4. Create an “Empty Activity” and call it `SDPSalaryCompActivity`
5. Download and save the archive available here, which contains three files provided to help you make sure that you used the right identifiers and strings, as we mentioned above. **These files will not check everything for you**, but if you use them, they will

prevent many simple errors. The files, which we describe how to use in the rest of the steps, are:

- **`SanityCheck.java`** prevents your app from compiling if certain identifiers, your activity name, or your package name are incorrect.
- **`strings.xml`** provides some constant strings that you can reference to avoid typos in error messages and city names. You are welcome to add additional strings to it, but please keep in mind that **our tests will expect the identifiers and error messages to match the ones provided, exactly**.
- **`SampleTests.java`** contains Espresso tests similar to the tests we will run on your code.

6. Extract `strings.xml` from the archive and copy it to your project at `<dir>/SDPSalaryComp/app/src/main/res/values`
7. Implement the primary functionality of the app in `SDPSalaryCompActivity.`
8. Define the IDs for the key widgets in the app as described above.
9. Extract `SanityCheck.java` from the archive and copy it to your project at `<dir>/SDPSalaryComp/app/src/main/java/edu/gatech/seclass/sdpsalarycomp`
   - Rebuild the project in Android Studio.
   - If the project does not compile, and you have added all the required identifiers to your layout, it should mean that there are issues with your activity name/package, your identifiers, or both. **Make sure to correct your project, and not the provided `SanityCheck.java` file.**
10. Extract `SampleTests.java` from the archive and copy it to your project at `<dir>/SDPSalaryComp/app/src/androidTest/java/edu/gatech/seclass/sdpsalarycomp`
    - Add the necessary dependencies to your build.gradle.
    - Rebuild the project in Android Studio
    - Run the tests by either setting up a new run configuration or right clicking the file and choosing "Run 'SampleTests'".
11. Create a `manual.md` file (in Github flavored MD format) that describes how to use the app. Put the file in `<dir>`, not in `SDPSalaryComp`. Think of this file as a (very concise) user manual. You should not need more than one page for the manual. **It should use at least some markdown formatting.** Feel free to add screenshots to the manual, but that is not mandatory. You can view your markdown file in Github or using an MD viewer.
12. Commit and push your project from within Android Studio (or from the command line, if you know what you are doing). Doing it from Android Studio should help ensure that all the required files are committed. Be sure to use your existing, assigned repository, rather than creating a new repository from Android Studio. No matter how you commit and push your project, we strongly recommend that you (1) clone your repo in a different directory, (2) open the project in Android Studio from this new directory, (3) compile the project, and (4) run it on a (virtual) device that does not already have the app installed (i.e., you may have to remove the app from the device if you have ran the app there before).
13. As usual, submit your solution by doing the following:

    - Pushing your code to your assigned remote GitHub repository.
    - Submitting the final commit ID for your submission on Canvas. (You can get your commit ID by running "git log -1" and copying the hexadecimal ID it produces.)

Notes

- **You should commit early and often**. Verify that you are able to correctly push your code to the assigned repository as soon as possible. You can perform multiple commits and work on multiple branches as you produce your solution. This is not only fine, but actually encouraged. Just make sure that your final solution is committed to the `master` branch.
- You should complete the assignment using Android Studio 3.*. Earlier versions of Android Studio may work as well, but we have not tested our solution there.
- Feel free to **take inspiration** from online resources when developing your app. However, be careful not to copy and paste entire pieces of functionality. The tool we use to identify cases of plagiarism is likely to have access to the same online resources that are available to you.
- In case you want to use automated testing for your app, you may find Espresso and either Barista (documentation here) or Android Studio Test Recorder useful. This is completely optional; that is, developing tests for your app, whether manual or automated, is not required. If you decide to use Espresso, here's a couple of potentially useful tips:
    - If anything covers your fields or buttons (even invisible boxes), the tests may fail to complete.
    - If you use buttons that call private methods, Espresso may be unable to click the button.