Software Engineering Large Practical: Preferences, storage, and testing

Stephen Gilmore
(Stephen.Gilmore@ed.ac.uk)
School of Informatics

November 9, 2016

Contents

A simple counter activity

- We created a simple counter activity which recorded button clicks.

```
1  public class MainActivity extends AppCompatActivity {
2
3      private int clicks = 0;
4      private static final String TAG = "MainActivity";
5
6      private void setClicks (int c) {
7          clicks = c;
8      }
9
10     private int getClicks () {
11         return clicks ;
12     }
13     ...
14 }
```

Adding listeners to buttons

```
1  Button b = (Button) findViewById(R.id.button);
2  b.setOnClickListener(new View.OnClickListener(){
3      @Override
4      public void onClick(View v) {
5          clicks++;
6      }
7  });
```

```
1  FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
2  fab.setOnClickListener(new View.OnClickListener() {
3      @Override
4      public void onClick(View view) {
5          Snackbar.make(view, "Clicks so far: " + clicks,
6              Snackbar.LENGTH_LONG)
7                  .setAction("Action", null).show();
8      }
9  });
```

Making values persistent

- In order to have values retained between user sessions with an application Android provides a framework for storing *key-value pairs of primitive data types*.
- The `SharedPreferences` class can be used to save any primitive data: booleans, floats, ints, longs, and strings.
- Preference files can be named, if you need more than one.
- Stored values can be restored in the `onCreate` method.
- Updated values can be written in the `onStop` method.
- Any value can be considered a preference: it doesn't have to be user preferences (such as *"Sounds: on/off"* etc).

`developer.android.com/guide/topics/data/data-storage.html`

Reading in saved preferences

```
1  private static final String PREFS_NAME = "MyPrefsFile";
```

```
1  // Restore preferences (in the ''onCreate'' method)
2  SharedPreferences settings =
3      getSharedPreferences(PREFS_NAME, MODE_PRIVATE);
4
5  // use 0 as the default value
6  int storedClicks = settings.getInt("storedClicks", 0);
7
8  setClicks( storedClicks );
```

developer.android.com/guide/topics/data/data-storage.html

Writing out updated preferences

```
1  @Override
2  protected void onStop(){
3      super.onStop();
4
5      // All objects are from android.context.Context
6      SharedPreferences settings =
7          getSharedPreferences(PREFS_NAME, MODE_PRIVATE);
8
9      // We need an Editor object to make preference changes.
10     SharedPreferences.Editor editor = settings.edit();
11     editor.putInt("storedClicks", getClicks());
12
13     // Apply the edits!
14     editor.apply();
15 }
```

---

developer.android.com/guide/topics/data/data-storage.html

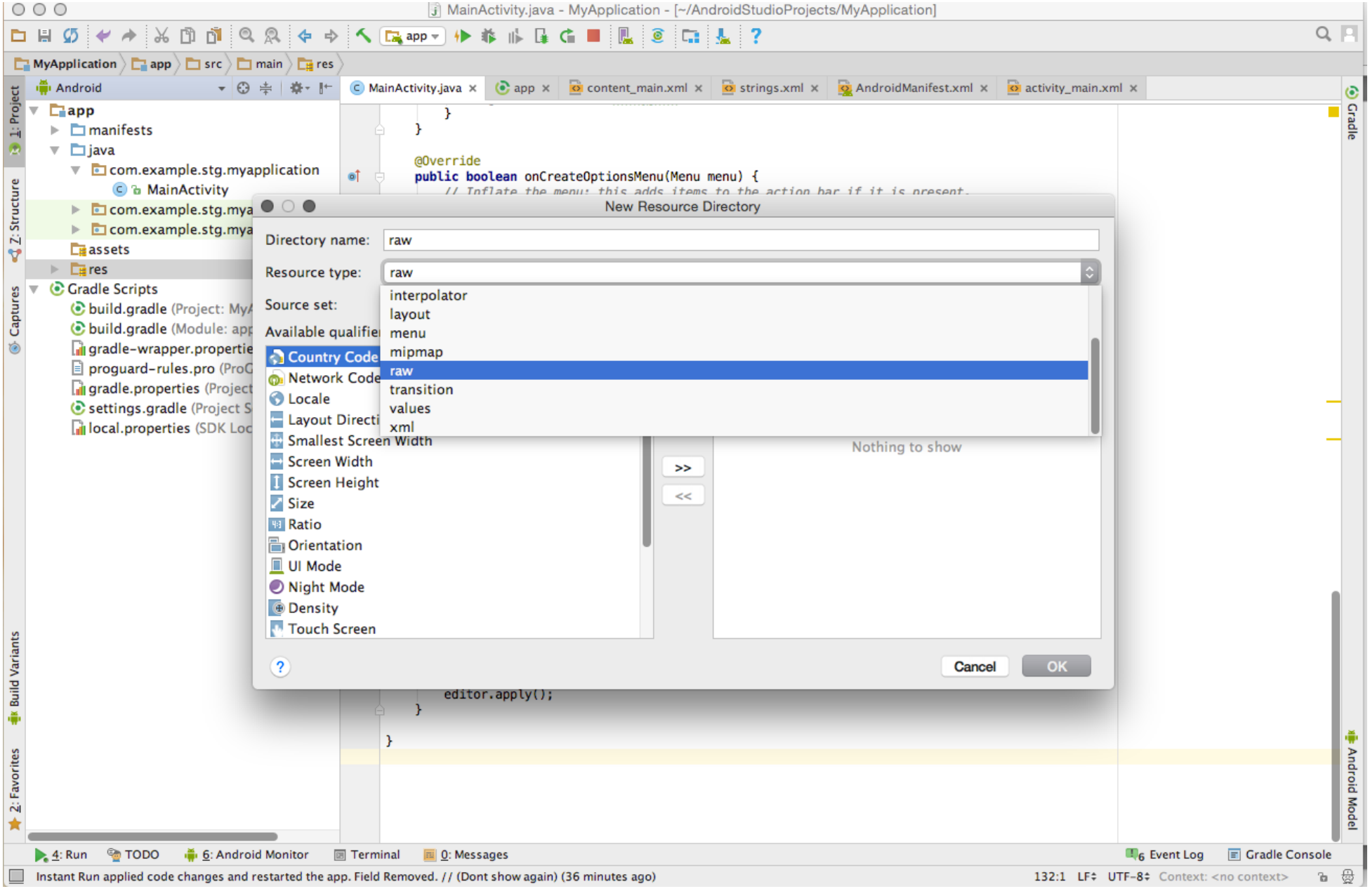
Using the device's internal storage

- Not everything is a key-value pair dictionary, so sometimes you might need to access a general file from your Android application.
- Files can be bundled with Android applications, and made private. When the user uninstalls the application, these files are removed.
- Static, read-only files are saved in the project `res/raw/` directory.

`developer.android.com/guide/topics/data/data-storage.html`

Creating the raw directory

In Android Studio (res > New > Android resource directory)



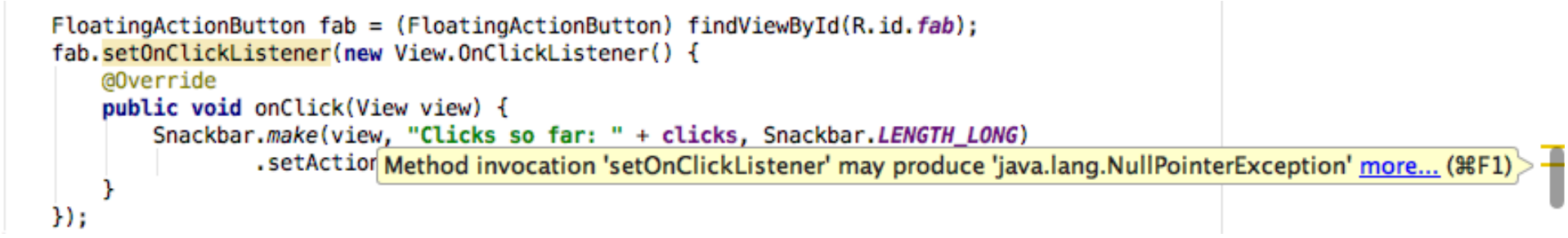
Reading `res/raw/myfile.txt`

```
1  try {
2      InputStream is = getResources().openRawResource(R.raw.myfile);
3      BufferedReader reader =
4          new BufferedReader(new InputStreamReader(is));
5
6      String line = reader.readLine();
7      while (line != null) {
8          // Do something with "line"
9          line = reader.readLine();
10     }
11     reader.close();
12     is.close();
13 } catch (Resources.NotFoundException e) {
14     Log.e(TAG, "Could not find resource file ...");
15 } catch (IOException e) {
16     Log.e(TAG, "An I/O exception occurred ...");
17 }
```

Note: We don't need to declare `R.raw.myfile`. It is enough that `res/raw/myfile.txt` exists. The `R` class is auto-generated.

Code analysis

- As we would expect from a modern IDE, Android Studio finds potential bugs in our code by *static analysis* (checking the code without running it). This can show potential errors such as *null pointer exceptions*.



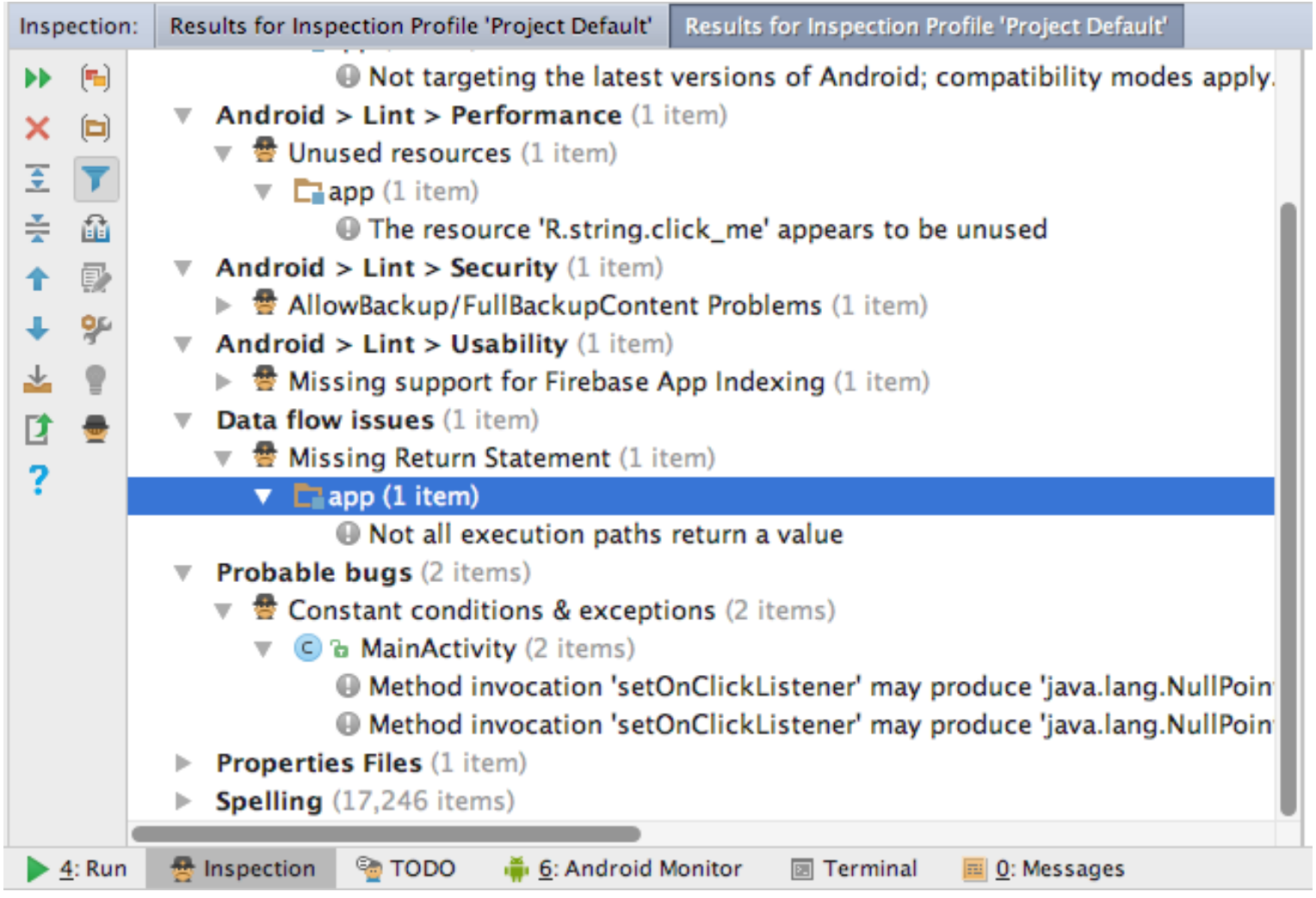
- This is very helpful for our Java coding, but an Android application also consists of XML layout files and resource files, property files and Gradle build files. *How do we find errors in the project as a whole?*

Whole-project analysis

- Android Studio provides whole-project analysis of Android applications (*Analyse* > *Inspect Code . . .*)
- This provides *lint*-like analysis of projects (not on every edit, as with a Java class, but only on-demand).
- Results are provided in an *Inspection* window which categorises problems in terms of:
  - Correctness
  - Performance
  - Security
  - Usability
  - Data flow issues
  - Probable bugs

  and others.

Results of code inspection

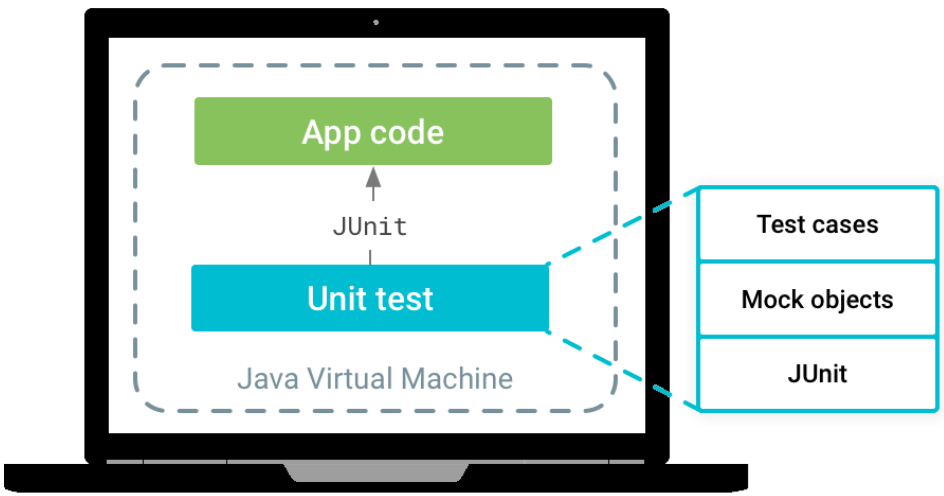


Android testing

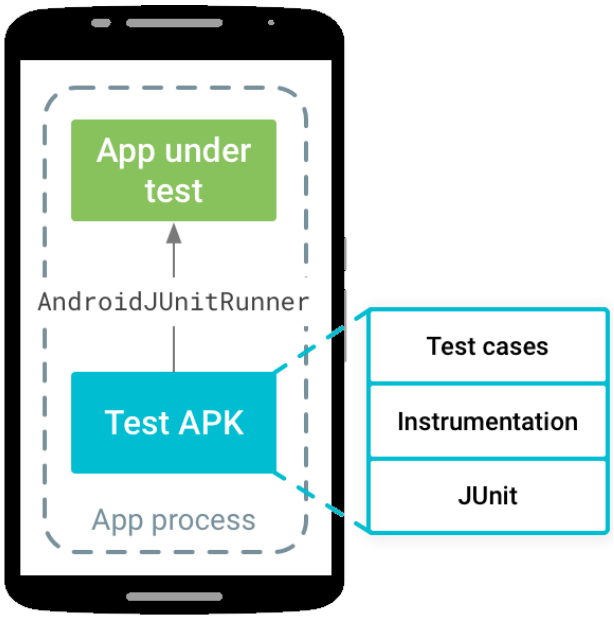
- Android supports two types of testing: *local unit tests* and *instrumented tests*.
- Local unit tests are located under `src/test/java`, run on the JVM, and do not have access to Android APIs.
- Instrumented tests are located under `src/androidTest/java`, run on a hardware device or the emulator, and can invoke methods and modify fields in your application.

`developer.android.com/training/testing/start/`

Unit tests and instrumented tests



Local unit test
`src/test/java/`



Instrumented test
`src/androidTest/java/`

`developer.android.com/training/testing/start/`

A simple instrumented test

We begin by importing classes and methods that we need.

```
1  package com.example.stg.myapplication;
2
3  import android.support.test.rule.ActivityTestRule;
4  import android.test.suitebuilder.annotation.LargeTest;
5
6  import org.junit.Rule;
7  import org.junit.Test;
8  import org.junit.runner.RunWith;
9  import android.support.test.runner.AndroidJUnit4;
10
11 import static android.support.test.espresso.Espresso.onView;
12 import static android.support.test.espresso.action.ViewActions.click;
13 import static android.support.test.espresso.matcher.ViewMatchers.withId;
```

A simple instrumented test

Tests are marked with annotations.

```
14  @RunWith(AndroidJUnit4.class)
15  @LargeTest
16  public class MainActivityInstrumentationTest {
17
18      @Rule
19      public ActivityTestRule mActivityRule = new ActivityTestRule<>(
20              MainActivity.class);
21
22      @Test
23      public void performThreeClicks_checkFab(){
24          onView(withId(R.id.button)).perform(click());
25          onView(withId(R.id.button)).perform(click());
26          onView(withId(R.id.button)).perform(click());
27
28          onView(withId(R.id.fab)).perform(click());
29      }
30  }
```

Running the test

- When we run the test, the emulator launches and the application runs without user interaction.
- At the end of a test run we are informed if all tests passed.
- Note that because this was an instrumented test, it must be stored under `src/androidTest/java`.

Links

- developer.android.com/studio/test/
- developer.android.com/training/testing/start/
- developer.android.com/training/testing/unit-testing/
- google.github.io/
  android-testing-support-library/
- google.github.io/
  android-testing-support-library/
  docs/espresso/