

# Mobile and Ubiquitous Computing

## Labsheet 7 – Storage and Menus

In this sheet, we're going to begin by focusing on how you can create Android applications that are able to store information. Being able to use storage is vital to any useful application – if our applications forget everything each time they are closed, then they will almost never have any practical utility. We're only going to look at the simplest way of storing data in this labsheet, *shared preferences*. There are other ways of storing data, like Firebase, Google's cloud-based storage engine or Room, the Android persistence library but they are out of the scope of this labsheet, and unfortunately, the scope of the module.

### Storing with Shared Preferences

#### *Goals:*

1. To be able to create a simple data-input application for collecting data.
2. To be able to store this data using Android Shared Preferences.

Create a new project, this time called 'BritishLibrary'. Create an empty activity called MainActivity. Edit your layout so that you have a `TextView` containing 'Borrower ID:' as its text (and ID `borrowLabel`), an `EditText` with the ID `borrowerID` and a `Button` with the ID `setBorrowerID`. Set the text for these widgets in your `strings.xml` file (think back to Labsheet 2 if you're not sure). We're doing this so that we have a data-entry tool that can produce data that we'll store.

I'm not going to show you how to lay this out. Your code should pass this test (NB– Don't forget your activity `@Rule`, see last week for an example):

```
@Test
public void checkLayout() {
    onView(withId(R.id.borrowLabel)).check(matches(isDisplayed()));
    onView(withId(R.id.borrowerID)).check(matches(isDisplayed()));
    onView(withId(R.id.setBorrowerID)).check(matches(isDisplayed()));
    onView(withId(R.id.borrowLabel)).check(isCompletelyAbove(withId(R.id.borrowerID)));
    onView(withId(R.id.borrowLabel)).check(isCompletelyAbove(withId(R.id.setBorrowerID)));
    onView(withId(R.id.borrowerID)).check(isCompletelyLeftOf(withId(R.id.setBorrowerID)));
}
```

#### *Storing our Borrower ID as a shared preference:*

We're first going to explore the use of shared preferences. Shared preferences in Android are a quick and easy way to save *primitives* using key-values pairs. Despite the name, their use is not

limited to storing preferences (to set preferences for an application formally, there is a Preference API that's built using shared preferences). We can store whatever we like, but, as I said, we're limited to storing *primitive* types (e.g., strings, ints, booleans).

Create a listener, `setBorrowerIDClick` for the button that you've created so that it does something. When this button is pressed, we're going to store whatever text is in our `borrowerID` field as a shared preference. First of all, we're going to need to adjust our `MainActivity` class and our `onCreate()` method to create a new instance through which we can access shared preferences:

```
private SharedPreferences sharedPref; //In the class definition
sharedPref = this.getSharedPreferences(Context.MODE_PRIVATE); //In the onCreate method
```

Now we're going to use our button listener to get the current contents of the `borrowerID` `EditText`:

```
public void setBorrowerIDClick(View view) {
    EditText bID = (EditText) findViewById(R.id.borrowerID);
    SharedPreferences.Editor editor = sharedPref.edit();
    editor.putString("borrower_id", bID.getText().toString());
    editor.commit();
}
```

You can see here that we're storing the value of `borrowerID` against the key `borrower_id`.

We want to load the contents of that back up when we reload the application, so we need to add some code to our `onCreate()` function to pull the data out of the shared preference storage and put it back into our `EditText`:

```
((EditText) findViewById(R.id.borrowerID)).setText(sharedPref.getString("borrower_id", "No ID set."));
```

Note that we provide two parameters, one is the key we want to retrieve, the other is a default value. Note the casting required on `findViewById()`.

Now, open the app, type in some text and click *Set*. Then, quit the app and reload it. It should remember whatever you last set it to. And that's all you need to do for storing very simple key-value data. In some situations you might find that the easiest way to save information is simply to serialize it (libraries like GSON will manage this process) and stuff it into a shared preference as a string. Then deserialize it on the other side. There are potentially performance issues with taking this approach for more complex data structures though, and it becomes necessary to move onto more complex models for persistence like *Room*.

## *Testing this*

Well this is an interesting challenge now, because we're going to need to start an activity, have the test perform some action, close the activity and then start it back up again to check that the shared preferences have worked correctly! Of course, they have thought about it. Make sure your code passes this test:

```

@Test
public void checkSharedPref(){
    final String TEST_ID = "a-very-test-id";
    onView(withId(R.id.borrowerID)).perform(clearText());
    onView(withId(R.id.borrowerID)).perform(typeText(TEST_ID));
    onView(withId(R.id.setBorrowerID)).perform(click());
    act.getScenario().close();
    ActivityScenario.launch(MainActivity.class);
    onView(withId(R.id.borrowerID)).check(matches(withText(TEST_ID)));
}

```

There are a few new things in here for you. First of all this is the first appearance we've seen of `clearText()` as an action to be performed. This does exactly what you'd expect. It clears the text that's currently in the `borrowerID` instance of `EditText`. Once we've emptied it, we can type our test string in and then `click()` the `setBorrowerID` instance of `Button`. This will cause shared preferences to be updated. Once we've done this we need to test if destroying and recreating the activity will correctly reload the data from shared preferences. To do this we call `close()` on the running activity scenario. Once `close()` has been called the activity scenario has a 'dead' state and can't be used again. So we need to launch a new activity using the `ActivityScenario.launch()` static method. We can then check that the text has been correctly reloaded. Note that `ActivityScenario` has a `recreate()` method. You cannot use it for testing what we want here though, because this method takes care of bundling and restoring state; this means that our text would be restored from the bundle and not from shared preferences.

## Making use of menus

In the labsheets so far, we've focused on creating activities and then adding widgets like `TextView` and `EditView` to those activities. We've also taken a look at a couple of different layout models like `ConstraintLayout` and `LinearLayout`. That's all well and good if you are developing an applications where everything takes place in a single activity. Although we've seen how to programmatically link activities using intents, we have not yet considered the kinds of interface elements that can be use to navigate multiple activities to form something that looks like the kinds of apps you are used to using. We're going to look at some of the options for adding some structure to your applications.

### *Goals:*

1. Use the `CoordinatorLayout` and `AppBarLayout` to create an application that has a menu bar.
2. Calling other activities from the menu bar.

### *Building an AppBar*

We're going to build on our British Library activity. There is a template available that would allow us to create a new project using the `CoordinatorLayout` and the `AppBarLayout`, but we're going to do it 'by hand' so that we get a better understanding of the details of what is happening.

Google call the `CoordinatorLayout` a 'supercharged' `FrameLayout`. A `FrameLayout` is a simple layout model that often contains a single view that expands to fill the 'frame'. The `CoordinatorLayout` is designed specifically for organising the top-level decoration of an app. Things like menu bars. We need to import the library for the constraint layout you can either add this to your app's build.gradle file:

**implementation "androidx.coordinatorlayout:coordinatorlayout:1.1.0"**

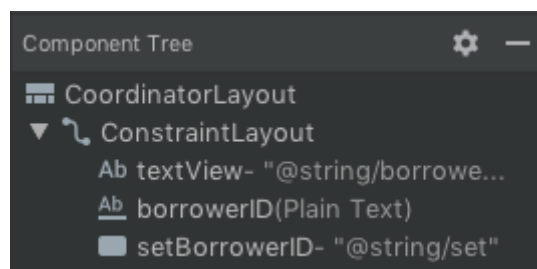
Or import it using the GUI. Once you've done this, we're going to be adjusting our `activity_main.xml` file in the text view rather than the design view. At the moment our top level tag is the `ConstraintLayout`. Between the XML definition on the first line and the opening tag of the `ConstraintLayout`, we're going to add this definition:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
```

At the very end of the XML document, close this new tag with:

```
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Now your `ConstraintLayout` is a sub-element of the `CoordinatorLayout`. If you switch back to design view, you will see the component tree in Android Studio has been updated accordingly:



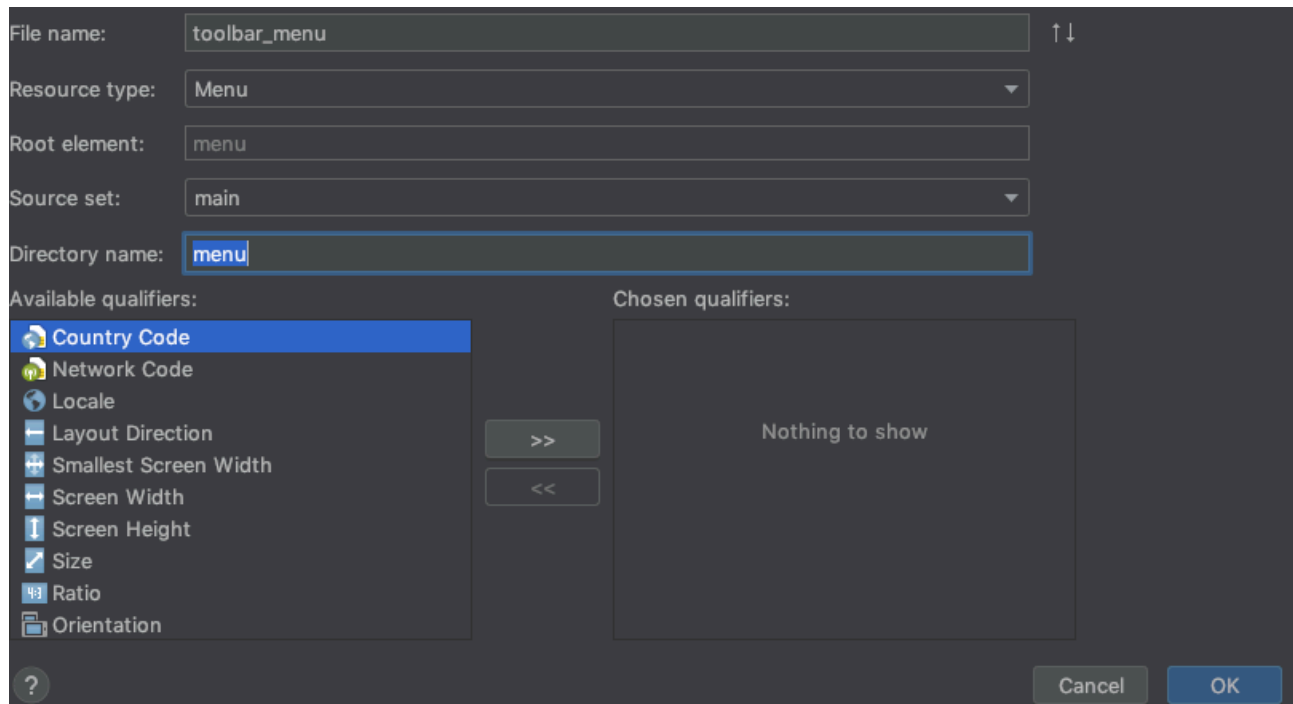
Now compile and run your app. You might be a little disappointed to find that, so far, absolutely nothing has changed. Do not worry. Now we have the `CoordinatorLayout` set-up we can start making some real progress.

## Adding a menu

We're going to add a menu to our action bar. This is going to contain a couple of options that will open other activities for us using intents. We've used `CoordinatorLayout` because it's what we'd use for creating complex toolbars. But in this case we're just going to create a simple text-

based menu. We get this for ‘free’ from the window that our app runs in, so we just need to define our menu and give it some options.

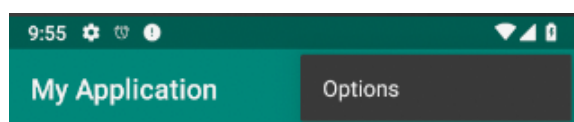
The first thing we need to do is to define the menu that’s going on our action bar. Like with most things in Android, we’re going to do this declaratively in an XML file. In your Android project tree view, right click the ‘res’ folder that contains your resources. Select New... → Android Resource File. A wizard dialog will now appear. Complete it as so:



You now have another XML file that presents a design view. From the component palette, drag a ‘Menu Item’ onto your component tree. In either design or text view, set the ID of this item to options. Set the text of the item to **Options** too. Before this option can do anything, we need some way of linking-up the menu we’ve created with the rest of our app. We do this by adjusting our MainActivity.java class.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.toolbar_menu, menu);
    return true;
}
```

Here we’re overriding `onCreateOptionsMenu()`, which is a method of `Activity`. This is called by Android the first time an activity is run so that a menu can be built. Note that a `menu` object is passed in. The `getMenuInflater().inflate()` method takes this as a parameter along with the menu we specified in our resource file. Run your app and you should see that a menu button has now appeared in the top-right of your app. If you tap it, your menu will appear:



Note that the behaviour of this menu depends on how the `showAsAction` attribute is set for the menu item. There are a few other options per the Android API:

Value	Description
<code>ifRoom</code>	Only place this item in the app bar if there is room for it. If there is not room for all the items marked " <code>ifRoom</code> ", the items with the lowest <code>orderInCategory</code> values are displayed as actions, and the remaining items are displayed in the overflow menu.
<code>withText</code>	Also include the title text (defined by <code>android:title</code> ) with the action item. You can include this value along with one of the others as a flag set, by separating them with a pipe <code> </code> .
<code>never</code>	Never place this item in the app bar. Instead, list the item in the app bar's overflow menu.
<code>always</code>	Always place this item in the app bar. Avoid using this unless it's critical that the item always appear in the action bar. Setting multiple items to always appear as action items can result in them overlapping with other UI in the app bar.
<code>collapseActionView</code>	The action view associated with this action item (as declared by <code>android:actionLayout</code> or <code>android:actionViewClass</code> ) is collapsible. Introduced in API Level 14.

Have a go at setting different values for `showAsAction`. What effect does `ifRoom` have?

Now we have successfully created our menu, it would be good if we could get the menu to do something, wouldn't it? To do this, we're going to override another method in `MainActivity.java`. So get that opened back up. The method we are overriding is

`onOptionsItemSelected()`. It will look like this:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.options:
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

Note that the clicked `MenuItem` is supplied as a parameter to this method. By using the `MenuItem.getItemId()` instance method we can match it up with the IDs of the item resources we defined in `toolbar_menu.xml`. Let's have it so that when 'Options' is selected a new `Activity` containing a list of options is opened. To do this you will need to:

- Create a new Empty Activity . (I called mine `OptionsActivity`)
- Add a `TextView` to it with some text so that you can see when it appears.
- Change the case for the `options MenuItem` in the switch so that it fires an `Intent` that opens this activity.

Once you do this, you will find there's a small problem, though. When the new activity is run, it doesn't know anything about the menu we created (the bar is there, but look: no three-dot menu). If we wanted the same menu we'd need to re-implement that inflater code that we just did for our MainActivity. We don't actually want to do this though: the actions on the AppBar should be a list of options contextually relevant to the activity we're on (e.g., share, move, whatever), but it shouldn't act as the primary means of navigating our application. There are better UI elements for doing this and better ways of moving between activities in an app, like bottom navigation and fragments. We probably won't have time to come on to these, but it's still important that you know how this works.

### *Quick test to finish*

```
@Test
public void menuInteraction(){

    openActionBarOverflowOrOptionsMenu(InstrumentationRegistry.getInstrumentation().
    getContext());
    onView(withText("Options")).perform(click());
    onView(withId(R.id.optionsLabel)).check(matches(isDisplayed()));
}
```

A couple of new things here. First, to open the app bar, and knowing that it might be in an Overflow state and so not immediately visible, we can use the special `openActionBarOverflowOrOptionsMenu` method (very specific!). Once it's open then we can look for an item that has the text "Options", and click on it. I have tried doing this with a direct reference to `R.id.options`, which is the resource of that actual menu item, but this doesn't work; I think for some reason it does not actually appear in the view hierarchy as the same way as other View elements.

Once we've located that menu item we click it and check to see if the `optionsLabel` instance of `TextView` is visible (as it should be, if the activity has been changed successfully).