Mobile Application Development - Part 4

Menus and Multiple Activities

In this section we will:

- Look at menus
- Look at how to create a multiple-activity application

Menus and Multiple Activities

So far, our apps have contained just one ***Activity***, or screen to interact with the user. However, a typical app will contain ***several*** activities, each of which gathers different data from the user. The next section will cover how to create a second Activity, and how to pass data between activities. The second Activity is going to be a screen containing two buttons, to allow the user to choose the style of map to show on our map app (regular map, or hikebikemap view).

Clone your project from GitHub

Clone your map project from last week, using the Git Cheat Sheet - you will add to it this week.

The goal

We will be allowing the user to choose between the regular map and the "Hike Bike Map" - a different style oriented towards walkers and cyclists. To see the latter effectively, ensure your default map location is:

- Latitude 51.05
- Longitude -0.72
- Zoom level 16

Menus

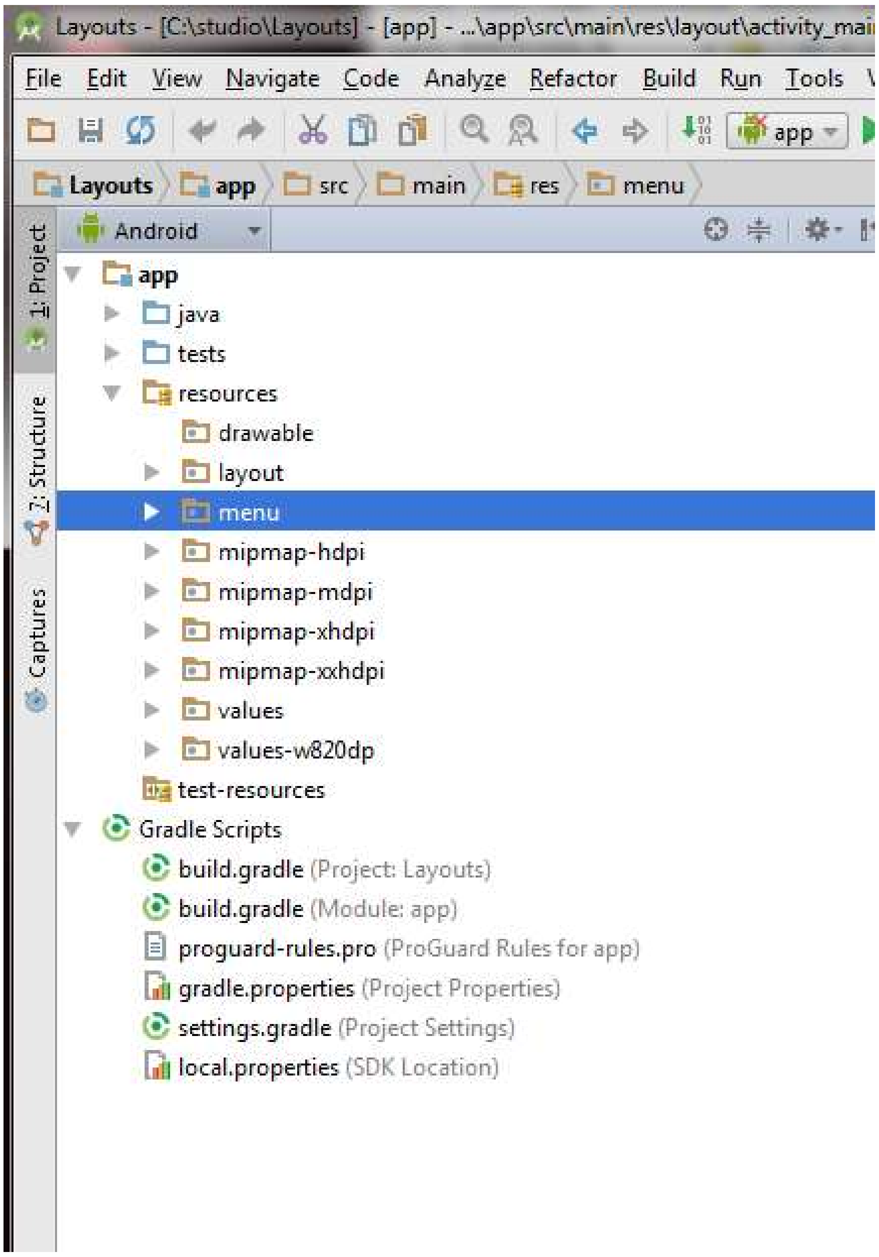
Before we create the second Activity, we are going to add a ***menu*** to our map app from the previous section. Most Android apps contain a menu of some sort. There are various methods for implementinç a menu in Android. Menus are activated via the ***action bar*** - the bar at the top of the app, often in a custom colour, which contains the app's actions.

There are two common styles of menu:

- First, the classic "three dots" style of menu on the right-hand side of the action bar. The menu appears when you press on the three dots.
- The "hamburger menu" (three horizontal lines) on the left of the action bar which brings up a slide-out menu (or ***navigation drawer***) on the left of the screen.

The "three dots" style menu is the classic style, used extensively in Android 4 apps and still used by some apps today. We will be examining this style of menu in this unit; the navigation drawer is used b
many contemporary apps but is more complex to implement so we will leave it until next year.

To create your menu, open the ***resources*** folder of your app and then the ***menu*** folder, as shown below:


Layouts - [C:\studio\Layouts] - [app] - ...\app\src\main\res\layout\activity_mai
File Edit View Navigate Code Analyze Refactor Build Run Tools
app
Layouts app src main res menu
Android
1: Project
2: Structure
Captures
app
java
tests
resources
drawable
layout
menu
mipmap-hdpi
mipmap-mdpi
mipmap-xhdpi
mipmap-xxhdpi
values
values-w820dp
test-resources
Gradle Scripts
build.gradle (Project: Layouts)
build.gradle (Module: app)
proguard-rules.pro (ProGuard Rules for app)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)

There should be an auto-created menu file in there already, added when you setup your project. It is likely to have a name such as ***menu.xml***. Delete all the code in the file and replace it with this:

```
<menu
  xmlns:android="http://schemas.android.com/apk/res/android">
<item android:id="@+id/choosemap" android:title="@string/choosemap"/>
</menu>
```

Note that there is a <menu> tag to define the menu as a whole, while each item is defined as an <item> tag. The item has an ID of ***choosemap*** and a title equal to the string with the name of ***choosemap*** (which should be defined in the strings.xml file).

Making the menu appear

The code above defines a menu in XML, however it does not actually make the menu appear. In our Java code we have to explicitly ***inflate the menu***. To do this you add an ***onCreateOptionsMenu()*** method to your main activity.

```
public boolean onCreateOptionsMenu(Menu menu)
{
    MenuInflater inflater=getMenuInflater();
    inflater.inflate(R.menu.menu, menu);
    return true;
}
```

and you have to add the following new import lines:

```
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
```

The code above will run automatically when the activity is created; it basically loads in the menu from the XML (***R.menu.menu*** is referencing the ***menu.xml*** file) and "inflates" it so that it can be seen in the GUI. Note that the example assumes that your menu is saved in the file ***menu.xml***. If your menu is saved in another file, you need to change the reference to ***R.menu.menu*** appropriately.

If you add this code to your map app and run it, you should find that a menu with one item appears when you press the Menu button.

Reacting to a menu item selection

Next thing we want to do is to launch a second Activity when the user selects the menu item. First thing to do is to write code to respond to the user selecting a menu item. To do this, you override ***onOptionsItemSelected()*** in Activity. Here is an example:

```
public boolean onOptionsItemSelected(MenuItem item)
{
    if(item.getItemId() == R.id.choosemap)
    {
        // react to the menu item being selected...
        return true;
    }
    return false;
}
```

The ***onOptionsItemSelected()*** method takes a MenuItem as a parameter, which is the menu item which was selected. We can find out which MenuItem was selected by using the ***getItemId()*** method of the MenuItem. This will return the item's ID as defined in the menu.xml file. So in this example, if the menu item with the ID of ***choosemap*** was selected, we do something.

Launching a second Activity

To launch a second Activity we create an ***Intent***. An Intent basically represents an instruction to do something, such as launch a second Activity or even use the services of an entirely different app. Data can be added to the Intent; that is how we send data between activities.

Add the following code to the if statement above:

```
Intent intent = new Intent(this,MapChooseActivity.class);
```

```
startActivity(intent);
```

and add the following additional import:

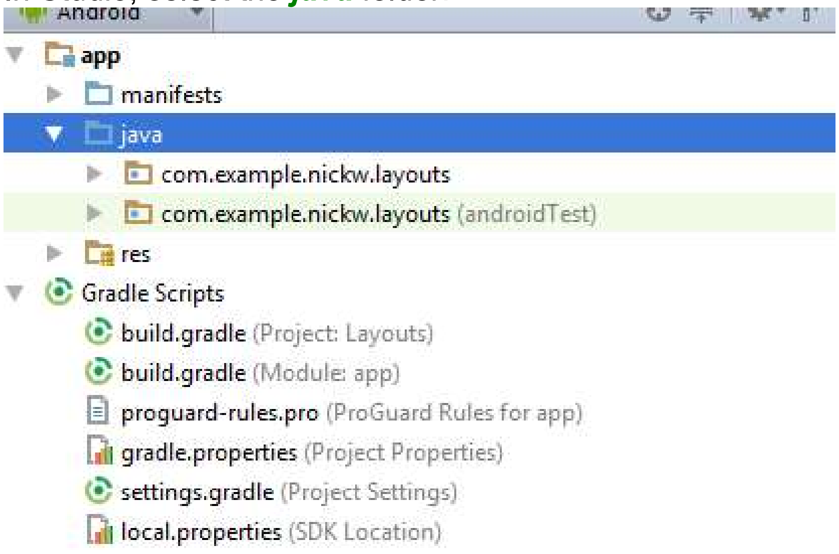
```
import android.content.Intent;
```

This code creates a new Intent. The two parameters are a reference to the current Activity (this) and the MapChooseActivity ***class*** (note: not an instance of it!). We haven't created this Activity yet but this is intended to be our second Activity. We then start up our activity with ***startActivity()***.

Creating the second Activity

We now need to create our second Activity. This will offer the user a choice to which map style to use either the regular style, or a public transport view showing bus routes, train routes, etc. This will be done using a pair of buttons. The appropriate map style will be set depending on which button the use pressed. (This is not the most friendly way, it would be better to use radio buttons or a list, but it will dc for now). Do this as follows:

- In Studio, select the ***java*** folder:


Android
app
manifests
java
com.example.nickw.layouts
com.example.nickw.layouts (androidTest)
res
Gradle Scripts
build.gradle (Project: Layouts)
build.gradle (Module: app)
proguard-rules.pro (ProGuard Rules for app)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)

  and then select ***File-New-Java Class***. Enter the name ***MapChooseActivity***.
- Add the following code to the new class:

```
    package com.example.map; // or whatever it is in your case

    import android.support.v7.app.AppCompatActivity;
    import android.os.Bundle;
    import android.content.Intent;
    import android.widget.Button;
    import android.view.View;

    public class MapChooseActivity extends AppCompatActivity
    {

        public void onCreate(Bundle savedInstanceState)
        {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_map_choose);
        }
    }
```

- Note the ***setContentView(R.layout.activity_map_choose)***. This sets the main content view of the activity to a layout defined in an (as yet uncreated) XML file called ***activity_map_choose.xml***. You are going to add this next.
- Now add this new layout file. Highlight the ***layout*** folder inside ***res*** (in a similar way to how you highlighted the ***menu*** folder when creating a menu) and select ***File-New-Layout resource file*** and enter the filename ***activity_map_choose.xml***. Add the following code to the activity_map_choose.xml file:

```
    <LinearLayout
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical"
      android:layout_width="match_parent"
      android:layout_height="match_parent">
    <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/regularview"
    android:id="@+id/btnRegular" />
    <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hikebikemapview"
    android:id="@+id/btnHikeBikeMap" />
    </LinearLayout>
```

- Add appropriate entries to your strings.xml file, i.e. "regularview" and "hikebikemapview" with appropriate text.
- Lastly (and this is a step which is often forgotten) you have to add the second activity to the manifest file. This step actually associates the activity with this app. Open your manifest file again, and, immediately after the ***</activity>*** tag of the main activity, add the code:

```
<activity android:name=".MapChooseActivity" android:label="@string/selectmap">
</activity>
```

(The label should be a reference into your strings.xml file. You'll need to add a "selectmap" entry in your strings.xml file).

Run your app now. You should find that if you select the menu option, the second activity with the three buttons appears. Note that you can then press the phone's Back button to return to your original activity.

Sending data back to the original activity

OK, we have a second activity created but we don't yet send the user's chosen option back to the original activity. How is this done? Basically we have to create another Intent in the second activity an send it back to the first.

You'll need to add some code to the onCreate() inside ***the second activity*** to handle the button press such as:

```
Button regular = (Button)findViewById(R.id.btnRegular);
regular.setOnClickListener(this);
Button hikebikemap = (Button)findViewById(R.id.btnHikeBikeMap);
hikebikemap.setOnClickListener(this);
```

and make the second activity implement ***View.OnClickListener***, as you did in week 2.

We now need to send a value back to the main activity. This can be a boolean representing whether t show the regular or HikeBikeMap view. As briefly mentioned above, we can do this by sending an Intent back to the main activity, containing this boolean value. As well as launching a given Activity, we can use Intents to ***pass information between activities***. Add an ***onClick()*** method to the second activity, with the following code:

```
public void onClick(View v)
{
    Intent intent = new Intent();
    Bundle bundle=new Bundle();
    boolean hikebikemap=false;
    if (v.getId()==R.id.btnHikeBikeMap)
    {
        hikebikemap=true;
    }
    bundle.putBoolean("com.example.hikebikemap",hikebikemap);
    intent.putExtras(bundle);
    setResult(RESULT_OK,intent);
    finish();
}
```

This code creates something called a ***Bundle***. A Bundle is basically a collection of data which can be passed around between Activities - think of it as a collection of key-value pairs. Here, the bundle contains one item - a boolean storing whether the HikeBikeMap is being used. We obtain this by working out which button was pressed (using the ***getId()*** method of the View passed into ***onClick***), and setting it to true if the hikebikemap button was pressed.

Note also how the entry in the bundle is labelled with the identifier ***com.example.hikebikemap***. Many entries can be placed in one bundle, so each needs to be identified uniquely. It is convention to start each entry with your domain name (such as ***com.example*** here), hence ***com.example.hikebikemap*** the ***hikebikemap*** bundle entry belonging to the ***com.example*** domain.

Having created the Bundle, we then add it to the intent and then call ***setResult*** to send a ***result*** back to the parent activity. You can think of a ***result*** as a little like a return code for a function. Here, we send back the result ***RESULT_OK*** to indicate to the calling activity (the app's main activity) that the secondary activity completed successfully. (You can also send back ***RESULT_CANCELED*** to indicate that the user cancelled the action in the second activity). Finally we call ***finish()*** to forcibly finish the activity.

Reading the data from the first activity

The final part of the equation is to read the data sent back from the second activity in the original activity. There are two steps to this:

1. First, when launching the second activity from the first, we have to specify that we are expecting a result to be sent back. To do this, change the line:

```
    startActivity(intent);
```

to

```
startActivityForResult(intent,0);
```

This launches the second activity and states that we are expecting a result to be sent back. The 0 is an ID that we use to determine which child activity produced the result (a parent activity could launch several child activities, so when we get a result, we need to identify which child activity produced the result).

2. This brings us straight on to the result-handling code, which runs in the first activity when we get a result from the second. This takes place in a method called ***onActivityResult()***. Add this method to your parent activity as follows:

```
    protected void onActivityResult(int requestCode,int resultCode,Intent inte
    {

        if(requestCode==0)
        {

            if (resultCode==RESULT_OK)
            {
                Bundle extras=intent.getExtras();
                boolean hikebikemap = extras.getBoolean("com.example.hikebikema
                if(hikebikemap==true)
                {
                    mv.setTileSource(TileSourceFactory.HIKEBIKEMAP);
                }
                else
                {
                    mv.setTileSource(TileSourceFactory.MAPNIK);
                }
            }
        }
   }
```

Note that you need the following additional import:

```
import org.osmdroid.tileprovider.tilesource.TileSourceFactory;
```

Note how the ***onActivityResult()*** is working. It takes three parameters, including the ***requestCode*** (the ID of 0 that we used to identify our activity launch, see above), the ***resultCode*** (the code that the second Activity sent back - RESULT_OK in our case to indicate all was OK) and the Intent used to send the result back to the first activity. So our ***onActivityResult()*** checks that this result was sent back from a request with an ID of 0 (see above) and a successful result was returned. If so, we retrieve the button ID sent back from the second Activity within the Bundle (see above) and turn hikebikemap mode on or off depending on what was sent back from the second activity.

Exercises

1. **Comment out the text fields which allow the user to set the latitude and longitude from the main activity that you did in Topic 3. You are now going to do it from a separate activity**, as described below.
2. Add a **third activity** to allow the user to enter a latitude and longitude and set the map to this location. To do this:
    - Add a further menu option to your app from above labelled "Set Location".
    - Create a third activity to allow the user to enter a latitude and longitude. Give it an XML layout file with two EditTexts, one for latitude and one for longitude, and a button. When th button is clicked, the latitude and longitude should be read from the EditTexts and sent back to the main activity in a Bundle in an Intent.
    - In your ***onOptionsItemSelected()*** method, launch the third activity, with a request code of 1, if the "Set Location" menu option is selected.
    - In your ***onActivityResult()*** method, set the latitude and longitude of the map to the contents of the Bundle from the Intent if the request code is 1.