Advanced Mobile Application Development
Week 14: Persistent Data

Persistent Data

There are multiple data persistence approaches on Android.

Files

Android can access many different types of files in the res/raw directory of your project.
Use the **Resources** class and its **getResources()** method to access the file.

Java File I/O

- In Java files are treated as streams of data, and the key pair of objects you will use are InputStream and OutputStream, for reading and writing data to/from files, respectively.
- These streams are provided by calling the openFileInput() or openFileOutput() methods from your context or activity.
- When a stream is available, your program logic is then responsible for actions such as reading in from the InputStream or writing out to the OutputStream
- You can present the file through an **InputStream** by calling the **openRawResources()** method to access the data from the file.
- You can use the **DocumentBuilder** class to parse the contents of the file which will provide you with a DOM representation of your data.

Pros

- Easy to package files with your app
- Resources easily available
- Simple static data in easy to read formats like xml

Cons

- By default the data is read-only. Editing resources or dynamically creating resources is complex
- Data is static

Internal storage(memory)

- Internal storage should be used for a larger amount on unstructured data private to your app

Pros

- Always available
- By default files saved are private to your app
- Other apps and the user can't access the files
- Files are removed when the user uninstalls your app
- Internal storage is best when you want to be sure that neither the user or other apps can access your files

Cons

- Hard to share data
- Internal storage might have limited capactiy

External storage(removeable)

- Similar to internal storage but the data files are world-readable without developer or user permission
- Only available when the storage is accessible

- When the user uninstalls the app the files are removed only if you saved them in the directory **getExternalFilesDir()**
- External storage is best for files that don't require access permissions or that you want to share with other apps or users
- Covered in chapter 17

SQL Database
- Android includes the APIs you need to use with their embedded SQLite database in the **android.database.sqlite** package
- A SQL database is a good choice for a large amount of structured data
- Covered in chapter 18

Shared Preferences (different than user preference)
- You can save a small amount of data as key-value sets using the **SharedPreferences** API
- You can use **getPreferences()** if you're only using one shared preference file in an activity as this uses the activity name as the preference set name
- Use **getSharedPreferences()** if you need multiple shared preferences files each with a unique set name
- Usually **MODE_PRIVATE** as the security visibility parameter so the preference is private and no other application can access it.
- Covered in the beginning of chapter 19

Writing to a shared preferences file
1. create a **SharedPreferences.Editor** by calling **edit()** on your SharedPreferences object
2. Add the key-value pairs using methods like **putInt(), putString(),** and **putStringSet()**
3. Call **commit()** to save the changes

Reading from a shared preferences file
1. create a **SharedPreferences.Editor** by calling **edit()** on your SharedPreferences object
2. Read in the key-value pairs using methods like **getInt(), getString(),** and **getStringSet()**

Network Connection
- You can use the network, when it's available, to store and retrieve data on your own web-based services
- To use network operations use the **java.net.*** and **android.net.*** classes
- We'll be looking at using Firebase next week

Superheroes
We're going to add data persistence to our superheroes app using shared preferences.
Update Hero.java

Change Hero[] to
```
public static final Hero[] heroes = {
        new Hero("DC", new ArrayList<String>()),
        new Hero("Marvel", new ArrayList<String>())
};
```

Add methods to write to, and read from, shared preferences.

```
public void storeHeroes(Context context, long universeId){
    //get access to a shared preferences file
    SharedPreferences sharedPrefs = context.getSharedPreferences("Superheroes",
Context.MODE_PRIVATE);
    //create an editor to write to the shared preferences file
    SharedPreferences.Editor editor = sharedPrefs.edit();
    //create a set
    Set<String> set = new HashSet<String>();
    //add heroes to the set
    set.addAll(heroes[(int) universeId].getSuperheroes());
    //pass the key/value pair to the shared preference file
    editor.putStringSet(heroes[(int) universeId].getUniverse(), set);
    //save changes
    editor.commit();
}

public void loadHeroes(Context context, int universeId){
    //get access to a shared preferences file
    SharedPreferences sharedPrefs = context.getSharedPreferences("Superheroes",
Context.MODE_PRIVATE);
    //create an editor to read from the shared preferences file
    SharedPreferences.Editor editor = sharedPrefs.edit();
    //create a set with the hero list
    Set<String> set =sharedPrefs.getStringSet(heroes[universeId].getUniverse(), null);
    //if there was a saved list add it to the heroes array
    if (set != null){
        Hero.heroes[universeId].superheroes.addAll(set);
    }
    //if no hero list was saved, use the defaults
    else {
        switch (universeId) {
            case 0:
                Hero.heroes[0].superheroes.addAll(Arrays.asList("Superman", "Batman", "Wonder
Woman", "The Flash", "Green Arrow", "Catwoman"));
                break;
            case 1:
                Hero.heroes[1].superheroes.addAll(Arrays.asList("Iron Man", "Black Widow", "Captain
America", "Jean Grey", "Thor", "Hulk"));
                break;
            default:
                break;
        }
    }
}

import android.content.SharedPreferences;
import android.content.Context;
import java.util.HashSet;
import java.util.Set;
import java.util.Arrays;
```

Update HeroDetailFragment.java

Update onCreateView()

```
//if the hero list is empty, load heroes
if (Hero.heroes[0].getSuperheroes().size() == 0 ) {
    Hero.heroes[0].loadHeroes(getActivity(), 0);
}
if (Hero.heroes[1].getSuperheroes().size() == 0 ) {
    Hero.heroes[1].loadHeroes(getActivity(), 1);
}
```

We need to save our hero list whenever we add or delete a hero.
Update addhero() and onContextItemSelected() after you call notifyDataSetChanged()

```
Hero.heroes[(int) universeId].storeHeroes(getActivity(), universeId);
```

To test, exit out of the app and restart it. Note that the list order might be different.
(DC: Black Canary; Marvel: Jessica Jones)

<u>Phone and tablet layouts</u>
One of the reasons we're using fragments is so our app can look different on a tablet than on a phone. On a tablet we want to see both fragments, as we are now. On a phone since the screen is smaller we want to only see one at a time. Just like with other resources you can create multiple layout folders with names to target specific specifications.
Switch to the Project view hierarchy and go to app/src/main/res and create a new resource directory.
Directory name: layout-sw400dp
Resource type: layout
Source set: main
This will target devices with the smallest available width of 400dp.
You can also use layout-w500dp(available width) or layout-land for landscape phone and tablet but not portrait phone or tablet.
(starting with Android 3.2, API 13, layout folders no longer use screen size values like –large)
http://developer.android.com/guide/practices/screens_support.html#DeclaringTabletLayouts
http://labs.rampinteractive.co.uk/android_dp_px_calculator/

Select your activity_main.xml file in your layout folder and copy/paste it into your new layout-sw400dp folder. This layout with side by side fragments will be used for devices with the smallest available width of 400dp.

Now we need to create a new Activity called DetailActivity that will use HeroDetailFragment. So instead of using both fragments in MainActivity, MainActivity will use UniverseListFragment and DetailActivity will use HeroDetailFragment when the user clicks on a universe.

Use the wizard to create a new empty activity called DetailActivity with a layout file named activity_detail. Make sure activity_detail.xml is in your layout folder so any device can use it.

Now let's update activity_main.xml in the layout folder to change the layout for smaller devices. Make sure you open activity_main.xml in the original layout folder.

Remove(cut) the FrameLayout so all you have is the one fragment for the universe list fragment. Change the layout_width for the one fragment to match_parent so it takes up the full width.

Paste it into activity_detail.xml as a fragment

```
<fragment
    android:id="@+id/fragment_container"
    class="com.example.aileen.superheroes.HeroDetailFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginEnd="8dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.47"
    tools:layout="@layout/fragment_hero_detail" />
```

If the app is running on a phone MainActivity will need to start DetailActivity with an intent and pass the universe id to the HeroDetailFragment using its setUniverse() method.

In DetailActivity.java add to onCreate()

```
//get reference to the hero detail fragment
    HeroDetailFragment heroDetailFragment = (HeroDetailFragment)
getFragmentManager().findFragmentById(R.id.fragment_container);
    //get the id passed in the intent
    int universeId = (int) getIntent().getExtras().get("id");
    //pass the universe id to the fragment
    heroDetailFragment.setUniverse(universeId);
```

Since DetailActivity.java will be running the HeroDetailFragment we need DetailActivity to implement the ButtonClickListener as well.

```
public class DetailActivity extends Activity implements HeroDetailFragment.ButtonClickListener

@Override public void addheroclicked(View view){
    HeroDetailFragment fragment =
(HeroDetailFragment)getFragmentManager().findFragmentById(R.id.fragment_container);
    fragment.addhero();
}
```

DetailActivity will get the id from the intent that starts it. We need MainActivity to start DetailActivity only when the app is running on a phone. So we want MainActivity to perform different actions if the device is using activity_main.xml in the layout or layout-sw400dp folder.
If the app is running on a phone it will be running activity_main.xml in the layout folder which doesn't include the HeroDetailFragment so we'd want MainActivity to start DetailActivity.
If the app is running on a tablet it will be running activity_main.xml in the layout-sw400dp folder which includes a frame layout with the id fragment_container so we'd need to display a new instance of HeroDetailFragment in the fragment container(as we have been doing).

Update MainActivity.java

Update itemClicked()

```
//get a reference to the frame layout that contains HeroDetailFragment
View fragmentContainer = findViewById(R.id.fragment_container);
//large layout device
if (fragmentContainer != null) {
** put the rest of the existing code in the body of the if statement
} else{ //app is running on a device with a smaller screen
   Intent intent = new Intent(this, DetailActivity.class);
   intent.putExtra("id", (int) id);
   startActivity(intent);
}
```

```
import android.view.View;
import android.content.Intent;
```

Define an AVD that is a tablet so you can test it on a phone and tablet. Chose category Tablet and pick one(I picked Nexus 7 API 23).

You should see both fragments side by side on the tablet and only 1 fragment at a time on a phone. You can probably modify the layout_width for the fragments in your layout-w500dp/activity_main layout file.
Make sure you've tested rotation as well and if you've clicked on an item other than the first when you rotate it stays on that data.

Don't forget launcher icons.