Advanced Mobile Application Development
Week 13: Data Persistence

Communicating with the User

Toasts

Toasts are used to provide the user with a little bit of information with no action required from the user.

SnackBars

A SnackBar is very similar to a toast but more customizable. It also allows for an action like undo. Snackbars animate up from the bottom of the screen and can be swiped away from the user. Otherwise they automatically disappear like Toasts.

Dialogs

https://developer.android.com/guide/topics/ui/dialogs.html
https://material.io/guidelines/components/dialogs.html
Dialogs present the user with a small amount of information(two sentences or less) and two actions. They are disruptive as the user must act on them in order to continue. A dialog takes over the screen and requires the user to take an action before they can proceed. Use a Dialog when the user's response is critical to your app flow as they are disruptive, otherwise use a SnackBar or Toast.
The Dialog class is the base class for dialogs, but should be subclasses and not used directly. Instead, use one of the following subclasses:

- AlertDialog: can show a title, up to three buttons, a list of selectable items, or a custom layout.
    - AlertDialog.Builder is used to create an alert dialog
    - Title (optional)
    - Content area
        - Message
        - Add selectable widgets, edittext, list
    - Three different action buttons
        - Positive to accept and continue with the action (the "OK" action).
        - Negative to cancel the action
        - Neutral for when the user may not want to proceed with the action, but doesn't necessarily want to cancel. It appears between the positive and negative buttons. For example, the action might be "Remind me later."
- DatePickerDialog or TimePickerDialog: allows the user to select a date or time.
- You can also create custom dialogs with custom layouts

Menus

We've seen menus used in the app bar and navigation drawer. Contextual menus can be used for actions specific to a portion of a view.
We'll use a contextual menu for deleting an item in the list.

RecyclerView Adapter Updates

The RecyclerView Adapter has methods that you need to call when there are changes to the data.

notifyItemInserted(int pos)	Notify that item reflected at position has been newly inserted.
notifyItemRemoved(int pos)	Notify that an item at a position has been removed from the data set.
notifyItemChanged(int pos)	Notify that item at position has changed.
notifyDataSetChanged()	Notify that the dataset has changed. Use only as last resort.

List
Create a new project called List
Basic Activity template
Language: Java
Minimum SDK: API 21 (21 is the minimum API for Material Design)

The basic template includes a little more than the empty template.
activity_main.xml sets the AppBarLayout which includes a Toolbar.
Includes the content_main xml layout which right now is just a textview.
Includes a floating action button.

MainActivity.java sets up the floating action button as well as an onClickListener for it.
A menu has also been added and set up.

List Layout
content _main.xml
Remove the textview
In design view drag out a RecyclerView to display the list(can be found in common or containers) below the image.
Add Project Dependency to add the support library with the RecyclerView. This will add it to your build.gradle(module: app) file.
Add start, end, top, and bottom constraints.
Make sure the RecyclerView has layout_width and layout_height set to “match_constraint” (0dp).
Make sure the RecyclerView has an id.

List Item Layout
We also need a layout file to determine what each row in our list will look like. Ours will be super simple with just a TextView.
File | New | Layout resource file
File name: list_item
Root element: android.support.constraint.ConstraintLayout
Source set: main
Directory name: layout

Drag out a TextView and make sure it has an id (textView)
Make sure the TextView has layout_width and layout_height set to “wrap_content”.
Add missing constraints.
Also change the constraint layout’s height to “wrap_content”. If the height is match_constraint each text view will have the height of a whole screen.

```
android:layout_height="wrap_content"
```

I also added some bottom padding so the text is in the vertical center of the row and made the text larger.

```
android:textAppearance="@android:style/TextAppearance.Material.Medium"
android:paddingBottom="10dp"
```

Once you don’t need the default text to help with layout, remove it.

Java class
We’re going to create a custom Java class for our data.
In the java folder select the main folder (not androidTest or test)

File | New | Java class
Name: Item
Kind: Class

We're going to create an Item class with just one data member to store the name, a getter method for the name, and a constructor that takes a name.
We'll also create an ArrayList of class Item to store our list. For now we'll put some test data in our ArrayListwe can make sure we get the recyclerview working.

```
public class Item {
   private String name;

   private Item(String newItem){
      name = newItem;
   }

   public static List<Item> myItems = new ArrayList<Item>(){{
      add(new Item("my test item"));
   }};

   public String getName(){
      return name;
   }
}
```

The static keyword means it belongs to the class instead of a specific instance. Only one instance of a static field exists even if you create a million instances of the class or you don't create any. It will be shared by all instances.

In Java the List interface can be implemented as an ArrayList or a LinkedList, we're implementing it as an ArrayList(usually better performing). Defining it as a List is more flexible than defining it as an ArrayList and is recommended.

Adapter

Now we need to create an adapter which will be used to populate the data into the RecyclerView. The adapter's role is to convert an object at a position into a list row item to be inserted.

File | New | Activity | Empty
ListAdapter
Uncheck Generate Layout File and Launcher Activity.
Backwards compatibility doesn't matter as we'll be changing the superclass.
Change the superclass to RecyclerView.Adapter<ListAdapter.ViewHolder>. This will give you errors until you implement the required methods.
Select the light bulb and choose implement methods to get rid of the error.
You'll still have an error that we need a ViewHolder class.

With a RecyclerView the adapter requires the existence of a ViewHolder object which describes and provides access to all the views within each item row.
So in our ListAdapter class we'll create a ViewHolder class to render the item and a default constructor.

```
public class ListAdapter extends RecyclerView.Adapter<ListAdapter.ViewHolder> {
    class ViewHolder extends RecyclerView.ViewHolder{
        TextView textView;

        public ViewHolder(@NonNull View itemView) {
            super(itemView);
            textView = itemView.findViewById(R.id.textView);
        }
    }
…
}
```

If the onCreate() method is giving you an error you can delete it, we won't need it.
Now all the errors should be gone.
Now that we have our adapter and viewholder defined, let's set up the adapter. We'll define a list of items and define a constructor.

```
private List<Item> mItems;

public ListAdapter(List<Item> items){
    mItems = items;
}
```

Implement the three required methods for the adapter.

```
    @NonNull
    @Override
public ListAdapter.ViewHolder onCreateViewHolder(@NonNull ViewGroup viewGroup, int i) {
    LayoutInflater layoutInflater = LayoutInflater.from(viewGroup.getContext());
    View itemView = layoutInflater.inflate(R.layout.list_item, viewGroup, false);
    ViewHolder viewHolder = new ViewHolder(itemView);
    return viewHolder;
}

@Override
public void onBindViewHolder(@NonNull ListAdapter.ViewHolder viewHolder, int i) {
    Item item = mItems.get(i);
    viewHolder.textView.setText(item.getName());
}

@Override
public int getItemCount() {
    return mItems.size();
}
```

Go into MainActivity.java.
With the adapter set up, we need to bind the data from the adapter to the data source to populate the RecyclerView and set a layout manager.

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        …

        //get the recycler view
        RecyclerView recyclerView = findViewById(R.id.recyclerView);

        //define an adapter
        ListAdapter listAdapter = new ListAdapter(Item.myItems);

        //divider line between rows
        recyclerView.addItemDecoration(new DividerItemDecoration(this,
LinearLayoutManager.VERTICAL));

        //assign the adapter to the recycle view
        recyclerView.setAdapter(listAdapter);

        //set a layout manager on the recycler view
        recyclerView.setLayoutManager(new LinearLayoutManager(this));
    }
```

You should be able to run it and see the test data in the recyclerview.

Add Items

Add a method to the ListAdapter class to add an item.

```
public void addItem(String newItem){
    Item.myItems.add(new Item(newItem));
    notifyItemInserted(getItemCount());
}
```

In Item.java remove the dummy data.

```
public static List<Item> myItems = new ArrayList<Item>();
```

In activity_main.xml change the fab to +

```
app:srcCompat="@android:drawable/ic_input_add"
```

In MainActivity.java update the onClick method for the floating action button.

```
FloatingActionButton fab = findViewById(R.id.fab);
fab.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(final View view) {
        //create alert dialog
        AlertDialog.Builder dialog = new AlertDialog.Builder(MainActivity.this);
        //create edit text
        final EditText edittext = new EditText(getApplicationContext());
        //add edit text to dialog
        dialog.setView(edittext);
        //set dialog title
        dialog.setTitle("Add Item");
        //sets OK action
        dialog.setPositiveButton("Add", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int whichButton) {
                //get item entered
                String newItem = edittext.getText().toString();
                if (!newItem.isEmpty()) {
                    // add item
                    listAdapter.addItem(newItem);
                }
                Snackbar.make(view, "Item added", Snackbar.LENGTH_LONG)
                        .setAction("Action", null).show();
            }
        });
        //sets cancel action
        dialog.setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int whichButton) {
                // cancel
            }
        });
        //present alert dialog
        dialog.show();
    }
});
```

The FAB should now let you add items in an alert dialog and then be added and visible in the view. The SnackBar's not really needed since you can see the item added to the view, I just left it there.

Delete Items

When the user does a long-press on an item we want them to be able to delete it. We'll do this though a context menu so they can chose to delete the item or not.
In ListAdapter.java add a method to delete an item.

```
public void removeItem(int itemPosition){
    Item.myItems.remove(itemPosition);
    notifyItemRemoved(itemPosition);
}
```

Then we'll update onBindViewHolder() to set up the long click listener and create the context menu. (Note I had to change viewHolder to be 'final' and also view in onCreateContextMenu below).

```
@Override
public void onBindViewHolder(@NonNull final ListAdapter.ViewHolder viewHolder, int i) {
    final Item item = mItems.get(i);
    viewHolder.textView.setText(item.getName());

    //long click listener
    viewHolder.itemView.setOnLongClickListener(new View.OnLongClickListener(){

        @Override
        public boolean onLongClick(View v) {
            return false;
        }
    });

    //context menu
    viewHolder.itemView.setOnCreateContextMenuListener(new
View.OnCreateContextMenuListener(){
        @Override
        public void onCreateContextMenu(ContextMenu menu, final View v,
ContextMenu.ContextMenuInfo menuInfo) {
            //set the menu title
            menu.setHeaderTitle("Delete " + item.getName());
            //add the choices to the menu
            menu.add(1, 1, 1, "Yes").setOnMenuItemClickListener(new
MenuItem.OnMenuItemClickListener() {
                @Override
                public boolean onMenuItemClick(MenuItem item) {
                    removeItem(viewHolder.getAdapterPosition());
                    Snackbar.make(v, "Item removed", Snackbar.LENGTH_LONG)
                            .setAction("Action", null).show();
                    return false;
                }
            });
            menu.add(2, 2, 2, "No");
        }
    });
}
```

Data Persistence
There are multiple data persistence approaches on Android. The approach you pick should be based on

- What kind of data you need to store
- How much space your data requires
- Whether the data should be private to your app or accessible to other apps and the user

Device Storage
All Android devices have two file storage areas: "internal" and "external" storage. These names come from the early days of Android, when most devices offered built-in non-volatile memory (internal storage), plus a removable storage medium such as a micro SD card (external storage). Many devices now divide the permanent storage space into separate "internal" and "external" partitions.

Internal file storage
Internal storage is best when you want to be sure that neither the user nor other apps can access your files.
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

Cons
- Hard to share data
- Internal storage might have limited capacity

For temporary storage you should use an internal cache file.

External file storage
External storage is best for files that don't require access permissions or that you want to share with other apps or users. It's called external because it's not a guaranteed to be accessible—it is a storage space that users can mount to a computer as an external storage device, and it might even be physically removable (such as an SD card). Most often, you should use external storage for user data that should be accessible to other apps and saved even if the user uninstalls your app, such as captured photos or downloaded files.
- Similar to internal storage but the data files are world-readable and can be modified by the user without developer or user permission when they enable USB mass storage to transfer files on a computer.
- Only available when the storage is accessible
- When the user uninstalls your app, the system removes your app's files from here only if you save them in the directory from getExternalFilesDir().

After you request storage permissions and verify that storage is available, you can save two different types of files:
- Public files: Files that should be freely available to other apps and to the user. When the user uninstalls your app, these files should remain available to the user.
- Private files: Files that rightfully belong to your app and will be deleted when the user uninstalls your app. Although these files are technically accessible by the user and other apps because they are on the external storage, they don't provide value to the user outside of your app.

Database
Android provides full support for SQLite databases. Any database you create is accessible only by your app.
- A SQL database is a good choice for a large amount of structured data
- Any database you create is accessible only by your app
- Covered in chapter 18
- Android also includes the Room library which provides an abstraction layer over SQLLite https://developer.android.com/training/data-storage/room/index.html
    - Highly recommended over SQLLite
    - Still requires SQL
    - Need to add the components to your gradle file

Shared Preferences (different than user preferences)
https://developer.android.com/training/data-storage/shared-preferences.html

SharedPreferences is a good choice if you don't need to store a lot of data and it doesn't require structure.

- Read and write key-value pairs of primitive data types: boolean, float, int, long, string, and stringset.
- Persistent across user sessions even if your app is killed
- 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()**

Except for some types of files on external storage, all these options are intended for app-private data—the data is not naturally accessible to other apps. If you want to share files with other apps, you should use the FileProvider API.
If you want to expose your app's data to other apps, you can use a ContentProvider. Content providers give you full control of what read/write access is available to other apps, regardless of the storage medium you've chosen for the data (though it's usually a database).

List (cont.)
We're going to add data persistence using shared preferences.
In Item.java add a constant for the name of your shared preferences file and methods to write to and read from a shared preferences file.

```
private static final String PREFS_NAME  = "ITEMS";

public static void storeData(Context context) {
   //get access to a shared preferences file
   SharedPreferences sharedPrefs = context.getSharedPreferences(PREFS_NAME,
Context.MODE_PRIVATE);
   //create an editor to write to the shared preferences file
   SharedPreferences.Editor editor = sharedPrefs.edit();
   //add size to the editor
   editor.putInt("size", myItems.size());
   //add items to the editor
   for (int i = 0; i < myItems.size(); i++) {
      editor.putString("item" + i, myItems.get(i).getName());
   }
   editor.apply();
}
```

```
public static void loadData(Context context) {
    //get access to a shared preferences file
    SharedPreferences sharedPrefs = context.getSharedPreferences(PREFS_NAME,
Context.MODE_PRIVATE);
    //create an editor to read from the shared preferences file
    SharedPreferences.Editor editor = sharedPrefs.edit();

    int size = sharedPrefs.getInt("size", 0);

    if(size>0) {
        for (int i = 0; i < size; i++) {
            myItems.add(new Item(sharedPrefs.getString("item" + i, "")));
        }
    }
}
```

We'll save the data when the app goes into the Paused state. Implement the onPause() lifecycle method in MainActivity.java.

```
@Override
protected void onPause() {
    super.onPause();
    //save data
    Item.storeData(this);
}
```

We'll read in the data in MainActivity in onCreate(). We'll only load the data if there is none in our ArrayList so we don't read it in every time the device is rotated and the activity is restarted.

```
//load data
if(Item.myItems.isEmpty()) {
    Item.loadData(this);
}
```

To test the data persistence make sure you don't just stop the emulator from AS, you must go into Paused state which you can do by using the right most button on the device, or just going to the home screen or another app.