1)Myfavourite library is **Butterknife**. It provides features like :

(i)**Bind views** to Activity,Fragments by using @Bind annotation.

(ii) Remove the unnecessary call to invoke **listeners** by creating anonymous classes in create methods by using annotations like @OnClick

(iii)**Group** similar views into list and for binding viewholder pattern as well

(iv**) Bind resources** like strings, drawable,intetc using @BindString,@BindDrawable,@BindInt for ease in access.

For example, I have a button named button1 and textView named text1 in my layout resource file fragment\_main. To use this button in my Fragment, I would have to search the view hierarchy for this view. To perform any action on click of this button, I would have to make an anonymous inner class as well.

There is a lot of boilerplate code written in this case. The same example when used with Butterknife will be

publicMyFragment extends Fragment {

@Bind(R.id.text1)

TextView text1;

//text1 is the reference of the Text used in my layout

@Override

public void onCreateView(LayoutInflaterinflater,ViewGroupviewGroup,BundlesavedInstanceState) {

............

//This statement will bind all the components in my Layout to their references like btn

ButterKnife.bind(this,view);

text1.setText("hi");

}

@OnClick(R.id.button1)

voidOnClick() { //No need to configure listeners

.....

}

}

Butterknife reduces not only the lines of code to be written but also simplifies **readability** and **reusability** of code.

2) Open map from app provided the location

Say the address be called as "address", city be called as "city", state be called as "state" and zip code be called as "zip".

To open the map app, I will use **implicit intent** to launch any map based app in the device to locate the address matching the location provided.

Steps involved will be

(i) Constructing the Uri :

The Uri will be the location which needs to be parsed.Spaces in the location can be concatenated using "+" symbol.

Uri uri= Uri.parse("geo:0,0:q=address,city,state,zip"); //pass the location query

(ii) Create the implicit intent by providing the category used for viewing the map and this uri

Intent locationIntent = new Intent(Intent.ACTION\_VIEW,uri);

(iii) Set package for this app, which will be google maps in this case which will handle this intent

locationIntent.setPackage("com.google.android.apps.maps):

(iv) Check if any app in the device could handle this intent, if so then launch the app using this intent.

if(locationIntent.resolveActivity(getPackageManager()!=null)

{

startActivity(locationIntent);

}

With intents we can use bundles, parcelable objects, data types and collection maps to send data as well.

3) **String compression**

publicstatic String compress(String input) {

//variable to count frequency of each letter

intfreq=0;

char alphabet='\u0000'; //initialize null character

//buffer to store compressed string with frequency

StringBufferresultStr=newStringBuffer();

// resultant compressed string

String compressedStr = null; if(input==null||input.isEmpty()) {

return "";

}

//if input string has length equal to 1

elseif(input.length()==1) {

return input;

}

for(inti=0;i<input.length();i+=freq) {

//check if last character has been reached

if(i==input.length()-1)

{

alphabet = input.charAt(i);

//if last character is different than previous character then frequency should be zero

if(input.charAt(i)!=input.charAt(i-1)) {

freq=0;

}

break;

}

else {

freq=0;

//loop for comparing adjacent characters

for(int j=i+1;j<input.length();j++) {

//increment count if characters are same

if(input.charAt(i)==input.charAt(j)) {

++freq;

}

else

{

//add to buffer the character along with its frequency

resultStr.append(input.charAt(i));

++freq;

resultStr.append(freq);

break;

}

}

}

}

//add to buffer the character along with its frequency for last character

resultStr.append(alphabet);

++freq;

resultStr.append(freq);

compressedStr = resultStr.toString();

if(compressedStr.length()>input.length()) //length comparison

{

compressedStr = input;

}

return compressedStr;

}

4) Four options to save data while making an app :

To save information like user specific information, user choices and data that needs to be presented to user while developing an android app, there are four options to persist data :

(i) **SQLite databases** - Content providers are provided to store information in a SQLite database which provides relational organization of data in the form of tables. For using content providers with Android app, we need to implement a Database contract(detail about the underlying table structure), a DB helper(for providing operations on DB) and Content provider(which interacts with the UI and DB helper)

(ii)**SharedPreferences** :SharedPreferences are used to store key/value pair of information.

(iii**)Internal storage** : We can save information in the internal storage of the device. The data saved here would be private to the app and would be deleted if the app is uninstalled. We can manipulate this information using normal File I/O operations.

(iv) **External storage** : We can also save information in the external storage or the SD card of the device. The information saved here would be public and would be accessible to external apps/programs and when the device is connected to a computer.

Also data can be saved on a **network** to be accessed through cloud to web based services.

SharedPreferences are used to store key/value pair of data which is in the form of primitive datat ypes like int,String,boolean. For example, in a weather app I need to save location of the user whenever user updates the location. I get an instance of SharedPreference and get refrerence of SharedPreference Editor and write the updated location for a key say "location" and commit this change. To read this location in my app, I get a reference of SharedPreferences and perform a read operation for this key "location" and update my weather app for weather related information for this location.

5) Algorithm to **find route** between two nodes

public class GraphNode {

boolean visited;

List<GraphNode<T>>getChildren() {

}

}

To check if route exists between two nodes, I have used Breadth first algorithm to maintain a queue of nodes connected to each node and check if the end node has been reached.

**Assumptions**

(i) There is a boolean field named visited in GraphNode which keeps track of whether a node has been visited or not.

(ii) Initially before calling findPath(), all the nodes have been marked as unvisited.

public static <T>booleanfindPath(GraphNode<T> start, GraphNode<T> end) {

boolean found = false;

//Queue used to maintain FIFO collection

Queue<GraphNode<T>>graphNodeDeque = new LinkedList<GraphNode<T>>();

if(start==null || end==null)

{

return false;

}

graphNodeDeque.add(start);

while (!graphNodeDeque.isEmpty()) {

GraphNode<T> node = graphNodeDeque.remove();

node.visited = true;

if (node == end) { //current node and end node are same

found = true;

break;

}

else

{

//Iterate each child node of the graph to check if it has been visited or not.

for (GraphNode<T> child : node.getChildren()) {

if (!child.visited) {

graphNodeDeque.add(node);

}

}

}

}

return found;

}

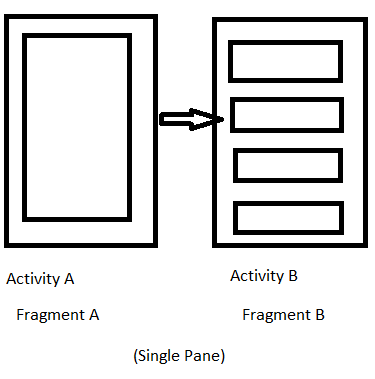
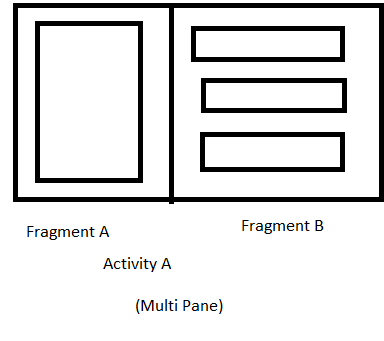
6) Views about fragments

Fragments are part of activity to perform some meaningful operation. There lifecycle is almost similar to the lifecycle of activities and their lifetime is coupled to the lifetime of its parent activity. We can have multiple fragments for an acitivity and one fragment for multiple activities as well.

I am in favour of using them as they provide various advantages :

1. They make it easy to support multiple kind of layouts like **master-detail layout** for tablets where we can have 2 fragments managed by a single activity one displaying the list, the other the detail.

In the example below in single pane layout we will a fragment each for each activity for showing the list and detail respectively. But in the case of tablets, we can use a single activity and attach two fragments to it, one for showing the list and the other for the detail.

(ii) Ease of use in **fragment transaction** and **backstack managment**. The fragment manager allows us to manipulate fragments by adding them to backstack so that its easy to restore the previous state of the app when back button is pressed.

(iii) Makes it easy to implement **responsive design** and transitions.We can add, remove fragments when the activity is running and when the activity is stopped/paused ,the corresponding fragments will be paused as well.

(iv) Ease in debugging as the logic of the app is handled in the fragment.

(v) Provides **modularity** and code reusability as its possible to use the same fragment for multiple activities.

(vi) We can use fragments without views to perform logical operations only without creating the layout resources.

So instead of managing the entire lifecycle of Activity for each interaction, we can use reusable fragments to attach it to Activity and perform user operations. Hence I am in favour of using fragments with activities.