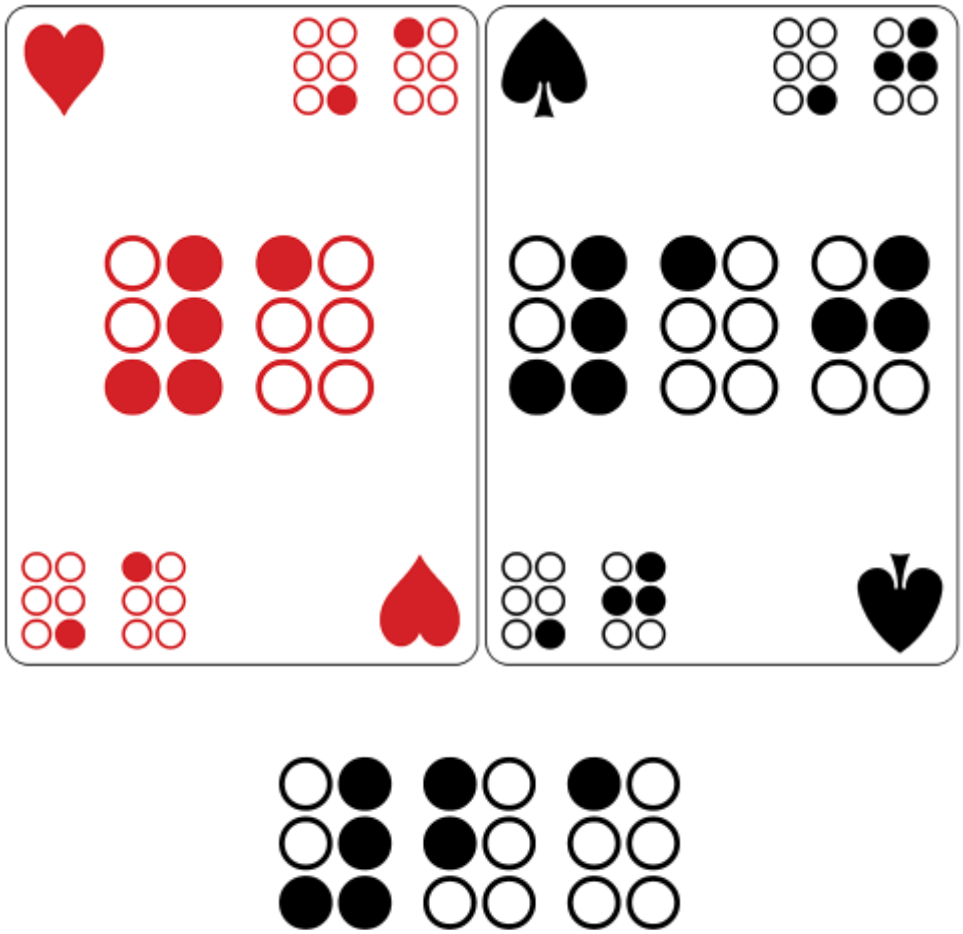


ECE 558 - FINAL PROJECT

UEB BLACKJACK

PSU WINTER 2015

By Hussein AlAmeer & Tu Truong

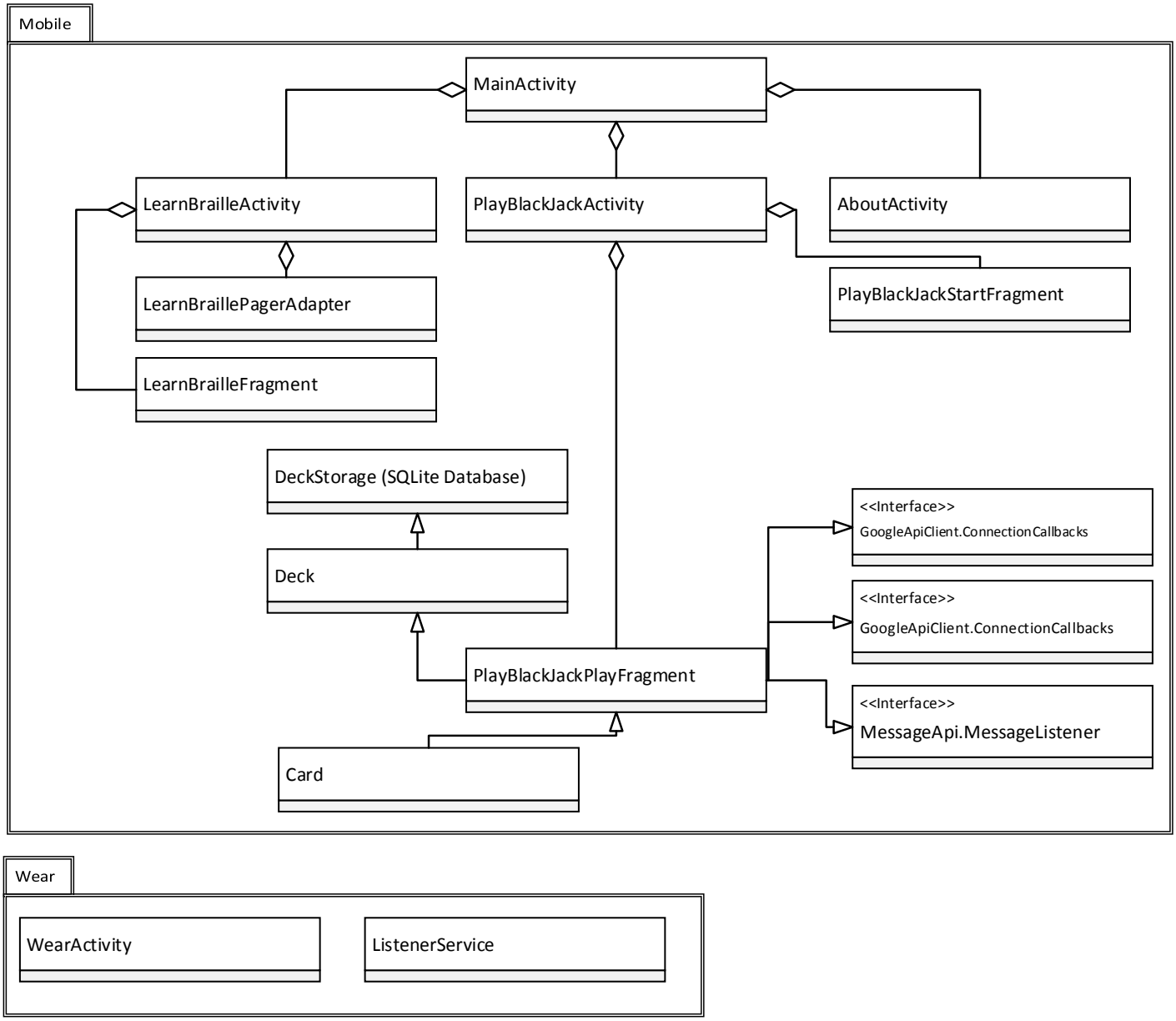
Table of Contents

Repository Address:_____ 1
Project Diagram _____ 2
Project Application Flow_____ 3
Application User Interface_____ 4
Blackjack Logic _____ 6
Storage _____ 7
Phone & Wear Communication_____ 8
Transition Animations _____10

Repository Address:

https://github.com/halameer/BrailleBlackjack.Manager.git

Project Diagram



Project Application Flow

This app contains 4 Activities and 3 Fragments.

1. Mobile: MainActivity Activity
    a. ListView that redirects the user to 3 other activities
        i. LearnBraille Activity ViewPager
            1. LearnBrailleFragment
                a. Swipes through note cards
                    i. UEB Numbers
                    ii. UEB Capital Letter & Capital Word
                    iii. UEB Alphabet A-N
                    iv. UEB Alphabet M-Z
        ii. Play BlackJack Activity
            1. PlayBlackJackStart Fragment
                a. Shows the game layout and a start button that switches to the PlayBlackJackGame Fragment.
            2. PlayBlackJackGame Fragment
                a. The Black Jack Game
                    i. Player is dealt 2 cards, Dealer is dealt Right card.
                    ii. If Player Has a BlackJack
                        1. Player Wins
                    iii. Player Can Hit
                        1. If Player gets 21
                            a. Player Stands
                        2. If Player gets < 21
                            a. Update Totals and Wait for Player's next choice
                        3. If Player gets > 21
                            a. Player Busts and Loses
                    iv. Player Can Stand
                        1. Deal the Dealer's Second Card
                        2. Continue Dealer's cards until…
                            a. Dealer Gets > 21
                            b. Dealer Total > Player Total
                            c. Dealer Gets total > 17
                    v. Player Can Start Over
                        1. Replace PlayBlackJackGame Fragment with itself
                    vi. Player Can Reveal Numbers
                        1. Display the current Dealer and Player Current Cards and Totals.
        iii. About Activity

1. Displays Credit to Hussein and Tu
2. Displays Credit to http://developer.android.com/
   And to TutorialsPoint.com for helping with Android Text To Speech
   http://www.tutorialspoint.com/android/android_text_to_speech.htm
   And to Alex Lockwood from AndroidDesignPatterns.com with Tranisition Animations
   http://www.androiddesignpatterns.com/2014/12/activity-fragment-transitions-in-android-lollipop-part1.html
   And to Maciej Ciemięga from StackOverflow.com and Curtis Martin from ToastDroid.com with Messages Api
   http://stackoverflow.com/questions/24711232/button-click-in-android-wear
   http://toastdroid.com/2014/08/18/messageapi-simple-conversations-with-android-wear

2. Wear:
   a. WearActivity Activity
      i. Hit Button that forward message to Phone App
      ii. Stand Button that forward message to Phone App
   b. ListenerService
      i. Start Message starts activity with intent
      ii. Win Message vibrates once for a long time
      iii. Lose Message vibrates twice for a short time
      iv. Draw Message vibrates thrice for a very short time

Application User Interface

On the phone side the application has 4 activities as indicated in the diagram and flow. The MainActivity fragment is a list fragment that points to the 3 other activities we have. Below can be seen the layout of that activity:

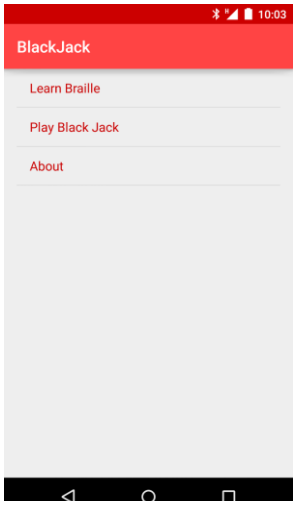


Figure 1 - MainActivity Layout

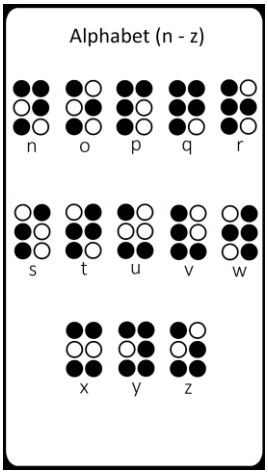


Figure 2 - An example of a card

The LearnBraille activity houses a pager and fragment. It simply swipes cards of braille characters to teach it to the user as can be seen above.

The PlayBlackJack Activity houses two fragment, a start fragment to simply display the layout of how the game looks like as can be seen below:

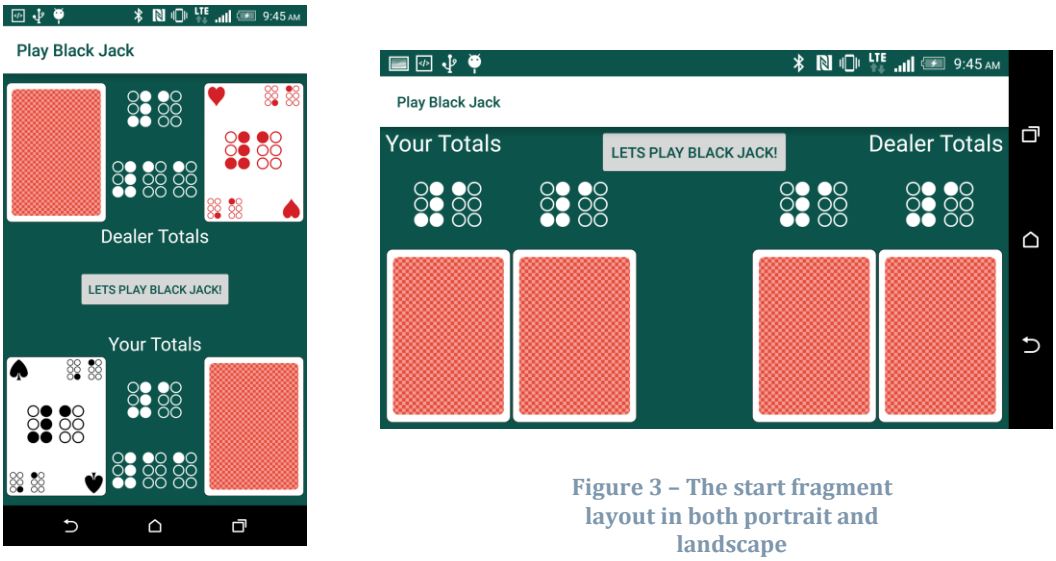


Figure 3 – The start fragment layout in both portrait and landscape

The other fragment, playing fragment, has the same layout with four buttons and can be seen below:

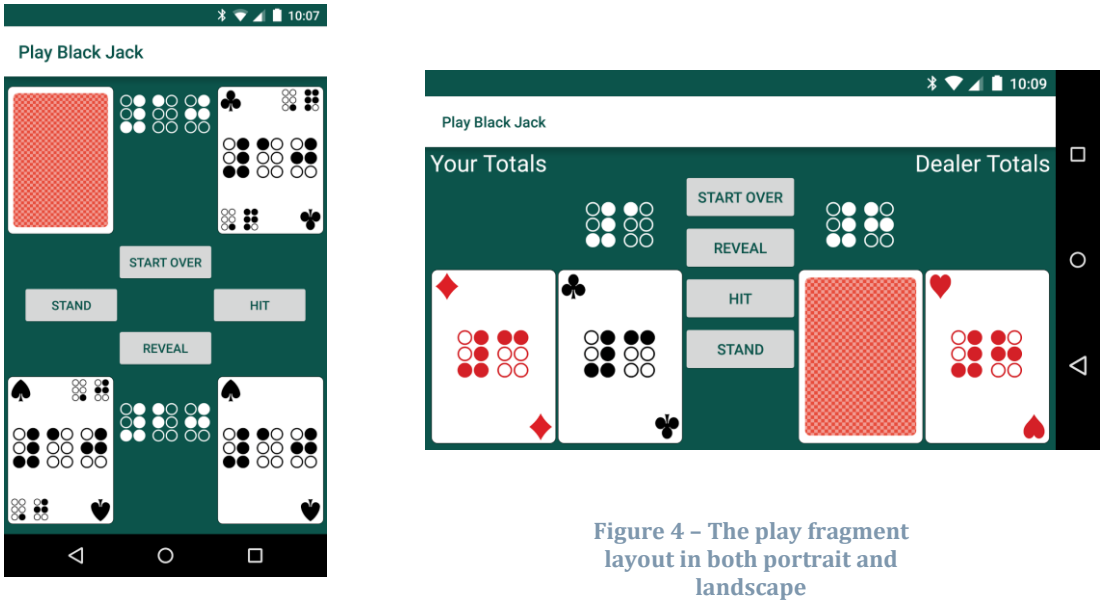


Figure 4 – The play fragment layout in both portrait and landscape

The third activity simply has TextViews to display our names and the acknowledgements.

On the Android Wear (smartwatch) side we only have a single activity in terms of UI. The activity houses two buttons and can be seen below:

Figure 4 – The Watch layout for both round and square watches

The Braille Font used in all the media assets is free for use under the Ubuntu Font License 1.0. It is called WLM Braille font and can be found in http://www.fontspace.com/wlm-fonts/wlm-braille.

Blackjack Logic

Wikipedia has a good and concise explanation of how blackjack works so we have quoted it below from the following link (http://en.wikipedia.org/wiki/Blackjack):

"*Blackjack is a comparing card game between a player and dealer, meaning that players compete against the dealer but not against any other players. It is played with one or more decks of 52 cards. The object of the game is to beat the dealer, which can be done in a number of ways:*

- *Get 21 points on the player's first two cards (called a blackjack), without a dealer blackjack;*
- *Reach a final score higher than the dealer without exceeding 21; or*
- *Let the dealer draw additional cards until his or her hand exceeds 21.*

The player or players are dealt an initial two-card hand and add together the value of their cards. Face cards (kings, queens, and jacks) are counted as ten points. A player and the dealer can count his or her own ace as 1 point or 11 points. All other cards are counted as the numeric value shown on the card. After receiving their initial two cards, players have the option of getting a "hit", or taking an additional card. In a given round, the player or the dealer wins by having a score of 21 or by having the highest score that is less than 21. Scoring higher than 21 (called "busting" or "going bust") results in a loss. A player may win by having any final score equal to or less than 21 if the dealer busts. If a player holds an ace valued as 11, the hand is called "soft", meaning that the player cannot go bust by taking an additional card; 11 plus the value of any other card will always be less than or equal to 21. Otherwise, the hand is "hard".

The dealer has to take hits until his or her cards total 17 or more points. (In some casinos the dealer also hits on a "soft" 17, e.g. an initial ace and six.) Players win if they do not bust and have a total that is higher than the dealer's. The dealer loses if he or she busts or has a lesser hand than the player who has not

busted. If the player and dealer have the same total, this is called a "push" and the player typically does not win or lose money on that hand."

Storage

Two storage methods were used for our application. The first is the use of Shared Preferences to save a single flag. The flag tells us if this is the first time the user has installed the application. If it is we create a new SQLite database that contains a single deck of 52 cards. Otherwise we continue on for the remainder of any launches of the app in the future. Below is a code snippet of how we do this:

```
SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
boolean doesDatabaseExist = settings.getBoolean("DatabaseExists", false);
if(!doesDatabaseExist){
        //... (code to create new DB goes here)
        SharedPreferences.Editor editor = settings.edit();
        doesDatabaseExist = true;
        editor.putBoolean("DatabaseExists", doesDatabaseExist);
        editor.commit();
}else{}
```

The second storage method, which was already mentioned, is the SQLite database. This database houses a card object. We define a card object of having a key to identify the card, a description to be used for printing and talk-back, a drawable pointer to point to the image path, and finally the card's value. In SQLite you basically create a table of what you want with the fields and field types this table you want to have. To do that we declare a String as can be seen below and note the use of an "`INTEGER PRIMARY KEY AUTOINCREMENT`" as we'll discuss why it was used later:

```
private static final String CREATE_TABLE_CARD = "CREATE TABLE " + TABLE_NAME + "(" + UNIQUE_ID
+ " INTEGER PRIMARY KEY AUTOINCREMENT," + CARD_KEY + " TEXT," + CARD_DESCRIPTION + " TEXT," +
CARD_DRAWABLE + " INTEGER," + CARD_VALUE + " INTEGER" + ")";
```

Now in android to use SQLite you need to extend `SQLiteOpenHelper` and implement two methods, `onCreate` and `onUpgrade`. Below we show how we create our database:

```
    @Override
    public void onCreate(SQLiteDatabase db){
        Log.d(LOG, "The Database onCreate called");
        // Create the table
        db.execSQL(CREATE_TABLE_CARD);
    }
```

Inserting to the database is straightforward and that is done by using Android's `contentValues.put()` method and inserting the content values with the `insert()` method. The method can be seen below:

```
public long insertCard(String key, String description, int drawable, int value){
        SQLiteDatabase db = storage.getWritableDatabase();
        ContentValues contentValues = new ContentValues();
        contentValues.put(DeckDatabase.getCardKey(), key);
        contentValues.put(DeckDatabase.getCardDescription(), description);
        contentValues.put(DeckDatabase.getCardDrawable(), drawable);
        contentValues.put(DeckDatabase.getCardValue(), value);

        long id = db.insert(DeckDatabase.getTableName(), null, contentValues);
```

```
        db.close();
        return id;
    }
```

To query the database we use a Cursor object to navigate through it. The Cursor object requires a unique ID and this is ensured by the "`INTEGER PRIMARY KEY AUTOINCREMENT`". Without it the cursor would not function properly. The query method returns a card object to the method caller. For a detailed look see the `getCard(String key)` method in the Deck class in the JavaDocs.

Phone & Wear Communication

Communication between the phone application and the wear application uses the Google MessageApi. As per Android Developer's website explains in the following:

"*Messages are delivered to connected network nodes. A message is considered successful if it has been queued for delivery to the specified node. A message will only be queued if the specified node is connected. The DataApi should be used for messages to nodes which are not currently connected (to be delivered on connection).*

Messages should generally contain ephemeral, small payloads. Use assets with the DataApi to store more persistent or larger data efficiently.

A message is private to the application that created it and accessible only by that application on other nodes."

First and for most to be able to send or receive messages a `GoogleApiClient` needs to be set up. Below is the set-up for the client on the phone side:

```
mGoogleApiClient = new GoogleApiClient.Builder(context)
                .addApi(Wearable.API)
                .addConnectionCallbacks(this)
                .addOnConnectionFailedListener(this)
                .build();
```

Note the `addApi(Wearable.API)` and `build()` to note that type of api we want to use and to build that api. On the Wear side only those two methods were used to set-up the client.

On the phone side since our app only cares about message events when the user is interacting with the app and does not need a long-running service to handle every data change, we can listen for events in the fragment by implementing `MessageApi.MessageListener` when. We connect the client in `onStart` using the `mGoogleApiClient.connect()` method. When a connection is established we add the listener by calling `Wearable.MessageApi.addListener(mGoogleApiClient, this)`. Finally our listener is all set-up now and can handle messages when received using the method below:

```
    @Override
    public void onMessageReceived(MessageEvent messageEvent) {
        Log.d(TAG, "In onMessageReceived");
```

```
        if("#MESSAGE".equals(messageEvent.getPath())) {
            // launch some Activity or do anything you like
            CharSequence text = "Wear Sent a Message! Hit was Pressed!!! :)";
            int duration = Toast.LENGTH_SHORT;

            Toast toast = Toast.makeText(context, text, duration);
            toast.show();
        }  else if("#HIT".equals(messageEvent.getPath())){
            getActivity().runOnUiThread(new Runnable(){
                public void run() {
                    if(button_hit_state){
                        button_hit.performClick();
                    }
                }
            });
        } else if("#STAND".equals(messageEvent.getPath())){
            getActivity().runOnUiThread(new Runnable(){
                public void run() {
                    if(button_stand_state){
                        button_stand.performClick();
                    }
                }
            });
        }
    }
```

Note that we have implemented the methods needed in `MessageApi.MessageListener`. The messages passed to the listener are strings. Only two messages can be sent to the phone, #HIT and #MISS, indicating the button presses.

On the Wear we want a dedicated `ListenerService` to be listening to messages sent from the phone. We implement a service because we would like to launch an activity and that can only be done from the background in our case which the `ListenerService` can provide. We simply declare the service in the Android Manifest on the Wear app and implement the `WearableListenerService` in its own class.

Five messages can be received by the service on the wear. The first is #MESSAGE that was used for testing, second is #START to start the wear activity from an intent, third is #WIN that vibrates once for a long time, fourth is #LOSE that vibrates twice for a short time, fifth and final is #DRAW that vibrates thrice for a very short time.

To send an actual method we first need to get a list of connected nodes or devices, if a node exists we send a message using the `Wearable.MessageApi.sendMessage` method. Below is a snippet of this being done on the Wear application (note that it's the same on the phone side):

```
final PendingResult<NodeApi.GetConnectedNodesResult> nodes
= Wearable.NodeApi.getConnectedNodes(mGoogleApiClient);
nodes.setResultCallback(new ResultCallback<NodeApi.GetConnectedNodesResult>() {
@Override
public void onResult(NodeApi.GetConnectedNodesResult result) {
       final List<Node> nodes = result.getNodes();
              if (nodes != null) {
                     Log.d(TAG, "Going to send message");
                     for (int i = 0; i < nodes.size(); i++) {
                            Log.d(TAG, "Sending part: " + i);
                            final Node node = nodes.get(i);
```

```
                                    // You can just send a message
                                    Wearable.MessageApi.sendMessage(mGoogleApiClient,
                                     node.getId(), SEND_HIT_MESSAGE, null);
                                }
                            }
                        }
});
```

Transition Animations

To display card changes on the screen during the game we immediately recognized that we cannot have cards come into view instantly. If that happen when a user hits for example and then stand the change will be immediate and quite jarring. We want the user to be able to notice the change and give him time to process the change. Ultimately we want him to see the new card with UEB numbers, recognize the number and be able to count the card totals in their heads.

To achieve this goal we use transition animations introduced in Android Lollipop. We specifically a transition called “Explode” which as the name suggests a view item (such as an `ImageView`) explodes into or out of view. To do this we use declare a `TransitionManager` as can be seen below:

```
TransitionManager.beginDelayedTransition(viewGroup, new Explode());
```

After we declare the transition we need to toggle the visibility of the View item and we use the following method to do that:

```
 private static void toggleVisibility(View... views) {
        for (View view : views) {
            boolean isVisible = view.getVisibility() == View.VISIBLE;
            view.setVisibility(isVisible ? View.INVISIBLE : View.VISIBLE);
        }
    }
```

Since we need to make the view item visible again we need to call the `toggleVisibility` twice. Calling it twice one after the other though will make the transition happen instantly rending the animation useless. We needed to add a delay for the animation to go out and in to view. To accomplish this we use an `AsyncTask`. We use the `onPreExecute` to hide the view, `doInBackground` to wait (sleep) for 1 second and `onPostExecute` to reveal the view again. One example can be seen below:

```
private class AnimatePlayerCards extends AsyncTask<Integer, Void, Integer[]> {
        @Override
        protected void onPreExecute() {
            Log.d(TAG, "AnimatePlayerCards");
            TransitionManager.beginDelayedTransition(group, new Explode());
            toggleVisibility(player_left_slot, player_right_slot);

            changeAllButtonStates(false, false, false, false);
        }

        @Override
        protected Integer[] doInBackground(Integer... params) {
```

```
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return params;
        }

        @Override
        protected void onPostExecute(Integer... params) {
            player_left_slot.setImageResource(params[0]);
            player_right_slot.setImageResource(params[1]);

            TransitionManager.beginDelayedTransition(group, new Explode());
            toggleVisibility(player_left_slot, player_right_slot);

            changeAllButtonStates(true, true, true, true);
        }
    }
```