Section C:

Techniques Used:

1. Array of raw bike station data.
2. Parsing the raw data array and creating multiple arrays to organize data.
3. Class extending to AsyncTasks
4. Using a Recycler View for the user interface
5. Use implicit and explicit intents.

Technique 1:

```
public final class ParseJsonLiveDataFeed {

    private static final String TAG = "parseActivity";

    public static String [] getBikeStaionDataFromJson (String bikeStaionLiveJsonDataStr) throws JSONException {
        final String LBSD_LIST = "stationBeanList";
        final String LBSD_STATION_NAME = "stationName";
        final String LBSD_AVAILABLE_BIKES = "availableBikes";
        final String LBSD_AVAILABLE_DOCKS = "availableDocks";
        final String LBSD_LATITUDE = "latitude";
        final String LBSD_LONGITUDE = "longitude";
        final String LBSD_STATUS = "statusValue";

        JSONObject liveBikeJsonFeed = new JSONObject(bikeStaionLiveJsonDataStr);

        JSONArray bikeStaionArray = liveBikeJsonFeed.getJSONArray(LBSD_LIST);
        System.out.println(bikeStaionArray.length());

        String [] parsedBikeStationData = new String[bikeStaionArray.length()];

        for (int i = 0; i < bikeStaionArray.length(); i++) {
            String stationName;
            int availableDocks;
            int availableBikes;

            double latitude;
            double longitude;
            String status;

            JSONObject bikeStationObject = bikeStaionArray.getJSONObject(i);
            System.out.println(bikeStaionArray.getJSONObject(i));

            stationName = bikeStationObject.getString(LBSD_STATION_NAME);
            System.out.println(stationName);
            availableDocks = bikeStationObject.getInt(LBSD_AVAILABLE_DOCKS);
            availableBikes = bikeStationObject.getInt(LBSD_AVAILABLE_BIKES);
            latitude = bikeStationObject.getDouble(LBSD_LATITUDE);
            longitude = bikeStationObject.getDouble(LBSD_LONGITUDE);
            status = bikeStationObject.getString(LBSD_STATUS);

            parsedBikeStationData[i] = (stationName + "\n" + status + "\n" + availableBikes + "\n" + availableDocks
                    + "\n" + latitude + "\n" + longitude);
        }

        return parsedBikeStationData;
    }
}
```

The above picture of code is the entire class used for parsing the JSON data feed, which is super critical for technique one. At first I established a connection between my app and the JSON data feed, I then created a JSON array and this was because I needed to figure out how many bike station JSON objects there were. Once I got that I created a String array containing all

the raw data values and separated each raw data value by a new line for each bike station. This method goes to the JSON data source and looks for the keywords of the JSON object then stores it into the bike station array. The reason I chose to do this raw data String array was because I wanted to return all of the values and eventual easily using another class take each individual value and use it to add more functions to the app. (How to Parse JSON in Android).

Technique 2:

```
public class JsonDataContainer {

    private String [] mBikeStaitonName;
    private String [] mBikeStationStatus;
    private int [] mBikesAvailable;
    private int [] mDocksAvailable;
    private double [] mLatitude;
    private double [] mLongitude;
    private String [] mBikeStationDetails;

    public void parseRawBikeStationData (String [] mBikeStaitonDataRaw) {
        String [] mName = new String [mBikeStaitonDataRaw.length];
        String [] mStatus = new String[mBikeStaitonDataRaw.length];
        int [] mBikes =  new int[mBikeStaitonDataRaw.length];
        int [] mDocks = new int[mBikeStaitonDataRaw.length];
        double [] mLat = new double[mBikeStaitonDataRaw.length];
        double [] mLon = new double[mBikeStaitonDataRaw.length];
        String [] mDetails = new String[mBikeStaitonDataRaw.length];

        for (int i = 0; i < mBikeStaitonDataRaw.length; i++) {
            Scanner sc = new Scanner(mBikeStaitonDataRaw[i]);
            mName[i] = sc.nextLine();
            mStatus[i] = sc.nextLine();
            mBikes [i]= sc.nextInt();
            mDocks[i] = sc.nextInt();
            mLat[i] = sc.nextDouble();
            mLon[i] = sc.nextDouble();
            mDetails [i] = ("The Bike Station at " + mName[i] + " is " + mStatus[i] + " and has " + mBikes[i] +
                    " bikes and " + mDocks[i]+ " parking docks.");
        }

        mBikeStaitonName = mName;
        mBikeStationStatus = mStatus;
        mBikesAvailable = mBikes;
        mDocksAvailable = mDocks;
        mLatitude = mLat;
        mLongitude = mLon;
        mBikeStationDetails = mDetails;

    }

    public String [] getmBikeStaitonName () { return mBikeStaitonName; }

    public String [] getmBikeStationStatus () { return mBikeStationStatus; }

    public int [] getmBikesAvailable () { return mBikesAvailable; }

    public int [] getmDocksAvailable () { return mDocksAvailable; }

    public double [] getmLatitude () { return mLatitude; }

    public double [] getmLongitude () { return mLongitude; }

    public String [] getmBikeStationDetails () { return mBikeStationDetails; }
}
```

The above picture of code is the entire class used for taking the raw data array and storing it in to new data arrays. Once I took the raw bike station data, created a Container to store every single individual value in arrays. Since there is a new line separating each individual array I either called scanner.nextLine(), to get the String values or scanner.nextInt() for int values and scanner.nextDouble() for double values. The for loop was used to fill all of the arrays and make sure they went into the right station. I then created all of these getters so if a JsonDataContainer objects is present within the other classes, it would be very easy to grab any piece of data from the JSON data feed. The reason for making this class was again to when called, return different types of data allowing for a source of all data to be made.

Technique 3:

```
public class FetchBikeStaionData extends AsyncTask<String, Void, JsonDataContainer> {

    @Override
    protected JsonDataContainer doInBackground(String... params) {
        if (params.length == 0) {
            return  null;
        }

        String citiBikeUrlString = params[0];
        URL bikeStaitonResponseUrl = EstabilshConnectionToNetwork.urlBuilder(citiBikeUrlString);

        try {
            String jsonBikeStationResponse = EstabilshConnectionToNetwork.getResponseFromUrlConnection(bikeStaitonResponseUrl);
            String [] simpleJsonBikeData = ParseJsonLiveDataFeed
                    .getBikeStaionDataFromJson(jsonBikeStationResponse);
            JsonDataContainer jsonDataContainer = new JsonDataContainer();
            jsonDataContainer.parseRawBikeStationData(simpleJsonBikeData);
            return jsonDataContainer;
        } catch (Exception e){
            e.printStackTrace();
            return null;
        }
    }


    @Override
    protected void onPostExecute(final JsonDataContainer rawBikeStationData) {
        mLoadingIndicator.setVisibility(View.INVISIBLE);
        if (mSwipeRefreshLayout.isRefreshing()){
            mSwipeRefreshLayout.setRefreshing(false);
        }
        if (rawBikeStationData != null ) {
            showBikeSationData();
            mBikeStationAdapter.setmBikeStationDataRaw(rawBikeStationData);
        }
        else {
            showErrorMessage();
        }
    }
}
```

The above picture of code is the entire class which extends AsyncTasks which allows me to move in between threads. For this project, I used two threads to control the background processes and main process of the app using a class extending AsyncTask. The whole purpose of having this class extend AsyncTaks is so my application is able to do background processes, such as connecting to the internet and parsing all of the data without causing the application to either crash or lag. By overriding the doInBackground method, I tell Java to work in the

background thread to create the URL, establish a connection, and parse the JSON data feed while not having to worry about the application timing out and crashing. Then by overriding the onPostExecute I am able to send all of the parsed data back to the main thread and do things like set the title to the text view holder or send it to the detail activity or Google Maps with the required values. (AsyncTask), (Addingg Swipe-to-Refresh To Your App), (Developing Android Apps).

Technique 4:

```
@Override
public BikeStationAdapterViewHolder onCreateViewHolder(ViewGroup viewGroup, int viewType) {
    Context context = viewGroup.getContext();
    int layouIdForListItem = R.layout.bike_station_list;
    LayoutInflater inflater = LayoutInflater.from(context);
    boolean shouldAttachToParentImmediately = false;

    View view = inflater.inflate(layouIdForListItem,viewGroup,shouldAttachToParentImmediately);
    return new BikeStationAdapterViewHolder(view);
}

@Override
public void onBindViewHolder(BikeStationAdapterViewHolder bikeStationAdapterViewHolder, int position) {
    String nameForSingleBikeStation = mBikeStationName[position];
    int bikeNumberForSingleBikeStaiton = mStationBikes [position];
    String statusForSingleBikeStation = mServiceStatus [position];
    if (bikeNumberForSingleBikeStaiton == 0) {
        bikeStationAdapterViewHolder.mBikeStationTextView.setTextColor(Color.GRAY);
    }
    else if (statusForSingleBikeStation.equals("Not In Service")){
        bikeStationAdapterViewHolder.mBikeStationTextView.setTextColor(Color.RED);
    }
    else {
        bikeStationAdapterViewHolder.mBikeStationTextView.setTextColor(Color.BLACK);
    }
    bikeStationAdapterViewHolder.mBikeStationTextView.setText(nameForSingleBikeStation);
}

@Override
public int getItemCount() {
    if (mBikeStationName == null){
        return 0;
    }
    return mBikeStationName.length;
}

public void setmBikeStationDataRaw (JsonDataContainer jsonDataContainer) {
    mBikeStationName = jsonDataContainer.getmBikeStaitonName();
    mBikeStationDetails = jsonDataContainer.getmBikeStationDetails();
    mStationLatitude = jsonDataContainer.getmLatitude();
    mStationLongitude = jsonDataContainer.getmLongitude();
    mStationBikes = jsonDataContainer.getmBikesAvailable();
    mServiceStatus = jsonDataContainer.getmBikeStationStatus();
    notifyDataSetChanged();
}
```

The section of code above are the four main methods in a Recycler View adapter which is needed in order to make the Recycler View work effectively. A Recycler View is a type of view that is very similar to a scroll view however, a recycler view recycles what is on the screen reducing the amount of memory needed to execute and display the list of view holders, creating a smoother, user friendly user interface. The onCreateViewHolder method is overridden in this class and is used to inflate the view holder by the size of the text going into the view holder. The onBindViewHolder is also overridden in this class and the purpose of it is to set the text to the view holder and the text color of the text if there are any errors. The getItemCount is overridden in this class and its function is to simply return the length of the list. Then the setBikeStationDataRaw is used in this class to get values out of the Json data container. (Developing Android Apps).

Technique 5:

```
@Override
public void onClick(String singleBikeStationData, double stationLatitude, double stationLongitude) {
    Class destinationClass = BikeStationDetail.class;
    Intent intentToDetail = new Intent(this,destinationClass);
    Bundle extras = new Bundle();
    extras.putString("EXTRA_STATION_DETIALS", singleBikeStationData);
    extras.putDouble("EXTRA_LATITUDE", stationLatitude);
    extras.putDouble("EXTRA_LONGITUDE", stationLongitude);
    intentToDetail.putExtras(extras);
    startActivity(intentToDetail);
}
```

```
    Intent intentThatStartedThisActivity = getIntent();
    Bundle extras = intentThatStartedThisActivity.getExtras();

    if(intentThatStartedThisActivity != null) {
        mBikeStationData = extras.getString("EXTRA_STATION_DETIALS");
        mBikeStationDetailDisplay.setText(mBikeStationData);
        mBikeStationDetailDisplay.setTextColor(Color.BLACK);

        mStationLatitude = extras.getDouble("EXTRA_LATITUDE");
        mStationLongitude = extras.getDouble("EXTRA_LONGITUDE");
    }
}

private void openMapIntent () {
    Uri geoLocation = Uri.parse("google.navigation:q=" + mStationLatitude + "," + mStationLongitude);
    Intent openMap = new Intent(Intent.ACTION_VIEW);
    openMap.setData(geoLocation);
    if (openMap.resolveActivity(getPackageManager()) != null) {
        startActivity(openMap);
    }
    else {
        Toast.makeText(this, "You have no apps that can open up a map! Please install one and try again later.", Toast.LENGTH_LONG).show();
    }
}
```

The first picture of code with the overridden onClick method from the main class demonstrates an explicit intent. When a view holder is clicked, a bundle of extras would be passed into a child activity, and this was done because it was important not to overpopulate the main activity with the list. Then in the second picture, the extraction of the bundle of extras can be seen above the openMapIntent method. The openMapIntent method is actually an implicit intent to Google

Maps with the coordinates stored in the extras passed through here, allowing the user to get directions to the chosen bike station. (Launching Google Maps Directions via an Intent on Android), (Android, Can I Use PutExtra to Pass Multiple Values), (Developing Android Apps).

WC: 802