Introduction

Sometimes it is needed to display a set of scrollable items in android that can be dynamically manipulated. The scrollable items are collected at a List (ArrayList) object that holds all items that are desired to be properly displayed. For example, users are the scrollable items in the developed app. The main goal is to show the items (users) of the List on the screen through a scrollable visualization. For this purpose, two objects are needed namely ListView and ArrayAdapter.

A ListView is a Java public class that extends form AbsListView that groups several items and shows them in a vertical scrollable list. It should be noted that the list is automatically made scrollable whenever it is needed based on the amount of the data. It is very necessary to understand that the ListView is only responsible for visually presenting the items.

An ArrayAdapter is another Java public class that extends BaseAdapter class that adapts the items form the List and provide them for the ListView. The ArrayAdapter conceptually is a bridge between the items List and ListView objects. Indeed, the ArrayAdapter is responsible for providing a view that locates at a specific position for the ListView.

Creating the ListView

Creating a ListView consists of two separate steps that are mentioned in this subsection. In the first step, an XML layout should be designed to represent the view template of each item (user). In the developed app, each user is supposed to have username, phone number, email address, and a self-picture. Hence, the XML layout should be properly designed that is able to represent the related data of each item (user). For this purpose, the XML layout (located at /res/layout/item.xml) is designed as follows:

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal" android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="3dp"
    android:paddingBottom="3dp"
    android:paddingRight="3dp"
    android:paddingTop="3dp">
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:layout_gravity="center_horizontal|center_vertical">
        <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:orientation="vertical"
            android:layout_gravity="center_horizontal|center_vertical">
            <ImageView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:id="@+id/iv"
                />
        </LinearLayout>
        <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:orientation="vertical"
            android:layout_gravity="center_horizontal|center_vertical">
            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text=""
                android:textColor="#000000"
                android:textSize="20dp"
                android:id="@+id/tvu"/>
        </LinearLayout>
    </LinearLayout>
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:layout_gravity="center_horizontal|center_vertical"
        android:paddingLeft="10dp">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text=""
            android:textColor="#000000"
            android:textSize="20dp"
            android:id="@+id/tve"/>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text=""
            android:textColor="#000000"
            android:textSize="20dp"
            android:id="@+id/tvp"/>
    </LinearLayout>
</LinearLayout>
```

As can be seen, the design has free structure and any view template can be considered for each item in the ListView.

In the second step, the ListView should be added into the main-activity’s layout as an ordinary view which is explained in the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.example.listviewapplication.MainActivity">

    <ListView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/lv">

    </ListView>
</RelativeLayout>
```

Note that, the ListView must be inserted into the main XML layout of the activity such as above. Now, the ListView is fully designed and created and it can be exploited in the app which is explained in the next subsections.

Defining the ArrayAdapter

This subsection describes how to define the appropriate ArrayAdapter. For this purpose, myAdapter class is defined through extending an ordinary class for the ArrayAdapter class as given below:

```
public class myAdapter extends ArrayAdapter<User> {
    public ArrayList<User> users;
    public myAdapter(Context context, int resource,ArrayList<User> users) {
        super(context, 0, users);
        this.users=users;
    }
}
```

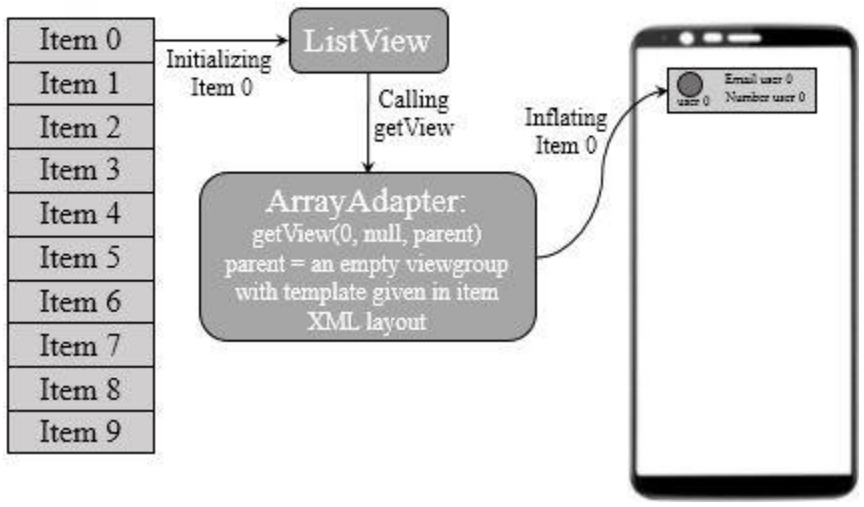
It should be noted that the ArrayAdapter should have the constructor method that supers its parent object. As can be seen, the ArrayAdapter directly uses the ArrayList<User> that contains the users' data.

As stated earlier, the ArrayAdapter is responsible for binding items (users from the ArrayList<User>) for the ListView. The ListView asks the ArrayAdapter to update specifically one of the items (users) through calling method getView() in the ArrayAdapter as presented here:

```
@Override
public View getView(int position,View convertView,ViewGroup parent){

}
```

The ListView calls getView() method in the ArrayAdapter class whenever it is needed to update an item. This method is called at two situations which are the initializing (or invalidation) and the recycling situation. Fig. 1 is provided to describe the initializing situation for Item 0:



As Fig. 1 explains, the ListView calls the getView method with the mentioned input arguments. The first and second arguments are respectively set 0 and null and the third one is an empty viewgroup whose template is the same as created in the item's XML layout (item.xml in the app).

In the initializing situation, the ArrayAdapter should properly inflate the view based on the view template (in the parent argument) and informations of the position specified Item (users[position] contains the data). For this purpose, the getView method is recoded as here:

```
public View getView(int position,View convertView,ViewGroup parent){
    User user=users.get(position);
    View rowView=convertView;
    if(convertView==null){
        rowView=
LayoutInflater.from(getContext()).inflate(R.layout.item,parent,false);
    }
    ImageView iv = (ImageView) rowView.findViewById(R.id.iv);
    TextView tvu = (TextView) rowView.findViewById(R.id.tvu);
    TextView tvp = (TextView) rowView.findViewById(R.id.tvp);
    TextView tve = (TextView) rowView.findViewById(R.id.tve);
    iv.setImageResource(user.imageSource);
    tvu.setText(user.username);
    tvp.setText("Phone = "+user.phoneNumber);
    tve.setText("Email = "+user.emailAddress);
    return rowView;
}
```

As can be easily seen, the ArrayAdapter utilizes the position index and its related user properties to update the views of the rowView. The rowView is obtained through inflating the parent and considering the item XML layout. Then, the adapter uses the item’s data (username, phone number, email address, and image resource) in the children of the rowView via setting these texts.

Attaching the ArrayAdapter to the ListView

This subsection explains how the implemented ArrayAdapter (myAdapter) is attached to the created ListView. For this purpose, the code is written as presented in the main activity:

```
public class MainActivity extends AppCompatActivity {
    public ListView listView;
    public myAdapter adapter;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        User user=new User("person 1","9999","A@B.com",R.drawable.pic1);
        ArrayList<User> users=new ArrayList<User>();
        users.add(user);
        user=new User("person 2","9998","C@B.com",R.drawable.pic2);
        users.add(user);
        adapter=new myAdapter(getApplicationContext(),0,users);
        listView=(ListView)findViewById(R.id.lv);
        listView.setAdapter(adapter);
    }
}
```

The ArrayAdapter is attached by calling the setAdapter() method in the ListView scope as presented above. This method gets the corresponding ArrayAdapter to automatically call its getView() method whenever it is required.

Note that, the ListView asks the ArrayAdapter to initially display the items eventfully after evaluating its setAdapter() method. Hence, the List of items (ArrayList<User> users) must be set in the ArrayAdapter to inform the ListView about the existing items.

The user class is also considered to be as follows:

```
public class User {
    public String username;
    public String phoneNumber;
    public String emailAddress;
    public int imageSource;
    public User(String username,String phoneNumber,String emailAddress,int
imageSource){
        this.username=username;
        this.phoneNumber=phoneNumber;
        this.emailAddress=emailAddress;
        this.imageSource=imageSource;
    }
}
```

The results are presented in the Fig. 2:

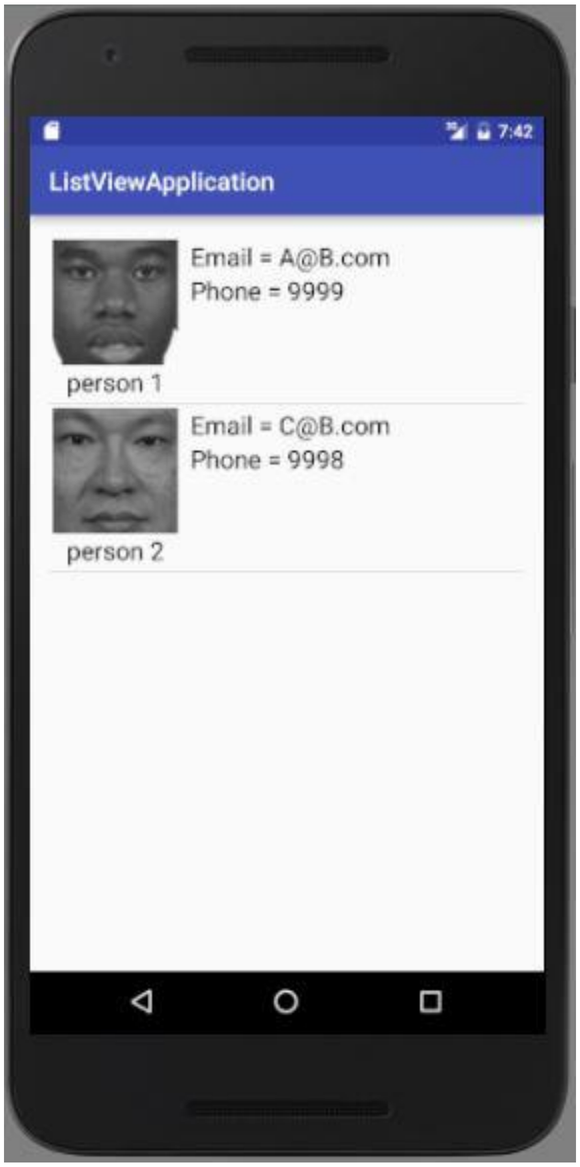


Fig. 2 properly shows the stored items in the ArrayAdapter.

ListView recycling

It must be emphasized that only a few numbers of items are inflated in the initial (invalidated) situation and the other ones will be shown via a recycling trick. Android framework exploits this trick for preventing the memory shortage whenever the number of items is very large. To schematically describe this trick, the following image is provided in the following:

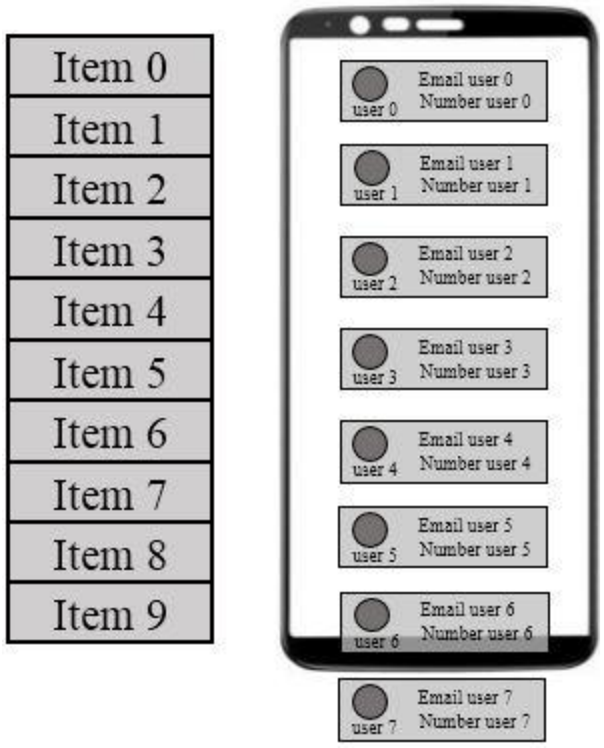
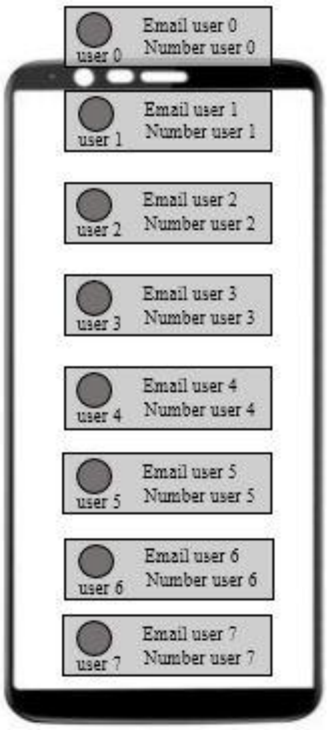
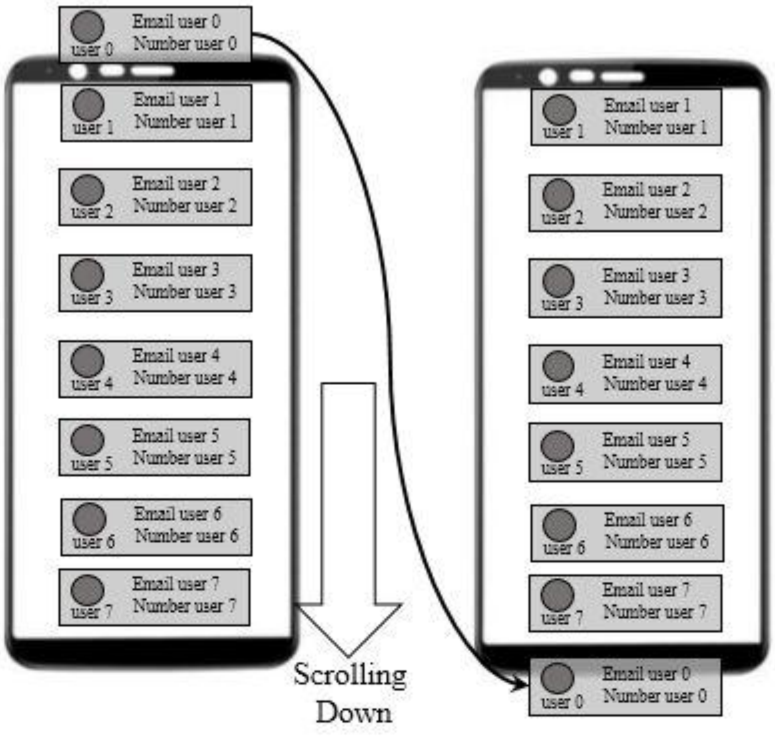


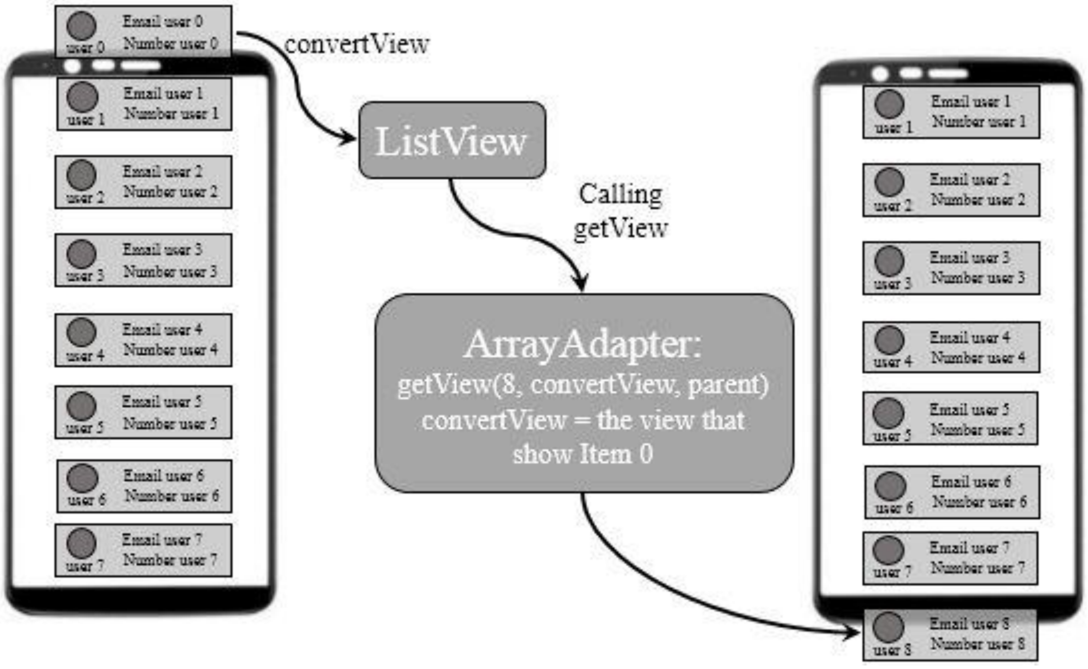
Fig. 3 demonstrates that the ListView shows 7 items (note that, this specific number is only used as an instance value) while there are 10 items in the ArrayAdapter. Now, the ListView becomes scrollable to be able to show the whole items. Assume we scroll down the screen in Fig. 3 to see Item 8 that is not inflated in the initial situation. In this situation, Item 0 is fully scraped and Item 8 is needed to be shown as presented in the next image:



In this situation, the ListView recycle Item 0 through locating it at the bottom of the view. This trick known as the ListView recycling that is schematically shown in the following figure:



The ListView recycling is another situation that asks the ArrayAdapter to update the recycled item (as explained in Fig. 4). In fact, the ListView calls the getView() method to update the view as explained the following figure:



According to Fig. 6, the ArrayAdapter should update convertView which is implemented by two method namely findViewById-based and View Holder-based method. These method are separately described in the next subsections.

ListView recycling: findViewById-based method

This method simply updates the convertView data based on the position of the new item. This method can be easily implemented as presented here:

```
public View getView(int position,View convertView,ViewGroup parent){
    User user=users.get(position);
    View rowView=convertView;
    if(convertView==null){
        rowView=
LayoutInflater.from(getContext()).inflate(R.layout.item,parent,false);
    }else{
        rowView=convertView;
        Toast.makeText(getContext(),"hi",Toast.LENGTH_LONG).show();
    }
    ImageView iv = (ImageView) rowView.findViewById(R.id.iv);
    TextView tvu = (TextView) rowView.findViewById(R.id.tvu);
    TextView tvp = (TextView) rowView.findViewById(R.id.tvp);
    TextView tve = (TextView) rowView.findViewById(R.id.tve);
    iv.setImageResource(user.imageSource);
    tvu.setText(user.username);
    tvp.setText("Phone = "+user.phoneNumber);
    tve.setText("Email = "+user.emailAddress);
    return rowView;
}
```

Indeed, the ArrayAdapter can easily distinguish the recycling and initializing situations through checking the nullity of its second input argument convertView. If it is not null, the recycling occurs and the ArrayAdapter should update the values of the recycled view that is convertView in this situation. Hence. It can easily update the values by setting rowView to be an instance of the convertView as given in the above code.

Fig. 7 shows the results of evaluating the above code:

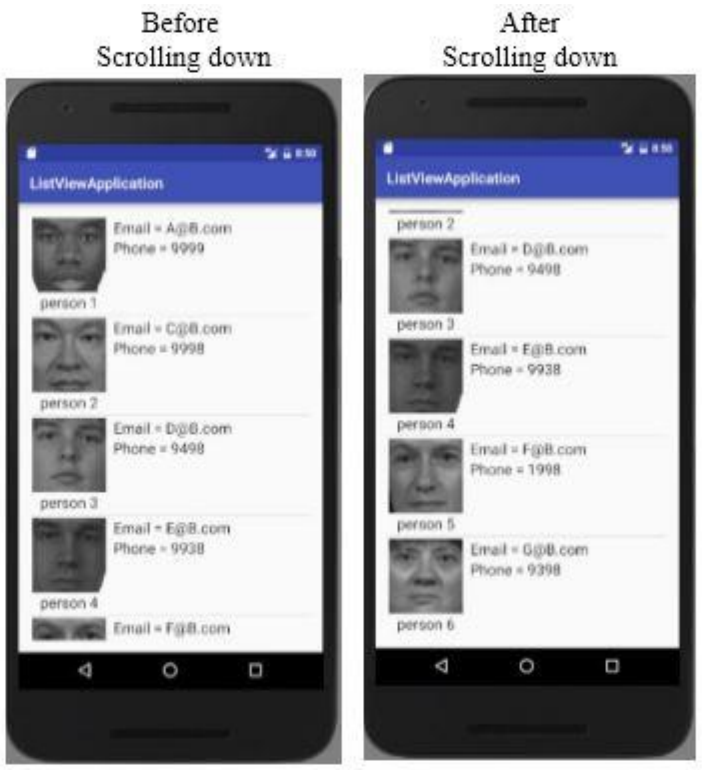


Fig. 7 reveals the proper functionality of the implemented code.

Although the findViewById-based method solves the recycling problem, this method is relatively slow. To cope with this issue, the ViewHolder-based method is developed that is proposed in the next subsection.

ListView recycling: ViewHolder-based method

The slowness of the previous method significantly disturbs the app's performance at the recycling situation. The ViewHolder-based method exploits the tag of the inflated items to get its child views instead of utilizing the findViewById method. In the ViewHolder-based method, the internal views of each inflated item are kept via setting them in a ViewHolder object at its related initializing situation. Then, the internal views that their values should be updated can be easily obtained via getting the ViewHolder. For the better understanding, the ViewHolder class of the developed app is given in the following:

```java
public  class  ViewHolder {
    public ImageView iv;
    public TextView tvu,tvp,tve;
    public ViewHolder(ImageView iv,TextView tvu,TextView tvp,TextView tve){
        this.iv=iv;
        this.tvu=tvu;
        this.tve=tve;
        this.tvp=tvp;
    }
}
```

In fact, the ViewHolder easily contains the internal vies of the item layout which are an imageview and three textviews in the developed app.

Using the ViewHolder, the getView() method is modified as follows:

```java
public View getView(int position,View convertView,ViewGroup parent){
    User user=users.get(position);
    View rowView=convertView;
    ImageView iv;
    TextView tvu,tvp,tve;
    if(convertView==null){
        rowView=
LayoutInflater.from(getContext()).inflate(R.layout.item,parent,false);
        iv = (ImageView) rowView.findViewById(R.id.iv);
        tvu = (TextView) rowView.findViewById(R.id.tvu);
        tvp = (TextView) rowView.findViewById(R.id.tvp);
        tve = (TextView) rowView.findViewById(R.id.tve);
        ViewHolder viewHolder=new ViewHolder(iv,tvu,tvp,tve);
        rowView.setTag(viewHolder);
    }else{
        rowView=convertView;
        ViewHolder viewHolder=(ViewHolder)rowView.getTag();
        iv=viewHolder.iv;
        tvu=viewHolder.tvu;
        tve=viewHolder.tve;
        tvp=viewHolder.tvp;
    }

    iv.setImageResource(user.imageSource);
    tvu.setText(user.username);
    tvp.setText("Phone = "+user.phoneNumber);
    tve.setText("Email = "+user.emailAddress);
    return rowView;
}
```

Above, an instance of ViewHolder is generated for each inflated item which contains all its internal views. The ViewHolder instance is kept as a tag in the rowView as presented above. Then, the internal views can be given via getting the tag of the convertView in the recycling situation. The rest of the code is the same as the previous one.