|  |
| --- |
| Android Tutorial – Part 3 |

|  |
| --- |
| 6-11-2018 |



Table of Contents

[Introduction 2](#_Toc516952318)

[Load contact List in the phone to a search box 3](#_Toc516952319)

[AutoCompleteTextView 4](#_Toc516952320)

[Class hierarchy of “**AutoCompleteTextView**” 4](#_Toc516952321)

[Important methods of AutoCompleteTextView 4](#_Toc516952322)

[Adapters in Android 5](#_Toc516952323)

[Why Adapters 6](#_Toc516952324)

[Commonly used Adapters in Android 6](#_Toc516952325)

[Cursors 6](#_Toc516952326)

[User Interface Design – Modification of **activity\_send\_message.xml** 7](#_Toc516952327)

[Remove the EditText from Layout 7](#_Toc516952328)

[Add a AutoCompleteTextView to Layout 8](#_Toc516952329)

[Java Code Modification – **SendMessageActivity.java** 11](#_Toc516952330)

[Remove all the references to “**editTextContactNumber**” from java code 11](#_Toc516952331)

[Access AutoCompleteTextView from Java 14](#_Toc516952332)

[App Permissions in Android 17](#_Toc516952333)

[Protection levels 17](#_Toc516952334)

[Load Contact List from The device 21](#_Toc516952335)

[Send a SMS in Android 27](#_Toc516952336)

[References 32](#_Toc516952337)

# Introduction

This is the part three of the android tutorial series. It is a continuation from last week. In order to follow this successfully, it is required to have,

* A basic understanding given about android in last session.
* The environment set up.
* The project created during last tutorial, opened in Android Studio.
* AVD or an Actual device ready for app deployment.

To catch up, in the last session (Android Tutorial Part 2),

* Android Studio’s UI Design View (Layout Editor)
* Android studio functionality
  + How to Rename a File
  + How to Compare Two Files
  + How to Delete a File
* Android Views and View Groups
  + Different Types of Views
* Design a User interface
  + How to add a Layout XML file
  + Positioning and Resizing Views
  + Constructing UIs using components in pallet
  + Changing attributes of different view
    - By Editing XML
    - By using “Attribute” window
  + Usage of strings.xml file
    - how to add a string resource
    - how to use a string resource in a layout
  + Usage of colors.xml file
    - how to add a color resource
    - how to use a color resource in a layout
  + Usage of styles.xml file
    - how to add a style resource
    - how to use a style resource in a layout
* Access and Modify UI from Activity class
  + **setContentView** Method
  + change application's startup activity
  + **findViewById** Method
  + **TextChangedListener**
  + **OnClickListener**
* Toasts
* Intents
  + Navigating between Activities
* Source code for this part can be found in Git Repository given below :- <https://github.com/nadee158/android_tutorial_part_2.git>

With that knowledge in hand, in this session below areas will be covered,

* AutoCompleteTextView
  + Attributes
  + Methods
* Adapters in Android
* App Permissions in Andorid
* Set a random string list as data source for AutoCompleteTextView
* Adding special permissions to “**AndroidManifest.xml**”
* Checking for app permissions
* Requesting permission from user
* Reading contact list on device
* Send an SMS from app

# Load contact List in the phone to a search box

First, recall the “**SendMessageActivity**” and “**activity\_send\_message.xml**”, from the last tutorial. The requirement which was given, and the resulted design was as below;

|  |  |
| --- | --- |
| Requirement:- | Result:- |
|  |  |

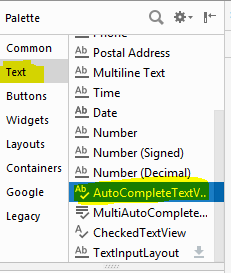
In the above requirement, it was only asked to let the user type the desired contact number in to the text box. Instead of doing this, it will be a lot easier for the user, if he could simply select from the existing contact list in the phone.

Therefore, as an enhancement to the work we already did, let’s make the “Contact Number” filed, a search from contact list rather than a text box.

Modified Requirement:-

* ***Let the user select Contact Number from the contact list rather than asking to type in***
* ***If the Contact Number is not in the list, allow the user to type the number***

In order to match the above requirement, we need to choose an appropriate view to replace the existing “**EditText**” component with id “**editTextContactNumber**”. For this purpose, a view exists in Android as “**AutoCompleteTextView**”, which is available under “**Text**” category in “**Palette**” view in android studio.



## AutoCompleteTextView

This is an editable text view that shows completion suggestions automatically while the user is typing in android apps, so no need to write all the characters of the word.

* Is an editable text field.
* Displays a list of suggestions in a drop down menu from which user can select only one suggestion or value.
* The list of suggestions is obtained from an **adapter** and it appears only after a number of characters that are specified in the threshold.
* Is a subclass of “**EditText**” class.
* The “**MultiAutoCompleteTextView**” is the subclass of “**AutoCompleteTextView**” class.

### Class hierarchy of “**AutoCompleteTextView**”

Object

View

TextView

EditText

AutoCompleteTextView

MultiAutoCompleteTextView

.

### Important methods of AutoCompleteTextView

* **getAdapter**() : This method returns a filterable **list adapter** used for auto completion
* **getCompletionHint**() : This method returns optional hint text displayed at the bottom of the matching list
* **getDropDownAnchor**() : This method returns the id for the view that the auto-complete drop down list is attached to
* **getListSelection**() : This method returns the position of the dropdown view selection, if there is any
* **isPopupShowing**() : This method indicates whether the popup menu is showing
* **setText**(CharSequence text, boolean filter) : This method sets text except that it can disable filtering
* **showDropDown**() : This method displays the drop down on screen
* **setAdapter** method is used to set the adapter of the “**AutoCompleteTextView”**.

## Adapters in Android

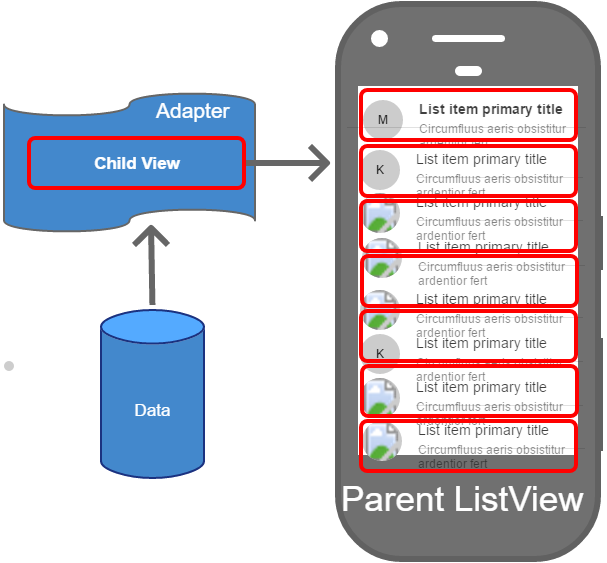
If you could notice, the term “Adapter” came up when referring to a “**AutoCompleteTextView”** above. Before going in to coding and design part with “**AutoCompleteTextView”**, it is important to understand the “**Adapter**” s available in Android and their purpose.

In technical terms;

* An adapter is an object of a class that implements the Adapter interface.
  + It acts as a link between a data set and an adapter view,
* It is an object of a class that extends the abstract “**AdapterView**” class.
* The data set can be anything that presents data in a structured manner.
  + Arrays, List objects, and Cursor objects are commonly used data sets.

In simple Terms;

* **Adapter** is a **bridge** between **UI component** and **Data Source.**
  + Helps to fill data in UI component.
* Holds the data and send the data to an Adapter view.
  + then view takes data from the adapter view and shows on different views like as ListView, GridView, Spinner, “**AutoCompleteTextView”,** etc.
* For more customization in Views base adapter or custom adapters can be used.



UI Component, can be a **ListView**, **Spinner**, **AutoCompleteTextView**, etc. View Item s are displayed here

Adapter; Acts as a bridge, and convert **data item** to **view item**

Data source; can be ArrayList, HashMap, SQLite DB, etc.

### Why Adapters

Adapter views can display large data sets very efficiently. For instance, the **ListView** and **GridView** widgets can display millions of items without any noticeable lag while keeping memory and CPU usage very low. Different adapter views follow different strategies. Most of them are as below,

* They render only those View objects that are either already on-screen or that are about to move on-screen.
  + This way, the memory consumed by an adapter view can be constant and independent of the size of the data set.
* They also allow developers to minimize expensive layout inflate operations and recycle existing View objects that have move off-screen.
  + This keeps CPU consumption low.

### Commonly used Adapters in Android

* **BaseAdapter** – It is parent adapter for all other adapters
* **ArrayAdapter** – It is used whenever we have a list of single items which is backed by an array
* **SimpleCursorAdapter** - The SimpleCursorAdapter links the data contained in a **Cursor** to an Adapter View
  + Binds the Cursor data to an Adapter View.
  + Can define a layout that controls how each row of data is displayed.
  + Each row’s view is populated using the column values of the corresponding row in the cursor.
  + This layout is then displayed in the Adapter View, like a **ListView** for example.
* **CursorAdapter** - A CursorAdapter links a Cursor’s data to a List View.
  + Must include the database’s \_id column as it’s used in processing the list item’s selection.
  + The SimpleCursorAdapter is a subclass of CursorAdapter.
  + The SimpleCursorAdapter is easier to use while the CursorAdapter requires more work but allows more customization.
* **SimpleAdapter** – It is an easy adapter to map static data to views defined in the XML file
* **ListAdapter** - links the data and a **ListView** displaying the data.
  + The List View can display any data type provided it’s wrapped in a ListAdapter.

### Cursors

A cursor is a set of data. Usually a cursor can be obtained when doing a database query. The result of the query is contained in the cursor.

* The basic purpose of a cursor is to point to a single row of the result fetched by the query.
* Load the row pointed by the cursor object.
* By using cursor, can save lot of ram and memory.

In the next section of this tutorial, further use of “Cursor” will be demonstrated

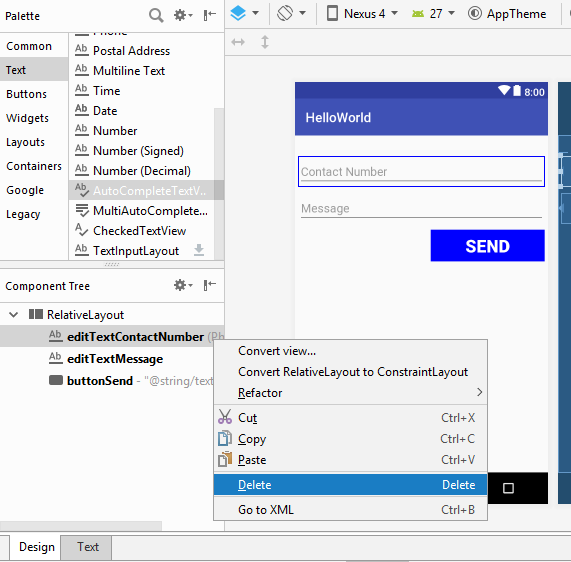
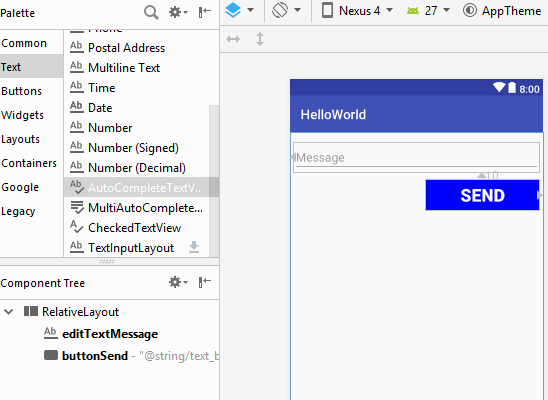
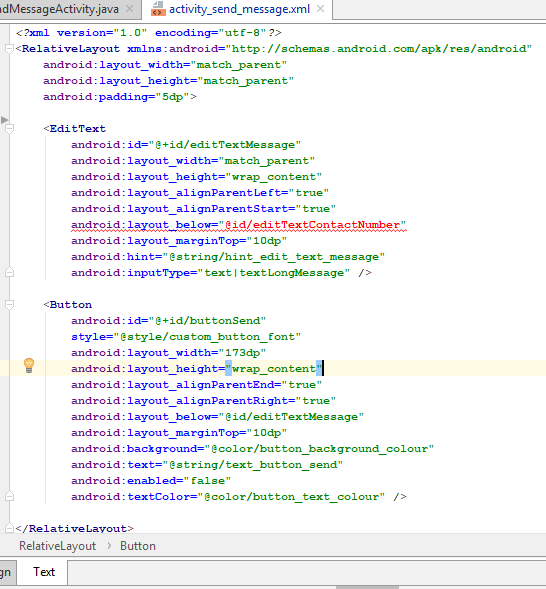
With the current theoretical understand of an “**AutoCompleteTextView**” and “**Adapter**”, lets load the contact list of the device in to a **AutoCompleteTextView** in our “**SendMessageActivity**” and “**activity\_send\_message.xml**”.

## User Interface Design – Modification of **activity\_send\_message.xml**

Regarding the user interface (layout.xml) file, one change is required to the current design. That is to replace the “**EditText**” view with id “**editTextContactNumber**” from “**activity\_send\_message.xml**” layout file and add an “**AutoCompleteTextView**” view in its position with id “**autoCompleteTextContactNumber**”.

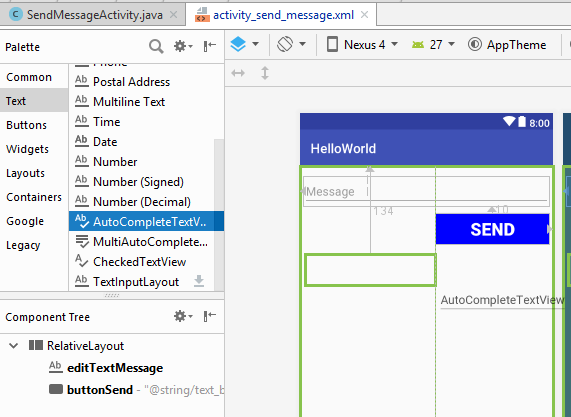
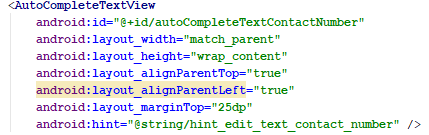
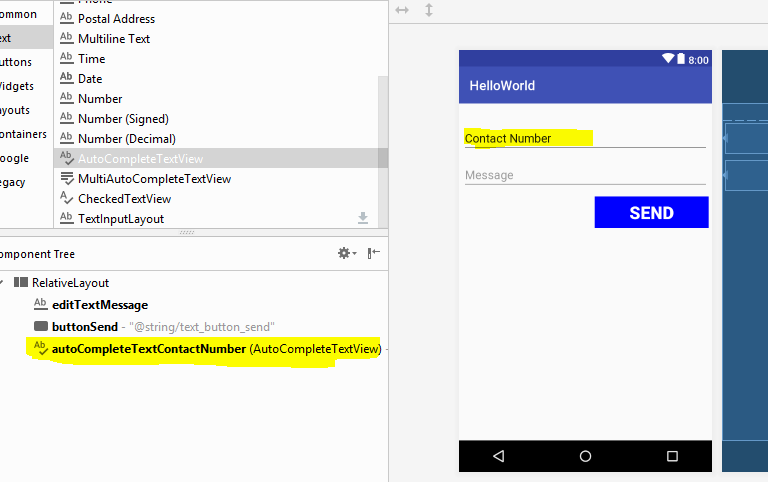
### Remove the EditText from Layout

To remove the “**EditText**” view with id “**editTextContactNumber**” from “**activity\_send\_message.xml**” layout;

1. Open “**activity\_send\_message.xml**” in res/layout and go to “Design” view
2. On “**Component Tree**” view, right click on “**editTextContactNumber**”, and click “**Delete**”
3. After deleting, the layout will look like below;  
   
4. Go to “Text” view of “**activity\_send\_message.xml**”, it will look like below;  
   
   1. Notice the error in layout, it is because the id of the deleted “EditText” is still used. (Ignore it for now)

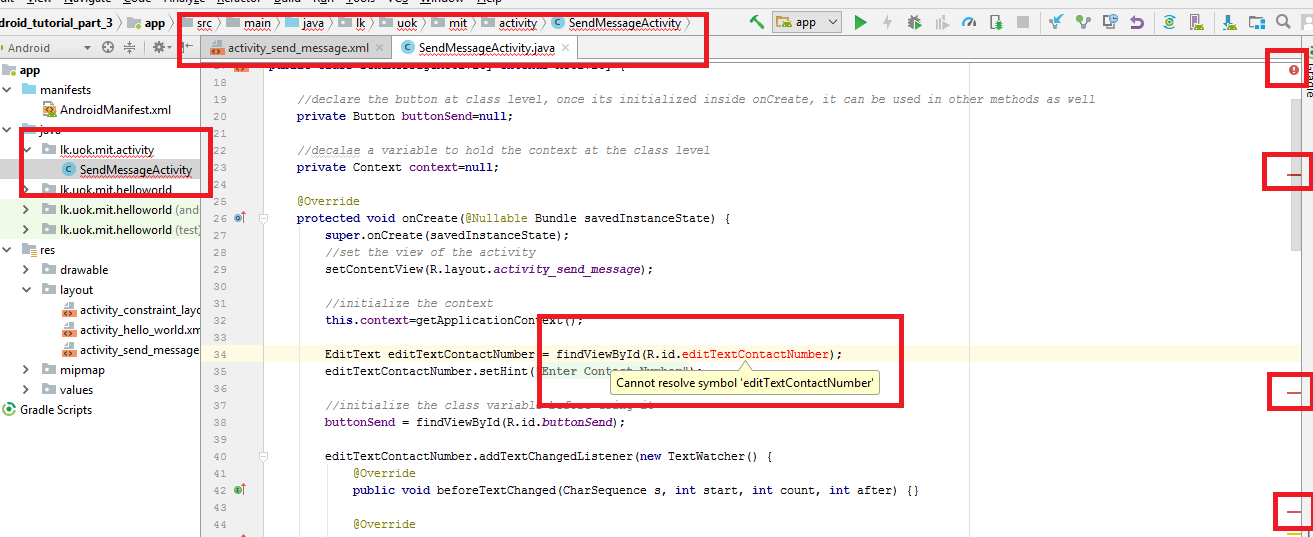
### Add a AutoCompleteTextView to Layout

To add an “**AutoCompleteTextView**” view with id “autoCompleteTextContactNumber” to “**activity\_send\_message.xml**”, and position it in the same place as deleted “EditText” view’s;

1. Open “**activity\_send\_message.xml**” in res/layout and go to “Design” view
2. Drag a “**AutoCompleteTextView**” under the category “**Text**” from “**Palette**” and drop in to any place on “**activity\_send\_message.xml**” layout file  
   
   1. Lets do the positioning by editing XML code this time.
3. To position the dragged “**AutoCompleteTextView**”, go to the “Text” view of “**activity\_send\_message.xml**”, it should look like below (apart for the positioning attributes)  
   
4. Change below attributes in the XML file to **position** the UI component properly
   1. **android:layout\_alignParentTop="true"** - aligns the view to the top of parent layout (view group)
   2. **android:layout\_alignParentLeft="true"**- aligns the view to the left of parent layout (view group)
   3. **android:layout\_marginTop="25dp"** – keeps a margin of 25 dp (density) units from top of the parent layout
5. Change below attributes in the XML file to **resize** the UI component properly
   1. **android:layout\_width="match\_parent"** – makes the width as same as the parent layout
   2. **android:layout\_height="wrap\_content"** – makes the height based on the content of the view
6. Give the requested id to the view
   1. **android:id="@+id/autoCompleteTextContactNumber"** – gives the id **“autoCompleteTextContactNumber”** to the “**AutoCompleteTextView**”
7. Change the reference to **“editTextContactNumber”** in **“editTextMessage”** like below;
   1. **android:layout\_below="@id/autoCompleteTextContactNumber"** – position the “EditText” view for “Message” with id “**editTextMessage**” under the new view
   2. **Note:-** the above property should be changed in “**editTextMessage**” EditText, in order to fix the error mentioned above
8. Change the hint of the view
   1. **android:hint="@string/hint\_edit\_text\_contact\_number"** – give the created string resource in previous tutorial as the text of the new view
9. Now the XML code of “**AutoCompleteTextView**” in “**activity\_send\_message.xml**” should look like below;  
   
10. The design view of “**activity\_send\_message.xml**” should look like below;  
    
11. With this, the UI design part is completed. Next,
    1. Any references to “**editTextContactNumber**” in Java code should be removed (Already there will are compile errors in Activity class)
    2. The contact list from phone should be loaded
    3. the adapter should be created,
    4. adapter should be bind in to the “**autoCompleteTextContactNumber**”

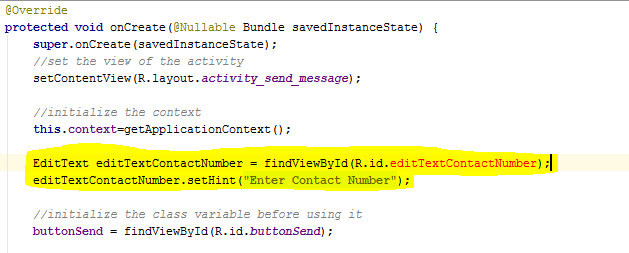
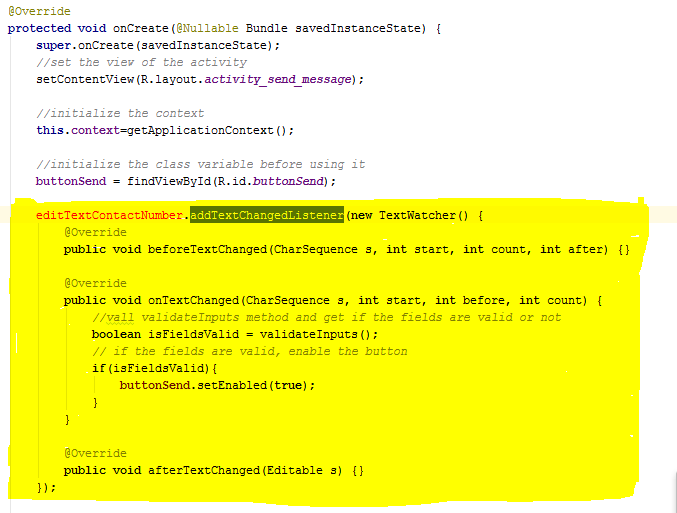
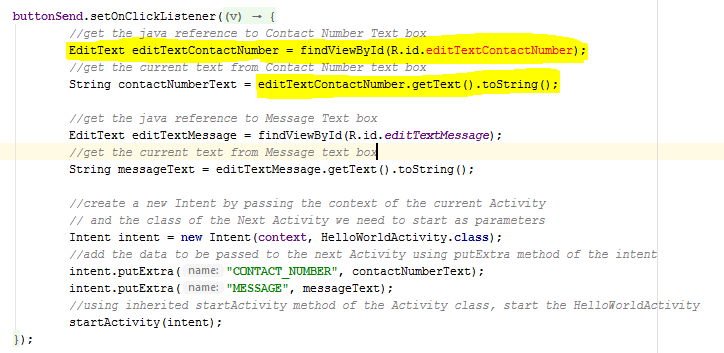
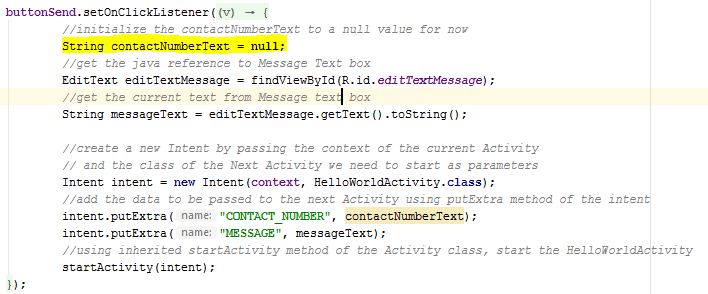
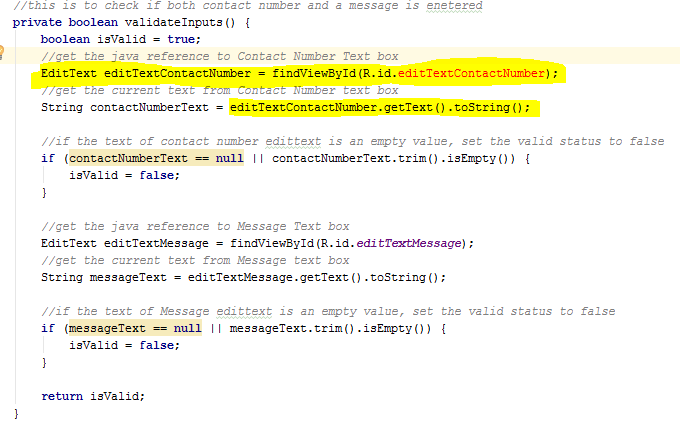
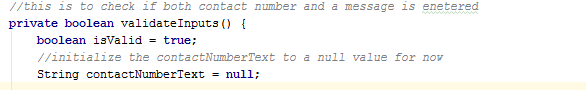
## Java Code Modification – **SendMessageActivity.java**

When referring to the “**SendMessageActivity.java**” class in java folder, there will be multiple compile errors occurred due to the removal of EditText with id “**editTextContactNumber**” from the “activity\_send\_message.xml” like shown below;



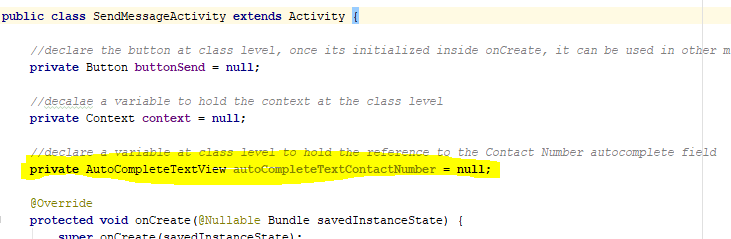
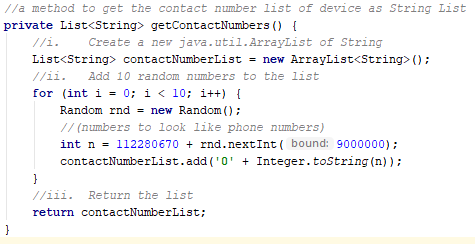
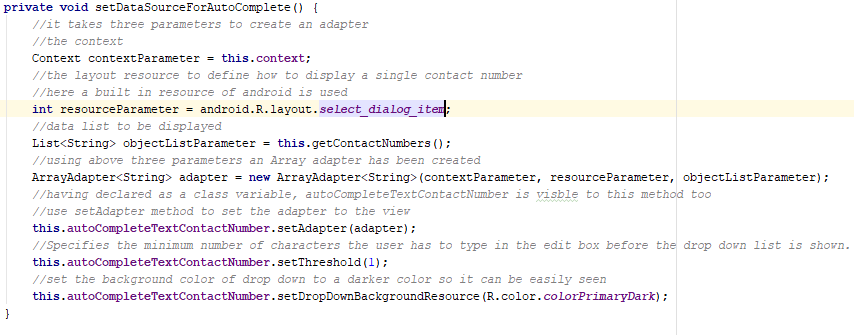
### Remove all the references to “**editTextContactNumber**” from java code

When doing you will have to remove the below code snippets.

1. Inside “**onCreate**” method, delete the reference taken in to a variable named “**editTextContactNumber**” of type “**EditText**” and also delete the line in which the “hint” was set  
   
   1. Remove the highlighted rows as shown above
2. Inside “**onCreate**” method, delete the “**addTextChangedListener**” method of the deleted “**editTextContactNumber**”  
   
   1. Remove the highlighted code snippet above (the whole block in yellow colour)
3. Inside “**onCreate**” method, inside the “**setOnClickListener**” of the “**buttonSend**”, remove the reference, and remove getText() method of “**editTextContactNumber**”  
   
   1. Initialize the “contactNumberText” to a null value for now.  
      
4. Inside “**validateInputs**” method, remove the reference to “**editTextContactNumber**”, and remove getText() method of “**editTextContactNumber**”.  
     
   1. As same as in step 3, Initialize the “contactNumberText” to a null value for now.  
      
5. With all references to the “**editTextContactNumber**” removed from the Activity class, it should be free of any compile errors now.

### Access AutoCompleteTextView from Java

After removing the old EditText view for Contact Number from layout xml and all its references from java code, now its time to see how to access the newly added “**AutoCompleteTextView**” with the id “**autoCompleteTextContactNumber**” from java code.

1. Create a class level variable to hold the reference to the “**AutoCompleteTextView**” with the id “**autoCompleteTextContactNumber**”
   1. Variable Name:- **autoCompleteTextContactNumberView**
   2. Variable Type **:- AutoCompleteTextView** (android.widget.AutoCompleteTextView)  
      
2. Inside “onCreate” method, get the reference to the “**autoCompleteTextContactNumber**” and assign it to the created variable above  
   
3. **Write a method to retrieve a list of contact numbers (list of strings) to be used as the data source for the initialized autocomplete text view**
   1. Method Name :- **getContactNumbers**
   2. Method Return Type:- **java.util.List** of **String** values
   3. Method Parameters:- **None**
   4. Method Access Modifier:- **private**
   5. Method Logic:-
      1. Create a new **java.util.ArrayList** of **String**
      2. Add 10 random numbers to the list (numbers to look like phone numbers)
      3. Return the list
   6. In the next step, we can replace these random numbers from the actual contact numbers from device’s contact list
   7. The method’s full syntax will look like below;  
      
4. As the next step, write a method to set the data source to the autocomplete text view
   1. Method Name :- **setDataSourceForAutoComplete**
   2. Method Return Type:- **void**
   3. Method Parameters:- **None**
   4. Method Access Modifier:- **private**
   5. Method Logic:-
      1. Create a new **android.widget.ArrayAdapter** of type **String**
      2. Set the newly created adapter to the “**autoCompleteTextContactNumber**”
   6. The method’s syntax will look like below  
      
      1. it takes three parameters to create an adapter
         1. the context
            1. ***Context contextParameter = this.context;***

Use the context initialized in “onCreate” method

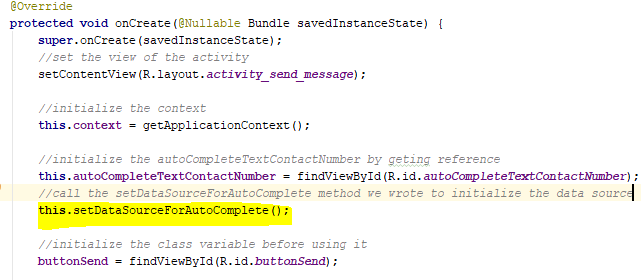
* + - 1. the layout resource to define how to display a single contact number
         1. ***int resourceParameter = android.R.layout.select\_dialog\_item;***

here a built in resource of android is used

* + - 1. data list to be displayed
         1. ***List<String> objectListParameter = this.getContactNumbers();***

Call the method we wrote in above step 3 to get data list

* + 1. using above three parameters an Array adapter has been created
       1. ***ArrayAdapter<String> adapter = new ArrayAdapter<String>(contextParameter, resourceParameter, objectListParameter);***
          1. having declared as a class variable, “**autoCompleteTextContactNumber**” is visible to this method too
    2. use “**setAdapter**” method to set the adapter to the view
       1. ***this.autoCompleteTextContactNumber.setAdapter(adapter);***
    3. use “**setThreshold**” to specify the minimum number of characters the user has to type in the edit box before the drop down list is shown.
       1. ***this.autoCompleteTextContactNumber.setThreshold(1);***
       2. We have set it to 1, so the dropdown will immediately show up
    4. Use “**setDropDownBackgroundResource**” method to set a darker colour to the drop down area of the autocomplete
       1. ***this.autoCompleteTextContactNumber.setDropDownBackgroundResource(R.color.colorPrimaryDark);***

1. Now we have to call the “**setDataSourceForAutoComplete**” from “onCreate” method to initialize and bind the data set to autocomplete text view upon UI initialization  
     
   

1. Now let’s run the application and check the output

|  |  |
| --- | --- |
| Initial State | Typing on AutoCompleteTextView |
|  |  |

## App Permissions in Android

The purpose of a permission is to protect the privacy of an Android user. Android apps must request permission to access sensitive user data (such as contacts and SMS), as well as certain system features (such as camera and internet). Depending on the feature, the system might grant the permission automatically or might prompt the user to approve the request

An app must publicize the permissions it requires by including **<uses-permission>** tags in the app manifest.

### Protection levels

Permissions are divided into several protection levels. The protection level affects whether runtime permission requests are required.

There are three protection levels that affect third-party apps:

* Normal permissions
* Signature permissions
* Dangerous permissions

#### Normal Permissions

Normal permissions cover areas where your app needs to access data or resources outside the app's sandbox, but where there's very little risk to the user's privacy or the operation of other apps.   
E.g.:- permission to set the time zone is a normal permission.

* If an app declares in its manifest that it needs a normal permission, the system automatically grants the app that permission at install time.
* The system doesn't prompt the user to grant normal permissions, and users cannot revoke these permissions.

As of Android 8.1 (API level 27), the following permissions are classified as **PROTECTION\_NORMAL**:

|  |
| --- |
| ACCESS\_LOCATION\_EXTRA\_COMMANDS |
| ACCESS\_NETWORK\_STATE |
| ACCESS\_NOTIFICATION\_POLICY |
| ACCESS\_WIFI\_STATE |
| BLUETOOTH |
| BLUETOOTH\_ADMIN |
| BROADCAST\_STICKY |
| CHANGE\_NETWORK\_STATE |
| CHANGE\_WIFI\_MULTICAST\_STATE |
| CHANGE\_WIFI\_STATE |
| DISABLE\_KEYGUARD |
| EXPAND\_STATUS\_BAR |
| GET\_PACKAGE\_SIZE |
| INSTALL\_SHORTCUT |
| INTERNET |
| KILL\_BACKGROUND\_PROCESSES |
| MANAGE\_OWN\_CALLS |
| MODIFY\_AUDIO\_SETTINGS |
| NFC |
| READ\_SYNC\_SETTINGS |
| READ\_SYNC\_STATS |
| RECEIVE\_BOOT\_COMPLETED |
| REORDER\_TASKS |
| REQUEST\_COMPANION\_RUN\_IN\_BACKGROUND |
| REQUEST\_COMPANION\_USE\_DATA\_IN\_BACKGROUND |
| REQUEST\_DELETE\_PACKAGES |
| REQUEST\_IGNORE\_BATTERY\_OPTIMIZATIONS |
| SET\_ALARM |
| SET\_WALLPAPER |
| SET\_WALLPAPER\_HINTS |
| TRANSMIT\_IR |
| USE\_FINGERPRINT |
| VIBRATE |
| WAKE\_LOCK |
| WRITE\_SYNC\_SETTINGS |

#### Signature permissions

The system grants these app permissions at install time, but only when the app that attempts to use a permission is signed by the same certificate as the app that defines the permission.

As of Android 8.1 (API level 27), the following permissions that third-party apps can use are classified as **PROTECTION\_SIGNATURE**:

|  |
| --- |
| BIND\_ACCESSIBILITY\_SERVICE |
| BIND\_AUTOFILL\_SERVICE |
| BIND\_CARRIER\_SERVICES |
| BIND\_CHOOSER\_TARGET\_SERVICE |
| BIND\_CONDITION\_PROVIDER\_SERVICE |
| BIND\_DEVICE\_ADMIN |
| BIND\_DREAM\_SERVICE |
| BIND\_INCALL\_SERVICE |
| BIND\_INPUT\_METHOD |
| BIND\_MIDI\_DEVICE\_SERVICE |
| BIND\_NFC\_SERVICE |
| BIND\_NOTIFICATION\_LISTENER\_SERVICE |
| BIND\_PRINT\_SERVICE |
| BIND\_SCREENING\_SERVICE |
| BIND\_TELECOM\_CONNECTION\_SERVICE |
| BIND\_TEXT\_SERVICE |
| BIND\_TV\_INPUT |
| BIND\_VISUAL\_VOICEMAIL\_SERVICE |
| BIND\_VOICE\_INTERACTION |
| BIND\_VPN\_SERVICE |
| BIND\_VR\_LISTENER\_SERVICE |
| BIND\_WALLPAPER |
| CLEAR\_APP\_CACHE |
| MANAGE\_DOCUMENTS |
| READ\_VOICEMAIL |
| REQUEST\_INSTALL\_PACKAGES |
| SYSTEM\_ALERT\_WINDOW |
| WRITE\_SETTINGS |
| WRITE\_VOICEMAIL |

* If your app needs a dangerous permission, you must check whether you have that permission every time you perform an operation that requires that permission.
* Beginning with Android 6.0 (API level 23), users can revoke permissions from any app at any time, even if the app targets a lower API level.
* So even if the app used the camera yesterday, it can't assume it still has that permission today.
* To check if you have a permission, call the **ContextCompat.checkSelfPermission()** method.
* **For example, this snippet shows how to check if the activity has permission to write to the calendar:**

#### Dangerous permissions

Dangerous permissions cover areas where the app wants data or resources that involve the user's private information, or could potentially affect the user's stored data or the operation of other apps.   
E.g.:- **the ability to read the user's contacts is a dangerous permission.**

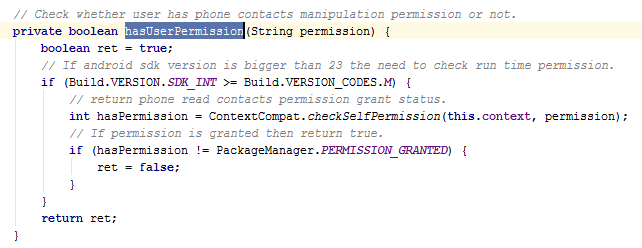
* If an app declares that it needs a dangerous permission, the user has to explicitly grant the permission to the app.
* Until the user approves the permission, your app cannot provide functionality that depends on that permission.
* To use a dangerous permission, your app must prompt the user to grant permission at runtime.

A list of dangerous permissions and permission groups are listed below;

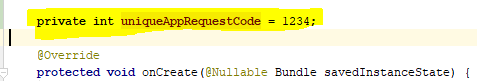
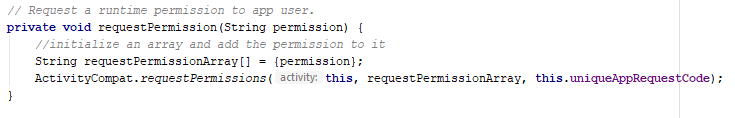
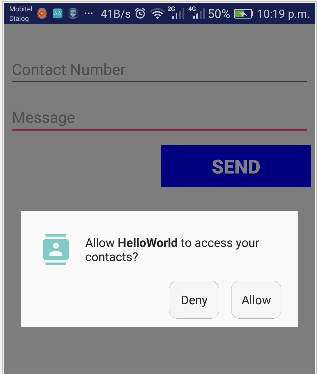
|  |  |
| --- | --- |
| Permission Group | Permissions |
| CALENDAR | READ\_CALENDAR |
| WRITE\_CALENDAR |
| CAMERA | CAMERA |
| CONTACTS | READ\_CONTACTS |
| WRITE\_CONTACTS |
| GET\_ACCOUNTS |
| LOCATION | ACCESS\_FINE\_LOCATION |
| ACCESS\_COARSE\_LOCATION |
| MICROPHONE | RECORD\_AUDIO |
| PHONE | READ\_PHONE\_STATE |
| READ\_PHONE\_NUMBERS |
| CALL\_PHONE |
| ANSWER\_PHONE\_CALLS |
| READ\_CALL\_LOG |
| WRITE\_CALL\_LOG |
| ADD\_VOICEMAIL |
| USE\_SIP |
| PROCESS\_OUTGOING\_CALLS |
| SENSORS | BODY\_SENSORS |
| SMS | SEND\_SMS |
| RECEIVE\_SMS |
| READ\_SMS |
| RECEIVE\_WAP\_PUSH |
| RECEIVE\_MMS |
| STORAGE | READ\_EXTERNAL\_STORAGE |
| WRITE\_EXTERNAL\_STORAGE |

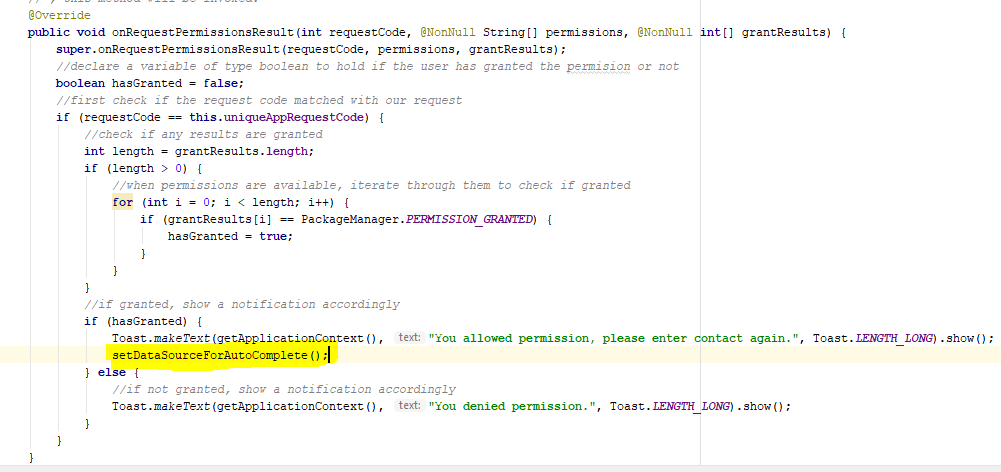
## Load Contact List from The device

As the next step, let’s load the contact list from device and replace the randomly generated numbers of our code.

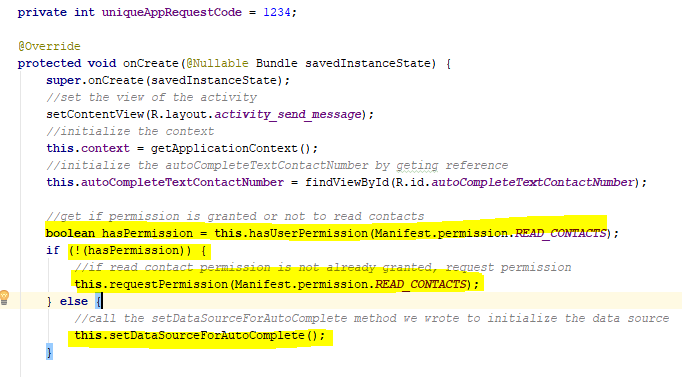
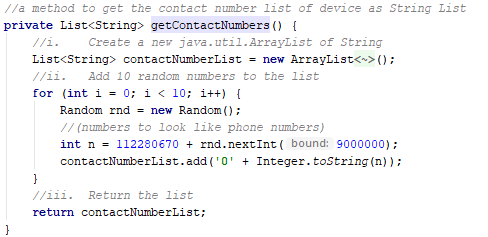
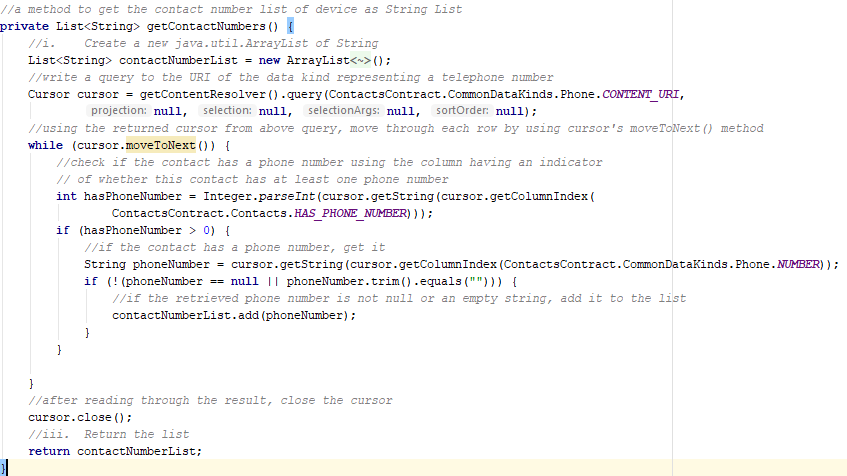
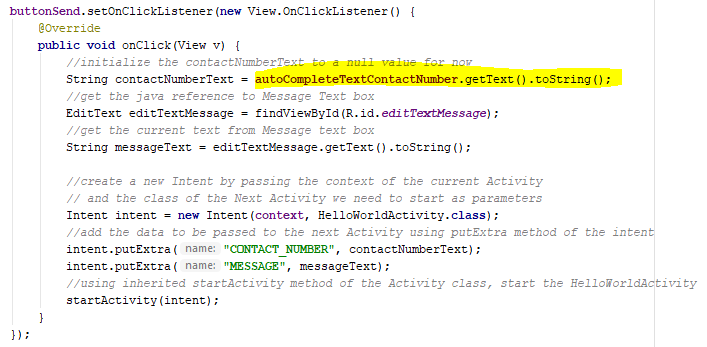
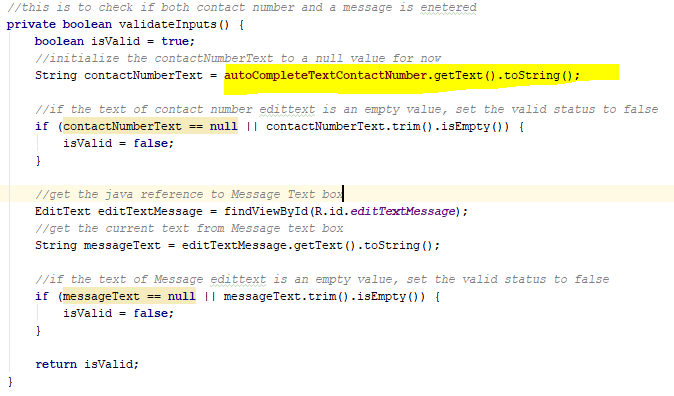
1. First we need to add app permission in to the app’s manifest file to read contact list.
   1. Open “**AndroidManifest.xml**” inside “**manifests**” folder
   2. add the following line to read contact before Application tag (“<application ..”)  
      ***<uses-permission android:name="android.permission.READ\_CONTACTS" />***  
      
2. The app permission we are going to use is categorized as a “Dangerous permission”, as it is mentioned above, “**To use a dangerous permission, the app must prompt the user to grant permission at runtime**”.   
   Therefore as the next step, let’s write a method to check if user’s permission is provided.
   1. Method Name :- **hasUserPermission ()**
   2. Method Return Type:- **Boolean --> (if permission is provided or not)**
   3. Method Parameters:- **String permission --> (permission name)**
   4. Method Access Modifier:- **private**
   5. Method Logic:-
      1. Check If android sdk version is bigger than API level 23,if yes then as mentioned in next step, need to check run time permission. ***if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.M) {***
         1. Build.VERSION.SDK\_INT - The user-visible SDK version of the framework
         2. Build.VERSION\_CODES.M - android version codes - **M** is for Marshmallow
      2. Check if the passed permission to the method as a parameter is included in the app’s permission list;  
         ***int hasPermission = ContextCompat.checkSelfPermission(this.context, permission);***
         1. android.support.v4.content.**ContextCompat** – is a support class to access variables and status from context
      3. Check if the return state from above method is **GRANTED** status or not  
         ***if (hasPermission != PackageManager.PERMISSION\_GRANTED) {***
      4. If it is not in GRANTED state, return false, else return true
   6. The method’s syntax will look like below  
      
   7. **NOTE**:- there is a static method available in “**android.support.v4.app.ActivityCompat**” class (a helper class for Activity) for the same purpose as the method we just wrote, named as “**shouldShowRequestPermissionRationale**”;  
        
      ***ActivityCompat.shouldShowRequestPermissionRationale(SendMessageActivity.this, Manifest.permission.READ\_CONTACTS)*** 
      1. We could replace the “**hasUserPermission**” method with above code, but for better understanding of the logic, let’s continue with our own method.
3. If the permission is not available to read contacts, that is if “false” is returned from above method, we should ask the user for permissions.   
   Therefore as the next step, let’s write a method to check if user’s permission is provided.
   1. Method Name :- **requestPermission ()**
   2. Method Return Type:- **void**
   3. Method Parameters:- **String permission --> (permission name)**
   4. Method Access Modifier:- **private**
   5. Method Logic:-
      1. Call the “**requestPermissions**” method in “**android.support.v4.app.ActivityCompat**”

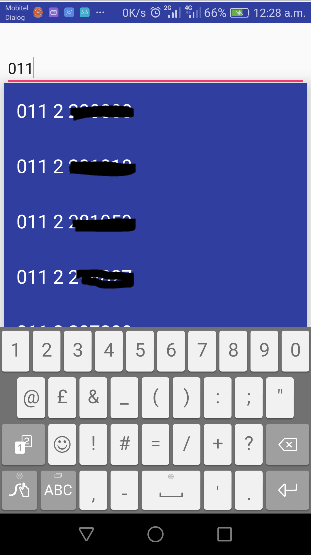
Parameters:

* + - 1. **activity** - The target activity.
      2. **permissions** - The requested permissions. Must me non-null and not empty.
      3. **requestCode** - Application specific request code to match with a result reported to (declare a variable in class level for this as “**uniqueAppRequestCode**”)  
         
    1. This method requires a permission array as a parameter, initialize it
  1. The method’s syntax will look like below  
     
  2. The result of the above method will be a dialog to ask for permission like shown below;  
     
  3. The above dialog is shown by android OS, and there should be a way for our app to get the result from dialog – that is to know whether user allowed or denied the permission
     1. For this, a method is available from “Activity” class for us through inheritance named as “**onRequestPermissionsResult**”
     2. By overriding this method we can get if user granted or denied the request, the complete overriding is shown below;



* + 1. Notice the highlighted line in above figure; we skipped setting the data source if permission is not given in “onCreate” method, therefore, after granting permission, we should set the data source to autocomplete again.

1. Now we have written two separate methods regarding app permissions to READ\_CONTACTS.
   1. **boolean hasUserPermission(String permission)** 
      1. to check if the app has the permission passed as the String parameter
   2. **void requestPermission(String permission)**
      1. to request the permission (passed as a string parameter) from user
2. It’s time to call the above two methods inside “onCreate” method, as shown below;  
   
   1. Note the highlighted lines,
      1. first check for permissions using the “**hasUserPermission**” method passing “**Manifest.permission.READ\_CONTACTS**” as the parameter.
         1. **Manifest.permission.READ\_CONTACTS –** is a constant declared for the ease of developers to represent “READ\_CONTACTS” permission
      2. If the permission is not already granted, ask for the same permission using “**requestPermission**” we wrote
      3. In the else part, if permission available set the data source for the autocomplete view
3. As the final step, lets load the contact list from device, for this lets modify the “**getContactNumbers**” method we wrote above, below is how it currently looks  
   
   1. We created random values as contact numbers to start with, **let’s replace them with actuals**.
4. The modified method to get actual contact list from should look like below;  
   
   1. The only parts kept from earlier version of the method are the very first line and last line of the method, that is to initialize an empty array and to return that array;
      1. First write a query to a URI which contain contact data  
         ***Cursor cursor = getContentResolver().query(ContactsContract.CommonDataKinds.Phone.CONTENT\_URI, null, null, null, null);***
         1. The result will be a cursor referring to the result set’s each row
      2. Next, using the returned cursor from above query, move through each row by using cursor's **moveToNext()** method  
         ***while (cursor.moveToNext()) {}***
         1. This while loop will continue till all the rows of the resulting dataset is read
      3. Each row (cursor) will contain a set of columns, therefore a contact dataset row will have below columns;
         1. **ContactsContract.Contacts.HAS\_PHONE\_NUMBER**
            1. a column to indicate if this contact has a phone number
         2. **ContactsContract.CommonDataKinds.Phone.NUMBER**
            1. A column with the actual phone number
         3. **ContactsContract.CommonDataKinds.Phone.DISPLAY\_NAME**
            1. A column with the contact name
      4. Using values in each above columns,
         1. first check if the contact has a phone number,   
            ***int hasPhoneNumber = Integer.parseInt(cursor.getString(cursor.getColumnIndex(ContactsContract.Contacts.HAS\_PHONE\_NUMBER)));***
         2. and if a phone number exists, retrieve it  
            ***String phoneNumber = cursor.getString(cursor.getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER));***
         3. check if its null or an empty string, and if it’s a valid phone number, add to the list  
            ***if (!(phoneNumber == null || phoneNumber.trim().equals(""))) {  
             contactNumberList.add(phoneNumber);  
            }***
5. There are two more things to be donefor the app to work properly; if you could recall, while removing the old “EditText” for contact number, we removed the text value retrieving part from it and initialized the string value to null.   
   Since we have the autocomplete text view to replace it, lets retrieve text from it, in locations as shown below;
   1. Inside “**onCreate**” method, in the “**SEND**” buttons “**setOnClickListener**” method;  
      
   2. Inside “**validateInputs**” method;  
      

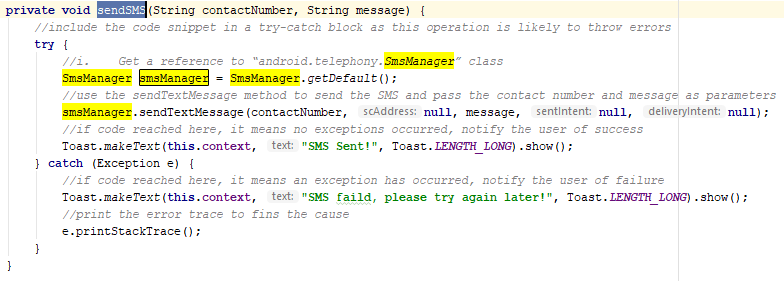
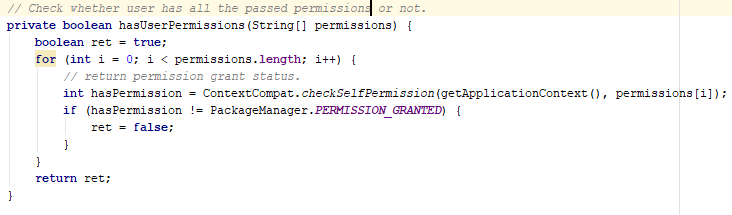
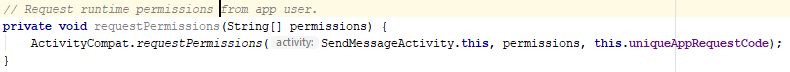
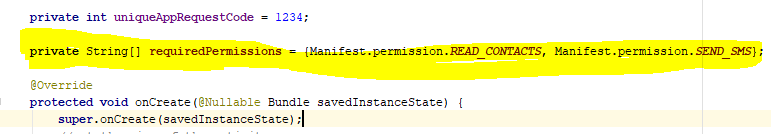
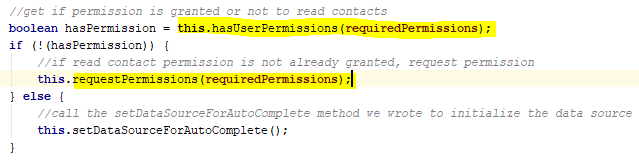
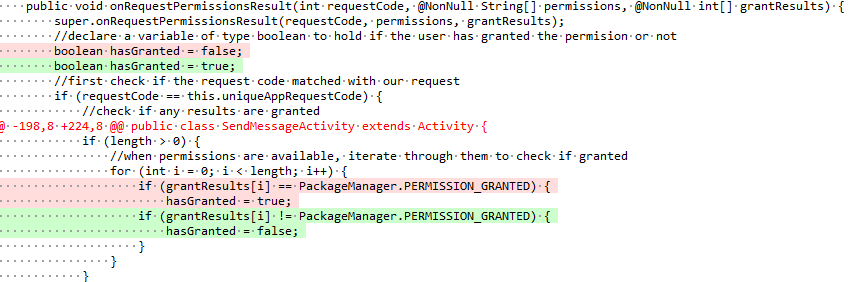
1. Now save everything and run the app and check; instead the auto generated numbers you will see actual numbers in your contact list in the drop down list of the autocomplete like shown below;  
   

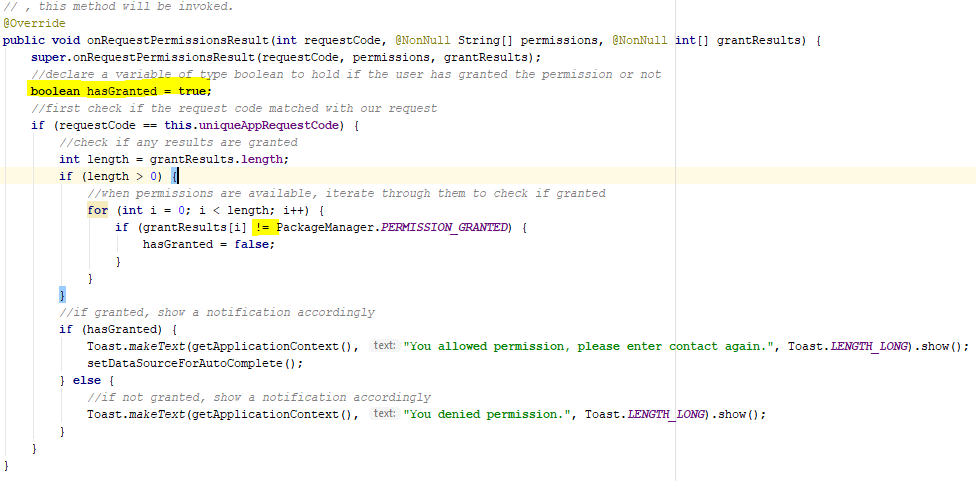
# Send a SMS in Android

After loading the actual contact list, lets see how to send a SMS from our app. If you could recall, previously, upon clicking on “SEND” button, we loaded a new UI by passing the selected contact number and message to that UI to display.

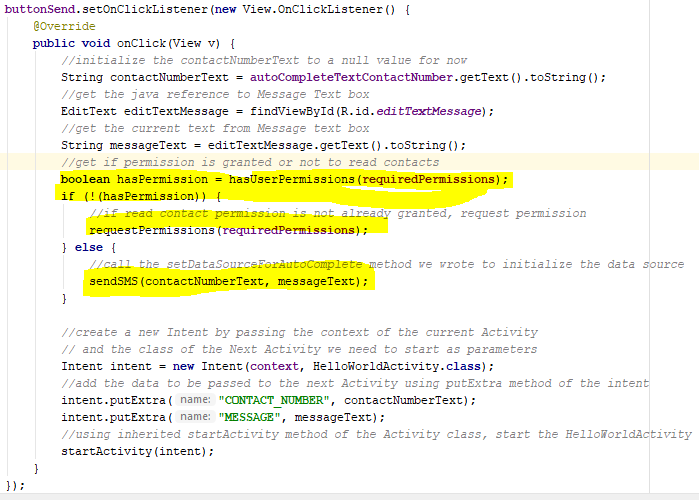
Instead let’s try to send the SMS now;

1. If you could refer to the permissions tables above “SMS” category is included in the dangerous permission section. Therefore we need to include the permission in to the manifest file below;
   1. Open **“AndroidManifest.xml”** inside “manifests” folder
   2. add the following line to read contact before Application tag (“<application ..”)  
      **<uses-permission android:name="android.permission.SEND\_SMS " />**

1. Next lets write a method to send the SMS in the “**SendMessageActivity**” class
   1. Method Name :- **sendSMS ()**
   2. Method Return Type:- **void**
   3. Method Parameters:-
      1. **String contactNumber** --> (the text from autoCompleteTextContactNumber input)
      2. **String message** --> (the text from editTextMessage input)
   4. Method Access Modifier:- **private**
   5. Method Logic:-
      1. Put the entire code inside the method in a try catch block
      2. Get a reference to “**android.telephony.SmsManager**” class like below;  
         ***SmsManager smsManager = SmsManager.getDefault();***
      3. Use the “**sendTextMessage**” method to send the SMS and pass the contact number and message as parameters  
         ***smsManager.sendTextMessage(contactNumber, null, message, null, null);***
      4. if code execution reached line after, it means no exceptions occurred, notify the user of success  
         ***Toast.makeText(this.context, "SMS Sent!", Toast.LENGTH\_LONG).show()***;
      5. Inside catch clause; if code execution reach inside catch, it means an exception has occurred, notify the user  
         ***Toast.makeText(this.context, "SMS faild, please try again later!", Toast.LENGTH\_LONG).show();***
      6. Write to the logger of the exception details for debugging purposes  
         ***e.printStackTrace();***
   6. The whole code block will look like below;  
      
2. If you could recall, we wrote two separate methods regarding app permissions to “**READ\_CONTACTS**” above. Now we need to change them both to support multiple permissions since we need “**SEND\_SMS**” permission as well
   1. **boolean hasUserPermission(String permission)**this method should change like below;
      1. Change the method name to – **hasUserPermissions**
      2. Change the parameters to - **String[] permissions**
      3. Iterate over the passed in array of permissions and check if all has permitted  
         ***for (int i = 0; i < permissions.length; i++) {***
      4. If either any one of them is not permitted, return false, else return true  
         ***int hasPermission = ContextCompat.checkSelfPermission(getApplicationContext(), permissions[i]);  
          if (hasPermission != PackageManager.PERMISSION\_GRANTED) {  
          ret = false;  
          }***
      5. The new code block will look like below;  
         
   2. **void requestPermission(String permission)**
      1. Change the method name to – **requestPermissions**
      2. Change the parameters to - **String[] permissions**
      3. Now remove the initialization of the string array, we are already passing an array of strings
      4. The new code block will look like below;  
         
3. It’s time to change the calling parts of the above two methods inside “onCreate” method, as shown below;
   1. First initialize a class level variable named “**requiredPermissions**” of type “String[]” with two permissions we need for the app as shown below;  
      ***private String[] requiredPermissions = {Manifest.permission.READ\_CONTACTS, Manifest.permission.SEND\_SMS};***  
      
   2. Now change the method names in calling part and pass the above declared parameter to the calling place of the two method
      1. Inside “onCreate” method;  
         
4. Now change the code inside “**onRequestPermissionsResult**” as shown below;  
   
   1. New code should be like in “green” colur, since we need to check for multiple permissions now, if any of them are not granted, our app cannot work properly

The complete code looks like below now; It’s a simple change of logic  


1. Now its time to call the “**sendSMS**” method within the onclick listener of the “SEND” button, that is upon click of it, the message will be send.



Source code for this tutorial part can be found in Git Repository given below: - <https://github.com/nadee158/android_tutorial_part_3.git>

# References

*Android AutoCompleteTextView Example*. (2018, January 1). Retrieved from JavaTPoint: https://www.javatpoint.com/android-autocompletetextview-example

*Application Fundamentals*. (2018, May 2018). Retrieved from Android Developers: https://developer.android.com

CHUGH, A. (2018, April 2). *Android AutoCompleteTextView Example Tutorial*. Retrieved from JournalDev: https://www.journaldev.com/9574/android-autocompletetextview-example-tutorial

MULLIS, A. (2017, November 11). *Android Studio tutorial for beginners*. Retrieved from Android Authority: https://www.androidauthority.com/android-studio-tutorial-beginners-637572/

SINICKI, A. (2017, December 16). *Android SDK tutorial for beginners*. Retrieved from Android Authority: https://www.androidauthority.com/android-sdk-tutorial-beginners-634376/

Vogel, L. (2016, June 20). *Getting started with Android development - Tutorial*. Retrieved from vogella: http://www.vogella.com/tutorials/Android/article.html#high-level-overview-of-android-development