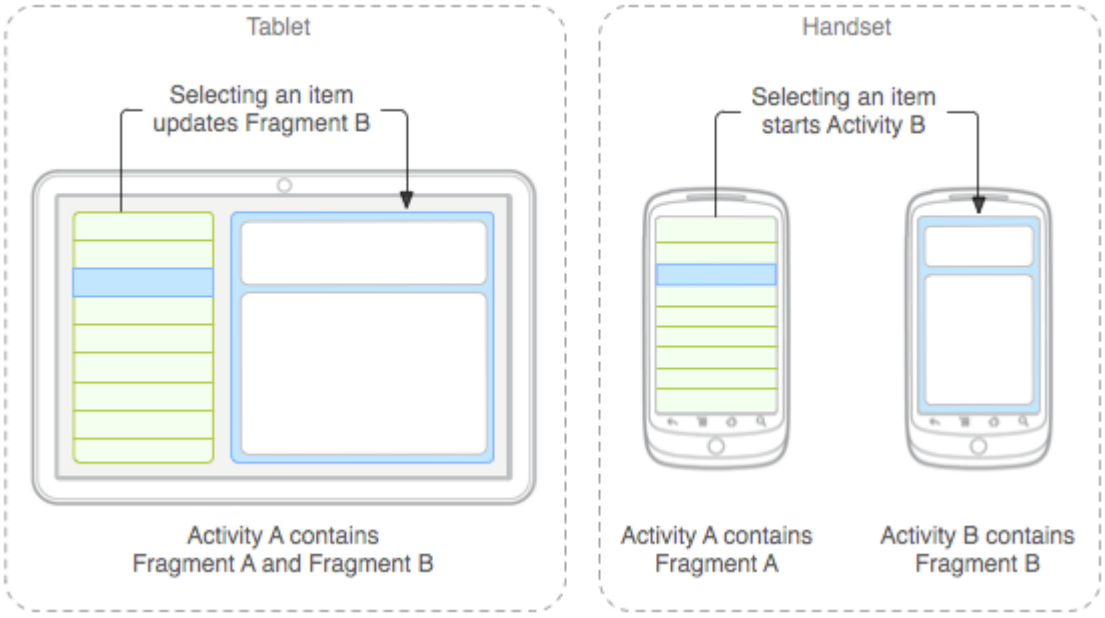
Fragments

This lecture discusses Android **Fragments**. A Fragment is "a behavior or a *portion* of user interface in Activity." You can think of them as "mini-activities" or "sub-activities". Fragments are designed to be **reusable** and **composable**, so you can mix and match them within a single screen of a user interface. While XML resource provide reusable and composable *views*, Fragments provide reusable and composable *controllers*. Fragments allow us to make re-usable pieces of Activities that can have their own layouts, data models, event callbacks, etc.

This lecture references code found at https://github.com/info448-s17/lecture05-fragments. Note that this code builds upon the example developed in Lecture 4.

Fragments were introduced in API 11 (Honeycomb), which provided the first "tablet" version of Android. Fragments were designed to provide a UI component that would allow for side-by-side activity displays appropriate to larger screens.



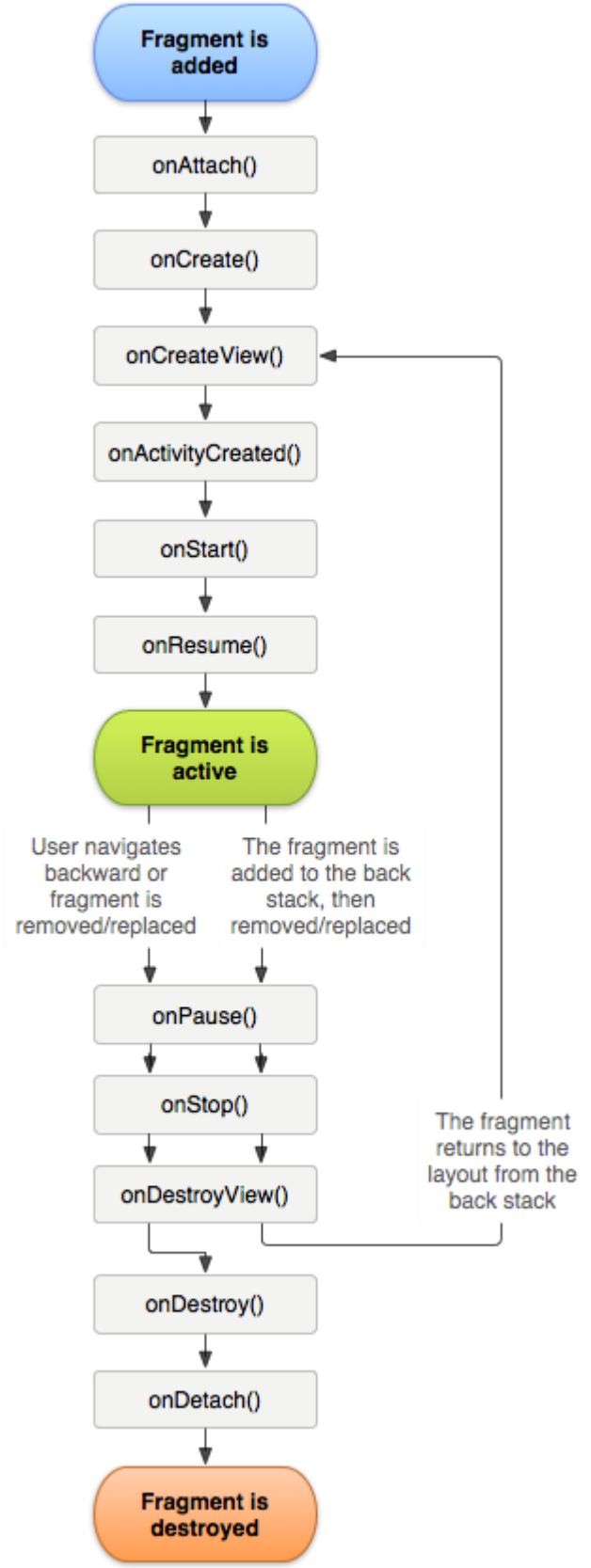
Fragment example, from Google[17]

Instead of needing to navigate between two related views (particularly for this "master and detail" setup), the user can see both views within the same Activity… but those "views" could also be easily split between two Activities for smaller screens, because their required *controller logic* has been isolated into a Fragment.

Fragments are intended to be **modular**, **reusable** components. They should ***not*** depend on the Activity they are inside, so that you can be flexible about when and where they are displayed!

Although Fragments are like "mini-Activities", they are *always* embedded inside an `Activity`; they cannot exist independently. While it's possible to have

Fragments that are not visible or that don’t have a UI, they still are part of an Activity. Because of this, a Fragment’s lifecycle is directly tied to its containing Activity’s lifecycle. (e.g., if the Activity is paused, the Fragment is too. If the Activity is destroyed, the Fragment is too). However, Fragments also have their own lifecycle with corresponding lifecycle callbacks functions.



Fragment lifecycle state diagram, from Google[18]

The Fragment lifecycle is very similar to the Activity lifecycle, with a a couple of additional steps:

- `onAttach()`: called when the Fragment is first associated with ("added to") an Activity, and thus gains a **Context**. This callback is generally used for initializing communication between the Fragment and its Activity.
  This callback is mirrored by `onDetach()`, for when the Fragment is removed from an Activity.
- `onCreateView()`: called when the View (the user interface) is about to be drawn. This callback is used to establish any details dependent on the View (including adding event listeners, etc).
  Note that code intializing data models, or anything that needs to be *persisted* across configuration changes, should instead be done in the `onCreate()` callback. `onCreate()` is not called if the fragment is *retained* (see below).
  This callback is mirrored by `onDestroyView()`, for when the Fragment's UI View hierarchy is being removed from the screen.
- `onActivityCreated()`: called when the *containing Activity's* `onCreate()` method has returned, and thus indicates that the Activity is fully created. This is useful for *retained* Fragments.
  This callback has no mirror!

5.1 Creating a Fragment

In order to illustrate how to make a Fragment, we will **refactor** the `MainActivity` to use Fragments for displaying the list of movies. This will help to illustrate the relationship between Activities and Fragments.
To create a Fragment, you subclass the `Fragment` class. Let's make one called `MovieFragment` (in the `MovieFragment.java` file). You can use Android Studio to do this work: via the `File > New > Fragment > Fragment (blank)` menu option. (**DO NOT** select any of the other options for in the wizard for now; they provide template code that can distract from the core principles).
There are two versions of the `Fragment` class: one in the framework's `android.app` package and one in the `android.support.v4` package. The later package refers to the Support Library. These are libraries of classes designed to make Android applications *backwards compatible*: for example, `Fragment` and its related classes came out in API 11 so aren't in the `android.app` package for earlier devices. By including the support library, we can include those classes as well!

- Support libraries *also* include additional convenience and helper classes that are not part of the core Android package. These include interface elements (e.g., `ConstraintLayout`, `RecyclerView`, or `ViewPager`) and accessibility classes. See the features list for details. Thus it is often useful to include and utilize support library versions of classes so that you don't need to "roll your own" versions of these convenience classes.
- The main disadvantage to using support libraries is that they need to be included in your application, so will make the final `.apk` file larger (and may potentially require workarounds for method count limitations). You will also run into problems if you try and mix and match versions of the classes (e.g., from different versions of the support library). But as always, you should *avoid premature optimization*.

Thus in this course you should **default** to using the support library version of a class when given a choice!
After we've created the `MovieFragment` Java file, we'll want to specify a layout for that Fragment (so it can be shown on the screen). As part of using the New Fragment Wizard we were provided with a `fragment_movie` layout that we can use.

- Since we want the Movie list to live in that Fragment, we can move (copy) the View definitions from `activity_main` into `fragment_movie`.
- We will then adjust `activity_main` so that it instead contains an empty `FrameLayout`. This will act as a simple **"container"** for our Fragment (similar to an empty `<div>` in HTML). *Be sure to give it an `id` so we can refer to it later!*.
  It is possible to include the Fragment directly through the XML, using the XML to instantiate the Fragment (the same way that we have the XML instantiate Buttons). We do this by specifying a `<fragment>` element, with a `android:name` attribute assigned a reference to the `Fragment` class:

```
<fragment
   android:id="@+id/frag_movie"
     android:name="edu.uw.fragmentdemo.MovieFragment"
     android:layout_width="match_parent"
     android:layout_height="match_parent"/>
```

Defining the Fragment in the XML works (and will be fine to start with), but in practice it is *much* more worthwhile to instantiate the Fragemnts **dynamically** at runtime in the Java code—thereby allowing the Fragments to be dynamically determined and changed. We will start with the XML version to built the Fragment, and then shift to the Java version.

We can next begin filling in the Java logic for the Fragment. Android Studio provides a little starter code: a constructor and the `onCreateView()` callback—the later is more relevant since we will use that to set up the layout (similar to in the `onCreate()` function of `MainActivity`). But the `MainActivity#onCreate()` method specifies a layout by calling `setContentView()` and pasing a resource id. With Fragments, we can't just "set" the View because the Fragment *belongs to* an Activity, and so will exist *inside* its View hierarchy! Instead, we need to figure out which ViewGroup the Fragment is inside of, and then **inflate** the Fragment inside that View.
This "inflated" View is referred to as the ***root view***: it is the "root" of the Fragment's View tree (the View that all the Views inside the Fragment's layout will be attached to). We access the root view by *inflating* the fragment's layout, and saving a reference to the inflated View:

```
View rootView = inflater.inflate(R.layout.fragment_layout, container, false);
```

- Note that the `inflater` object we are calling `inflate()` on is passed as a parameter to the `onCreateView()` callback. The parameters to the `inflate()` method are: the layout to inflate, the `ViewGroup` (`container`) into which the layout should be inflated (also passed as a parameter to the callback), and whether ot not to "attach" the inflated layout to the container (`false` in this case because the Fragment system already handles the attachment, so the inflate method doesn't need to).

The `onCreateView()` callback must `return` the inflated *root view*, so that the system can perform this attachment.
With the Fragment's layout defined, we can start moving functionality from the Activity into the Fragment.

- The the background `ASyncTask` can be moved over directly, so that it belongs to the Fragment instead of the Activity.
- The `adapter` declaration will need to be moved as well.
- The UI setup (including initializing the Adapter) will be moved from the Activity's `onCreate()` to the Fragment's `onCreateView()`. However, you will need to make a few changes during this refactoring:
  - The `findViewById()` method is a method of the `Activity` class, and thus can't be called on an implicit `this` inside the Fragment. Instead, the method can be called on the **root view**, searching just that View and its children.
  - The Adapter's constructor requires a `Context` as its first parameter; while an `Activity` is a `Context`, a `Fragment` is not—Fragments operate in the Context of their containing Activity! Fragments can refer to the Activity that they are inside (and the `Context` it represents) by using the `getActivity()` method. Note that this method is used *primarily* for getting a reference to a `Context`, not for arbitrart commuication withe Activity (see below for details)

5.1.1 Activity-to-Fragment Communication

The example code intentionally has left the *input controls* (the search field and button) in the Activity, rather than making them part of the Fragment. Apart from being a useful demonstration, this allows the Fragment to have a single purpose (showing the list of movies) and would let us change the search UI independent of the displayed results. But since the the button is in the Activity but the downloading functionality is in the Fragment, we need a way for the Activity to "talk" to the Fragment. We thus need a reference to the contained Fragment—access to the XML similar to that provided by `findViewById`.
We can get a reference to a contained Fragment from an Activity by using a `FragmentManager`. This is an object responsible for (ahem) managing Fragment. It allows us to "look up" Fragments, as well as to manipulate which Fragments are shown. We access this FragmentManager by calling the `getSupportFragmentManager()` method on the Activity, and then can use `findFragmentById()` to look up an XML-defined Fragment by its `id`:

```
//MovieFragment example
MovieFragment fragment =
(MovieFragment)getSupportFragmentManager().findFragmentById(R.id.fragment);
```

- Note that we're using a method to explicit access the **support** `FragmentManager`. The Activity class (API level 15+) is able to work with both the platform and support `FragmentManager` classes. But because these classes don't have a shared `interface`, the Activity needs to provide different Java methods which can return the correct type.
Once you have a reference to the Fragment, this acts just like an other object—you can call any `public` methods it has! For example, if you give the Fragment a

public method (e.g., `searchMovies()`), then this method can be called from the Activity:

```
//called from Activity on the referenced fragment

fragment.searchMovies(searchTerm)
```

(The parameter to this public method allows the Activity to provide information to the Fragment!)

At this point, the program should be able to be executed… and continue to function in exactly the same way! The program has just been refactored, so that all the movie downloading and listing work is **encapsulated** inside a Fragment that can be used in different Activities.

- In effect, we've created our own "widget" that can be included in any other screen, such as if we always wanted the list of movies to be available alongside some other user interface components.

5.2 Dynamic Fragments

The real benefit from encapsulating behavior in a Fragment is to be able to support multiple Fragments within a single Activity. For example, in the the archetypal "master/detail" navgiation flow, one screen (Fragment) holds the "master" (list) and another screen (Fragment) holds details about a particular item. This is a very common navigation pattern for Android apps, and can be seen in most email or news apps.

- On large screens, Fragments allow these two screens to be placed side by side!

In this section, we will continue to refine the Movie app so that when the user clicks on a Movie in the list, the app shows a screen (Fragment) with details about the selected movie.

5.2.1 Instantiating Fragments

To do this, we will need to instantiate the Fragments dynamically (in Java code), rather than statically in the XML using the `<fragment>` element. This is because we need to be able to dynamically change which Fragment is currently being shown, which is not possibly for Fragments that are "hard-coded" in the XML.
Unlike Activities, Fragments (such as `MovieFragment`) **do** have constructor methods that can be called—in fact, Android *requires* that every Fragment include a default (no-argument) constructor that is called when Fragments are created by the system. While we are able to call this constructor, it is considered best practice to **not** call this constructor directly when you want to instantiate a Fragment, and to in fact leave the method empty. This is because we do not have full control over when the constructor is executed: the Android system may call the no-argument constructor whenever it needs to recreate the Activity (or just the Fragment), which can happen at arbitrary times. Since only this default constructor is called, we can't add an additional constructor with any arguments we may want the Fragment to have (e.g., the `searchTerm`)… and thus it's best to not use it at all.

Instead, we specify a **simple factory** method (by convention called `newInstance()`) which is able to "create" an instance of the Fragment for us. This factory method can take as many arguments as we want, and then does the work of passing these arguments into the Fragment instantiated with the default constructor:

```
public static MyFragment newInstance(String argument) {

    MyFragment fragment = new MyFragment(); //instantiate the Fragment

    Bundle args = new Bundle(); //an (empty) Bundle for the arguments

    args.putString(ARG_PARAM_KEY, argument); //add the argument to the Bundle

    fragment.setArguments(args); //add the Bundle to the Fragment

    return fragment; //return the Fragment

}
```

In order to pass the arguments into the new Fragment, we wrap them up in a `Bundle` (an object containing basic *key-value pairs*). Values can be added to a `Bundle` using an appropriate `putType()` method; note that these do need to be primative types (`int`, `String`, etc.). The `Bundle` of arguments can then be assignment to the Fragment by calling the `setArguments()` method.

- We will be able to access this `Bundle` from inside the Fragment (e.g., in the `onCreateView()` callback) by using the `getArguments()` method (and `getType()` to retrieve the values from it). This allows us to dynamically adjust the content of the Fragment's Views! For example, we can run the `downloadMovieData()` function using this argument, fetching movie results as soon as the Fragment is created (e.g., on a button press).
- Since the `Bundle` is a set of *key-value* pairs, each value needs to have a particular key. These keys are usually defined as private constants (e.g., `ARG_PARAM_KEY` in the above example) to make storage and retrieval easier.

We will then be able to instantiate the Fragment (e.g., in the Activity class), passing it any arguments we wish:

```
MyFragment fragment = MyFragment.newInstance("My Argument");
```

5.2.2 Transactions

Once we've instantiated a Fragment in the Java, we need to attach it to the view hierarchy: since we're no longer using the XML `<fragment>` element, we need some other way to load the Fragment into the `<FrameLayout>` container.

We do this loading using a **`FragmentTransaction`**[19]. A transaction represents a *change* in the Fragment that is being displayed. You can think of this like a bank (or database) transaction: they allow you to add or remove Fragments like we would add or remove money from a bank account. We instantiate new transactions representing the change we wish to make, and then "run" that transaction in order to apply the change.

To create a transaction, we utilize the `FragmentManager` again; the `FragmentManager#beginTransaction()` method is used to instantiate a **new** `FragmentTransaction`.

Transactions represent a set of Fragment changes that are all "applied" at the same time (similar to depositing and withdrawing money from multiple accounts all at once). We specify these transactions using by calling the `add()`, `remove()`, or `.replace()` methods on the `FragmentTransaction`.

- The `add()` method lets you specify which View **container** you want to add a particular Fragment to. The `remove()` method lets you remove a Fragment you have a reference to. The `replace()` method removes any Fragments in a container and then adds the specified Fragment instead.
- Each of these methods returns the modified `FragmentTransaction`, so they can be "chained" together.
  Finally, we call the `commit()` method on the transaction in order to "submit" it and have all of the changes go into effect.
  We can do this work in the Activity's search click handler to add a Fragment, rather than specifying the Fragment in the XML:

```
FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
//params: container to add to, Fragment to add, (optional) tag
transaction.add(R.id.container, myFragment, MOVIE_LIST_FRAGMENT_TAG);
transaction.commit();
```

- The third argument for the `add()` method is a "tag" we apply to the Fragment beign added. This gives it a name that we can use to find a reference to this Fragment later if we want (via `FragmentManager#findFragmentByTag(tag)`). Alternatively, we can save a reference to the Fragment as an instance variable; this is faster but more memory intensive (and can cause possible leaks, since the reference keeps the Fragment from being reclaimed by the system).

5.2.3 Inter-Fragment Communication

We can use this structure for instantiating and loading (via transactions) a **second Fragment** (e.g., a "detail" view for a selected Movie). We can add functionality (e.g., in the `onClick()` handler) so that when the user clicks on a movie in the list, we **`replace()`** the currently displayed Fragment with this new detailed Fragment. However, remember that Fragments are supposed to be **modular**—each Fragment should be *self-contained*, and not know about any other Fragments that may exist (after all, what if we wanted the master/detail views to be side-by-side on a large screen?)

> Using `getActivity()` to reference the Activity and `getSupportFragmentManager()` to access the manager is a violation of the Law of Demeter—don't do it!

Instead, we have Fragments communicate by passing messages through their contained Activity: the `MovieFragment` should tell its Activity that a particular movie has been selected, and then that Activity can determine what to do about it (e.g., creating a `DetailFragment` to dispaly that information).
The recommended way to provide Fragment-to-Activity communication is to define an **interface**. The Fragment class should specify an `interface` (for one or more public methods) that its containing Activity *must* support—and since the Fragment can only exist within an Activity that implements that interface, it knows the Activity has the specified public methods that it can call to pass information to that Activity.
As an example of this process:

- Create a new `interface` inside the Fragment (e.g., `OnMovieSelectedListener`). This interface needs a public method (e.g., `onMovieSelected(Movie movie)`) that the Fragment can call to give instructions or messages to the Activity.
- In the Fragment's `onAttach()` callback (called when the Fragment is first associated with an Activity), we can check that the Activity actually implements the interface by trying to *cast* it to that interface. We can also save a reference to this Activity for later:

```
public void onAttach(Context context) {
    super.onAttach(context);

    try {
        callback = (OnMovieSelectedListener)context;
    } catch (ClassCastException e) {
        throw new ClassCastException(context.toString() + " must implement
OnMovieSelectedListener");
    }
}
```

- Then when an action occurs in the Fragment (e.g., a movie is selected), you call the interface's method on the `callback` reference.
- Finally, you will need to make sure that the Activity `implements` this callback. Remember that a class can implement multiple interfaces!
  In the Activity's implementation of the interface, you can handle the information provided. For example, use the `FragmentManager` to create a `replace()` transaction to load a new `DetailFragment` for the appropriate data.
  In the end, this will allow you to have one Fragment cause the application to switch to another!

> This is not the only way for Fragments to communicate. It is also possible to have a Fragment send an `Intent` to the Activity, who then responds to that as appropriate. But using the Intent system is more resource-intensive than using interfaces.

5.2.4 The Back Stack

But what happens when we hit the "back" button? The Activity exits! *Why?* Because "back" normally says to "leave the Activity"—we only had one Activity, just multiple fragments.

Recall that the Android system may have lots of Activities (even across multiple apps!) with the user moving back and forth between them. As described in lecture2, each new Activity is associated with a "task" and placed on a **stack**[20]. When the "back" button is pressed, that Activity is popped off the stack, and the user is taken to the Activity that is now at the top.

Fragments by default are not part of this "back-stack", since they are just components of Activities. However, you *can* specify that a transaction should include the Fragment change as part of the stack navigation by

calling `FragmentTransaction#addToBackStack()` as part of your transaction (e.g., right before you `commit()`):

```
getSupportFragmentManager().beginTransaction()
                           .add(detailFragment, "detail")
                           // Add this transaction to the back stack
                           .addToBackStack()
                           .commit();
```

Note that the “back” button will cause *the entire transaction* to “reverse”. Thus if you performed a `remove()` then an `add()` (e.g., via a `replace()`), then hitting “back” will cause the the previously added Fragment to be removed *and* the previously removed Fragment to be added.

- `FragmentManager` also includes numerous methods for manually manipulating the back-stack (e.g., “popping” off transactions) if necessary.

Android View Classes

Updated on March 22, 2018

by Neeraj Agarwal

In this tutorial I’ve given brief explanation about Andriod View Class. **View Class** are the basic building block for user interface components. A View occupies a 2-dimensional area (say: rectangle) on the screen, which is responsible for framing and handling different type of events.

In our previous blog posts, we have learned Android installation steps and made first program called Hello world (built in XML code). The program was created to show text output in the AVD.

Views are used to create input and output fields in the an Android App. It can be input text field, radio field, image field etc. They are same as, input text field, image tag to show images, radio field in HTML.

Most Commonly Used Android View classes:

These views can be used to create a useful input and output fields.

- Text View
- EditText
- Button
- ImageView
- ImageButton
- CheckBox
- Radio button
- RadioGroup
- ListView
- Spinner
- AutoCompleteTextView

View Group Class

The View-Group is the base class for ***layouts*****,** which holds other Views and define their properties. Actually an application comprises combination and nesting of Views-Group and Views Classes.

Text View:

This class is used to display text on the android application screen. It also allows user to optionally edit it. Although it contains text editing operations, the basic class does not allow editing, So Edit Text class is included to do so.

Syntax For Text View In XML Coding Is:

XML coding of text view:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:layout_width="fill_parent"

android:layout_height="fill_parent"
android:orientation="vertical" >



<TextView
```

```
android:id="@+id/myTextview"

android:layout_width="fill_parent"

android:layout_height="wrap_content"

android:text="Mangnet Brains"

android:textSize="25dp"

android:textColor="@android:color/black"

android:typeface="serif"

android:gravity="center"

android:padding="10dp"

android:layout_margin="20dp" />

</LinearLayout>
```

Copy

Edit Text:

This class makes text to be editable in Android application. It helps in building the data interface taken from any user, also contains certain features through which we can hide the data which are confidential.

Syntax For Edit Text In XML Coding Is:

XML coding of edit text:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:layout_width="fill_parent"

android:layout_height="fill_parent"
android:orientation="vertical" >



<EditText
```

```
android:id="@+id/myEdittext"

android:layout_width="fill_parent"

android:layout_height="wrap_content"

android:textSize="20dp"

android:textStyle="bold"

android:typeface="serif"

android:gravity="center"

android:padding="10dp"

android:layout_margin="20dp"

android:hint="Enter a Number"

android:singleLine="true"
android:inputType="textPassword" />


</LinearLayout>
```

Copy

Image view:

Image view helps to display images in an android application. Any image can be selected, we just have to paste our image in a drawable folder from where we can access it. For example: In below Code **“@drawable/ic_laucher”** has been taken.

Syntax For Image View In XML Coding Is:

XML coding of Image view:

```
<LinearLayout

xmlns:android="http://schemas.android.com/apk/res/android"

android:layout_width="fill_parent"

android:layout_height="fill_parent"

android:orientation="vertical"
android:gravity="center_horizontal" >



<ImageView

android:id="@+id/myimageview"

android:layout_width="100dp"

android:layout_height="100dp"
```

```
android:layout_margin="20dp"

android:gravity="center"

android:padding="10dp"
android:src="@drawable/ic_launcher" />
```

Copy

Check Box:

Checkbox is used in that applications where we have to select one option from multiple provided. Checkbox is mainly used when 2 or more options are present.

Syntax For CheckBox In XML Coding Is:

XML coding of checkbox:

```
<LinearLayout
```

```
xmlns:android="http://schemas.android.com/apk/res/android"

android:layout_width="fill_parent"

android:layout_height="fill_parent" >



<CheckBox

android:id="@+id/checkBox1"

android:layout_width="100dp"

android:layout_height="wrap_content"

android:layout_margin="20dp"

android:text="Formget."

android:checked="true" />

<CheckBox

android:id="@+id/checkBox2"

android:layout_width="100dp"

android:layout_height="wrap_content"

android:layout_margin="20dp"

android:text="Mailget." />



</LinearLayout>
```

Copy

Radio Button:

Radio button is like checkbox, but there is slight difference between them. Radio button is a two-states button that can be either checked or unchecked.

Syntax For Radio Button In XML Coding Is:

XML coding of Radio Button:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:layout_width="fill_parent"

android:layout_height="fill_parent"

android:orientation="vertical"
android:gravity="center_horizontal" >



<RadioButton

android:id="@+id/radioButton1"
```

```
android:layout_width="100dp"

android:layout_height="wrap_content"

android:layout_margin="20dp"

android:text="Formget"

android:checked="true" />

<RadioButton

android:id="@+id/radioButton1"

android:layout_width="100dp"

android:layout_height="wrap_content"

android:layout_margin="20dp"

android:text="Mailget" />

</LinearLayout>
```

Copy

Button View:

This class is used to create a button on an application screen. Buttons are very helpful in getting into a content. Android button represents a clickable push-button widget.

Syntax For Button In XML Coding Is:

XML coding of button:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:layout_width="fill_parent"

android:layout_height="fill_parent" >



<Button

android:id="@+id/button1"

android:layout_width="match_parent"

android:layout_height="wrap_content"

android:text="Click Here !" />



</LinearLayout>
```

Copy

Image Button View:

Image button is a button but it carries an image on it. We can put an image or a certain text on it and when we click it provides the operations assigned to it.

Syntax For Image Button In XML Coding Is:

XML coding of Image button:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:layout_width="fill_parent"

android:layout_height="fill_parent"
android:orientation="vertical" >



<ImageButton

android:id="@+id/imageButton1"
```

```
android:layout_width="match_parent"

android:layout_height="wrap_content"

android:layout_gravity="center"
android:src="@drawable/ic_launcher" />


</LinearLayout>
```

Layouts

Part of Android Jetpack.

A layout defines the structure for a user interface in your app, such as in an activity. All elements in the layout are built using a hierarchy of `View` and `ViewGroup` objects. A `View` usually draws something the user can see and interact with. Whereas a `ViewGroup` is an invisible container that defines the layout structure for `View` and other `ViewGroup` objects, as shown in figure 1.

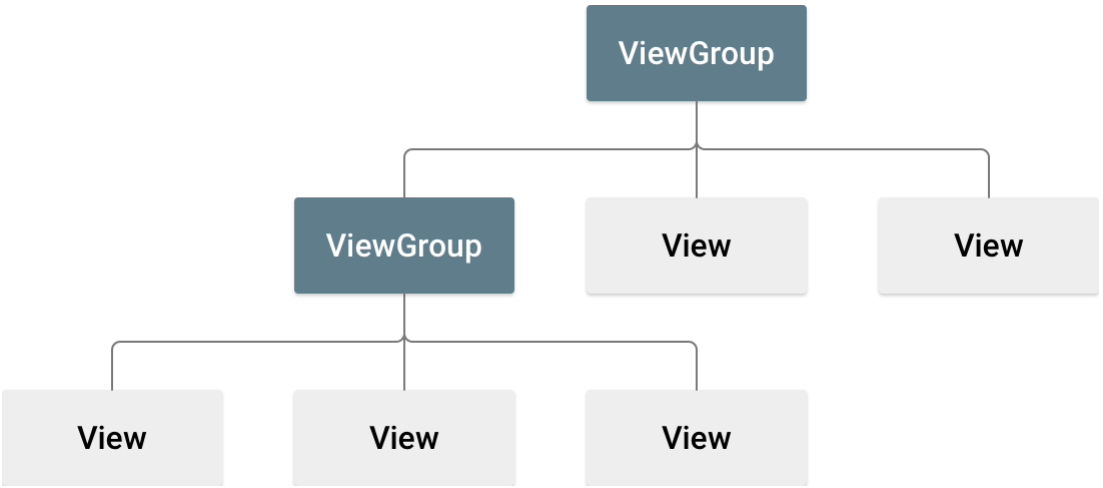


Figure 1. Illustration of a view hierarchy, which defines a UI layout

The `View` objects are usually called "widgets" and can be one of many subclasses, such as `Button` or `TextView`. The `ViewGroup` objects are usually called "layouts" can be one of many types that provide a different layout structure, such as `LinearLayout` or `ConstraintLayout`.

You can declare a layout in two ways:

- **Declare UI elements in XML**. Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts.

  You can also use Android Studio's Layout Editor to build your XML layout using a drag-and-drop interface.

- **Instantiate layout elements at runtime**. Your app can create View and ViewGroup objects (and manipulate their properties) programmatically.

  Declaring your UI in XML allows you to separate the presentation of your app from the code that controls its behavior. Using XML files also makes it easy to provide different layouts for different screen sizes and orientations (discussed further in Supporting Different Screen Sizes).

  The Android framework gives you the flexibility to use either or both of these methods to build your app's UI. For example, you can declare your app's default layouts in XML, and then modify the layout at runtime.

Tip: To debug your layout at runtime, use the Layout Inspector tool.

Write the XML

Using Android's XML vocabulary, you can quickly design UI layouts and the screen elements they contain, in the same way you create web pages in HTML — with a series of nested elements.

Each layout file must contain exactly one root element, which must be a View or ViewGroup object. Once you've defined the root element, you can add additional layout objects or widgets as child elements to gradually build a View hierarchy that defines your layout. For example, here's an XML layout that uses a vertical `LinearLayout` to hold a `TextView` and a `Button`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a Button" />
</LinearLayout>
```

After you've declared your layout in XML, save the file with the `.xml` extension, in your Android project's `res/layout/` directory, so it will properly compile.

More information about the syntax for a layout XML file is available in the Layout Resources document.

Load the XML Resource

When you compile your app, each XML layout file is compiled into a `View` resource. You should load the layout resource from your app code, in your `Activity.onCreate()` callback implementation. Do so by calling `setContentView()`, passing it the reference to your layout resource in the form of: `R.layout.layout_file_name`. For example, if your XML layout is saved as `main_layout.xml`, you would load it for your Activity like so:

KOTLINJAVA

```
fun onCreate(savedInstanceState: Bundle) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.main_layout)
}
```

The `onCreate()` callback method in your Activity is called by the Android framework when your Activity is launched (see the discussion about lifecycles, in the Activities document).

Attributes

Every View and ViewGroup object supports their own variety of XML attributes. Some attributes are specific to a View object (for example, TextView supports the `textSize` attribute), but these attributes are also inherited by any View objects that may extend this class. Some are common to all View objects, because they are inherited from the root View class (like the `id` attribute). And, other attributes are considered "layout parameters," which are attributes that describe certain layout orientations of the View object, as defined by that object's parent ViewGroup object.

ID

Any View object may have an integer ID associated with it, to uniquely identify the View within the tree. When the app is compiled, this ID is referenced as an integer, but the ID is typically assigned in the layout XML file as a string, in the `id` attribute. This is an XML attribute common to all View objects (defined by the `View` class) and you will use it very often. The syntax for an ID, inside an XML tag is:

```
android:id="@+id/my_button"
```

The at-symbol (@) at the beginning of the string indicates that the XML parser should parse and expand the rest of the ID string and identify it as an ID resource. The plus-symbol (+) means that this is a new resource name that must be created and added to our resources (in the `R.java` file). There are a number of other ID resources that are offered by the Android framework. When referencing an Android resource ID, you do not need the plus-symbol, but must add the `android` package namespace, like so:

```
android:id="@android:id/empty"
```

With the `android` package namespace in place, we're now referencing an ID from the `android.R` resources class, rather than the local resources class.

In order to create views and reference them from the app, a common pattern is to:

1. Define a view/widget in the layout file and assign it a unique ID:

```
<Button android:id="@+id/my_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/my_button_text"/>
```

2. Then create an instance of the view object and capture it from the layout (typically in the `onCreate()` method):

KOTLINJAVA

```
val myButton: Button = findViewById(R.id.my_button)
```

Defining IDs for view objects is important when creating a `RelativeLayout`. In a relative layout, sibling views can define their layout relative to another sibling view, which is referenced by the unique ID.

An ID need not be unique throughout the entire tree, but it should be unique within the part of the tree you are searching (which may often be the entire tree, so it's best to be completely unique when possible).

Note: With Android Studio 3.6 and higher, the view binding feature can replace **`findViewById()`** calls and provides compile-time type safety for code that interacts with views. Consider using view binding instead of **`findViewById()`**.

Layout Parameters

XML layout attributes named `layout_`*`something`* define layout parameters for the View that are appropriate for the ViewGroup in which it resides.

Every ViewGroup class implements a nested class that extends `ViewGroup.LayoutParams`. This subclass contains property types that define the size and position for each child view, as appropriate for the view group. As

you can see in figure 2, the parent view group defines layout parameters for each child view (including the child view group).

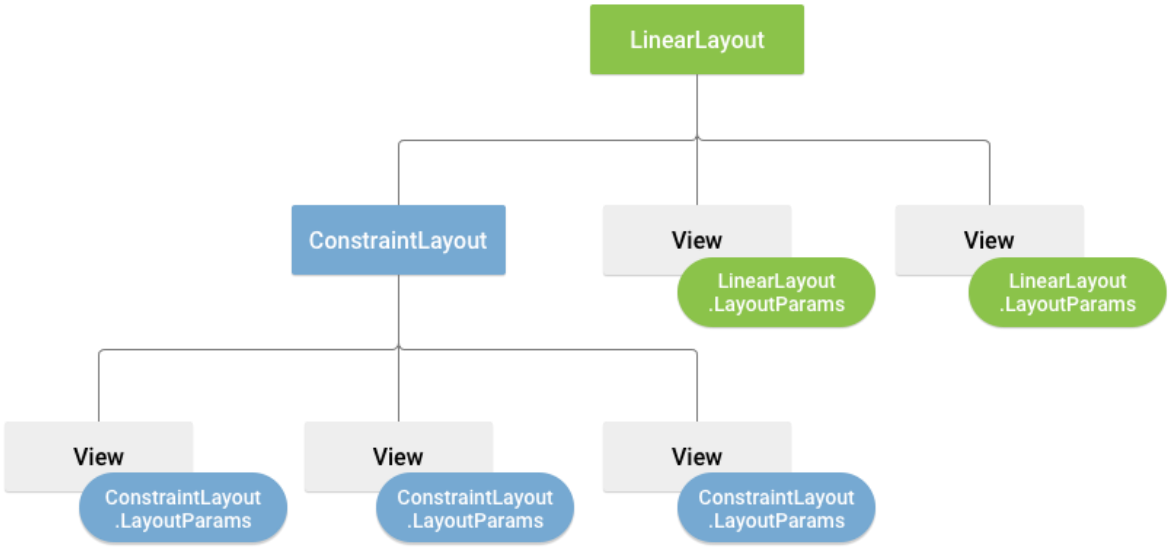


Figure 2. Visualization of a view hierarchy with layout parameters associated with each view

Note that every LayoutParams subclass has its own syntax for setting values. Each child element must define LayoutParams that are appropriate for its parent, though it may also define different LayoutParams for its own children.

All view groups include a width and height (`layout_width` and `layout_height`), and each view is required to define them. Many LayoutParams also include optional margins and borders.

You can specify width and height with exact measurements, though you probably won't want to do this often. More often, you will use one of these constants to set the width or height:

- ***wrap_content*** tells your view to size itself to the dimensions required by its content.
- ***match_parent*** tells your view to become as big as its parent view group will allow.

In general, specifying a layout width and height using absolute units such as pixels is not recommended. Instead, using relative measurements such as density-independent pixel units (***dp***), ***wrap_content***, or ***match_parent***, is a better approach, because it helps ensure that your app will display properly across a variety of device screen sizes. The accepted measurement types are defined in the Available Resources document.

Layout Position

The geometry of a view is that of a rectangle. A view has a location, expressed as a pair of *left* and *top* coordinates, and two dimensions, expressed as a width and a height. The unit for location and dimensions is the pixel.

It is possible to retrieve the location of a view by invoking the methods `getLeft()` and `getTop()`. The former returns the left, or X, coordinate of the rectangle representing the view. The latter returns the top, or Y, coordinate of the rectangle representing the view. These methods both return the location of the view relative to its parent. For instance, when `getLeft()` returns 20, that means the view is located 20 pixels to the right of the left edge of its direct parent.

In addition, several convenience methods are offered to avoid unnecessary computations, namely `getRight()` and `getBottom()`. These methods return the coordinates of the right and bottom edges of the rectangle representing the view. For instance, calling `getRight()` is similar to the following computation: `getLeft() + getWidth()`.

Size, Padding and Margins

The size of a view is expressed with a width and a height. A view actually possesses two pairs of width and height values.

The first pair is known as *measured width* and *measured height*. These dimensions define how big a view wants to be within its parent. The measured dimensions can be obtained by calling `getMeasuredWidth()` and `getMeasuredHeight()`.

The second pair is simply known as *width* and *height*, or sometimes *drawing width* and *drawing height*. These dimensions define the actual size of the view on screen, at drawing time and after layout. These values may, but do not have to, be different from the measured width and height. The width and height can be obtained by calling `getWidth()` and `getHeight()`.

To measure its dimensions, a view takes into account its padding. The padding is expressed in pixels for the left, top, right and bottom parts of the view. Padding can be used to offset the content of the view by a specific number of pixels. For instance, a left padding of 2 will push the view's content by 2 pixels to the right of the left edge. Padding can be set using the `setPadding(int, int, int, int)` method and queried by calling `getPaddingLeft()`, `getPaddingTop()`, `getPaddingRight()` and `getPaddingBottom()`.

Even though a view can define a padding, it does not provide any support for margins. However, view groups provide such a support. Refer to `ViewGroup` and `ViewGroup.MarginLayoutParams` for further information.

For more information about dimensions, see Dimension Values.

Common Layouts

Each subclass of the `ViewGroup` class provides a unique way to display the views you nest within it. Below are some of the more common layout types that are built into the Android platform.

Note: Although you can nest one or more layouts within another layout to achieve your UI design, you should strive to keep your layout hierarchy as shallow as possible. Your layout draws faster if it has fewer nested layouts (a wide view hierarchy is better than a deep view hierarchy).

Linear Layout

A layout that organizes its children into a single horizontal or vertical row. It creates a scrollbar if the length of the window exceeds the length of the screen.

Relative Layout

Enables you to specify the location of child objects relative to each other (child A to the left of child B) or to the parent (aligned to the top of the parent).

Web View

Displays web pages.

Building Layouts with an Adapter

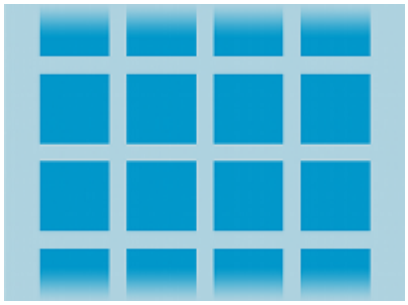
When the content for your layout is dynamic or not pre-determined, you can use a layout that subclasses `AdapterView` to populate the layout with views at runtime. A subclass of the `AdapterView` class uses an `Adapter` to bind data to its layout. The `Adapter` behaves as a middleman between the data source and the `AdapterView` layout—the `Adapter` retrieves the data (from a source such as an array or a database query) and converts each entry into a view that can be added into the `AdapterView` layout.

Common layouts backed by an adapter include:

List View

Displays a scrolling single column list.

Grid View



Displays a scrolling grid of columns and rows.

Filling an adapter view with data

You can populate an `AdapterView` such as `ListView` or `GridView` by binding the `AdapterView` instance to an `Adapter`, which retrieves data from an external source and creates a `View` that represents each data entry.

Android provides several subclasses of `Adapter` that are useful for retrieving different kinds of data and building views for an `AdapterView`. The two most common adapters are:

`ArrayAdapter`

Use this adapter when your data source is an array. By default, `ArrayAdapter` creates a view for each array item by calling `toString()` on each item and placing the contents in a `TextView`.

For example, if you have an array of strings you want to display in a `ListView`, initialize a new `ArrayAdapter` using a constructor to specify the layout for each string and the string array:

KOTLINJAVA

```
val adapter = ArrayAdapter<String>(this,
android.R.layout.simple_list_item_1, myStringArray)
```

The arguments for this constructor are:

- Your app `Context`
- The layout that contains a `TextView` for each string in the array
- The string array

Then simply call `setAdapter()` on your `ListView`:

KOTLINJAVA

```
val listView: ListView = findViewById(R.id.listview)
listView.adapter = adapter
```

To customize the appearance of each item you can override the `toString()` method for the objects in your array. Or, to create a view for each item that's something other than a `TextView` (for example, if you want an `ImageView` for each array item), extend the `ArrayAdapter` class and override `getView()` to return the type of view you want for each item.

`SimpleCursorAdapter`

Use this adapter when your data comes from a `Cursor`. When using `SimpleCursorAdapter`, you must specify a layout to use for each row in the `Cursor` and which columns in the `Cursor` should be inserted into which views of the layout. For example, if you want to create a list of people's names and phone numbers, you can perform a query that returns a `Cursor` containing a row for each person and columns for the names and numbers. You then create a string array specifying which columns from

the `Cursor` you want in the layout for each result and an integer array specifying the corresponding views that each column should be placed:

KOTLINJAVA

```
val fromColumns = arrayOf(ContactsContract.Data.DISPLAY_NAME,
                          ContactsContract.CommonDataKinds.Phone.
NUMBER)
val toViews = intArrayOf(R.id.display_name, R.id.phone_number)
```

When you instantiate the `SimpleCursorAdapter`, pass the layout to use for each result, the `Cursor` containing the results, and these two arrays:

KOTLINJAVA

```
val adapter = SimpleCursorAdapter(this,
        R.layout.person_name_and_number, cursor, fromColumns,
toViews, 0)
val listView = getListView()
listView.adapter = adapter
```

The `SimpleCursorAdapter` then creates a view for each row in the `Cursor` using the provided layout by inserting each `fromColumns` item into the corresponding `toViews` view.

.

If, during the course of your app's life, you change the underlying data that is read by your adapter, you should call `notifyDataSetChanged()`. This will notify the attached view that the data has been changed and it should refresh itself.

extView Attributes

Following are the important attributes related to TextView control. You can check Android official documentation for complete list of attributes and related methods which you can use to change these attributes are run time.

Sr.No.	Attribute & Description
1	**android:id** This is the ID which uniquely identifies the control.
2	**android:capitalize** If set, specifies that this TextView has a textual input method and should automatically capitalize what the user types. • Don't automatically capitalize anything - 0

	• Capitalize the first word of each sentence - 1 • Capitalize the first letter of every word - 2 • Capitalize every character - 3
3	**android:cursorVisible** Makes the cursor visible (the default) or invisible. Default is false.
4	**android:editable** If set to true, specifies that this TextView has an input method.
5	**android:fontFamily** Font family (named by string) for the text.
6	**android:gravity** Specifies how to align the text by the view's x- and/or y-axis when the text is smaller than the view.
7	**android:hint** Hint text to display when the text is empty.
8	**android:inputType** The type of data being placed in a text field. Phone, Date, Time, Number, Password etc.
9	**android:maxHeight** Makes the TextView be at most this many pixels tall.
10	**android:maxWidth** Makes the TextView be at most this many pixels wide.
11	**android:minHeight** Makes the TextView be at least this many pixels tall.
12	**android:minWidth** Makes the TextView be at least this many pixels wide.
13	**android:password**

	Whether the characters of the field are displayed as password dots instead of themselves. Possible value either "true" or "false".
14	**android:phoneNumber** If set, specifies that this TextView has a phone number input method. Possible value either "true" or "false".
15	**android:text** Text to display.
16	**android:textAllCaps** Present the text in ALL CAPS. Possible value either "true" or "false".
17	**android:textColor** Text color. May be a color value, in the form of "#rgb", "#argb", "#rrggbb", or "#aarrggbb".
18	**android:textColorHighlight** Color of the text selection highlight.
19	**android:textColorHint** Color of the hint text. May be a color value, in the form of "#rgb", "#argb", "#rrggbb", or "#aarrggbb".
20	**android:textIsSelectable** Indicates that the content of a non-editable text can be selected. Possible value either "true" or "false".
21	**android:textSize** Size of the text. Recommended dimension type for text is "sp" for scaled-pixels (example: 15sp).
22	**android:textStyle** Style (bold, italic, bolditalic) for the text. You can use or more of the following values separated by '|'. • normal - 0 • bold - 1 • italic - 2

23	**android:typeface** Typeface (normal, sans, serif, monospace) for the text. You can use or more of the following values separated by '|'. • normal - 0 • sans - 1

Button Attributes

Following are the important attributes related to Button control. You can check Android official documentation for complete list of attributes and related methods which you can use to change these attributes are run time.

Inherited from **android.widget.TextView** Class −

Sr.No	Attribute & Description
1	**android:autoText** If set, specifies that this TextView has a textual input method and automatically corrects some common spelling errors.
2	**android:drawableBottom** This is the drawable to be drawn below the text.
3	**android:drawableRight** This is the drawable to be drawn to the right of the text.
4	**android:editable** If set, specifies that this TextView has an input method.
5	**android:text** This is the Text to display.

Inherited from **android.view.View** Class −

Attribute	Description
1	**android:background** This is a drawable to use as the background.

2	**android:contentDescription** This defines text that briefly describes content of the view.
3	**android:id** This supplies an identifier name for this view.
4	**android:onClick** This is the name of the method in this View's context to invoke when the view is clicked.
5	**android:visibility** This controls the initial visibility of the view.

Example

This example will take you through simple steps to show how to create your own Android application using Linear Layout and Button.

Step	Description
1	You will use Android studio IDE to create an Android application and name it as *myapplication* under a package *com.example.saira_000.myapplication* as explained in the *Hello World Example* chapter.
2	Modify *src/MainActivity.java* file to add a click event.
3	Modify the default content of *res/layout/activity_main.xml* file to include Android UI control.
4	No need to declare default string constants at string.xml, Android studio takes care of default string constants.
5	Run the application to launch Android emulator and verify the result of the changes done in the application.

Following is the content of the modified main activity file **src/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.saira_000.myapplication;

import android.content.Intent;
```

```
import android.net.Uri;
import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;

import android.view.Menu;
import android.view.MenuItem;
import android.view.View;

import android.widget.Button;
import android.widget.Toast;

public class MainActivity extends ActionBarActivity {
   Button b1,b2,b3;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      b1=(Button)findViewById(R.id.button);
      b1.setOnClickListener(new View.OnClickListener() {
         @Override
         public void onClick(View v) {
            Toast.makeText(MainActivity.this,"YOUR
MESSAGE",Toast.LENGTH_LONG).show();
         }
      });
   }
}
```

Following will be the content of **res/layout/activity_main.xml** file −

```
<?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:paddingLeft="@dimen/activity_horizontal_margin"
   android:paddingRight="@dimen/activity_horizontal_margin"
   android:paddingTop="@dimen/activity_vertical_margin"
   android:paddingBottom="@dimen/activity_vertical_margin"
   tools:context=".MainActivity">

   <TextView
      android:id="@+id/textView1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Button Control"
      android:layout_alignParentTop="true"
      android:layout_centerHorizontal="true"
      android:textSize="30dp" />

   <TextView
```

```
      android:id="@+id/textView2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Tutorials point"
      android:textColor="#ff87ff09"
      android:textSize="30dp"
      android:layout_below="@+id/textView1"
      android:layout_centerHorizontal="true" />

   <ImageButton
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/imageButton"
      android:src="@drawable/abc"
      android:layout_below="@+id/textView2"
      android:layout_centerHorizontal="true" />

   <EditText
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/editText"
      android:layout_below="@+id/imageButton"
      android:layout_alignRight="@+id/imageButton"
      android:layout_alignEnd="@+id/imageButton" />

   <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Button"
      android:id="@+id/button"
      android:layout_alignTop="@+id/editText"
      android:layout_alignLeft="@+id/textView1"
      android:layout_alignStart="@+id/textView1"
      android:layout_alignRight="@+id/editText"
      android:layout_alignEnd="@+id/editText" />

</RelativeLayout>
```

Following will be the content of **res/values/strings.xml** to define these new constants −

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="app_name">myapplication</string>
</resources>
```

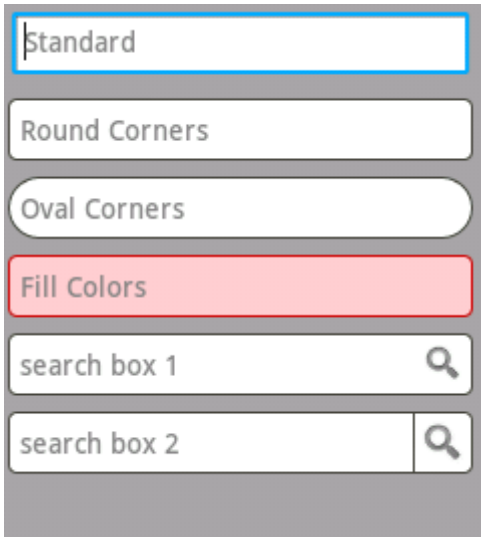
Following is the default content of **AndroidManifest.xml** −

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.saira_000.myapplication" >

   <application
```

```
      android:allowBackup="true"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name"
      android:theme="@style/AppTheme" >

      <activity
         android:name="com.example.guidemo4.MainActivity"
         android:label="@string/app_name" >

         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category
android:name="android.intent.category.LAUNCHER" />
         </intent-filter>

      </activity>

   </application>
</manifest>
```

Let's try to run your **GUIDemo4** application. I assume you had created your **AVD** while doing environment setup. To run the app from Android Studio, open one of your project's activity files and click Run icon from the toolbar.Android Studio installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display following Emulator window −

A EditText is an overlay over TextView that configures itself to be editable. It is the predefined subclass of TextView that includes rich editing capabilities.



Styles of edit text

EditText Attributes

Following are the important attributes related to EditText control. You can check Android official documentation for complete list of attributes and related methods which you can use to change these attributes are run time.

Inherited from **android.widget.TextView** Class −

Sr.No	Attribute & Description
1	**android:autoText** If set, specifies that this TextView has a textual input method and automatically corrects some common spelling errors.
2	**android:drawableBottom** This is the drawable to be drawn below the text.
3	**android:drawableRight** This is the drawable to be drawn to the right of the text.
4	**android:editable** If set, specifies that this TextView has an input method.
5	**android:text**

	This is the Text to display.

Inherited from **android.view.View** Class −

Sr.No	Attribute & Description
1	**android:background** This is a drawable to use as the background.
2	**android:contentDescription** This defines text that briefly describes content of the view.
3	**android:id** This supplies an identifier name for this view.
4	**android:onClick** This is the name of the method in this View's context to invoke when the view is clicked.
5	**android:visibility** This controls the initial visibility of the view.

Example

This example will take you through simple steps to show how to create your own Android application using Linear Layout and EditText.

Step	Description
1	You will use Android studio IDE to create an Android application and name it as *demo* under a package *com.example.demo* as explained in the *Hello World Example* chapter.
2	Modify *src/MainActivity.java* file to add a click event.
3	Modify the default content of *res/layout/activity_main.xml* file to include Android UI control.
4	Define required necessary string constants in *res/values/strings.xml* file

5	Run the application to launch Android emulator and verify the result of the changes done in the application.

Following is the content of the modified main activity file **src/com.example.demo/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

```
package com.example.demo;

import android.os.Bundle;
import android.app.Activity;

import android.view.View;
import android.view.View.OnClickListener;

import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;

public class MainActivity extends Activity {
   EditText eText;
   Button btn;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      eText = (EditText) findViewById(R.id.edittext);
      btn = (Button) findViewById(R.id.button);
      btn.setOnClickListener(new OnClickListener() {
         public void onClick(View v) {
            String str = eText.getText().toString();
            Toast msg =
Toast.makeText(getBaseContext(),str,Toast.LENGTH_LONG);
            msg.show();
         }
      });
   }
}
```

Following will be the content of **res/layout/activity_main.xml** file −

```
<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=".MainActivity" >
```

```
   <TextView
      android:id="@+id/textView1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_alignParentLeft="true"
      android:layout_alignParentTop="true"
      android:layout_marginLeft="14dp"
      android:layout_marginTop="18dp"
      android:text="@string/example_edittext" />

   <Button
      android:id="@+id/button"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_alignLeft="@+id/textView1"
      android:layout_below="@+id/textView1"
      android:layout_marginTop="130dp"
      android:text="@string/show_the_text" />

   <EditText
      android:id="@+id/edittext"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_alignLeft="@+id/button"
      android:layout_below="@+id/textView1"
      android:layout_marginTop="61dp"
      android:ems="10"
      android:text="@string/enter_text" android:inputType="text"
/>

</RelativeLayout>
```

Following will be the content of **res/values/strings.xml** to define these new constants −

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="app_name">demo</string>
   <string name="example_edittext">Example showing
EditText</string>
   <string name="show_the_text">Show the Text</string>
   <string name="enter_text">text changes</string>
</resources>
```

Following is the default content of **AndroidManifest.xml** −

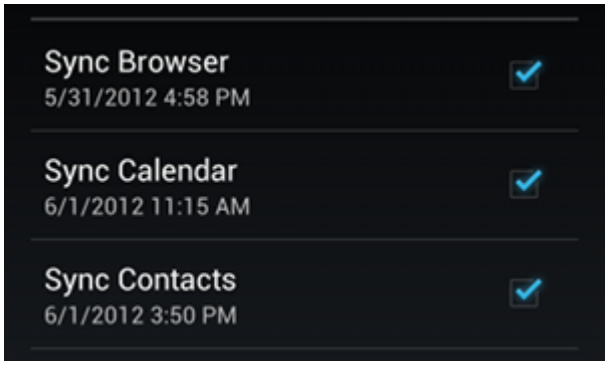
```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.demo" >

   <application
      android:allowBackup="true"
```

```
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name"
      android:theme="@style/AppTheme" >

      <activity
         android:name="com.example.demo.MainActivity"
         android:label="@string/app_name" >

         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category
android:name="android.intent.category.LAUNCHER" />
         </intent-filter>

      </activity>
   </application>

</manifest>
```

Let's try to run your **demo** application. I assume you had created your **AVD** while doing environment setup. To run the app from Android studio, open one of your project's activity files and click Run icon from the toolbar. Android Studio installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display following Emulator window −

Checkboxes

Checkboxes allow the user to select one or more options from a set. Typically, you should present each checkbox option in a vertical list.



To create each checkbox option, create a `CheckBox` in your layout. Because a set of checkbox options allows the user to select multiple items, each checkbox is managed separately and you must register a click listener for each one.

A key class is the following:

- `CheckBox`

Responding to Click Events

When the user selects a checkbox, the `CheckBox` object receives an on-click event.

To define the click event handler for a checkbox, add the `android:onClick` attribute to the `<CheckBox>` element in your XML layout. The value for this attribute must be the name of the method you want to call in response to a click event.
The `Activity` hosting the layout must then implement the corresponding method.

For example, here are a couple `CheckBox` objects in a list:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <CheckBox android:id="@+id/checkbox_meat"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/meat"
        android:onClick="onCheckboxClicked"/>
    <CheckBox android:id="@+id/checkbox_cheese"
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/cheese"
        android:onClick="onCheckboxClicked"/>
</LinearLayout>
```

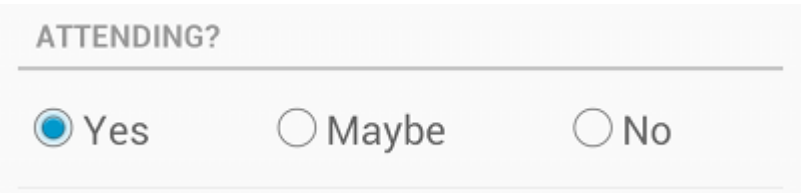
Within the `Activity` that hosts this layout, the following method handles the click event for both checkboxes:

KOTLINJAVA

```
fun onCheckboxClicked(view: View) {
    if (view is CheckBox) {
        val checked: Boolean = view.isChecked

        when (view.id) {
            R.id.checkbox_meat -> {
                if (checked) {
                    // Put some meat on the sandwich
                } else {
                    // Remove the meat
                }
            }
            R.id.checkbox_cheese -> {
                if (checked) {
                    // Cheese me
                } else {
                    // I'm lactose intolerant
                }
            }
            // TODO: Veggie sandwich
        }
    }
}
```

Radio Buttons

Radio buttons allow the user to select one option from a set. You should use radio buttons for optional sets that are mutually exclusive if you think that the user needs to see all available options side-by-side. If it's not necessary to show all options side-by-side, use a spinner instead.



To create each radio button option, create a `RadioButton` in your layout. However, because radio buttons are mutually exclusive, you must group them together inside a `RadioGroup`. By grouping them together, the system ensures that only one radio button can be selected at a time.

Key classes are the following:

- `RadioButton`
- `RadioGroup`

Responding to Click Events

When the user selects one of the radio buttons, the corresponding `RadioButton` object receives an on-click event.

To define the click event handler for a button, add the `android:onClick` attribute to the `<RadioButton>` element in your XML layout. The value for this attribute must be the name of the method you want to call in response to a click event.
The `Activity` hosting the layout must then implement the corresponding method.

For example, here are a couple `RadioButton` objects:

```
<?xml version="1.0" encoding="utf-8"?>
<RadioGroup xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <RadioButton android:id="@+id/radio_pirates"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/pirates"
        android:onClick="onRadioButtonClicked"/>
    <RadioButton android:id="@+id/radio_ninjas"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/ninjas"
        android:onClick="onRadioButtonClicked"/>
</RadioGroup>
```

Toggle Buttons

A toggle button allows the user to change a setting between two states.

You can add a basic toggle button to your layout with the `ToggleButton` object. Android 4.0 (API level 14) introduces another kind of toggle button called a switch that provides a slider control, which you can add with

a `Switch` object. `SwitchCompat` is a version of the Switch widget which runs on devices back to API 7.

If you need to change a button's state yourself, you can use the `CompoundButton.setChecked()` or `CompoundButton.toggle()` method.

Toggle buttons

Switches (in Android 4.0+)

Key classes are the following:

- `ToggleButton`
- `Switch`
- `SwitchCompat`
- `CompoundButton`

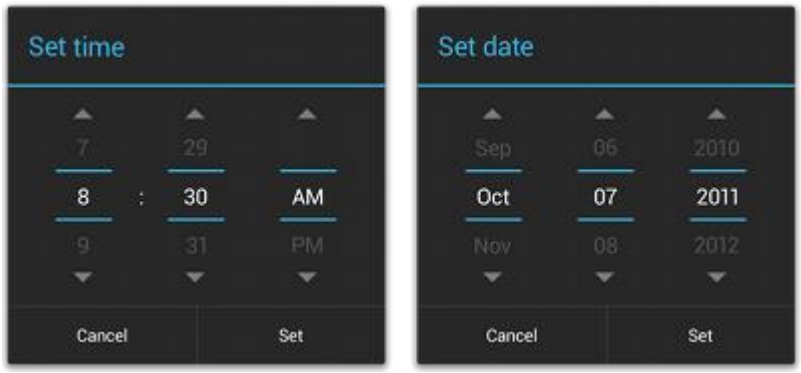
Responding to Button Presses

To detect when the user activates the button or switch, create an `CompoundButton.OnCheckedChangeListener` object and assign it to the button by calling `setOnCheckedChangeListener()`. For example:

KOTLINJAVA
```
val toggle: ToggleButton = findViewById(R.id.togglebutton)
toggle.setOnCheckedChangeListener { _, isChecked ->
    if (isChecked) {
        // The toggle is enabled
    } else {
        // The toggle is disabled
    }
}
```

Pickers

Android provides controls for the user to pick a time or pick a date as ready-to-use dialogs. Each picker provides controls for selecting each part of the time (hour, minute, AM/PM) or date (month, day, year). Using these pickers helps ensure that your users can pick a time or date that is valid, formatted correctly, and adjusted to the user's locale.



We recommend that you use DialogFragment to host each time or date picker. The DialogFragment manages the dialog lifecycle for you and allows you to display the pickers in different layout configurations, such as in a basic dialog on handsets or as an embedded part of the layout on large screens.

Although DialogFragment was first added to the platform in Android 3.0 (API level 11), if your app supports versions of Android older than 3.0—even as low as Android 1.6—you can use the DialogFragment class that's available in the support library for backward compatibility.

Note: The code samples below show how to create dialogs for a time picker and date picker using the support library APIs for **DialogFragment**. If your app's **`minSdkVersion`** is 11 or higher, you can instead use the platform version of **DialogFragment**.

Key classes are the following:

- DatePickerDialog
- TimePickerDialog

Also see the Fragments overview.

Creating a Time Picker

To display a TimePickerDialog using DialogFragment, you need to define a fragment class that extends DialogFragment and return a TimePickerDialog from the fragment's onCreateDialog() method.

Note: If your app supports versions of Android older than 3.0, be sure you've set up your Android project with the support library as described in Setting Up a Project to Use a Library.

Extending DialogFragment for a time picker

To define a DialogFragment for a TimePickerDialog, you must:

- Define the onCreateDialog() method to return an instance of TimePickerDialog

- Implement the TimePickerDialog.OnTimeSetListener interface to receive a callback when the user sets the time.

  Here's an example:

  KOTLINJAVA

```
class TimePickerFragment : DialogFragment(),
TimePickerDialog.OnTimeSetListener {

    override fun onCreateDialog(savedInstanceState: Bundle): Dialog {
        // Use the current time as the default values for the picker
        val c = Calendar.getInstance()
        val hour = c.get(Calendar.HOUR_OF_DAY)
        val minute = c.get(Calendar.MINUTE)

        // Create a new instance of TimePickerDialog and return it
        return TimePickerDialog(activity, this, hour, minute,
DateFormat.is24HourFormat(activity))
    }

    override fun onTimeSet(view: TimePicker, hourOfDay: Int, minute:
Int) {
        // Do something with the time chosen by the user
    }
}
```

  See the TimePickerDialog class for information about the constructor arguments.

  Now all you need is an event that adds an instance of this fragment to your activity.

Showing the time picker

Once you've defined a DialogFragment like the one shown above, you can display the time picker by creating an instance of the DialogFragment and calling show().

For example, here's a button that, when clicked, calls a method to show the dialog:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/pick_time"
    android:onClick="showTimePickerDialog" />
```

When the user clicks this button, the system calls the following method:

KOTLINJAVA

```
fun showTimePickerDialog(v: View) {
    TimePickerFragment().show(supportFragmentManager, "timePicker")
}
```

This method calls show() on a new instance of the DialogFragment defined above. The show() method requires an instance of FragmentManager and a unique tag name for the fragment.

Caution: If your app supports versions of Android lower than 3.0, be sure that you call **getSupportFragmentManager()** to acquire an instance of **FragmentManager**. Also make sure that your activity that displays the time picker extends **FragmentActivity** instead of the standard **Activity** class.

Creating a Date Picker

Creating a DatePickerDialog is just like creating a TimePickerDialog. The only difference is the dialog you create for the fragment.

To display a DatePickerDialog using DialogFragment, you need to define a fragment class that extends DialogFragment and return a DatePickerDialog from the fragment's onCreateDialog() method.

Extending DialogFragment for a date picker

To define a DialogFragment for a DatePickerDialog, you must:

- Define the onCreateDialog() method to return an instance of DatePickerDialog
- Implement the DatePickerDialog.OnDateSetListener interface to receive a callback when the user sets the date.

Here's an example:

KOTLINJAVA

```
class DatePickerFragment : DialogFragment(),
DatePickerDialog.OnDateSetListener {

    override fun onCreateDialog(savedInstanceState: Bundle): Dialog {
        // Use the current date as the default date in the picker
        val c = Calendar.getInstance()
        val year = c.get(Calendar.YEAR)
        val month = c.get(Calendar.MONTH)
        val day = c.get(Calendar.DAY_OF_MONTH)

        // Create a new instance of DatePickerDialog and return it
        return DatePickerDialog(activity, this, year, month, day)
```

```
    }

    override fun onDateSet(view: DatePicker, year: Int, month: Int,
day: Int) {
        // Do something with the date chosen by the user
    }
}
```

See the DatePickerDialog class for information about the constructor arguments.

Now all you need is an event that adds an instance of this fragment to your activity.

Showing the date picker

Once you've defined a DialogFragment like the one shown above, you can display the date picker by creating an instance of the DialogFragment and calling show().

For example, here's a button that, when clicked, calls a method to show the dialog:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/pick_date"
    android:onClick="showDatePickerDialog" />
```

When the user clicks this button, the system calls the following method:

KOTLINJAVA

```
fun showDatePickerDialog(v: View) {
    val newFragment = DatePickerFragment()
    newFragment.show(supportFragmentManager, "datePicker")
}
```

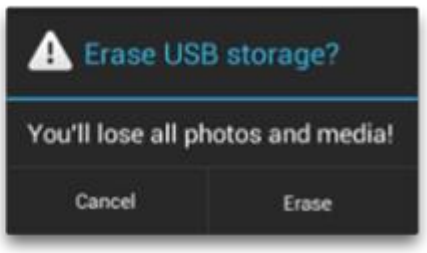
This method calls show() on a new instance of the DialogFragment defined above. The show() method requires an instance of FragmentManager and a unique tag name for the fragment.

Dialogs

A dialog is a small window that prompts the user to make a decision or enter additional information. A dialog does not fill the screen and is normally used for modal events that require users to take an action before they can proceed.

Dialog Design

For information about how to design your dialogs, including recommendations for language, read the Dialogs design guide.



The `Dialog` class is the base class for dialogs, but you should avoid instantiating `Dialog` directly. Instead, use one of the following subclasses:

`AlertDialog`

> A dialog that can show a title, up to three buttons, a list of selectable items, or a custom layout.

`DatePickerDialog` **or** `TimePickerDialog`

> A dialog with a pre-defined UI that allows the user to select a date or time.

Caution: Android includes another dialog class called **`ProgressDialog`** that shows a dialog with a progress bar. This widget is deprecated because it prevents users from interacting with the app while progress is being displayed. If you need to indicate loading or indeterminate progress, you should follow the design guidelines for Progress & Activity and use a **`ProgressBar`** in your layout, instead of using **`ProgressDialog`**.

These classes define the style and structure for your dialog, but you should use a `DialogFragment` as a container for your dialog. The `DialogFragment` class provides all the controls you need to create your dialog and manage its appearance, instead of calling methods on the `Dialog` object.

Using `DialogFragment` to manage the dialog ensures that it correctly handles lifecycle events such as when the user presses the *Back* button or rotates the screen. The `DialogFragment` class also allows you to reuse the dialog's UI as an embeddable component in a larger UI, just like a traditional `Fragment` (such as when you want the dialog UI to appear differently on large and small screens).

The following sections in this guide describe how to use a `DialogFragment` in combination with an `AlertDialog` object. If you'd like to create a date or time picker, you should instead read the Pickers guide.

Note: Because the **`DialogFragment`** class was originally added with Android 3.0 (API level 11), this document describes how to use the **`DialogFragment`** class that's provided with the Support Library. By adding this library to your app, you can use **`DialogFragment`** and a variety of other APIs on devices running Android 1.6 or higher. If the minimum version your app supports is API level 11 or higher, then you can use the framework version of **`DialogFragment`**, but be aware that the links in this document are for the support library APIs. When using the support library, be sure that you import **`android.support.v4.app.DialogFragment`** class and *not* **`android.app.DialogFragment`**.

Creating a Dialog Fragment

You can accomplish a wide variety of dialog designs—including custom layouts and those described in the Dialogs design guide—by extending `DialogFragment` and creating a `AlertDialog` in the `onCreateDialog()` callback method.

For example, here's a basic `AlertDialog` that's managed within a `DialogFragment`:

KOTLINJAVA

```
class FireMissilesDialogFragment : DialogFragment() {

    override fun onCreateDialog(savedInstanceState: Bundle): Dialog {
        return activity?.let {
            // Use the Builder class for convenient dialog construction
            val builder = AlertDialog.Builder(it)
            builder.setMessage(R.string.dialog_fire_missiles)
                    .setPositiveButton(R.string.fire,
                            DialogInterface.OnClickListener { dialog,
id ->
                                // FIRE ZE MISSILES!
                            })
                    .setNegativeButton(R.string.cancel,
                            DialogInterface.OnClickListener { dialog,
id ->
                                // User cancelled the dialog
                            })
            // Create the AlertDialog object and return it
            builder.create()
        } ?: throw IllegalStateException("Activity cannot be null")
    }
}
```

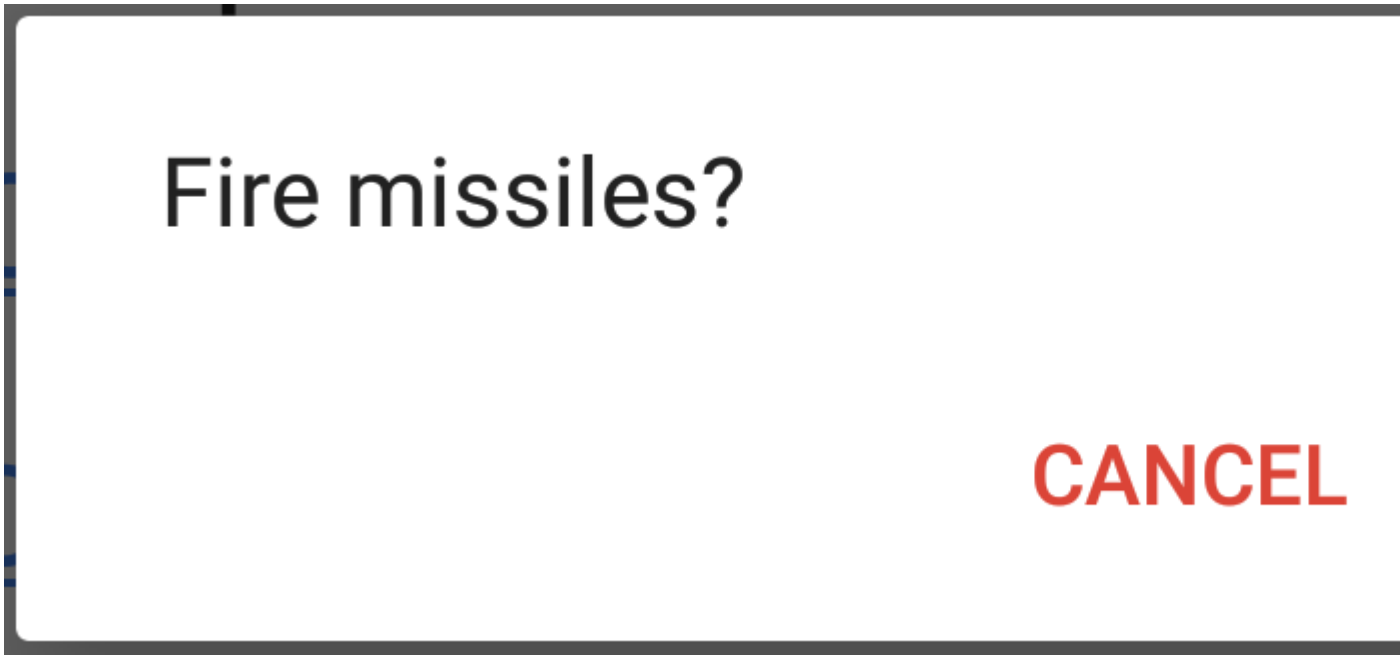


Figure 1. A dialog with a message and two action buttons.

Now, when you create an instance of this class and call `show()` on that object, the dialog appears as shown in figure 1.

The next section describes more about using the `AlertDialog.Builder` APIs to create the dialog.

Depending on how complex your dialog is, you can implement a variety of other callback methods in the `DialogFragment`, including all the basic fragment lifecycle methods.

Building an Alert Dialog

The `AlertDialog` class allows you to build a variety of dialog designs and is often the only dialog class you'll need. As shown in figure 2, there are three regions of an alert dialog:

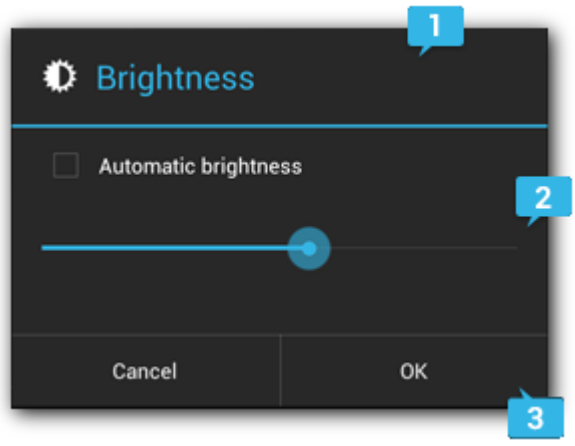


Figure 2. The layout of a dialog.

1. **Title**

   This is optional and should be used only when the content area is occupied by a detailed message, a list, or custom layout. If you need to state a simple message or question (such as the dialog in figure 1), you don't need a title.

2. **Content area**

   This can display a message, a list, or other custom layout.

3. **Action buttons**

   There should be no more than three action buttons in a dialog.

The `AlertDialog.Builder` class provides APIs that allow you to create an `AlertDialog` with these kinds of content, including a custom layout.

To build an `AlertDialog`:

KOTLINJAVA

```
// 1. Instantiate an <code><a
href="/reference/android/app/AlertDialog.Builder.html">AlertDialog.Buil
der</a></code> with its constructor
val builder: AlertDialog.Builder? = activity?.let {
    AlertDialog.Builder(it)
}

// 2. Chain together various setter methods to set the dialog
characteristics
builder?.setMessage(R.string.dialog_message)
        .setTitle(R.string.dialog_title)

// 3. Get the <code><a
href="/reference/android/app/AlertDialog.html">AlertDialog</a></code>
from <code><a
href="/reference/android/app/AlertDialog.Builder.html#create()">create(
)</a></code>
val dialog: AlertDialog? = builder?.create()
```

The following topics show how to define various dialog attributes using the `AlertDialog.Builder` class.

Adding buttons

To add action buttons like those in figure 2, call the `setPositiveButton()` and `setNegativeButton()` methods:

KOTLINJAVA

```
val alertDialog: AlertDialog? = activity?.let {
    val builder = AlertDialog.Builder(it)
    builder.apply {
        setPositiveButton(R.string.ok,
                DialogInterface.OnClickListener { dialog, id ->
                    // User clicked OK button
                })
        setNegativeButton(R.string.cancel,
                DialogInterface.OnClickListener { dialog, id ->
                    // User cancelled the dialog
                })
    }
    // Set other dialog properties
    ...

    // Create the AlertDialog
    builder.create()
}
```

The `set...Button()` methods require a title for the button (supplied by a string resource) and a `DialogInterface.OnClickListener` that defines the action to take when the user presses the button.

There are three different action buttons you can add:

Positive

You should use this to accept and continue with the action (the "OK" action).

Negative

You should use this to cancel the action.

Neutral

You should use this when the user may not want to proceed with the action, but doesn't necessarily want to cancel. It appears between the positive and negative buttons. For example, the action might be "Remind me later."

You can add only one of each button type to an `AlertDialog`. That is, you cannot have more than one "positive" button.

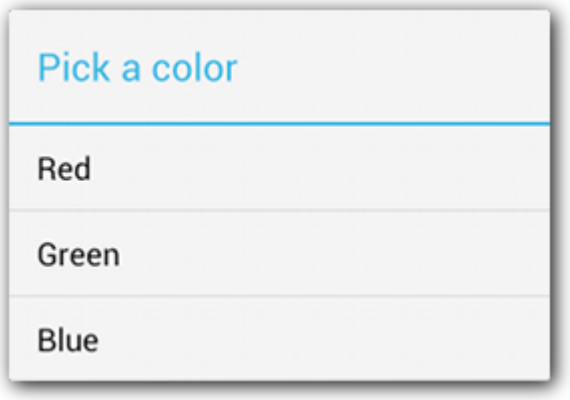


Figure 3. A dialog with a title and list.

Adding a list

There are three kinds of lists available with the `AlertDialog` APIs:

- A traditional single-choice list
- A persistent single-choice list (radio buttons)
- A persistent multiple-choice list (checkboxes)

To create a single-choice list like the one in figure 3, use the `setItems()` method:

KOTLINJAVA

```
override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
    return activity?.let {
        val builder = AlertDialog.Builder(it)
        builder.setTitle(R.string.pick_color)
                .setItems(R.array.colors_array,
                        DialogInterface.OnClickListener { dialog, which
->
                            // The 'which' argument contains the index
position
                            // of the selected item
                        })
        builder.create()
    } ?: throw IllegalStateException("Activity cannot be null")
}
```

Because the list appears in the dialog's content area, the dialog cannot show both a message and a list and you should set a title for the dialog with `setTitle()`. To specify the items for the list, call `setItems()`, passing an array. Alternatively, you can specify a list using `setAdapter()`. This allows you to back the list with dynamic data (such as from a database) using a `ListAdapter`.

If you choose to back your list with a `ListAdapter`, always use a `Loader` so that the content loads asynchronously. This is described further in Building Layouts with an Adapter and the Loaders guide.

Menus

Menus are a common user interface component in many types of applications. To provide a familiar and consistent user experience, you should use the `Menu` APIs to present user actions and other options in your activities.

Beginning with Android 3.0 (API level 11), Android-powered devices are no longer required to provide a dedicated *Menu* button. With this change, Android apps should migrate away from a dependence on the traditional 6-item menu panel and instead provide an app bar to present common user actions.

Although the design and user experience for some menu items have changed, the semantics to define a set of actions and options is still based on the `Menu` APIs. This guide shows how to create the three fundamental types of menus or action presentations on all versions of Android:

Options menu and app bar

> The options menu is the primary collection of menu items for an activity. It's where you should place actions that have a global impact on the app, such as "Search," "Compose email," and "Settings."
>
> See the section about Creating an Options Menu.

Context menu and contextual action mode

> A context menu is a floating menu that appears when the user performs a long-click on an element. It provides actions that affect the selected content or context frame.
>
> The contextual action mode displays action items that affect the selected content in a bar at the top of the screen and allows the user to select multiple items.
>
> See the section about Creating Contextual Menus.

Popup menu

> A popup menu displays a list of items in a vertical list that's anchored to the view that invoked the menu. It's good for providing an overflow of actions that relate to specific content or to provide options for a second part of a command. Actions in a popup menu should **not** directly affect the corresponding content—that's what contextual actions are for. Rather, the popup menu is for extended actions that relate to regions of content in your activity.

See the section about Creating a Popup Menu.

Defining a Menu in XML

For all menu types, Android provides a standard XML format to define menu items. Instead of building a menu in your activity's code, you should define a menu and all its items in an XML menu resource. You can then inflate the menu resource (load it as a Menu object) in your activity or fragment.

Using a menu resource is a good practice for a few reasons:

- It's easier to visualize the menu structure in XML.
- It separates the content for the menu from your application's behavioral code.
- It allows you to create alternative menu configurations for different platform versions, screen sizes, and other configurations by leveraging the app resources framework.

To define the menu, create an XML file inside your project's `res/menu/` directory and build the menu with the following elements:

`<menu>`

Defines a Menu, which is a container for menu items. A `<menu>` element must be the root node for the file and can hold one or more `<item>` and `<group>` elements.

`<item>`

Creates a MenuItem, which represents a single item in a menu. This element may contain a nested `<menu>` element in order to create a submenu.

`<group>`

An optional, invisible container for `<item>` elements. It allows you to categorize menu items so they share properties such as active state and visibility. For more information, see the section about Creating Menu Groups.

Here's an example menu named `game_menu.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/new_game"
          android:icon="@drawable/ic_new_game"
          android:title="@string/new_game"
          android:showAsAction="ifRoom"/>
    <item android:id="@+id/help"
          android:icon="@drawable/ic_help"
```

```
          android:title="@string/help" />
</menu>
```

The `<item>` element supports several attributes you can use to define an item's appearance and behavior. The items in the above menu include the following attributes:

`android:id`

A resource ID that's unique to the item, which allows the application to recognize the item when the user selects it.

`android:icon`

A reference to a drawable to use as the item's icon.

`android:title`

A reference to a string to use as the item's title.

`android:showAsAction`

Specifies when and how this item should appear as an action item in the app bar.

These are the most important attributes you should use, but there are many more available. For information about all the supported attributes, see the Menu Resource document.

You can add a submenu to an item in any menu by adding a `<menu>` element as the child of an `<item>`. Submenus are useful when your application has a lot of functions that can be organized into topics, like items in a PC application's menu bar (File, Edit, View, etc.). For example:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/file"
          android:title="@string/file" >
        <!-- "file" submenu -->
        <menu>
            <item android:id="@+id/create_new"
                  android:title="@string/create_new" />
            <item android:id="@+id/open"
                  android:title="@string/open" />
        </menu>
    </item>
</menu>
```

To use the menu in your activity, you need to inflate the menu resource (convert the XML resource into a programmable object) using MenuInflater.inflate(). In the following sections, you'll see how to inflate a menu for each menu type.

Creating an Options Menu

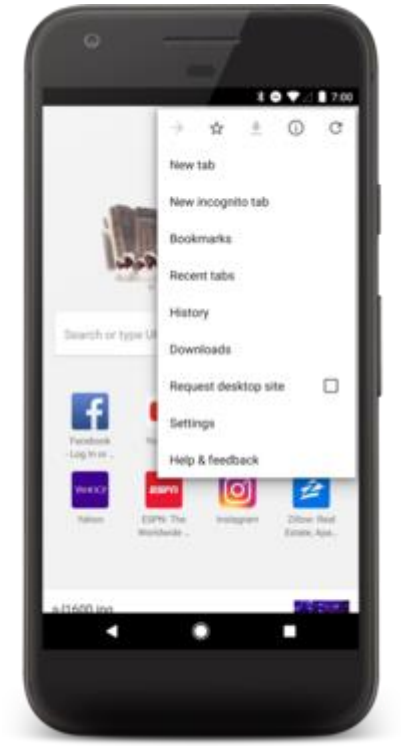


Figure 1. Options menu in the Browser.

The options menu is where you should include actions and other options that are relevant to the current activity context, such as "Search," "Compose email," and "Settings."

Where the items in your options menu appear on the screen depends on the version for which you've developed your application:

- If you've developed your application for **Android 2.3.x (API level 10) or lower**, the contents of your options menu appear at the top of the screen when the user presses the *Menu* button, as shown in figure 1. When opened, the first visible portion is the icon menu, which holds up to six menu items. If your menu includes more than six items, Android places the sixth item and the rest into the overflow menu, which the user can open by selecting *More*.
- If you've developed your application for **Android 3.0 (API level 11) and higher**, items from the options menu are available in the app bar. By default, the system places all items in the action overflow, which the user can reveal with the action overflow icon on the right side of the app bar (or by pressing the device *Menu* button, if available). To enable quick access to important actions, you can promote a few items to appear in the app bar by adding `android:showAsAction="ifRoom"` to the corresponding `<item>` elements (see figure 2).

For more information about action items and other app bar behaviors, see the Adding the App Bar training class.

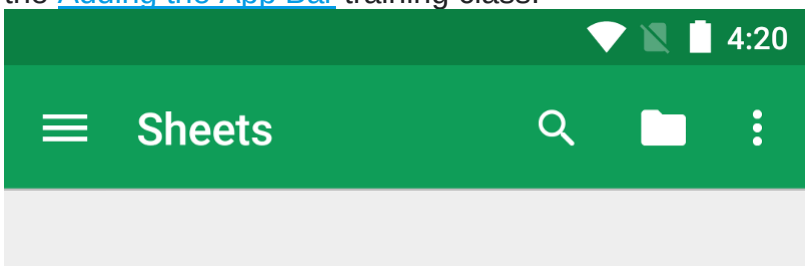


Figure 2. The Google Sheets app, showing several buttons, including the action overflow button.

You can declare items for the options menu from either your Activity subclass or a Fragment subclass. If both your activity and fragment(s) declare items for the options menu, they are combined in the UI. The activity's items appear first, followed by those of each fragment in the order in which each fragment is added to the activity. If necessary, you can re-order the menu items with the `android:orderInCategory` attribute in each `<item>` you need to move.

To specify the options menu for an activity, override onCreateOptionsMenu() (fragments provide their own onCreateOptionsMenu() callback). In this method, you can inflate your menu resource (defined in XML) into the Menu provided in the callback. For example:

KOTLINJAVA

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
    val inflater: MenuInflater = menuInflater
    inflater.inflate(R.menu.game_menu, menu)
    return true
}
```

You can also add menu items using add() and retrieve items with findItem() to revise their properties with MenuItem APIs.

If you've developed your application for Android 2.3.x and lower, the system calls onCreateOptionsMenu() to create the options menu when the user opens the menu for the first time. If you've developed for Android 3.0 and higher, the system calls onCreateOptionsMenu() when starting the activity, in order to show items to the app bar.

Handling click events

When the user selects an item from the options menu (including action items in the app bar), the system calls your activity's onOptionsItemSelected() method. This method passes the MenuItem selected. You can identify the item by calling getItemId(), which returns the unique ID for the menu item (defined by the `android:id` attribute in the menu resource or with an integer given to the add() method). You can match this ID against known menu items to perform the appropriate action. For example:

KOTLINJAVA

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    // Handle item selection
    return when (item.itemId) {
        R.id.new_game -> {
            newGame()
            true
        }
        R.id.help -> {
            showHelp()
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

When you successfully handle a menu item, return `true`. If you don't handle the menu item, you should call the superclass implementation of onOptionsItemSelected() (the default implementation returns false).

If your activity includes fragments, the system first calls onOptionsItemSelected() for the activity then for each fragment (in the order each fragment was added) until one returns `true` or all fragments have been called.

Android smartphones can send and receive messages to or from any other phone that supports Short Message Service (SMS). You have two choices for *sending* SMS messages:

- Use an implicit Intent to launch a messaging app with the ACTION_SENDTO intent action.
  - This is the simplest choice for sending messages. The user can add a picture or other attachment in the messaging app, if the messaging app supports adding attachments.
  - Your app doesn't need code to request permission from the user.
  - If the user has multiple SMS messaging apps installed on the Android phone, the App chooser will appear with a list of these apps, and the user can choose which one to use. (Android smartphones will have at least one, such as Messenger.)
  - The user can change the message in the messaging app before sending it.
  - The user navigates back to your app using the **Back** button.
- Send the SMS message using the sendTextMessage() method or other methods of the SmsManager class.
  - This is a good choice for sending messages from your app without having to use another installed app.
  - Your app must ask the user for permission before sending the SMS message, if the user hasn't already granted permission.
  - The user stays in your app during and after sending the message.
  - You can manage SMS operations such as dividing a message into fragments, sending a multipart message, get carrier-dependent configuration values, and so on.

To *receive* SMS messages, use the onReceive() method of the BroadcastReceiver class.

What you should already KNOW

You should already be able to:

- Create an onClick method for a button with the `android:onClick` attribute.
- Use an implicit intent to perform a function with another app.
- Use a broadcast receiver to receive system events.

What you will LEARN

In this practical, you will learn to:

- Launch an SMS messaging app from your app with a phone number and message.
- Send an SMS message from within an app.
- Check for the SMS permission, and request permission if necessary.
- Receive SMS events using a broadcast receiver.
- Extract an SMS message from an SMS event.

What you will DO

In this practical, you will:

- Create an app that uses an implicit intent to launch a messaging app.
- Pass data (the phone number) and the message with the implicit intent.
- Create an app that sends SMS messages using the SmsManager class.
- Check for the SMS permission, which can change at any time.
- Request permission from the user, if necessary, to send SMS messages.
- Receive and process an SMS message.

App overview

You will create two new apps based on apps you created previously for the lesson about making phone calls:

- **PhoneMessaging**: Rename and refactor the PhoneCallDial app from the previous chapter, and add code to enable a user to not only dial a hard-coded phone number but also send an SMS message to the phone number. It uses an implicit intent using `ACTION_SENDTO` and the phone number to launch a messaging app to send the message.
  As shown in the figure below, the PhoneCallDial app already has TextEdit views for the contact name and the hard-coded phone number, and an ImageButton for making a phone call. You will copy the app, rename it to PhoneMessaging, and modify the layout to include an EditText for entering the message, and another

ImageButton with an icon that the user can tap to send the message.

PhoneMessaging

Jane Doe

15555215556

Enter message here

- **SMS Messaging**: Change the PhoneCallingSample app from the previous chapter to enable a user to enter a phone number, enter an SMS message, and send the message from within the app. It checks for permission and then uses the SmsManager class to send the message.

  As shown in the figure below, the PhoneCallingSample app already has an EditText view for entering the phone number and an ImageButton for making a phone call. You will copy the app, rename it to **SmsMessaging**, and modify the layout to include another EditText for entering the message, and change the

ImageButton to an icon that the user can tap to send the message.

SMS Messaging

Enter a phone number

Enter message here

Task 1. Launch a messaging app to send a message

In this task you create an app called PhoneMessaging, a new version of the PhoneCallDial app from a previous lesson. The new app launches a messaging app with an implicit intent, and passes a fixed phone number and a message entered by the user.

The user can tap the messaging icon in your app to send the message. In the messaging app launched by the intent, the user can tap to send the message, or change the message or the phone number before sending the message. After sending the message, the user can navigate back to your app using the **Back** button.

1.1 Modify the app and layout

1. Copy the **PhoneCallDial** project folder, rename it to **PhoneMessaging**, and refactor it to populate the new name throughout the app project. (See the Appendix for instructions on copying a project.)
2. Add an icon for the messaging button by following these steps:

a. Select drawable in the Project: Android view and choose **File > New > Vector Asset**.

b. Click the Android icon next to "Icon:" to choose an icon. To find a messaging icon, choose **Communication** in the left column.

c. Select the icon, click **OK**, click **Next**, and then click **Finish**.

3. Add the following EditText to the existing layout after the `phone_icon` ImageButton:

```
...
<ImageButton
    android:id="@+id/phone_icon"
    ... />

<EditText
    android:id="@+id/sms_message"
    android:layout_width="200dp"
    android:layout_height="wrap_content"
    android:layout_below="@id/number_to_call"
    android:layout_marginTop="@dimen/activity_vertical_margin"
    android:layout_marginRight="@dimen/activity_horizontal_margin"
    android:hint="Enter message here"
    android:inputType="textMultiLine"/>
```

You will use the `android:id` `sms_message` to retrieve the message in your code. You can use `@dimen/activity_horizontal_margin` and `@dimen/activity_vertical_margin` for the EditText margins because they are already defined in the dimens.xml file. The EditText view uses the `android:inputType` attribute set to `"textMultiLine"` for entering multiple lines of text.

18. After adding hard-coded strings and dimensions, extract them into resources:

- `android:layout_width="@dimen/edittext_width"`: The width of the EditText message (200dp).
- `android:hint="@string/enter_message_here"`: The hint for the EditText ("Enter message here").

19. Add the following ImageButton to the layout after the above EditText:

```
<ImageButton
    android:id="@+id/message_icon"
    android:contentDescription="Send a message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/activity_vertical_margin"
    android:layout_toRightOf="@id/sms_message"
    android:layout_toEndOf="@id/sms_message"
    android:layout_below="@id/phone_icon"
    android:src="@drawable/ic_message_black_24dp"
    android:onClick="smsSendMessage"/>
```

    You will use the `android:id` `message_icon` to refer to the ImageButton for launching the messaging app. Use the vector asset you added previously (such as `ic_message_black_24dp` for a messaging icon) for the ImageButton.
31. After adding the hard-coded string for the `android:contentDescription` attribute, extract it into the resource `send_a_message`.
    The `smsSendMessage()` method referred to in the `android:onClick` attribute remains highlighted until you create this method in the MainActivity, which you will do in the next step.
32. Click `smsSendMessage` in the `android:onClick` attribute, click the red light bulb that appears, and then select **Create smsSendMessage(View) in 'MainActivity'**. Android Studio automatically creates the `smsSendMessage()` method in MainActivity as `public`, returning `void`, with a `View` parameter. This method is called when the user taps the `message_icon` ImageButton.

```
public void smsSendMessage(View view) {
}
```

Your app's layout should now look like the following figure:

PhoneMessaging

Jane Doe

15555215556

Enter message here

1.2 Edit the onClick method in MainActivity

1. Inside the `smsSendMessage()` method in MainActivity, get the phone number from the `number_to_call` TextView, and concatenate it with the `smsto:` prefix (as in `smsto:14155551212`) to create the phone number URI string `smsNumber`:

```
...
TextView textView = (TextView) findViewById(R.id.number_to_call);
// Use format with "smsto:" and phone number to create smsNumber.
String smsNumber = String.format("smsto: %s",
                            textView.getText().toString());
...
```

8. Get the string of the message entered into the EditText view:

```
...
// Find the sms_message view.
EditText smsEditText = (EditText) findViewById(R.id.sms_message);
// Get the text of the SMS message.
String sms = smsEditText.getText().toString();
...
```

15. Create an implicit intent (`smsIntent`) with the intent action `ACTION_SENDTO`, and set the phone number and text message as intent data and extended data, using `setData()` and `putExtra`:

```
...
// Create the intent.
Intent smsIntent = new Intent(Intent.ACTION_SENDTO);
// Set the data for the intent as the phone number.
smsIntent.setData(Uri.parse(smsNumber));
  // Add the message (sms) with the key ("sms_body").
smsIntent.putExtra("sms_body", sms);
...
```

The putExtra() method needs two strings: the key identifying the type of data (`"sms_body"`) and the data itself, which is the text of the message (`sms`). For more information about common intents and the `putExtra()` method, see Common Intents: Text Messaging.

24. Add a check to see if the implicit intent resolves to a package (a messaging app). If it does, send the intent with `startActivity()`, and the system launches the app. If it does not, log an error.

```
...
// If package resolves (target app installed), send intent.
if (smsIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(smsIntent);
} else {
    Log.e(TAG, "Can't resolve app for ACTION_SENDTO Intent");
}
...
```

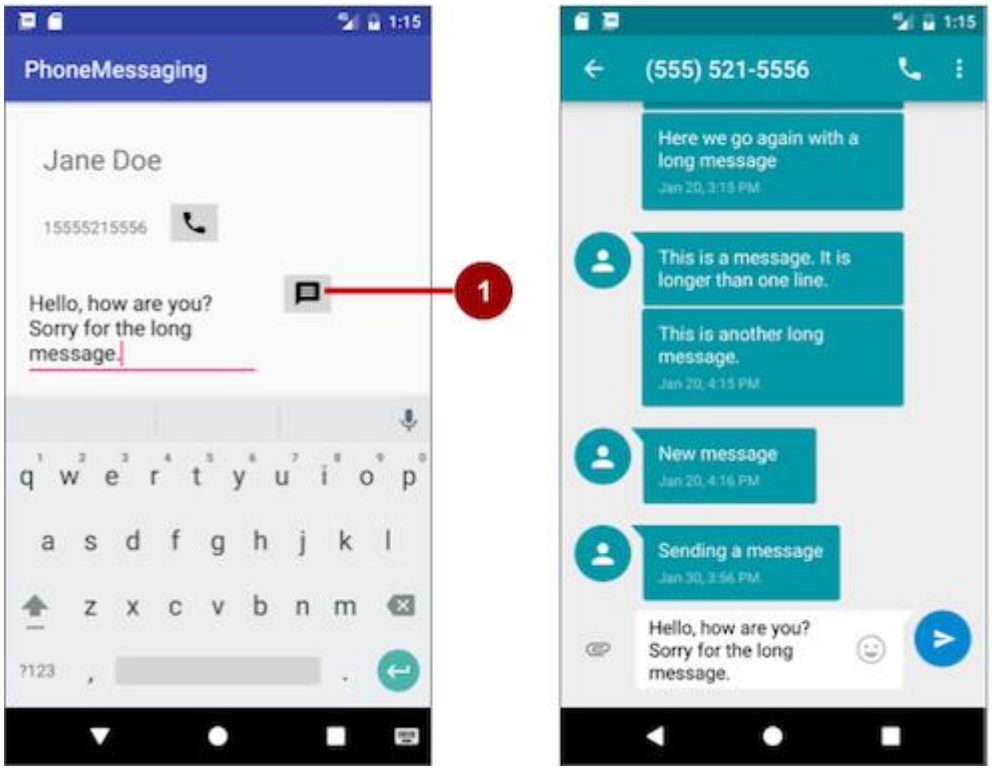
The full method should now look like the following:

```
public void smsSendMessage(View view) {
    TextView textView = (TextView) findViewById(R.id.number_to_call);
    // Use format with "smsto:" and phone number to create smsNumber.
    String smsNumber = String.format("smsto: %s",
                                     textView.getText().toString());
    // Find the sms_message view.
    EditText smsEditText = (EditText) findViewById(R.id.sms_message);
    // Get the text of the sms message.
    String sms = smsEditText.getText().toString();
    // Create the intent.
    Intent smsIntent = new Intent(Intent.ACTION_SENDTO);
    // Set the data for the intent as the phone number.
    smsIntent.setData(Uri.parse(smsNumber));
    // Add the message (sms) with the key ("sms_body").
```

```
    smsIntent.putExtra("sms_body", sms);
    // If package resolves (target app installed), send intent.
    if (smsIntent.resolveActivity(getPackageManager()) != null) {
        startActivity(smsIntent);
    } else {
        Log.d(TAG, "Can't resolve app for ACTION_SENDTO Intent");
    }
}
```

1.3 Run the app

1. Run the app on either an emulator or a device.
2. Enter a message, and tap the messaging icon (marked "1" in the left side of the figure below). The messaging app appears, as shown on the right side of the figure below.



3. Use the **Back** button to return to the PhoneMessaging app. You may need to tap or click it more than once to leave the SMS messaging app.

Solution code

Android Studio project: PhoneMessaging

Task 2. Send an SMS message from within an app

In this task you will copy the **PhoneCallingSample** app from the lesson on making a phone call, rename and refactor it to **SmsMessaging**, and modify its layout and

code to create an app that enables a user to enter a phone number, enter an SMS message, and send the message from within the app.

In the first step you will add the code to send the message, but the app will work only if you first turn on SMS permission manually for the app in Settings on your device or emulator.

In subsequent steps you will do away with setting this permission manually by requesting SMS permission from the app's user if it is not already set.

next →← prev

Android Google Map

Android provides facility to integrate Google map in our application. Google map displays your current location, navigate location direction, search location etc. We can also customize Google map according to our requirement.

Types of Google Maps

There are four different types of Google maps, as well as an optional to no map at all. Each of them gives different view on map. These maps are as follow:

1. **Normal:** This type of map displays typical road map, natural features like river and some features build by humans.
2. **Hybrid:** This type of map displays satellite photograph data with typical road maps. It also displays road and feature labels.
3. **Satellite:** Satellite type displays satellite photograph data, but doesn't display road and feature labels.
4. **Terrain:** This type displays photographic data. This includes colors, contour lines and labels and perspective shading.
5. **None:** This type displays an empty grid with no tiles loaded.

Syntax of different types of map

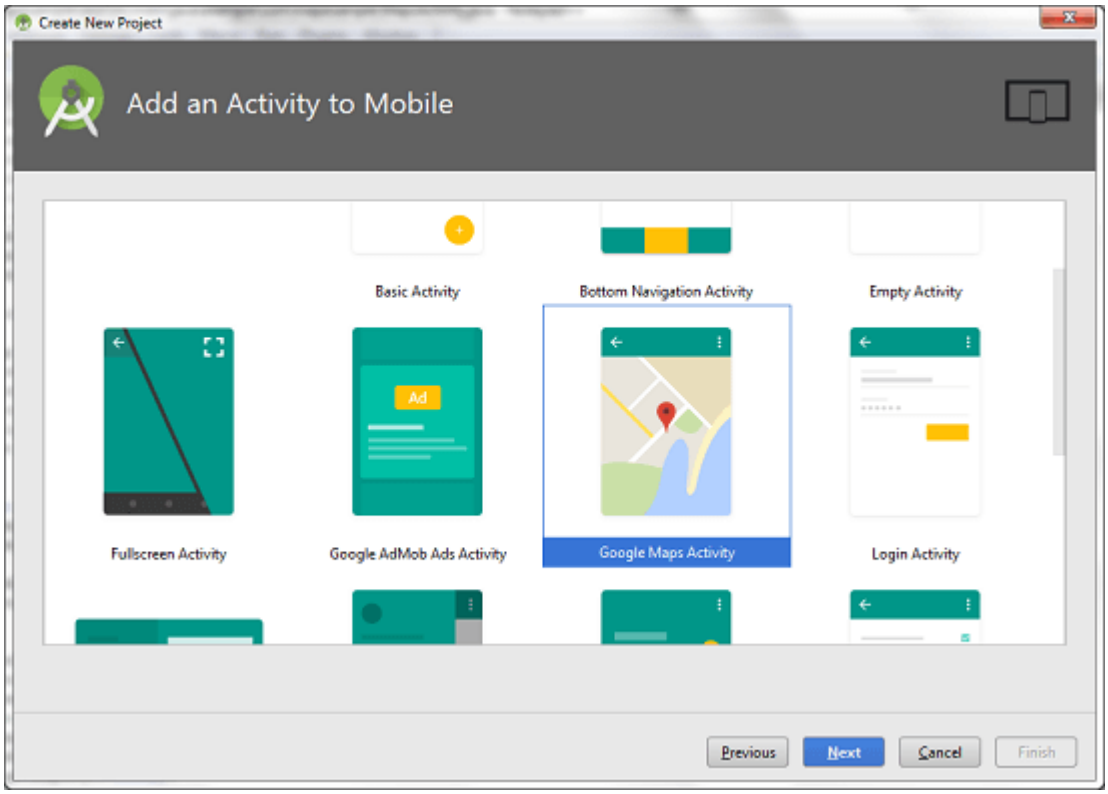
1. googleMap.setMapType(GoogleMap.MAP_TYPE_NORMAL);
2. googleMap.setMapType(GoogleMap.MAP_TYPE_HYBRID);
3. googleMap.setMapType(GoogleMap.MAP_TYPE_SATELLITE);
4. googleMap.setMapType(GoogleMap.MAP_TYPE_TERRAIN);

Methods of Google map

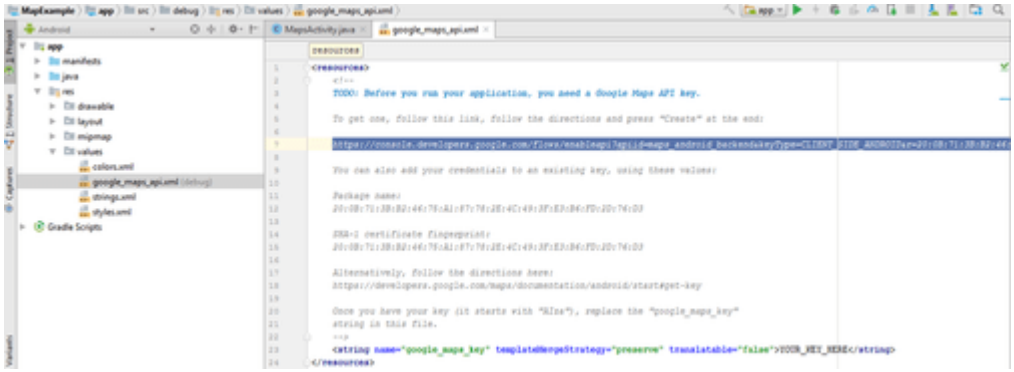
Google map API provides several methods that help to customize Google map. These methods are as following:

Example of Google Map

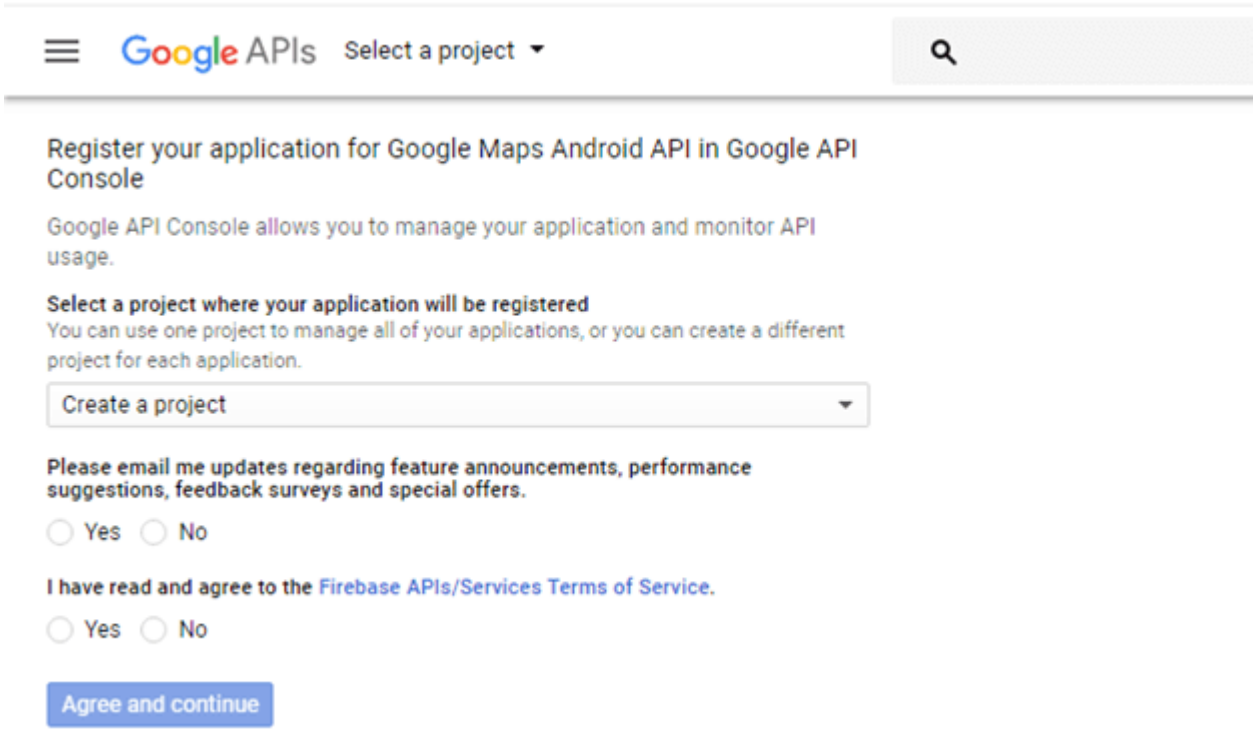
Let's create an example of Google map integrating within our app. For doing this we select Google Maps Activity.



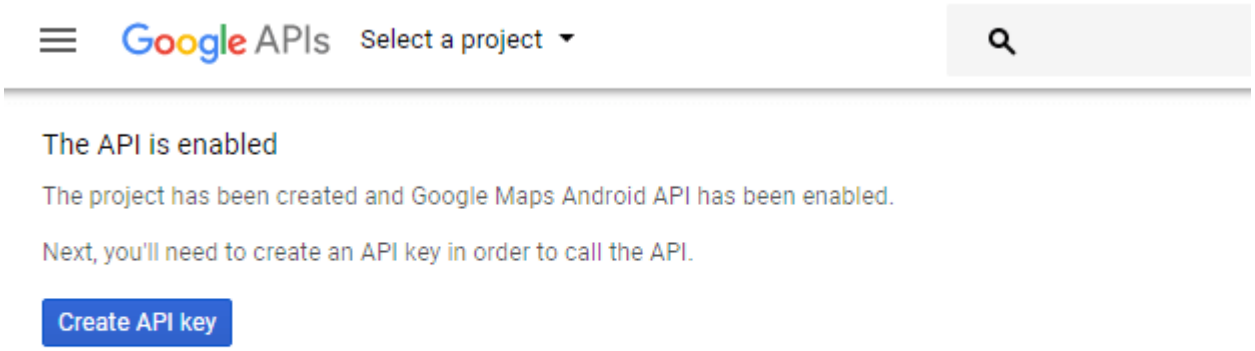
Copy the URL from google_map_api.xml file to generate Google map key.



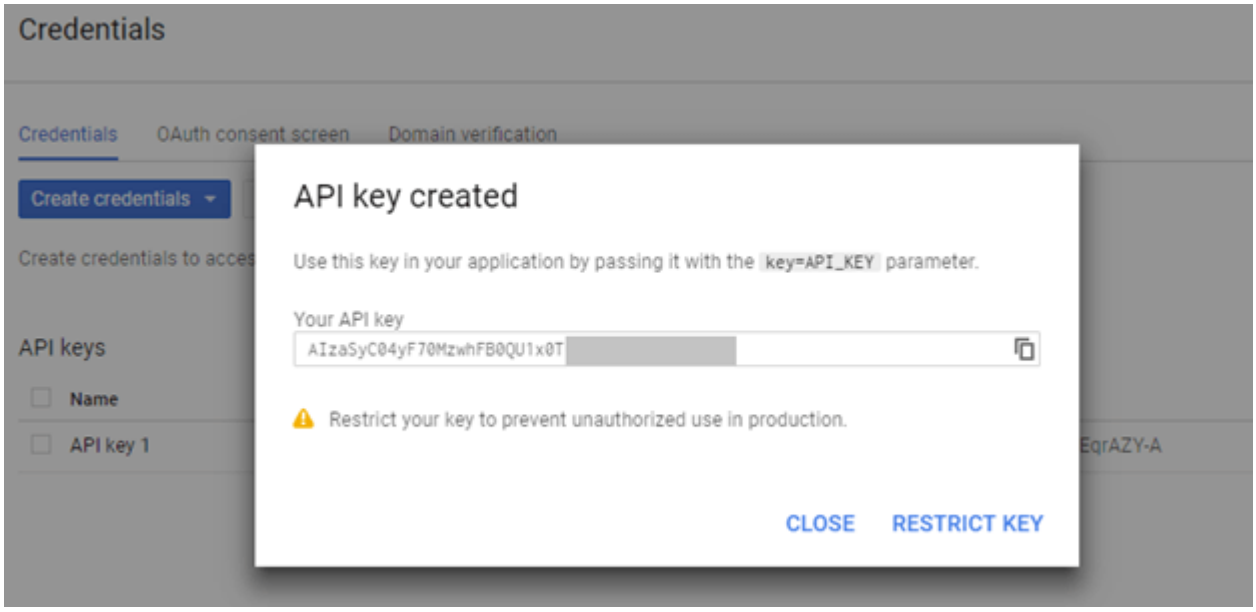
Paste the copied URL at the browser. It will open the following page.



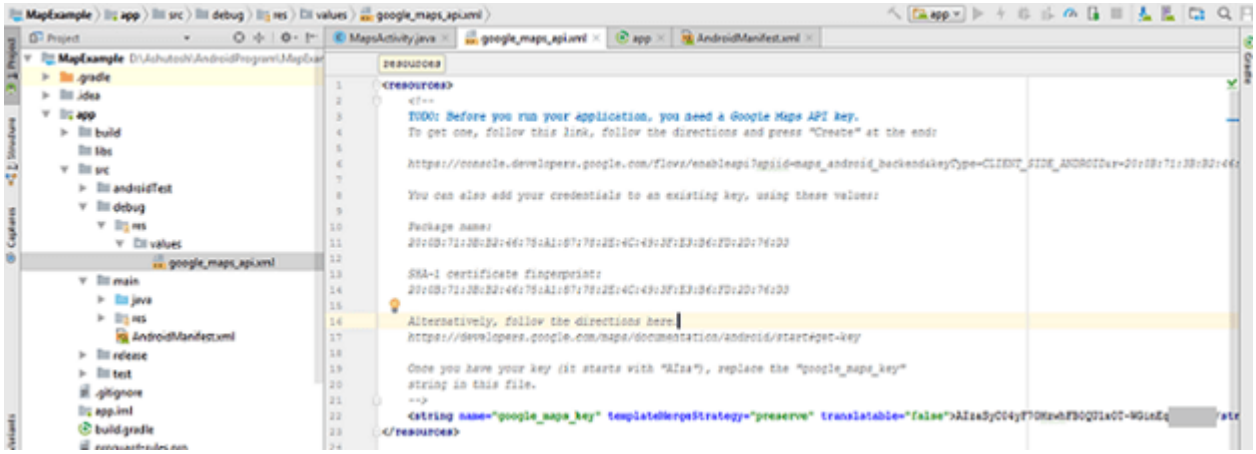
Click on Create API key to generate API key.



After clicking on Create API key, it will generate our API key displaying the following screen.



Copy this generated API key in our *google_map_api.xml* file



activity_maps.xml

```
1. <fragment xmlns:android="http://schemas.android.com/apk/res/android"
2.     xmlns:map="http://schemas.android.com/apk/res-auto"
3.     xmlns:tools="http://schemas.android.com/tools"
4.     android:id="@+id/map"
5.     android:name="com.google.android.gms.maps.SupportMapFragment"
6.     android:layout_width="match_parent"
7.     android:layout_height="match_parent"
8.     tools:context="example.com.mapexample.MapsActivity" />
```

MapsActivity.java

To get the GoogleMap object in our MapsActivity.java class we need to implement the OnMapReadyCallback interface and override the onMapReady() callback method.

```
1.  package example.com.mapexample;
2.
3.  import android.support.v4.app.FragmentActivity;
4.  import android.os.Bundle;
5.  import com.google.android.gms.maps.CameraUpdateFactory;
6.  import com.google.android.gms.maps.GoogleMap;
7.  import com.google.android.gms.maps.OnMapReadyCallback;
8.  import com.google.android.gms.maps.SupportMapFragment;
9.  import com.google.android.gms.maps.model.LatLng;
10. import com.google.android.gms.maps.model.MarkerOptions;
11.
12. public class MapsActivity extends FragmentActivity implements OnMapReadyCallback
    {
13.
14.     private GoogleMap mMap;
15.
16.     @Override
17.     protected void onCreate(Bundle savedInstanceState) {
18.         super.onCreate(savedInstanceState);
19.         setContentView(R.layout.activity_maps);
20.         // Obtain the SupportMapFragment and get notified when the map is ready to be u
    sed.
21.         SupportMapFragment mapFragment = (SupportMapFragment) getSupportFragmen
    tManager()
22.                 .findFragmentById(R.id.map);
23.         mapFragment.getMapAsync(this);
24.
25.     }
26.
27.     @Override
28.     public void onMapReady(GoogleMap googleMap) {
29.         mMap = googleMap;
30.
31.         // Add a marker in Sydney and move the camera
32.         LatLng sydney = new LatLng(-34, 151);
33.         mMap.addMarker(new MarkerOptions().position(sydney).title("Marker in Sydney"))
    ;
34.         mMap.moveCamera(CameraUpdateFactory.newLatLng(sydney));
35.
36.     }
37. }
```

Required Permission

Add the following user-permission in AndroidManifest.xml file.

```
1. <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
2. <uses-
   permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
3. <uses-permission android:name="android.permission.INTERNET" />
```

AndroidManifest.xml

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3.     package="example.com.mapexample">
4.     <!--
5.         The ACCESS_COARSE/FINE_LOCATION permissions are not required to use
6.         Google Maps Android API v2, but you must specify either coarse or fine
7.         location permissions for the 'MyLocation' functionality.
8.     -->
9.     <uses-
   permission android:name="android.permission.ACCESS_FINE_LOCATION" />
10.    <uses-
   permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
11.    <uses-permission android:name="android.permission.INTERNET" />
12.
13.    <application
14.        android:allowBackup="true"
15.        android:icon="@mipmap/ic_launcher"
16.        android:label="@string/app_name"
17.        android:roundIcon="@mipmap/ic_launcher_round"
18.        android:supportsRtl="true"
19.        android:theme="@style/AppTheme">
20.
21.        <meta-data
22.            android:name="com.google.android.geo.API_KEY"
23.            android:value="@string/google_maps_key" />
24.
25.        <activity
26.            android:name=".MapsActivity"
27.            android:label="@string/title_activity_maps">
28.            <intent-filter>
29.                <action android:name="android.intent.action.MAIN" />
30.
31.                <category android:name="android.intent.category.LAUNCHER" />
32.            </intent-filter>
33.        </activity>
```

```
    </application>

</manifest>
```

build.gradel

Add the following dependencies in *build.gradel* file.

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support:appcompat-v7:26.1.0'
    implementation 'com.google.android.gms:play-services-maps:11.8.0'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.1'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.1'
}
```

Output

SQLite is a opensource SQL database that stores data to a text file on a device. Android comes in with built in SQLite database implementation.

SQLite supports all the relational database features. In order to access this database, you don't need to establish any kind of connections for it like JDBC,ODBC e.t.c

Database - Package

The main package is android.database.sqlite that contains the classes to manage your own databases

Database - Creation

In order to create a database you just need to call this method openOrCreateDatabase with your database name and mode as a parameter. It returns an instance of SQLite database which you have to receive in your own object.Its syntax is given below

```
SQLiteDatabase mydatabase = openOrCreateDatabase("your database name",MODE_PRIVATE,null);
```

Apart from this , there are other functions available in the database package , that does this job. They are listed below

Sr.No	Method & Description
1	**openDatabase(String path, SQLiteDatabase.CursorFactory factory, int flags, DatabaseErrorHandler errorHandler)** This method only opens the existing database with the appropriate flag mode. The common flags mode could be OPEN_READWRITE OPEN_READONLY
2	**openDatabase(String path, SQLiteDatabase.CursorFactory factory, int flags)** It is similar to the above method as it also opens the existing database but it does not define any handler to handle the errors of databases
3	**openOrCreateDatabase(String path, SQLiteDatabase.CursorFactory factory)** It not only opens but create the database if it not exists. This method is equivalent to openDatabase method.
4	**openOrCreateDatabase(File file, SQLiteDatabase.CursorFactory factory)** This method is similar to above method but it takes the File object as a path rather then a string. It is equivalent to file.getPath()

Database - Insertion

we can create table or insert data into table using execSQL method defined in SQLiteDatabase class. Its syntax is given below

```
mydatabase.execSQL("CREATE TABLE IF NOT EXISTS
TutorialsPoint(Username VARCHAR,Password VARCHAR);");
mydatabase.execSQL("INSERT INTO TutorialsPoint
VALUES('admin','admin');");
```

This will insert some values into our table in our database. Another method that also does the same job but take some additional parameter is given below

Sr.No	Method & Description
1	**execSQL(String sql, Object[] bindArgs)** This method not only insert data , but also used to update or modify already existing data in database using bind arguments

Database - Fetching

We can retrieve anything from database using an object of the Cursor class. We will call a method of this class called rawQuery and it will return a resultset with the cursor pointing to the table. We can move the cursor forward and retrieve the data.

```
Cursor resultSet = mydatbase.rawQuery("Select * from
TutorialsPoint",null);
resultSet.moveToFirst();
String username = resultSet.getString(0);
String password = resultSet.getString(1);
```

There are other functions available in the Cursor class that allows us to effectively retrieve the data. That includes

Sr.No	Method & Description
1	**getColumnCount()** This method return the total number of columns of the table.
2	**getColumnIndex(String columnName)** This method returns the index number of a column by specifying the name of the column
3	**getColumnName(int columnIndex)** This method returns the name of the column by specifying the index of the column
4	**getColumnNames()** This method returns the array of all the column names of the table.

5	**getCount()** This method returns the total number of rows in the cursor
6	**getPosition()** This method returns the current position of the cursor in the table
7	**isClosed()** This method returns true if the cursor is closed and return false otherwise

Database - Helper class

For managing all the operations related to the database , an helper class has been given and is called SQLiteOpenHelper. It automatically manages the creation and update of the database. Its syntax is given below

```
public class DBHelper extends SQLiteOpenHelper {
   public DBHelper(){
      super(context,DATABASE_NAME,null,1);
   }
   public void onCreate(SQLiteDatabase db) {}
   public void onUpgrade(SQLiteDatabase database, int oldVersion,
int newVersion) {}
}
```

Example

Here is an example demonstrating the use of SQLite Database. It creates a basic contacts applications that allows insertion, deletion and modification of contacts.

To experiment with this example, you need to run this on an actual device on which camera is supported.

Steps	Description
1	You will use Android studio to create an Android application under a package com.example.sairamkrishna.myapplication.
2	Modify src/MainActivity.java file to get references of all the XML components and populate the contacts on listView.
3	Create new src/DBHelper.java that will manage the database work

4	Create a new Activity as DisplayContact.java that will display the contact on the screen
5	Modify the res/layout/activity_main to add respective XML components
6	Modify the res/layout/activity_display_contact.xml to add respective XML components
7	Modify the res/values/string.xml to add necessary string components
8	Modify the res/menu/display_contact.xml to add necessary menu components
9	Create a new menu as res/menu/mainmenu.xml to add the insert contact option
10	Run the application and choose a running android device and install the application on it and verify the results.

Following is the content of the modified **MainActivity.java**.

```
package com.example.sairamkrishna.myapplication;

import android.content.Context;
import android.content.Intent;
import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;

import android.view.KeyEvent;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;

import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ListView;

import java.util.ArrayList;
import java.util.List;

public class MainActivity extends ActionBarActivity {
   public final static String EXTRA_MESSAGE = "MESSAGE";
   private ListView obj;
   DBHelper mydb;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
```

```
      setContentView(R.layout.activity_main);

      mydb = new DBHelper(this);
      ArrayList array_list = mydb.getAllCotacts();
      ArrayAdapter arrayAdapter=new
ArrayAdapter(this,android.R.layout.simple_list_item_1,
array_list);

      obj = (ListView)findViewById(R.id.listView1);
      obj.setAdapter(arrayAdapter);
      obj.setOnItemClickListener(new OnItemClickListener(){
         @Override
         public void onItemClick(AdapterView<?> arg0, View arg1,
int arg2,long arg3) {
            // TODO Auto-generated method stub
            int id_To_Search = arg2 + 1;

            Bundle dataBundle = new Bundle();
            dataBundle.putInt("id", id_To_Search);

            Intent intent = new
Intent(getApplicationContext(),DisplayContact.class);

            intent.putExtras(dataBundle);
            startActivity(intent);
         }
      });
   }

   @Override
   public boolean onCreateOptionsMenu(Menu menu) {
      // Inflate the menu; this adds items to the action bar if
it is present.
      getMenuInflater().inflate(R.menu.menu_main, menu);
      return true;
   }

   @Override
   public boolean onOptionsItemSelected(MenuItem item){
      super.onOptionsItemSelected(item);

      switch(item.getItemId()) {
         case R.id.item1:Bundle dataBundle = new Bundle();
         dataBundle.putInt("id", 0);

         Intent intent = new
Intent(getApplicationContext(),DisplayContact.class);
         intent.putExtras(dataBundle);

         startActivity(intent);
         return true;
         default:
         return super.onOptionsItemSelected(item);
```

```
        }
    }

    public boolean onKeyDown(int keycode, KeyEvent event) {
        if (keycode == KeyEvent.KEYCODE_BACK) {
            moveTaskToBack(true);
        }
        return super.onKeyDown(keycode, event);
    }
}
```

Following is the modified content of display contact activity **DisplayContact.java**

```
package com.example.sairamkrishna.myapplication;

import android.os.Bundle;
import android.app.Activity;
import android.app.AlertDialog;

import android.content.DialogInterface;
import android.content.Intent;
import android.database.Cursor;

import android.view.Menu;
import android.view.MenuItem;
import android.view.View;

import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;

public class DisplayContact extends Activity {
    int from_Where_I_Am_Coming = 0;
    private DBHelper mydb ;

    TextView name ;
    TextView phone;
    TextView email;
    TextView street;
    TextView place;
    int id_To_Update = 0;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_display_contact);
        name = (TextView) findViewById(R.id.editTextName);
        phone = (TextView) findViewById(R.id.editTextPhone);
        email = (TextView) findViewById(R.id.editTextStreet);
        street = (TextView) findViewById(R.id.editTextEmail);
        place = (TextView) findViewById(R.id.editTextCity);

        mydb = new DBHelper(this);
```

```
      Bundle extras = getIntent().getExtras();
      if(extras !=null) {
         int Value = extras.getInt("id");

         if(Value>0){
            //means this is the view part not the add contact
part.
            Cursor rs = mydb.getData(Value);
            id_To_Update = Value;
            rs.moveToFirst();

            String nam =
rs.getString(rs.getColumnIndex(DBHelper.CONTACTS_COLUMN_NAME));
            String phon =
rs.getString(rs.getColumnIndex(DBHelper.CONTACTS_COLUMN_PHONE));
            String emai =
rs.getString(rs.getColumnIndex(DBHelper.CONTACTS_COLUMN_EMAIL));
            String stree =
rs.getString(rs.getColumnIndex(DBHelper.CONTACTS_COLUMN_STREET));
            String plac =
rs.getString(rs.getColumnIndex(DBHelper.CONTACTS_COLUMN_CITY));

            if (!rs.isClosed())  {
               rs.close();
            }
            Button b = (Button)findViewById(R.id.button1);
            b.setVisibility(View.INVISIBLE);

            name.setText((CharSequence)nam);
            name.setFocusable(false);
            name.setClickable(false);

            phone.setText((CharSequence)phon);
            phone.setFocusable(false);
            phone.setClickable(false);

            email.setText((CharSequence)emai);
            email.setFocusable(false);
            email.setClickable(false);

            street.setText((CharSequence)stree);
            street.setFocusable(false);
            street.setClickable(false);

            place.setText((CharSequence)plac);
            place.setFocusable(false);
            place.setClickable(false);
         }
      }
   }

   @Override
   public boolean onCreateOptionsMenu(Menu menu) {
```

```
      // Inflate the menu; this adds items to the action bar if
it is present.
      Bundle extras = getIntent().getExtras();

      if(extras !=null) {
         int Value = extras.getInt("id");
         if(Value>0){
            getMenuInflater().inflate(R.menu.display_contact,
menu);
         } else{
            getMenuInflater().inflate(R.menu.menu_main menu);
         }
      }
      return true;
   }

   public boolean onOptionsItemSelected(MenuItem item) {
      super.onOptionsItemSelected(item);
      switch(item.getItemId()) {
         case R.id.Edit_Contact:
         Button b = (Button)findViewById(R.id.button1);
         b.setVisibility(View.VISIBLE);
         name.setEnabled(true);
         name.setFocusableInTouchMode(true);
         name.setClickable(true);

         phone.setEnabled(true);
         phone.setFocusableInTouchMode(true);
         phone.setClickable(true);

         email.setEnabled(true);
         email.setFocusableInTouchMode(true);
         email.setClickable(true);

         street.setEnabled(true);
         street.setFocusableInTouchMode(true);
         street.setClickable(true);

         place.setEnabled(true);
         place.setFocusableInTouchMode(true);
         place.setClickable(true);

         return true;
         case R.id.Delete_Contact:

         AlertDialog.Builder builder = new
AlertDialog.Builder(this);
         builder.setMessage(R.string.deleteContact)
            .setPositiveButton(R.string.yes, new
DialogInterface.OnClickListener() {
               public void onClick(DialogInterface dialog, int
id) {
                  mydb.deleteContact(id_To_Update);
```

```
                    Toast.makeText(getApplicationContext(),
"Deleted Successfully",
                       Toast.LENGTH_SHORT).show();
                    Intent intent = new
Intent(getApplicationContext(),MainActivity.class);
                    startActivity(intent);
                }
            })
            .setNegativeButton(R.string.no, new
DialogInterface.OnClickListener() {
               public void onClick(DialogInterface dialog, int id) {
                  // User cancelled the dialog
               }
            });

            AlertDialog d = builder.create();
            d.setTitle("Are you sure");
            d.show();

            return true;
            default:
            return super.onOptionsItemSelected(item);

         }
      }

      public void run(View view) {
         Bundle extras = getIntent().getExtras();
         if(extras !=null) {
            int Value = extras.getInt("id");
            if(Value>0){

if(mydb.updateContact(id_To_Update,name.getText().toString(),
                  phone.getText().toString(),
email.getText().toString(),
                                    street.getText().toString(),
place.getText().toString())){
                  Toast.makeText(getApplicationContext(), "Updated",
Toast.LENGTH_SHORT).show();
                  Intent intent = new
Intent(getApplicationContext(),MainActivity.class);
                  startActivity(intent);
               } else{
                  Toast.makeText(getApplicationContext(), "not
Updated", Toast.LENGTH_SHORT).show();
               }
            } else{
               if(mydb.insertContact(name.getText().toString(),
phone.getText().toString(),
                                    email.getText().toString(),
street.getText().toString(),
                                    place.getText().toString())){
                     Toast.makeText(getApplicationContext(), "done",
```

```
Toast.LENGTH_SHORT).show();
            } else{
               Toast.makeText(getApplicationContext(), "not
done",
                                          Toast.LENGTH_SHORT).show();

            }
            Intent intent = new
Intent(getApplicationContext(),MainActivity.class);
            startActivity(intent);
         }
      }
   }
}
```

Following is the content of Database class **DBHelper.java**

```
package com.example.sairamkrishna.myapplication;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Hashtable;
import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.DatabaseUtils;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteDatabase;

public class DBHelper extends SQLiteOpenHelper {

   public static final String DATABASE_NAME = "MyDBName.db";
   public static final String CONTACTS_TABLE_NAME = "contacts";
   public static final String CONTACTS_COLUMN_ID = "id";
   public static final String CONTACTS_COLUMN_NAME = "name";
   public static final String CONTACTS_COLUMN_EMAIL = "email";
   public static final String CONTACTS_COLUMN_STREET = "street";
   public static final String CONTACTS_COLUMN_CITY = "place";
   public static final String CONTACTS_COLUMN_PHONE = "phone";
   private HashMap hp;

   public DBHelper(Context context) {
      super(context, DATABASE_NAME , null, 1);
   }

   @Override
   public void onCreate(SQLiteDatabase db) {
      // TODO Auto-generated method stub
      db.execSQL(
         "create table contacts " +
         "(id integer primary key, name text,phone text,email
text, street text,place text)"
      );
```

```
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
        // TODO Auto-generated method stub
        db.execSQL("DROP TABLE IF EXISTS contacts");
        onCreate(db);
    }

    public boolean insertContact (String name, String phone,
String email, String street,String place) {
        SQLiteDatabase db = this.getWritableDatabase();
        ContentValues contentValues = new ContentValues();
        contentValues.put("name", name);
        contentValues.put("phone", phone);
        contentValues.put("email", email);
        contentValues.put("street", street);
        contentValues.put("place", place);
        db.insert("contacts", null, contentValues);
        return true;
    }

    public Cursor getData(int id) {
        SQLiteDatabase db = this.getReadableDatabase();
        Cursor res =  db.rawQuery( "select * from contacts where
id="+id+"", null );
        return res;
    }

    public int numberOfRows(){
        SQLiteDatabase db = this.getReadableDatabase();
        int numRows = (int) DatabaseUtils.queryNumEntries(db,
CONTACTS_TABLE_NAME);
        return numRows;
    }

    public boolean updateContact (Integer id, String name, String
phone, String email, String street,String place) {
        SQLiteDatabase db = this.getWritableDatabase();
        ContentValues contentValues = new ContentValues();
        contentValues.put("name", name);
        contentValues.put("phone", phone);
        contentValues.put("email", email);
        contentValues.put("street", street);
        contentValues.put("place", place);
        db.update("contacts", contentValues, "id = ? ", new
String[] { Integer.toString(id) } );
        return true;
    }

    public Integer deleteContact (Integer id) {
        SQLiteDatabase db = this.getWritableDatabase();
```

```
      return db.delete("contacts",
      "id = ? ",
      new String[] { Integer.toString(id) });
   }

   public ArrayList<String> getAllCotacts() {
      ArrayList<String> array_list = new ArrayList<String>();

      //hp = new HashMap();
      SQLiteDatabase db = this.getReadableDatabase();
      Cursor res =  db.rawQuery( "select * from contacts", null
);
      res.moveToFirst();

      while(res.isAfterLast() == false){

array_list.add(res.getString(res.getColumnIndex(CONTACTS_COLUMN_N
AME)));
         res.moveToNext();
      }
      return array_list;
   }
}
```

Following is the content of the **res/layout/activity_main.xml**

```
<?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:paddingLeft="@dimen/activity_horizontal_margin"
   android:paddingRight="@dimen/activity_horizontal_margin"
   android:paddingTop="@dimen/activity_vertical_margin"
   android:paddingBottom="@dimen/activity_vertical_margin"
tools:context=".MainActivity">

   <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/textView"
      android:layout_alignParentTop="true"
      android:layout_centerHorizontal="true"
      android:textSize="30dp"
      android:text="Data Base" />

   <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Tutorials Point"
      android:id="@+id/textView2"
      android:layout_below="@+id/textView"
      android:layout_centerHorizontal="true"
```

```
      android:textSize="35dp"
      android:textColor="#ff16ff01" />

   <ImageView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/imageView"
      android:layout_below="@+id/textView2"
      android:layout_centerHorizontal="true"
      android:src="@drawable/logo"/>

   <ScrollView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/scrollView"
      android:layout_below="@+id/imageView"
      android:layout_alignParentLeft="true"
      android:layout_alignParentStart="true"
      android:layout_alignParentBottom="true"
      android:layout_alignParentRight="true"
      android:layout_alignParentEnd="true">

      <ListView
         android:id="@+id/listView1"
         android:layout_width="match_parent"
         android:layout_height="wrap_content"
         android:layout_centerHorizontal="true"
         android:layout_centerVertical="true" >
      </ListView>

   </ScrollView>

</RelativeLayout>
```

Following is the content of the **res/layout/activity_display_contact.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:id="@+id/scrollView1"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   tools:context=".DisplayContact" >

   <RelativeLayout
      android:layout_width="match_parent"
      android:layout_height="370dp"
      android:paddingBottom="@dimen/activity_vertical_margin"
      android:paddingLeft="@dimen/activity_horizontal_margin"
      android:paddingRight="@dimen/activity_horizontal_margin"
      android:paddingTop="@dimen/activity_vertical_margin">

      <EditText
```

```
        android:id="@+id/editTextName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_marginTop="5dp"
        android:layout_marginLeft="82dp"
        android:ems="10"
        android:inputType="text" >
    </EditText>

    <EditText
        android:id="@+id/editTextEmail"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/editTextStreet"
        android:layout_below="@+id/editTextStreet"
        android:layout_marginTop="22dp"
        android:ems="10"
        android:inputType="textEmailAddress" />

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBottom="@+id/editTextName"
        android:layout_alignParentLeft="true"
        android:text="@string/name"

android:textAppearance="?android:attr/textAppearanceMedium" />

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/editTextCity"
        android:layout_alignParentBottom="true"
        android:layout_marginBottom="28dp"
        android:onClick="run"
        android:text="@string/save" />

    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBottom="@+id/editTextEmail"
        android:layout_alignLeft="@+id/textView1"
        android:text="@string/email"

android:textAppearance="?android:attr/textAppearanceMedium" />

    <TextView
        android:id="@+id/textView5"
        android:layout_width="wrap_content"
```

```
         android:layout_height="wrap_content"
         android:layout_alignBottom="@+id/editTextPhone"
         android:layout_alignLeft="@+id/textView1"
         android:text="@string/phone"

android:textAppearance="?android:attr/textAppearanceMedium" />

      <TextView
         android:id="@+id/textView4"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:layout_above="@+id/editTextEmail"
         android:layout_alignLeft="@+id/textView5"
         android:text="@string/street"

android:textAppearance="?android:attr/textAppearanceMedium" />

      <EditText
         android:id="@+id/editTextCity"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:layout_alignRight="@+id/editTextName"
         android:layout_below="@+id/editTextEmail"
         android:layout_marginTop="30dp"
         android:ems="10"
         android:inputType="text" />

      <TextView
         android:id="@+id/textView3"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:layout_alignBaseline="@+id/editTextCity"
         android:layout_alignBottom="@+id/editTextCity"
         android:layout_alignParentLeft="true"
         android:layout_toLeftOf="@+id/editTextEmail"
         android:text="@string/country"

android:textAppearance="?android:attr/textAppearanceMedium" />

      <EditText
         android:id="@+id/editTextStreet"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:layout_alignLeft="@+id/editTextName"
         android:layout_below="@+id/editTextPhone"
         android:ems="10"
         android:inputType="text" >

         <requestFocus />
      </EditText>

      <EditText
         android:id="@+id/editTextPhone"
```

```
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:layout_alignLeft="@+id/editTextStreet"
         android:layout_below="@+id/editTextName"
         android:ems="10"
         android:inputType="phone|text" />

   </RelativeLayout>
</ScrollView>
```

Following is the content of the **res/value/string.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="app_name">Address Book</string>
   <string name="action_settings">Settings</string>
   <string name="hello_world">Hello world!</string>
   <string name="Add_New">Add New</string>
   <string name="edit">Edit Contact</string>
   <string name="delete">Delete Contact</string>
   <string
name="title_activity_display_contact">DisplayContact</string>
   <string name="name">Name</string>
   <string name="phone">Phone</string>
   <string name="email">Email</string>
   <string name="street">Street</string>
   <string name="country">City/State/Zip</string>
   <string name="save">Save Contact</string>
   <string name="deleteContact">Are you sure, you want to delete
it.</string>
   <string name="yes">Yes</string>
   <string name="no">No</string>
</resources>
```

Following is the content of the **res/menu/main_menu.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
>

   <item android:id="@+id/item1"
      android:icon="@drawable/add"
      android:title="@string/Add_New" >
   </item>

</menu>
```

Following is the content of the **res/menu/display_contact.xml**

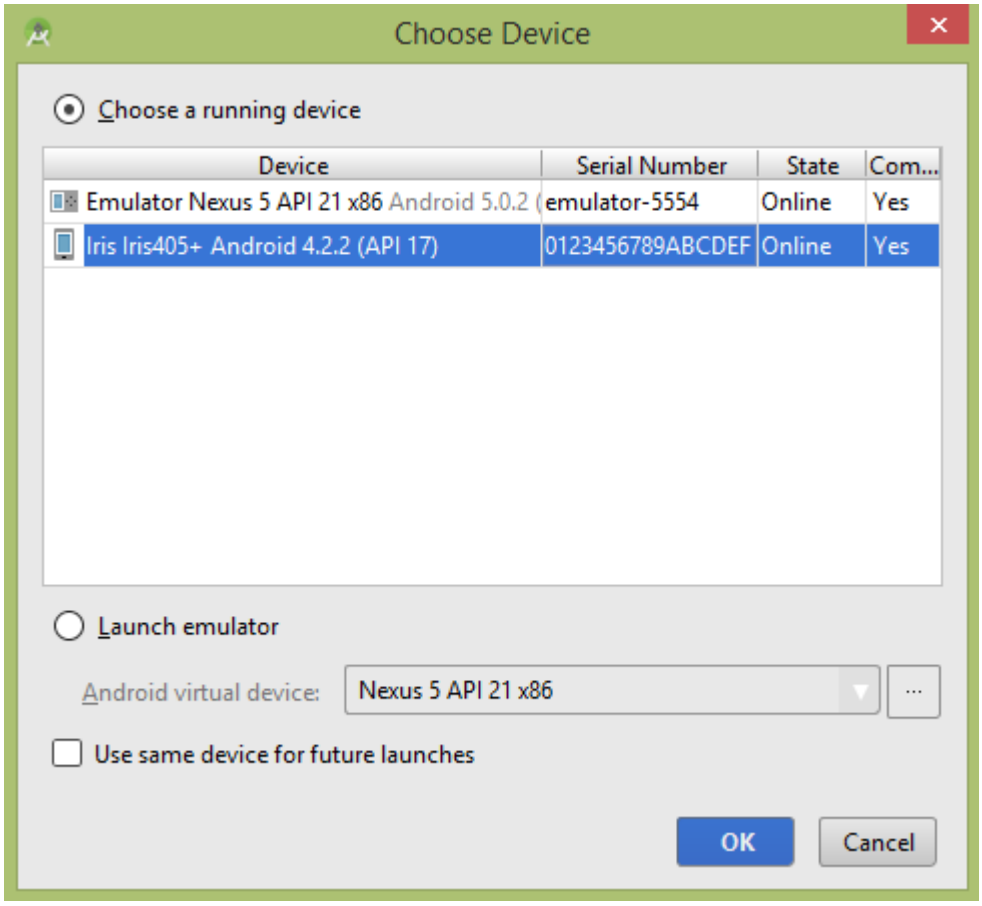
```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
>
   <item
      android:id="@+id/Edit_Contact"
      android:orderInCategory="100"
```

```
      android:title="@string/edit"/>

   <item
      android:id="@+id/Delete_Contact"
      android:orderInCategory="100"
      android:title="@string/delete"/>

</menu>
```

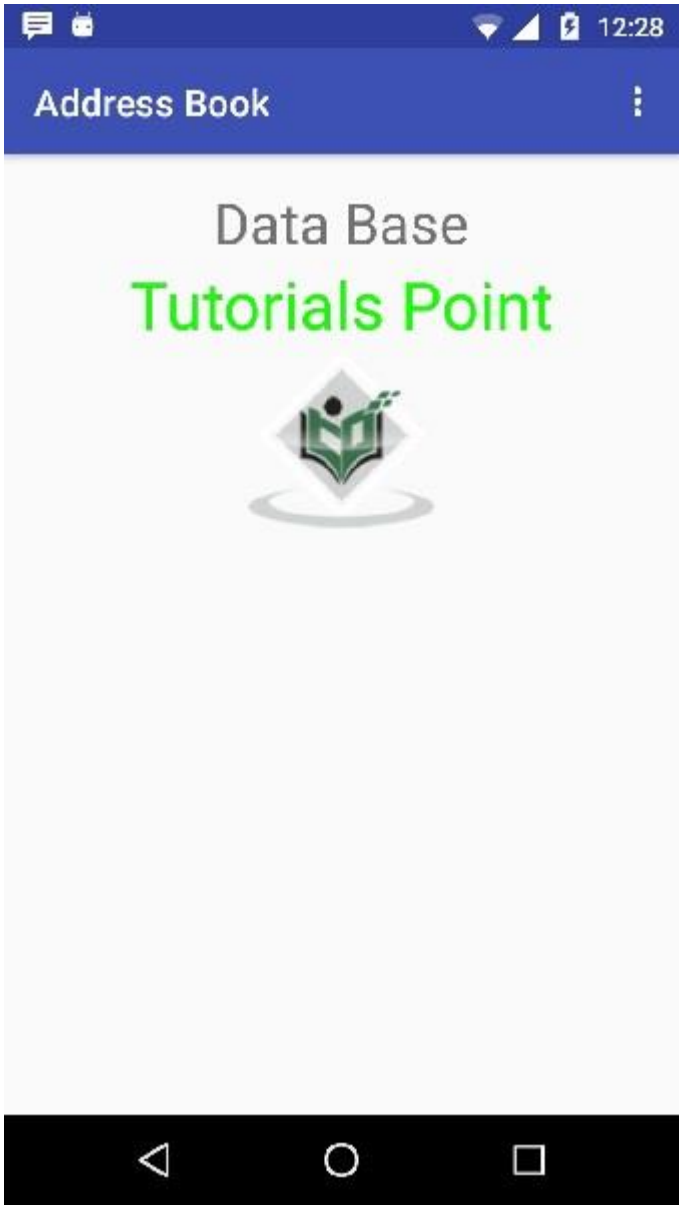
This is the defualt **AndroidManifest.xml** of this project

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.sairamkrishna.myapplication" >

   <application
      android:allowBackup="true"
      android:icon="@mipmap/ic_launcher"
      android:label="@string/app_name"
      android:theme="@style/AppTheme" >

      <activity
         android:name=".MainActivity"
         android:label="@string/app_name" >

         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category
android:name="android.intent.category.LAUNCHER" />
         </intent-filter>

      </activity>

      <activity android:name=".DisplayContact"/>

   </application>
</manifest>
```

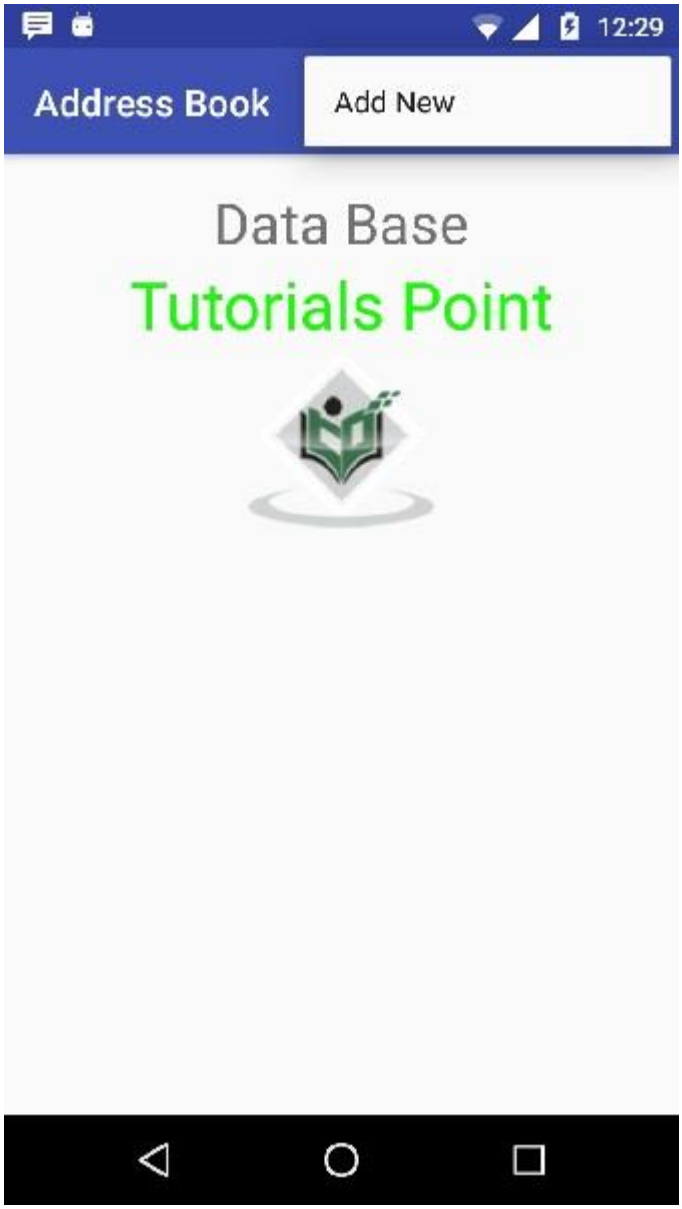
Let's try to run your application. I assume you have connected your actual Android Mobile device with your computer. To run the app from Android studio , open one of your project's activity files and click Run  icon from the tool bar. Before starting your application,Android studio will display following window to select an option where you want to run your Android application.



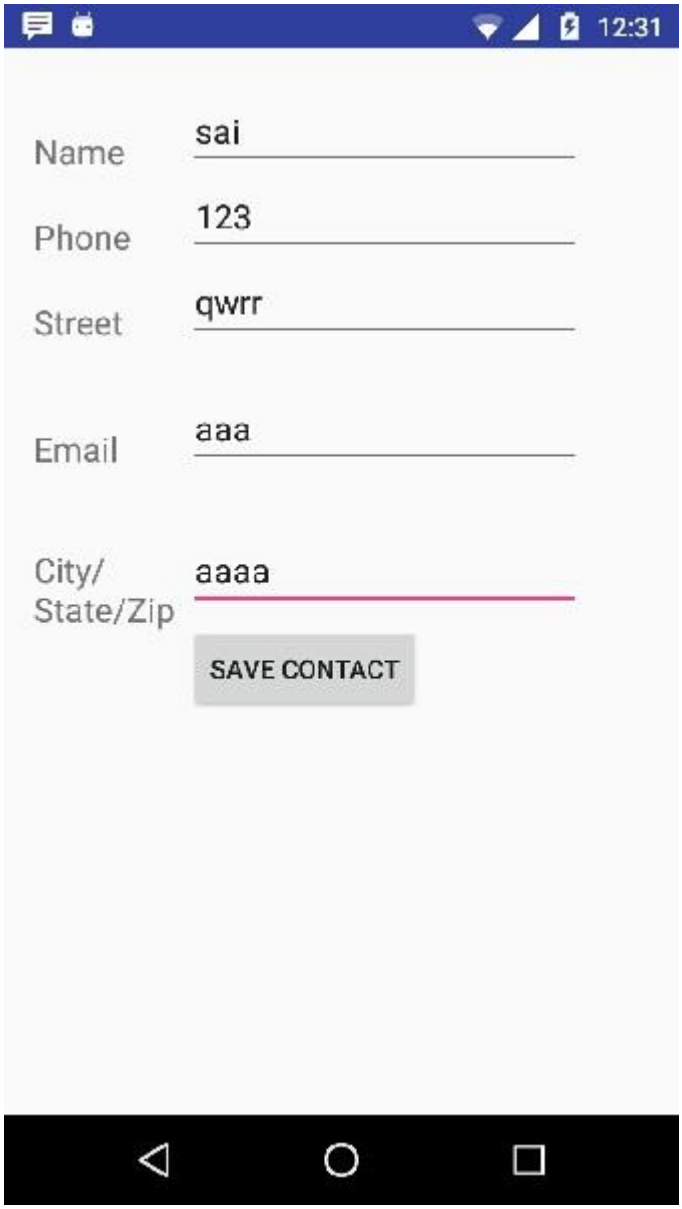
Select your mobile device as an option and then check your mobile device which will display following screen −



Now open your optional menu, it will show as below image: **Optional menu appears different places on different versions**



Click on the add button of the menu screen to add a new contact. It will display the following screen −



It will display the following fields. Please enter the required information and click on save contact. It will bring you back to main screen.

Now our contact sai has been added.In order to see that where is your database is created. Open your android studio, connect your mobile. Go **tools/android/android device monitor**. Now browse the file explorer tab. Now browse this folder **/data/data/<your.package.name>/databases<database-name>.**

ervices overview

A `Service` is an application component that can perform long-running operations in the background, and it doesn't provide a user interface. Another application component can start a service, and it continues to run in the background even if the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

These are the three different types of services:

Foreground

A foreground service performs some operation that is noticeable to the user. For example, an audio app would use a foreground service to play an audio track. Foreground services must display a Notification. Foreground services continue running even when the user isn't interacting with the app.

Note: The WorkManager API offers a flexible way of scheduling tasks, and is able to run these jobs as foreground services if needed. In many cases, using WorkManager is preferable to using foreground services directly.

Background

A background service performs an operation that isn't directly noticed by the user. For example, if an app used a service to compact its storage, that would usually be a background service.

Note: If your app targets API level 26 or higher, the system imposes restrictions on running background services when the app itself isn't in the foreground. In most situations, for example, you shouldn't access location information from the background. Instead, schedule tasks using WorkManager.

Bound

A service is *bound* when an application component binds to it by calling `bindService()`. A bound service offers a client-server interface that allows components to interact with the service, send requests, receive results, and even do so across processes with interprocess communication (IPC). A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

Although this documentation generally discusses started and bound services separately, your service can work both ways—it can be started (to run indefinitely) and also allow binding. It's simply a matter of whether you implement a couple of callback methods: `onStartCommand()` to allow components to start it and `onBind()` to allow binding.

Regardless of whether your service is started, bound, or both, any application component can use the service (even from a separate application) in the same way that any component can use an activity—by starting it with an `Intent`. However, you can declare the service as *private* in the manifest file and block access from other applications. This is discussed more in the section about Declaring the service in the manifest.