

# Unit - II

Building Blocks and Databases

# Unit- II - Building Blocks and Databases

- 1. Introduction to Activity and Intents
- 2. Understanding Activity Life Cycle
- 3. Linking Activities
- 4. Passing Data
- 5. Toast
- 6. Displaying Dialog Window
- 7. Notifications
- 8. Services
- 9. Broadcast Receivers
- 10. Content Provider
- 11. SQLite – Database
- 12. Publish App in Playstore
- 13. Sample Applications

# 1. Introduction to Activity and Intents

- An Activity is an application component that provides **a screen with which users can interact** in order to do something, such as dial the phone, take a photo, send an email, or view a map.
- Each activity is given a window in which to draw its **user interface**.
- The **window typically fills the screen**, but may be smaller than the screen and float on top of other windows.

# Introduction to Activity and Intents (con...)

- Intents in android are used as **message passing mechanism** that works both within your application and between applications.
- **Three of the core components** of an application — activities, services, and broadcast receivers are activated through messages, called intents
- E.g. Intents can be used to **start an activity** to send email.

## 2. Android Activity Life Cycle

- The steps that an application goes through from **starting to finishing**
- Slightly **different than normal Java life cycle** due to :
  - The difference in the **way Android application are defined**
  - The **limited resources** of the Android hardware platform

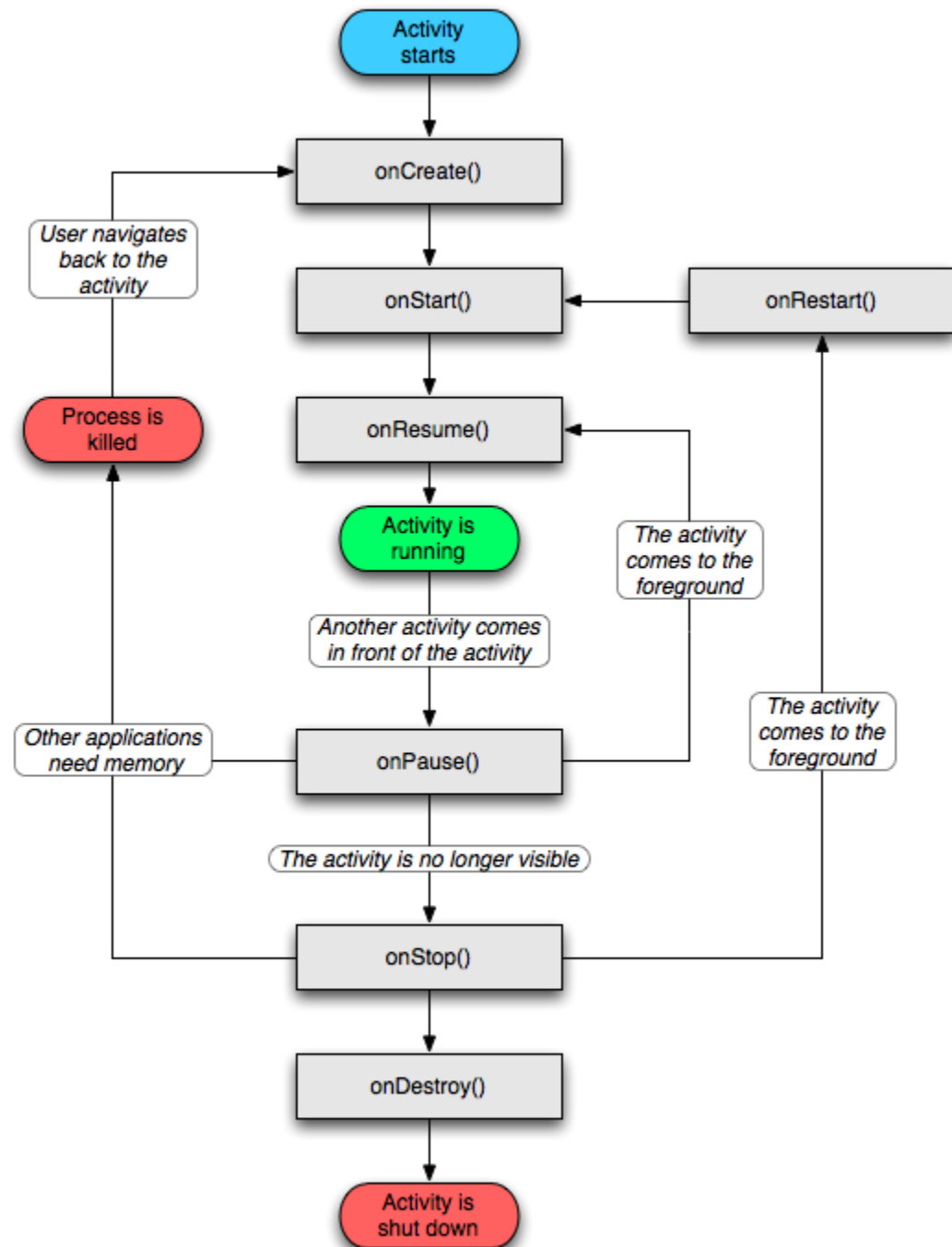
# Activity Life Cycle (con...)

- Each application runs in its own process.
- Each activity of an app is run in the apps process
- Processes are started and stopped as needed to run an apps components.
- Processes may be killed to reclaim needed resources.
- Killed apps may be restored to their last state when requested by the user

# Activity Life Cycle(con...)

- Management of the life cycle is done automatically by the system via the **activity stack**.
- The activity class has the **following method callbacks** to help you manage the app:
  - onCreate()
  - onStart()
  - onResume()
  - onPause()
  - onStop()
  - onRestart()
  - onDestroy()

# Activity Life Cycle





# Example

```
public class MainActivity extends Activity {  
    @Override  
    protected void onCreate(Bundle  
        savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        Toast.makeText(getApplicationContext(), "I am  
onCreate() Method",  
            Toast.LENGTH_SHORT).show();  
    }  
}
```

# Example (con...)

@Override

```
protected void onStart() {  
    Toast.makeText(getApplicationContext(), "I am  
    onStart() method", Toast.LENGTH_SHORT).show();  
    super.onStart();  
}
```

@Override

```
protected void onResume() {  
    Toast.makeText(getApplicationContext(), " I am  
    onResume() method ", Toast.LENGTH_SHORT).show();  
    super.onResume();  
}
```

# Example (con...)

@Override

```
protected void onPause() {  
    Toast.makeText(getApplicationContext(), " I am  
    onPause() method ", Toast.LENGTH_SHORT).show();  
    super.onPause();  
}
```

@Override

```
protected void onStop() {  
    Toast.makeText(getApplicationContext(), " I am  
    onStop() method ", Toast.LENGTH_SHORT).show();  
    super.onStop();  
}
```

# Example (con...)

@Override

```
protected void onRestart() {  
    Toast.makeText(getApplicationContext(), " I am  
    onRestart() method ", Toast.LENGTH_SHORT).show();  
    super.onRestart();  
}
```

@Override

```
protected void onDestroy() {  
    Toast.makeText(getApplicationContext(), " I am  
    onDestroy() method ", Toast.LENGTH_SHORT).show();  
    super.onDestroy();  
}}
```

### 3. Linking Activity

- Android Intent is an **abstract description of an operation** to be performed.
- It can be used with,
  - An **startActivity()** to launch an Activity
  - An **sendBroadcast()** to send it to any interested BroadcastReceiver components
  - An **startService(Intent)** to communicate with a background Service

# Linking Activity (con...)

- Intent Objects

- An Intent object is a **bundle of information** which is used by the component that receives the intent.

- Intent object can contain the following **components**

- **Action**

- A **string naming** the action to be performed.

- The action in an Intent object can be set by the **setAction()** method and read by **getAction()**.

# Linking Activity (con...)

## – Data

- Adds a data specification to an intent filter.
- The `setData()` method specifies data only as a URI, `setType()` specifies it only as a MIME type, and `setDataAndType()` specifies it as both a URI and a MIME type. The URI is read by `getData()` and the type by `getType()`.

# Linking Activity (con...)

## – Category

- The category is an optional part of Intent object and it's a string containing **additional information** about the kind of component that should handle the intent.
- The **addCategory()** method places a category in an Intent object, **removeCategory()** deletes a category previously added



# Linking Activity (con...)

## — Extras

- This will be in **key-value pairs** for additional information that should be delivered to the component handling the intent.
- The extras can be set and read using the **putExtras()** and **getExtras()** methods respectively.

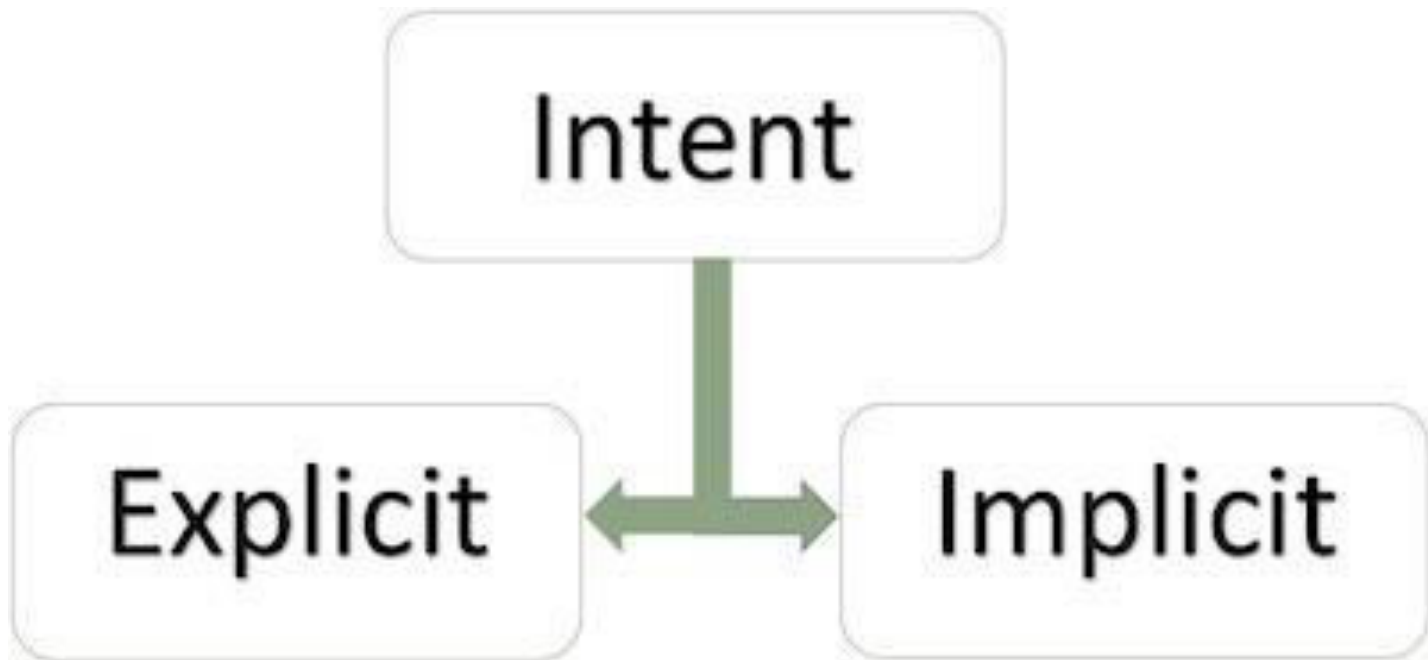
## — Flags

- The flags are **optional** part of Intent object
- It helps to instruct the Android system **how to launch an activity**, and **how to treat it after it's launched** etc.

# Linking Activity (con...)

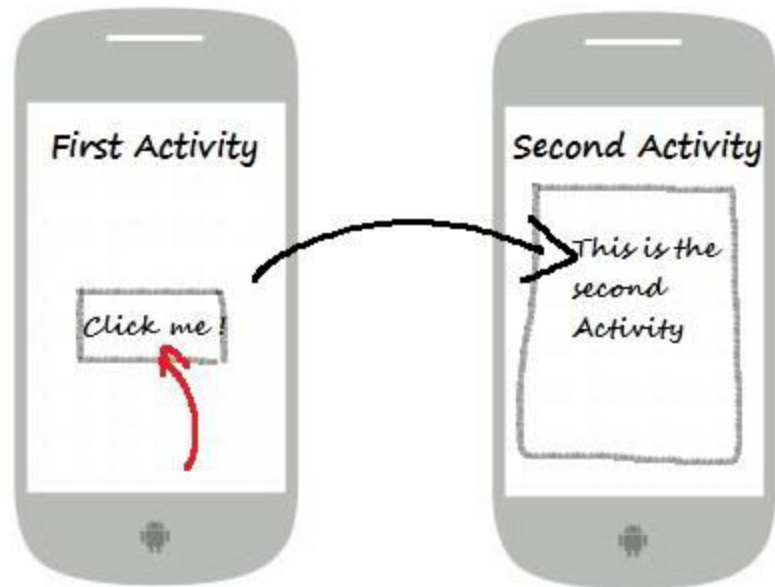
- **Types of Intents**

- There are following two types of intents supported by Android



# Linking Activity (con...)

- Explicit Intents
  - It going to be **connected internal world of application**, i.e. to connect **one activity to another activity**, below image is **connecting first activity to second activity** by clicking the button.



# Linking Activity (con...)

- Example

```
Intent i = new Intent(FirstActivity.this, SecondActivity.class);  
startActivity(i);
```

```
Intent i= new Intent(android.content.Intent.ACTION_VIEW,  
    Uri.parse("http://www.google.com"));  
startActivity(i);
```

```
Intent i= new Intent(android.content.Intent.ACTION_VIEW,  
    Uri.parse("tel:9943005903"));  
startActivity(i);
```

# Linking Activity (con...)

- Implicit Intents
  - These intents **do not name a target** and the field for the component name is left blank.
  - Implicit intents are often used to **activate components in other applications.**

- **Example**

```
Intent read1=new Intent();  
read1.setAction(android.content.Intent.ACTION_VIEW);  
read1.setData(ContactsContract.Contacts.CONTENT_URI);  
startActivity(read1);
```

## 4. Passing Data

- Activity is used to represent the **data to user and allows user interaction.**
- In an android application, we can have **multiple activities** and that can interact with each other.
- During activity interaction we might required **to pass data from one activity to other.**

# Passing Data (con...)

- Data is passed as **extras and are key/value pairs**.
- The **key** is always a **String** and the **value** you can use the **primitive data types** int, float, chars, etc.
- **Syntax for sending data**

```
Intent intent = new Intent(context,  
YourActivityClass.class);
```

```
intent.putExtra(KEY, <your value here>);  
startActivity(intent);
```

# Passing Data (con...)

- Syntax for retrieving data

```
Intent intent = getIntent();
```

```
String stringData= intent.getStringExtra(KEY);
```

```
int numberData = intent.getIntExtra(KEY,  
    defaultValue);
```

```
boolean booleanData = intent.getBooleanExtra(KEY,  
    defaultValue);
```

```
char charData = intent.getCharExtra(KEY,  
    defaultValue);
```



# Example

```
public class MainActivity extends Activity implements
    OnClickListener {
    Button btn;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        btn = (Button) findViewById(R.id.btnPassData);
        btn.setOnClickListener(this);
    }
```

# Example (con...)

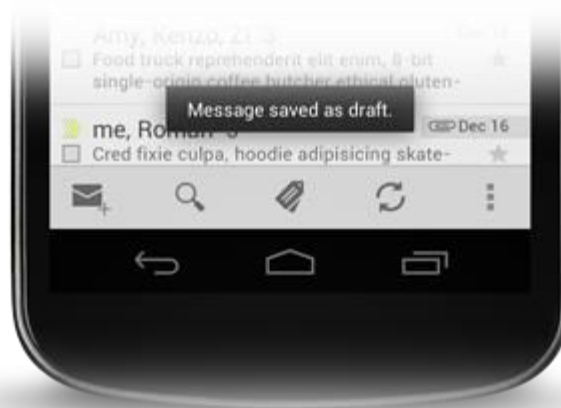
```
@Override  
public void onClick(View view) {  
    Intent intent = new  
Intent(getApplicationContext(), SecondActivity.class);  
    intent.putExtra("message", "Hello From  
MainActivity");  
startActivity(intent);  
}  
}
```

# Example (con...)

```
public class SecondActivity extends Activity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_another);  
        Intent intent = getIntent();  
        String msg = intent.getStringExtra("message");  
        Toast toast = Toast.makeText(this, msg,  
        Toast.LENGTH_LONG);  
        toast.show();  
    }  
}
```

# 5. Toasts

- A **toast provides simple feedback** about an operation in a small popup.
- It only **fills the amount of space required for the message** and the current activity remains visible and interactive
- Toasts automatically **disappear after a timeout.**



# Toasts (con...)

- First, instantiate a **Toast** object with one of the **makeText()** methods.
- This method takes three parameters: the application **Context**, the text message, and the duration for the toast. It returns a properly initialized Toast object.
- You can display the toast notification with **show()**.

# Toasts (con...)

```
Context context = getApplicationContext();  
CharSequence text = "Hello toast!";  
int duration = Toast.LENGTH_SHORT;  
Toast toast = Toast.makeText(context, text, duration);  
toast.show();
```

(or)

```
Toast.makeText(context, text, duration).show();
```

(or)

```
Toast.makeText(getApplicationContext(),"Hello  
toast!", Toast.LENGTH_SHORT).show();
```

# Positioning your Toast

- A standard toast notification appears near the bottom of the screen, centered horizontally.
- You can change this position with the `setGravity(int, int, int)` method.
- This accepts three parameters: a Gravity constant, an x-position offset, and a y-position offset.
- Exmample  
`toast.setGravity(Gravity.TOP | Gravity.LEFT, 0, 0);`

# Custom Toast

- To create a **customized layout** for your toast notification.
- To create a custom layout, define a View layout, in XML or in your application code, and pass the root **View** object to the **setView(View)** method.



# Example

<LinearLayout

    android:id="@+id/toast\_layout\_root"

    android:orientation="horizontal"

    android:layout\_width="fill\_parent"

    android:layout\_height="fill\_parent" >

    <TextView

        android:id="@+id/text"

        android:layout\_width="wrap\_content"

        android:layout\_height="wrap\_content" />

</LinearLayout>

# Example (con...)

```
LayoutInflater inflater = getLayoutInflater();  
    View layout = inflater.inflate(R.layout.custom_toast,  
                                (ViewGroup)  
    findViewById(R.id.toast_layout_root));  
  
    TextView text = (TextView) layout.findViewById(R.id.text);  
    text.setText("This is a custom toast");  
  
    Toast toast = new Toast(getApplicationContext());  
    toast.setGravity(Gravity.CENTER_VERTICAL, 0, 0);  
    toast.setDuration	Toast.LENGTH_LONG);  
    toast.setView(layout);  
    toast.show();
```

## 6. Displaying Dialog Window

- A dialog is a small window that **prompts the user to make a decision** or enter additional information.
- Creating alert dialog is **very easy**.
- The **Dialog** class is the base class for dialogs, but you should avoid instantiating Dialog directly.
- Instead, use one of the following subclass **AlertDialog** class

# Dialog Window (con...)

- **Three regions** of an alert dialog
  - **Title**
    - This is optional and should be used only when the content area is occupied by a detailed message.
  - **Content area**
    - This can display a message.
  - **Action buttons**
    - There should be no more than three action buttons in a dialog.

# Dialog Window (con...)

- Different action buttons

- Positive

- Use this to accept and continue with the action (the "OK" action).

- Negative

- Use this to cancel the action.

- Neutral

- 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."

# Dialog Window (con...)

- Different alert dialogue methods
  - one button(ok button) - `setPositiveButton()`
  - two buttons(yes or no buttons) - `setNegativeButton()`
  - three buttons(yes, no and cancel buttons) - `setNeutralButton()`

# Example

```
AlertDialog.Builder alertDialog = new  
    AlertDialog.Builder(AlertDialogActivity.this);  
    // Setting Dialog Title  
    alertDialog.setTitle("Confirm Delete...");  
    // Setting Dialog Message  
    alertDialog.setMessage("Are you sure you want delete  
this?");  
    // Setting Icon to Dialog  
    alertDialog.setIcon(R.drawable.delete);
```

# Example (con...)

```
// Setting Positive "Yes" Button
```

```
    alertDialog.setPositiveButton("YES", new  
    DialogInterface.OnClickListener() {  
        public void onClick(DialogInterface dialog,int  
        which) {
```

```
            // Write your code here to invoke YES event
```

```
            Toast.makeText(getApplicationContext(), "You  
            clicked on YES", Toast.LENGTH_SHORT).show();  
        }  
    });
```



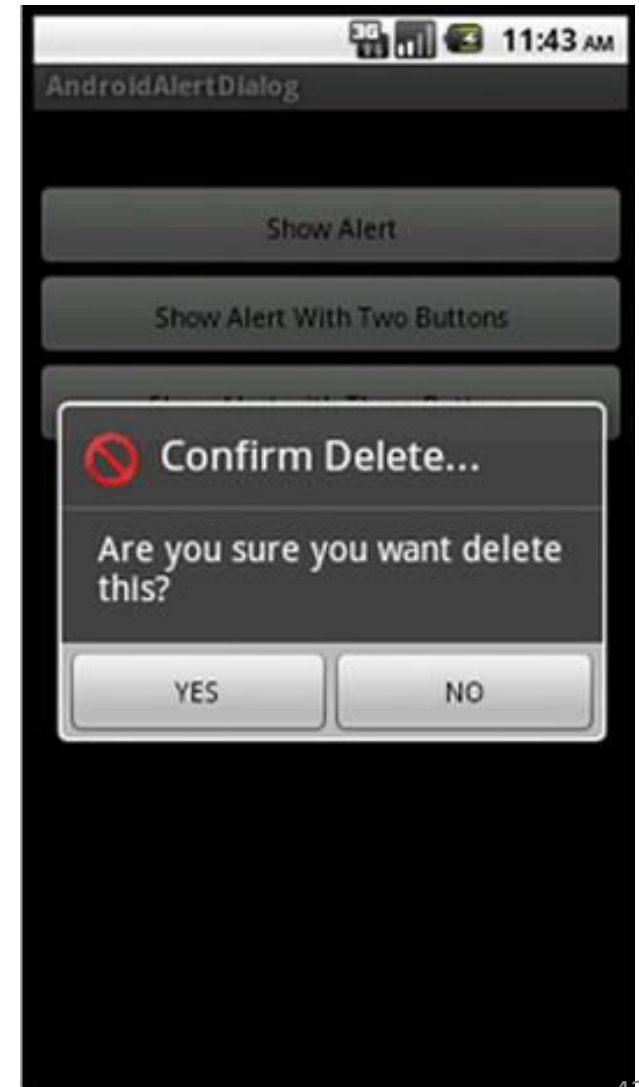
# Example (con...)

```
// Setting Negative "NO" Button
```

```
    alertDialog.setNegativeButton("NO", new  
    DialogInterface.OnClickListener() {  
        public void onClick(DialogInterface dialog, int which) {  
            // Write your code here to invoke NO event  
            Toast.makeText(getApplicationContext(), "You clicked  
on NO", Toast.LENGTH_SHORT).show();  
            dialog.cancel();  
        }  
    });
```

# Output

```
// Showing Alert Message  
alertDialog.show();
```



# 7. Notification

- A **notification is a message** you can display to the user outside of your application's normal UI.
- When you tell the system to issue a notification, it first appears as an icon in the **notification area**.
- To see the details of the notification, the user opens the **notification drawer**.
- Both the notification area and the notification drawer are system-controlled areas **that the user can view at any time**.
- Android **Toast class provides a handy way to show users alerts** but problem is that these alerts are not persistent which means **alert flashes on the screen for a few seconds** and then disappears.

# Step 1 - Create Notification Builder

- A first step is to create a notification builder using `NotificationCompat.Builder.build()`.
- Use Notification Builder to set various Notification properties like its small and large icons, title, priority etc.
- Syntax

```
NotificationCompat.Builder mBuilder = new  
NotificationCompat.Builder(this);
```

# Step 2 - Setting Notification Properties

- To set its **Notification properties** using Builder object as per your requirement.
  - A small icon, set by **setSmallIcon()**
  - A title, set by **setContentTitle()**
  - Detail text, set by **setContentText()**

- Example

```
mBuilder.setSmallIcon(R.drawable.notification_icon);  
mBuilder.setContentTitle("Notification Alert, Click Me!");  
mBuilder.setContentText("Hi, This is Android Notification  
Detail!");
```

## Step 3 - Attach Actions

- The action is defined by a **PendingIntent** containing an **Intent** that starts an Activity in your application.
- A PendingIntent object helps you to perform an action on your applications behalf, often at a later time, without caring of whether or not your application is running.
- We take help of **stack builder object** which will contain an artificial back stack for the started Activity.
- This ensures that navigating backward from the Activity leads out of your application to the Home screen.

## Step 3 - Attach Actions (con...)

```
Intent resultIntent = new Intent(this,  
    ResultActivity.class);
```

```
TaskStackBuilder stackBuilder =  
    TaskStackBuilder.create(this);
```

```
stackBuilder.addParentStack(MainActivity.this);
```

```
stackBuilder.addNextIntent(resultIntent);
```

```
PendingIntent resultPendingIntent =  
    stackBuilder.getPendingIntent(0, PendingIntent.FLAG  
        _UPDATE_CURRENT);
```

```
mBuilder.setContentIntent(resultPendingIntent);
```

# Step 4 - Issue the notification

- Finally, you pass the Notification object to the system by calling `NotificationManager.notify()` to send your notification.
- Make sure you call `NotificationCompat.Builder.build()` method on builder object before notifying it.
- Example

```
NotificationManager mNotificationManager =  
    (NotificationManager)  
        getSystemService(Context.NOTIFICATION_SERVICE)  
mNotificationManager.notify(notificationID, mBuilder.build());
```



# Example

```
Button b;
```

```
b=(Button)findViewById(R.id.notify_btn);
```

```
    b.setOnClickListener(new View.OnClickListener() {
```

```
@Override
```

```
public void onClick(View v) {
```

```
// TODO Auto-generated method stub
```

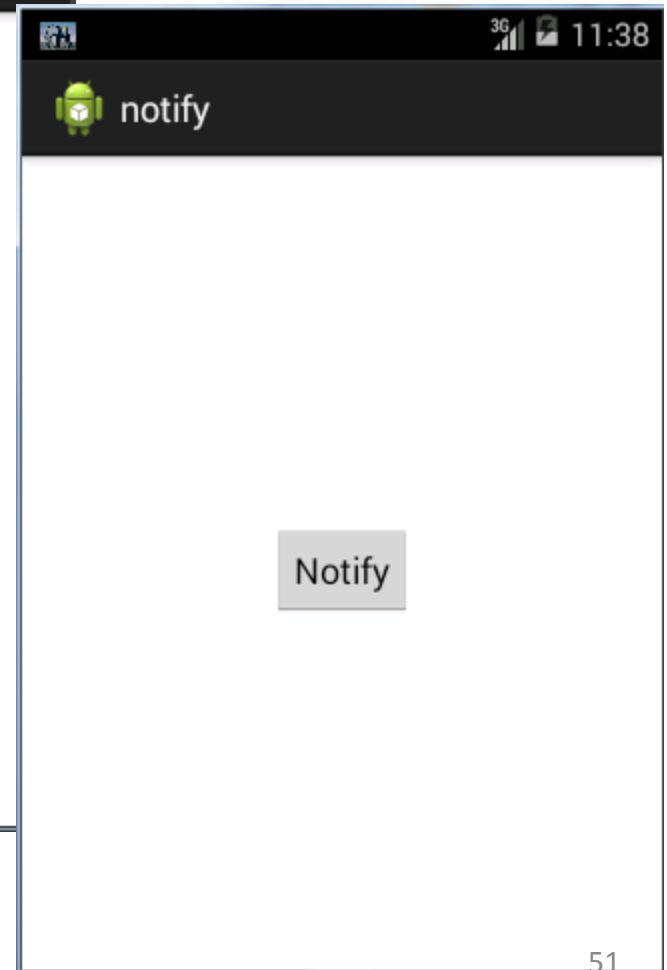
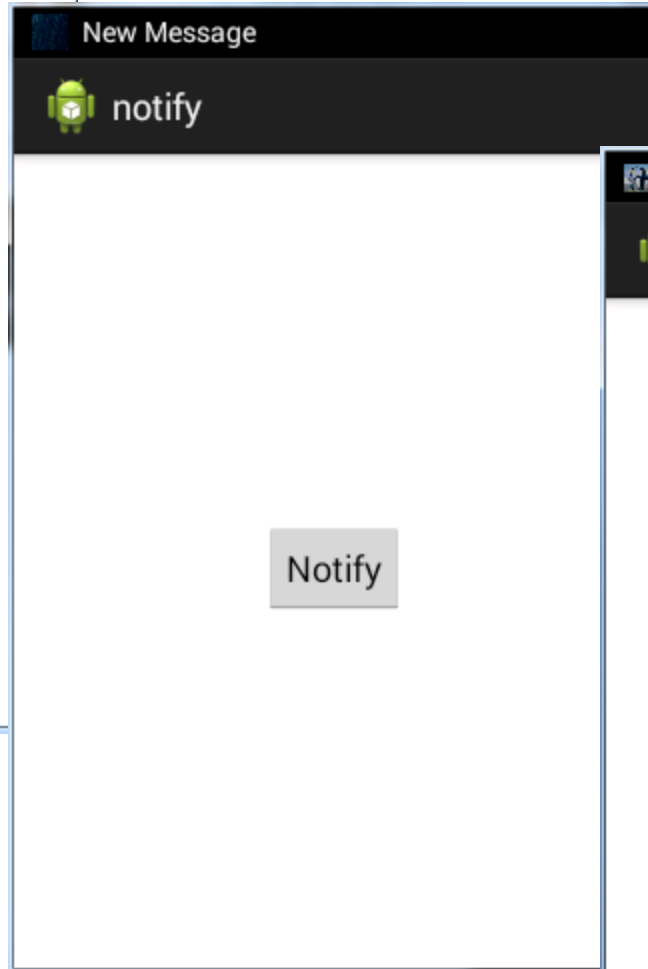
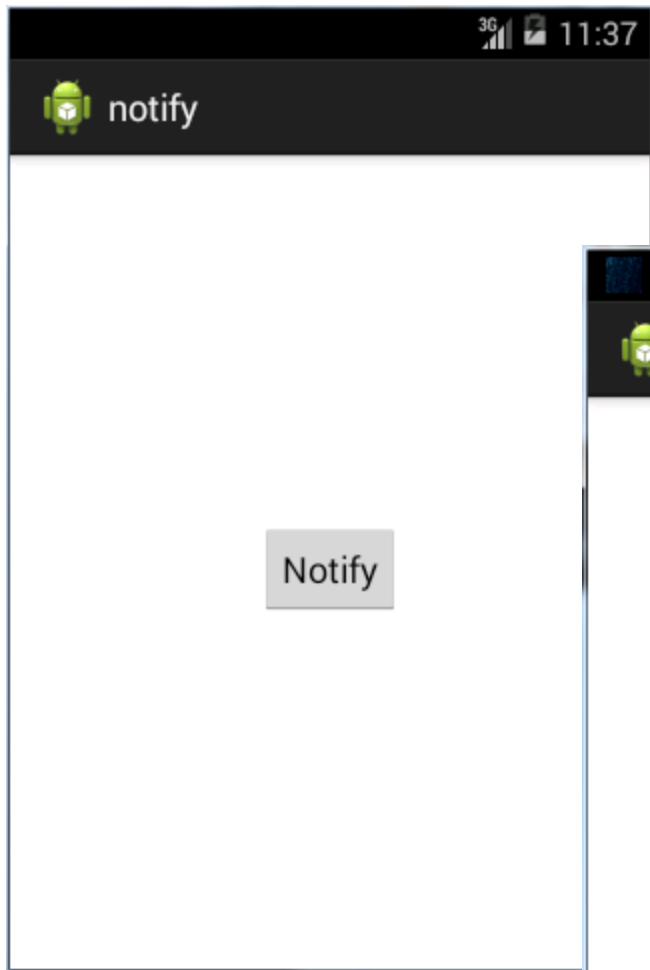
```
Notify_method("Test notify message");
```

```
}
```

# Example (con...)

```
private void Notify_method(String string) {  
    NotificationManager notificationManager = (NotificationManager)  
        getSystemService(NOTIFICATION_SERVICE);  
    Notification notification = new Notification(R.drawable.abc, "New Message",  
        System.currentTimeMillis());  
    Intent notificationIntent = new  
        Intent(MainActivity.this, NotifyDisplay.class);  
    PendingIntent pendingIntent = PendingIntent.getActivity(MainActivity.this,  
        0, notificationIntent, 0);  
    notification.setLatestEventInfo(MainActivity.this, "Notification", string,  
pendingIntent);  
    notificationManager.notify(9999, notification);  
}  
});
```

# Output



# 8. Services

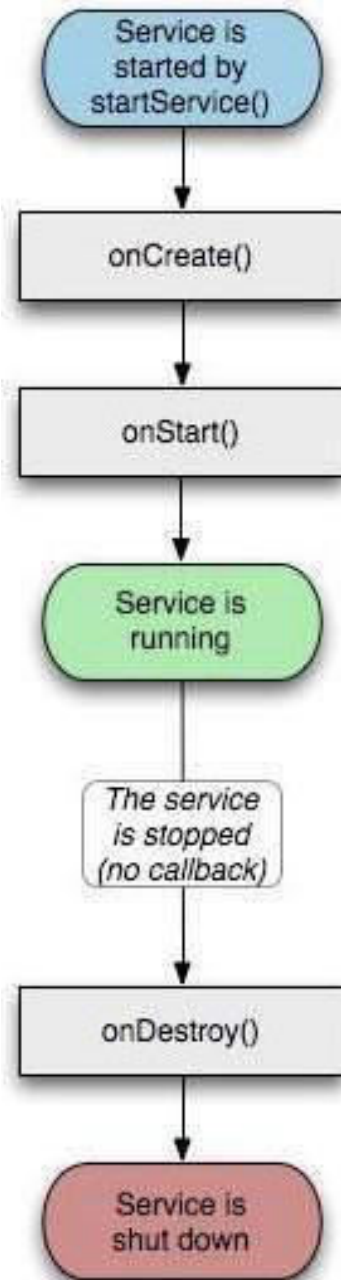
- A **service is a component that runs in the background** to perform long-running operations without needing to interact with the user and it works even if application is destroyed.
- A service can essentially take **two states**
  - **Started**
    - A service is **started** when an application component, such as an activity, starts it by calling ***startService()***.
    - Once started, a service can **run in the background indefinitely**, even if the component that started it is destroyed.

# Services (con...)

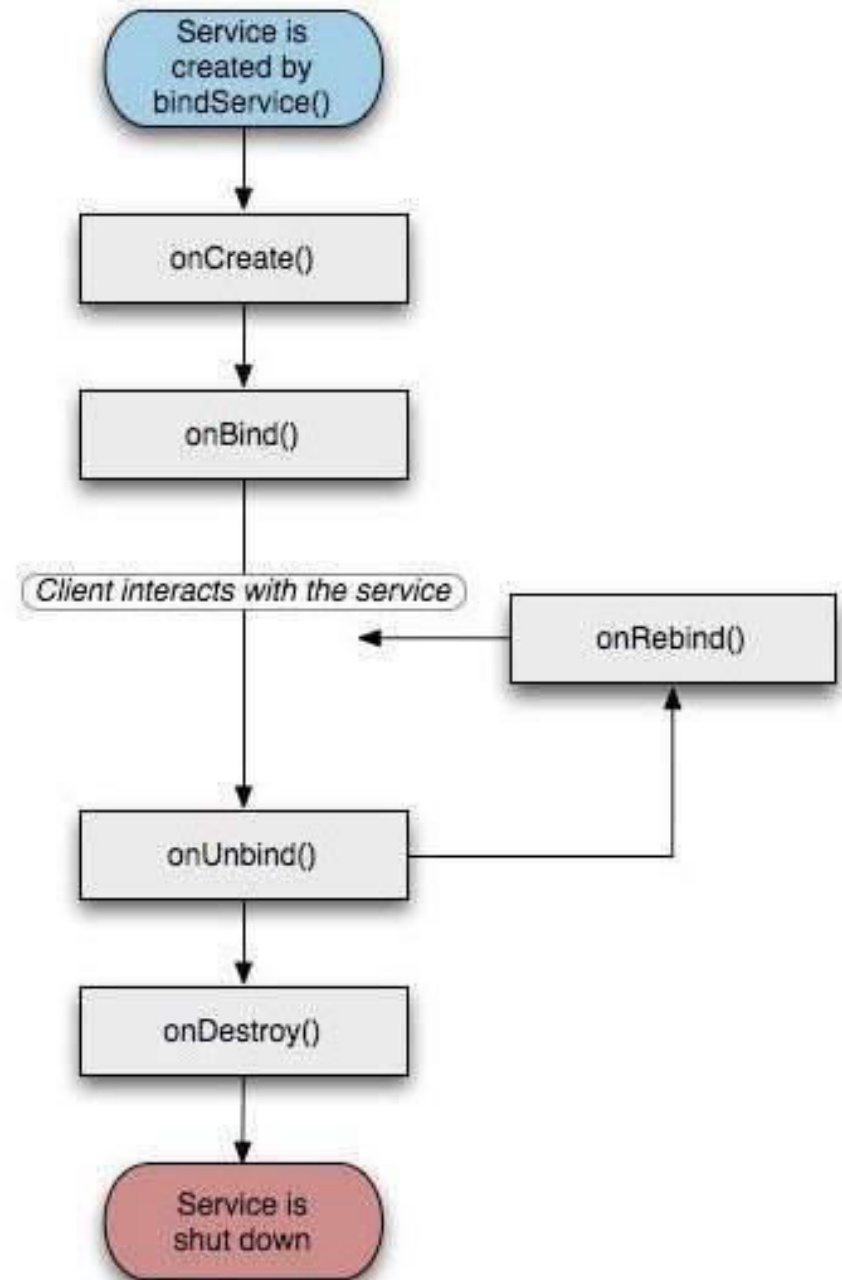
## – Bound

- 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, get results, and even do so across processes with inter-process communication (IPC).
- A service **has life cycle callback methods** that you can implement to monitor changes in the service's state and you can perform work at the appropriate stage

# Service Life Cycle



UN Bounded Service



Bounded services

# Services (con...)

- To create an service, you **create a Java class that extends the Service** base class or one of its existing subclasses.
- The **Service base class defines various callback methods** and the most important are given below.
- Don't need to implement all the callbacks methods

# Services (con...)

- **onStartCommand()**
  - The system calls this method when another component, such as an activity, requests that the service be started, by calling *startService()*.
  - If you implement this method, it is your responsibility to stop the service when its work is done, by calling *stopSelf()* or *stopService()* methods.
- **onBind()**
  - The system calls this method when another component wants to bind with the service by calling *bindService()*



# Services (con...)

- **onUnbind()**
  - The system calls this method **when all clients have disconnected** from a particular interface published by the service.
- **onRebind()**
  - The system calls this method **when new clients have connected to the service**, after it had previously been notified that all had disconnected in its *onUnbind(Intent)*.

# Services (con...)

- **onCreate()**
  - The system calls this method **when the service is first created** using *onStartCommand()* or *onBind()*. This call is required to perform one-time set-up.
- **onDestroy()**
  - The system calls this method **when the service is no longer used** and is being destroyed.
  - Your service should implement this to clean up any resources such as threads, registered listeners, receivers, etc.

# Declaring a service in the manifest

- Like activities (and other components), you must **declare all services in your application's manifest file.**
- To declare your service, add a **<service> element** as a child of the <application> element.

- Example

```
<manifest ... >
```

```
...
```

```
<application ... >
```

```
    <service android:name=".ExampleService" />
```

```
...
```

```
</application>
```

```
</manifest>
```

# Example

```
public class MainActivity extends Activity {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
    }  
    public void startService(View view) {  
        startService(new Intent(getApplicationContext(), MyService.class));  
    }  
    public void stopService(View view) {  
        stopService(new Intent(getApplicationContext(), MyService.class));  
    }  
}
```

# XML file

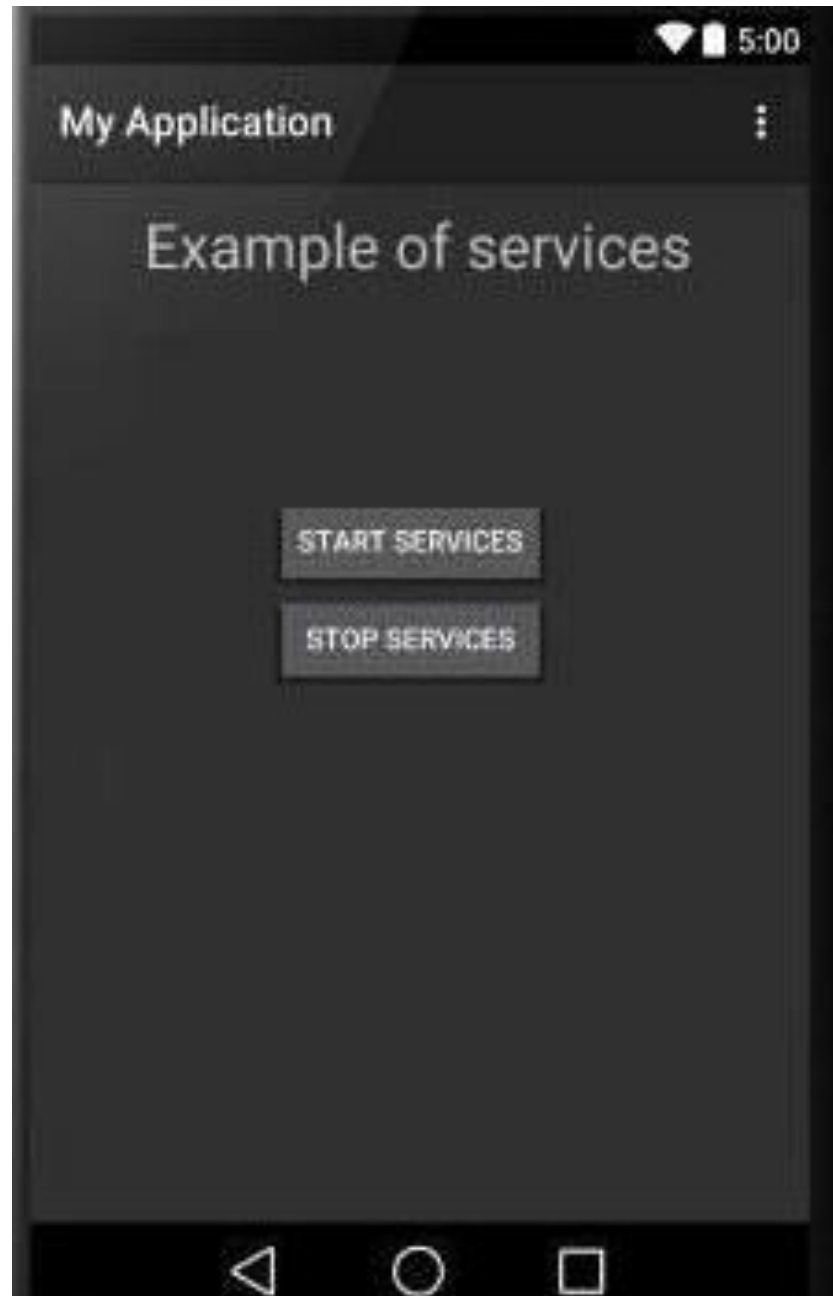
<Button

```
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:id="@+id/button1"  
    android:text="START SERVICES"  
    android:onClick="startService" />
```

<Button

```
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="STOPSERVICES"  
    android:id="@+id/button2"  
    android:onClick="stopService" />
```

# Output



# Example

```
public class MyService extends Service {  
    int mStartMode;    // indicates how to behave if the  
    service is killed  
    IBinder mBinder;    // interface for clients that bind  
    boolean mAllowRebind; // indicates whether onRebind  
    should be used  
    @Override  
    public int onStartCommand(Intent intent, int flags, int  
    startId) {  
        Toast.makeText(this, "Service Started",  
        Toast.LENGTH_LONG).show();  
        return mStartMode;  
    }  
}
```

# Example (con...)

```
@Override
    public IBinder onBind(Intent intent) {
        // A client is binding to the service with
        onBindService()
        return mBinder;
    }
    @Override
    public boolean onUnbind(Intent intent) {
        // All clients have unbound with unbindService()
        return mAllowRebind;
    }
```



# Example (con...)

```
@Override
```

```
public void onRebind(Intent intent) {  
    // A client is binding to the service with bindService(),  
    // after onUnbind() has already been called  
}
```

```
@Override
```

```
public void onDestroy() {  
    // The service is no longer used and is being destroyed  
    Toast.makeText(this, "Service Destroyed",  
        Toast.LENGTH_LONG).show();  
}
```

# 9. Broadcast Receivers

- Simply **respond to broadcast messages** from other applications or from the system itself.
- These messages are sometime **called events or intents**.
- Broadcast receiver who will **intercept this communication and will initiate appropriate action**.
- **Three important steps**
  - Creating the Broadcast Receiver.
  - Registering Broadcast Receiver
  - Broadcasting Custom Intents

# Creating the Broadcast Receiver

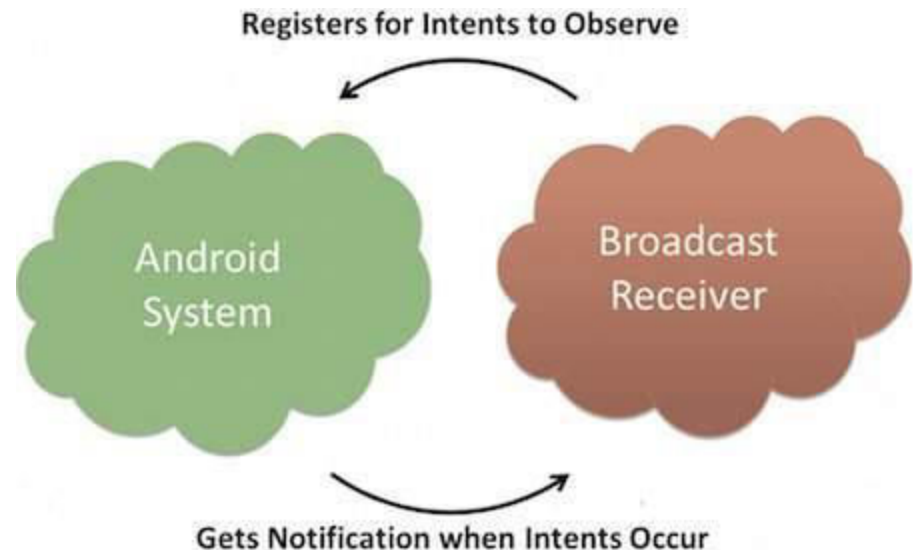
- A broadcast receiver is implemented as a subclass of **BroadcastReceiver** class and overriding the **onReceive()** method where each message is received as a **Intent** object parameter.

- **Example**

```
public class MyReceiver extends BroadcastReceiver {  
    @Override  
    public void onReceive(Context context, Intent intent) {  
        Toast.makeText(context, "Intent Detected.",  
            Toast.LENGTH_LONG).show();  
    }  
}
```

# Registering Broadcast Receiver

- An **application listens** for specific broadcast intents by registering a broadcast receiver in **AndroidManifest.xml** file.
- Consider we are going to register **MyReceiver** for system generated event **ACTION\_BOOT\_COMPLETED** which is fired by the system once the Android system has completed the boot process.



## ...Broadcast Receiver (con...)

- To modify in **AndroidManifest.xml** file

```
<receiver android:name="MyReceiver">  
  <intent-filter>  
    <action  
      android:name="android.intent.action.BOOT_COMPL  
      ETED">  
    </action>  
  </intent-filter>  
</receiver>
```

# Few important system events

| Event Constant                                     | Description   |
|--|---|
| <code>android.intent.action.BATTERY_CHANGED</code> | Containing the charging state, level, and other information |
| <code>android.intent.action.BATTERY_LOW</code>     | Indicates low battery condition                             |
| <code>android.intent.action.BATTERY_OKAY</code>    | Indicates the battery is now okay after being low.          |
| <code>android.intent.action.BOOT_COMPLETED</code>  | After the system has finished booting.                      |
| <code>android.intent.action.BUG_REPORT</code>      | Show activity for reporting a bug.                          |
| <code>android.intent.action.CALL</code>            | Perform a call to someone specified by the data.            |
| <code>android.intent.action.CALL_BUTTON</code>     | The user pressed the "call" button to go to the dialer      |
| <code>android.intent.action.DATE_CHANGED</code>    | The date has changed.                                       |
| <code>android.intent.action.REBOOT</code>          | Have the device reboot.                                     |

# Broadcasting Custom Intents

- If you **want your application itself should generate and send custom intents** then you will have to create and send those intents by using the ***sendBroadcast()*** method inside your activity class.
- To modify the **AndroidManifest.xml** file

```
<receiver android:name="MyReceiver">  
    <intent-filter>  
        <action  
            android:name="com.example.CUSTOM_INTENT">  
        </action>  
    </intent-filter>  
</receiver>
```

# Example

```
public void broadcastIntent(View view) {  
    Intent intent = new Intent();  
    intent.setAction("com.example.CUSTOM_INTEN  
        T");  
    sendBroadcast(intent);  
}
```



# Example (Overall)

```
public class MainActivity extends Activity {  
    @Override public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
    }  
    // broadcast a custom intent.  
    public void broadcastIntent(View view){  
        Intent intent = new Intent();  
        intent.setAction("com.example.CUSTOM_INTENT");  
        sendBroadcast(intent);  
    }  
}
```

## Example (Overall) (con...)

```
public class MyReceiver extends  
    BroadcastReceiver {  
    @Override public void onReceive(Context  
        context, Intent intent) {  
        Toast.makeText(context, "Intent Detected.",  
            Toast.LENGTH_LONG).show();  
    }  
}
```

# AndroidManifest.xml

```
<receiver android:name="MyReceiver">  
  <intent-filter>  
    <action  
      android:name="com.example.CUSTOM_INTEN  
T">  
    </action>  
  </intent-filter>  
</receiver>
```

# XML File

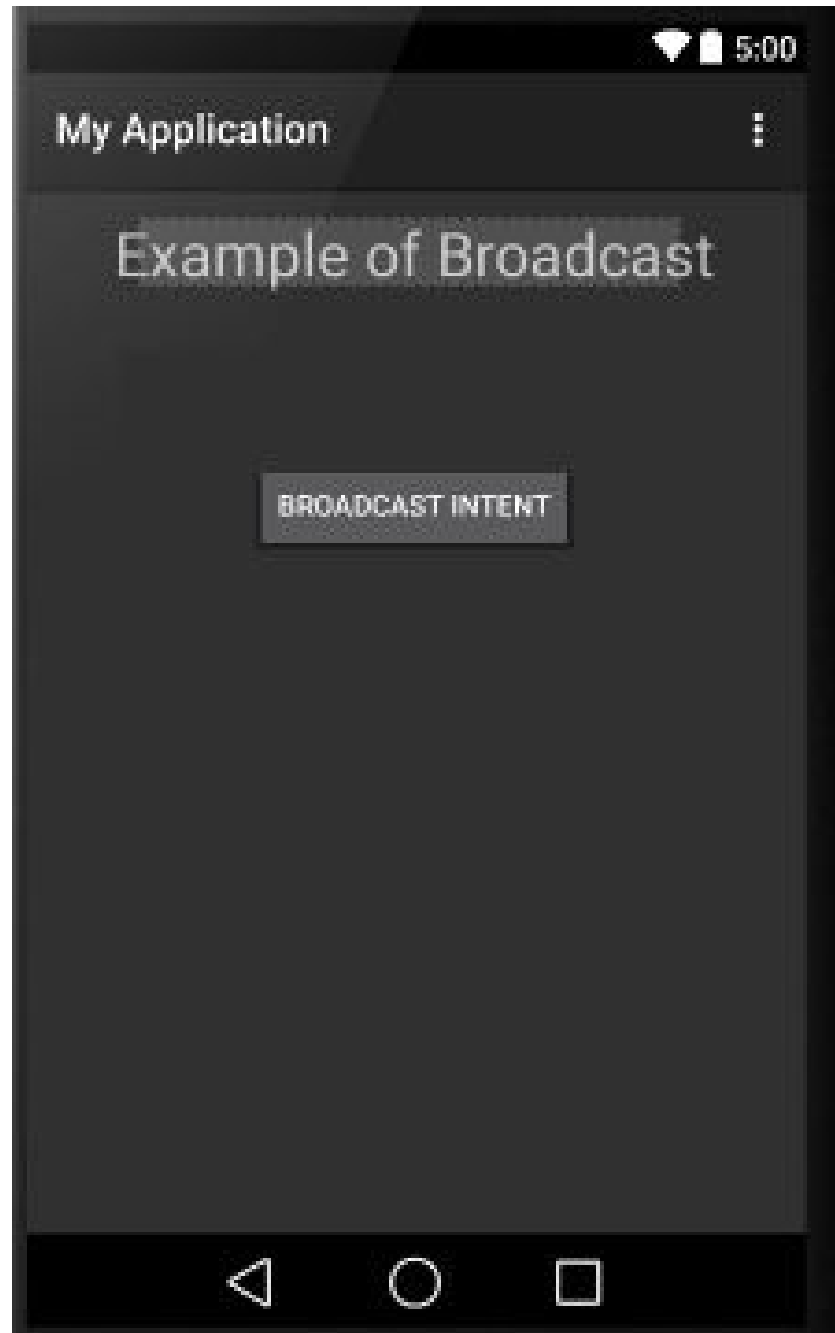
<TextView

```
    android:id="@+id/textView1"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Example of Broadcast" />
```

<Button

```
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:id="@+id/button1"  
    android:text="BROADCAST INTENT"  
    android:onClick="broadcastIntent" />
```

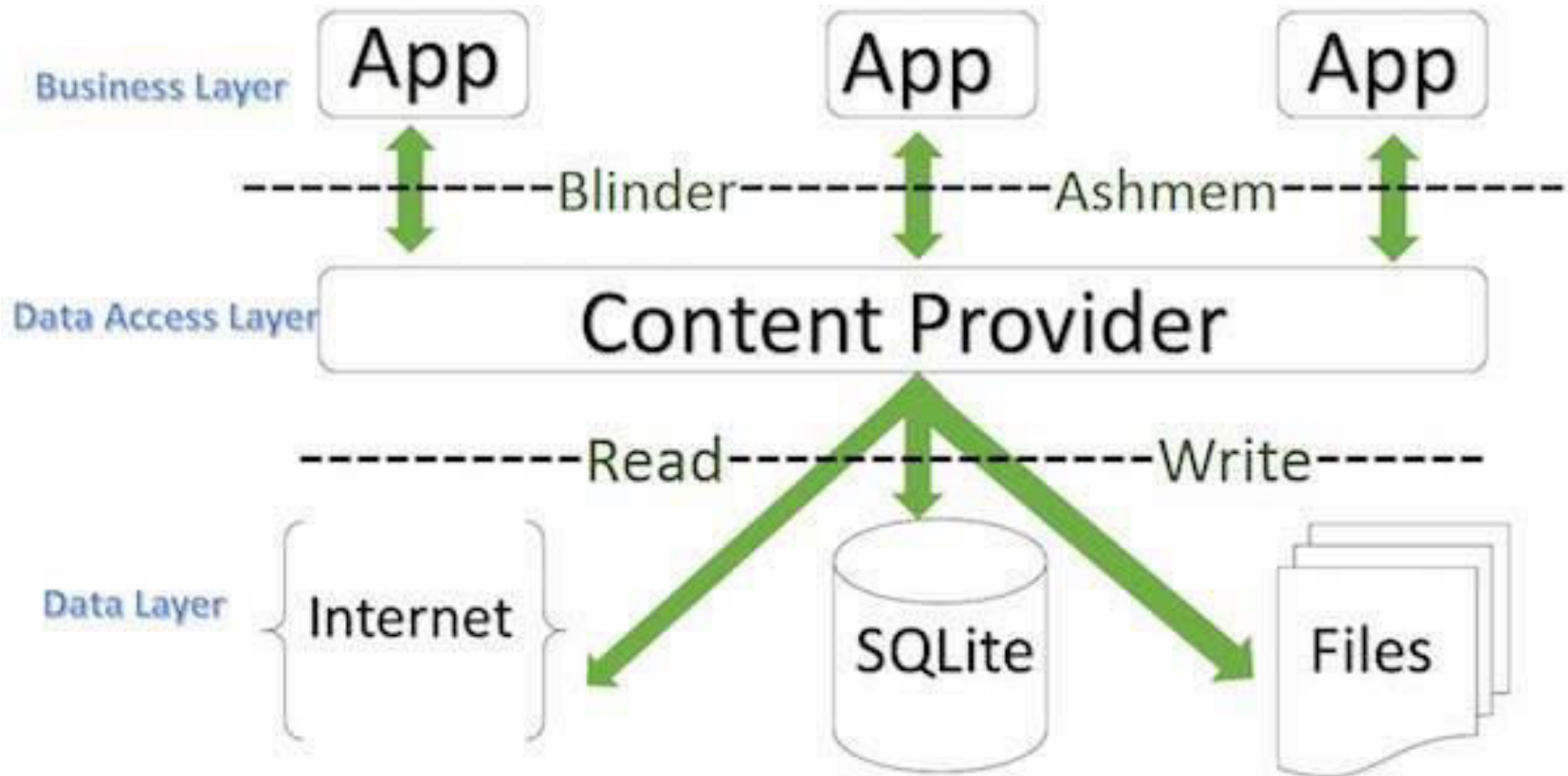
# Output



# 10. Content Provider

- A content provider manages access to a central repository of data.
- A provider is part of an Android application, which often provides its own UI for working with the data.
- A content provider component supplies data from one application to others on request.
- Such requests are handled by the methods of the ContentResolver class.
- A content provider can use different ways to store its data and the data can be stored in a database, in files, or even over a network

# Content Provider (con...)



# Content Provider (con...)

- Sometimes it is required to share data across applications. This is where content providers become very useful.
- Content providers let you centralize content in one place and have many different applications access it as needed.
- A content provider behaves very much like a database where you can query it, edit its content, as well as add or delete content using insert(), update(), delete(), and query() methods.



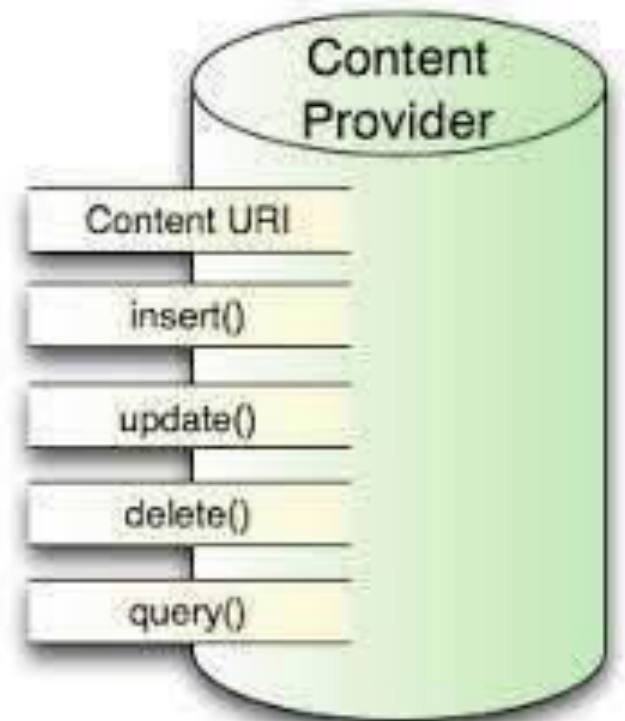
# Create Content Provider

- First of all you need to create a Content Provider class that extends the *ContentProvider* *baseclass*.
- Second, you need to **define your content provider URI address** which will be used to access the content.
- Next you will need to **create your own database** to keep the content.
- Next you will have **to implement Content Provider** queries to perform different database specific operations.
- Finally **register your Content Provider** in your activity file using <provider> tag.

# Content Provider (con...)

- **Syntax:**

```
public class MyApplication extends  
    ContentProvider {  
    //To do the code here  
}
```



# Content Provider (con...)

- **onCreate()** This method is called when the provider is started.
- **query()** This method receives a request from a client. The result is returned as a Cursor object.
- **insert()** This method inserts a new record into the content provider.
- **delete()** This method deletes an existing record from the content provider.
- **update()** This method updates an existing record from the content provider.
- **getType()** This method returns the MIME type of the data at the given URI.

# *AndroidManifest.xml* file.

```
<provider android:name="StudentsProvider" >  
    <android:authorities="com.example.provider.  
        College">  
</provider>
```

# Example

```
public void onClickAddName(View view) {  
    // Add a new student record  
    ContentValues values = new ContentValues();  
    values.put(StudentsProvider.NAME,  
        ((EditText)findViewById(R.id.txtName)).getText().toString());  
    values.put(StudentsProvider.GRADE,  
        ((EditText)findViewById(R.id.txtGrade)).getText().toString());  
    Uri uri = getContentResolver().insert(  
        StudentsProvider.CONTENT_URI, values);  
    Toast.makeText(getBaseContext(), uri.toString(),  
        Toast.LENGTH_LONG).show(); }  
}
```

# Example (con..)

```
public void onClickRetrieveStudents(View view) {  
    // Retrieve student records String URL =  
        "content://com.example.provider.College/students";  
    Uri students = Uri.parse(URL);  
    Cursor c = managedQuery(students, null, null, null, "name");  
    if (c.moveToFirst()) {  
        do{ Toast.makeText(this,  
            c.getString(c.getColumnIndex(StudentsProvider._ID)) + ", " +  
            c.getString(c.getColumnIndex( StudentsProvider.NAME)) + ", "  
            + c.getString(c.getColumnIndex( StudentsProvider.GRADE)),  
            Toast.LENGTH_SHORT).show();  
        } while (c.moveToNext()); }  
}
```

# Example (con..)

@Override

```
public Uri insert(Uri uri, ContentValues values) {  
    long rowID = db.insert( STUDENTS_TABLE_NAME, "", values);  
    if (rowID > 0) {  
        Uri _uri = ContentUris.withAppendedId(CONTENT_URI, rowID);  
        getContext().getContentResolver().notifyChange(_uri, null);  
        return _uri;  
    }  
    throw new SQLException("Failed to add a record into " + uri);  
}
```

# XML file

```
<Button android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:id="@+id/button1"  
    android:text="Add Name"  
    android:onClick="onClickAddName"/>
```

```
<Button android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Retrive student"  
    android:id="@+id/button2"  
    android:onClick="onClickRetrieveStudents"/>
```



# Output



# 11. Databases - SQLite

- SQLite is at the heart of **Android's database** support.
- **Simple, small (~350KB), light weight RDMS** implementation with simple API
- Each database is stored as a **single file** containing both Pragma & Data
  - **Writes** cause file locking and are always sequential and blocking
  - **Reads** can be multi-tasked

# SQLite (con...)

- SQLite is so dominant in the **embedded** and also the **mobile** world due to,
  - Low memory consumption
  - Ease of use
  - Free availability
- **Open Source**
- **ACID** Compliant
- Uses **SQL query language**

# SQLite (con...)

- **SQLite Version** included with Android varies with OS version
  - $\leq 2.1$  : SQLite 3.5.9
  - 2.2 - 2.3.3 : SQLite 3.6.22
  - 3.0 – 4.0.3 : SQLite 3.7.4
  - 4.1 – 4.4.x : SQLite 3.7.11
  - 4.5 : SQLite 3.8.4
  - **Latest SQLite Version: 3.8.10**

# SQLite (con...)

- It **differs in many aspects** from a conventional database system.
  - SQLite is serverless
  - SQLite stores data in one database file
  - SQLite offers only a few data types
  - SQLite uses manifest typing instead of static types
  - SQLite has no fixed column length
  - SQLite uses cross-platform database files

# Data Types in SQLite

- **NULL** – The null value
- **INTEGER** - Any number which is no floating point number
- **REAL** - A floating point value, 8-byte IEEE floating point number.
- **TEXT** - text string and also single characters, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).
- **BLOB** - The value is a blob of data, stored exactly as it was input.

# SQLite Classes

- **SQLiteCursor** - A Cursor implementation that exposes results from a query on a SQLiteDatabase.
- **SQLiteDatabase** - Exposes methods to manage a SQLite database. It has methods to create, delete, execute SQL commands, and perform other common database management tasks.
- **SQLiteOpenHelper** - A helper class to manage database creation and version management.
- **SQLiteProgram** - A base class for compiled SQLite programs.

# SQLite Classes (con...)

- **SQLiteQuery** – A SQLite program that represents a query that reads the resulting rows into a CursorWindow.
- **SQLiteQueryBuilder** - A convenience class that helps build SQL queries to be sent to SQLiteDatabase objects.
- **SQLiteStatement** - A pre-compiled statement against a SQLiteDatabase that can be reused.



# android.database.sqlite.SQLiteDatabase

- Contains the **methods** for:
  - creating
  - opening
  - closing
  - inserting
  - updating
  - deleting
  - querying

# openOrCreateDatabase()

- This method will **open an existing database or create** one in the application data area

```
SQLiteDatabase myDatabase;  
myDatabase = openOrCreateDatabase  
              ("my_sqlite_database.db" ,  
              SQLiteDatabase.CREATE_IF_NECESSARY, null);
```

# Create Table

```
String createTable = "CREAT TABLE  
demo(id INTEGER PRIMARY KEY  
AUTOINCREMENT,  
firstName TEXT, lastName TEXT);  
myDatabase.execSQL(createTable);
```

# Insert Records

```
long insert(String table, String  
            nullColumnHack, ContentValues values)
```

```
import android.content.ContentValues;  
ContentValues values = new ContentValues( );  
values.put("firstname" , "First Name");  
values.put("lastname" , "Last Name");  
long newAuthorID =  
    myDatabase.insert("demo" , "" , values);
```

# Update Records

```
int update(String table, ContentValues values,  
           String whereClause, String[ ] whereArgs)
```

```
Integer demold=1;
```

```
ContentValues values = new ContentValues();
```

```
values.put("firstname" , "New First Name");
```

```
myDatabase.update("demo" , values ,
```

```
"id=?" , new String[ ] {demold.toString() } );
```

# Record Deletion

```
int delete(String table, String whereClause,  
           String[] whereArgs)
```

```
String [] whereArgs= {"20", "30"};  
recAffected= myDatabase.delete("demo",  
                                "recID> ? and recID< ?", whereArgs);
```

# Queries using SQLite

Android offers **two mechanisms** for phrasing SQL-select statements:

- **Raw queries** take for input a syntactically correct SQL-select statement. The select query could be as complex as needed and involve any number of tables
- **Simple queries** are compact parameterized select statements that operate on a single table

# Raw Query

1. `Cursor c1 = db.rawQuery("select count(*)  
as Total from demo",null);`
2. `String mySQL= "select count(*) as Total  
from demo where recID> ? and name = ?";  
String[] args= {"1", "Sathyabama"};  
Cursor c1 = db.rawQuery(mySQL, args);`



# Simple Queries

- Simple queries use a template implicitly representing a condensed version of a typical (non-joining) SQL select statement.
- No explicit SQL statement is made.
- Simple queries can only retrieve data from a single table.

# Simple Queries (con...)

- The method's signature has a **fixed sequence of seven arguments** representing:
  1. the table name
  2. the columns to be retrieved
  3. the search condition (where-clause)
  4. arguments for the where-clause
  5. the group-by clause
  6. having-clause
  7. the order-by clause

# query method

```
query(String table,  
      String[] columns,  
      String selection,  
      String[] selectionArgs,  
      String groupBy,  
      String having,  
      String orderBy)
```

# Simple Query - Example

```
String[] columns = {"Dno","Avg(Salary) as AVG"};
String[] conditionArgs= {"30", "Chennai"};
Cursor c = db.query("EmployeeTable",
                    columns,
                    " age>= ? And location= ? " ,
                    conditionArgs,
                    "Dno",
                    "Count(*) > 2",
                    "AVG Desc " );
```

# Cursors

- Android cursors are used to **gain (random) access to tables** produced by SQL select statements.
  - Cursors primarily provide **one row-at-the-time operations** on a table.
1. **Positional awareness operators** (isFirst(), isLast(), isBeforeFirst(), isAfterLast())
  2. **Record Navigation** (moveToFirst(), moveToLast(), moveToNext(), moveToPrevious(), move(n))
  3. **Field extraction** (getInt, getString, getFloat, getBlob, getDate)
  4. **Schema inspection** (getColumnName, getColumnNames, getColumnIndex, getColumnCount, getCount)

# Cursor Example

```
String [] columns ={"id", "firstname", "lastname"};
Cursor myCur= db.query("demo", columns, null, null, null,
    null, "recID");
int idCol= myCur.getColumnIndex("id");
int fnameCol= myCur.getColumnIndex("firstname");
int lnameCol= myCur.getColumnIndex("lastname");
while(myCur.moveToNext()) {
    columns[0] = Integer.toString(myCur.getInt(idCol));
    columns[1] = myCur.getString(fnameCol);
    columns[2] = myCur.getString(lnameCol);
    txtMsg.append("\n" + columns[0] + " " + columns[1] + " " +
        columns[2] );
}
```

# 12. Publish App in Playstore

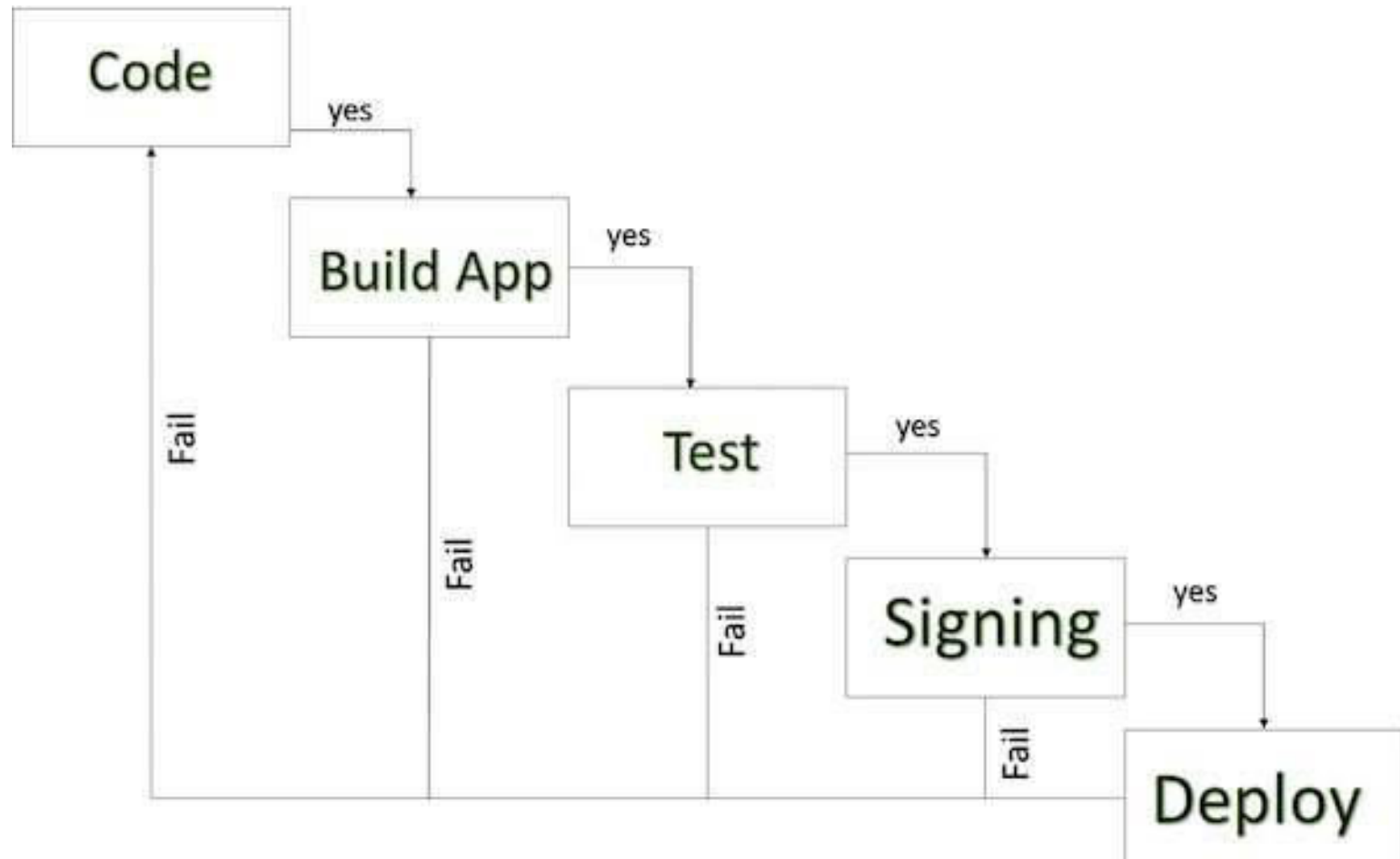
- In order to publish applications on Google play, it is **necessary to have a publisher account**. To sign up for a publisher account follow these steps:
- 1. Visit the Google Play Android Developer console at <https://play.google.com/apps/publish>.
- 2. Enter basic information about your developer identity.
- 3. Read and accept the Developer Distribution Agreement for your locale.
- 4. Pay the \$25 USD registration fee.
- 5. Confirm verification by e-mail.
- 6. After the account has been created, it is possible to publish applications using Google Play.

# Publish App in Playstore (con...)

- Android application publishing is a process that makes your **Android applications available to users**.
- Infact, **publishing is the last phase** of the Android application development process.
- Once you developed and fully tested your Android Application, you can **start selling or distributing free using Google Play**.
- You can also release your applications by **sending them directly to users or by letting users download them from your own website**.



# Publish App in Playstore (con...)



# Publish App in Playstore (con...)

- A simplified **check list** which will help you in launching your Android application.
  - 1. **Regression Testing** Before you publish your application, you need to make sure that its meeting the basic quality expectations for all Android apps, on all of the devices that you are targeting. So perform all the required testing on different devices including phone and tablets.
  - 2. **Application Rating** When you will publish your application at Google Play, you will have to specify a content rating for your app, which informs Google Play users of its maturity level. Currently available ratings are (a) Everyone (b) Low maturity (c) Medium maturity (d) High maturity.

# Publish App in Playstore (con...)

- 3. **Targeted Regions** Google Play lets you control what countries and territories where your application will be sold. Accordingly you must take care of setting up time zone, localization or any other specific requirement as per the targeted region.
- 4. **Application Size** Currently, the maximum size for an APK published on Google Play is 50 MB. If your app exceeds that size, or if you want to offer a secondary download, you can use APK Expansion Files, which Google Play will host for free on its server infrastructure and automatically handle the download to devices.

# Publish App in Playstore (con...)

- 5. **SDK and Screen Compatibility** It is important to make sure that your app is designed to run properly on the Android platform versions and device screen sizes that you want to target.
- 6. **Application Pricing** Deciding whether your app will be free or paid is important because, on Google Play, free app's must remain free. If you want to sell your application then you will have to specify its price in different currencies.

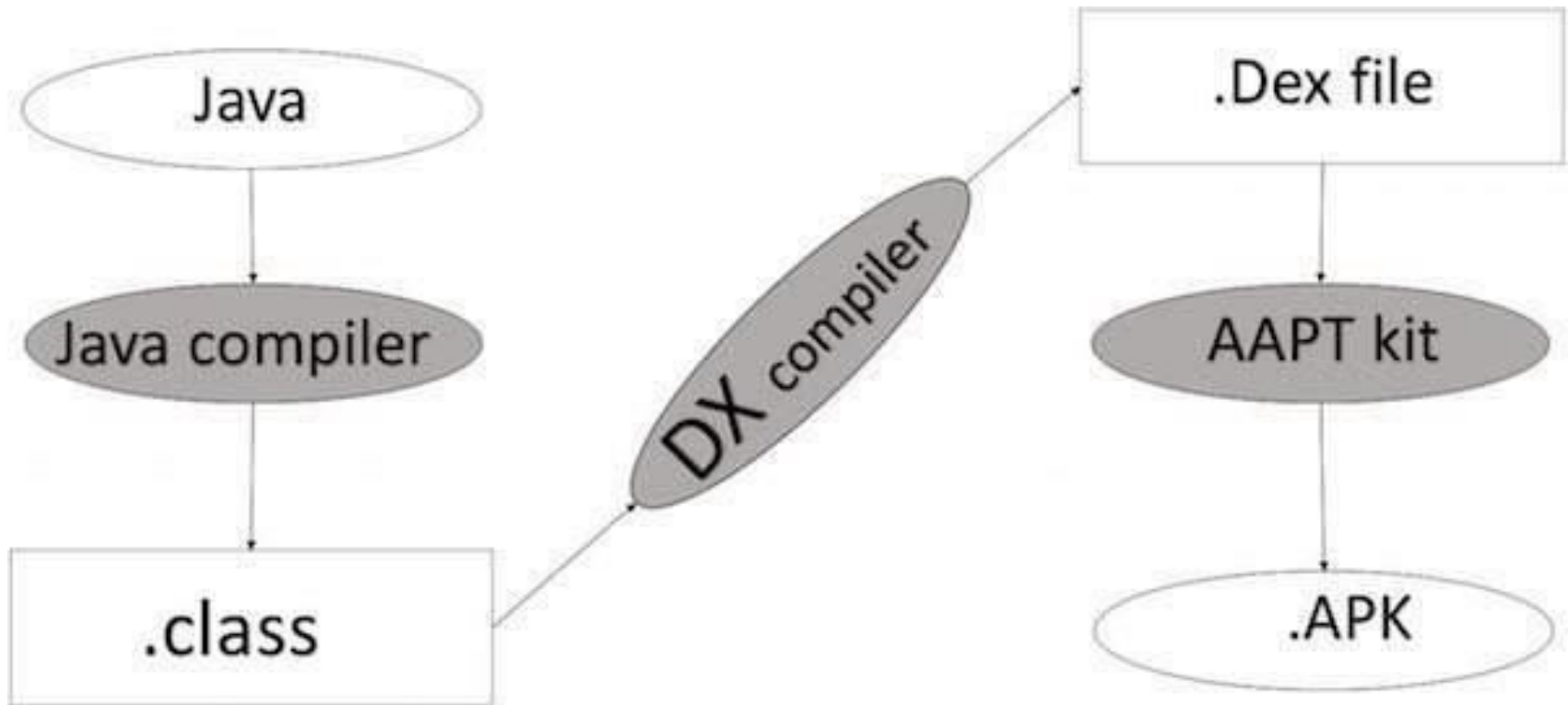
# Publish App in Playstore (con...)

- 7. **Promotional Content** It is a good marketing practice to supply a variety of high-quality graphic assets to showcase your app or brand. After you publish, these appear on your product details page, in store listings and search results, and elsewhere.
- 8. **Build and Upload release-ready APK** The release-ready APK is what you will upload to the Developer Console and distribute to users. You can check complete detail on how to create a release-ready version of your app: **Preparing for Release**

# Publish App in Playstore (con...)

- 9. **Finalize Application Detail** Google Play gives you a variety of ways to promote your app and engage with users on your product details page, from colour-ful graphics, screen shots, and videos to localized descriptions, release details, and links to your other apps. So you can decorate your application page and provide as much as clear crisp detail you can provide.

# Publish App in Playstore (con...)



APK DEVELOPMENT PROCESS

# Publish App in Playstore (con...)

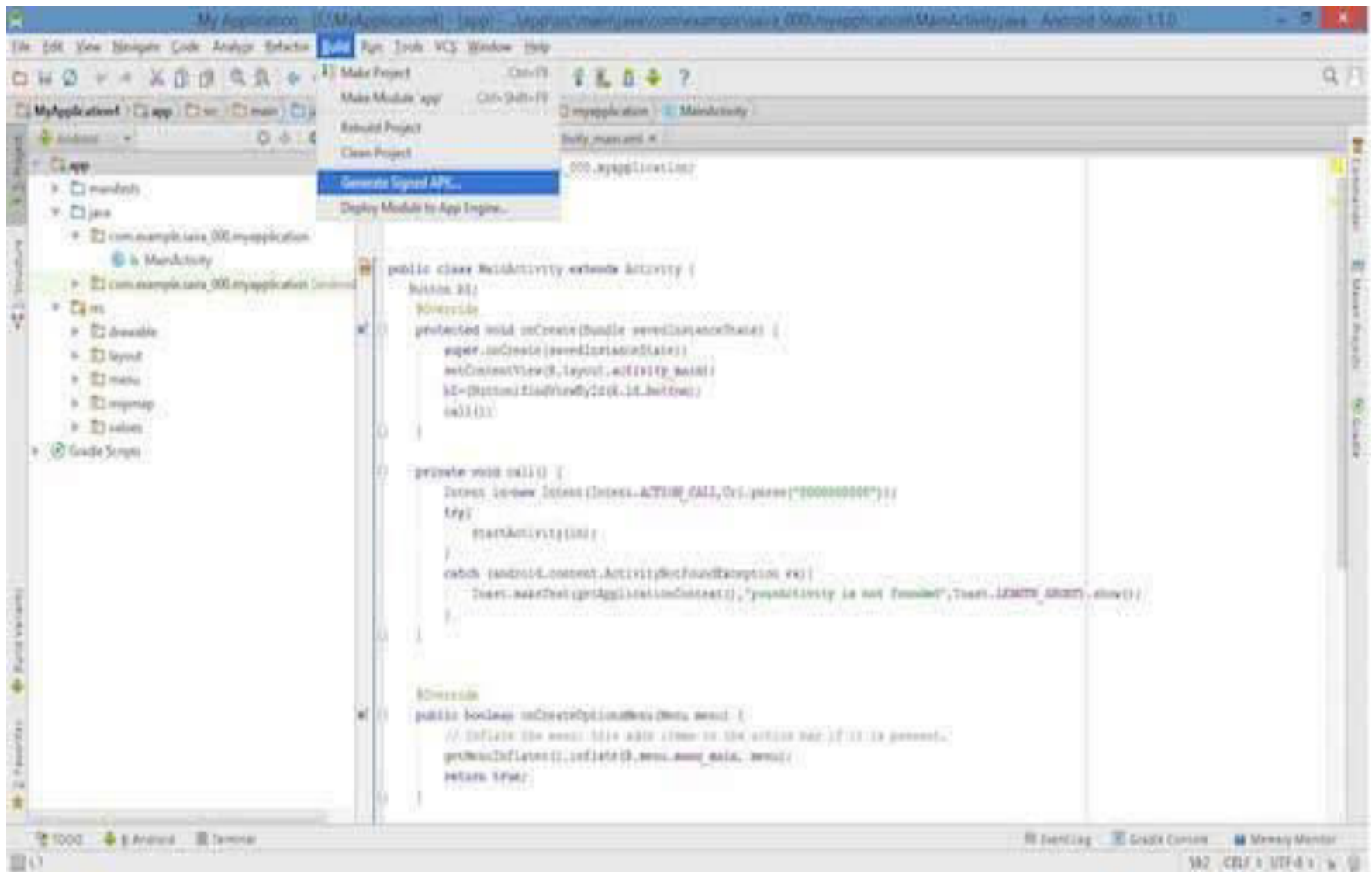
- Before exporting the apps, you must know some of the tools
  - **Dx tools(Dalvik executable tools )**: It going to convert **.class file to .dex file**. it has useful for memory optimization and reduce the boot-up speed time
  - **AAPT(Android assistance packaging tool)**: It has useful to convert **.Dex file to .apk file**
  - **APK(Android packaging kit)**: The final stage of deployment process is called as .apk.
- You will need to export your application as an APK (Android Package) file before you upload it Google Play.



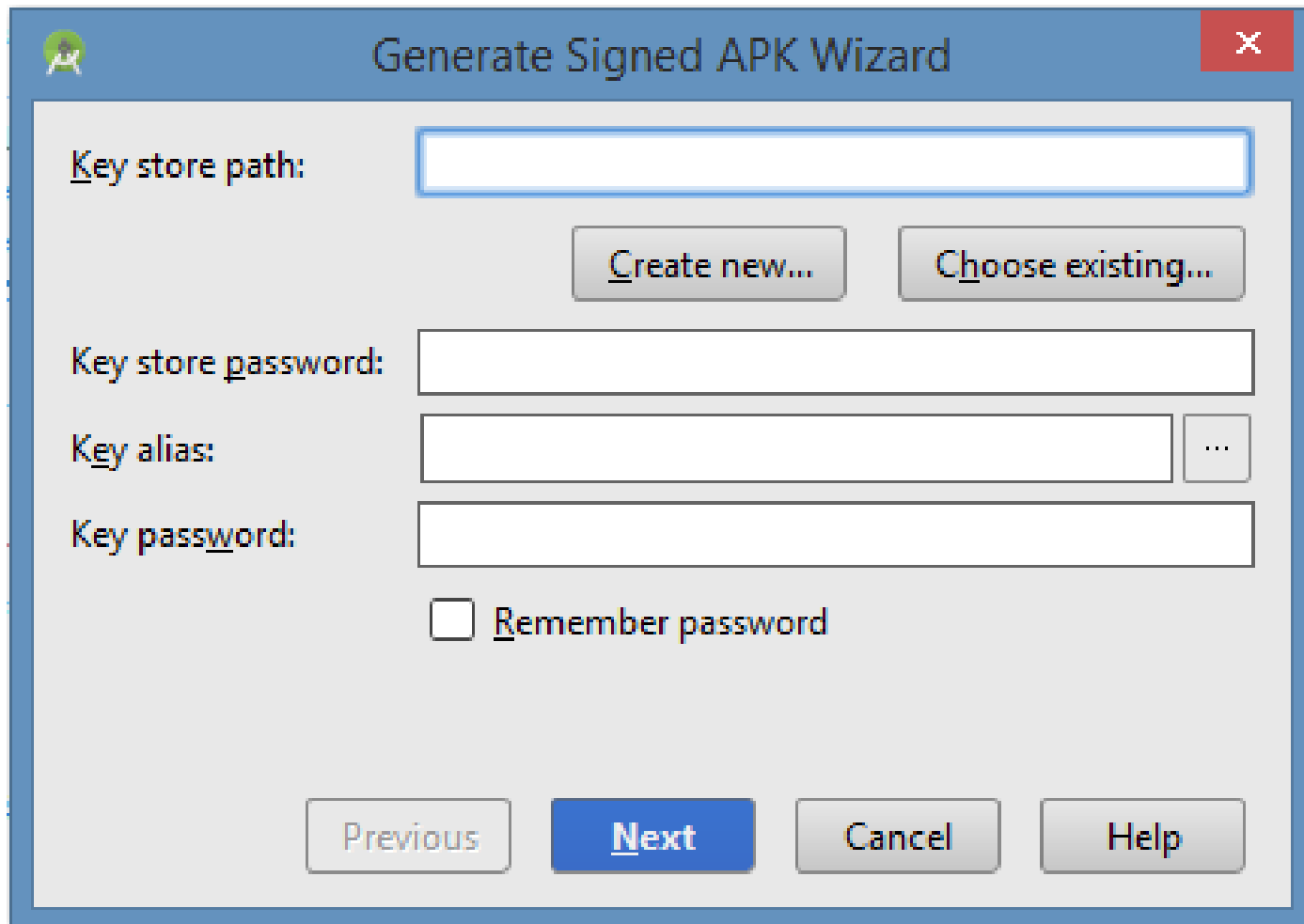
# Publish App in Playstore (con...)

- To export an application, just open that application project in Eclipse IDE / Android studio and select **Build → Generate Signed APK** from your Eclipse IDE / Android studio and follow the simple steps to export your application.
- Next select, **Generate Signed APK** option as shown in the above screen shot and then click it so that you get following screen where you will choose **Create new keystore** to store your application.

# Publish App in Playstore (con...)



# Publish App in Playstore (con...)



The image shows a Windows-style dialog box titled "Generate Signed APK Wizard". It has a blue title bar with a green icon on the left and a red close button on the right. The main area is light gray and contains several input fields and buttons. The first field is "Key store path:" followed by a text box. Below it are two buttons: "Create new..." and "Choose existing...". The next field is "Key store password:" followed by a text box. Below that is "Key alias:" followed by a text box and a small button with three dots. The next field is "Key password:" followed by a text box. Below that is a checkbox labeled "Remember password". At the bottom are four buttons: "Previous", "Next" (highlighted in blue), "Cancel", and "Help".

Generate Signed APK Wizard

Key store path:

Key store password:

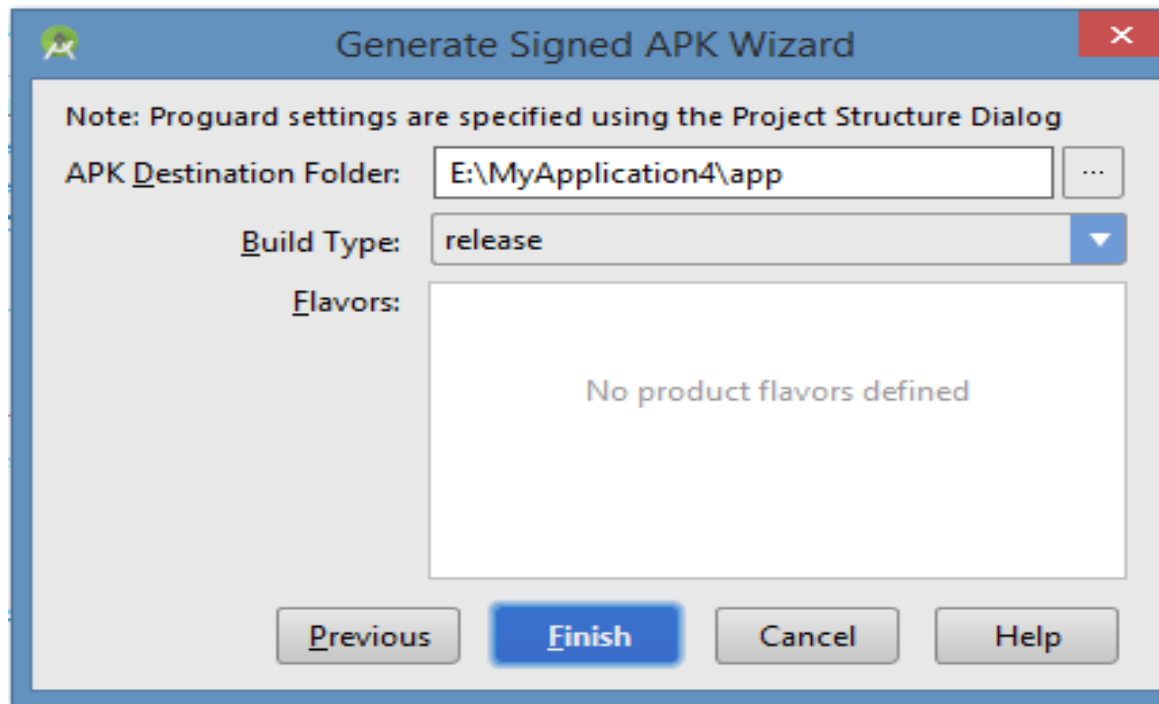
Key alias:  

Key password:

☐ Remember password

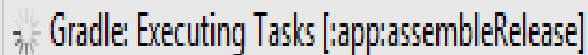
# Publish App in Playstore (con...)

- Enter your **key store path, key store password, key alias and key password** to protect your application and click on Next button once again. It will display following screen to let you create an application:



# Publish App in Playstore (con...)

- Once you filled up all the information, like app destination, build type and flavours click finish button While **creating an application** it will show as below



Gradle: Executing Tasks [:app:assembleRelease]

- Finally, it will generate your **Android Application as APK format File** which will be uploaded at Google Play marketplace.

# 13. Sample Applications

- To develop an App for sending data from one activity to another
- To develop an App for web browser
- To develop an App for sending SMS
- To develop an App for Placement Registration
- To develop an App for Resume
- To develop an App for phone profile changer
- To develop an App for sending notifications