Example 1 A:

Develop an application that uses GUI components

GUI Components:

GUI Components **- G**raphical **U**ser Interface Components.

In android **GUI Components** (Graphical User Interface Components) are used to design the user interface of an application. We have a wide variety of **GUI Components** available; those are **TextView**, **EditText** and **Button**...etc.

Generally the user interface of an android application is made with a collection of **View** and **ViewGroup** Objects.

View: View is a base class for all **GUI Components** in android and it is used to create an interactive **GUI Components** such as **TextView**, **EditText** and **Button**...etc.

ViewGroup: ViewGroup is a subclass of **View** and it will act as a base class for Layouts and Layout parameters. ViewGroup will provide invisible containers to hold other **Views** and **ViewGroups** and to define layout properties.

In this example, we will concentrate on three GUI Components

- TextView
- EditText
- Button

TextView:

TextView is a Graphical User Interface Component that is used to display the text to user.

EditText:

EditText is a Graphical User Interface Component that is used to get inputs from the user.

Button:

EditText is a Graphical User Interface Component that is used to perform an action when the user clicks or tap on it.

Example 1 B:

Develop an application that uses Font

In android application development, we can able to use custom fonts in two ways

- Using Assets Folder
- Using Fonts Folder

Using Assets Folder:

In this approach you need to add fonts in your project Assets folder.

To add fonts in Assets folder, perform the following steps in the Android Studio:

1) Select Project
2) Select File>New>Folder>**Assets** Folder
3) Click finish
4) Right click on **assets** and create a folder called **fonts**
5) Put your font(**.ttf**) file in assets > fonts
6) Use the below code to change your GUI components font programmatically

```
TextView textView = (TextView) findViewById(R.id.textView);
Typeface typeface = Typeface.createFromAsset(getAssets(), "fonts/yourfont.ttf");
textView.setTypeface(typeface);
```

Using Fonts Folder:

In this approach you need to add fonts as resources in your project folder.

To add fonts as resources, perform the following steps in the Android Studio:

1) Right-click the res folder and go to New > Android resource directory.
2) In the Resource type list, select font, and then click OK
3) Add your font(.ttf) file in the font folder.

Now you can use the custom fonts in your android application in two ways

a) You can directly use the custom font in xml as

```
android:fontFamily="@font/jose_sans "
```

b) Programmatically you can achieve this by using

```
TextView textView = (TextView) findViewById(R.id.textView);
Typeface typeface = ResourcesCompat.getFont(this, R.font. jose_sans);
textView.setTypeface(typeface);
```

Example 1 C:

Develop an application that uses Colors

In android application development, we can able to use custom fonts in two ways

- Using Color Codes
- Using Color from Colors.xml file

Using Color Codes:

In this approach you will able to use the color codes programmatically in your android application.

```
int deepColor = Color.parseColor("#808000");

int lightColor = Color.parseColor("#FFF5EE");
```

Now you will able to use the above color code in your android application.

```
Button btn = (Button) findViewById(R.id.button);
btn.setBackgroundColor(deepColor);
```

Using Color from Colors.xml file

In this approach you need to add colors in your project colors.xml file.

To add colors in colors.xml file, perform the following steps in the Android Studio:

1. Select Project
2. Right click app / res / values folder. Click New —> Values resource file menu.
3. Input colors in File name input box, click OK button.
4. Then colors.xml file will be created in left panel under app / res / values folder. Double click it, input below color definition xml data in it.

```
<color name="colorRed">#ff0000</color>
<color name="colorOrange">#f1a86c</color>
```

Now color variables can be used in both xml and java file. It is easy to refer them.

a) Use custom color variable in xml file such as layout xml file

@color/colorVariableName @color/colorOrange.

Example:

```
<TextView
   android:id="@+id/textView1"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="TextView"
   android:background="@color/colorRed" />
```

b) Use custom color variable in java file

R.color.colorVariableName R.color.colorRed.

R is the android auto created resource class.

Example:

```
Button btn = (Button) findViewById(R.id.button);
btn.setBackgroundColor(R.color.colorRed);
```

Example 2 A:

Develop an application that uses Layout Managers

Layout Manager (ViewGroup)

ViewGroup is a subclass of View. One or more Views can be grouped together into a ViewGroup. A ViewGroup provides the android layout in which we can order the appearance and sequence of views.

Examples of ViewGroup are LinearLayout, FrameLayout, RelativeLayout etc.

Layout Managers Types:

- **LinearLayout:** is a ViewGroup that aligns all children in a single direction, vertically or horizontally.
- **RelativeLayout:** is a ViewGroup that displays child views in relative positions
- **AbsoluteLayout:** allows us to specify the exact location of the child views and widgets
- **TableLayout:** is a view that groups its child views into rows and columns
- **FrameLayout:** is a placeholder on screen that is used to display a single view

In this example we will see the detailed explanations about LinearLayout
and RelativeLayout below

LinearLayout:

LinearLayout organizes elements along a single line. We can specify whether that line is vertical or horizontal using “android: orientation”. The orientation is horizontal by default.

A vertical LinearLayout will only have one child per row (so it is a column of single elements), and a horizontal LinearLayout will only have one single row of elements on the screen.

android:layout_weight attribute depicts the importance of the element. An element with larger weight occupies more screen space.

RelativeLayout:

Android RelativeLayout lays out elements based on their relationships with one another, and with the parent container. This is one of the most complicated layout and we need several properties to actually get the layout we desire.

Using RelativeLayout we can position a view to be **toLeftOf**, **toRightOf**, **below** or **above** its siblings.

We can also position a view with respect to its parent such as **centered, horizontally and vertically** or both, or aligned with any of the edges of the parent RelativeLayout. If none of these attributes are specified on a child view then the view is by default rendered to the top left position.

Example 2 B:

Develop an application that uses Event Listeners

Event Listeners

In android, Event Listener is an interface in the View class that contains a single call-back method. These methods will be called by the Android framework when the View which is registered with the listener is triggered by user interaction with the item in UI.

There are three types of event listeners there in android

- View Event Listeners
- EditText Common Listeners
- Input View Listeners

View Event Listeners:

Any View (Button, TextView, etc) has many event listeners that can be attached using the setOnEvent pattern which involves passing a class that implements a particular event interface. The listeners available to any View include:

- **setOnClickListener** - Callback when the view is clicked.
- **setOnDragListener** - Callback when the view is dragged.
- **setOnCheckedChangeListener** - Callback when the view changes checked option.
- **setOnGenericMotionListener** - Callback for arbitrary gestures.
- **setOnHoverListener** - Callback for hovering over the view.
- **setOnKeyListener** - Callback for pressing a hardware key when view has focus.
- **setOnLongClickListener** - Callback for pressing and holding a view.
- **setOnTouchListener** - Callback for touching down or up on a view.

EditText Common Listeners:

In addition to the listeners described above, there are a few other common listeners for input fields in particular.

- **addTextChangedListener** - Fires each time the text in the field is being changed.
- **setOnEditorActionListener** - Fires when an "action" button on the soft keyboard is pressed.

Input View Listeners:

Similarly to EditText, many common input views have listeners of their own including **NumberPicker** has **setOnValueChangedListener** and **SeekBar** has **setOnSeekBarChangeListener** which allow us to listen for changes

Example 3:

Write an application that draws Basic Graphical Primitives on the screen.

Graphical Primitives:

The most important step in drawing a Graphical Primitives to override the **onDraw()** method. The parameter to **onDraw()** is a Canvas object that the view can use to draw itself.
The Canvas class defines methods for drawing text, lines, bitmaps, and many other graphics primitives. You can use these methods in **onDraw()** to create your custom Graphical Primitives.

The **android.graphics** framework divides drawing into two areas:

- What to draw, handled by **Canvas.**
- How to draw, handled by **Paint**.

For instance, Canvas provides a method to draw a line, while Paint provides methods to define that line's color. Canvas has a method to draw a rectangle, while Paint defines whether to fill that rectangle with a color or leave it empty. Simply put, Canvas defines shapes that you can draw on the screen, while Paint defines the color, style, font, and so forth of each shape you draw. So, before you draw anything, you need to create one or more Paint objects.

In this example, we are going to display 2D graphics in android. In 2D graphics we usually opt for any of the two following options:

1. Graphics or animation object is drawn into View object from layout.
2. We can draw graphics directly onto the canvas.

Android Canvas class encapsulates the bitmaps used as surface. It exposes the draw methods which can be used for designing. Let us first clear the following terms:

- **Bitmap**: The surface being drawn on.
- **Paint**: It lets us specify how to draw the primitives on bitmap. It is also referred to as "Brush".
- **Canvas**: It supplies the draw methods used to draw primitives on underlying bitmap.

Each drawing object specifies a paint object to render. Let us see the few available list of drawing objects and they are as follows:

1. **drawCircle:** This draws a circle on a specified radius centered on a given point.
2. **drawLine(s):** it draws a line (or series of lines) between points.
3. **drawOval:** it draws an oval which is bounded by the area of rectangle.
4. **drawRoundRect:** it draws a rectangle with round edges.
5. **drawText:** It draws a text string on canvas.

Example 4:

Develop an application that makes use of SQLite Database.

SQLite Database.

In Android, there are several ways to store persistent data. **SQLite** is one way of storing app data. It is very lightweight database that comes with Android OS. In Android, integrating SQLite is a tedious task as it needs writing lot of boilerplate code to store simple data. Consider SQLite when your app needs to store simple data objects. Alternatively you can consider Room Persistence Library for better APIs and easier integration.

In this example we are going to learn basics of SQLite database CRUD operations, Creating Database, **Creating Tables, Creating Records, Reading Records, Updating Records and Deleting Records.**

Before that we need to know about some basic things:

What is SQLite?

SQLite is an SQL Database. I am assuming here that you are familiar with SQL databases. So in SQL database, we store data in tables. The tables are the structure of storing data consisting of rows and columns. We are not going in depth of what is an SQL database and how to work in SQL database. If you are going through this post, then you must know the Basics of SQL.

What is CRUD?

As the heading tells you here, we are going to learn the CRUD operation in SQLite Database. But what is CRUD? CRUD is nothing but an abbreviation for the basic operations that we perform in any database. And the operations are

- Create
- Read
- Update
- Delete

SQLiteOpenHelper

Android has features available to handle changing database schemas, which mostly depend on using the SQLiteOpenHelper class. SQLiteOpenHelper **is designed to get rid of two very common problems.**

1. When the application runs the first time – At this point, we do not yet have a database. So we will have to create the tables, indexes, starter data, and so on.

2. When the application is upgraded to a newer schema – Our database will still be on the old schema from the older edition of the app. We will have option to alter the database schema to match the needs of the rest of the app.

SQLiteOpenHelper wraps up these logic to create and upgrade a database as per our specifications. For that we'll need to create a custom subclass of SQLiteOpenHelper implementing at least the following three methods.

- **Constructor:** This takes the Context (e.g., an Activity), the name of the database, an optional cursor factory (we'll discuss this later), and an integer representing the version of the database schema you are using (typically starting from 1 and increment later).

```
public DatabaseHelper(Context context) {
super(context, DB_NAME, null, DB_VERSION);
}
```

- **onCreate(SQLiteDatabase db) :** It's called when there is no database and the app needs one. It passes us a SQLiteDatabase object, pointing to a newly-created database, that we can populate with tables and initial data.

- **onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) :** It's called when the schema version we need does not match the schema version of the database, It passes us a SQLiteDatabase object and the old and new version numbers. Hence we can figure out the best way to convert the database from the old schema to the new one.

We define a DBManager class to perform all database CRUD(Create, Read, Update and Delete) operations.

Opening and Closing Android SQLite Database Connection

Before performing any database operations like insert, update, delete records in a table, first open the database connection by calling **getWritableDatabase()** method as shown below:

```
public DBManager open() throws SQLException {
    dbHelper = new DatabaseHelper(context);
    database = dbHelper.getWritableDatabase();
    return this;
}
```

The dbHelper is an instance of the subclass of SQLiteOpenHelper. To close a database connection the following method is invoked.

```
public void close() {
    dbHelper.close();
  }
```

Inserting new Record into Android SQLite database table

The following code snippet shows how to insert a new record in the android SQLite database.

```
public void insert(String name, String desc) {
    ContentValues contentValue = new ContentValues();
    contentValue.put(DatabaseHelper.SUBJECT, name);
    contentValue.put(DatabaseHelper.DESC, desc);
    database.insert(DatabaseHelper.TABLE_NAME, null, contentValue);
  }
```

Content Values creates an empty set of values using the given initial size. We'll discuss the other instance values when we jump into the coding part.

Updating Record in Android SQLite database table

The following snippet shows how to update a single record.

```
public int update(long _id, String name, String desc) {
    ContentValues contentValues = new ContentValues();
    contentValues.put(DatabaseHelper.SUBJECT, name);
    contentValues.put(DatabaseHelper.DESC, desc);
    int i = database.update(DatabaseHelper.TABLE_NAME, contentValues,
DatabaseHelper._ID + " = " + _id, null);
    return i;
  }
```

Android SQLite – Deleting a Record

We just need to pass the id of the record to be deleted as shown below.

```
public void delete(long _id) {
    database.delete(DatabaseHelper.TABLE_NAME, DatabaseHelper._ID + "=" + _id, null);
  }
```

Android SQLite Cursor

A Cursor represents the entire result set of the query. Once the query is fetched a call to **cursor.moveToFirst()** is made. Calling moveToFirst() does two things:

- It allows us to test whether the query returned an empty set (by testing the return value)
- It moves the cursor to the first result (when the set is not empty)

The following code is used to fetch all records:

```
public Cursor fetch() {
    String[] columns = new String[] { DatabaseHelper._ID, DatabaseHelper.SUBJECT,
DatabaseHelper.DESC };
    Cursor cursor = database.query(DatabaseHelper.TABLE_NAME, columns, null, null,
null, null, null);
    if (cursor != null) {
       cursor.moveToFirst();
    }
    return cursor;
  }
```

Example 5:

Develop an application that makes use of Notification Manager.

Notification

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.

Android notification message fills up the void in such situations. Android notification is a message that we can display to the user outside of our application's normal UI. Notifications in android are built using **NotificationCompat** library.

Creating Android Notification

A Notification is created using the **NotificationManager** class as shown below:

```
NotificationManager notificationManager = (NotificationManager)
  getSystemService(NOTIFICATION_SERVICE);
```

The **Notification.Builder** provides an builder interface to create an Notification object as shown below:

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

Android Notification Methods

We can set the notification properties on this builder object. Some of the frequently used methods and there descriptions are given below.

1. **Notification build()** : Combines all of the options that have been set and returns a new Notification object
2. **NotificationCompat.Builder setAutoCancel (boolean autoCancel)** : Setting this flag will make it such that the notification is automatically canceled when the user clicks it in the panel
3. **NotificationCompat.Builder setContent (RemoteViews views)** : Supplies a custom RemoteViews to use instead of the standard one

4. **NotificationCompat.Builder setContentInfo (CharSequence info)** : Sets the large text at the right-hand side of the notification
5. **NotificationCompat.Builder setContentIntent (PendingIntent intent)** : Supplies a PendingIntent to send when the notification is clicked
6. **NotificationCompat.Builder setContentText (CharSequence text)** : Sets the text (second row) of the notification, in a standard notification
7. **NotificationCompat.Builder setContentTitle (CharSequence title)** : Sets the text (first row) of the notification, in a standard notification
8. **NotificationCompat.Builder setDefaults (int defaults)** : Sets the default notification options that will be used.

   An example is;

   ```
   mBuilder.setDefaults(Notification.DEFAULT_LIGHTS |
   Notification.DEFAULT_SOUND)
   ```
9. **NotificationCompat.Builder setLargeIcon (Bitmap icon) :** Sets the large icon that is shown in the ticker and notification
10. **NotificationCompat.Builder setNumber (int number)** : Sets the large number at the right-hand side of the notification
11. **NotificationCompat.Builder setOngoing (boolean ongoing)** : Sets whether this is an ongoing notification
12. **NotificationCompat.Builder setSmallIcon (int icon)** : Sets the small icon to use in the notification layouts
13. **NotificationCompat.Builder setStyle (NotificationCompat.Style style)** : Adds a rich notification style to be applied at build time
14. **NotificationCompat.Builder setTicker (CharSequence tickerText)** : Sets the text that is displayed in the status bar when the notification first arrives
15. **NotificationCompat.Builder setVibrate (long[] pattern)** : Sets the vibration pattern to use
16. **NotificationCompat.Builder setWhen (long when)** : Sets the time that the event occurred. Notifications in the panel are sorted by this time.

PendingIntent

Android **PendingIntent** is an object that wraps up an intent object and it specifies an action to be taken place in future. In other words, **PendingIntent** lets us pass a future Intent to another application and allow that application to execute that Intent as if it had the same permissions as our application, whether or not our application is still around when the Intent is eventually invoked.

A **PendingIntent** is generally used in cases were an **AlarmManager** needs to be executed or for Notification (that we'll implement later in this tutorial). A **PendingIntent** provides a means for applications to work, even after their process exits.

For security reasons, the base Intent that is supplied to the **PendingIntent** must have the component name explicitly set to ensure it is ultimately sent there and nowhere else. Each explicit intent is supposed to be handled by a specific app component like Activity, **BroadcastReceiver** or a Service. Hence **PendingIntent** uses the following methods to handle the different types of intents:

1. **PendingIntent.getActivity()** : Retrieve a PendingIntent to start an Activity
2. **PendingIntent.getBroadcast()** : Retrieve a PendingIntent to perform a Broadcast
3. **PendingIntent.getService()** : Retrieve a PendingIntent to start a Service

An example implementation of **PendingIntent** is given below.

```
Intent intent = new Intent(this, SomeActivity.class);

// Creating a pending intent and wrapping our intent

PendingIntent pendingIntent = PendingIntent.getActivity(this, 1, intent,
PendingIntent.FLAG_UPDATE_CURRENT);

try {
   // Perform the operation associated with our pendingIntent
   pendingIntent.send();
} catch (PendingIntent.CanceledException e) {
   e.printStackTrace();
}
```

The operation associated with the **pendingIntent** is executed using the send() method.

The parameters inside the **getActivity()** method and there usages are described below :

- **this (context)** : This is the context in which the **PendingIntent** starts the activity
- **requestCode** : "1" is the private request code for the sender used in the above example. Using it later with the same method again will get back the same pending intent. Then we can do various things like cancelling the pending intent with **cancel()**, etc.
- **intent :** Explicit intent object of the activity to be launched
- **flag** : One of the **PendingIntent** flag that we've used in the above example is FLAG_UPDATE_CURRENT. This one states that if a previous **PendingIntent** already exists, then the current one will update it with the latest intent. There are many other flags like FLAG_CANCEL_CURRENT etc.

Cancelling Android Notification

We can also call the **cancel()** for a specific notification ID on the **NotificationManager**.
The **cancelAll()** method call removes all of the notifications you previously issued.

Example 6:

Implement an application that uses Multi-Threading.

Multi-Threading:

Multi-Threading in Android is a unique feature through which more than one threads execute together without hindering the execution of other threads. Multi-Threading in Android is not different from conventional multi-Threading. A class can be thought of as a process having its method as it's sub-processes or threads. All these methods can run concurrently by using feature of Multi-Threading. In android, multi-Threading can be achieved through the use of many in-built classes. Out of them, Handler class is most commonly used.

Handler class in Android:

Handler class come from the Package android.os.Handler package and is most commonly used for multi-threading in android. Handler class provide sending and receiving feature for messages between different threads and handle the thread execution which is associated with that instance of Handler class.

In android class, every thread is associated with an instance of Handler class and it allows the thread to run along with other threads and communicate with them through messages.

Instantiating Handler class:

There are following two ways in which Handler class is usually instantiated for supporting multi-threading:

- Through default constructor.

  Handler handlerObject = new Handler();
- Through Parameterized constructor

  Handler handleObject = new Handler(Runnable runnableObject, Handler.Callback callbackObject);

Methods of Handler class for Multi-Threading:

- **Public final Boolean post(Runnable runnableObject){ return booleanValue; }**

This Method attach a runnable instance with it's associated thread and the body of that runnable instance will execute every time the thread gets executed.

- **Public final Boolean postAtTime((Runnable runnableObject, long timeinMillisecondObject){ return booleanValue; }**

This Method attach a runnable instance with it's associated thread and the body of that runnable instance will execute every time the thread gets executed at a time specified by the second argument.

- **Public final Boolean postDelayed((Runnable runnableObject, long timeinMillisecondObject){ return booleanValue; }**

This Method attach a runnable instance with its associated thread and the body of that runnable instance will execute every time the thread gets executed after a time specified by the second argument.

Runnable Interface:

Runnable interface is used in multi-threading to be called in a loop when the thread starts.

It is a type of thread that executes the statement in its body or calls other methods for a specified or infinite number of times.

This runable interface is used by the Handler class to execute the multi-threading, i.e., to execute one or more thread in specified time.

Runnable is an interface which is implemented by the class desired to support multithreading and that class must implements it's abstract method public void run().

run() method is the core of multithreading as it includes the statement or calls to other methods that the thread needs to be made for multithreading.

```
class ClassName implements Runnable
{
        @override
        Public void run()
        {
                Body of method
        }
}
```

Runnable interface can also be used by using adapter class as explained below:

```
Runnable runnableObject =new Runnable()
{
        @override
        Public void run()
        {

        };
}
```

Example 7:

Develop a native application that uses GPS Location Information.

GPS Location Information:

One of the major features of android framework is location API. You can see, the location module widely used in lot of apps those provides services like food ordering, transportation, health tracking, social networking and lot more. The location module is part of Google Play Services and in the same module geofencing and activity recognition are included.

Earlier, getting location is very easy with couple of API calls. But to provide more accurate locations and optimizing the battery usage, Android introduced set APIs that should be combined to get the best results from the location API. We will be using **Fused Location API** that combines signals from **GPS**, **Wi-Fi**, and **cell networks**, as well as **accelerometer**, **gyroscope**, **magnetometer** and other sensors to provide more accurate results.

Before moving forward, we need to understand the location permissions.

Location Permissions:

If you want to get the current location of your user, then you have to add the location permission to your application. Android provides the following location permission:

- **ACCESS_COARSE_LOCATION:** By adding this in your application, allows you to use WIFI or mobile cell data(or both) to determine the device's location. The approximation by using this permission is close to the city level.

- **ACCESS_FINE_LOCATION:** This uses your GPS as well as WIFI and mobile data to get the most precise location as possible. By using this, you will get a precise location of your user.

- **ACCESS_BACKGROUND_LOCATION:** This permission was introduced in Android 10. So, for Android 10 or higher, if you want to access the user's location in the background, then along with any of the above two permissions, you need to add the ACCESS_BACKGROUND_LOCATION permission also.

Fused Location API:

In this example, we will use **Fused Location API** to get the changed location or you may say, get the current location of a user. We will be using **LocationRequest** that are used to request quality of service for location updates from the **FusedLocationProviderApi**.

Apart from getting the updated location, the **LocationRequest** includes various methods for retrieving locations like a pro. Some of the methods are:

- **setInterval(long millis):** This is used to set the desired interval after which you want to check for a location update. It is in milliseconds.
- **setFastestInterval(long millis):** This is used to set the fastest interval for a location update. This might be faster than your `setInterval(long)` in many cases because if other applications on the device are triggering location updates at an interval lesser than your interval then it will use that update.
- **setSmallestDisplacement(float smallestDisplacementMeters):** This will set the minimum displacement between location updates i.e. the smallest displacement required for a location update. It is in meters and the default value of this is 0.
- **setPriority(int priority):** This is used to set the priority of location received. It can be **PRIORITY_BALANCED_POWER_ACCURACY** (for accuracy up to **block** level) or it can be **PRIORITY_HIGH_ACCURACY**(to get the most accurate result) or it can be **PRIORITY_LOW_POWER**(to get the accuracy up to **city** level) or it can be **PRIORITY_NO_POWER**(to get the most accurate information without providing some additional power).

Example 8:

Implement an application that writes data to the SD Card.

External Storage

External storage such as SD card can also store application data, there's no security enforced upon files you save to the external storage.
In general there are two types of External Storage:

- **Primary External Storage**: In built shared storage which is "accessible by the user by plugging in a USB cable and mounting it as a drive on a host computer

- **Secondary External Storage**: Removable storage. Example: SD Card

All applications can read and write files placed on the external storage and the user can remove them. We need to check if the SD card is available and if we can write to it.

The **FileInputStream** and **FileOutputStream** classes are used to read and write data into the file.

It is necessary to add external storage the permission to read and write. For that you need to add permission in android Manifest file.

Methods to Store Data in Android:

1. **getExternalStorageDirectory()** – Older way to access external storage in API Level less than 7. It is absolute now and not recommended. It directly get the reference to the root directory of your external storage or SD Card.

2. **getExternalFilesDir(String type) –** It is recommended way to enable us to create private files specific to app and files are removed as app is uninstalled. Example is app private data.

3. **getExternalStoragePublicDirectory() :** This is current recommended way that enable us to keep files public and are not deleted with the app uninstallation. Example images clicked by the camera exists even we uninstall the camera app.

FileInputStream:

This class reads the data from a specific file (byte by byte). It is usually used to read the contents of a file with raw bytes, such as images.

To read the contents of a file using this class −

First of all, you need to instantiate this class by passing a String variable or a File object, representing the path of the file to be read.

```
FileInputStream inputStream = new FileInputStream("file_path");

or,

File file = new File("file_path");

FileInputStream inputStream = new FileInputStream(file);
```

Then read the contents of the specified file using either of the variants of read() method −

- **int read()** − This simply reads data from the current InputStream and returns the read data byte by byte (in integer format).

  This method returns -1 if the end of the file is reached.

- **int read(byte[] b)** − This method accepts a byte array as parameter and reads the contents of the current InputStream, to the given array

  This method returns an integer representing the total number of bytes or, -1 if the end of the file is reached.

- **int read(byte[] b, int off, int len)** − This method accepts a byte array, its offset (int) and, its length (int) as parameters and reads the contents of the current InputStream, to the given array.

  This method returns an integer representing the total number of bytes or, -1 if the end of the file is reached.

FileOutputStream:

This writes data into a specific file or, file descriptor (byte by byte). It is usually used to write the contents of a file with raw bytes, such as images.

To write the contents of a file using this class −

First of all, you need to instantiate this class by passing a String variable or a **File** object, representing the path of the file to be read.

```
FileOutputStream outputStream = new FileOutputStream("file_path");

or,

File file = new File("file_path");

FileOutputStream outputStream = new FileOutputStream (file);
```

You can also instantiate a FileOutputStream class by passing a FileDescriptor object.

```
FileDescriptor descriptor = new FileDescriptor();

FileOutputStream outputStream = new FileOutputStream(descriptor);
```

Then write the data to a specified file using either of the variants of write() method −

- **int write(int b)** − This method accepts a single byte and writes it to the current OutputStream.

- **int write(byte[] b)** − This method accepts a byte array as parameter and writes data from it to the current OutputStream.

- **int write(byte[] b, int off, int len)** − This method accepts a byte array, its offset (int) and, its length (int) as parameters and writes its contents to the current OutputStream.

Example 9:

Implement an application that create an alert upon receiving a Message (SMS)

Broadcast Receivers:

Broadcast Receiver simply respond to broadcast messages from other applications or from the system itself. These messages are sometime called events or intents. For example, applications can also initiate broadcasts to let other applications know that some data has been downloaded to the device and is available for them to use, so this is broadcast receiver who will intercept this communication and will initiate appropriate action.

There are following two important steps to make **BroadcastReceiver** works for the system broadcasted intents −

- Creating the Broadcast Receiver.
- Registering Broadcast Receiver

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.

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.

Now whenever your Android device gets booted, it will be intercepted by BroadcastReceiver, MyReceiver and implemented logic inside onReceive() will be executed.

There are several system generated events defined as final static fields in the Intent class. The following table lists a few important system events.

- **android.intent.action.BATTERY_LOW**
  Indicates low battery condition on the device.

- **android.intent.action.BOOT_COMPLETED**
  This is broadcast once, after the system has finished booting.

- **android.intent.action.DATE_CHANGED**
  The date has changed.

- **android.intent.action.REBOOT**
  Have the device reboot.

In this example, we will see how to receive SMS messages

Receive SMS Permissions

We only need receive permission android.permission.RECEIVE_SMS. In case you also want to read SMS messages from the Inbox then you need android.permission.READ_SMS.

```
<uses-permission android:name="android.permission.RECEIVE_SMS"/>
```

Intent Filter to receive SMS

We also need to tell Android that we want to handle incoming SMS messages. In order to do this, we will add a <receiver> to register a broadcast receiver to the manifest XML. We will also add an <intent-filter> to let Android know that we want to launch a specific class when an SMS comes in.

```
<receiver android:name="com.javarticles.android.SMSReceiver">
   <intent-filter>
      <action android:name="android.provider.Telephony.SMS_RECEIVED" />
   </intent-filter>
</receiver>
```

Broadcast Receiver

SMSReceiver is a **BroadcastReceiver**. When SMS is received, **onReceive()** will be called

```
public class SMSReceiver extends BroadcastReceiver {
   @Override
   public void onReceive(Context context, Intent intent) {
      Bundle bundle = intent.getExtras();
      Object[] pdus = (Object[]) bundle.get("pdus");
      SmsMessage[] messages = new SmsMessage[pdus.length];
         for (int i = 0; i < messages.length; i++)
         {
                  messages[i]= SmsMessage.createFromPdu((byte[]) pdus[i]);
                  String address = messages[i].getOriginatingAddress();
                  if (phoneEnrties.contains(address)) {
                  Intent newintent = new Intent(context, MainActivity.class);
                  newintent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
                  newintent.putExtra("address", address);
                  newintent.putExtra("message",
                  messages[i].getDisplayMessageBody());
                  context.startActivity(newintent);
                  }
         }
   }
}
```

Example 10:

Write a mobile application that makes use of RSS feed.

RSS:

RSS stands for Really Simple Syndication. RSS is an easy way to share your website updates and content with your users so that users might not have to visit your site daily for any kind of updates.

RSS Elements:

An RSS document such as above has the following elements.

- **channel**

  This element is used to describe the RSS feed

- **title**

  Defines the title of the channel

- **link**

  Defines the hyper link to the channel

- **description**

  Describes the channel

Parsing RSS:

Parsing an RSS document is more like parsing XML. So now lets see how to parse an XML document.

For this, We will create **XMLPullParser** object , but in order to create that we will first create **XmlPullParserFactory** object and then call its **newPullParser()** method to create **XMLPullParser**.

Its syntax is given below −

```
private XmlPullParserFactory xmlFactoryObject = XmlPullParserFactory.newInstance();
private XmlPullParser myparser = xmlFactoryObject.newPullParser();
```

The next step involves specifying the file for **XmlPullParser** that contains XML. It could be a file or could be a Stream. In our case it is a stream.

Its syntax is given below –

```
myparser.setInput(stream, null);
```

The last step is to parse the XML. An XML file consist of events , Name , Text , **AttributesValue** e.t.c. So **XMLPullParser** has a separate function for parsing each of the components of XML file

Its syntax is given below –

```
int event = myParser.getEventType();
while (event != XmlPullParser.END_DOCUMENT)  {
  String name=myParser.getName();

  switch (event){
    case XmlPullParser.START_TAG:
    break;

    case XmlPullParser.END_TAG:
    if(name.equals("temperature")){
      temperature = myParser.getAttributeValue(null,"value");
    }
    break;
  }
  event = myParser.next();
}
```

The method **getEventType** returns the type of event that happens. e.g: Document start, tag start e.t.c. The method **getName** returns the name of the tag and since we are only interested in temperature, so we just check in conditional statement that if we got a temperature tag, we call the method **getAttributeValue** to return us the value of temperature tag.

Apart from the these methods, there are other methods provided by this class for better parsing XML files. These methods are listed below −

- **getAttributeCount()**

  This method just Returns the number of attributes of the current start tag.

- **getAttributeName(int index)**

  This method returns the name of the attribute specified by the index value.

- **getColumnNumber()**

  This method returns the Returns the current column number, starting from 0.

- **getDepth()**

  This method returns Returns the current depth of the element.

- **getLineNumber()**

  Returns the current line number, starting from 1.

- **getNamespace()**

  This method returns the name space URI of the current element.

- **getPrefix()**

  This method returns the prefix of the current element.

- **getName()**

  This method returns the name of the tag.

- **getText()**

  This method returns the text for that particular element.

- **isWhitespace()**

  This method checks whether the current TEXT event contains only white space characters.

Example 11:

Develop a mobile application send an Email.

Email is messages distributed by electronic means from one system user to one or more recipients via a network.

Before starting Email Activity, You must know Email functionality with intent, Intent is carrying data from one component to another component with-in the application or outside the application.

To send an email from your application, you don't have to implement an email client from the beginning, but you can use an existing one like the default Email app provided from Android, Gmail, Outlook, K-9 Mail etc. For this purpose, we need to write an Activity that launches an email client, using an implicit Intent with the right action and data. In this example, we are going to send an email from our app by using an Intent object that launches existing email clients.

Following section explains different parts of our Intent object required to send an email.

Intent Object - Action to send Email:

You will use **ACTION_SEND** action to launch an email client installed on your Android device. Following is simple syntax to create an intent with ACTION_SEND action.

```
Intent emailIntent = new Intent(Intent.ACTION_SEND);
```

Intent Object - Data/Type to send Email:

To send an email you need to specify **mailto:** as URI using setData() method and data type will be to **text/plain** using setType() method as follows −

```
emailIntent.setData(Uri.parse("mailto:"));
emailIntent.setType("text/plain");
```

Intent Object - Extra to send Email:

Android has built-in support to add TO, SUBJECT, CC, TEXT etc. fields which can be attached to the intent before sending the intent to a target email client.

You can use following extra fields in your email −

- **EXTRA_BCC**

  A String[] holding e-mail addresses that should be blind carbon copied.

- **EXTRA_CC**

  A String[] holding e-mail addresses that should be carbon copied.

- **EXTRA_EMAIL**

  A String[] holding e-mail addresses that should be delivered to.

- **EXTRA_HTML_TEXT**

  A constant String that is associated with the Intent, used with ACTION_SEND to supply an alternative to EXTRA_TEXT as HTML formatted text.

- **EXTRA_SUBJECT**

  A constant string holding the desired subject line of a message.

- **EXTRA_TEXT**

  A constant CharSequence that is associated with the Intent, used with ACTION_SEND to supply the literal data to be sent.

- **EXTRA_TITLE**

  A CharSequence dialog title to provide to the user when used with a ACTION_CHOOSER.