Module 3 : UI Design and Data storage

UI components: -Layout: Linear, Absolute, Table, Frame. - Views: Text, Edit, Button, ImageButton, CheckBox, ToggleButton, RadioButton, RadioGroup, List, Image, Grid . Menus – Options, Context- Action bar, Notifications- data storage in Android- various storage technologies- operations for data storage and retrieval to/from internal and external memory - SQLite database- - content Providers and their relative advantages and disadvantages - SMS service in Android - publish application in Google Play Store.

Understanding The Components Of A Screen

The basic unit of an Android application is an *activity*. An *activity* displays the user interface of your application, which may contain widgets like buttons, labels, text boxes, and so on. Typically, you define your UI using an XML file (e.g., the main.xml fi le located in the res/layout folder), which may look like this:

```
<?xml version=”1.0” encoding=”utf-8”?>
<LinearLayout xmlns:android=”http://schemas.android.com/apk/res/android”
        android:orientation=”vertical”

        android:layout_width=”fill_parent”
        android:layout_height=”fill_parent”
        >
<TextView
        android:layout_width=”fill_parent”
        android:layout_height=”wrap_content”
        android:text=”@string/hello”
        />
</LinearLayout>
```

During run time, you load the XML UI in the onCreate() event handler in your Activity class, using the setContentView() method of the Activity class:

```
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
}
```

During compilation, each element in the XML file is compiled into its equivalent Android GUI class, with attributes represented by methods. The Android system then creates the UI of the activity when it is loaded.

Views And Viewgroups

An activity contains Views and ViewGroups. A view is a widget that has an appearance on screen. Examples of views are buttons, labels, and text boxes. A view derives from the base class android.view.View. One or more views can be grouped together into a ViewGroup. A ViewGroup (which is itself a special type of view) provides the layout in which you can order the appearance and sequence of views. Examples of ViewGroups include LinearLayout and FrameLayout. A ViewGroup derives from the base class android.view.ViewGroup.

Android supports the following ViewGroups:

- ➤➤ LinearLayout
- ➤➤ AbsoluteLayout
- ➤➤ TableLayout
- ➤➤ RelativeLayout
- ➤➤ FrameLayout
- ➤➤ ScrollView

The following sections describe each of these ViewGroups in more detail. Note that in practice it is common to combine different types of layouts to create the UI you want.

Linear Layout

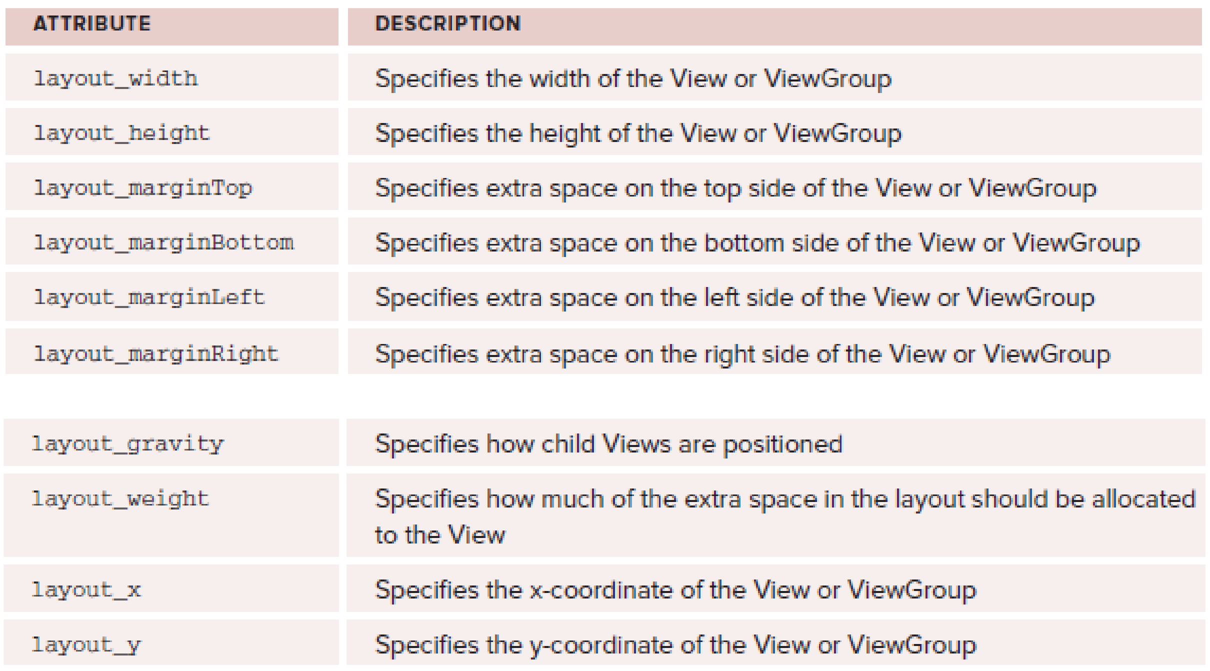
The Linear Layout arranges views in a single column or a single row. Child views can be arranged either vertically or horizontally. To see how Linear Layout works, consider the following elements typically contained in the main.xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
>
<TextView
android:layout_width="fill_parent"
```

```
android:layout_height=”wrap_content”
android:text=”@string/hello”
/>
</LinearLayout>
```

In the main.xml file, observe that the root element is <LinearLayout> and it has a <TextView> element contained within it. The <LinearLayout> element controls the order in which the views contained within it appear. Each View and ViewGroup has a set of common attributes, some of which are described in Table 3-1.

Tab le 3-1: Common Attributes Used in Views and ViewGroups



ATTRIBUTE	DESCRIPTION
`layout_width`	Specifies the width of the View or ViewGroup
`layout_height`	Specifies the height of the View or ViewGroup
`layout_marginTop`	Specifies extra space on the top side of the View or ViewGroup
`layout_marginBottom`	Specifies extra space on the bottom side of the View or ViewGroup
`layout_marginLeft`	Specifies extra space on the left side of the View or ViewGroup
`layout_marginRight`	Specifies extra space on the right side of the View or ViewGroup
`layout_gravity`	Specifies how child Views are positioned
`layout_weight`	Specifies how much of the extra space in the layout should be allocated to the View
`layout_x`	Specifies the x-coordinate of the View or ViewGroup
`layout_y`	Specifies the y-coordinate of the View or ViewGroup

NOTE Some of these attributes are applicable only when a View is in a specific ViewGroup. For example, the layout_weight and layout_gravity attributes are applicable only when a View is in either a LinearLayout or a TableLayout.

For example, the width of the <TextView> element fills the entire width of its parent (which is the screen in this case) using the fill_parent constant. Its height is indicated by the wrap_content constant, which means that its height is the height of its content (in this case, the text contained within it). If you don’t want to have the <TextView> view occupy the entire row, you can set its layout_width attribute to wrap_content, like this:

```
< TextView
android:layout_width=”wrap_content
android:layout_height=”wrap_content”
```

```
android:text=”@string/hello”
/>
```

This will set the width of the view to be equal to the width of the text contained within it.

Consider the following layout:

```
<?xml version=”1.0” encoding=”utf-8”?>
<LinearLayout xmlns:android=”http://schemas.android.com/apk/res/android”
android:orientation=”vertical”
android:layout_width=”fill_parent”
android:layout_height=”fill_parent”
>
<TextView
android:layout_width=”105dp”
android:layout_height=”wrap_content”
android:text=”@string/hello”
/>
<Button
android:layout_width=”160dp”
android:layout_height=”wrap_content”
android:text=”Button”
/>
</LinearLayout>
```