# Unit – 2

## Application Development - II

# Toast

- 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 are not clickable.

# Toast

- First, instantiate a Toast object with one of the makeText() methods.
- makeText() 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(), as shown in the following example:

  ***Toast toast = Toast.makeText(context, text, duration);***
  ***toast.show();***

# Toast

- You can also chain your methods and avoid holding on to the Toast object, like:

  ***Toast.makeText(context, text, duration).show();***
- 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

  ***toast.setGravity(Gravity.TOP|Gravity.LEFT, 0, 0);***

# Alert Dialog

- A Dialog is small window that prompts the user to a decision or enter additional information.
- Some times in your application, if you want to ask the user about taking a decision between yes or no in response of any particular action taken by the user, by remaining in the same activity and without changing the screen, you can use Alert Dialog.

# Alert Dialog

- In order to make an alert dialog, you need to make an object of AlertDialogBuilder, which an inner class of AlertDialog. Its syntax is given below:

  ***AlertDialog.Builder alertDialogBuilder = new AlertDialog.Builder(this);***

# Alert Dialog

- Now you have to set the positive (yes) or negative (no) button using the object of the AlertDialogBuilder class. Its syntax is:

  ***alertDialogBuilder.setPositiveButton(CharSequence text, DialogInterface.OnClickListener listener)***

  ***alertDialogBuilder.setNegativeButton(CharSequence text, DialogInterface.OnClickListener listener)***

# Alert Dialog

- Some popular methods of builder class

1. setTitle(CharSequence title)
   - This method sets the title to be appear in the dialog
2. setCancelable(boolean cancelable)
   - This method sets the property that the dialog can be cancelled or not
3. setMessage(CharSequence message)
   - This method sets the message to be displayed in the alert dialog
4. setIcon(Drawable icon)
   - This method set the icon of the alert dialog box

# Alert Dialog

- After creating and setting the dialog builder , you will create an alert dialog by calling the **create()** method of the builder class.
- Its syntax is

  ***AlertDialog alertDialog = alertDialogBuilder.create();***
  ***alertDialog.show();***
- This will create the alert dialog and will show it on the screen

# User Interfaces

- Following are terminologies regarding UI in android.

1. Views
2. View Groups
3. Fragments
4. Activities

# Views

- Views are the base class for all visual interface elements aka controls or widgets.
- A View derives from the base class ***android.view.View***
- An activity contains *Views and ViewGroups*.
- *A view is a widget that has an appearance on screen*.
- Examples of Views are buttons, labels, and textboxes.

# View Groups

- 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.
- View Groups are extensions of the View class that can contain multiple child Views.
- A ViewGroup derives from the base class ***android.view.ViewGroup***.

# Fragments

- Fragments are used to encapsulate portions of your UI, to optimize UI layouts for different screen sizes and creating reusable UI elements.
- Each Fragment includes its own UI layout and receives the related input events, but is tightly bound to the Activity into which each must be embedded.

# Activities

- Activities represent the window, or screen, being displayed.
- Activities are the Android equivalent of Forms in traditional Windows desktop development.
- To display a UI, you assign a View , a Layout, or a Fragment to an activity.

# Views

- **Basic views** — Commonly used views such as the TextView, EditText, and Button views
- **Picker views** — Views that enable users to select from a list, such as the TimePicker and DatePicker views
- **List views —** Views that display a long list of items, such as the List View and the Spinner View views

# Basic Views

- **ImageView** A view that displays an image
- **CheckBox** A box that can be checked or unchecked
- **RadioGroup** A group of radio buttons
- **RadioButton** A button that may be selected or unselected. Out of all the radio buttons in a RadioGroup, at most one button can be in selected state at any point in time. Thus, selecting one radio button automatically deselects all other in the same group
- **ToggleButton** A button that may be in either on or off state at any point in time
- **Button** A button that can be clicked to invoke an event handler
- **ImageButton** A button with an image on it instead of text
- **TextView** An uneditable view for displaying text
- **EditText** An editable view for displaying and editing text

# Basic Views

- **TextView:** TextView is an uneditable view for displaying text
- Example:

  ***.. // In onCreate() ..***

  ***TextView textViewMessage = (TextView)findViewById(R.id.textViewMessage);***

  ***.. textViewMessage.setText("Welcome");***

# Basic Views

- **Button:** A button is a view that can be clicked to invoke an event handler
- Example:

```
.. // In onCreate()...
Button buttonOK = (Button)findViewById(R.id.buttonOK);
buttonOK.setOnClickListener(new View.OnClickListener()
{
public void onClick(View view)
{
    ..
    ..
}
});
```

# Picker Views

- **DatePicker** A view used for input of a date
- **TimePicker** A view used for input of a time value

# List Views

- **ListView** A view holding a scrollable list of list items
- **Spinner** A dropdown list of items from which one item is selected

# List View

- In the OnCreate() method, you use the ***setListAdapter()*** method to programmatically fill the entire screen of the activity with a ListView.
- The ***ArrayAdapter*** object manages the array of strings that are ***simple_list_item_1*** mode.

***ArrayAdapter<String> adapter =***

***new ArrayAdapter<>(this, android.R.layout.simple_list_item_1, values); ("values" is the array of strings in this example)***

- The value of adapter is assigned to list view as:

  ***lstvu.setAdapter(adapter);***

# ListView

# ViewGroups

- 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

# Constraint Layout

- Constraint layout is now the default layout in Android Studio.
- It gives you many ways to place objects.
- You can constrain them to their container, to each other or to guidelines.
- this allows you to create large, complex, dynamic and responsive views in a flat hierarchy.
- It supports animations, too.
- This layout is not the best choice for simple layouts; it is suitable for complex layouts.

# Linear Layout

- The Linear Layout arranges views in a single column or a single row.
- Child views can be arranged either vertically or horizontally.
- In the main.xml file, the root element is <LinearLayout> and it has other elements contained within it.
- The <LinearLayout> element controls the order in which the views contained within it appear.

# Units of Measurements

- **dp** — Density-independent pixel. 1 dp is equivalent to one pixel on a 160 dpi screen. This is the recommended unit of measurement when specifying the dimension of views in your layout. You can specify either “dp” or “dip” when referring to a density-independent pixel.
- **sp** — Scale-independent pixel. This is similar to dp and is recommended for specifying font sizes.
- **pt** — Point. A point is defined to be 1/72 of an inch, based on the physical screen size.
- **px** — Pixel. Corresponds to actual pixels on the screen. Using this unit is not recommended, as your UI may not render correctly on devices with a different screen resolution.

# Absolute Layout

- The AbsoluteLayout enables you to specify the exact location of its children.
- **<AbsoluteLayout**
  android:layout_width=”fill_parent”
  android:layout_height=”fill_parent”
  xmlns:android=”http://schemas.android.com/apk/res/android” >
  **<Button**
  android:layout_width=”188dp”
  android:layout_height=”wrap_content”
  android:text=”Button”
  **android:layout_x=”126px”**
  **android:layout_y=”361px”** />
  **</AbsoluteLayout>**

# Absolute Layout

- However, there is a problem with the AbsoluteLayout when the activity is viewed on a highresolution screen.
- For this reason, the AbsoluteLayout has been deprecated since Android 1.5 (although it is still supported in the current version).
- One should avoid using the AbsoluteLayout in UI, as it is not guaranteed to be supported in future versions of Android.

# Table Layout

- The TableLayout groups views into rows and columns. You use the <TableRow> element to designate a row in the table.
- Each row can contain one or more views.
- Each view you place within a row forms a cell.
- The width of each column is determined by the largest width of each cell in that column.

# Table Layout

```
<TableLayout
   xmlns:android=”http://schemas.android.com/apk/res/android”
   android:layout_height=”fill_parent”
   android:layout_width=”fill_parent” >
   <TableRow>
      <TextView
   android:text=”User Name:”
   android:width =”120dp”
   />
    <EditText
   android:id=”@+id/txtUserName”
   android:width=”200dp”
    />
   </TableRow>
</TableLayout>
```

# Relative Layout

- The Relative Layout enables you to specify how child views are positioned relative to each other.
- Notice that each view embedded within the Relative Layout has attributes that enable it to align with another view.

```
<RelativeLayout
    android:id=”@+id/RLayout”
    android:layout_width=”fill_parent”
    android:layout_height=”fill_parent”
    xmlns:android=”http://schemas.android.com/apk/res/android” >
  <Button
    android:id=”@+id/btnCancel”
    android:layout_width=”124px”
    android:layout_height=”wrap_content”
    android:text=”Cancel”
    android:layout_below=”@+id/txtComments”
    android:layout_alignLeft=”@+id/txtComments” />
</RelativeLayout>
```

# Frame Layout

- The Frame Layout is a placeholder on screen that you can use to display a single view.
- Views that you add to a Frame Layout are always anchored to the top left of the layout.
- If you add another view (such as a Button view) within the Frame Layout, the view will overlap the previous view.
- You can add multiple views to a Frame Layout, but each will be stacked on top of the previous one.
- This is when you want to animate a series of images, with only one visible at a time.

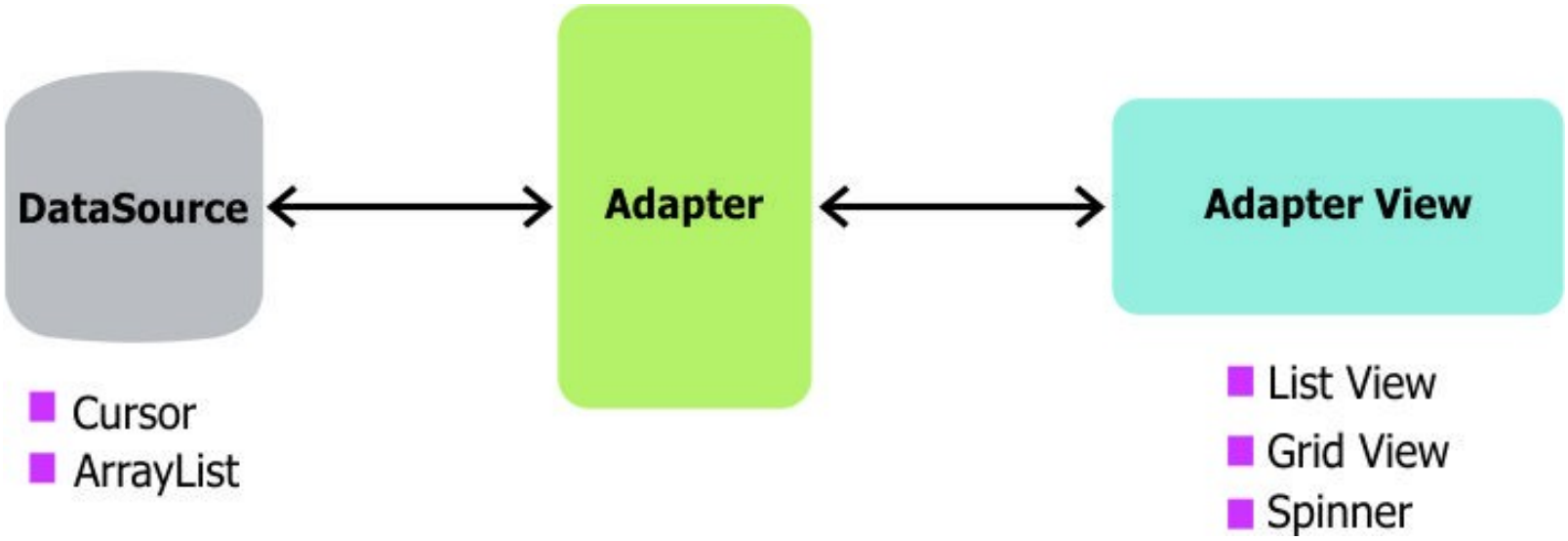
# Frame Layout

```
<RelativeLayout
    android:id=”@+id/RLayout”
    android:layout_width=”fill_parent”
    android:layout_height=”fill_parent”
    xmlns:android=”http://schemas.android.com/apk/res/android”>
    <FrameLayout
    android:layout_width=”wrap_content”
    android:layout_height=”wrap_content”
    android:layout_alignLeft=”@+id/lblComments”
    android:layout_below=”@+id/lblComments”
    android:layout_centerHorizontal=”true” >
        <ImageView
        android:src = “@drawable/droid”
        android:layout_width=”wrap_content”
        android:layout_height=”wrap_content” />
        <Button
        android:layout_width=”124dp”
        android:layout_height=”wrap_content”
        android:text=”Print Picture” />
    </FrameLayout>
</RelativeLayout>
```

# Adapter

- In Android, Adapter is a bridge between UI component and data source that helps us to fill data in UI component.
- It holds the data and send the data to an Adapter view then view can take the data from the adapter view and shows the data on different views like as ListView, Spinner etc.
- The Adapter is also responsible for making a View for each item in the data set.

# Adapter


DataSource
Adapter
Adapter View
Cursor
ArrayList
List View
Grid View
Spinner

# Array Adapters

- Whenever we have a list of single items which is backed by an Array, we can use ArrayAdapter.
- For instance, list of phone contacts, countries or names.
- Array Adapter method accepts four arguments as below:
- ***ArrayAdapter(Context context, int resource, int textViewResourceId, T[] objects)***

# Cursor Adapter

- In Android development, any time you want to show a vertical list of items you will want to use a ListView which is populated using an Adapter to a data source.
- When we want the data for the list to be sourced directly from a SQLite database query, we can use a CursorAdapter.
- The CursorAdapter fits in between a Cursor (data source from SQLite query) and the ListView (visual representation) and configures two aspects:
  - Which layout template to inflate for an item
  - Which fields of the cursor to bind to which views in the template

# Intent

- An intent is an abstract description of an operation to be performed.
- Intents are messages used for communication between components (like activities, services, broadcast receivers, etc) of the same app or different apps
- **Explicit Intent**
  - One may use an explicit intent to send the message to a specific target activity or service.
  - This type of intent specifies the class of the component (activity / service) to be used for performing the operation.
  - The system will hand over the intent to the specified activity or service for taking appropriate action
  - Explicit intents specify which application will satisfy the intent, by supplying either the target app's package name or a fully-qualified component class name.

# Intent

- **Implicit Intent**
  - An implicit intent only specifies the operation to be performed.
  - It does not specify the component that may perform the operation.
  - The Android operating system uses its own internal algorithm to decide which component will handle the intent
  - Implicit intents do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it.

# Notifications

- Notifications are a user notification framework.
- Notifications let you signal users without stealing focus or interrupting their current Activities.
- They're the preferred technique for getting a user's attention when your activity does not have the focus, say, from within a service or broadcast Receiver.
- For example, when a device receives a text message or misses an incoming call, it alerts you by flashing lights, making sounds, displaying icons, or showing messages.
- There is a notification pull-down where these notifications are stored until the user acts on them or dismisses them.
- On newer versions, the notifications in the notification pull-down may also include images or actionable elements, like buttons.

# Notifications

- To get started, you need to set the notification's content and channel using a ***NotificationCompat.Builder*** object. The following example shows how to create a notification with the following:

1. A small icon, set by ***setSmallIcon()***. This is the only user-visible content that's required.
2. A title, set by ***setContentTitle()***.
3. The body text, set by ***setContentText()***.
4. The notification priority, set by ***setPriority()***.

- To make the notification appear, call ***NotificationManagerCompat.notify()***, passing it a unique ID for the notification and the result of ***NotificationCompat.Builder.build()***.

# Notifications

```
NotificationCompat.Builder builder = new
    NotificationCompat.Builder(this, CHANNEL_ID)
            .setSmallIcon(R.drawable.notification_icon)
            .setContentTitle(textTitle)
            .setContentText(textContent)
            .setPriority(NotificationCompat.PRIORITY_DEFAULT);
NotificationManagerCompat.notify(notification_ID,builder.build());
```

- Notice that the ***NotificationCompat.Builder*** constructor requires that you provide a channel ID.
- This is required for compatibility with Android 8.0 (API level 26) and higher, but is ignored by older versions.

# Notification Channel

- Starting in Android 8.0 (API level 26), all notifications must be assigned to a channel.
- For each channel, you can set the visual and auditory behaviour that is applied to all notifications in that channel.
- Then, users can change these settings and decide which notification channels from your app should be intrusive or visible at all.

# Notification Channel

- To create a notification channel, follow these steps:

1. Construct a ***NotificationChannel*** object with a unique channel ID, a user-visible name, and an importance level.
2. Optionally, specify the description that the user sees in the system settings with ***setDescription()***.
3. Register the notification channel by passing it to ***createNotificationChannel()***.