
UNIT-III

1. ANDROID UI DESIGN ESSENTIALS

Creating an effective and user-friendly interface in Android involves understanding and implementing key design essentials. Here are some crucial aspects to consider when designing an Android UI:

1. Understanding the Material Design Guidelines

Google's Material Design is a design language that offers comprehensive guidance on the design and interaction patterns for Android apps. It includes specifications for layout structure, components, animations, transitions, and more. Adhering to these guidelines ensures your app feels familiar to users and integrates well with the Android ecosystem.

2. Layout and Structure

Responsive Design: Design your UI to be responsive across a wide range of device sizes and orientations. Use constraints and relative sizing to make your layout adapt to different screen sizes.

Navigation: Implement clear and intuitive navigation. Consider the use of navigation drawers, bottom navigation bars, and proper back stack management to enhance user experience.

Consistency: Maintain consistency in the UI elements such as buttons, fonts, and color schemes across different screens to create a cohesive experience.

3. Typography

Readability: Choose font sizes and styles that are easy to read. Material Design suggests a scale for typography with specific styles for different elements like headlines, titles, body text, etc.

Hierarchy: Use typography to establish a clear hierarchy of information, making it easier for users to scan and understand the content.

4. Color and Theme

Color Scheme: Select a primary and secondary color that reflects your brand and is accessible. Use contrasting colors for text to ensure readability.

Dark Theme: Consider supporting a dark theme to reduce eye strain in low light conditions and save battery life on OLED screens.

5. Icons and Imagery

High Quality: Use high-resolution images and icons to prevent pixelation on devices with high-density displays.

Adaptive Icons: Design adaptive icons for app icons to ensure they maintain a consistent appearance across different device models and launcher themes.

6. User Input and Interaction

Feedback: Provide immediate feedback in response to user interactions (e.g., visual or haptic feedback when buttons are pressed).

Form Design: Design forms to be efficient and easy to use. Group related information, and use appropriate input types and validation to enhance usability.

7. Accessibility

Contrast Ratios: Ensure sufficient contrast between text and background colors to aid readability for users with vision impairments.

Touchable Targets: Make touch targets (e.g., buttons, links) large enough to be easily tapped without precision.

Content Descriptions: Use content descriptions for non-text elements (like images) so screen readers can describe them to users with visual impairments.

8. Performance and Optimization

Avoid Overdraw: Minimize the number of layers and transparent backgrounds to reduce overdraw, where pixels are painted multiple times in a single frame, which can affect performance.

Image Loading and Caching: Use libraries like Glide or Picasso to efficiently load and cache images, reducing memory usage and speeding up image display.

9. Testing Across Devices

Device Fragmentation: Test your app on multiple devices with different screen sizes, resolutions, and Android versions to ensure consistent behavior and appearance.

10. Following Latest Trends and Feedback

Stay Updated: Keep an eye on the latest UI/UX trends and Android features.

User Feedback: Actively seek and incorporate user feedback to refine and improve the UI/UX of your app.

Incorporating these essentials into your Android UI design will help create an app that is not only visually appealing but also functional, accessible, and intuitive for users.

2. USER INTERFACE SCREEN ELEMENTS IN ANDROID

In Android development, the user interface (UI) is constructed using a hierarchy of views and view groups. Views are basic building blocks of UI elements such as buttons, text fields, images, etc., while View Groups are containers that hold and organize a collection of views and other view groups to define the layout structure.

2.1 Basic UI Elements

1. **Text View**: Displays text to the user.
2. **Edit Text**: Allows users to enter and modify text.
3. **Button**: A clickable element used for triggering actions.
4. **Image View**: Displays an image.
5. **Image Button**: A button with an image (instead of text) that can be clicked.
6. **Checkbox**: Allows users to select one or more options from a set.
7. **Radio Button**: Allows users to select one option from a set. Often used with RadioGroup to group multiple options.
8. **Toggle Button**: Offers a two-state toggle that can be either checked or unchecked.
9. **Switch**: A two-state toggle switch widget that can select between two options.
10. **ProgressBar**: Shows the progress of an operation. Can be in a spinning wheel or horizontal bar form.
11. **SeekBar**: A variation of ProgressBar that allows for user interaction by sliding a thumb to adjust the value.
12. **Spinner**: A dropdown list that allows users to select one value from a list.
13. **ScrollView**: A view group that allows its child views to be scrolled vertically.

2.2 Layouts (View Groups)

1. **Linear Layout**: Positions its children in a single direction, vertically or horizontally, based on its orientation attribute.
2. **Relative Layout**: Enables you to specify the location of child views relative to each other or to the parent view.
3. **Frame Layout**: Designed to block out an area on the screen to display a single item. You can add multiple children to a Frame Layout, but all children are pinned to the top left of the screen.
4. **Constraint Layout**: Offers more flexibility in positioning its children using constraints, making it possible to create complex and flexible layouts with a flat view hierarchy.
5. **Table Layout**: Groups views into rows and columns.
6. **Grid Layout**: Positions its child views in a grid of rows and columns, similar to TableLayout but more flexible.

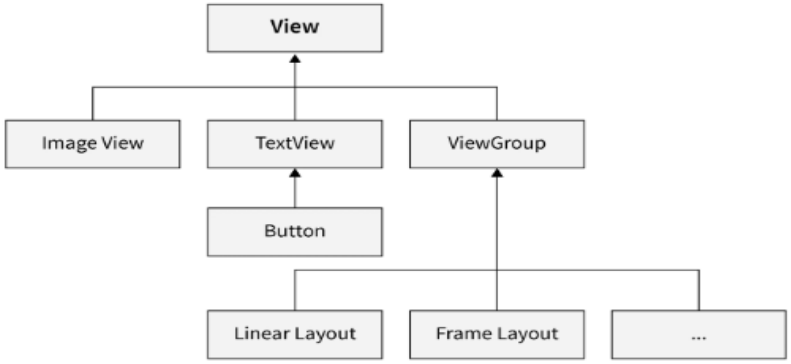
2.3 Advanced UI Elements

1. **Navigation View**: Typically used for navigation drawers.
2. **View Pager**: Allows users to swipe left or right to view an entire set of pages or tabs.
3. **Recycler View**: An advanced and flexible version of List View that supports large datasets by reusing view items as they scroll off-screen.
4. **Card View**: Displays information inside cards with a consistent look across the app.
5. **Snack bar**: Provides lightweight feedback about an operation by showing a brief message at the bottom of the screen.

When designing the UI for an Android app, developers typically use XML files to define the layout and appearance of these UI elements, though they can also be created and manipulated programmatically in Java or Kotlin code. The Android Studio IDE provides a visual layout editor that helps in designing UIs more efficiently.

3. DESIGNING USER INTERFACE WITH LAYOUTS IN ANDROID

Designing a user interface (UI) in Android involves using different types of layouts to organize UI components on the screen effectively. Layouts are essentially view groups that control how child views are positioned and displayed. Each type of layout comes with its own set of attributes for arranging child views in a specific manner. Here's an overview of designing UI with layouts in Android:



1. Understanding Core Layout Types

- **Linear Layout**: Arranges its child views in a single column or row, depending on its orientation attribute (**vertical** or **horizontal**). It's useful for creating simple layouts like forms.
- **Relative Layout**: Allows positioning child views in relation to each other or to the parent. It's versatile for creating complex layouts, but performance-wise it's generally better to use Constraint Layout for more complex arrangements.
- **Frame Layout**: Designed to block out an area on the screen to display a single item. You can stack multiple children, but they will be positioned based on the top left of the screen unless you use gravity to change their alignment. It's useful for simple scenarios or as a container for fragments.
- **Constraint Layout**: Offers a high level of flexibility and control, allowing you to create large and complex layouts with a flat view hierarchy (no nested view groups). It uses constraints to define the position of each view according to relationships between sibling views and the parent layout, which helps in building responsive UI designs.
- **Table Layout**: Works in a way similar to HTML tables, organizing child views into rows and columns. Not commonly used for dynamic content or complex layouts.
- **Grid Layout**: Similar to Table Layout but more flexible, allowing you to define the structure of a grid and place child views into specific positions within the grid.

2. Designing for Different Screen Sizes

To ensure your UI looks good on different devices:

- **Use Constraint Layout**: It offers the most flexibility for creating responsive designs that can adjust to various screen sizes and orientations.
- **Provide Multiple Layouts**: Create different layouts for different screen sizes by placing them in the appropriate resource folders (e.g., **layout-large**, **layout-xlarge**).

- **Use Size Qualifiers**: Android allows you to specify different resources for different screen sizes, orientations, and densities. Utilize these qualifiers to provide optimized layouts and drawable resources.

3. Implementing Layouts

When implementing layouts, you'll typically do so in XML:

- **Define the Layout**: Create an XML file in the **res/layout** directory. Start with a root layout tag (e.g., **<LinearLayout>**, **<RelativeLayout>**, **<ConstraintLayout>**) and nest other UI elements inside.
- **Specify Layout Properties**: Each view and layout has a set of XML attributes that you can use to define its properties, such as width, height, layout constraints, padding, margins, and alignment.
- **Preview and Test**: Use the layout editor in Android Studio to preview your layout on different devices and screen sizes. Always test on actual devices or emulators to ensure the UI appears as expected.

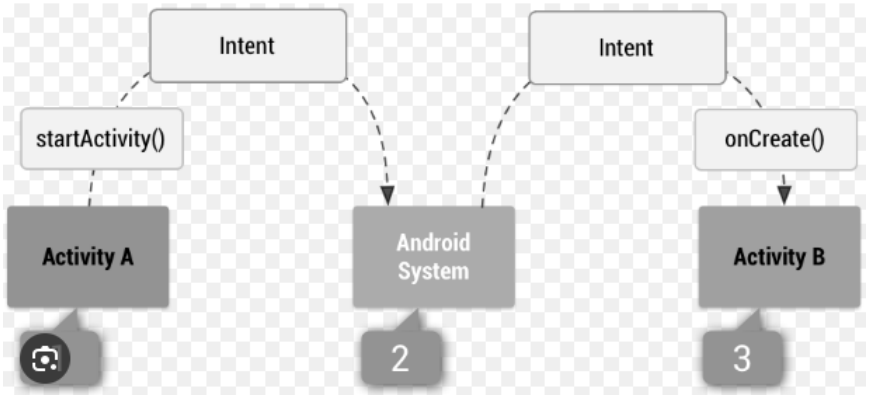
4. Best Practices

- **Keep Layouts Simple**: Aim for a flat layout hierarchy because deeply nested views can impact performance.
- **Reuse Components**: Use **<include>** and **<merge>** tags to reuse common layout parts and reduce code duplication.
- **Consider Accessibility**: Ensure your layouts are accessible. Provide descriptive content descriptions for interactive elements, ensure touch targets are adequately sized, and test with screen readers.
- **Optimize for Performance**: Use tools like the Layout Inspector and Hierarchy Viewer to analyze your layouts and identify potential performance bottlenecks.

In summary, effective UI design in Android requires a solid understanding of the available layouts and how to use them to create responsive, efficient, and accessible interfaces. Always consider the user experience and test your designs on a variety of devices and configurations to ensure the best possible outcome.

4. DESIGNING USER INTERFACE USING INTENT FILTER IN ANDROID

Designing a user interface (UI) in Android does not directly involve using intent filters. Intent filters are not UI elements but a way for your app to handle incoming data or requests. However, understanding how intent filters work is crucial for creating a seamless user experience, especially when your app interacts with other apps or handles various kinds of data and actions.



What are Intent Filters?

Intent filters are XML elements in your app's manifest file (**AndroidManifest.xml**) that specify the types of intents your app can respond to. When an app (including the system) broadcasts an intent, the Android system checks which apps have intent filters compatible with the intent. If your app has a matching intent filter, it can be started or receive the data from that intent, depending on the intent type.

Use Cases in UI Design

While intent filters themselves are not UI components, they influence the user experience in several key scenarios:

1. Opening Links or Custom Data from Other Apps

If your app can handle specific data types or URI schemes, you can use intent filters to open your app from links in other apps. For example, if you're developing a social media app and want links starting with **https://example.com/** to open in your app, you would configure an intent filter to handle those URIs.

```
<intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="https" android:host="example.com" />
</intent-filter>
```

2. Sharing Content

If your app can share content like images or text, you can declare an intent filter for the **SEND** action, allowing other apps to share content through your app.

```
<intent-filter>
    <action android:name="android.intent.action.SEND" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="image/*" />
</intent-filter>
```

3. Handling External Actions

Your app might perform specific actions, such as taking a photo or picking a contact. By setting an intent filter, your app can be listed as an option when a user wants to perform these actions, enhancing the integrated experience across apps.

```
<intent-filter>
    <action android:name="android.media.action.IMAGE_CAPTURE" />
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

Design Considerations

- **User Experience**: When your app can handle intents from other apps, consider the user experience. Ensure that your app processes the incoming data smoothly and presents relevant content or actions to the user without confusion.
- **Deep Linking**: Use intent filters for deep linking to direct users to specific content or screens within your app. This can significantly enhance the user experience by providing shortcuts to content that users are interested in.
- **Dynamic Features**: For apps that include dynamic feature modules, intent filters can be used to launch features on demand, making the app more modular and lightweight.

While intent filters are not UI elements, they play a crucial role in how your app interacts with the Android ecosystem and can significantly impact the user experience. By declaring intent filters in your app, you can ensure that your app is an integrated part of the user's device, capable of handling specific actions and data seamlessly.

5. DESIGNING USER INTERFACE WITH PERMISSIONS IN ANDROID

Designing a user interface (UI) that incorporates permissions in Android is an essential aspect of creating a seamless and user-friendly experience. Permissions are required for an app to access sensitive data (like contacts or location) or system features (like the camera or microphone). Starting with Android 6.0 (API level 23), users grant permissions to apps while the app is running, not at installation time. This model gives users more control over the app's functionality, but it also means that developers need to integrate permission requests smoothly into the UI design.

1. Permission group icon
2. App name
3. “Never ask again” check box (This is displayed if a user denies a permission twice)
4. Indicator for multiple dialogs
5. Action

Integrating Permissions into UI Design

1. **Explain Why the Permission is needed**

Before prompting for a permission, use dialogues or UI elements to explain why the app needs this permission and how it will be used. This preemptive explanation can increase the chances of the user granting the permission.

2. **Request Permissions at the Point of Need**

Request permissions in context, ideally when the user tries to access a feature that requires the permission. This makes the need for the permission more understandable and relevant.

3. **Provide Immediate Feedback**

If a permission is granted, the app should immediately proceed with the action that required the permission. If denied, consider offering an alternative way to complete the task without needing the permission.

4. **Handle Denials Gracefully**

Design your UI to handle permission denials gracefully. Inform users of the functionality they lose access to by not granting the permission and consider providing instructions on how they can enable it later if they change their mind.

5. **Consider Permissions in Onboarding**

For permissions critical to your app's functionality, consider including a permissions walkthrough in your app's onboarding process. This can help set expectations early on.

Technical Implementation

Here's a simplified example of how you might request a permission in an Android app, which is crucial for the UI design incorporating permissions:

```
if (ContextCompat.checkSelfPermission(thisActivity, Manifest.permission.ACCESS_FINE_LOCATION)
      != PackageManager.PERMISSION_GRANTED)
{  if (ActivityCompat.shouldShowRequestPermissionRationale(thisActivity,
      Manifest.permission.ACCESS_FINE_LOCATION)) {  } else {
      ActivityCompat.requestPermissions(thisActivity, new String[]{Manifest.permission.ACCESS_FINE_LOCATION},
            MY_PERMISSIONS_REQUEST_ACCESS_FINE_LOCATION);    } }
else { }
```

Best Practices

- **Minimize Permission Requests**: Only request permissions that are essential for your app to function. Unnecessary permissions can frustrate users or deter them from using the app.
- **Test Thoroughly**: Test your app's permission requests on devices running different Android versions, as the system's handling of permissions can vary.
- **Update UI Based on Permissions**: Your app's UI should adapt based on the permissions that have been granted. For example, if location permission isn't granted, hide or disable features that rely on location data.
- **Respect the User's Decision**: If a user decides not to grant a permission, respect their choice and do not repeatedly ask for the permission in a way that degrades the user experience.

Incorporating permissions into your UI design thoughtfully can lead to a better user experience, increase the likelihood of users granting permissions, and ultimately make your app more effective and enjoyable to use.

6. CREATING YOUR FIRST ACTIVITY WITH UI IN ANDROID

Creating your first activity with a user interface (UI) in Android involves several steps. An activity is a single, focused thing that the user can do. Most activities interact with the user, so the Activity class takes care of creating a window for you in which you can place your UI. Here’s a simple guide to get you started:

Step 1: Setup Your Android Development Environment

Ensure you have Android Studio installed on your computer. Android Studio is the official Integrated Development Environment (IDE) for Android app development.

Step 2: Create a New Android Project

1. Open Android Studio and click on "Start a new Android Studio project".
2. Choose a template for your project. For a simple start, you can select the "Empty Activity".
3. Enter your application name, package name, save location, and other details. Choose language as Java or Kotlin.
4. Select the minimum API level for your app. This determines which devices can run your app.

Step 3: Design the User Interface

User interfaces in Android are defined using XML. Navigate to the **res/layout** directory, and you will find a file named **activity_main.xml**. This XML file represents the layout for your activity's UI.

1. Open **activity_main.xml**.
2. Use the Palette in Android Studio to drag and drop UI components onto the canvas or the component tree view. Common components include **TextView**, **Button**, **EditText**, etc.

Here's an example where we add a **TextView** to say "Hello, World!" and a **Button**:

```
<LinearLayout xmlns:android=http://schemas.android.com/apk/res/android
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">
    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, World!"
        android:textSize="24sp" />
    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click Me" />
</LinearLayout>
```

Step 4: Add Functionality in Your Activity

Open the **MainActivity.java** or **MainActivity.kt** file. This file is where you write Java or Kotlin code to add logic to your activity.

For instance, to make the button show a toast message when clicked, add the following code inside the **onCreate** method:

If you're using Java:

```
Button button = findViewById(R.id.button);
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Toast.makeText(MainActivity.this, "Button clicked!", Toast.LENGTH_SHORT).show();
    }
});
```

If you're using Kotlin:

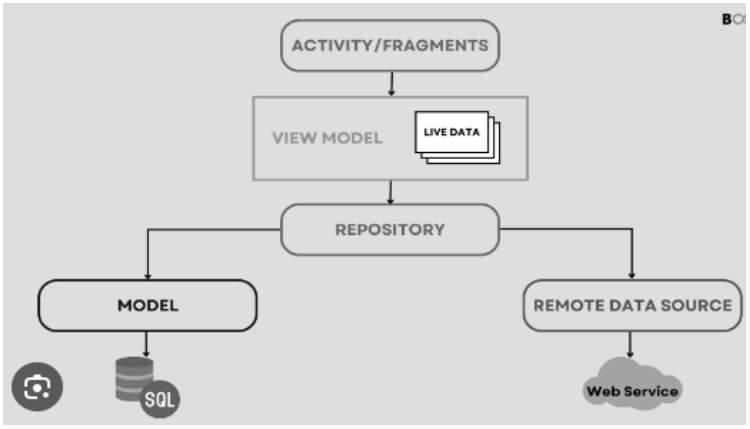
```
val button: Button = findViewById(R.id.button)
button.setOnClickListener {
    Toast.makeText(this, "Button clicked!", Toast.LENGTH_SHORT).show()
}
```

Step 5: Run Your Application

1. Click the "Run" button (a green arrow) in Android Studio.
2. Choose your device: You can run your app on a physical Android device connected to your computer or an emulator.
3. Android Studio installs your app on the device and runs it.

7. WORKING WITH ANDROID FRAMEWORK CLASSES

Working with Android framework classes is a fundamental part of Android app development. The Android framework provides a vast library of classes and interfaces that facilitate the development of robust and high-performing applications. These classes are organized into various packages, such as **android.app**, **android.content**, **android.view**, **android.widget**, and many more, each serving different aspects of app development like user interface (UI) design, data storage, system services, etc. Here's an overview of how to work with some key Android framework classes:



1. Activity and Fragment

- **Activity**: Represents a single screen with a user interface. For example, an email application might have one activity that shows a list of emails, another activity to compose an email, and another activity for reading emails. You define activities in your app by extending the **Activity** class or one of its subclasses.

Java

```
public class MainActivity extends AppCompatActivity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
   }
}
```

- **Fragment**: Represents a behavior or a portion of the user interface in an **Activity**. You can combine multiple fragments in a single activity to build a multi-pane UI and reuse a fragment in multiple activities.

Java

```
public class ExampleFragment extends Fragment {
   @Override
   public View onCreateView(LayoutInflater inflater, ViewGroup container,
                            Bundle savedInstanceState) {
      return inflater.inflate(R.layout.fragment_example, container, false);
   }
}
```

2. **Intent**

- Used for asynchronous messaging between components in your app or between your app and other apps. Intents are used for a variety of tasks, such as starting an activity, starting a service, or delivering a broadcast.
    - Intent intent = new Intent(this, AnotherActivity.class);
    - startActivity(intent);

3. Services

- A **Service** is an application component that can perform long-running operations in the background. It does not provide a user interface. **For example**, a service might play music in the background while the user is in a different application.

```
public class ExampleService extends Service {
   @Override       public int onStartCommand(Intent intent, int flags, int startId) {
          return START_NOT_STICKY;
   }
   @Override      public IBinder onBind(Intent intent) {
      return null;
   } }
```

4. Broadcast Receivers

- Used for listening and reacting to broadcast messages from other applications or from the system itself. **For example**, applications can start up when the device boots by listening to the **BOOT_COMPLETED** broadcast.

```
public class ExampleReceiver extends BroadcastReceiver {
   @Override
   public void onReceive(Context context, Intent intent) {
      // Your code to react to the broadcast
   }
}
```

5. **Content Providers**

- Manage access to a structured set of data. They encapsulate the data and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process.

```
Java
Cursor cursor = getContentResolver().query(
   Uri.parse("content://com.example.app.provider"),
   projection,
   selection,
   selectionArgs,
   sortOrder);
```

6. **View and ViewGroup**

- **View**: The basic building block for user interface components. A View occupies a rectangular area on the screen and is responsible for drawing and event handling. Button, TextView, EditText are all subclasses of View.
    - TextView textView = new TextView(this);
    - textView.setText("Hello, Android!");
    - setContentView(textView);
- **ViewGroup**: A special view that can contain other views (called children). The view group is the base class for layouts and views containers, such as **LinearLayout** and **RelativeLayout**.

```
XML
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical">
      <TextView
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello, Android!" /> </LinearLayout>
```

Working with these Android framework classes involves understanding their lifecycle, knowing how to configure them through both code and XML, and understanding how they interact with each other within an app's architecture. By mastering these components, you can create sophisticated and efficient Android applications.

------ End of Unit-III -----