UNIT-3

Designing with Layouts: Creating User Interfaces in Android- Creating Layouts Using XML Resources, Creating Layouts Programmatically, Organizing Your User Interface-Using ViewGroup Subclasses for Layout Design, Using ViewGroup Subclasses as View Containers, Using Built-in Layout Classes- Using LinearLayout, Using RelativeLayout, Using FrameLayout, Using TableLayout, Using GridLayout, Using Multiple Layouts on a Screen, Partitioning the User Interface with Fragments-Understanding Fragments, Understanding the Fragment Lifecycle, Working with Special Types of Fragments, Designing Fragment-Based Applications, Using the Android Support Package, Adding Fragment Support to Legacy Applications, Using Fragments in New Applications Targeting Older Platforms, Linking the Android Support Package to Your Project, Displaying Dialogs-**Choosing Your Dialog Implementation**, **Exploring the Different Types of Dialogs**, **Working with Dialogs and Dialog Fragments**- Tracing the Lifecycle of a Dialog and DialogFragment, Working with Custom Dialogs.

Creating Layouts Using XML Resources

Creating layouts using XML resources is a fundamental aspect of Android app development, providing a declarative way to define the structure and appearance of user interfaces. XML layouts are stored in the `res/layout` directory and offer a clear separation between design and code. For instance, a simple XML layout for displaying a TextView might look like this:

```xml
<!-- res/layout/activity_main.xml -->
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

```
    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, World!"
        android:layout_centerInParent="true" />

</RelativeLayout>
```

Creating layouts programmatically

in Android involves constructing the user interface using Java or Kotlin code rather than defining it in XML. This approach is particularly useful for dynamic UI elements that need to be created or modified at runtime. Programmatically creating layouts allows for greater flexibility and control over the UI components, enabling you to respond to user interactions or other runtime conditions more effectively.

Example in Java:

```java
// MainActivity.java
import android.os.Bundle;
import android.widget.LinearLayout;
```

```
import android.widget.TextView;

import androidx.appcompat.app.AppCompatActivity;


public class MainActivity extends AppCompatActivity {

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);


        // Create a LinearLayout instance

        LinearLayout layout = new LinearLayout(this);

        layout.setOrientation(LinearLayout.VERTICAL);


        // Create a TextView instance

        TextView textView = new TextView(this);

        textView.setText("Hello, World!");


        // Add the TextView to the LinearLayout

        layout.addView(textView);


        // Set the LinearLayout as the content view of the activity
```

```
        setContentView(layout);
    }
}
```

In this example, a `LinearLayout` is created and configured with a vertical orientation. A `TextView` is then created, and its text is set to "Hello, World!". The `TextView` is added to the `LinearLayout`, and finally, the `LinearLayout` is set as the content view of the activity using `setContentView(layout)`. This programmatic approach enables dynamic and flexible UI creation in Android applications.

Organizing Your User Interface Using ViewGroup Subclasses for Layout Design

In Android development, organizing the user interface effectively is crucial for creating intuitive and responsive applications. ViewGroup subclasses, such as `LinearLayout`, `RelativeLayout`, `FrameLayout`, `TableLayout`, and `GridLayout`, provide various ways to arrange and manage UI components. Each subclass offers unique layout capabilities: `LinearLayout` aligns children either vertically or horizontally, `RelativeLayout` positions them relative to each other, `FrameLayout` stacks them, `TableLayout` arranges them in a grid of rows and columns, and `GridLayout` provides a more flexible grid system. Utilizing these ViewGroup subclasses allows developers to create complex and well-structured user interfaces that enhance the user experience.

Using ViewGroup Subclasses as View Containers

In Android, `ViewGroup` subclasses act as containers that hold and manage the layout and behavior of child views. These subclasses, such as `LinearLayout`, `RelativeLayout`, `FrameLayout`, `TableLayout`, and `GridLayout`, provide different ways to arrange and organize user interface components. By using these ViewGroup containers, developers can create complex and responsive layouts that adapt to various screen sizes and orientations. Each `ViewGroup` subclass offers unique properties and methods that help in aligning, positioning,

and managing the visibility of its child views, enabling the creation of sophisticated and well-structured user interfaces.

Using Built-in Layout Classes

Using LinearLayout

`LinearLayout` is a versatile layout manager that arranges its child views in a single direction, either vertically or horizontally. It simplifies the process of aligning components linearly, making it ideal for creating simple, column or row-based layouts. Each child can specify its size using layout parameters like `layout_width` and `layout_height`.

Example:

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

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

```

## Using RelativeLayout

`RelativeLayout` enables more flexible positioning of child views relative to each other or to the parent layout. This makes it a powerful tool for creating more complex layouts without nesting multiple views. You can position a view to the left, right, above, or below another view using layout attributes like `layout_toLeftOf`, `layout_below`, etc.

**Example:**

```xml
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, World!" />
```
```

```
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/textView"
        android:text="Click Me" />
</RelativeLayout>
```

Using FrameLayout

`FrameLayout` is designed to block out an area on the screen to display a single item. Child views are stacked on top of each other, and only the last added view is visible. This layout is useful for creating overlay effects, where one view overlaps another.

Example:

```xml
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
```

```
        android:src="@drawable/background_image" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Overlay Text"
        android:layout_gravity="center" />
</FrameLayout>
```

Using TableLayout

`TableLayout` arranges its child views into rows and columns. Each row is defined by a `TableRow` object, which can contain multiple views. This layout is ideal for creating grid-like structures where content needs to be organized into a tabular format.

Example:

```xml
<TableLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

```
<TableRow>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Row 1, Column 1" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Row 1, Column 2" />
</TableRow>
<TableRow>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Row 2, Column 1" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Row 2, Column 2" />
```

```
    </TableRow>

</TableLayout>
```

Using GridLayout

`GridLayout` provides a more flexible grid structure compared to `TableLayout`. It allows specifying the number of rows and columns and offers more control over how views are placed within each cell. This layout is suitable for creating complex grid-based designs.

Example:

```xml
<GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:columnCount="2">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Item 1" />
    <TextView
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Item 2" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Item 3" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Item 4" />
</GridLayout>
```

Using Multiple Layouts on a Screen

Combining multiple layouts on a single screen allows for creating complex and responsive UI designs. You can nest layouts within each other to achieve the desired structure and appearance.

Example:

```xml
<LinearLayout
```

```
xmlns:android="http://schemas.android.com/apk/res/android"

android:orientation="vertical"

android:layout_width="match_parent"

android:layout_height="match_parent">


<RelativeLayout

   android:layout_width="match_parent"

   android:layout_height="wrap_content">


   <TextView

      android:id="@+id/textView"

      android:layout_width="wrap_content"

      android:layout_height="wrap_content"

      android:text="Relative Layout Text" />


   <Button

      android:layout_width="wrap_content"

      android:layout_height="wrap_content"

      android:layout_below="@id/textView"

      android:text="Relative Layout Button" />
```

```
    </RelativeLayout>

    <FrameLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <ImageView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:src="@drawable/frame_image" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Frame Layout Text"
            android:layout_gravity="center" />
    </FrameLayout>
</LinearLayout>
```

These examples demonstrate how to use various built-in layout classes to organize and structure user interfaces effectively in Android applications.

Partitioning the User Interface with Fragments

Understanding Fragments

Fragments are modular sections of an activity, representing a portion of the user interface. They enable more flexible and reusable UI designs, as they can be combined, reused, and managed independently within activities. Each fragment has its own lifecycle and can handle its own events, making it a powerful tool for creating dynamic UIs.

Understanding the Fragment Lifecycle

The fragment lifecycle is similar to the activity lifecycle, with stages such as `onAttach`, `onCreate`, `onCreateView`, `onActivityCreated`, `onStart`, `onResume`, `onPause`, `onStop`, `onDestroyView`, `onDestroy`, and `onDetach`. These methods allow you to manage the fragment's behavior and resources at different points in its existence, ensuring proper handling of user interactions and system changes.

Working with Special Types of Fragments

There are several special types of fragments designed for specific tasks, such as `DialogFragment`, which displays a floating dialog, and `ListFragment`, which simplifies the creation of lists. These specialized fragments provide additional functionality and simplify common UI patterns.

Designing Fragment-Based Applications

Fragment-based applications use fragments to create flexible and dynamic user interfaces. By modularizing the UI into fragments, developers can build more responsive and adaptable applications, allowing different layouts and behaviors for different screen sizes and orientations.

Using the Android Support Package

The Android Support Package provides libraries that enable the use of fragments and other newer Android features on older platform versions. This ensures that applications can maintain backward compatibility while taking advantage of modern UI components.

Adding Fragment Support to Legacy Applications

To add fragment support to legacy applications, integrate the Android Support Library into the project. This allows the use of `FragmentActivity` and other support library classes to manage fragments in applications targeting older Android versions.

Using Fragments in New Applications Targeting Older Platforms

When developing new applications that need to support older Android versions, use the Android Support Library to ensure compatibility. This enables the use of fragments and other modern features while maintaining functionality on older devices.

Linking the Android Support Package to Your Project

To use the Android Support Package in your project, add the appropriate dependencies to your `build.gradle` file. This includes libraries such as `com.android.support:appcompat-v7` and `com.android.support:support-fragment`, which provide the necessary classes and methods to implement fragments and other support features in your application.

By understanding and effectively utilizing fragments, developers can create modular, flexible, and backward-compatible user interfaces that enhance the overall user experience across different devices and Android versions.

Displaying Dialogs

Choosing Your Dialog Implementation

When it comes to displaying dialogs in Android, there are several implementations to choose from. The most common options include using `AlertDialog` for simple alerts, `DatePickerDialog` and `TimePickerDialog` for selecting dates and times, and `DialogFragment` for more complex dialogs. Choosing the right dialog implementation depends on the specific needs of your application and the complexity of the user interaction required.

Exploring the Different Types of Dialogs

Android provides a variety of built-in dialogs to cater to different needs. `AlertDialog` is widely used for showing alerts with a title, message, and action buttons. `DatePickerDialog` and `TimePickerDialog` are specialized dialogs for picking dates and times, respectively. Additionally, custom dialogs can be created using `DialogFragment` for scenarios where the built-in dialogs do not meet the application's requirements.

Working with Dialogs and Dialog Fragments

Dialogs in Android can be created and managed through the `DialogFragment` class, which allows for better control over the dialog's lifecycle and state management. A `DialogFragment` can be displayed by using the `show` method, which attaches the dialog to the fragment manager. This approach provides better integration with the activity lifecycle and allows for more flexible and reusable dialog implementations.

Tracing the Lifecycle of a Dialog and DialogFragment

The lifecycle of a `DialogFragment` is similar to that of a regular fragment, with methods such as `onCreate`, `onCreateDialog`, `onCreateView`, `onStart`, `onResume`, `onPause`, `onStop`, and `onDestroyView`. Understanding these lifecycle methods is crucial for managing the dialog's state and ensuring it behaves correctly in response to configuration changes and user interactions.

Working with Custom Dialogs

Creating custom dialogs involves extending `DialogFragment` and overriding the `onCreateDialog` method to provide a custom layout and functionality. Custom dialogs offer greater flexibility and control over the appearance and behavior of the dialog. They can include complex layouts, custom animations, and advanced interactions that are not possible with the standard dialog types.

Example:

```java
// CustomDialogFragment.java
public class CustomDialogFragment extends DialogFragment {
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
        LayoutInflater inflater = requireActivity().getLayoutInflater();
        View view = inflater.inflate(R.layout.custom_dialog, null);
```

```
        builder.setView(view)
                .setPositiveButton("OK", new DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int id) {
                        // Handle positive button action
                    }
                })
                .setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int id) {
                        // Handle negative button action
                    }
                });
        return builder.create();
    }
}
```

This example demonstrates creating a custom dialog by inflating a custom layout and using `AlertDialog.Builder` to set up the dialog. This approach allows for extensive customization and complex UI designs within dialogs.