Making Apps More Accessible

In this document

See also

- Accessibility Developer Checklist
- Accessibility Testing Checklist
- Material Design: Accessibility
- Designing Effective Navigation
- Training: Implementing Accessibility

Android apps should be usable by everyone, including people with disabilities.

Common disabilities that affect a person's use of an Android device include blindness or low vision, color blindness, deafness or impaired hearing, and restricted motor skills. When you develop apps with accessibility in mind, you make the user experience better not only for users with these disabilities, but also for all of your other users.

This document presents guidelines for improving your app's accessibility. It also lists resources that provide additional details and information related to accessibility features in Android.

Labeling UI Elements

It's important to provide useful and descriptive labels that explain the meaning and purpose of each interactive element to users. These labels allow screen readers, such as TalkBack, to properly explain the function of a particular control to users who rely on these services.

You can provide labels for elements in the following two ways:

- When labeling *static* elements, which don't change appearance throughout an activity's lifecycle, add an attribute to the corresponding XML element within the activity's layout resource file.
- When labeling *dynamic* elements, which change appearance during an activity's lifetime, set the element's label in the dynamic logic that changes the element's appearance.

The actual attributes and methods that you use to apply the element's label depend on the type of element:

- When labeling graphical elements, such as `ImageView` and `ImageButton` objects, use the `android:contentDescription` XML attribute for static elements and the `setContentDescription()` method for dynamic elements.

For graphical elements that are purely decorative, set their respective `android:contentDescription` XML attributes to `"@null"`. If your app only supports devices running Android 4.1 (API level 16) or higher, you can instead set these elements' `android:isImportantForAccessibility` XML attributes to `"no"`.

- When labeling editable elements, such as `EditText` objects, use the `android:hint` XML attribute for static elements and the `setHint()` method for dynamic elements to indicate each element's purpose.
- If your app is installed on a device running Android 4.2 (API level 17) or higher, use the `android:labelFor` attribute when labeling `View` objects that serve as content labels for other `View` objects.

> **Note:** Accessibility services automatically capture the text that appears in `TextView` objects, so you usually don't need to label these elements.

In the following example, a static `ImageButton` object designed to provide sharing functionality is given a label of `"share"` (defined in `values/strings.xml`) as the value:

```
<ImageButton
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_alignParentBottom="true"
   android:layout_alignParentLeft="true"
   android:layout_alignParentStart="true"
   android:contentDescription="@string/share"
   android:src="@drawable/ic_share" />
```

> **Note:** Many accessibility services, such as TalkBack and BrailleBack, automatically announce an element's type after announcing its label, so you shouldn't include element types in your labels. For example, `"submit"` is a good label for a `Button` object, but `"submitButton"` isn't a good label.

This next example shows how to update a dynamic `ImageView` object that displays a play or pause icon within an activity:

```
ImageView playPauseImageView = new ImageView();
boolean mediaCurrentlyPlaying = true;
...
private void updateImageButton() {
    if (mediaCurrentlyPlaying) {
       playPauseImageView.setImageResource(R.drawable.ic_pause);

       // In res/values/strings.xml, "pause" contains a value of "Pause".
       playPauseImageView.setContentDescription(getString(R.string.pause));
    } else {
       playPauseImageView.setImageResource(R.drawable.ic_play);

       // In res/values/strings.xml, "play" contains a value of "Play".
       playPauseImageView.setContentDescription(getString(R.string.play));
    }
}
```

When adding labels to the elements that appear in a given activity, make sure that each label is unique so that users can identify each element accurately. In particular, you should include additional text or contextual information in elements within reused layouts, such as `ListView` and `RecyclerView` objects, so that each child element is uniquely identified.

Grouping Content

You should arrange related content into groups so that accessibility services announce the content in a way that reflects its natural groupings. Users of assistive technology then don't need to swipe, scan, or wait as much to discover all information on the screen.

The two most effective methods of grouping related content are the following:

- For smaller or simpler groups of content, you can organize all content into a single announcement.
- For larger or more complex content structures, you can create natural groupings for the content.

Organizing content into a single announcement

If users should treat a set of elements as a single unit of information, you can group these elements in a focusable container. That way, accessibility services present the grouped content in a single announcement. In the following example, a `RelativeLayout` element contains pieces of content that relate to one another:

```
<RelativeLayout
    android:id="@+id/song_data_container"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:focusable="true">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/song_title"
        android:text="@string/song_title" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/singer"
        android:layout_toRightOf="@id/song_title"
        android:text="@string/singer" />
    ...
</RelativeLayout>
```

Note: You can specify the `android:contentDescription` XML attribute on a container to override automatic grouping and ordering of contained items.

When grouping element labels, make sure that you don't create unnecessarily verbose announcements for accessibility services to present.

Creating natural groupings

If you need to convey more complex structures, such as tables, you can assign focus to one piece of the structure at a time, such as a single row. To define the proper focusing pattern for a set of related content, place each piece of the structure into its own focusable `ViewGroup`. In the following example, a structure comprising 6 `TextView` objects is divided into 3 pieces, which the `RelativeLayout` elements define:

```
<LinearLayout
    ...
    orientation="vertical">
    <RelativeLayout
        ...
        android:focusable="true">
        <TextView ... />
        <TextView ... />
    </RelativeLayout>
    <RelativeLayout
        ...
        android:focusable="true">
        <TextView ... />
        <TextView ... />
    </RelativeLayout>
    <RelativeLayout
        ...
        android:focusable="true">
        <TextView ... />
        <TextView ... />
    </RelativeLayout>
</LinearLayout>
```

Making Touch Targets Large

Many people have difficulty interacting with small touch targets on a device's screen. This could be because their fingers are large or because they have a motor or visual impairment. By providing larger touch targets, you make it substantially easier for users to navigate your app.

In general, you want the touchable area of focusable items to be a minimum of 48dp x 48dp. Larger is even better.

To ensure that each focusable item in your app has a large enough touch target, set the `android:minWidth` and `android:minHeight` attributes of each interactive element to 48dp or greater:

```
<ImageButton
   ...
   android:minWidth="48dp"
   android:minHeight="48dp" />
```

You can also add padding or use the `TouchDelegate` API to increase the size of your elements' touch targets without increasing the size of the elements themselves.

Providing Adequate Color Contrast

People with low vision and those who use devices with dimmed displays can have difficulty reading information on the screen. By providing increased contrast ratios between the foreground and background colors in your app, you make it easier for users to navigate within and between screens.

To help developers use sufficient contrast ratios in their apps, The World Wide Web Consortium (W3C) has created a set of color contrast accessibility guidelines:

- For large text, 18 points or higher for regular text and 14 points or higher for bold text, you should use a contrast ratio of at least **3.0 to 1**.
- For small text, smaller than 18 points for regular text and smaller than 14 points for bold text, you should use a contrast ratio of at least **4.5 to 1**.

Figure 1 shows two version of an activity. One version uses a low contrast ratio between background and foreground colors, and the other version uses an increased contrast ratio:

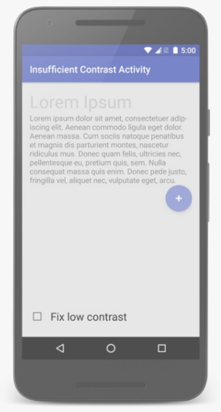
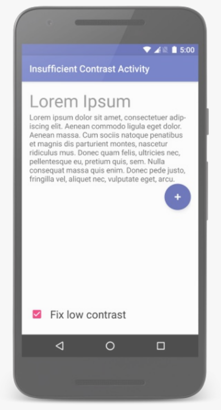
 5:00 Insufficient Contrast Activity Lorem Ipsum Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. ☐ Fix low contrast	 5:00 Insufficient Contrast Activity Lorem Ipsum Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. ☑ Fix low contrast
Low contrast ratios Title: 1.42 to 1 Content: 3.49 to 1	**Increased contrast ratios** Title: 4.61 to 1 Content: 10.05 to 1

Figure 1: Example of low and increased contrast ratios between foreground and background colors

To check your contrast ratios, you can use the Accessibility Scanner or one of the many contrast checkers available online.

> **Note:** If your app's color scheme uses partially transparent colors, keep in mind that these non-opaque colors might appear lighter than the color defined by their RGB values.

Using Cues Other Than Color

To assist users with color vision deficiencies, use cues other than color to distinguish UI elements within your app's screens. These techniques could include using different shapes or sizes, providing text or visual patterns, or adding audio- or touch-based (haptic) feedback to mark the elements' differences.

Figure 2 shows two versions of an activity. One version uses only color to distinguish between two possible actions in a workflow, and the other uses shapes and text to highlight the differences between the two options:

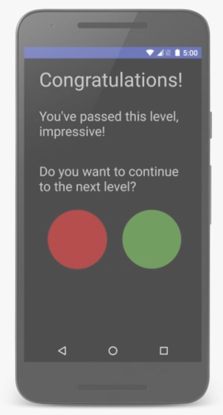
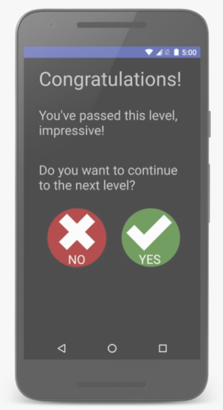
 Congratulations! You've passed this level, impressive! Do you want to continue to the next level?	 Congratulations! You've passed this level, impressive! Do you want to continue to the next level?
Insufficient: Using color only	**Acceptable: Using color, shapes, and text**

Figure 2: Examples of differentiating UI elements using color only and using color, shapes, and text

Presenting Media Content

If you're developing an app that includes media content, such as a video clip or an audio recording, make sure that users with different types of accessibility needs can understand this material, as well. In particular, you should make sure that you've provided the following accommodations:

- All video and audio materials should include controls that allow users to pause or stop the media, change the volume, and toggle subtitles (captions).
- If a video presents information that is vital to completing a workflow, you should provide the same content in an alternative format, such as a transcript.

Applying Ideas from Accessibility Resources

The following resources describe other accessibility features that Android supports. After you've applied the guidelines in the previous sections, use the following information to further improve your app's accessibility.

Applying Accessibility Design Principles

Google's material design guidelines include a set of recommendations for structuring your app's UI so that all users, including users with disabilities, can interact easily with your app. To view these guidelines, navigate to the Accessibility page within the material design site.

Activating Accessibility Settings

In addition to offering the accessibility services described in other sections on this page, the Android platform includes several accessibility settings, such as increased font size and magnified screen area, that users can adjust. As you develop your app, you should adjust these settings yourself to ensure that your app's important UI elements remain fully visible and usable.

Testing Your App's Accessibility

As you develop your app, it's important to test its accessibility using a combination of manual tests, automated tests, and user tests. To learn more about the aspects of your app that you should test, see Testing Your App's Accessibility.

Working with Custom Views

The UI components built into the Android framework have predefined accessibility capabilities. These components include useful metadata that accessibility services access to present these components successfully to users who use assistive technology.

When you create your own custom view, such as an animated bar graph widget, you need to manually define the accessibility metadata so that accessibility services can interpret the custom view properly. For more information about how to make custom views accessible, read Build Accessible Custom Views.

Sample Accessibility App

The Basic Accessibility sample, available on GitHub, presents an app containing a variety of UI elements. It demonstrates how to add accessibility markup to each of these elements and how accessibility services respond to this markup.