

# **Intermediate Android**

## **Advanced UI, Animation, Localization and Media**

# Ground Rules

Observe the following rules to ensure a supportive, inclusive, and engaging classes



Give full attention  
in class



Mute your microphone  
when you're not talking



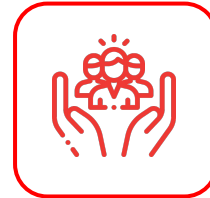
Keep your  
camera on



Turn on the CC Feature  
on Meet



Use raise hand or chat  
to ask questions



Make this room a safe place  
to learn and share

# Material/Review

# Material that has been Studied

- Learn how to **create custom view** for more specialized component.
- Learn how to **create widget** for displaying data.
- Learn how to **draw objects** on a Canvas.
- Learn how to **build animation** using Property Animation, Transition Activity, and Motion Layout.
- Learn how to **take advantage of localization** to support multiple languages, alternative layouts, and formats for information.
- Learn how to **use accessibility** to reach a variety of user backgrounds.
- Learn how to **play music and video** using Soundpool, Multiplayer, and ExoPlayer.
- Learn how to **get image** from CameraX, Intent Camera, dan Intent Gallery.
- Learn how to **upload files** to a server with Multipart in Retrofit.

# CustomView

# Create a CustomView

Creating a custom view can be done in two ways, namely modifying an existing view component or creating it from scratch.

A custom view should:

- Conform to Android standards
- Provide custom styleable attributes that work with Android XML layouts
- Send accessibility events
- Be compatible with multiple Android platforms.

An example of a Custom View is a Button which is a modification of a TextView.

Android Developers > Docs > Reference

## Button

---

```
public class Button  
    extends TextView
```

```
java.lang.Object  
↳ android.view.View  
    ↳ android.widget.TextView  
        ↳ android.widget.Button
```

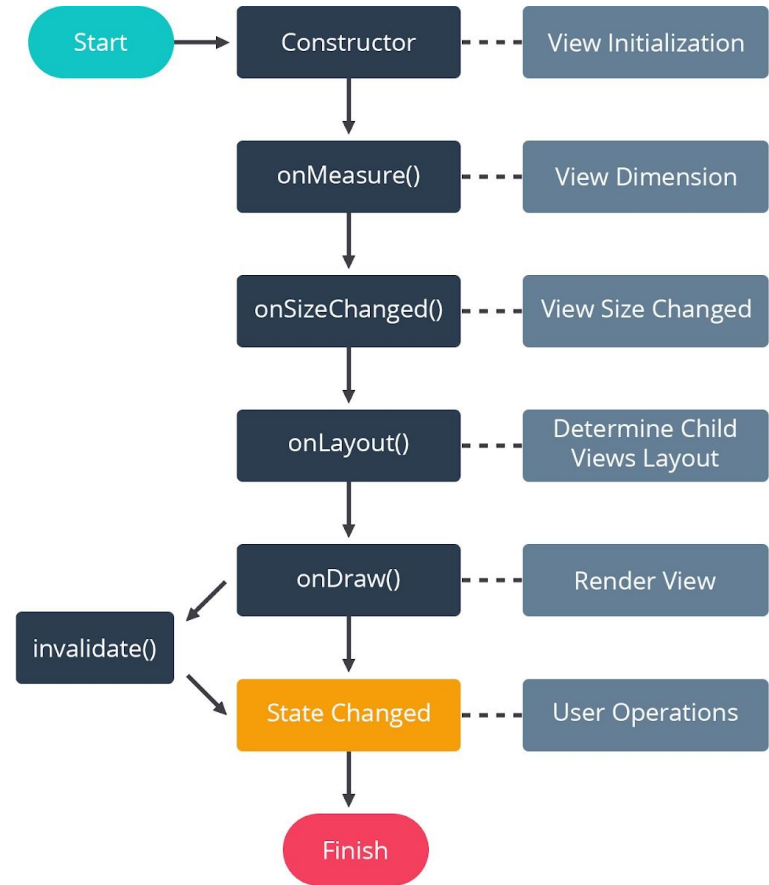
# Create a CustomView

Creating a custom view can be done in two ways, namely modifying an existing view component or creating it from scratch.

A custom view should:

- Conform to Android standards
- Provide custom styleable attributes that work with Android XML layouts
- Send accessibility events
- Be compatible with multiple Android platforms.

An example of a Custom View is a Button which is a modification of a TextView.





# Canvas

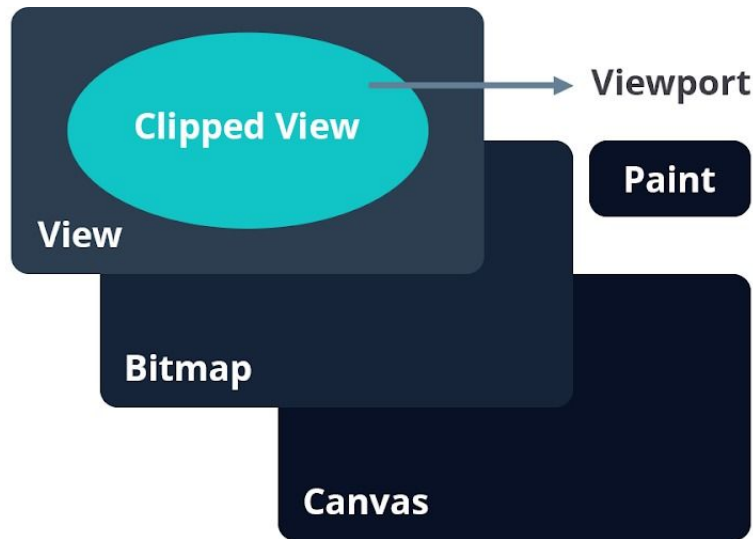


# Draw objects on a Canvas

The Canvas class holds the "draw" calls. To draw something, you need 4 basic components:

- A Bitmap to hold the pixels.
- A Canvas to host the draw calls (writing into the bitmap).
- A drawing primitive (e.g. Rect, Path, text, Bitmap).
- A Paint (to describe the colors and styles for the drawing).

The parameter to **onDraw()** is a Canvas object that the view can use to draw itself.



# Android Widget

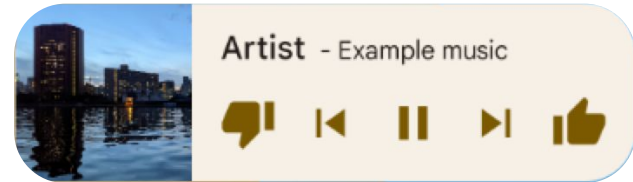
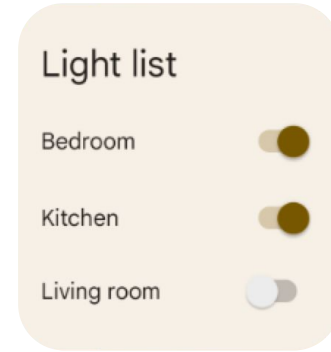
# Create a **App Widget**

Widgets are an essential aspect of home screen customization. App Widgets can be categorized into 4:

- Information widgets
- Collection widgets
- Control widgets
- Hybrid widgets

Some things to consider to maximize the use of widgets:

- Widget content
- Widget navigation
- Widget resizing
- Layout considerations
- Widget configuration by users



# Android 12 Widgets Improvements

Android 12 (API level 31) revamps the existing Widgets API to improve the user and developer experience in the platform and launchers.

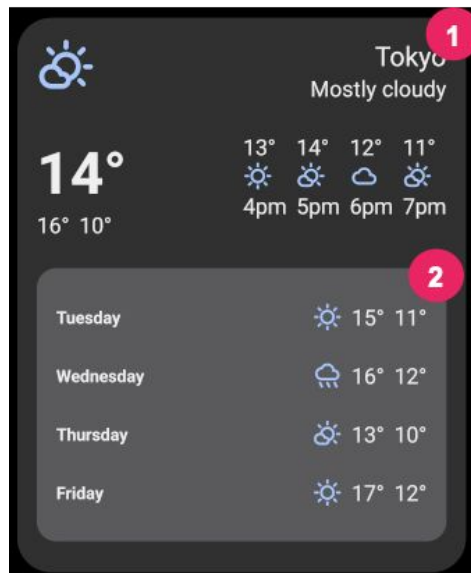
- Ensure your widget is compatible with Android 12
- Implement rounded corners
- Add device theming
- Add new compound buttons
- Use improved APIs for widget sizes and layouts
- Improve your app's widget picker experience
- Enable smoother transitions



# Android 12 Widgets Improvements

Android 12 (API level 31) revamps the existing Widgets API to improve the user and developer experience in the platform and launchers.

- Ensure your widget is compatible with Android 12
- Implement rounded corners
- Add device theming
- Add new compound buttons
- Use improved APIs for widget sizes and layouts
- Improve your app's widget picker experience
- Enable smoother transitions



1. Corner of the widget.
2. Corner of a view inside the widget.

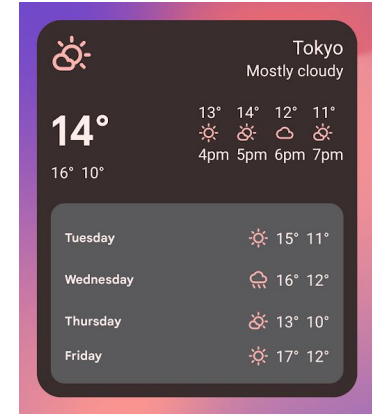
# Android 12 Widgets Improvements

Android 12 (API level 31) revamps the existing Widgets API to improve the user and developer experience in the platform and launchers.

- Ensure your widget is compatible with Android 12
- Implement rounded corners
- Add device theming
- Add new compound buttons
- Use improved APIs for widget sizes and layouts
- Improve your app's widget picker experience
- Enable smoother transitions



Widget in light theme

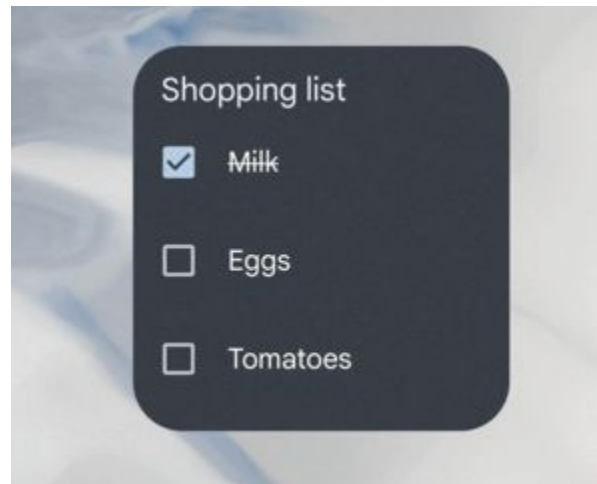


Widget in dark theme

# Android 12 Widgets Improvements

Android 12 (API level 31) revamps the existing Widgets API to improve the user and developer experience in the platform and launchers.

- Ensure your widget is compatible with Android 12
- Implement rounded corners
- Add device theming
- Add new compound buttons
- Use improved APIs for widget sizes and layouts
- Improve your app's widget picker experience
- Enable smoother transitions



# Property Animation



# Property Animation

The **property animation** system is a robust framework that allows you to animate almost anything.

The property animation system lets you define the following characteristics of an animation:

- Duration
- Time interpolation
- Repeat count and behavior
- Animator sets
- Frame refresh delay

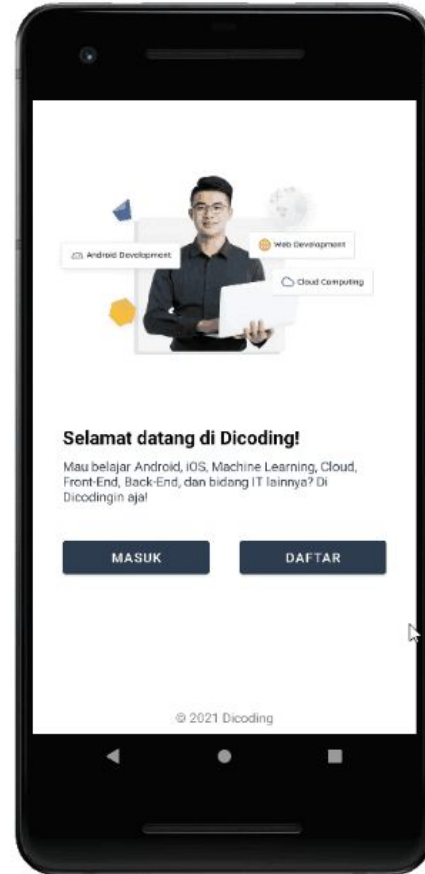
```
ObjectAnimator.ofFloat(  
    textView, // View Component  
    "translationX", // Property Name  
    100f // Value Property  
).apply {  
    duration = 1000 // Duration  
    start() // Play Animation  
}
```

# Property Animation

The **property animation** system is a robust framework that allows you to animate almost anything.

The property animation system lets you define the following characteristics of an animation:

- Duration
- Time interpolation
- Repeat count and behavior
- Animator sets
- Frame refresh delay



# Transition Activity

# Activity Transition

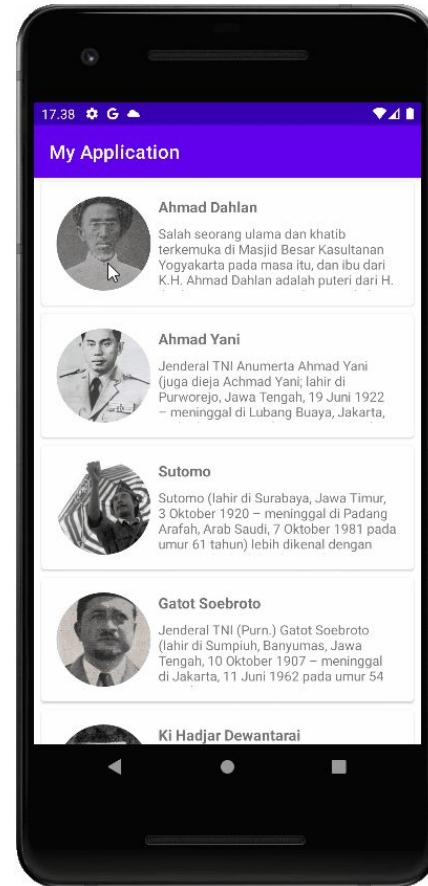
**Activity transitions** in material design apps provide visual connections between different states through motion and transformations between common elements.

- An enter transition.
- An exit transition.
- A shared elements transition.

Activity transition APIs are available on Android 5.0 (API 21) and up.

```
with(window) {  
    requestFeature(Window  
        .FEATURE_ACTIVITY_TRANSITIONS)  
    exitTransition = Slide()  
}
```

Destination Activity



# Shared Element Transition

A **shared elements transition** determines how views that are shared between two activities transition between these activities.

To make a screen transition animation between two activities that have a shared element:

- Enable window content transitions in your theme.
- Specify a shared elements transition in your style.
- Define your transition as an XML resource.
- Assign a common name to the shared elements in both layouts with the **android:transitionName** attribute.
- Use the **ActivityOptions.makeSceneTransitionAnimation()** function.

```
<ImageView
    android:id="@+id/profileImageView"
    android:transitionName="profile"
    ... />
```

activity\_destination.xml

```
val optionsCompat =
    ActivityOptionsCompat
        .makeSceneTransitionAnimation(
            context, // Context
            imgPhoto, // ImageView
            "profile" // Transition name
        )
```

MainActivity.kt

```
startActivity(
    intent, // Intent to Another Activity
    optionsCompat.toBundle()
)
```

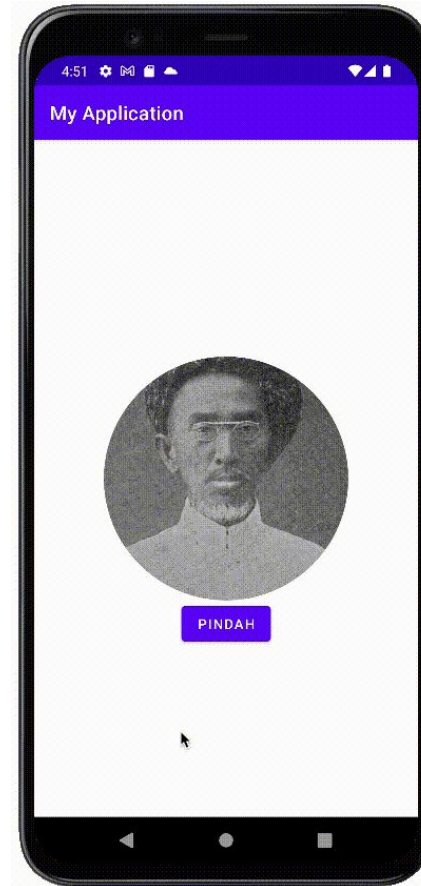
Intent Explicit

# Shared Element Transition

A **shared elements transition** determines how views that are shared between two activities transition between these activities.

To make a screen transition animation between two activities that have a shared element:

- Enable window content transitions in your theme.
- Specify a shared elements transition in your style.
- Define your transition as an XML resource.
- Assign a common name to the shared elements in both layouts with the **android:transitionName** attribute.
- Use the **ActivityOptions.makeSceneTransitionAnimation()** function.



# Motion Layout

# Motion Layout

**MotionLayout** is a layout type that helps you manage motion and widget animation in your app.

MotionLayout is a subclass of **ConstraintLayout** and builds upon its rich layout capabilities.

As part of the ConstraintLayout library, MotionLayout is available as a support library and is backwards-compatible to **API level 14**.

Following are the components required to use MotionLayout:

- Dependency
- MotionLayout File
- MotionScene File

```
dependencies {  
    implementation  
    'androidx.constraintlayout:constraintlayout:$latestVersion'  
}
```

Dependency

```
<!-- before -->  
<androidx.constraintlayout.widget.ConstraintLayout .../>  
  
<!-- after -->  
<androidx.constraintlayout.motion.widget.MotionLayout .../>
```

activity\_main.xml

```
<MotionScene ...>  
    <Transition .../>  
    <ConstraintSet android:id="@+id/start"/>  
    <ConstraintSet android:id="@+id/end"/>  
</MotionScene>
```

scene\_motion.xml



# Motion Layout

**MotionLayout** is a layout type that helps you manage motion and widget animation in your app.

MotionLayout is a subclass of **ConstraintLayout** and builds upon its rich layout capabilities.

As part of the ConstraintLayout library, MotionLayout is available as a support library and is backwards-compatible to **API level 14**.

Following are the components required to use MotionLayout:

- Dependency
- MotionLayout File
- MotionScene File



# Localization and Accessibility

# Localization

**Localization** is an effort to translate content into local language and local context.

The goal is that the application can adapt to the cultural needs and development of users in many countries that are the target market.

The following are some of the things that can be done with localization, including:

- Multiple language support.
- Multiple layout support.
- Information formatting support.

```
<resources>
    <string name="hello_world">Hola
Mundo!</string>
</resources>
```

values-es/strings.xml (Spanish)

```
<resources>
    <string name="hello_world">Bonjour le
monde!</string>
</resources>
```

values-fr/strings.xml (French)

```
res/layout/main_activity.xml
# For handsets (smaller than 600dp
available width)

res/layout-land/main_activity.xml
# For handsets in landscape

res/layout-sw600dp/main_activity.xml
# For 7" tablets (600dp wide and bigger)
```

res/layout/...

```
val df = DateFormat
    .getDateInstance(DateFormat.FULL,
Locale.CANADA)
```

DateFormat

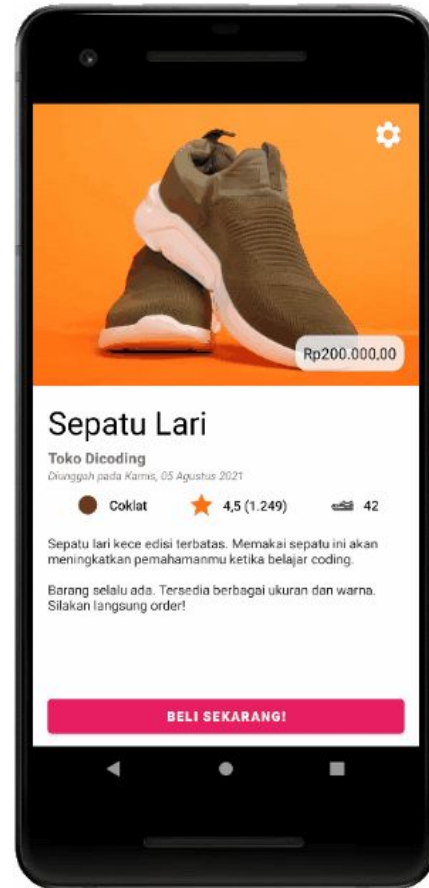
# Localization

**Localization** is an effort to translate content into local language and local context.

The goal is that the application can adapt to the cultural needs and development of users in many countries that are the target market.

The following are some of the things that can be done with localization, including:

- Multiple language support.
- Multiple layout support.
- Information formatting support.



# Multiple Language Support

The following is an overview of the steps for preparing an app for a different language:

- **Extract all strings** as resources into strings.xml
- Use **Android Studio's Translations Editor** to add a language, and add translations for the strings
- **Don't concatenate** pieces of text to make a sentence or phrase.
- If you must combine pieces of text, use format strings as placeholders, and use **String.format()** to supply the arguments to create the completed string.
- To **test your app**, change the device's language in the Settings app before each test run.

```
<resources>  
    <string name="hello_world">Hola  
Mundo!</string>  
</resources>
```

values-es/strings.xml (Spanish)

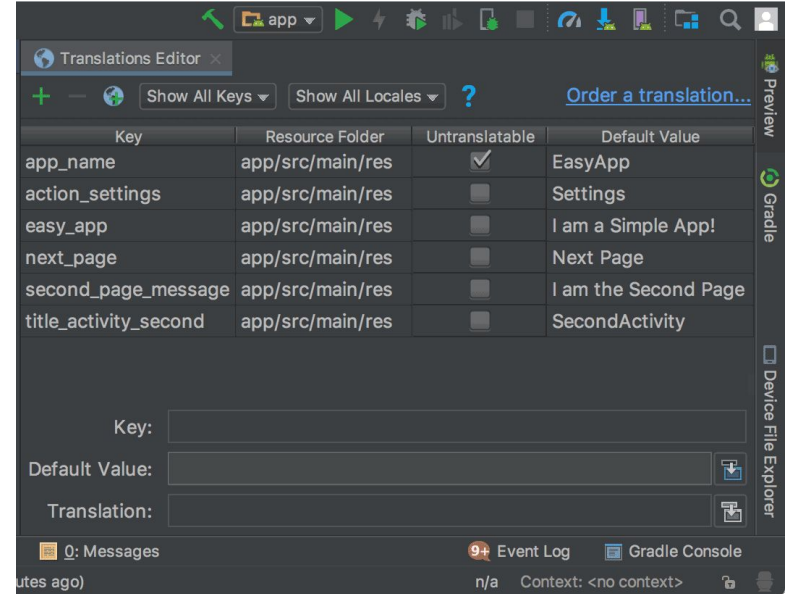
```
<resources>  
    <string name="hello_world">Bonjour le  
monde!</string>  
</resources>
```

values-fr/strings.xml (French)

# Multiple Language Support

The following is an overview of the steps for preparing an app for a different language:

- **Extract all strings** as resources into strings.xml
- Use **Android Studio's Translations Editor** to add a language, and add translations for the strings
- **Don't concatenate** pieces of text to make a sentence or phrase.
- If you must combine pieces of text, use format strings as placeholders, and use **String.format()** to supply the arguments to create the completed string.
- To **test your app**, change the device's language in the Settings app before each test run.



# Multiple Language Support

The following is an overview of the steps for preparing an app for a different language:

- **Extract all strings** as resources into strings.xml
- Use **Android Studio's Translations Editor** to add a language, and add translations for the strings
- **Don't concatenate** pieces of text to make a sentence or phrase.
- If you must combine pieces of text, use format strings as placeholders, and use **String.format()** to supply the arguments to create the completed string.
- To **test your app**, change the device's language in the Settings app before each test run.

strings.xml

```
<string name="app_name">Notes
App</string>
<string name="author">Dicoding</string>
```

MainActivity.xml

```
val description =
    "${getString(R.string.app_name)} created
    by ${getString(R.string.author)}"

</resources>
```



strings.xml

```
<resources>
<string name="description">%1$s created
by %2$s</string>
</resources>
```

MainActivity.xml

```
val description = String.format(
    getString(R.string.description),
    getString(R.string.app_name),
    getString(R.string.author))
```

# Multiple Language Support

The following is an overview of the steps for preparing an app for a different language:

- **Extract all strings** as resources into strings.xml
- Use **Android Studio's Translations Editor** to add a language, and add translations for the strings
- **Don't concatenate** pieces of text to make a sentence or phrase.
- If you must combine pieces of text, use format strings as placeholders, and use **String.format()** to supply the arguments to create the completed string.
- To **test your app**, change the device's language in the Settings app before each test run.

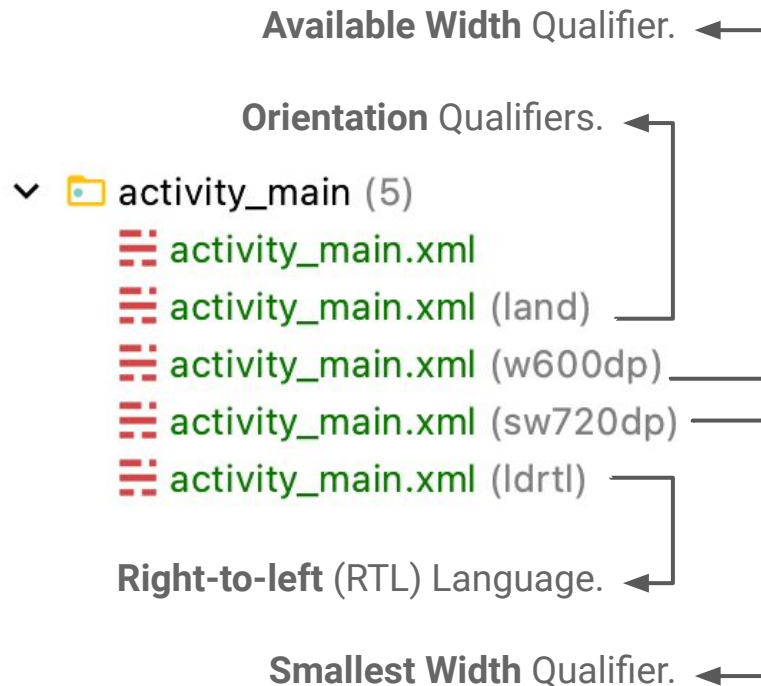
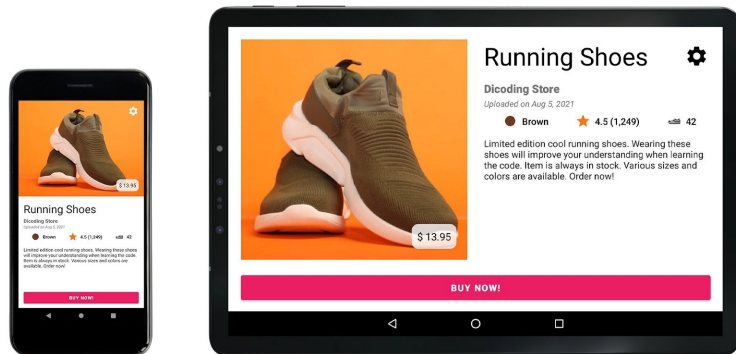




# Multiple Display Support

The following is an overview of the steps for preparing an app for a different layout:

- Adding a **right-to-left (RTL) language**.
- Use the **smallest width qualifier**.
- Use the **available width qualifier**.
- Add **orientation qualifiers**.



# Information Formatting Support

The following is an overview of the steps for preparing an app for a Information formatting support:

- Android provides the **DateFormat** class and methods to apply the **date format** of the user-chosen locale.
- **NumberFormat** is the abstract base class for all **number formats**.
- Use the **NumberFormat** class to format **currency numbers**.
- The **Locale** class provides a number of convenient constants that you can use to create Locale objects for commonly used locales.

DateFormat

```
val formattedDate = DateFormat
    .getDateInstance(DateFormat.FULL)
val result = formattedDate
    .format(value).toString()
```

Output

Wednesday, March 16, 2022

# Information Formatting Support

The following is an overview of the steps for preparing an app for a Information formatting support:

- Android provides the **DateFormat** class and methods to apply the **date format** of the user-chosen locale.
- **NumberFormat** is the abstract base class for all **number formats**.
- Use the **NumberFormat** class to format **currency numbers**.
- The **Locale** class provides a number of convenient constants that you can use to create Locale objects for commonly used locales.

NumberFormat

```
val numberFormat = NumberFormat
    .getNumberInstance()
val result = numberFormat
    .format(1000000)
```

Output

1,000,000

# Information Formatting Support

The following is an overview of the steps for preparing an app for a Information formatting support:

- Android provides the **DateFormat** class and methods to apply the **date format** of the user-chosen locale.
- **NumberFormat** is the abstract base class for all **number formats**.
- Use the **NumberFormat** class to format **currency numbers**.
- The **Locale** class provides a number of convenient constants that you can use to create Locale objects for commonly used locales.

CurrencyFormat

```
val currencyFormat = NumberFormat
    .getCurrencyInstance()
val result = currencyFormat
    .format(number)
```

Output

\$1,000,000.00

# Information Formatting Support

The following is an overview of the steps for preparing an app for a Information formatting support:

- Android provides the **DateFormat** class and methods to apply the **date format** of the user-chosen locale.
- **NumberFormat** is the abstract base class for all **number formats**.
- Use the **NumberFormat** class to format **currency numbers**.
- The **Locale** class provides a number of convenient constants that you can use to create Locale objects for commonly used locales.

Locale

```
val formattedDate = DateFormat
    .getDateInstance(DateFormat.FULL,
        Locale("ID"))
val result = formattedDate
    .format(value).toString()
```

Output

Rabu, 16 Maret 2022

# Accessibility

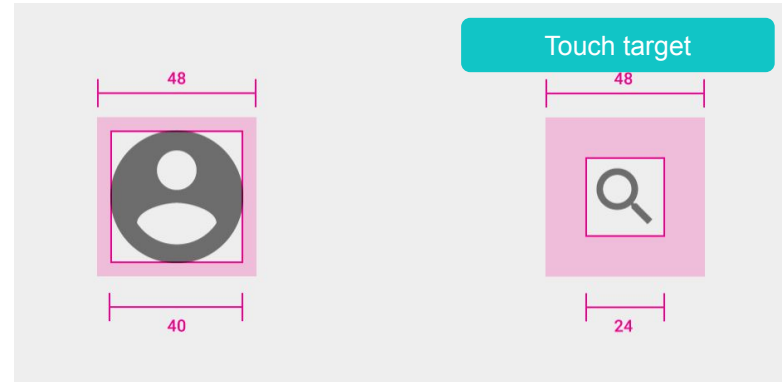
**Accessibility** is a set of design, implementation, and testing techniques that enable your app to be usable by everyone, including people with disabilities.

The following are some things to consider regarding accessibility.

- Layout design
- Touch target
- Selection and focus

To support Accessibility, you must implement **content labels** and **description**.

Use **TalkBack** to enable accessibility in the app.



Touch target

48

24

Content Description

```
<Button
    android:id="@+id/pause_button"
    android:src="@drawable/pause"
    android:contentDescription="@string/pause" .../>
```

Hint

```
<EditText
    android:id="@+id/edittext_message"
    android:hint="Enter your message" .../>
```

Label

```
<TextView
    android:id="@+id/textview_label"
    android:labelFor="@+id/edittext_message" .../>
```

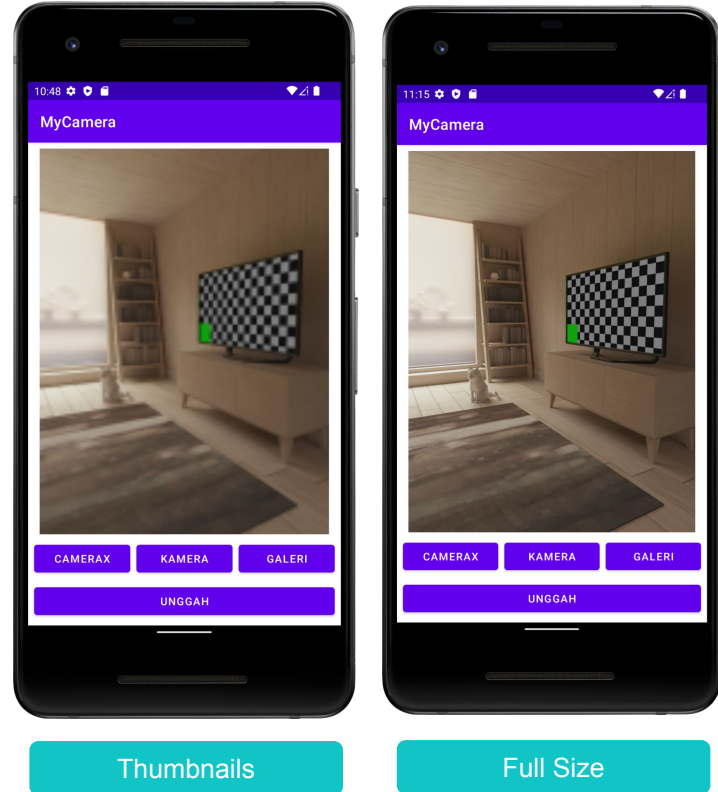
# Intent Camera

# Intent Camera

Android provides full access to the device camera hardware so you can build a wide range of camera or vision-based apps.

There are two ways to use the Intent Camera:

- **Thumbnails:** Get images with a small resolution (*thumbnails*) and usually only for logos or icons in applications.
- **Full Size:** Get an image with the full size or resolution that matches the camera's results.





# Intent Gallery

# Intent Gallery

Similar to Intent Camera, Intent Gallery also works to get images into your app. The difference is, the pictures taken are from the gallery.

To request that the user select a file such as a document or photo and return a reference to your app, use the **ACTION\_GET\_CONTENT** action and specify your desired MIME type.

The result intent delivered to your **registerForActivityResult** method includes data with a URI pointing to the file. The URI could be anything, such as an **http: URI**, **file: URI**, or **content: URI**.

```
private val launcherIntentGallery =
    registerForActivityResult(
        ActivityResultContracts.StartActivityForResult()
    ) { result ->
        if (result.resultCode == RESULT_OK) {
            val selectedImg: Uri = result.data?.data as Uri
            binding.previewImageView.setImageURI(selectedImg)
        }
    }
```

Activity Result

```
binding.galleryButton.setOnClickListener {
    val intent = Intent()
    intent.action = ACTION_GET_CONTENT
    intent.type = "image/*"
    val chooser = Intent.createChooser(
        intent,
        "Choose a Picture"
    )
    launcherIntentGallery.launch(chooser)
}
```

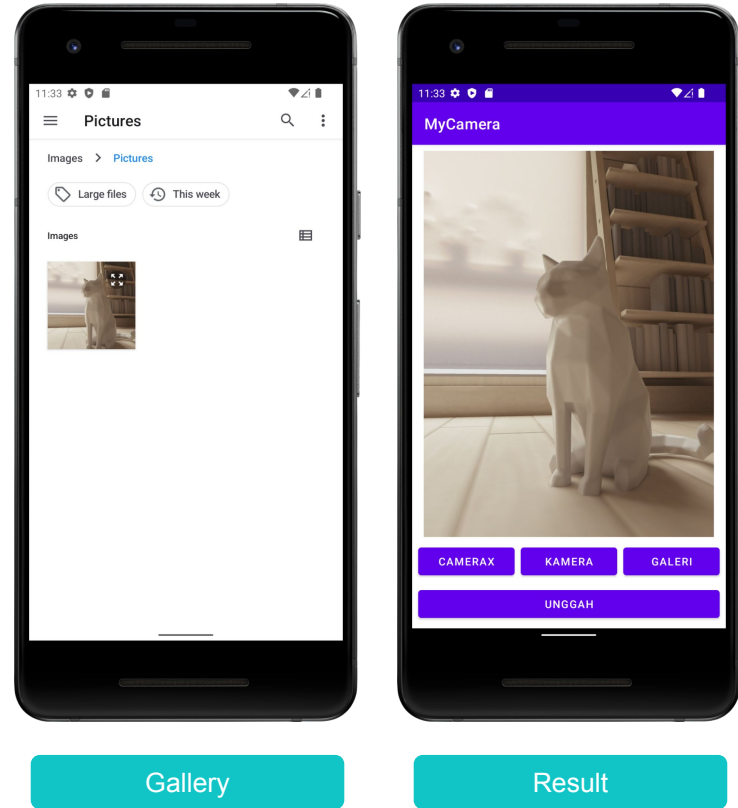
Intent

# Intent Gallery

Similar to Intent Camera, Intent Gallery also works to get images into your app. The difference is, the pictures taken are from the gallery.

To request that the user select a file such as a document or photo and return a reference to your app, use the **ACTION\_GET\_CONTENT** action and specify your desired MIME type.

The result intent delivered to your **registerForActivityResult** method includes data with a URI pointing to the file. The URI could be anything, such as an **http: URI**, **file: URI**, or **content: URI**.



Gallery

Result

# CameraX

# CameraX

**CameraX** is a Jetpack support library, built to help you make camera app development easier. It provides a **consistent** and **easy-to-use** API surface that works across most Android devices, with **backward-compatibility** to Android 5.0 (API level 21).

CameraX improves the developer experience in the following ways:

- Broad device coverage
- Ease of use
- Consistency across devices
- New camera experiences



An image captured with the Bokeh (Portrait) effect using CameraX.

# Upload a File

## Retrofit

Retrofit is a library made by Square, which is popularly used for Networking the Web API.

With Retrofit, setting up API endpoints and parsing JSON is much easier.



```
implementation "com.squareup.retrofit2:retrofit:$retrofitVersion"  
implementation "com.squareup.retrofit2:converter-gson:$retrofitVersion"
```

# Upload Image using Retrofit

- **URL**
  - /stories/guest
- **Method**
  - POST
- **Headers**
  - Content-Type: multipart/form-data
- **Body**
  - description : String
  - photo : File gambar.
  - lat : Long (optional)
  - lon : Long (optional)

```
interface ApiService {  
    @Multipart  
    @POST("/v1/stories/guest")  
    fun uploadImage(  
        @Part file: MultipartBody.Part,  
        @Part("description") description:  
        RequestBody,  
    ): Call<FileUploadResponse>  
}
```



# Upload Image using Retrofit

- **URL**
  - /stories/guest
- **Method**
  - POST
- **Headers**
  - Content-Type: multipart/form-data
- **Body**
  - description : String
  - photo : File gambar.
  - lat : Long (optional)
  - lon : Long (optional)

```
val imageDescription: String = "Ini adalah deskripsi"
val fileImage: File = //file image.

val description = imageDescription
    .toRequestBody("text/plain".toMediaType())

val requestImageFile = fileImage
    .asRequestBody("image/jpeg"
        .toMediaTypeOrNull())

val imageMultipart: MultipartBody.Part =
    MultipartBody.Part.createFormData("photo",
        file.name, requestImageFile)

val service = ApiConfig().getApiService()
    .uploadImage(imageMultipart, description)
```

# Multipart

**multipart/form-data** is one of the value of enctype attribute, which is used in form element that have a **file upload**.

Multi-part means form data divides into multiple parts and send to server.

Multipart requests combine one or more sets of data into a **single body**, separated by **boundaries**.

Use Postman or similar apps to test endpoints before implementing them in Android apps.

One of the popular libraries commonly used to upload images is **Retrofit**.

```
Content-Type: multipart/form-data;  
boundary="babfeb1d"
```

```
--babfeb1d
```

```
Content-Disposition: form-data; name="photo"  
value1
```

```
--babfeb1d
```

```
Content-Disposition: form-data;  
name="description"; filename="example.txt"  
value2
```

```
--babfeb1d--
```

Response Multipart

**Demo Link**

[Upload Image using Retrofit](#)  
[Story API Documentation](#)

# Sharing

**Demo Link**

[ILT 4](#)

# Discussion

# Quiz

**Thank You**

