# Table of Contents

# Advanced Android Development — Concepts

This is the *concepts reference* for Advanced Android Development, a training course created by the Google Developers Training team. This course builds on the skills you learned in the Android Developer Fundamentals course.

This course is intended to be taught in a classroom, but all the materials are available online, so if you like to learn by yourself, go ahead!

## Prerequisites

The Advanced Android Development course is intended for experienced developers who have Java programming experience and know the fundamentals of how to build an Android app using the Java language. This course assumes you have mastered the topics in Units 1 to 4 of the Android Developer Fundamentals course.

Specifically, this course assumes you know how to:

- Install and use Android Studio
- Run apps from Android Studio on both a device and an emulator
- Create and use `Activity` instances
- Use `View` instances to create your app's user interface
- Enable interaction through click handlers
- Create layouts using the Android Studio layout editor
- Create and use `RecyclerView` and `Adapter` classes
- Run tasks in the background
- Save data in Android shared preferences
- Save data in a local SQL database

## Course materials

The course materials include:

- This concept reference
- A practical workbook, Advanced Android Development – Practicals, which guides you through creating Android apps to practice and perfect the skills you're learning
- Slide decks for optional use by instructors
- Source code in GitHub for apps that you create during the practical exercises

## What topics are covered?

## Unit 1: Expand the user experience

Lesson 1: Fragments

Lesson 2: App widgets

Lesson 3: Sensors

## Unit 2: Make your apps fast and small

Lesson 4: Performance

## Unit 3: Make your apps accessible

Lesson 5: Localization

Lesson 6: Accessibility

## Unit 4: Add geo features to your apps

Lesson 7: Location

Lesson 8: Places

Lesson 9: Mapping

## Unit 5: Advanced graphics and views

Lesson 10: Custom views

Lesson 11: Canvas

Lesson 12: Animations

*Developed by the Google Developers Training Team*


*Last updated: October 2017*

# 1.1: Fragments

**Contents:**



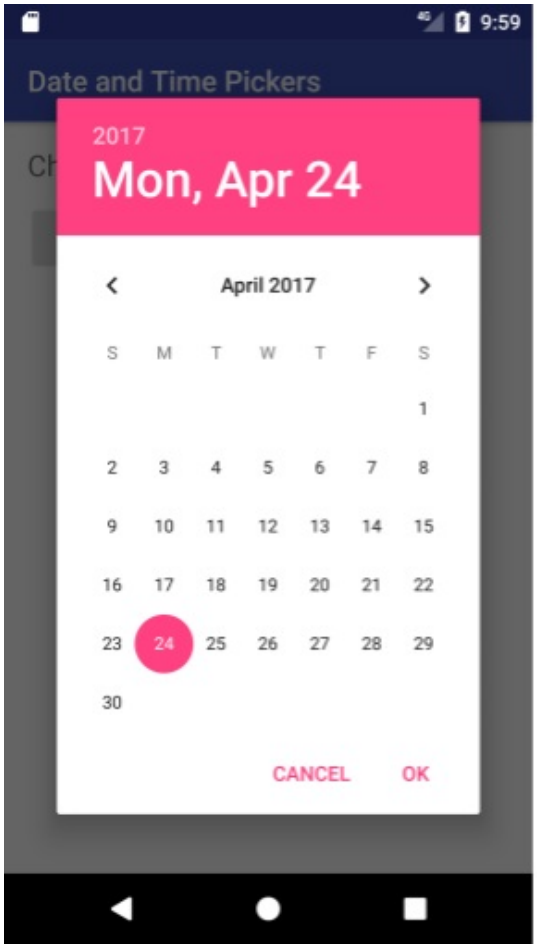
A `Fragment` is a self-contained component with its own user interface (UI) and lifecycle that can be reused in different parts of an app's UI. This chapter explains how a `Fragment` can be useful for a UI design. (A `Fragment` can also be used without a UI, in order to retain values across configuration changes, but this chapter does not cover that usage.)

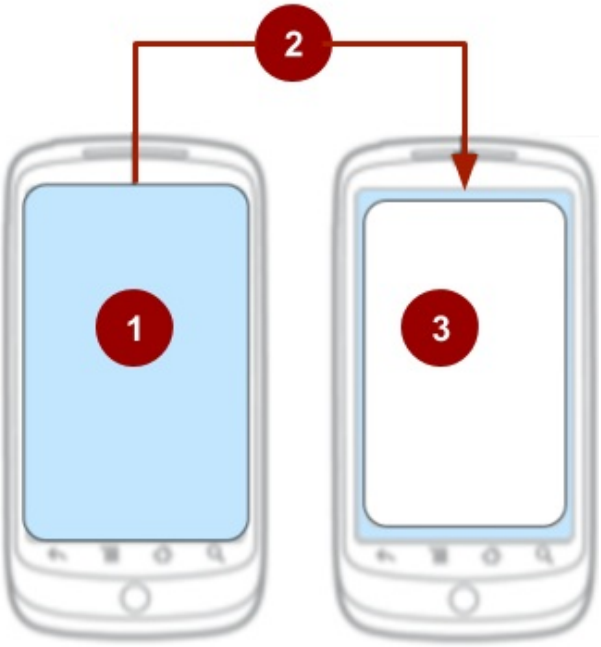
## Understanding fragments

A `Fragment` is a class that contains a portion of an app's UI and behavior, which can be added as part of an `Activity` UI. While a single `Fragment` can be shared by different activities, each specific instance of the `Fragment` is exclusively tied to the `Activity` that hosts it.

A `Fragment` is like a miniature `Activity`. Although it must be hosted by an `Activity`, a `Fragment` has its own lifecycle. Also like an `Activity`, a `Fragment` receives its own input events. For example, the standard date picker is a `Fragment`—an instance of `DialogFragment`, a subclass of `Fragment`—that enables the user to input a date. The standard date picker shows a dialog window floating on top of the `Activity` window.

For maximum reusability, a single `Fragment` should contain the code to define its layout *and* its behavior for user interaction.



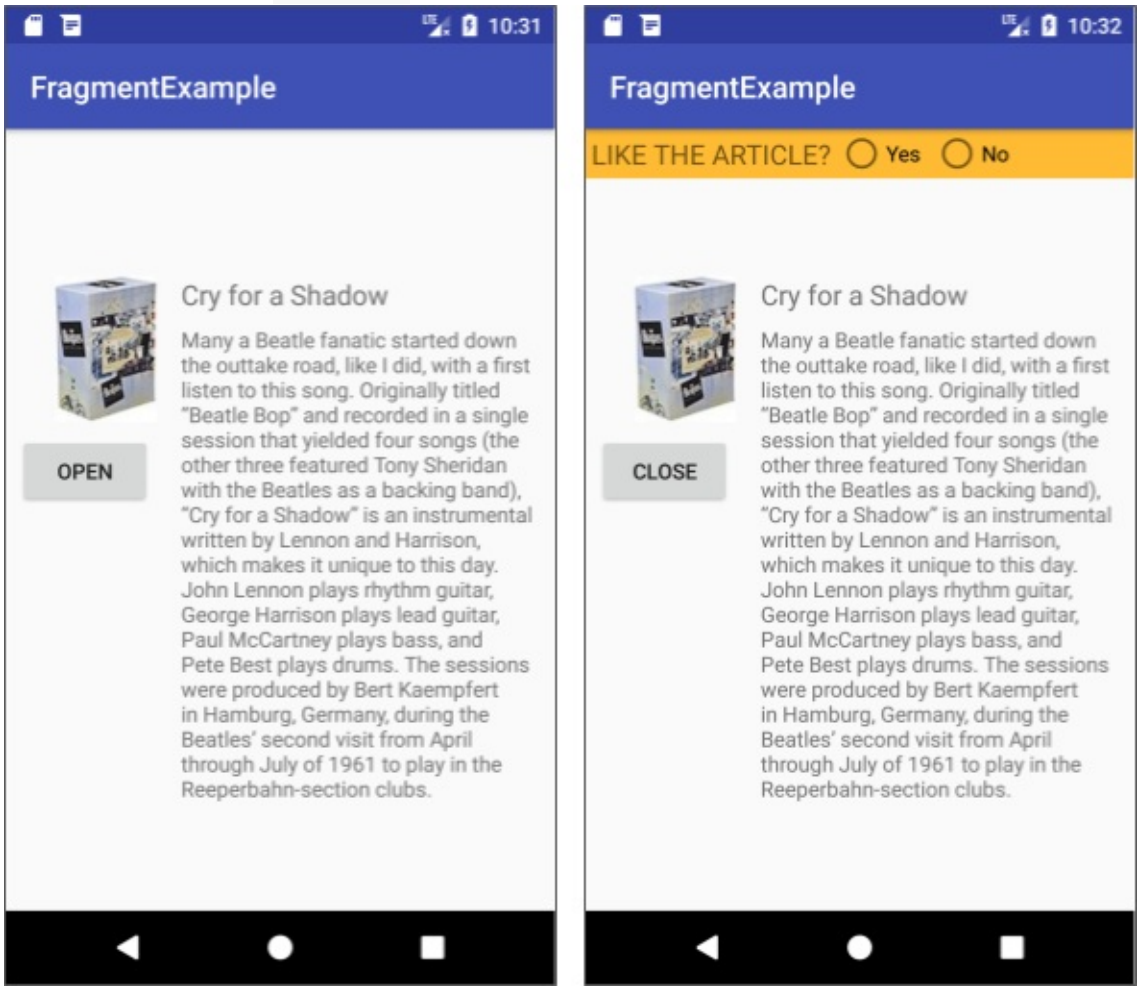




In the above figure, on the right side, the numbers mean the following:

1. The `Activity` *before* the user event that adds the date picker.
2. A user event, such as clicking a button, adds the date picker to the UI of the `Activity`.
3. The date picker, an instance of `DialogFragment` (a subclass of `Fragment`), which displays a dialog floating on top of the `Activity`.

A `Fragment` can be a static part of an `Activity` UI so that it remains on the screen during the entire lifecycle of the `Activity`, or it can be a dynamic part of the UI, added and removed while the `Activity` is running. For example, the `Activity` can include buttons to

open and close the `Fragment`.




## The benefits of using fragments

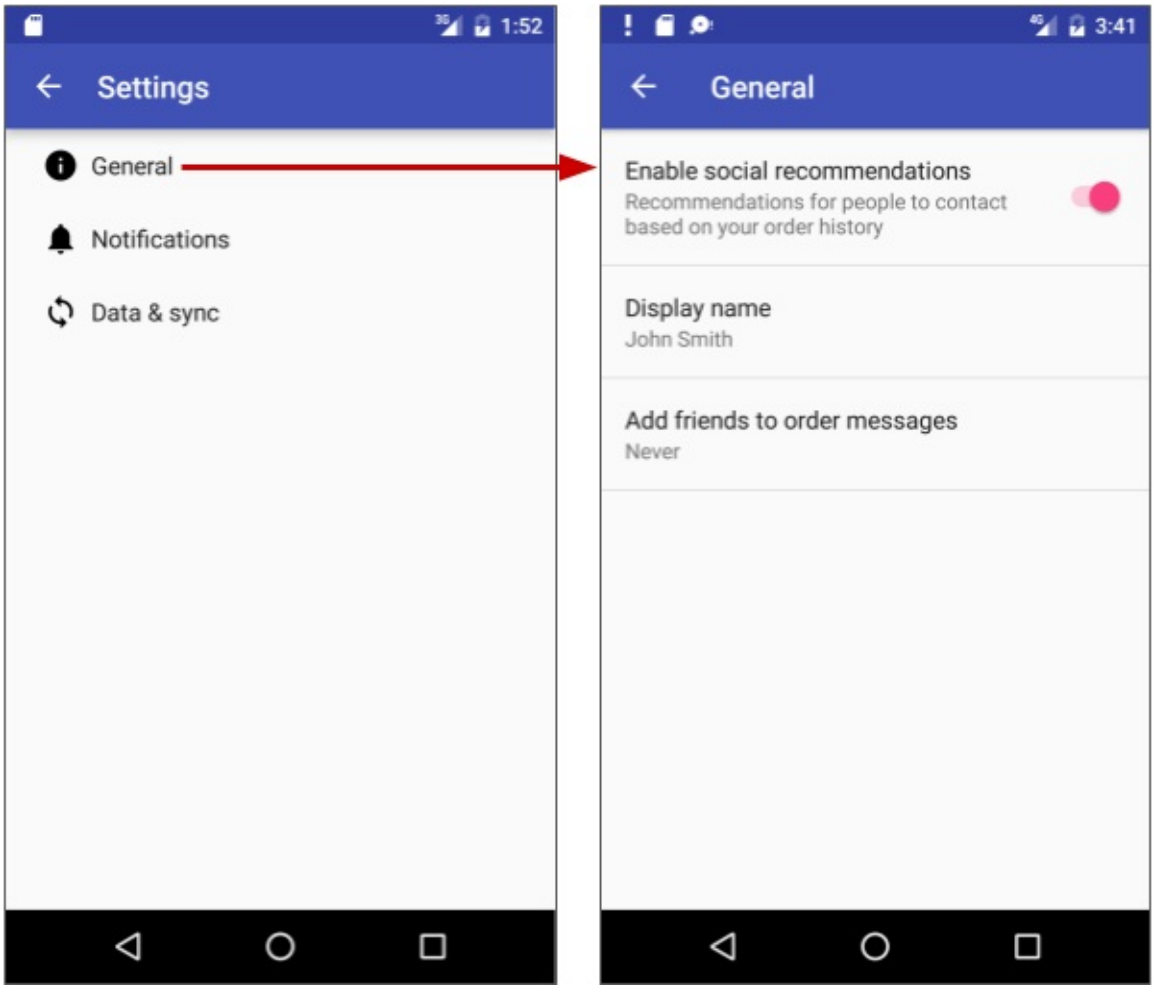
Like any `Fragment`, the date picker includes the code for user interaction (in this case, selecting the date). The `Fragment` also has its own lifecycle—it can be added and removed by the user. These two characteristics of `Fragment` let you:

- **Reuse a `Fragment`**. Write the `Fragment` code once, and reuse the `Fragment` in more than one `Activity` without having to repeat code.
- **Add or remove a `Fragment` dynamically**. Add, replace, or remove a `Fragment` from an `Activity` as needed.
- **Integrate a mini-UI within the `Activity`**. Integrate a `Fragment` with an `Activity` UI or overlay the UI, so that the user can interact with the `Fragment` UI without leaving the `Activity`.
- **Retain data instances after a configuration change.** Since a `Fragment` has its own lifecycle, it can retain an instance of its data after a configuration change (such as changing the device orientation).
- **Represent sections of a layout for different screen sizes**. Encapsulating an interactive UI within a `Fragment` makes it easier to display the interactive UI on different

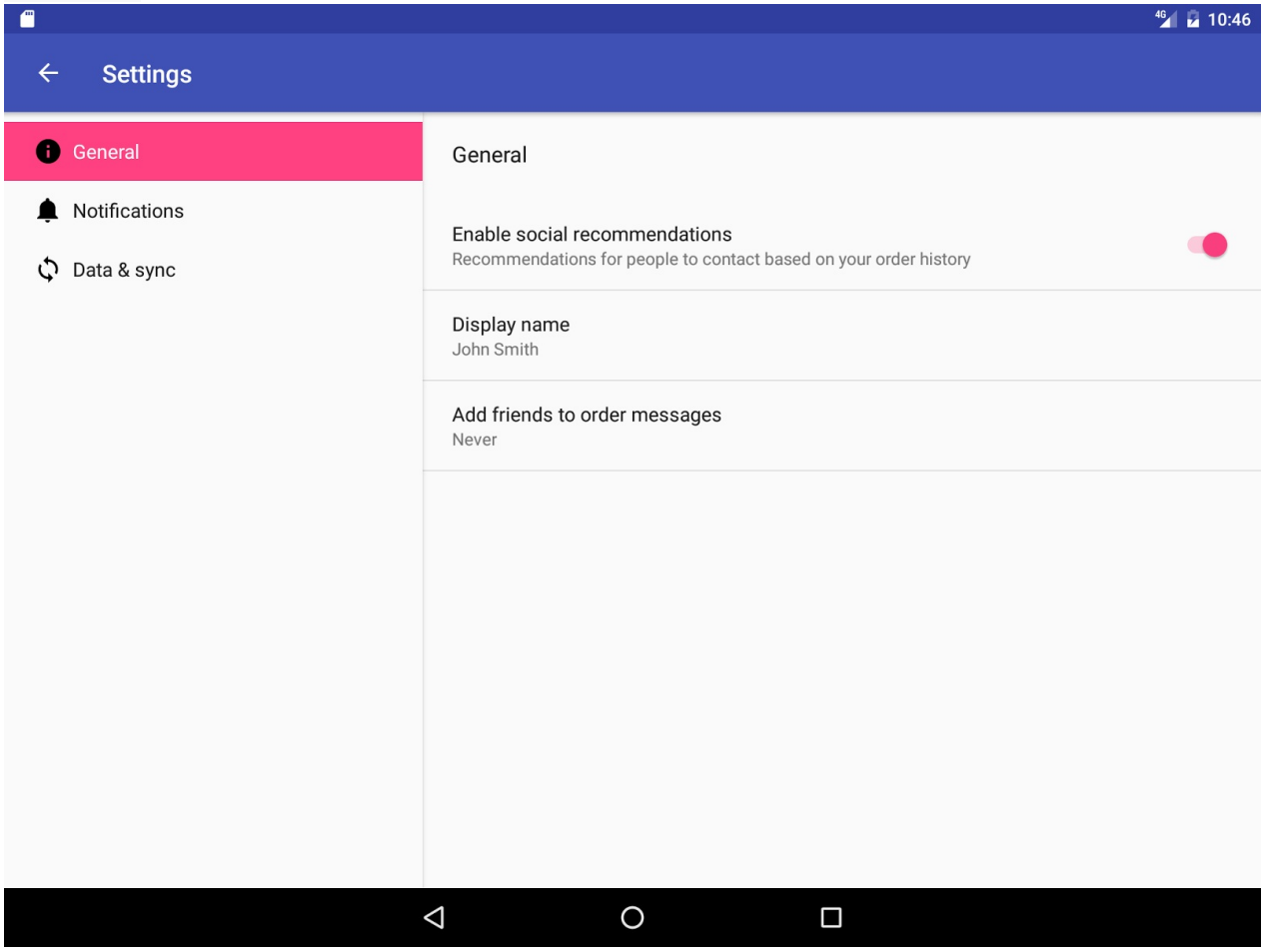
screen sizes.

For an example of how a `Fragment` can be used to show a UI in different screen sizes, start a new Android Studio project for an app and choose the Settings Activity template. Run the app.

The template provides a `Fragment` to show the list of categories (left side of figure below), and a `Fragment` for each category (such as General) to show the settings in that category (right side of figure below). In layout terms, the list screen is known as the "master," and the screen showing the settings in a category is known as the "detail."



If you run the same app on a large-screen tablet in landscape orientation, the UI for each `Fragment` appears with the master and detail panes side by side, as shown below.




## Using a fragment

The general steps to use a `Fragment`:

1. Create a subclass of `Fragment`.
2. Create a layout for the `Fragment`.
3. Add the `Fragment` to a host `Activity`, either statically or dynamically.

# Creating a fragment

To create a `Fragment` in an app, extend the `Fragment` class, then override key lifecycle methods to insert your app logic, similar to the way you would with an `Activity` class.

Instead of extending the base `Fragment` class, you can extend one of these other, more specific `Fragment` subclasses:

- `DialogFragment`: Displays a floating dialog, such as a date picker or time picker.
- `ListFragment`: Displays a list of items that are managed by an adapter (such as a

`SimpleCursorAdapter`).

- `PreferenceFragment`: Displays a hierarchy of `Preference` objects as a list, similar to `PreferenceActivity`. This is useful when creating a "settings" `Activity` for your app.

You can create a `Fragment` in Android Studio by following these steps:

1. In **Project: Android** view, expand **app > java** and select the folder containing the Java code for your app.
2. Choose **File > New > Fragment > Fragment (Blank)**.
3. Name the `Fragment` something like **SimpleFragment**, or use the supplied name (`BlankFragment`).
4. If your `Fragment` has a UI, check the **Create layout XML** option if it is not already checked. Other options include:
    - **Include fragment factory methods**: Include sample factory method code to initialize the `Fragment` arguments in a way that encapsulates and abstracts them. Select this option if the number of arguments would make a constructor too complex.
    - **Include interface callbacks**: Select this option if you want to include sample code that defines an interface in the `Fragment` with callback methods that enable the `Fragment` to communicate with its host `Activity`.
5. Click **Finish** to create the `Fragment`.

If you named the `Fragment` `SimpleFragment`, the following code appears in the `Fragment`:

```
public class SimpleFragment extends Fragment {

    public SimpleFragment() {
        // Required empty public constructor
    }
    // ...
}
```

All subclasses of `Fragment` must include a public no-argument constructor as shown, with the code `public SimpleFragment()`. The Android framework often re-instantiates a `Fragment` class when needed, in particular during state restore. The framework needs to be able to find this constructor so it can instantiate the `Fragment`.

# Creating a layout for a fragment

If you check the **Create layout XML** option when creating a `Fragment`, the layout file is created for you and named after the `Fragment`, for example "fragment_simple.xml" for `SimpleFragment`. As an alternative, you can manually add a layout file to your project. The

layout includes all UI elements that appear in the `Fragment`.

For maximum reusability, make your `Fragment` self-contained. A single `Fragment` should contain all necessary code to define its layout and its behavior for user interaction. Similar to an `Activity`, a `Fragment` inflates its layout to make it appear on the screen. Android calls the `onCreateView()` callback method to display a `Fragment`. Override this method to inflate the layout for a `Fragment`, and return a View that is the root of the layout for the `Fragment`.

For example, if you chose **Fragment (Blank)** with just the **Create layout XML** option, and you name the `Fragment` **SimpleFragment**, the following code is generated in `SimpleFragment`:

```
@Override
public View onCreateView(LayoutInflater inflater,
                         ViewGroup container, Bundle savedInstanceState) {
    // Inflate the layout for this fragment
    return inflater.inflate(R.layout.fragment_simple, container, false);
}
```

The `container` parameter passed to `onCreateView()` is the parent `ViewGroup` from the `Activity` layout. Android inserts the `Fragment` layout into this `ViewGroup`.

The `onCreateView()` callback provides a `LayoutInflater` object to inflate the UI for the `Fragment` from the `fragment_simple` layout resource. The method returns a `View` that is the root of the layout for the `Fragment`.

The `savedInstanceState` parameter is a `Bundle` that provides data about the previous instance of the `Fragment`, in case the `Fragment` is resuming.

The `inflate()` method inside `onCreateView()` displays the layout:

```
// Inflate the layout for this fragment
return inflater.inflate(R.layout.fragment_simple, container, false);
```

The `inflate()` method takes three arguments:

- The resource ID of the layout you want to inflate ( `R.layout.fragment_simple` ).
- The `ViewGroup` to be the parent of the inflated layout ( `container` ).
- A boolean indicating whether the inflated layout should be attached to the `ViewGroup` ( `container` ) during inflation. This should be `false` because the system is already inserting the inflated layout into the container. Passing `true` would create a redundant `ViewGroup` in the final layout.

**Tip**: The `Fragment` class contains other lifecycle callback methods to override besides `onCreateView()`, such as `onCreate()`, `onStart()`, `onPause()`, and `onStop()`. The only lifecycle callback you need to inflate the layout is `onCreateView()`. To learn about other lifecycle callbacks, see the lesson on `Fragment` lifecycle and communications.

# Adding a fragment to an activity

A `Fragment` must be hosted by an `Activity` and included in its layout. There are two ways you can use a `Fragment` in an `Activity` layout:

- Add the `Fragment` *statically,* inside the XML layout file for the `Activity`, so that it remains on the screen during the entire lifecycle of the `Activity`.

  For example, you may want to devote a portion of a UI for an `Activity` to a `Fragment` that provides its own user interaction and behavior, such as a set of social media "Like" buttons. You can add this `Fragment` to the layouts of different activities.

- Add the `Fragment` *dynamically,* using fragment transactions. During the lifecycle of the `Activity`, your code can add or remove the `Fragment`, or replace it with another `Fragment`, as needed.

## Adding a fragment statically

Declare the `Fragment` inside the layout file for the `Activity` (such as `activity_main.xml`) using the `<fragment>` tag. You can specify layout properties for the `Fragment` as if it were a `View`. For example, the following shows two `Fragment` objects included in the `Activity` layout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment android:name="com.example.news.ArticleListFragment"
            android:id="@+id/list"
            android:layout_weight="1"
            android:layout_width="0dp"
            android:layout_height="match_parent" />

    <fragment android:name="com.example.news.ArticleReaderFragment"
            android:id="@+id/viewer"
            android:layout_weight="2"
            android:layout_width="0dp"
            android:layout_height="match_parent" />
</LinearLayout>
```

When the system creates the `Activity` layout, it instantiates each `Fragment` specified in the layout, and calls the `onCreateView()` method for each one, to retrieve the layout for each `Fragment`. Each `Fragment` returns a `View`, and the system inserts this `View` directly in place of the `<fragment>` element.

The code above uses the `android:id` attribute to identify each `Fragment` element. The system uses this `id` to restore the `Fragment` if the `Activity` is restarted. You also use it in your code to refer to the `Fragment`.

## Adding a fragment dynamically

A great feature of the `Fragment` class is the ability to add, remove, or replace a `Fragment` dynamically, while an `Activity` is running. A user performs an interaction in the `Activity`, such as tapping a button, and the `Fragment` appears in the UI of the `Activity`. The user taps another button to remove the `Fragment`.

To add a `Fragment`, your `Activity` code needs to specify a `ViewGroup` as a placeholder for the `Fragment`, such as a `LinearLayout` or a `FrameLayout`:

```
<FrameLayout
    android:id="@+id/fragment_container"
    android:name="SimpleFragment"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:layout="@layout/fragment_simple">
</FrameLayout>
```

To manage a `Fragment` in your `Activity`, create an instance of the `Fragment`, and an instance of `FragmentManager` using `getSupportFragmentManager()`. With `FragmentManager` you can use `FragmentTransaction` methods to perform `Fragment` operations while the `Activity` runs.

`Fragment` operations are wrapped into a *transaction* so that all of the operations finish before the transaction is committed for the final result. You start a transaction with `beginTransaction()` and end it with `commit()`.

Within the transaction you can:

- Add a `Fragment` using `add()`.
- Remove a `Fragment` using `remove()`.
- Replace a `Fragment` with another `Fragment` using `replace()`.
- Hide and show a `Fragment` using `hide()` and `show()`.

The best practice for instantiating the `Fragment` in the A`ctivity` is to provide a `newinstance()` factory method in the `Fragment`. For example, if you choose **New > Fragment > Fragment Blank** to add a `Fragment` to your Android Studio project, and you select the **Include fragment factory methods** option, Android Studio automatically adds a `newinstance()` method to the `Fragment` as a factory method to set arguments for the `Fragment` when the `Fragment` is called by the `Activity`.

Add a simple `newinstance()` method to `SimpleFragment`, and instantiate the `Fragment` in `MainActivity`:

1. Open `SimpleFragment`, and add the following method to the end for instantiating the `Fragment`:

```
public static SimpleFragment newInstance() {
        return new SimpleFragment();
}
```

2. In the `Activity`, instantiate the `Fragment` by calling the `newInstance()` method in `SimpleFragment`:

```
SimpleFragment fragment = SimpleFragment.newInstance();
```

3. In the `Activity`, get an instance of `FragmentManager` with `getSupportFragmentManager()`.

```
FragmentManager fragmentManager = getSupportFragmentManager();
```

   **Tip**: Use the Support Library version— `getSupportFragmentManager()` rather than `getFragmentManager()` —so that the app remains compatible with devices running earlier versions of Android platform.

4. Use `beginTransaction()` with an instance of `FragmentTransaction` to start a series of edit operations on the `Fragment`:

```
FragmentTransaction fragmentTransaction =
                                    fragmentManager.beginTransaction();
```

5. You can then add a `Fragment` using the `add()` method, and commit the transaction with `commit()`. For example:

```
fragmentTransaction.add(R.id.fragment_container, fragment);
fragmentTransaction.commit();
```

The first argument passed to `add()` is the `ViewGroup` in which the `fragment` should be placed (specified by its resource ID `fragment_container`). The second parameter is the `fragment` to add.

In addition to the `add()` transaction, call `addToBackStack(null)` in order to add the transaction to a back stack of `Fragment` transactions. This back stack is managed by the `Activity`. It allows the user to return to the previous `Fragment` state by pressing the **Back** button:

```
fragmentTransaction.add(R.id.fragment_container, fragment);
fragmentTransaction.addToBackStack(null);
fragmentTransaction.commit();
```

To replace a `Fragment` with another `Fragment`, use the `replace()` method. To remove a `Fragment`, use `remove()`. Once you've made your changes with `FragmentTransaction`, you must call `commit()` for the changes to take effect.

The following shows a transaction that removes the `Fragment` `simpleFragment` using `remove()`:

```
// Get the FragmentManager.
FragmentManager fragmentManager = getSupportFragmentManager();
// Check to see if the fragment is already showing.
SimpleFragment simpleFragment = (SimpleFragment) fragmentManager
                                .findFragmentById(R.id.fragment_container);
if (simpleFragment != null) {
    // Create and commit the transaction to remove the fragment.
    FragmentTransaction fragmentTransaction =
                                        fragmentManager.beginTransaction();
    fragmentTransaction.remove(simpleFragment).commit();
}
```

In addition to learning about how a `Fragment` can be added, replaced, and removed, you should also learn how to manage the lifecycle of a `Fragment` within the `Activity`, as described in the lesson on `Fragment` lifecycle and communications.

# Related practical

The related practical is Creating a Fragment with a UI.

# Learn more

Android developer documentation:

- Fragments
- FragmentManager
- FragmentTransaction
- Creating a Fragment
- Building a Flexible UI
- Building a Dynamic UI with Fragments
- Supporting Tablets and Handsets

Video:

- What the Fragment? (Google I/O 2016)
- Fragment Tricks (Google I/O '17)
- Por que Precisamos de Fragments?

# 1.2: Fragment lifecycle and communications

**Contents:**



Like an `Activity`, a `Fragment` has its own lifecycle. Understanding the relationship between `Activity` and `Fragment` lifecycles helps you design fragments that can save and restore variables and communicate with activities.

An `Activity` that hosts a `Fragment` can send information to that `Fragment`, and receive information from that `Fragment`. This chapter describes the mechanisms for passing data and how to manage the `Fragment` lifecycle within an `Activity`.

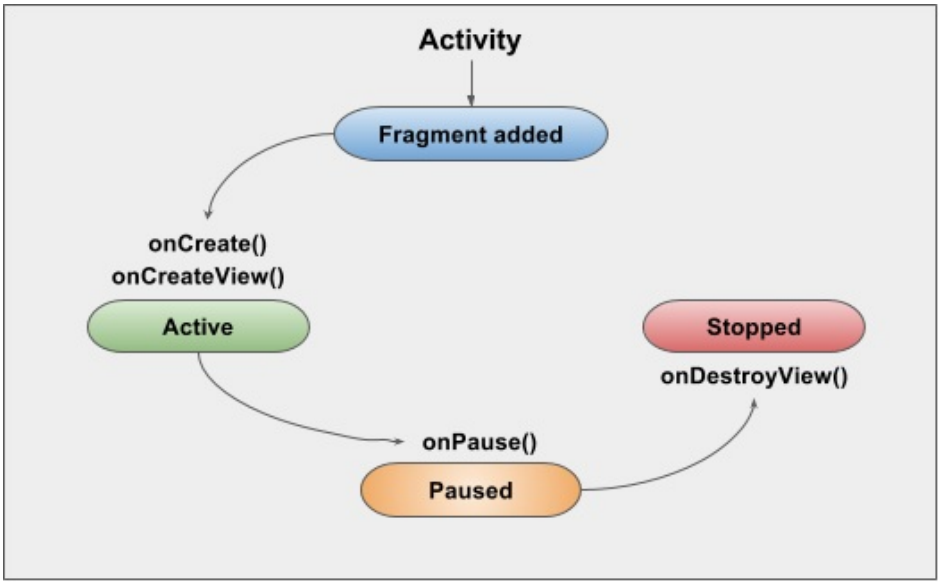
## Understanding the Fragment lifecycle

Using a `Fragment` lifecycle is a lot like using an `Activity` lifecycle (see The Activity Lifecycle for details). Within the `Fragment` lifecycle callback methods, you can declare how your `Fragment` behaves when it is in a certain state, such as active, paused, or stopped.

### Fragment lifecycle states

The `Fragment` is added by an `Activity` (which acts as the host of the `Fragment`). Once added, the `Fragment` goes through three states, as shown in the figure below:

- Active (or resumed)
- Paused

- Stopped 


## How the Activity state affects the Fragment

Because a `Fragment` is always hosted by an `Activity`, the `Fragment` lifecycle is directly affected by the host `Activity` lifecycle. For example, when the `Activity` is paused, so are all `Fragments` in it, and when the `Activity` is destroyed, so are all `Fragments`.

Each lifecycle callback for the `Activity` results in a similar callback for each `Fragment`, as shown in the following table. For example, when the `Activity` receives `onPause()`, it triggers a `Fragment` `onPause()` for each `Fragment` in the `Activity`.

| `Activity` State | `Fragment` Callbacks Triggered | `Fragment` Lifecycle |
|---|---|---|
| Created | onAttach(), onCreate(), onCreateView(), onActivityCreated() | `Fragment` is added and its layout is inflated. |
| Started | onStart() | `Fragment` is active and visible. |
| Resumed | onResume() | `Fragment` is active and ready for user interaction. |
| Paused | onPause() | `Fragment` is paused because the `Activity` is paused. |
| Stopped | onStop() | `Fragment` is stopped and no longer visible. |
| Destroyed | onDestroyView(), onDestroy(), onDetach() | `Fragment` is destroyed. |

As with an `Activity`, you can save the variable assignments in a `Fragment`. Because data in a `Fragment` is usually relevant to the `Activity` that hosts it, your `Activity` code can use a callback to retrieve data from the `Fragment`, and then restore that data when

recreating the `Fragment`. You learn more about communicating between an `Activity` and a `Fragment` later in this chapter.

**Tip**: For more information about the `Activity` lifecycle and saving state, see The Activity Lifecycle.

# Using Fragment lifecycle callbacks

As you would with `Activity` lifecycle methods, you can override `Fragment` lifecycle methods to perform important tasks when the `Fragment` is in certain states. Most apps should implement at least the following methods for every `Fragment`:

- `onCreate()`: Initialize essential components and variables of the `Fragment` in this callback. The system calls this method when the `Fragment` is created. Anything initialized in `onCreate()` is preserved if the `Fragment` is paused and resumed.
- `onCreateView()`: Inflate the XML layout for the `Fragment` in this callback. The system calls this method to draw the `Fragment` UI for the first time. As a result, the `Fragment` is visible in the `Activity`. To draw a UI for your `Fragment`, you must return the root `View` of your `Fragment` layout. Return `null` if the `Fragment` does not have a UI.
- `onPause()`: Save any data and states that need to survive beyond the destruction of the `Fragment`. The system calls this method if any of the following occurs:
    - The user navigates backward.
    - The `Fragment` is replaced or removed, or another operation is modifying the `Fragment`.
    - The host `Activity` is paused.

A paused `Fragment` is still alive (all state and member information is retained by the system), but it will be destroyed if the `Activity` is destroyed. If the user presses the **Back** button and the `Fragment` is returned from the back stack, the lifecycle resumes with the `onCreateView()` callback.

The `Fragment` class has other useful lifecycle callbacks:

- `onResume()`: Called by the `Activity` to resume a `Fragment` that is visible to the user and actively running.
- `onAttach()`: Called when a `Fragment` is first attached to a host `Activity`. Use this method to check if the `Activity` has implemented the required listener callback for the `Fragment` (if a listener interface was defined in the `Fragment`). After this method, `onCreate()` is called.
- `onActivityCreated()`: Called when the `Activity` `onCreate()` method has returned. Use it to do final initialization, such as retrieving views or restoring state. It is also useful

for a `Fragment` that uses `setRetainInstance()` to retain its instance, as this callback tells the `Fragment` when it is fully associated with the new `Activity` instance. The `onActivityCreated()` method is called after `onCreateView()` and before `onViewStateRestored()`.

- `onDestroyView()`: Called when the `View` previously created by `onCreateView()` has been detached from the `Fragment`. This call can occur if the host `Activity` has stopped, or the `Activity` has removed the `Fragment`. Use it to perform some action, such as logging a message, when the `Fragment` is no longer visible. The next time the `Fragment` needs to be displayed, a new `View` is created. The `onDestroyView()` method is called after `onStop()` and before `onDestroy()`.

For a complete description of all `Fragment` lifecycle callbacks, see Fragment.

# Using Fragment methods and the Activity context

An Activity can use methods in a `Fragment` by first acquiring a reference to the `Fragment`. Likewise, a `Fragment` can get a reference to its hosting `Activity` to access resources, such as a `View`.

## Using the Activity context in a Fragment

When a `Fragment` is in the active or resumed state, it can get a reference to its hosting `Activity` instance using `getActivity()`. It can also perform tasks such as finding a `View` in the `Activity` layout:

```
View listView = getActivity().findViewById(R.id.list);
```

Note that if you call `getActivity()` when the `Fragment` is not attached to an `Activity`, `getActivity()` returns `null`.

## Using the Fragment methods in the host Activity

Likewise, your `Activity` can call methods in the `Fragment` by acquiring a reference to the `Fragment` from `FragmentManager`, using `findFragmentById()`. For example, to call the `getSomeData()` method in the `Fragment`, acquire a reference first:

```
ExampleFragment fragment = (ExampleFragment)
                getFragmentManager().findFragmentById(R.id.example_fragment);
// ...
mData = fragment.getSomeData();
```

## Adding the Fragment to the back stack

A significant difference between an `Activity` and a `Fragment` is how activities and fragments use their respective back stacks, so that the user can navigate back with the **Back** button (as discussed in Tasks and Back Stack).

- For an `Activity`, the system automatically maintains a back stack of activities.
- For a `Fragment`, the hosting `Activity` maintains a back stack, and you have to explicitly add a `Fragment` to that back stack by calling `addToBackStack()` during any transaction that adds the `Fragment`.

Keep in mind that when your app replaces or removes a `Fragment`, it's often appropriate to allow the user to navigate backward and "undo" the change. To allow the user to navigate backward through `Fragment` transactions, call `addToBackStack()` before you commit the `FragmentTransaction`:

```
fragmentTransaction.add(R.id.fragment_container, fragment);
fragmentTransaction.addToBackStack(null);
fragmentTransaction.commit();
```

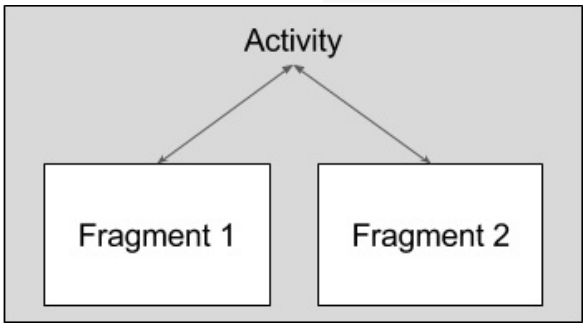
**Tip**: The `addToBackStack()` method takes an optional string parameter that specifies a unique name for the transaction. Specify `null` because the name isn't needed unless you plan to perform advanced `Fragment` operations using the `FragmentManager.BackStackEntry` interface.

When removing a `Fragment`, remember that the hosting `Activity` maintains a back stack for the `Fragment` (if you add the transaction to it, as described above). However, the `Fragment` is not destroyed. If the user navigates back to restore the `Fragment`, it restarts.

# Communicating between a Fragment and an Activity

The `Activity` that hosts a `Fragment` can send information to that `Fragment`, and receive information from that `Fragment`, as described in "Sending data from a fragment to its host activity" in this chapter.

However, a `Fragment` can't communicate directly with another `Fragment`. All `Fragment`-to-`Fragment` communication is done through the `Activity` that hosts them. One `Fragment` communicates with the `Activity`, and the `Activity` communicates to the other `Fragment`.




## Sending data to a Fragment from an Activity

To send data to a `Fragment` from an `Activity`, set a `Bundle` and use the `Fragment` method `setArguments(Bundle)` to supply the construction arguments for the `Fragment`.

For example, if the user previously made a choice in the `Fragment`, and you want to send that choice back to the `Fragment` when starting it from the `Activity`, you would create the `arguments` `Bundle` and insert the `String` value of the choice into the `Bundle` mapping for the key ( `"choice"` ).

The best practice for initializing the data in a `Fragment` is to perform this initialization in the `Fragment` in a factory method. As you learned in the lesson on fragments, you can create the `Fragment` instance with a `newinstance()` method in the `Fragment` itself:

```
public static SimpleFragment newInstance() {
        return new SimpleFragment();
}
```

Then instantiate the `Fragment` in an `Activity` by calling the `newInstance()` method in the `Fragment`:

```
SimpleFragment fragment = SimpleFragment.newInstance();
```

**Tip**: If you choose **New > Fragment > Fragment Blank** to add a `Fragment` to your Android Studio project, and you select the **Include fragment factory methods** option, Android Studio automatically adds a `newinstance()` method to the `Fragment` as a factory method to set arguments for the `Fragment` when the `Fragment` is called by the `Activity`.

For example, to open the `Fragment` with the user's previously selected choice, all you need to do in the `Activity` is to provide the preselected choice as an argument for the `newInstance()` method when instantiating the `Fragment` in the `Activity`:

```
SimpleFragment fragment = SimpleFragment.newInstance(mRadioButtonChoice);
```

The `newInstance()` factory method in the `Fragment` can use a `Bundle` and the `setArguments(Bundle)` method to set the arguments before returning the `Fragment`:

```
public static SimpleFragment newInstance(int choice) {
        SimpleFragment fragment = new SimpleFragment();
        Bundle arguments = new Bundle();
        arguments.putInt(CHOICE, choice);
        fragment.setArguments(arguments);
        return fragment;
}
```

The following shows how you would use `getSupportFragmentManager()` to get an instance of `FragmentManager`, create the `fragmentTransaction`, acquire the `Fragment` using the layout resource, and then create the `Bundle`.

```
// Get the FragmentManager and start a transaction.
FragmentManager fragmentManager = getSupportFragmentManager();
FragmentTransaction fragmentTransaction = fragmentManager
                                          .beginTransaction();
// Instantiate the fragment.
SimpleFragment fragment =
                SimpleFragment.newInstance(mRadioButtonChoice);
// Add the fragment.
fragmentTransaction.add(R.id.fragment_container, fragment).commit();
```

The `SimpleFragment` `newInstance()` method receives the value of the `mRadioButtonChoice` argument in `choice`:

```
public static SimpleFragment newInstance(int choice) {
        SimpleFragment fragment = new SimpleFragment();
        Bundle arguments = new Bundle();
        arguments.putInt(CHOICE, choice);
        fragment.setArguments(arguments);
        return fragment;
}
```

Before you draw the `View` for the `Fragment`, retrieve the passed-in arguments from the `Bundle`. To do this, use `getArguments()` in the `Fragment` `onCreate()` or `onCreateView()` callback, and if `mRadioButtonChoice` is not `NONE`, check the radio button for the choice:

```
if (getArguments().containsKey(CHOICE)) {
    // A choice was made, so get the choice.
    mRadioButtonChoice = getArguments().getInt(CHOICE);
    // Check the radio button choice.
    if (mRadioButtonChoice != NONE) {
         radioGroup.check
             (radioGroup.getChildAt(mRadioButtonChoice).getId());
    }
}
```

## Sending data from a Fragment to its host Activity

To have a `Fragment` communicate to its host `Activity`, follow these steps in the `Fragment`:

1. Define a listener interface, with one or more callback methods to communicate with the `Activity`.
2. Override the `onAttach()` lifecycle method to make sure the host `Activity` implements the interface.
3. Call the interface callback method to pass data as a parameter.

In the host `Activity`, follow these steps:

1. Implement the interface defined in the `Fragment`. (All the `Activity` classes that use the `Fragment` have to implement the interface.)
2. Implement the `Fragment` callback method(s) to retrieve the data.

The following is an example of defining the `OnFragmentInteractionListener` interface in the `Fragment`, including the `onRadioButtonChoice()` callback to communicate to the host `Activity`. The `Activity` must implement this interface. The `onAttach()` method gets a reference to the listener if the `Activity` implemented this interface; if not, this method throws an exception:

```
// Interface definition and onFeedbackChoice() callback.
interface OnFragmentInteractionListener {
    void onRadioButtonChoice(int choice);
}

@Override
public void onAttach(Context context) {
    super.onAttach(context);
    if (context instanceof OnFragmentInteractionListener) {
        mListener = (OnFragmentInteractionListener) context;
    } else {
        throw new ClassCastException(context.toString()
                    + " must implement OnFragmentInteractionListener");
    }
}
```

The following shows how the `Fragment` uses the `onCheckedChanged()` listener for checked radio buttons in the `Fragment`, and uses the `onRadioButtonChoice()` callback to provide data to the host `Activity`.

```
public void onCheckedChanged(RadioGroup group, int checkedId) {
    View radioButton = radioGroup.findViewById(checkedId);
    int index = radioGroup.indexOfChild(radioButton);
    switch (index) {
        case YES: // User chose "Yes."
            textView.setText(R.string.yes_message);
            mRadioButtonChoice = YES;
            mListener.onRadioButtonChoice(YES);
            break;
        case NO: // User chose "No."
            textView.setText(R.string.no_message);
            mRadioButtonChoice = NO;
            mListener.onRadioButtonChoice(NO);
            break;
        default: // No choice made.
            mRadioButtonChoice = NONE;
            mListener.onRadioButtonChoice(NONE);
            break;
    }
}
```

To use the `Fragment` callback method(s) to retrieve data, the `Activity` must implement the interface defined in the `Fragment` class:

```
public class MainActivity extends AppCompatActivity
                implements SimpleFragment.OnFragmentInteractionListener {
   //...
}
```

The Activity can then use the onRadioButtonChoice() callback to get the data.

```
@Override
public void onRadioButtonChoice(int choice) {
    mRadioButtonChoice = choice;
    Toast.makeText(this, "Choice is " + Integer.toString(choice),
                                            LENGTH_SHORT).show();
}
```

Some data in a `Fragment` is may be relevant to the `Activity` that hosts it. Your `Activity` code can use a callback to retrieve relevant data from the `Fragment`. The `Activity` can then send that data to the `Fragment` when recreating the `Fragment`.

# Related practical

The related concept documentation is Communicating with a Fragment.

# Learn more

Android developer documentation:

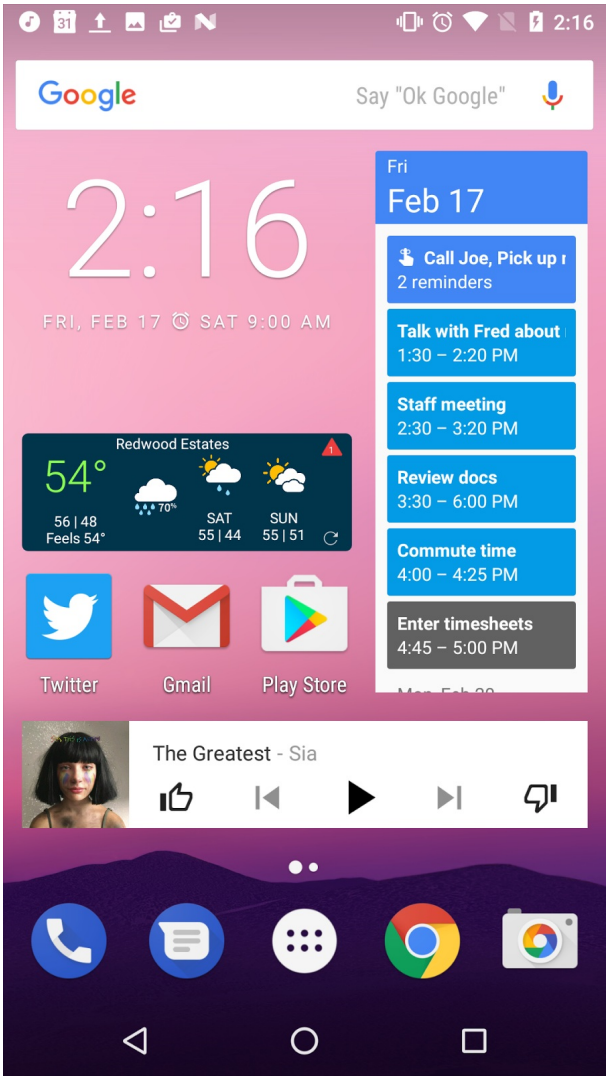
- Fragments
- FragmentManager
- FragmentTransaction
- Communicating with Other Fragments
- The Activity Lifecycle
- Tasks and Back Stack

# 2.1: App widgets

**Contents:**



An *app widget* is a miniature app view that appears on the Android home screen and can be updated periodically with new data. App widgets display small amounts of information or perform simple functions such as showing the time, summarizing the day's calendar events,




or controlling music playback.

In this chapter you learn how to provide an app widget for your Android app, how to update that app widget with new data, and how to provide click actions for that widget or any of its parts.

**Note:** The term "widget" in Android also commonly refers to the user interface elements (views) you use to build an app, such as buttons and checkboxes. In this chapter, all instances of the word widget refer to app widgets.

# About app widgets

App widgets are miniature apps that appear on any Android home screen panel. Widgets provide an at-a-glance view of your app's most important data and features, even if that app is not running.

When the user installs your app on a device, that app's associated widgets are listed on the widget install screen. To get to the widget install screen, touch & hold on an empty spot on a home screen and tap **Widgets**. Touch & hold any widget to pick it up and add it to a home

screen, move it around, or resize it, if it is resizeable.

**Note:** As of Android 5.0, widgets can only be placed on the Android home screen. Previous versions of Android (4.2/API 17 to 4.4/API 19) also allowed widgets to appear on the lock screen (keyguard). Although the app widget tools and APIs still occasionally mention lock screen widgets, that functionality is deprecated. This chapter discusses only the home-screen app widgets.
App widgets can display data, be interactive, or both. Data widgets can have a simple flat layout (such as a clock or weather widget) or use a scrolling list for a collection of items (such as a calendar or to do list widget). App widget controls can provide shortcut actions (such as play/pause for a music player). Even simple app widgets may include an action to open their associated app when tapped.

As a general rule, app widgets should be small, simple, and provide limited information or functionality.

## The components of an app widget

App widgets are add-ons for an existing app and are made available to users when the app is installed on a device. Your app can have multiple widgets. You cannot create a stand-alone app widget without an associated app.

App widgets include the following components in your Android Studio project. You learn more about these components in the latter part of this chapter.

- Provider-info file: App widgets include an XML file that defines metadata about the widget. That metadata includes the widget's initial or minimum size, its update interval, configuration activity (if any), and preview image.
- Layout: You define the user interface for your app widget in an XML layout file, just as you do for regular apps. However, app widgets support a more limited set of layouts and views than regular apps.
- Widget-provider class: The app widget provider contains the Java code for your app widget. App widgets are actually broadcast receivers—they extend the `AppWidgetProvider` class, which in turn extends from `BroadcastReceiver`. `AppWidgetProvider` objects receive only broadcasts relevant to the app widget, including intents for update, enabled, disabled, and deleted. As with all broadcast receivers, app widget providers are declared in the Android manifest.
- Configuration activity (optional): If your app widget must be configured before it is used, you can implement an activity that provides that configuration. The configuration activity is then listed in the app widget's provider-info file and is automatically launched when the user adds a widget to the home screen. You can see this behavior with the Android News & Weather widget, which asks you to choose whether to display weather and

stories, weather only, or stories only.

App widgets exist within a system framework controlled by the app widget *manager* and the app widget *host.*

The app widget *manager* ( `AppWidgetManager` ) manages widget updates and sends the broadcast intents that app widget providers receive to do the actual work of updating.

The app widget *host* ( `AppWidgetHost` ) is an app component that can hold and display other app widgets. The Android home screen is by far the most frequently used app widget host, although it is possible to create your own app widget host.

## App widget updates

App widgets that display information can be updated regularly for new or updated information. The data a widget contains can be updated in two ways:

- The widget updates itself at regular intervals. You can define the interval in the widget's provider-info file, but that interval must be at least 30 minutes.
- The widget's associated app can request a widget update explicitly.

In both these cases, the update occurs because the app widget manager sends a broadcast intent with the action `ACTION_APPWIDGET_UPDATE` . The app widget-provider class receives that intent and calls the `onUpdate()` method.

Implement the `onUpdate()` method in your widget-provider class to update your widget. For simple widgets, you might just update the data in the widget's views. More complex widgets may need to interact with additional services to retrieve data from content providers or other data storage locations.

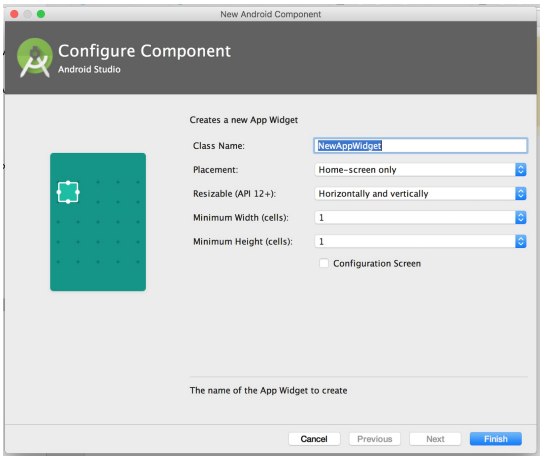
## Update performance

Widget updates consume device resources such as battery life. Updates occur whether or not the underlying app is running, and widgets can wake a sleeping device to perform the update. It is a best practice to design your widgets so that they do not need frequent updates, to conserve these device resources.

**Note:** If your widget must update more frequently than every 30 minutes or you do not need to update while the device is asleep, use an `AlarmManager` . Set the alarm type to either `ELAPSED_REALTIME` or `RTC` , which will only deliver the alarm when the device is awake.

## Configurable app widgets

Some widgets need to be configured before they can be used. For example, a widget that displays a single photo would need to prompt the user to pick the photo from a gallery. A stock-ticker widget would need a list of stock symbols to display.

The configuration activity appears when the user first adds your widget onto the home screen. 

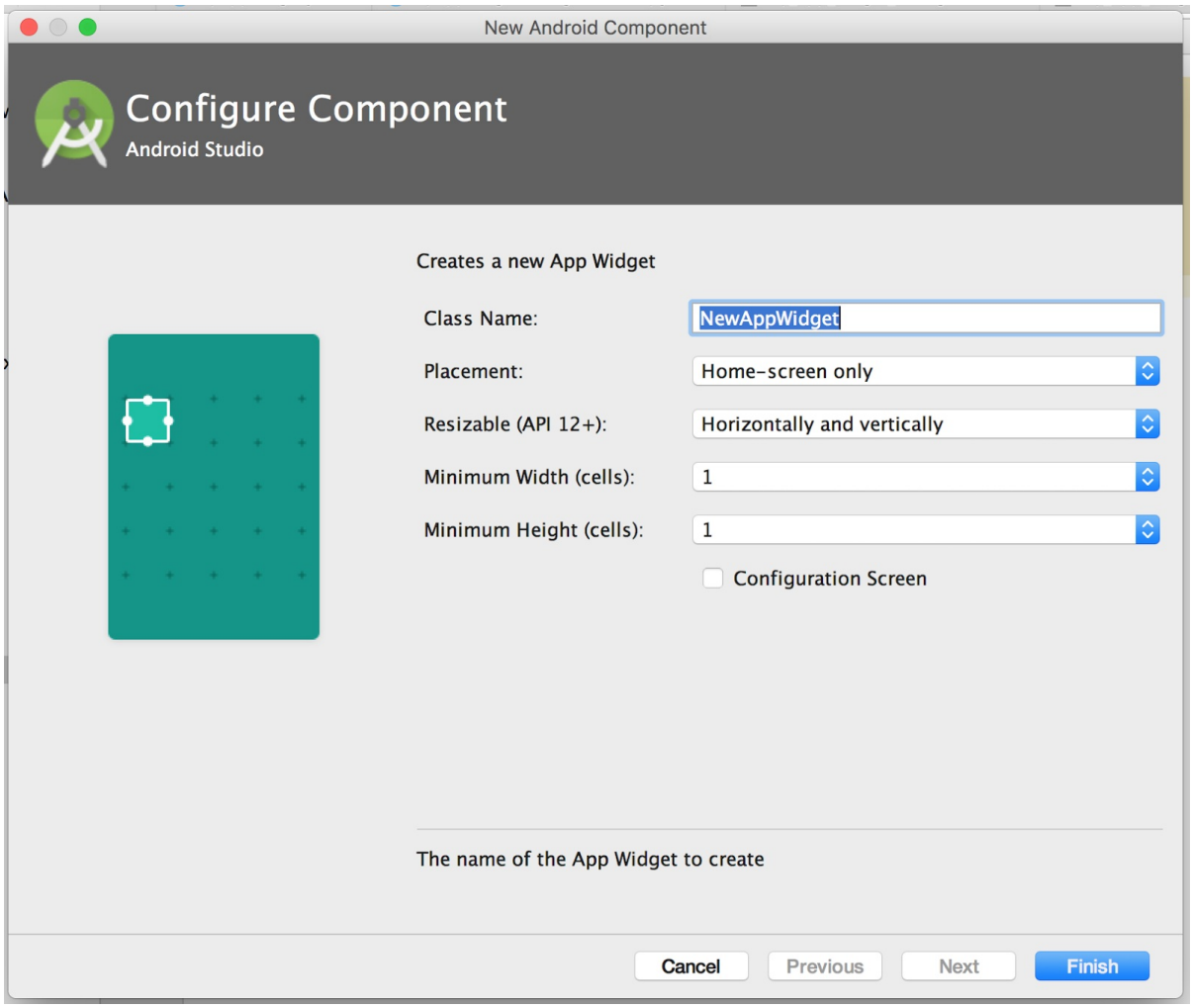

With configurable widgets, the user can place multiple copies of the same widget on their home screens, with different configurations. If any of those widgets receives an update request, you must update *all* the installed widgets at once.

You learn more about widget configuration activities in Using a configuration activity, below.

# Adding an app widget to an app

To add a widget to your app project in Android Studio, select **File > New > Widget > App Widget**.

The **New Android Component** dialog appears for your new widget:




In the above figure:

- **Class Name** is the name for your widget's provider class. This class should contain the word "Widget."
- **Placement** should almost always be **Home-screen only**.
- **Resizeable** indicates whether your widget should be resizeable after it is placed. You can restrict the resizability of your widget to horizontal or vertical only, or make your widget not resizeable.
- **Minimum Width** and **Minimum Height** are the minimum sizes for your widget, measured in cells on the home screen.
- **Configuration Screen** is an option to include an initial widget configuration activity.

After you click **Finish**, several files are added to or modified in your project. For example, if you name your app widget `NewAppWidget`:

- A provider class called `NewAppWidget.java` is added to your project.
- A new layout file is added for the widget in `res/layouts/new_app_widget.xml`.
- A new provider-info file is added in `res/xml/new_app_widget_info.xml`.
- If you selected **Configuration Screen,** a new configuration activity class called

`NewAppWidgetConfigurationActivity.java` is added to your project.
- The Android manifest is updated to include the provider and configuration activity classes.

# Updating the widget provider-info file

The provider-info file defines metadata and initial properties for your app widget, including the widget's initial size, update interval, and configuration activity. These properties are used to display your widget in the widget picker, to configure the widget (if configuration is needed), and to place the widget in the right number of cells on the home screen.

The provider info is an XML resource located in the `res/xml/` folder, and it contains a single `<appwidget-provider>` element. Android Studio creates a template provider-info file when you add a new widget, based on the values you provided in the **New Android Component** dialog.

Here's an example of a provider-info file.

```
<?xml version="1.0" encoding="utf-8"?>
<appwidget-provider
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:minHeight="40dp"
   android:minWidth="40dp"
   android:initialLayout="@layout/new_app_widget"
   android:updatePeriodMillis="86400000"
   android:previewImage="@drawable/new_appwidget_preview"
   android:resizeMode="horizontal|vertical"
   android:widgetCategory="home_screen"
   android:configure=
      "com.example.android.widgettest.MyAppWidgetConfigureActivity">
</appwidget-provider>
```

The following sections describe key attributes of the file. See the `AppWidgetProviderInfo` class for more information on the attributes for the `<appwidget-provider>` element.

## android:minHeight and android:minWidth attributes

The `android:minHeight` and `android:minWidth` attributes define the minimum initial size of the widget, in dp. Android Studio provides these values in the provider-info file based on the number of grid spaces you specify when you create the widget.

The Android home screen provides a grid of available spaces (cells) into which users can place widgets and icons. The available cells in the grid vary by device—phones may only allow a 4x4 grid, but tablets can offer a larger grid. When your widget is added to a user's

home screen, it is stretched both horizontally and vertically to occupy as many grid cells as satisfy the `minWidth` and `minHeight` values.

The rule for how many dp fit into a grid cell is based on the equation 70 × *grid_size* − 30, where *grid_size* is the number of cells you want your widget to take up. Generally speaking, you can use this table to determine what your `minWidth` and `minHeight` should be:

| # of columns or rows | `minWidth` or `minHeight` |
|---|---|
| 1 | 40 dp |
| 2 | 110 dp |
| 3 | 180 dp |
| 4 | 250 dp |

## android:initialLayout attribute

The `android:initialLayout` attribute defines the XML layout file for your widget, for example `@layout/my_app_widget`. You will learn more about widget layout in the next section.

## android:updatePeriodMillis attribute

The `android:updatePeriodMillis` attribute defines update interval in milliseconds for the app widget, that is, how often the app widget manager should send a broadcast intent to update the widget. The default update interval is 86,400,000 milliseconds, or 24 hours. The minimum possible update interval is 1,800,000 milliseconds or 30 minutes. Because widget updates use device resources and may even awaken a sleeping device, design your widget to update as infrequently as possible (use a larger value for `android:updatePeriodMillis`).

To disable automatic updates, set `android:updatePeriodMillis` to 0.

**Note:** The `android:updatePeriodMillis` value determines how often the app widget manager requests an update for an app widget. A running app can update its associated widget at any time.

## android:previewImage attribute

The `android:previewImage` attribute specifies an image or drawable that previews what the app widget will look like when it has been configured and placed on the user's home screen. The user sees the preview image in the widget picker when they choose an app widget.

The widget preview can be any image or drawable in the `res/drawable` folder. When you create a new widget, Android Studio provides an "example" widget preview image for you ( `@drawable/new_appwidget_preview` )


Replace this sample preview with any image or drawable that better represents your app widget. If you do not include the `android:previewImage` line in the provider-info file, the launcher icon for your app is used instead.

## android:resizeMode attribute

If your app widget is resizeable, the `android:resizeMode` attribute specifies whether your widget can be resized horizontally, vertically, both, or neither. The possible values are `"horizontal"` , `"vertical"` , `"horizontal|vertical"` , and `"none"` .

Users touch & hold a widget to show the widget's resize handles, then drag the handles to change the size of the widget on the home screen grid.

## android:widgetCategory attribute

The `android:widgetCategory` attribute declares whether your app widget can be displayed on the home screen ( `"home_screen"` ), the lock screen ( `"keyguard"` ), or both. Android Studio provides this attribute based on the value you choose for the **Placement** drop-down menu when you create the widget. For Android 5.0 and higher, only `home_screen` is valid. Keyguard (lock screen) widgets were only available in Android 4.2 (API 17) to Android 4.4.4 (API 19.0).

## android:configure attribute

The `android:configure` attribute defines the app widget's configuration activity, if you selected the **Configuration Screen** checkbox in the **Add Android Component** dialog. The configuration activity is specified in this attribute with its fully qualified class name. You will learn more about configuration activities later in this chapter.

# Defining the widget layout

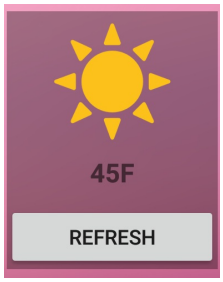
You can create app widget layouts in the Android Studio **Design** tab, the same way you create activity layouts. However, app widget layouts are based on `RemoteViews` object hierarchies, not `View` object hierarchies. Remote views do not support every kind of layout or view that standard views support, and remote views do not support custom views.

How are remote views different from regular views? A remote view is a view hierarchy that can be displayed in a process different from the original app but with the same permissions. Widgets and custom notifications are the primary examples for this situation.

Each time your widget updates, you create a `RemoteViews` object from the layout, update the views in the layout with data, and then pass that remote view to the app widget manager to display.

Because app widgets are typically small in size and display limited information, you should design your widgets with simple layouts. The Widgets and App Widget Design Guidelines pages provide guidelines and suggestions for good app widget designs.

Here's a simple example layout for a 1x2 weather widget (one cell wide and two cells high). This app widget shows an image, a text view, and a button, in a linear layout.



```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:background="@color/colorPrimary"
   android:orientation="vertical"
   android:padding="@dimen/widget_margin">

   <ImageView
       android:id="@+id/imageView"
       android:layout_width="wrap_content"
       android:layout_height="0dp"
       android:paddingTop="8dp"
       android:layout_gravity="center"
       android:layout_weight="2"
       android:src="@drawable/sun" />

   <TextView
       android:id="@+id/textView"
       android:layout_width="match_parent"
       android:layout_height="0dp"
       android:layout_weight="1"
       android:gravity="center"
       android:text="45F"
       android:textSize="18dp"
       android:textStyle="bold" />

   <Button
       android:id="@+id/actionButton"
       android:layout_width="match_parent"
       android:layout_height="0dp"
       android:layout_weight="1"
       android:text="Refresh" />
</LinearLayout>
```

## Supported views

A `RemoteViews` object (and, consequently, an app widget) can support the following layout classes:

- `FrameLayout`
- `LinearLayout`
- `RelativeLayout`
- `GridLayout`

And the following view classes:

- `AnalogClock`
- `Button`
- `Chronometer`

- `ImageButton`
- `ImageView`
- `ProgressBar`
- `TextView`
- `ViewFlipper`
- `ListView`
- `GridView`
- `StackView`
- `AdapterViewFlipper`

Descendants of these classes are not supported. The `RemoteViews` class also supports `ViewStub`, which is an invisible, zero-sized `View`. You can use `ViewStub` to lazily inflate layout resources at runtime.

## Widget margins

App widgets look best with a little extra space around the edges so that the widgets do not display edge-to-edge on the user's home screen. Before Android 4.0 (API 14), you had to add that extra space to your widget layout yourself. After API 14 the system adds the space for you. If your app targets a version of Android higher than API 14, do not add space to the edges of your app widgets.

When you use Android Studio to create your app widget, Android Studio automatically creates a layout and dimension resources that support both older (pre-API 14) and newer Android versions. Specifically, the layout template that Android Studio creates includes an `android:padding` attribute:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:background="#09C"
   android:padding="@dimen/widget_margin">
...
</RelativeLayout>
```

The `widget_margin` value is defined in the default `dimens.xml` file ( `res/values/dimens.xml` ):

```
<dimen name="widget_margin">8dp</dimen>
```

When Android Studio creates your widget it also adds a `dimens.xml (v14)` ( `res/values-v14/dimens.xml` ) file, which removes the extra space for API 14 and higher versions:

```
<dimen name="widget_margin">0dp</dimen>
```

You do not need to do anything additional to support widget padding.

# Implementing the widget-provider class

The widget-provider class implements the widget's behavior such as updating the widget's data at regular intervals or providing on-click behavior for the widget itself or any of its components.

Widget providers are subclasses of the `AppWidgetProvider` class. The `AppWidgetProvider` class extends `BroadcastReceiver` as a convenience class to receive and handle broadcasts specific to app widgets, such as when the app widget is updated, deleted, enabled, and disabled.

To create a widget provider:

1. Extend `AppWidgetProvider` for your own class.
2. Override the `onUpdate()` method, at minimum, to construct the widget layout and manage widget updates.
3. Optionally, implement additional widget actions.
4. Declare the widget provider as a broadcast receiver in the app manifest ( `AndroidManifest.xml` ).

When you create a new widget, Android Studio provides a template widget-provider class for you that does most of this implementation for you.

## Create a widget-provider class

This is a skeleton widget-provider class:

```
public class NewAppWidget extends AppWidgetProvider {
   // Standard onUpdate() method override
   @Override
   public void onUpdate(Context context,
      AppWidgetManager appWidgetManager, int[] appWidgetIds) {

      // There may be multiple widgets active, so update all of them
      for (int appWidgetId : appWidgetIds) {
         // Update the app widget
       }
   }
}
```

In addition to `onUpdate()`, the `AppWidgetProvider` class includes other handler methods for different events including `onDeleted()`, `onEnabled()`, and `onDisabled()`. See `AppWidgetProvider` for more details on these methods.

## Declare the widget provider in the Android manifest

The app widget-provider class you create is a broadcast receiver, and it must be declared in the Android manifest the same way as any other receiver. Android Studio adds a `<receiver>` element to your manifest when you add a new widget:

```
<receiver android:name="NewAppWidgetProvider" >
    <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
    </intent-filter>
    <meta-data android:name="android.appwidget.provider"
               android:resource="@xml/example_appwidget_info" />
</receiver>
```

The key parts of this element are as follows:

- The `name` attribute in `<receiver>` contains the class name for the widget provider.
- The `<intent-filter>` element declares an action of `APPWIDGET_UPDATE`. This element indicates that the receiver can receive broadcast intents from the app widget manager to update the widget.
- The `<meta-data>` element specifies the location of the widget provider-info file.

## Implement widget updates

For your widget to receive updated data, implement the `onUpdate()` method in your provider class. The `onUpdate()` method is called when the user adds the app widget, and every time the widget receives an update broadcast intent from the app widget manager or from your associated app. Perform all essential widget setup in the `onUpdate()` method, such as creating the widget layout, defining event handlers for views, or starting services.

**Note:** If your app widget includes a configuration activity, the `onUpdate()` method is *not* called when the user adds the app widget, but it *is* called for all subsequent updates. Your configuration activity must request the first update when configuration is complete. See Using a configuration activity for details.
Here is a simple `onUpdate()` method:

```
@Override
public void onUpdate(Context context,
      AppWidgetManager appWidgetManager, int[] appWidgetIds) {

   // There may be multiple widgets active, so update all of them
   for (int appWidgetId : appWidgetIds) {
     // Construct the RemoteViews object
     RemoteViews views = new RemoteViews(
         context.getPackageName(), R.layout.new_app_widget2);
     // Update a text view to display the app widget ID
     views.setTextViewText(R.id.appwidget_id,
       String.format("%s",appWidgetId));
     // Instruct the widget manager to update the widget
     appWidgetManager.updateAppWidget(appWidgetId, views);
}
```

The core of the `onUpdate()` method is a loop that iterates over an array of widget IDs. All instances of your widget that are currently active are identified by an internal ID. When `onUpdate()` is called, it is passed and array of IDs for the widgets that need updating. Your widget may have multiple instances active for several reasons, including most typically one widget with different configurations. For example, multiple weather widgets may be active to display the current weather in different locations. Make sure your `onUpdate()` method loops through all the widgets by ID and updates each widget individually.

In each update, you also need to reconstruct the widget's layout by creating a new `RemoteViews` object and updating any data that the remote view contains. The last line in the loop tells the app widget manager to update the widget with the current `RemoteViews` object.

## Provide app widget actions

Some app widgets only display information. Other widgets perform actions when tapped, such as launching the associated app. You can create click-event handlers to perform actions for the widget as a whole, or for any view of the widget layout such as a button.

Unlike activities where the click-event handlers run methods in your activity code, you attach actions to your widget with pending intents. The `setOnClickPendingIntent ()` method (defined in the `RemoteViews` class) enables you to connect a `PendingIntent` object to one or more views in your widget. The Android system delivers those intents to your app (or to any other app).

Add click-event handlers to your `onUpdate()` method. For example, this code launches the widget's associated app.

```
public void onUpdate(Context context,
   AppWidgetManager appWidgetManager, int[] appWidgetIds) {
   ...
   // Create a new explicit intent object
   Intent intent = new Intent(context, MainActivity.class);
   // Wrap that intent in a pending intent that starts a new activity
   PendingIntent configPendingIntent =
      PendingIntent.getActivity(context, 0, intent, 0);
   // Attach the pending intent to a view in the layout file
   // for the widget (here, appwidget_layout is the entire widget)
   views.setOnClickPendingIntent(
      R.id.appwidget_layout, configPendingIntent);
   ...
}
```

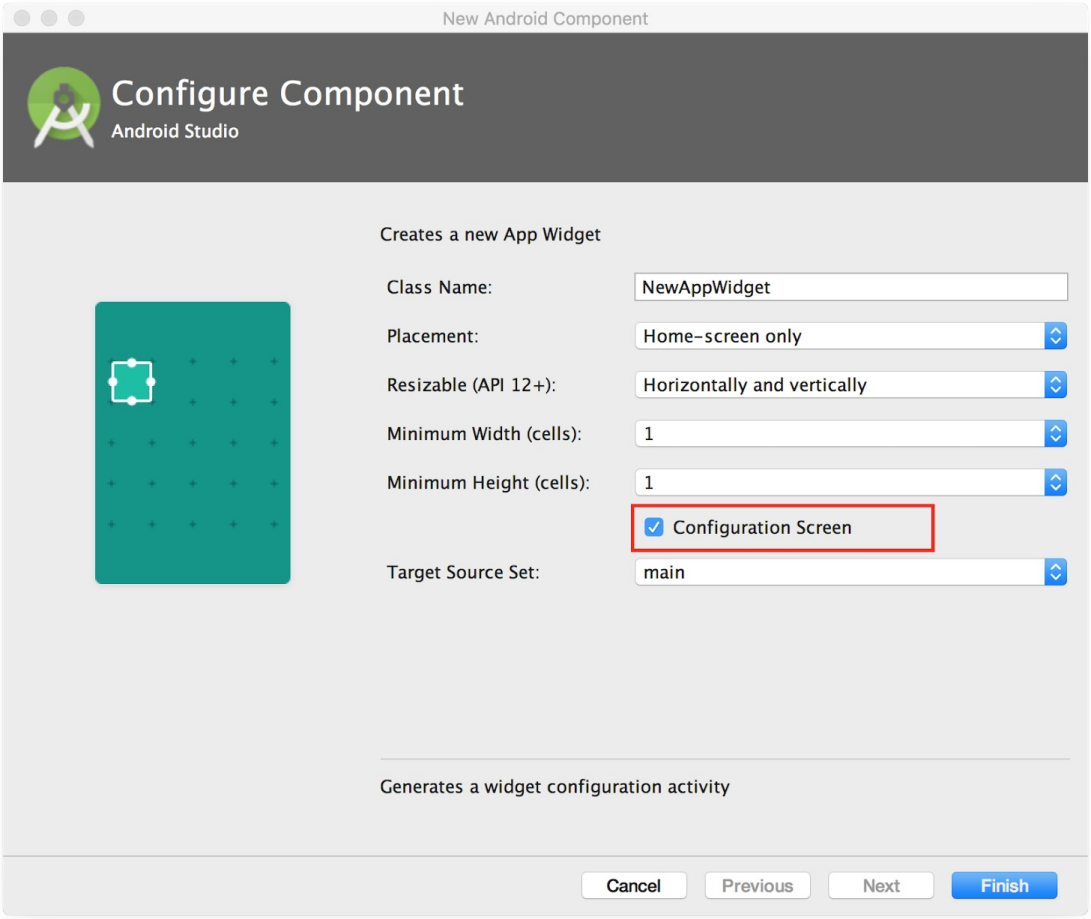
# Using a configuration activity

For some widgets it makes sense to let the user configure the settings for a new widget when they add that widget. For example, a stock-ticker widget needs the stock symbol that the user is interested in.

To provide widget configuration, you can add an optional widget configuration activity to your app's project. The configuration activity is launched when the user first adds your widget onto the home screen, and typically never appears again. (If the user needs to change the configuration of a widget, they have to delete the current widget and add a new one.)

## Add a configuration activity to your project

To add the widget configuration activity to your project when you initially create the widget in Android Studio, select the **Configuration Screen** checkbox.




You can also create a configuration activity manually, but adding it with the Android Studio wizard provides a lot of template code that can save you a lot of work.

The configuration activity is declared in the Android manifest file, with an intent-filter that accepts the `ACTION_APPWIDGET_CONFIGURE` action. For example:

```
<activity android:name=".ExampleAppWidgetConfigure">
    <intent-filter>
        <action
           android:name=
           "android.appwidget.action.APPWIDGET_CONFIGURE"/>
    </intent-filter>
</activity>
```

The configuration activity must also be declared in the provider-info XML file, using the `android:configure` attribute. (See Updating the provider info, above.) For example:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:configure="com.example.android.ExampleAppWidgetConfigure"
    ... >
</appwidget-provider>
```

Both of these declarations (in the manifest and in the provider-info file) are done for you automatically when you add a new widget using the Android Studio wizard.

## Implement the configuration activity

Implement your widget's configuration activity the same way you implement any other activity. If you created the configuration activity when you used the wizard to add a widget to your project, the generated configuration activity class can serve as a starting point.

It is the activity's responsibility to update the app widget when configuration is complete. Here's a summary of the procedure to properly update the app widget and close the configuration activity.

1. In your `onCreate()` method, get the widget ID from the intent that launched the activity. You'll need it to update the widget when configuration is complete. The app widget ID is in the incoming intent `extras`, with the key `AppWidgetManager.EXTRA_APPWIDGET_ID`.

   ```
   Intent intent = getIntent();
   Bundle extras = intent.getExtras();
   if (extras != null) {
       mAppWidgetId = extras.getInt(
          AppWidgetManager.EXTRA_APPWIDGET_ID,
          AppWidgetManager.INVALID_APPWIDGET_ID);
   }
   ```

2. Also in `onCreate()`, set the default activity result to `RESULT_CANCELED`. This covers the case where the user backs out of widget configuration before it is complete. A canceled result alerts the widget host (the Android home screen) that the widget should not be added.

   ```
   // Set the result to CANCELED. This causes the widget host to cancel
   // the widget placement if the user presses the back button.
   setResult(RESULT_CANCELED);
   ```

3. Get the widget configuration data from the user as needed, and store that data so the widget can access it later. For example, store the data in shared preferences.
4. The method that closes the activity is often a click-event handler for a button that the user clicks when the app widget configuration is complete. In that method, request an update for the widget. Here is the code to do that:

```
// Get an instance of the AppWidgetManager:
AppWidgetManager appWidgetManager = AppWidgetManager.getInstance(this);
// Create a new RemoteViews object with the widget layout resource
RemoteViews views = new RemoteViews(context.getPackageName(),
   R.layout.example_appwidget);
// Update the app widget with the widget ID and the new remote view:
appWidgetManager.updateAppWidget(mAppWidgetId, views);
```

5. To finish and close the activity, create a new intent and add the original app widget ID to the intent `extras` with the key `AppWidgetManager.EXTRA_APPWIDGET_ID`. Set the result to `RESULT_OK` and call the `finish()` method:

```
Intent resultValue = new Intent();
resultValue.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
   mAppWidgetId);
setResult(RESULT_OK, resultValue);
finish();
```

# Related practical

The related practical documentation is in Building app widgets.

# Learn more

Android developer documentation:

- App Widgets
- App widget design guidelines
- Determining a size for your widget

Android API Reference:

- `AppWidgetProvider` class
- `AppWidgetProviderInfo` class
- `AppWidgetManager` class
- `BroadcastReceiver` class
- `RemoteViews` class

# 3.1: Sensor basics

**Contents:**



Most Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy, and are useful if you want to monitor three-dimensional device movement or positioning, or you want to monitor changes in the ambient environment near a device.

For example, a game might track readings from a device's gravity sensor to infer complex user gestures and motions, such as tilt, shake, rotation, or swing. Likewise, a compass app might use the geomagnetic field sensor and accelerometer to report a compass bearing.

## About sensors

The Android platform supports three major categories of sensors:

- Motion sensors, to measure device motion. These sensors include accelerometers, gravity sensors, gyroscopes, and rotational vector sensors.
- Environmental sensors, to measure various environmental conditions, such as ambient air temperature and pressure, illumination, and humidity. These sensors include barometers, photometers (light sensors), and thermometers.
- Position sensors, to measure the physical position of a device. This category includes magnetometers (geomagnetic field sensors) and proximity sensors.

The device camera, fingerprint sensor, microphone, and GPS (location) sensor all have their own APIs and are not considered "sensors" for the purposes of the Android sensor framework. You learn more about location in a different chapter.

### Hardware and software sensors

Sensors can be hardware- or software-based. Hardware-based sensors are physical components built into a handset or tablet device. Hardware sensors derive their data by directly measuring specific environmental properties, such as acceleration, geomagnetic field strength, or angular change.

Software-based sensors are not physical devices, although they mimic hardware-based sensors. Software-based sensors derive their data from one or more of the hardware-based sensors and are sometimes called *virtual sensors* or *composite sensors* . The proximity sensor and step counter sensors are examples of software-based sensors.

## The Android sensor framework

You can access available sensors and acquire raw sensor data in your app with the Android sensor framework. The sensor framework, part of the `android.hardware` package, provides classes and interfaces to help you perform a wide variety of sensor-related tasks. With the Android sensor framework you can:

- Determine which sensors are available on a device.
- Determine an individual sensor's capabilities, such as its maximum range, manufacturer, power requirements, and resolution.
- Acquire raw sensor data and define the minimum rate at which you acquire sensor data.
- Register and unregister sensor event listeners that monitor sensor changes.

The following classes are the key parts of the Android sensor framework:

- `SensorManager` : Represents the Android sensor service. This class provides methods for accessing and listing sensors, registering and unregistering sensor event listeners, and acquiring orientation information. This class also provides several sensor constants. The constants are used to represent sensor accuracy, data acquisition rates, and sensor calibration.
- `Sensor` : Represents a specific sensor. This class provides methods that let you determine sensor's capabilities.
- `SensorEvent` : Represents information about a sensor event. A sensor event includes the raw sensor data, the type of sensor that generated the event, the accuracy of the data, and the timestamp for the event.
- `SensorEventListener` : An interface that includes callback methods to receive notifications (sensor events) when a sensor has new data or when sensor accuracy changes.

## Sensor types and availability

Few Android-powered devices have every type of sensor. For example, most handset devices and tablets have an accelerometer and a magnetometer, but many fewer devices have barometers or thermometers. Also, a device can have more than one sensor of a given type. For example, a device can have two gravity sensors, each one having a different range. The `Sensor` class defines the sensor types listed in the table below.

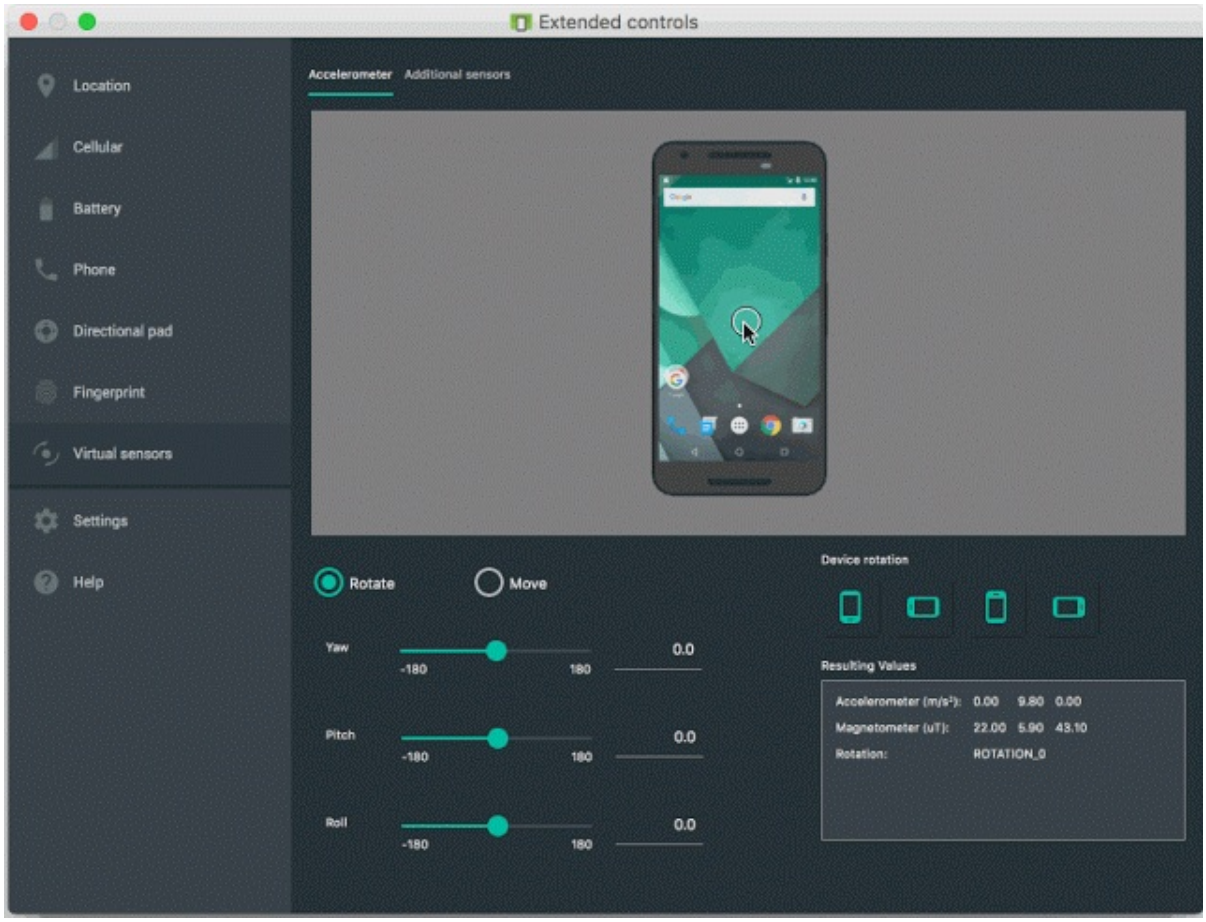
Sensor availability can also vary between Android versions, because the Android sensors have been introduced over the course of several platform releases.

The following table summarizes the sensors that the Android platform supports.

| Sensor | Used for |
|---|---|
| TYPE_ACCELEROMETER | Motion detection (shake, tilt, and so on). |
| TYPE_AMBIENT_TEMPERATURE | Monitoring air temperature. |
| TYPE_GRAVITY | Motion detection (shake, tilt, and so on). |
| TYPE_GYROSCOPE | Rotation detection (spin, turn, and so on). |
| TYPE_LIGHT | Controlling screen brightness. |
| TYPE_LINEAR_ACCELERATION | Monitoring acceleration along a single axis. |
| TYPE_MAGNETIC_FIELD | Creating a compass. |
| TYPE_ORIENTATION | Determining device position. |
| TYPE_PRESSURE | Monitoring air pressure changes. |
| TYPE_PROXIMITY | Phone position during a call. |
| TYPE_RELATIVE_HUMIDITY | Monitoring ambient humidity (relative and absolute), and dew point. |
| TYPE_ROTATION_VECTOR | Motion and rotation detection. |
| TYPE_TEMPERATURE | Monitoring temperatures. |

## Sensors and the Android emulator

The Android emulator includes a set of virtual sensor controls that let you to test sensors such as the accelerometer, the ambient temperature sensor, the magnetometer, the proximity sensor, the light sensor, and more.




The **Accelerometer** tab lets you test your app against changes in device position, orientation, or both. For example, you can simulate device motion such as tilt and rotation. The control simulates the way accelerometers and magnetometers respond when you move or rotate a real device. As you adjust the device in the emulators, the **Resulting Values** fields change accordingly. These fields show the values that an app can access.

The **Additional sensors** tab can simulate various position and environment sensors. In this tab you adjust the following sensors to test them with your app:

- Ambient temperature: This environmental sensor measures ambient air temperature.
- Magnetic field: This position sensor measures the ambient magnetic field at the *x* -axis, *y* -axis, and *z* -axis. The values are in microtesla (μT).
- Proximity: This position sensor measures the distance of the device from an object. For example, this sensor can notify a phone that a face is close to the phone to make a call.
- Light: This environmental sensor measures illuminance.
- Pressure: This environmental sensor measures ambient air pressure.
- Relative humidity: This environmental sensor measures ambient relative humidity.

# Discovering sensors and sensor capabilities

The Android sensor framework lets you determine at runtime which sensors are available on a device. The framework also provides methods that let you determine the capabilities of a sensor, such as its maximum range, its resolution, and its power requirements.

## Identifying sensors

To identify the device sensors you must first access the sensor manager, an Android system service. Create an instance of the `SensorManager` class by calling the `getSystemService()` method and passing in the `SENSOR_SERVICE` argument.

```
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
```

To get a listing of all the device sensors, use the `getSensorList()` method and the sensor type `TYPE_ALL`.

```
List<Sensor> deviceSensors = mSensorManager.getSensorList(Sensor.TYPE_ALL);
```

To list all of the sensors of a specific type, use another sensor type constant, for example `TYPE_PROXIMITY`, `TYPE_GYROSCOPE`, or `TYPE_GRAVITY`.

To determine whether a specific type of sensor exists on a device, and to get a `Sensor` object that represents that sensor, use the `getDefaultSensor()` method and pass in the type constant for a specific sensor. If a device has more than one sensor of a given type, the system designates one of the sensors as the default sensor. If a default sensor does not exist for a given type of sensor, the method call returns `null`, which means the device does not have that type of sensor.

For example, the following code checks whether there's a magnetometer on a device:

```
private SensorManager mSensorManager;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
if (mSensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD) != null){
  // Success! There's a magnetometer.
  }
else {
  // Failure! No magnetometer.
}
```

## Identifying sensor features

Use the public methods of the `Sensor` class to determine the capabilities and attributes of individual sensors. These methods useful if you want your app to behave differently based on which sensors or sensor capabilities are available on a device. For example, you can use

the `getResolution()` and `getMaximumRange()` methods to obtain a sensor's reported resolution and maximum range of measurement. You can also use the `getPower()` method to obtain a sensor's power requirements.

The `getVendor()` and `getVersion()` methods are particularly useful. Use these methods to optimize your app for different manufacturer's sensors or different versions of a sensor. For example, maybe your app needs to monitor device motion such as tilt and shake. In this case you could create two sets of data-filtering rules and optimizations: one set of rules and optimizations for newer devices that have a specific vendor's gravity sensor, and a second set of rules and optimizations for devices that do not have a gravity sensor and have only an accelerometer.

A *streaming sensor* is one that senses and reports new data as fast as possible. The `getMinDelay()` method returns the minimum time interval (in microseconds) that a sensor can use to sense data. Any sensor that returns a non-zero value for the `getMinDelay()` method is a streaming sensor. Streaming sensors sense data at regular intervals and were introduced in Android 2.3 (API Level 9). If a sensor returns zero when you call `getMinDelay()`, it means that the sensor is not a streaming sensor. Non-streaming sensors only report data when that data has changed.

The `getMinDelay()` method lets you determine the maximum rate at which a sensor can acquire data. If your app requires high data-acquisition rates or a streaming sensor, use `getMinDelay()` to determine whether a sensor meets your app's requirements. Then turn on or turn off the relevant features in your app accordingly.

**Note:** A sensor's maximum data acquisition rate is not necessarily the rate at which the sensor framework delivers sensor data to your app. The sensor framework reports data through sensor events, and several factors influence the rate at which your app receives sensor events.

# Handling different sensor configurations

Android does not specify a standard sensor configuration for devices, which means device manufacturers can incorporate any sensor configuration that they want into their Android-powered devices. As a result, devices include a variety of sensors in a wide range of configurations. If your app relies on a specific type of sensor, you have to ensure that the sensor is present on a device so your app can run successfully.

You have two options for ensuring that a given sensor is present on a device:

- Detect sensors at runtime, then turn on or turn off app features as appropriate.
- Use Google Play filters to target devices with specific sensor configurations.

## Detecting sensors at runtime

If your app uses a specific type of sensor but doesn't rely on it, you can use the sensor framework to detect the sensor at runtime, then turn off or turn on app features as appropriate. For example, a weather app might use the temperature sensor, pressure sensor, GPS sensor, and geomagnetic field sensor to display the temperature, barometric pressure, location, and compass bearing. You can use the sensor framework to detect the absence of the pressure sensor at runtime and then turn off the portion of your app's UI that displays pressure.

## Using Google Play filters to target specific sensor configurations

If you publish your app on Google Play, use the `<uses-feature>` element in your app manifest file to filter your app from devices that do not have the appropriate sensor configuration for your app. The `<uses-feature>` element has several hardware descriptors that let you filter apps based on the presence of specific sensors. The sensors you can list include: accelerometer, barometer, compass (geomagnetic field), gyroscope, light, and proximity.

The following is an example manifest entry that filters out apps that do not have an accelerometer:

```
<uses-feature android:name="android.hardware.sensor.accelerometer"
   android:required="true"
/>
```

If you add this element and descriptor to your app's manifest, users see your app on Google Play only if their device has an accelerometer.

About the `android:required` attribute:

- Set the descriptor to `android:required="true"` only if your app relies entirely on a specific sensor.
- If your app uses a sensor for some functionality but can run without the sensor, list the sensor in the `<uses-feature>` element and set the descriptor to `android:required="false"`. Doing this helps ensure that devices can install your app even if they do not have that particular sensor.

# Monitoring sensor events

The Android system generates sensor events each time the sensor has new data. To monitor sensor events in your app, you must:

- Implement the `SensorEventListener` interface, which includes the `onSensorChanged()` and `onAccuracyChanged()` callback methods.
- Register sensor event listeners for the specific sensor you want to monitor.
- Get sensor types and values from the `SensorEvent` object, and update your app accordingly.

## Implement the SensorEventListener interface

To monitor raw sensor data, implement the `SensorEventListener` interface's two callback methods: `onAccuracyChanged()` and `onSensorChanged()`.

```
public class SensorActivity extends Activity implements SensorEventListener { ... }
```

When a sensor's accuracy changes, the Android system calls `onAccuracyChanged()` method and passes in two arguments:

- A `Sensor` object to identify the sensor that changed.
- A new accuracy, one of five status constants: `SENSOR_STATUS_ACCURACY_LOW`, `SENSOR_STATUS_ACCURACY_MEDIUM`, `SENSOR_STATUS_ACCURACY_HIGH`, `SENSOR_STATUS_UNRELIABLE`, or `SENSOR_STATUS_NO_CONTACT`.

An example of a change in sensor accuracy might be a heart-rate monitor. If the sensor stopped reporting a heartbeat, the accuracy would be `SENSOR_STATUS_NO_CONTACT`. If the sensor was reporting an unrealistically low or high heartbeat, the accuracy might be `SENSOR_STATUS_UNRELIABLE`. You could override `onAccuracyChanged()` to catch these changes in accuracy and report errors to the user.

```
@Override
public final void onAccuracyChanged(Sensor sensor, int accuracy) {
    // Do something here if the sensor accuracy changes.
}
```

The Android system calls the `onSensorChanged()` method when the sensor reports new data, passing in a `SensorEvent` object. The `SensorEvent` object contains information about the new sensor data, including the accuracy of the data, the sensor that generated the data, the timestamp at which the data was generated, as well as the actual new data itself.

```
@Override
public void onSensorChanged(SensorEvent sensorEvent) {
        // Do something here if sensor data changes
}
```

You learn more about the `onSensorChanged()` method and sensor event objects below.

## Register listeners

To receive sensor events, you must register a listener for that event in your app. Sensors generate a lot of data at a very high rate, and they use power and battery when they do. Registering a particular sensor enables your app to handle only specific events at a specific rate. To preserve device resources, unregister the listener when you don't need it, for example when your app pauses.

To register a listener, use the `SensorManager.registerListener()` method. This method takes three arguments:

- An app or activity `Context`. You can use the current activity (`this`) as the context.
- The `Sensor` object to listen to. Use the `SensorManager.getDefaultSensor()` method in your `onCreate()` to get a `Sensor` object.
- A delay constant from the `SensorManager` class. The data delay (or sampling rate) controls the interval at which sensor events are sent to your app via the `onSensorChanged()` callback method. Make sure that your listener is registered with the minimum amount of new data that it needs.

  The default data delay (`SENSOR_DELAY_NORMAL`) is suitable for monitoring typical screen orientation changes and uses a delay of 200,000 microseconds (0.2 seconds). You can specify other data delays, such as `SENSOR_DELAY_GAME` (20,000 microseconds or 20 milliseconds), `SENSOR_DELAY_UI` (60,000 microseconds or 60 milliseconds), or `SENSOR_DELAY_FASTEST` (no delay, as fast as possible).

Register your sensor listeners in the `onStart()` lifecycle method, and unregister them in `onStop()`. Don't register your listeners in `onCreate()`, as that would cause the sensors to be on and sending data (using device power) even when your app was not in the foreground. To make sure sensors only use power when your app is running in the foreground, register and unregister your sensor listeners in the `onStart()` and `onStop()` methods.

The `onStart()` and `onStop()` methods are a better choice for registering listeners than `onResume()` and `onPause()`. As of Android 7.0 (API 24), apps can run in multi-window mode (split-screen or picture-in-picture mode). Apps running in this mode are paused (`onPause()`

has been called) but still visible on screen. Use `onStart()` and `onStart()` to ensure that sensors continue running even if the app is in multi-window mode.

```
@Override
protected void onStart() {
   super.onStart();

   if (mIsLightSensorPresent) {
       mSensorManager.registerListener(this, mSensorLight,
          SensorManager.SENSOR_DELAY_UI);
   }
}

@Override
protected void onStop() {
   super.onStop();
   mSensorManager.unregisterListener(this);
}
```

## Handle sensor data changes

To handle changes to sensor data, override the `onSensorChanged()` callback from the `SensorEventListener` class. The `onSensorChanged()` method is passed a `SensorEvent` object that contains information about the event. Your implementation typically will need two pieces of data from the `SensorEvent` object:

- `sensor` : The sensor that generated the event, as a `Sensor` object.
- `values` : The data the sensor generated, as an array of floats. Different sensors provide different amounts and types of data in that array. For example, the light sensor produces only one piece of data, which is stored in `values[0]` . The values for other sensors may have additional data—for example, the values array from the accelerometer includes data for the *x* -axis, *y* -axis, and *z* -axis in `values[0]` , `values[1]` , and `values[2]` , respectively.

**Note:** Sensor data can change at a high rate, which means the system may call the `onSensorChanged()` method quite often. As a best practice, do as little as possible within the `onSensorChanged()` method so you don't block it. If your app requires you to do any data filtering or reduction of sensor data, perform that work outside of the `onSensorChanged()` method.
This example shows an `onSensorChanged()` method that handles changes to the light sensor.

```
@Override
public void onSensorChanged(SensorEvent sensorEvent) {
   // The sensor type (as defined in the Sensor class).
   int sensorType = sensorEvent.sensor.getType();

   // The current value of the sensor.
   float currentValue = sensorEvent.values[0];

   // Event came from the light sensor.
   if (sensorType == Sensor.TYPE_LIGHT)
   {
       // Get the light sensor string from the resources, fill
       // in the data placeholder
       String str_current = getResources().getString(
          R.string.label_light, currentValue);
       mTextSensorLight.setText(str_current);
   }
}
```

# Related practical

The related practical documentation is in Working with Sensor Data.

# Learn more

Android developer documentation:

- Sensors Overview
- Motion Sensors
- Position Sensors

Android API Reference:

- `Sensor`
- `SensorEvent`
- `SensorManager`
- `SensorEventListener`

# 3.2: Motion and position sensors

**Contents:**



You can use motion and position sensors to monitor a device's movement or its position in space.

Use the classes and methods from the Android sensor framework to gain access to the motion and position sensors and to handle changes to sensor data.

All the motion sensors return multi-dimensional arrays of sensor values for each `SensorEvent`. For example, during a single sensor event the accelerometer returns acceleration force data for the three coordinate axes ( *x* , *y* , *z* ) relative to the device.

In this chapter you learn more about the motion and position sensors and how to manage the data they produce. In a later part of this chapter, you learn about the most common motion and position sensors.

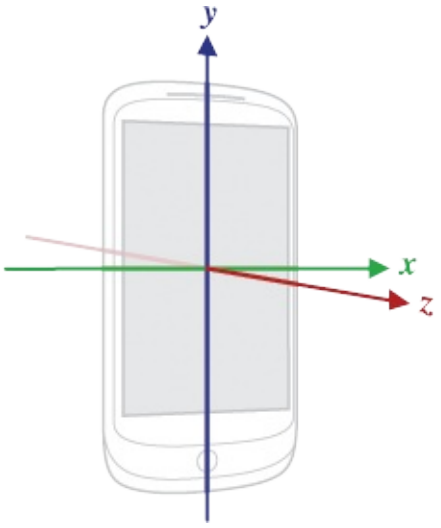
# Coordinate systems

The motion and position sensors in Android typically use two different coordinate systems: a device coordinate system relative to the device, and a coordinate system relative to the surface of the Earth. Both systems use a standard 3-axis system ( *x* , *y* , *z* ). In addition, some sensors and methods in the Android sensor framework provide their data as angles around the three axes.

## Device coordinates

Most Android sensors, including the accelerometer and gyroscope, use a standard 3-axis coordinate system defined relative to the device's screen when the device is held in its default orientation (portrait orientation, for a phone). In the standard 3-axis coordinate

system, the *x* -axis is horizontal and points to the right. The *y* -axis is vertical and points up, and the *z* -axis points toward the outside of the screen face. In this system, coordinates behind the screen have negative *z* values.




The most important point to understand about this coordinate system is that, unlike activity rotation, the axes are not swapped when the device's screen orientation changes—that is, the sensor's coordinate system never changes as the device rotates. This behavior is the same as the behavior of the OpenGL coordinate system.

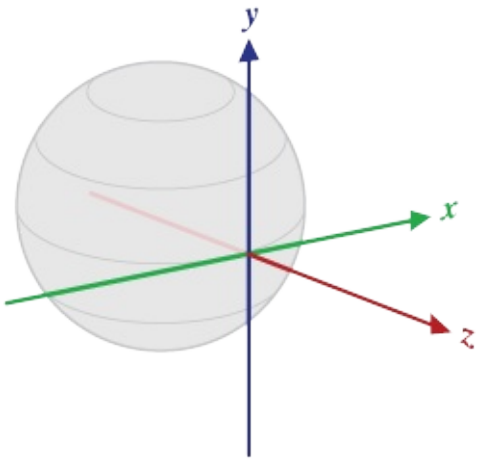
If your app uses sensor data to position views or other elements on the screen, you need to transform the incoming sensor data to match the rotation of the device. See Device rotation for more information.

Your app must not assume that a device's natural (default) orientation is portrait. The natural orientation for many tablet devices is landscape. And the sensor coordinate system is always based on the natural orientation of a device.

## Earth's coordinates

Some sensors and methods use a coordinate system that represents device motion or position relative to the Earth. In this coordinate system:

- *y* points to magnetic north along the surface of the Earth.
- *x* is 90 degrees from *y* , pointing approximately east.
- *z* extends up into space. Negative *z* extends down into the ground.




# Determining device orientation

Device orientation is the position of the device in space relative to the Earth's coordinate system (specifically, to the magnetic north pole). You can use orientation for compass apps, or to determine the degree of tilt for the device for games.

Although early versions of Android included an explicit sensor type for orientation ( `Sensor.TYPE_ORIENTATION` ), this sensor type was deprecated in API 8 and may be entirely unavailable in current devices.

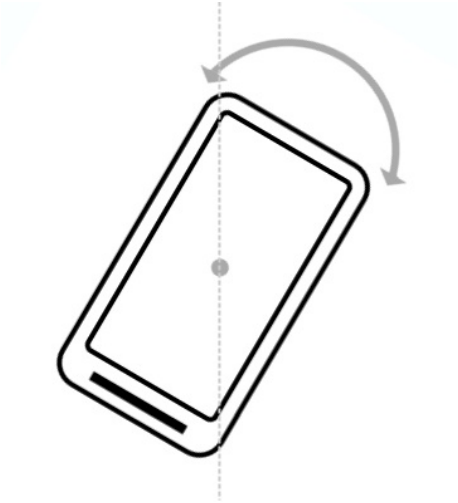
The recommended way to determine device orientation uses the accelerometer and geomagnetic field sensor and several methods in the `SensorManager` class. The following code shows you how to compute a device's orientation:

```
private SensorManager mSensorManager;
...
// Rotation matrix based on current readings from accelerometer and magnetometer.
final float[] rotationMatrix = new float[9];
mSensorManager.getRotationMatrix(rotationMatrix, null,
  accelerometerReading, magnetometerReading);

// Express the updated rotation matrix as three orientation angles.
final float[] orientationAngles = new float[3];
mSensorManager.getOrientation(rotationMatrix, orientationAngles);
```

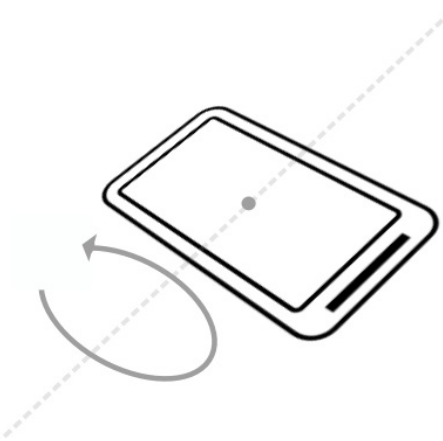
The `getRotationMatrix()` method generates a rotation matrix from the accelerometer and geomagnetic field sensor. A rotation matrix is a linear algebra concept that translates the sensor data from one coordinate system to another—in this case, from the device's coordinate system to the Earth's coordinate system.

The `getOrientation()` method uses the rotation matrix to compute the angles of the device's orientation. All of these angles are in radians, with values from -π to π. There are three components to orientation:

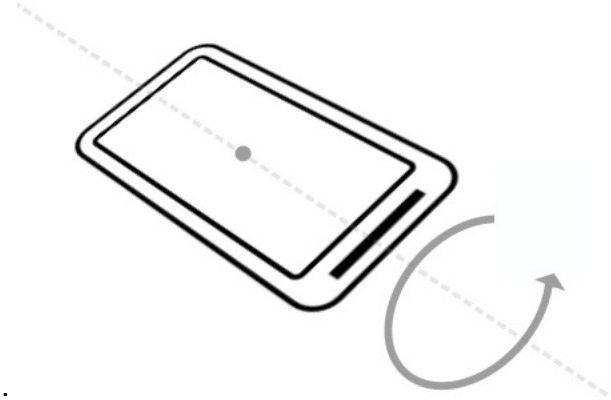
- *Azimuth* : The angle between the device's current compass direction and magnetic north. If the top edge of the device faces magnetic north, the azimuth is 0.



- *Pitch* : The angle between a plane parallel to the device's screen and a plane parallel to the ground.



- *Roll* : The angle between a plane perpendicular to the device's screen and a plane perpendicular to the ground.



**Note:** orientation angles use a different coordinate system than the one used in aviation (for yaw, pitch, and roll). In the aviation system, the *x* -axis is along the long side of the plane, from tail to nose.

# Understanding device rotation

If your app draws views on the screen in positions based on sensor data, you need to transform the sensor's coordinate system (which does not rotate) with the screen or activity's coordinate system (which does rotate).

To handle device and activity rotation in sensor-based drawing, query the current device orientation with the `getRotation()` method. Then remap the rotation matrix from the sensor data onto the desired axes with the `remapCoordinateSystem()` method.

The `getRotation()` method returns one of four integer constants:

- `ROTATION_0` : The default orientation of the device (portrait for phones).
- `ROTATION_90` : The "sideways" orientation of the device (landscape for phones). Different devices may report 90 degrees either clockwise or counterclockwise from 0.
- `ROTATION_180` : Upside-down orientation, if the device allows it.
- `ROTATION_270` : Sideways orientation, in the opposite direction from `ROTATION_90` .

Many devices do not have `ROTATION_180` at all. Many devices return `ROTATION_90` or `ROTATION_270` , regardless of whether the device was rotated clockwise or counterclockwise. It is best to handle all possible rotations rather than to make assumptions about any particular device.

The following sample code shows how to use `getRotation()` and `remapCoordinateSystem()` to get the rotation for the device.

```
// Compute the rotation matrix: merges and translates the data
// from the accelerometer and magnetometer, in the device coordinate
// system, into a matrix in Earth's coordinate system.
//
// The second argument to getRotationMatrix() is an
// inclination matrix, which isn't used in this example (and
// is thus null).
float[] rotationMatrix = new float[9];
boolean rotationOK = SensorManager.getRotationMatrix(rotationMatrix,
       null, mAccelerometerData, mMagnetometerData);

// Remap the matrix based on current device/activity rotation.
float[] rotationMatrixAdjusted = new float[9];
switch (mDisplay.getRotation()) {
   case Surface.ROTATION_0:
        rotationMatrixAdjusted = rotationMatrix.clone();
        break;
   case Surface.ROTATION_90:
        SensorManager.remapCoordinateSystem(rotationMatrix,
                SensorManager.AXIS_Y, SensorManager.AXIS_MINUS_X,
                rotationMatrixAdjusted);
        break;
   case Surface.ROTATION_180:
        SensorManager.remapCoordinateSystem(rotationMatrix,
                SensorManager.AXIS_MINUS_X, SensorManager.AXIS_MINUS_Y,
                rotationMatrixAdjusted);
        break;
   case Surface.ROTATION_270:
        SensorManager.remapCoordinateSystem(rotationMatrix,
                SensorManager.AXIS_MINUS_Y, SensorManager.AXIS_X,
                rotationMatrixAdjusted);
        break;
}
```

# Motion sensors

The Android platform provides several sensors that let you monitor device motion such as tilt, shake, rotation, or swing. The movement is usually a reflection of direct user input, for example a user steering a car in a game, or a user controlling a ball in a game. Movement can also be a reflection of the device's physical environment, for example the device moving with you while you drive your car.

- When movement is a reflection of direct user input, you are monitoring motion relative to the device's frame of reference, or your app's frame of reference.
- When movement is a reflection of the device's physical environment, you are monitoring motion relative to the Earth.

Motion sensors by themselves are not typically used to monitor device position, but they can be used with other sensors, such as the geomagnetic field sensor, to determine a device's position relative to the Earth's frame of reference.

This section describes many of the most common Android motion sensors. It is not a comprehensive list of all the motion sensors available in the Android sensor framework. See Motion Sensors and the `Senso r` class for more information.

## Accelerometer

The accelerometer ( `TYPE_ACCELEROMETER` ) measures the acceleration applied to the device along the three device axes ( *x* , *y* , *z* ), including the force of gravity. The inclusion of the force of gravity means that when the device lies flat on a table, the acceleration value along the *z* -axis is +9.81. This corresponds to the acceleration of the device (0 m/s^2) minus the force of gravity (-9.81 m/s^2). A device in freefall has a *z* value of 0, because the force of gravity is not in effect.

If you need only the acceleration forces without gravity, use the `TYPE_LINEAR_ACCELERATION` virtual sensor. If you need to determine the force of gravity without acceleration forces, use the `TYPE_GRAVITY` virtual sensor.

Nearly every Android-powered handset and tablet has an accelerometer in hardware. The accelerometer uses many times less power than other motion sensors. However, the raw accelerometer data is quite noisy and you may need to implement filters to eliminate gravitational forces and reduce noise. See this stack overflow question for more information on filtering accelerometer data.

The event data generated by the accelerometer is as shown in the following table:

| Event data | Description | Units |
|---|---|---|
| `SensorEvent.values[0]` | Acceleration force along the *x* -axis (including gravity). | m/s2 |
| `SensorEvent.values[1]` | Acceleration force along the *y* -axis (including gravity). | m/s2 |
| `SensorEvent.values[2]` | Acceleration force along the *z* -axis (including gravity). | m/s2 |

The following code shows you how to get an instance of the default linear acceleration sensor:

```
private SensorManager mSensorManager;
private Sensor mSensor;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_LINEAR_ACCELERATION);
```

## Gravity sensor

The gravity sensor ( `T Y PE_GRAVITY` ) measures the force of gravity along the three device-coordinate axes ( *x* , *y* , *z* ). Although gravity can be a hardware sensor, it is typically a virtual sensor based on data from the accelerometer. When a device is at rest, the output of the gravity sensor should be identical to that of the accelerometer.

The event data generated by the gravity sensor is as shown in the following table:

| Event data | Description | Units |
|---|---|---|
| SensorEvent.values[0] | Force of gravity along the *x* -axis. | m/s2 |
| SensorEvent.values[1] | Force of gravity along the *y* -axis. | m/s2 |
| SensorEvent.values[2] | Force of gravity along the *z* -axis. | m/s2 |

## Gyroscope

The gyroscope sensor ( `TYPE_GYROSCOPE` ) measures the rate of rotation around the three device axes ( *x* , *y* , *z* ). All values are in radians/second. Although you can use the gyroscope to determine the orientation of the device, it is generally better practice to use the accelerometer with other sensors such as the magnetometer.

Many current devices have a gyroscope sensor, although the gyroscope may be unavailable in older or lower-end devices.

The event data generated by the gyroscope sensor is as shown in the following table:

| Event data | Description | Units |
|---|---|---|
| `SensorEvent.values[0]` | Rate of rotation around the *x* -axis. | rad/s |
| `SensorEvent.values[1]` | Rate of rotation around the *y* -axis. | rad/s |
| `SensorEvent.values[2]` | Rate of rotation around the *z* -axis. | rad/s |

## Linear accelerator

The linear accelerator ( `TYPE_LINEAR_ACCELERATION` ) measures acceleration forces applied to the device along the three device axes ( *x* , *y* , *z* ), minus the force of gravity. Linear acceleration is usually a software sensor that gets its data from the accelerometer.

The event data generated by the linear acceleration sensor is as shown in the following table:

| Event data | Description | Units |
|---|---|---|
| `SensorEvent.values[0]` | Acceleration on the *x* -axis. | m/s2 |
| `SensorEvent.values[1]` | Acceleration on the *y* -axis. | m/s2 |
| `SensorEvent.values[2]` | Acceleration on the *z* -axis. | m/s2 |

## Rotation-vector sensor

The rotation-vector sensor ( `TYPE_ROTATION_VECTOR` ) provides the orientation of the device with respect to Earth's coordinate system as a unit quaternion. Specifically, the rotation vector represents the orientation of the device as a combination of an angle and an axis, in which the device has rotated through an angle θ around an axis < *x* , *y* , *z* >. The first four elements of the rotation vector are equal to the components of a unit quaternion x *sin(θ/2),* y *sin(θ/2), z* *sin(θ/2)>. The elements of the rotation vector are unitless.

The rotation-vector sensor is a software sensor that integrates data from the accelerometer, magnetometer, and gyroscope (if available). If you are comfortable with the math, the rotation vector is a more efficient and a more accurate way to determine device orientation than other methods.

The event data generated by the rotation-vector sensor is as shown in the following table:

| Event data | Description | Units |
|---|---|---|
| `SensorEvent.values[0]` | *x* *sin(θ/2) | none |
| `SensorEvent.values[1]` | *y* *sin(θ/2) | none |
| `SensorEvent.values[2]` | *z* *sin(θ/2) | none |
| `SensorEvent.values[3]` | cos(θ/2) (API 18 and higher only) | none |
| `SensorEvent.values[4]` | Estimated direction (heading) accuracy, -1 if unavailable (API 18 and higher only) | radians |

## Step counter and step detector

The step-counter sensor ( `TYPE_STEP_COUNTER` ) measures the number of steps taken by the user since the last reboot, while the sensor was registered and active. The step counter is a hardware sensor that has more latency (up to 10 seconds) but more accuracy than the step-detector sensor (see below). To preserve the battery on devices running your app, you should use the `JobScheduler` class to retrieve the current value from the step-counter sensor at a specific interval. Although different types of apps require different sensor-reading intervals, you should make this interval as long as possible unless your app requires real-time data from the sensor.

The event value for the step counter ( `value[0]` ) is a floating-point number with the fractional part set to zero. The value is reset to zero only on a system reboot. The timestamp of the event is set to the time when the last step for that event was taken.

The step-detector sensor ( `TYPE_STEP_DETECTOR` ) is a hardware sensor that triggers an event each time the user takes a step. Compared to the step counter, the step detector usually has a lower latency, less than 2 seconds.

The event value ( `value[0]` ) is always 1.0. The timestamp of the event corresponds to when the foot hits the ground.

See the BatchStepSample on GitHub for code that shows how to use the step-counter sensor and step-detector sensor.

# Position sensors

Position sensors are useful for determining a device's physical position in the Earth's frame of reference. For example, you can use the geomagnetic field sensor in combination with the accelerometer to determine a device's position relative to the magnetic north pole, as in a compass app. Position sensors are not typically used to monitor device movement or motion.

## Geomagnetic field sensor (magnetometer)

The geomagnetic field sensor ( `TYPE_MAGNETIC_FIELD` ), also known as the magnetometer, measures the strength of magnetic fields around the device on each of three axes ( $x$ , $y$ , $z$ ), including the Earth's magnetic field. Units are in microtesla (uT). Most Android devices include a hardware-based geomagnetic field sensor.

You can use this sensor to find the device's position in respect to the external world, for example, to create a compass app. However, magnetic fields can also be generated by other devices in the vicinity, by external factors such as your location on Earth (the magnetic

field is weaker toward the equator), or even from the current strength of solar winds. For a more accurate device orientation use the `SensorManager` class's `getRotationMatrix()` and `getOrientation()` methods, or the rotation-vector sensor.

The event data generated by the geomagnetic field sensor is as shown in the following table:

| Event data | Description | Units |
|---|---|---|
| `SensorEvent.values[0]` | Magnetic field strength along the *x* -axis. | uT |
| `SensorEvent.values[1]` | Magnetic field strength along the *y* -axis. | uT |
| `SensorEvent.values[2]` | Magnetic field strength along the *z* -axis. | uT |

### Orientation sensor

The orientation sensor ( `TYPE_ORIENTATION` ) measures degrees of rotation that a device makes around all three physical axes ( *x* , *y* , *z* ). Orientation was a software-only sensor that combined data from several other sensors to determine the position of the device in space. However, because of problems with the accuracy of the algorithm, this sensor type was *deprecated in API 8* and may be entirely unavailable in current devices. For a more accurate device orientation use the `SensorManager` class's `getRotationMatrix()` and `getOrientation()` methods, or the rotation-vector sensor.

## Related practical

The related practical documentation is in Working with Sensor-Based Orientation.

## Learn more

Android developer documentation:

- Sensors Overview
- Motion Sensors
- Position Sensors

Android API reference documentation:

- `Sensor`
- `SensorEvent`
- `SensorManager`
- `SensorEventListener`
- `Surface`

- `Display`

Other documentation:

- Accelerometer Basics
- Sensor fusion and motion prediction (written for VR, but many of the basic concepts apply to basic apps as well)
- Android phone orientation overview
- One Screen Turn Deserves Another

# 4.0: Performance

**Contents:**



As an Android developer, you are also an Android user, and you know what it feels like when an app doesn't load, takes a long time to load, stutters at every animation, crashes randomly, and on top of that, drains the battery, or uses up your whole data plan.

Users give higher ratings to apps that respond quickly to user input, perform smoothly and consistently, and respect mobile's limited battery resources.

To make a useful, interesting, and beautiful app stand out from the crowd, you should also make it as small, fast, and efficient as possible. Consider the impact your app might have on the device's battery, memory, and device storage, and be considerate of users' data plans when fetching and storing data on the Internet.

As with other bugs and design problems you may have with your app, it may be easy or hard to discover what is causing your app to perform below your expectations. The following two approaches combined have proven useful for tracking down, fixing, and testing for performance problems:

- Use tools to inspect your app and acquire performance data measurements.
- Use a systematic, iterative approach, so that you can measure improvements resulting from your changes to the app.

While much more comprehensive than the performance section in the Android Developer Fundamentals course, this chapter and its associated practicals are only starting points to get you on track to become a performance expert.

# What is good performance?

**App performance.** What happens on the screen is the user's most immediate experience of your app. Your app must consistently draw the screen fast enough for your users to see smooth, fluid motions, transitions, and responses. To accomplish this, your app has to do all its work in **16 milliseconds** or less for every screen refresh.

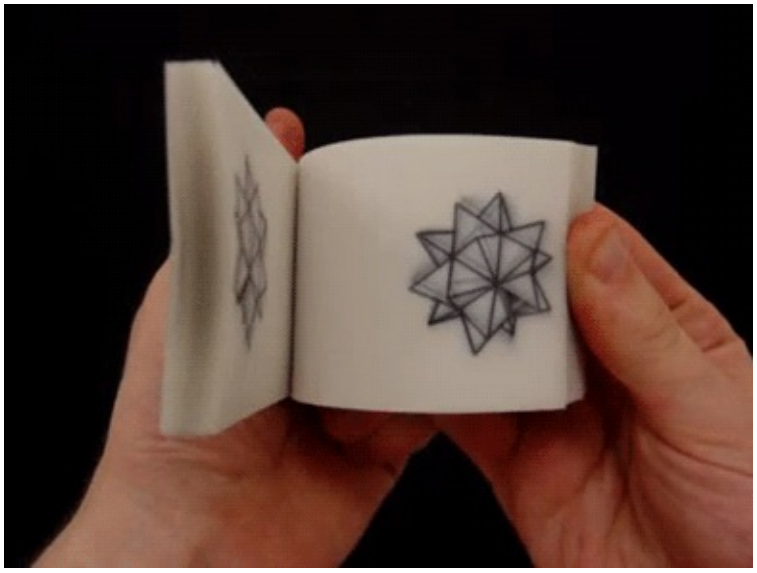
**Respecting the user.** When it comes to respecting the user's resources, it's about minimizing the use of mobile data and battery power. Users will uninstall apps that drain the battery.

**Balance.** Maximizing performance is all about balance. To give users the best possible experience, you want to find and make the best trade-offs between app complexity, functionality, and visuals.

# Why 16 milliseconds per frame?

## Human eyes and 16 ms per refresh

The human brain receives and processes visual information continuously. For example, when still images are displayed in sequence fast enough, people perceive them as motion, such as in the flip book shown below.



A flip book moves at around 10-12 pages per second. For movies, 24-30 images per second is enough, but still unconvincing without fancy cinematic effects. An ideal number is 60 images per second, as most people see this as high-quality, smooth motion.

This measurement is called "frame rate" or "frames per second" (FPS).

## Computer hardware and 60 frames per second == 16 ms per frame

The human eye is very discerning when it comes to motion inconsistencies. For example, if your app is running on average at 60 frames per second, and just one frame takes much longer than the preceding ones to display, users will notice a break in smoothness that is generally called "hitching," "lag," "stutter," or "jank."

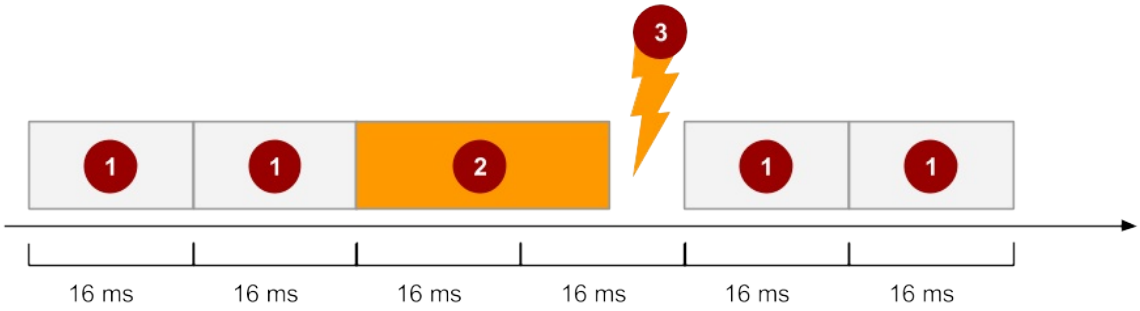
Most modern mobile devices refresh their displays at 60 frames per second. This number is dictated by a device's hardware, which defines how many times per second the screen can update.

To match the hardware screen's 60 updates per second, the system software attempts to redraw your current app activity every 16 ms (1000 ms / 60 frames = 16.666 ms/frame).

That's 16 ms total. Along with drawing the UI, the system has to take time to respond to intents, handle input events, and so on. Your app shares these 16 ms with a lot of other subsystems on Android, so 16 ms/frame is an upper bound.

In the diagram below:

1. Each frame takes 16 ms or less to present.
2. If your app takes too long and does not finish the update in 16 ms,
3. ... you get what is called a "dropped frame," "delayed frame," or "jank," which users see as a stutter.




As an app developer, you want your app to consistently use less than 16 ms per frame throughout your user's experience.

For more on this topic, see the Why 60 fps video.

# A basic performance test

What follows is the most basic performance test for your app . It requires no special setup or tools, just your app, you, and honest observation.

**Important:** To get accurate data for your app, you must perform any performance testing on a real device, not the emulator.

1. Install your app on the lowest-end device that your target audience might have. This is

very important. For example, if you built an app for mapping clean water wells in rural areas, your primary audience may not have the latest high-end devices.
2. Use your app as a user would, for as long as a typical user might. Keep detailed notes on even the smallest wait, stutter, or unresponsiveness. Watch out for increasing slowness over time.
3. Try to crash your app by tapping fast and randomly, even interacting in ways you would not want your users to use the app.
4. Hand the device to a friend and take notes as you watch them use it. Before they start, ask them about their expectations, then listen to their comments during use and get additional feedback when they are done. You will likely get feedback not just on performance, but also on UI efficiency.
5. Optionally, to get feedback from a larger audience, run a mini-usability test as demonstrated in the Usability Cafe video.

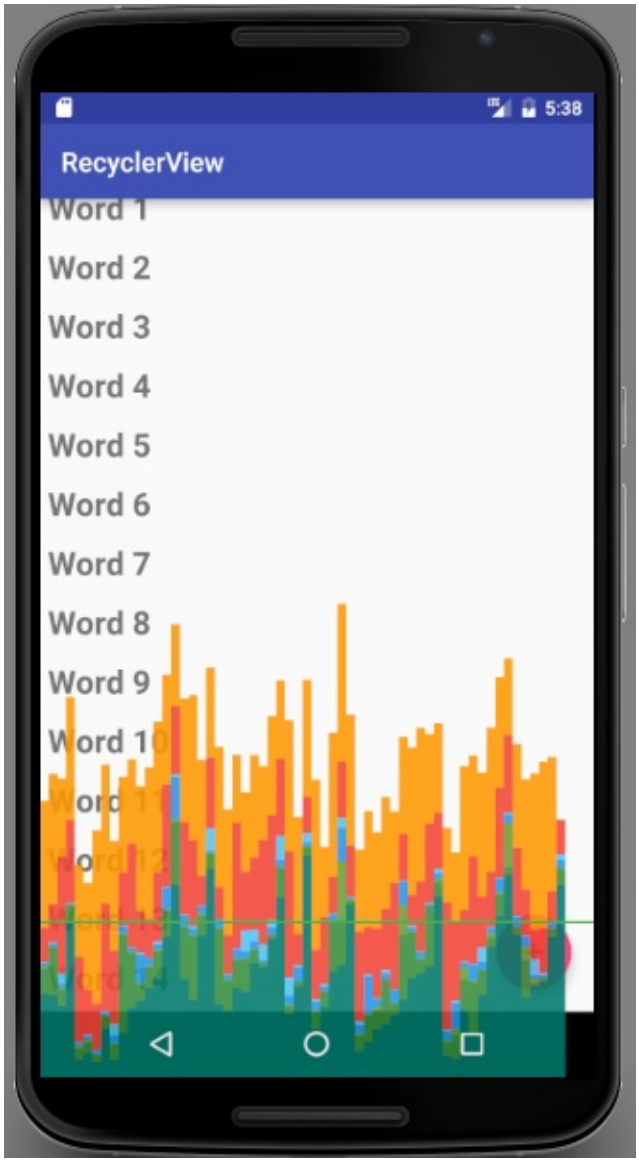
**Emulator tip:** To get accurate data, you need to do most of your performance testing on a real device. However, it might be hard to test with low-bandwidth and unreliable networks on a physical device. You can check how your app performs on low-bandwidth and unreliable networks by using the emulator and adjusting its settings. See Work with the Extended Controls and Start the emulator from the command line (look for the `netdelay` parameter). If these hands-on tests don't reveal any performance problems, you are off to a great start.

If you do gather any notes related to performance, then performance is a problem for your app, and you must figure out the underlying causes and fix them.

# Checking your frame rate

You can check how well your app does at rendering screens within the 16 ms/frame limit by using the Profile GPU Rendering tool on your Android device. You must have **Developer Options** turned on to use this tool.

1. Go to **Settings >Developer options**.
2. Scroll down to the **Monitoring** section.
3. Select **Profile GPU rendering**.
4. Choose **On screen as bars** in the dialog box. Immediately you will start seeing colored bars on your screen, similar to the ones in the screenshot below.

5. Return to your app. 

6. One bar represents one frame of rendering. The taller the bar, the longer the frame took to render.
7. The colors in each bar represent the different stages in rendering the screen.
8. If a bar goes above the green line, it took more than 16 ms to render. Ideally, most of the bars stay below the green line most of the time, but when you are loading an image for the first time, the bar may go significantly higher. Users may not perceive this as a problem, because they may expect to have to wait a moment for an image to load for the first time.

Below is an image of just bars for a device that is running Android 6.0 or higher. (The bars for older Android versions use different coloring. See Profile GPU Rendering Walkthrough for the color legend for older versions.)


For example, if the *green Input portion* of the bar is tall, your app spends a lot of time handling input events. Read more about what the different stages mean in Analyzing with Profile GPU Rendering. See the Profile GPU Rendering Walkthrough for a tutorial, and see the Profile GPU Rendering practical for details on how to use this tool to check and improve your app's performance.

# Approaching performance problems

## Use performance profiling tools

Android and Android Studio come with many performance profiling tools to help you identify performance problems and track down where they are in your code. For example:

- Test how fast the screen renders and where in the rendering pipeline your app may be doing too much work by using the Profile GPU Rendering tool described above.
- Visualize your app's view hierarchy by using the Layout Inspector (**Tools > Android > Layout Inspector**). Look for ways to improve view layout performance.
- Use `Batterystats` and Battery Historian to chart battery usage over time. Monitor battery level and charging state in your app using the BatteryManager class.

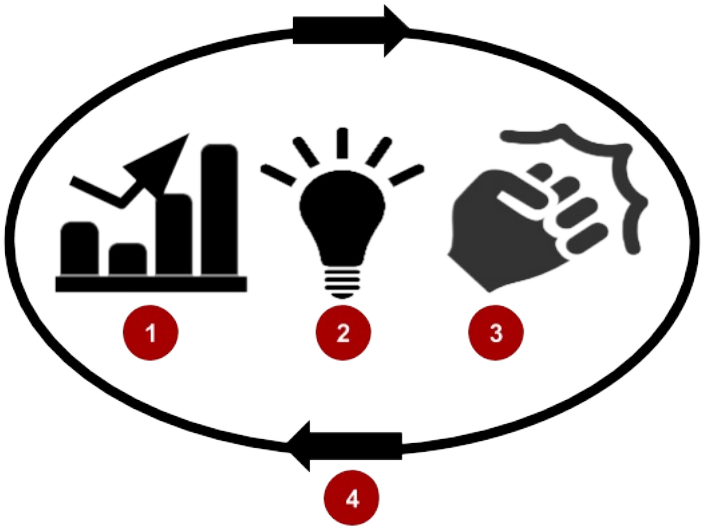
In Android Studio 3.0 and higher, you can use the Android Profiler tool:

- Use the CPU Profiler to monitor CPU usage over time.
- Find memory leaks and observe your app allocations with with the Memory Profiler.
- Monitor the frequency of data transfers with the Network Profiler, and identify where you might be able to transfer data less frequently or in larger chunks.

See Performance Profiling Tools for a comprehensive list and documentation.

## Follow the Performance Improvement Lifecycle

Use a systematic iterative approach to finding and fixing performance problems. This helps you track what you are testing, the data you gather, the fixes you implement, and your app's improvement over time. The performance tuning lifecycle is a three-step, repeatable way to




tune your app's performance.

The Performance Improvement Lifecycle consists of three repeated phases:

1. *Gather information.* Use the performance profiling tools described in Approaching performance problems, above, to capture data about your app. Record that data so that later, you can compare it to data you capture after you make changes to your app.

2. *Gain insight.* Take the time to understand what your gathered data means for your app. You may need to go through several iterations of gathering information with multiple tools to pinpoint what the problems are. For example, you may have a complex view hierarchy, but it renders fast, and the big problem is that your app is doing work on the UI thread that should be done on a background thread instead.

3. *Take action.* Evaluate ways to solve your problem. Taking action can take different forms, and what you decide to do depends on your situation and context. You may change the code and optimize layouts, or your backend. What you change may be affected by constraints such as coding resources, budgets, and deadlines.

4. *Verify.* Check to make sure that your changes had the desired outcome:

   1. Gather a second set of data. Run the tools again to measure the impact of your

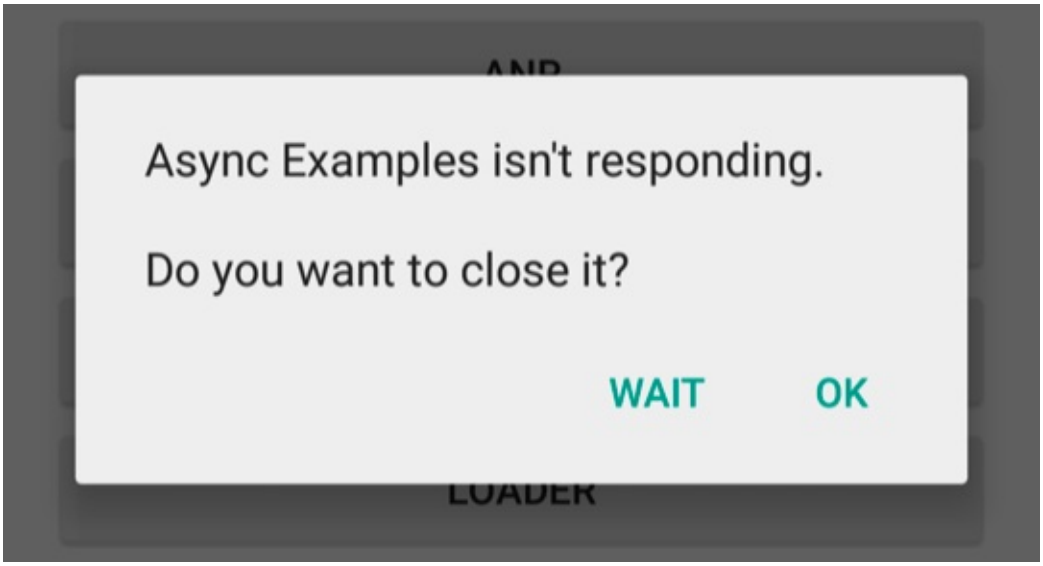
changes.
2. Compare it to your original data to make sure you've fixed the right problem, and to find out whether you need to look for additional performance improvements to make.
3. Make different or additional changes.

Repeat this process on all target devices until your app consistently responds smoothly to all user actions, never stutters, and never makes the user wait.

For a more on the Performance Improvement Cycle, watch Tools not Rules and The Performance Lifecycle.

# Keeping your app responsive

It's possible to write code that wins every performance contest in the world but still feels sluggish, hangs, freezes for significant periods, or takes too long to process input. The worst thing that can happen to your app's responsiveness is an "Application Not Responding" (ANR) dialog like the one shown below. At this point, your app has been unresponsive for a considerable period of time, so the system offers the user an option to quit the app. The system displays an "Application Not Responding" (ANR) dialog when an app is unresponsive.




Android displays an ANR dialog when it detects one of the following conditions:

- Your app doesn't respond within 5 seconds to an input event such as a key press or screen tap. This can happen if your app performs too much work on the UI thread, in particular in `onCreate()`.

  Examine your code. If any work is not required to complete before the app can proceed, move it off the UI thread, for example using a `Loader` or an `AsyncTask`. Show the user that progress is being made, for example with a progress bar in your UI.

- A `BroadcastReceiver` hasn't finished executing within 10 seconds.

  To prevent ANRs from happening for this reason, use broadcast receivers only for their intended purpose. Broadcast receivers are meant to do small, discrete amounts of work in the background, such as saving a setting or registering a notification.

It's critical to design responsiveness into your application so the system never displays an ANR dialog to the user. Use performance tools such as Systrace and Traceview to determine bottlenecks in your app's responsiveness. See the practicals for introductions to performance tools.

See Keeping Your App Responsive for more details.

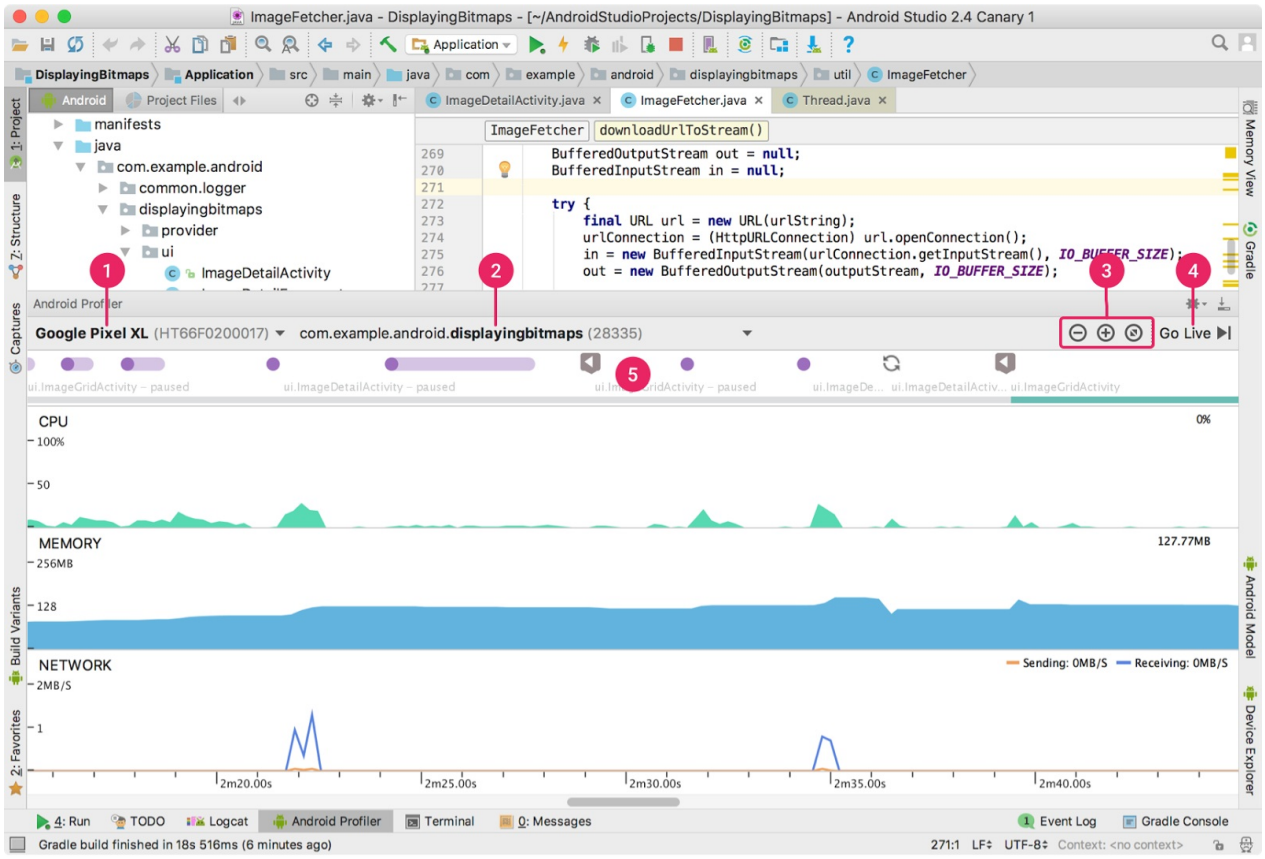
# Monitoring the performance of your running app

Android Studio has tools to measure your app's memory usage, CPU, and network performance. The advanced profiling tools display realtime data updates for CPU, memory, and network activity.

To use the Android Profiler tool (available in Android Studio 3.0 and higher):

1. In Android Studio, at the bottom of the window, click the **Android Profiler** tab.
2. Run your app and interact with it. The monitors update to reflect the app's use of resources. Note that to get accurate data, you should do this on a physical, not virtual, device.

The default view in the Android Profiler window, as shown below, displays a simplified set of data for each profiler. When you click on a tool, it opens to show a more details with more functionality.




The meaning of the numbers in the screenshot:

1. Dropdown for selecting the device.
2. Dropdown for selecting the app process you want to profile.
3. Timeline zoom controls.
4. Button to jump to the realtime data, for example after scrolling to inspect previous data.
5. Event timeline that shows the lifecycle of activities, all input events, and screen rotation events. This timeline can help you relate what you see in the graphs to areas of your code.

The event timeline shows three monitors:

- The *Memory Profiler* (the MEMORY area in the screenshot) shows you a realtime count of allocated objects and garbage collection events. It also allows you to capture heap dumps and record memory allocations.
- The *CPU Profiler* (the CPU area in the screenshot) shows realtime CPU usage for your app process and system-wide CPU usage.
- The *Network Profiler* (the NETWORK area in the screenshot) displays realtime network activity. It shows data sent and received, as well as the current number of connections. You can also see the event timeline and radio power state (high/low) vs Wi-Fi.

Read the Android Profiler documentation to learn more about using the monitors, and do the performance practicals to experiment with them.

# Learn more

Android developer documentation:

- Best Practices for Performance
- Performance Profiling Tools (the landing page for all the tools)
- Android Profiler
- Search developer.android.com for "performance" for a full list of the latest performance docs.

Articles:

- You, Your App, and Android Performance
- Exceed the Android Speed Limit!

Video and community:

- Android Performance Patterns on YouTube
- Why 60 fps?
- Tools not Rules
- The Performance Lifecycle
- Android Performance G+ Community

# 4.1: Rendering and layout

**Contents:**



The Android Developer Fundamentals course talked about how to make your apps look interesting and visually compelling using Material Design guidelines, and it taught you how to use the Layout Editor to create your layouts. You learned that you can create nested hierarchies of layouts. You learned how to use drawables as background elements for your views. These elements allow you to create complex nested layouts with backgrounds and views overlapping each other throughout your app.

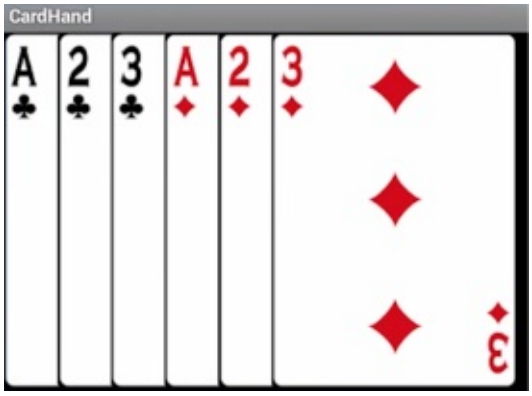
However, your layouts draw faster and use less power and battery if you spend time designing them in the most efficient way.

To design an efficient layout:

- Minimize overdraw.
- Simplify complex view hierarchies.

## Minimize overdraw

Every time your app draws a pixel on the screen, it takes time. Every time your app draws an opaque pixel to replace something that has already been drawn, it wastes time. Drawing a pixel more than once per screen refresh is called *overdraw* , and it's a common problem affecting the performance of modern applications. Strive to create an app that draws every changed pixel only once.

For example, an app might draw a stack of 52 overlapping cards, with only the last card fully visible. Completely drawing the 51 cards that are underneath and partially covered is an example of overdraw. 


The most likely symptom you will see in an app with overdraw is slow rendering and stuttering animations. This is the most generic of symptoms. Since overdraw is common, and straightforward to test for, make it a habit to check for it every time you change the views of your app.

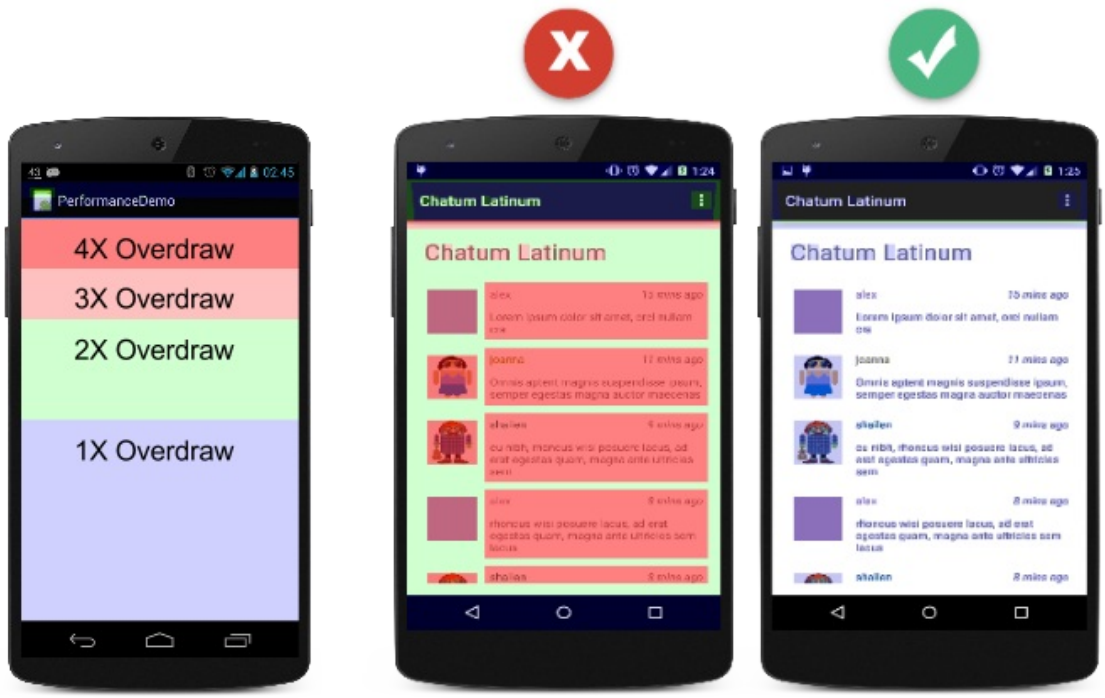
If at any point your app is drawing something that the user does not see, don't draw that thing.

## Test for overdraw

You can visualize overdraw using color tinting on your device with the Debug GPU Overdraw tool.

To turn on Debug GPU Overdraw on your mobile device:

1. In **Settings > Developer options**, scroll to **Hardware accelerated rendering**.
2. Select **Debug GPU Overdraw**.
3. In the **Debug GPU overdraw** dialog, select **Show overdraw areas**.
4. Watch your device turn into a rainbow of colors. The colors are hinting at the amount of overdraw on your screen for each pixel.
    - True color has no overdraw. ( *True color* means there's no change from what the app normally shows.)
    - Purple/blue is overdrawn once.
    - Green is overdrawn twice.
    - Pink is overdrawn three times.
    - Red is overdrawn four or more times.




## Remove unnecessary backgrounds

One simple thing you can do to reduce overdraw is to remove backgrounds that the user never sees:

1. Search your code for `android:background`.
2. For each background, determine whether it's needed and visible on the screen.
3. If the view's background is covered by something else, for example an image or the children's backgrounds, remove the `android:background` line of code from the view.

Before:

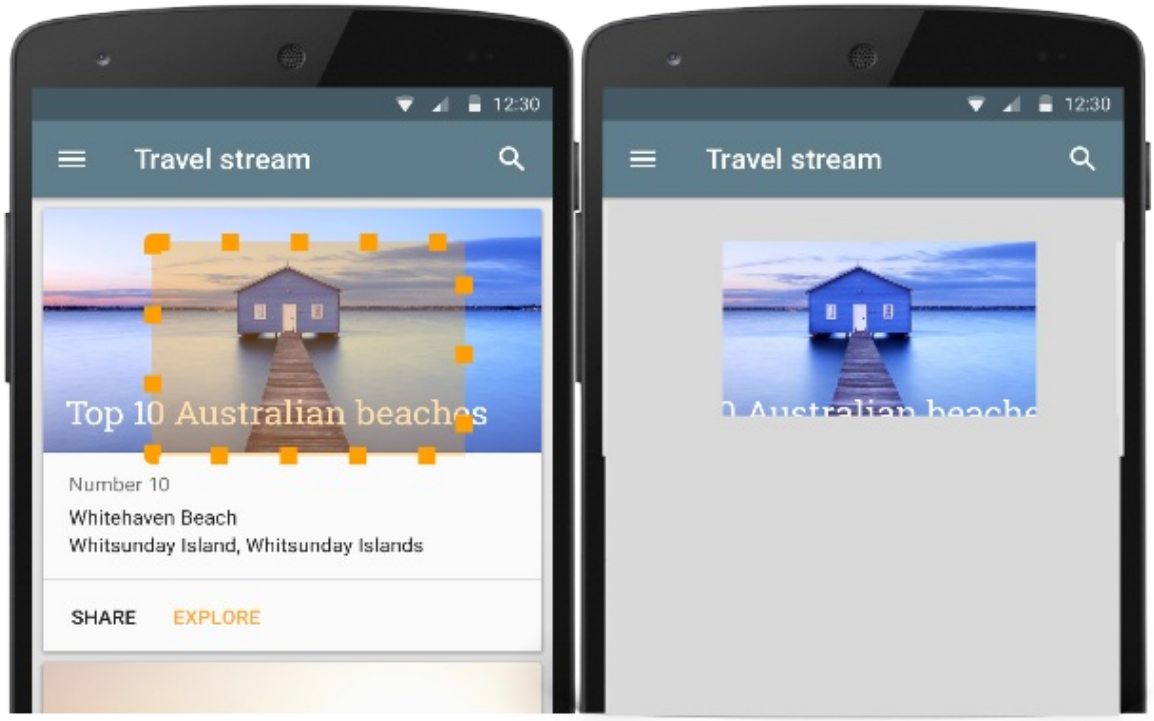
```
<ImageView
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:src="@drawable/beach"
   android:background="@android:color/white">
</ImageView>
```

After:

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

## When drawing custom views, clip generously

In the context of drawing on the screen, clipping is a way to exclude regions from being drawn. In the basic form of clipping, you provide the system with a rectangle and instruct it to only draw what falls inside that rectangle, as shown below.




One use for clipping is to only draw the parts of a view that the user sees, reducing the amount of rendering work the system has to do, which can improve the performance of your app. However, clipping is not free, and it is better to arrange your views without overlapping in the first place, for example by using `ConstraintLayout`.

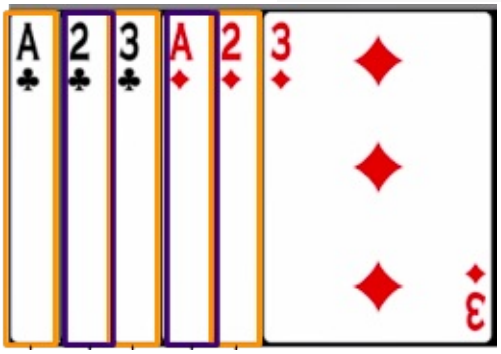
For standard views (like `ImageView`, `Button`, and `ProgressBar`), the Android system reduces overdraw as much as it can and avoids drawing views that are completely hidden. For example, nothing underneath an open navigation drawer is drawn.

For custom views where you override the `onDraw()` method, the underlying system doesn't have insight into how you're drawing your content, which makes it hard for the system to know what to avoid. For example, without clipping, you would draw the full cards even though only a small part of each card is visible.

## clipRect()

You can help the system draw custom views efficiently by using the `Canvas.clipRect()` method. This method allows you to define a rectangle for a view, and only the content inside the rectangle is drawn.

For the set of stacked overlapping cards, you can determine what part of the current card is visible, then use the `Canvas.clipRect()` method to set your clipping rectangle accordingly.



## quickReject()

Even if you only draw a small circle in a corner of your custom view, that entire view is rebuilt. Instead of recalculating and redrawing the whole screen, you can calculate the clipping rectangle for the changed area, then use the `quickReject()` method inside your `onDraw()` method to test for places where clipping rectangles intersect. If some part of a view that takes a lot of processing time is outside the clipping rectangle, `quickReject()` can tip you off, so that you can skip that processing altogether.

## Complex clipping

Beyond `clipRect()` and `quickReject()`, the Canvas class provides methods for complex clipping ( `clipPath()` , `clipRegion()` , or applying `clipRect()` while rotated). This kind of clipping can be expensive, and it isn't anti-aliased.

To minimize complex clipping, compose drawing elements of the right shape and size. For example, using `drawCircle()` or `drawPath()` is much cheaper than using `clipPath()` with `drawColor()` .

## Reduce transparency

Rendering transparent pixels on the screen is called *alpha rendering* . Alpha rendering contributes to overdraw, because the system has to render both the transparent pixel and what is underneath the pixel, then blend the two to create the see-through effect.

Visual effects like transparent animations, fade-outs, and drop shadows all involve transparency, and can therefore contribute significantly to overdraw. To improve overdraw in these situations, reduce the number of transparent objects you render. For example, consider making text a solid color and using transitions that don't use alpha, such as wipe transitions.

To learn more about the performance costs that transparency imposes throughout the entire drawing pipeline, watch the Hidden Cost of Transparency video.

### Overdraw still matters

As with many other performance challenges, overdraw may not matter on the most advanced mobile device. But most people in the world have less powerful devices. Reducing overdraw for them can save significant resources and greatly decrease the time it takes to draw a frame.

See Reducing Overdraw for more information.

## Simplify complex view hierarchies

At the heart of your app is the hierarchy of views that makes up the UI and the visual experience of users. With feature-rich apps, this hierarchy grows large and complex and can become a source of performance problems. The most likely symptom you will see is a generic general slowness of the app, especially when rendering complex views to the screen.

It is a common misconception that using the basic layout structures leads to the most efficient layouts. However, each widget and layout you add to your application requires initialization, layout, and drawing. For example, using nested instances of `LinearLayout` can lead to an excessively deep view hierarchy. Nesting several instances of `LinearLayout` that use the `layout_weight` parameter can be especially expensive, as each child needs to be measured twice.

Simplifying or rearranging the view hierarchy of your app can improve performance, especially on lower-end devices and earlier versions of `Android`. As an added benefit, your app will become easier to maintain.

To eliminate or reduce many of these issues, use a `ConstraintLayout` instance to build your UI when possible. For details, see Build a Responsive UI with `ConstraintLayout`.

### Measure and layout

The rendering pipeline includes a measure-and-layout stage, during which the system positions the items in your view hierarchy. The "measure" part of this stage determines the sizes and boundaries of `View` objects. The "layout" part determines where on the screen to position the `View` objects.

Both of these stages incur a small cost for each view or layout that they process. Most of the time, this cost is minimal and doesn't noticeably affect performance. However, it can be greater when an app adds or removes `View` objects, such as when a `RecyclerView` object recycles or reuses a view.

The cost can also be higher if a `View` object needs to resize to maintain its constraints. For example, if your app calls `setText()` on a `View` object that wraps text, the `View` may need to be larger or smaller to accommodate the characters. If this process takes too long, it can prevent a frame from rendering fast enough. The frame is dropped, and animation becomes janky.

You cannot move measure and layout to a worker thread, because your app must process this stage of rendering on the main thread. Your best bet is to optimize your views so that they take as little time as possible to measure and lay out.

## Double taxation

Typically, the system executes the measure-and-layout stage in a single pass and quite quickly. However, in complicated layout cases, the system may have to iterate multiple times on parts of the hierarchy before positioning the elements. Having to perform more than one measure-and-layout iteration is referred to as *double taxation* .

For example, the `RelativeLayout` container allows you to position `View` objects with respect to the positions of other `View` objects. When you use `RelativeLayout` , the system performs the following actions:

1. Executes a measure-and-layout pass. During this pass, the system calculates each child object's position and size, based on each child's request.
2. Uses this data to figure out the proper position of correlated views, taking object weights into account.
3. Performs a second layout pass to finalize the objects' positions.
4. Goes on to the next stage of the rendering process.

Using `ConstraintLayout` can help you minimize the double taxation that `RelativeLayout` causes. `ConstraintLayout` provides similar functionality to `RelativeLayout` , but at a significantly lower cost.

## Analyze the view hierarchy

Analyze the view hierarchy by using several tools together to pinpoint and fix performance problems. You may need to use just one of these tools to optimize performance, or you may need to use them all.

## Profile GPU Rendering

Run the Profile GPU Rendering tool and look at the light and dark blue segments of the bars. If the blue segments are tall and causing the bars to cross the green 16-ms-per-frame line, your app spends a lot of time updating display lists. The M release of Android adds additional color segments, including a light-green Measure/Layout segment. If this segment is large, your view hierarchies might be unnecessarily complex.

The Profile GPU Rendering tool only tells you there might be a problem; it doesn't tell you where to look.

## Show GPU view updates

On your physical mobile device, run the tool that shows GPU view updates.

1. In **Settings > Developer options**, scroll to **Hardware accelerated rendering**.
2. Enable **Show GPU view updates**.
3. Interact with your app.

When a view on the screen is updated, it flashes red. If views on your screen are flashing, and they have nothing to do with the area that should be updated, look at your code and determine if they get invalidated unnecessarily because, for example, you may have mistakes in parent-child view relationships.

## Layout Inspector

The Layout Inspector allows you to inspect your app's view hierarchy at runtime from within Android Studio. This is particularly useful when your layout is built at runtime rather than being defined entirely in an XML layout.

## Lint

Use the lint tool on your layout files to search for possible view hierarchy optimizations. Lint tool automatically runs whenever you compile your program.

In Android Studio, choose **Analyze > Inspect Code...** to manually run the

Lint tool on your whole project, a particular module, or a specific file.

To manage inspection profiles and configure inspections within Android Studio:

1. Choose **File > Other Settings > Default Settings**. The **Default Preferences** dialog appears.
2. Click **Editor** and then **Inspections** in the navigation bar.

3. Click **Android** and **Lint** in the right-hand pane.
4. Select or clear inspection checkboxes as desired.

The following are some of the lint inspections related to the view hierarchy. They are listed under **Android > Lint > Performance** in the **Editor > Inspections** preferences.

- **"Node can be replaced by a `TextView` with compound drawables"**

  A `LinearLayout` that contains an `ImageView` and a `TextView` can be more efficiently handled as a compound drawable. In a compound drawable, you add a `Drawable` to a `TextView` using one of the `setCompoundDrawables` methods, and you specify how the text flows around the `drawable`.

- **" `FrameLayout` can be replaced with `<merge>` tag"**

  If the root of a layout is a `FrameLayout` that does not provide background, padding, and so on, you can replace the `FrameLayout` with a merge tag, which is slightly more efficient.

- **"Useless leaf layout"**

  If a layout has no children or no background, it's invisible. You might be able to remove it for a flatter layout hierarchy.

- **"Useless parent layout"**

  A layout with children and no siblings that is not a `ScrollView` or a root layout, and does not have a background, can be removed for a flatter and more efficient layout hierarchy. For example, if you have one `LinearLayout` inside another `LinearLayout`, you can probably remove one of them.

- **"Layout hierarchy is too deep"**

  Consider using flatter layouts, for example by using the `ConstraintLayout` class. The default maximum depth in the lint tool is 10.

See Simplify your view hierarchy, below.

## Systrace and dumpsys

The Systrace tool, which is built into the Android SDK, provides excellent data about performance. Systrace allows you to collect and inspect timing information across an entire device, allowing you to see when layout performance problems cause performance problems. For more information, see Analyzing UI Performance with Systrace.

Systrace is sometimes used with `dumpsys`, a tool that runs on the device and dumps status information about system services.

You will use Systrace and `dumpsys` in the Systrace and dumpsys practical.

## Simplify your view hierarchy

**Remove views that do not contribute to the final image.**

Eliminate from your code the views that are completely covered, never displayed, or outside the screen. This seems obvious, but during development, unnecessary views can accumulate.

**Flatten the view hierarchy to reduce nesting.**

Android layouts allow you to nest UI objects in the view hierarchy. This nesting can impose a cost. When your app processes an object for layout, the app performs the same process on all children of the layout as well.

Keep your view hierarchy flat and efficient by using `ConstraintLayout` wherever possible.

**Reduce the number of views.**

If your UI has many simple views, you may be able to combine some of them without diminishing the user experience.

- Combining views may affect how you present information to the user and will include design trade-offs. Opt for simplicity wherever you can.
- Reduce the number of views by combining them into fewer views. For example, you may be able to combine `TextViews` if you reduce the number of fonts and styles.

**Simplify nested layouts that trigger multiple layout passes.**

Some layout containers, such a `RelativeLayout`, require two layout passes in order to finalize the positions of their child views. As a result, their children also require two layout passes. When you nest these types of layout containers, the number of layout passes increases exponentially with each level of the hierarchy. See the Optimizing View Hierarchies documentation and the Double Layout Taxation video.

Be conscious of layout passes when using:

- `RelativeLayout`
- `LinearLayout` that also use `measureWithLargestChild`
- `GridView` that also use `gravity`
- Custom view groups that are subclasses of the above
- Weights in `LinearLayout`, which can sometimes trigger multiple layout passes

Using any of the listed view groups as

- the root of a complex view hierarchy,

- the parent of a deep subtree,
- or using many of them in your layout,

can hurt performance.

Consider whether you can achieve the same layout using a view group configuration that does not result in these exponential numbers of layout passes, such as a `ConstraintLayout`. See Build a Responsive UI with ContstraintLayout and Build a UI with Layout Editor.

Another advanced technique is delayed loading of views, which is discussed in Delayed Loading of Views and Re-using Layouts with .

# Related practicals

Related practical documentation:

- 4.1A: Profile GPU Rendering
- 4.1B: Debug GPU Overdraw, Layout Inspector
- 4.1C: Systrace & dumpsys

# Learn more

Android developer documentation:

- Layouts
- Optimizing Layout Hierarchies
- Build a Responsive UI with ContstraintLayout
- Build a UI with Layout Editor
- Improving Layout Performance
- Performance and View Hierarchies

Articles:

- Draw What You See

Video and community:

- Android Performance Udacity course
- Android Performance Patterns on YouTube
- Android Performance G+ Community

Tool documentation:

- Performance Profiling Tools

- Analyzing with Profile GPU Rendering

# 4.2: Memory

**Contents:**



Android manages memory for you, but if your code uses memory inefficiently, all of Android's management cannot make your app perform well. Cleaning up memory resources takes time, which takes precious milliseconds away from other processes.

Memory is a huge topic. Teaching you the general basics, as well as the deep intricacies, is beyond the scope of this course. This chapter provides Android-related background, introduces the tools used in the practical, and describes common memory-related problems seen in apps.

## Memory in Android

Android is a managed-memory environment. You don't have to manually manage memory, because the system does it for you. The Android Team has put a lot of work into making memory management as fast and efficient as possible.

The Android runtime (ART) (5.0/API 21) and Dalvik virtual machine use paging and memory mapping to manage memory. When an app modifies a memory by allocating new objects or touching memory-mapped pages, the memory remains in RAM and cannot be paged out. The only way to release memory from an app is to release object references that the app holds, making the memory available to the garbage collector.

Learn more about ART and Dalvik. Get a refresher on Memory Management in Java.

## Garbage collection

The process of cleaning up and freeing memory resources is called *garbage collection* .

A managed-memory environment, like the ART or Dalvik virtual machine, keeps track of each memory allocation. When the system determines that a piece of memory is no longer used by the program, it frees the memory back to the heap, without your intervention. Garbage collection has two goals:

- Find data objects that are no longer referenced and therefore cannot be accessed in the future.
- Reclaim the resources used by those objects.

Garbage collection is a necessary process, but if it happens too often, it can seriously impact your app's performance. Even though garbage collection can be quite fast, it can still affect your app's performance. You don't generally control when a garbage collection event occurs from within your code.

The system has criteria for determining when to perform garbage collection. When the criteria are satisfied, the system begins garbage collection. If garbage collection occurs in the middle of an intensive processing loop such as an animation or during music playback, it can increase processing time. This increase can push code execution time past the recommended 16-millisecond threshold for efficient and smooth frame rendering.

Your code flow may force garbage collection events to occur more often than usual or make them last longer than normal. For example, if you allocate multiple objects in the innermost part of a for-loop during each frame of an alpha-blending animation, you might pollute your memory heap with a lot of objects. In that circumstance, the garbage collector executes multiple garbage collection events and can degrade the performance of your app.

To maintain a functional multitasking environment, Android sets a hard limit on the heap size (the potentially available memory for each app). The exact heap size varies based on how much RAM the device has available overall. If your app has reached the heap capacity and tries to allocate more memory, it can receive an `OutOfMemoryError` object.

Android tries to share RAM pages across processes. For details, see Sharing Memory. To learn how to properly determine your app's memory use, see Investigating Your RAM Usage.

For more about Android garbage collection, see Garbage collection in the Android documentation. For general information about garbage collection, see the Garbage collection article on Wikipedia.

## Switching apps

Apps in the foreground are visible to the user or are running a foreground service such as music playback. When a user switches between apps, Android keeps apps that are not in the foreground in a least-recently-used (LRU) cache. For example, when a user first

launches an app, a process is created for the app, but when the user leaves the app, that process does not quit. The system keeps the process cached. If the user returns to the app, the system reuses the process, which makes app switching faster.

If your app has a cached process, and it retains memory that it currently does not need, then your app—even while the user is not using it—affects the system's overall performance. As the system runs low on memory, the system kills processes in the LRU cache, beginning with the process least recently used. The system also accounts for processes that hold onto the most memory and can terminate them to free up RAM.

For more information, see the Processes and Threads guide.

# Memory leaks and memory churn

*Memory leaks:* A memory leak happens when your code allocates memory for objects, but never frees that memory. Over time, the memory allocated for these objects acts as a large, immovable block, forcing the rest of the app to operate in what's left of the heap. Eventually, the app can run out of memory and crash. Memory leaks can be huge and obvious, such as when your app loads an image that is larger than available memory. Memory leaks can also be tiny and hard to find, because the user will only notice the app getting slower and slower over time. The ultimate result of memory leaks is when the app runs out of available memory and, from the user's perspective, "suddenly crashes after running just fine for a long time."

*Memory churn:* Memory can also become tight if you allocate and free a large number of objects in a short time, saturating your heaps, and starting more garbage collections as a result. For example, memory churn can happen if you allocate new objects in the middle of nested loops, or in `onDraw()` methods. Even if the app does not run out of memory, the user will notice slowdown or stuttering of the app due to the frequent garbage collection.

To avoid memory leaks and churn:

- Don't leak `View` objects.
    - Do not reference views from outside the UI thread. If a view is unused but still referenced, this view and its whole subhierarchy of children cannot be freed. Views have references back to their activities, so if a view cannot be freed, its whole associated activity also stays in memory. And because activities are re-created when the user changes the orientation of their device, a lot of no-longer-used objects stay around.
    - Do not reference a view in an async callback. The view cannot be freed until the task is done, and the view may be invalid anyway, by the time the `AsyncTask` completes.

- Do not reference views from static objects. Static objects stay around for the lifetime of the process, which can be a lot longer than the activity.
- Don't put views into collections, such as `WeakHashmap`, that don't have clear memory patterns.
- For more information, watch the Do Not Leak Views video.
- Avoid looping allocations. Do not allocate objects in inner loops. Do it outside the loop, or redesign to avoid allocation in the first place.
- Avoid allocating objects in the `onDraw()` methods of your custom views. The `onDraw()` method is called on every frame.
- Use object pools. Allocate a group of objects and reuse them. Object pools are data structures which hold on to unused objects for you. Rather than free an object back to the heap when you're done with it, you hand the object off to the object pool. Later, when some function wants a new object of that type, you can request an object from the pool, rather than allocating a new one from the heap. Android provides the `Pool` class. However, if you use pools, you have to manage your own memory for these objects. Learn more with the Object Pools video.

To learn more, watch the Performance Cost of Memory Leaks and Memory Churn and Performance videos.

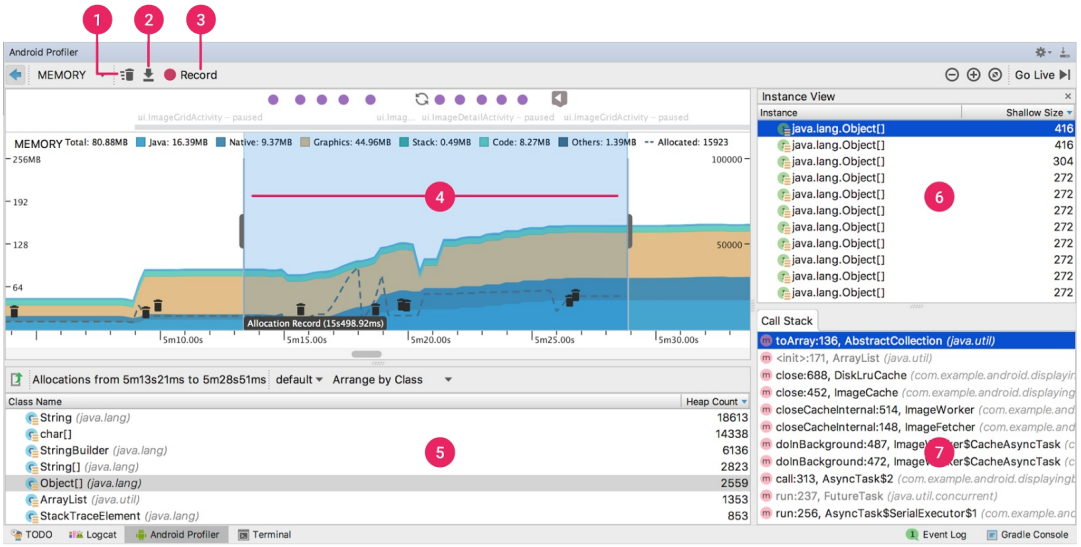
# The Memory Profiler tool

The Memory Profiler is a component of the Android Profiler. With the Memory Profiler you can

- view a real-time count of allocated objects and garbage collection events on a timeline,
- capture heap dumps,
- record memory allocations.

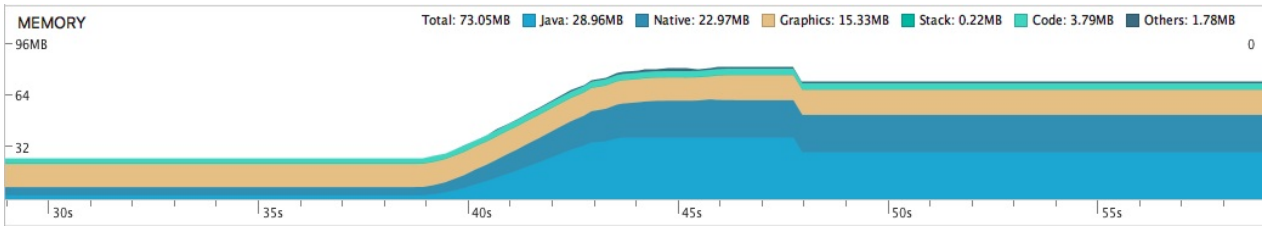
To start Memory Profiler (see image below):

1. Run your app on a device with **Developer options** enabled.
2. In Android Studio, open the **Android Profiler** (1) from the bottom toolbar.
3. Select your device and app, if they are not already selected (2). The **MEMORY** graph starts to display (3).

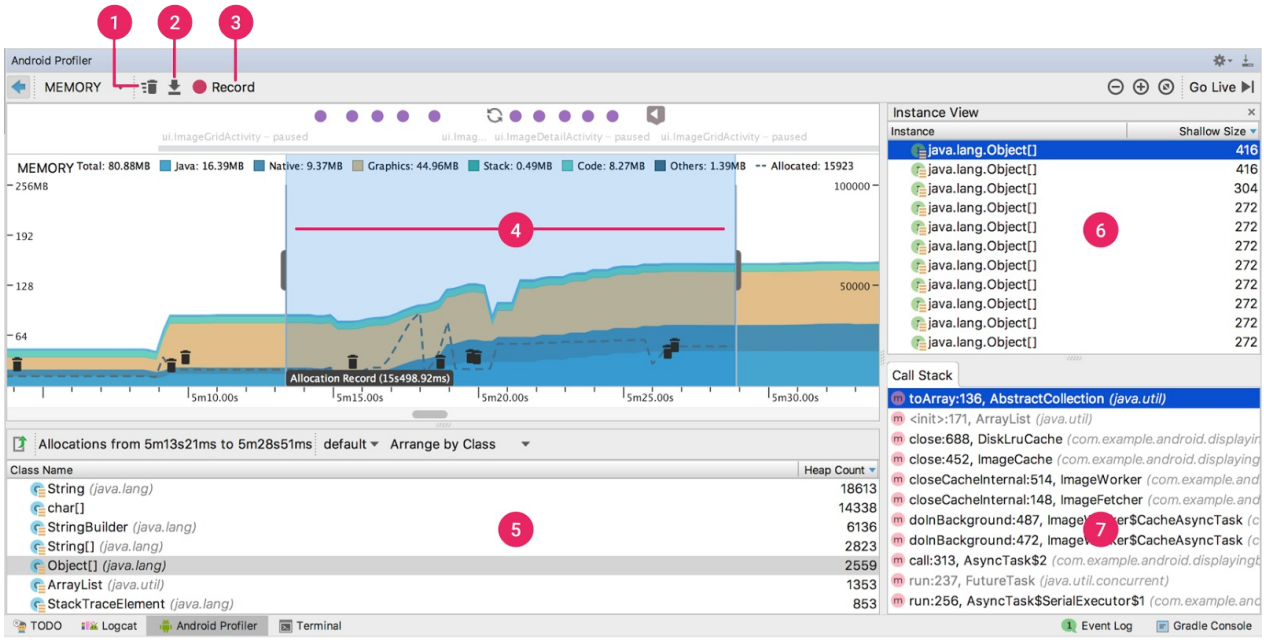



When you first open the Memory Profiler, the timeline shows the total amount of memory used by your app. Memory size is shown on the y-axis.


Click on the memory bar, and the graph expands into memory types. Each memory type (such as Java, native, and graphics) is shown with a different color in a stacked Memory graph. The legend above the graph shows the amount of each type of memory.




The figure and explanations below summarize the tools and functionality available in the Memory Profiler. See the Memory Profiler documentation and the Memory Profiler practical for details on how to use this tool.



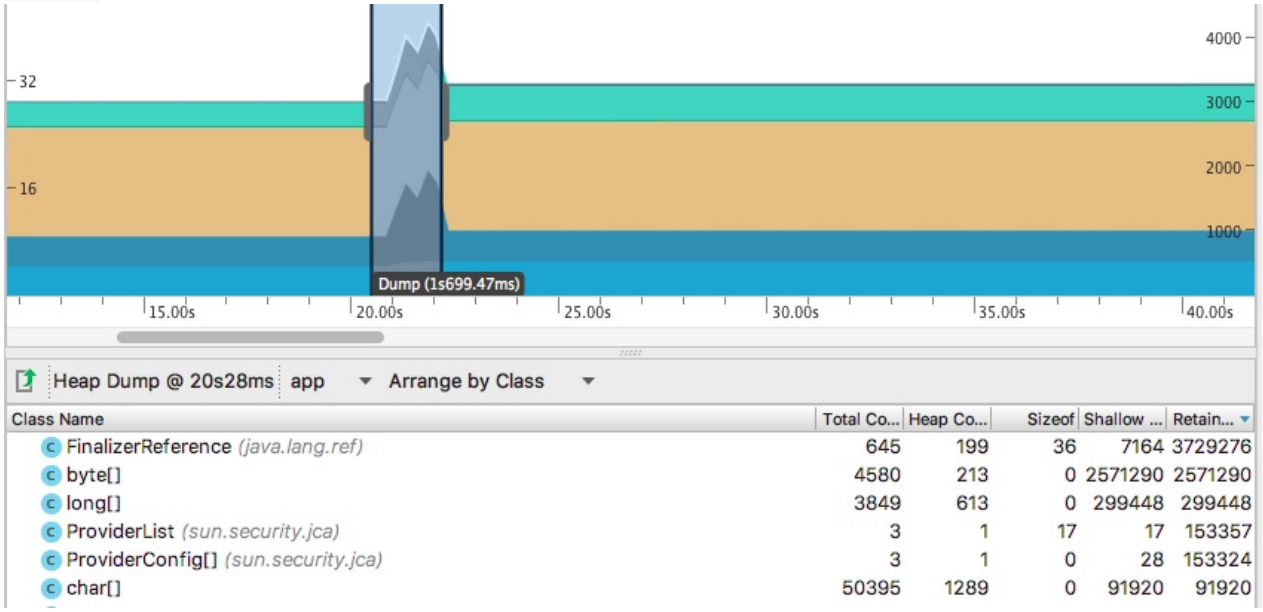

Legend for the image:

- (1) Force garbage collection.
- (2) Capture a heap dump.
- (3) Record memory allocations.
- (4) The period of time during which memory allocations were recorded.
- (5) Memory allocation results during the time indicated in the timeline. When viewing either a heap dump or memory allocations, you can select a class name from this list to view the list of instances on the right.
- (6) Click a class name to populate the **Instance View**.
- (7) Click on an instance to display its **Call Stack**. When you are viewing the allocation record, the **Call Stack** shows the stack trace for where that memory was allocated. When you are viewing a heap dump, the **Call Stack** shows the remaining references to that object.

To export the recordings to an `hprof` file (for heaps) or an `alloc` file (for allocations), click the **Export**

 button in the top-left corner of the **Heap Dump** or **Allocations** pane. Load the file into Android Studio later for exploration.

See the `HPROF` documentation for details on the user interface and more on working with `HPROF` files.




# Related practical

The related exercises and practical documentation is in Memory Profiler.

# Learn more

- Android Profiler overview
- Memory Profiler
- Overview of Android Memory Management
- Understanding Memory Management (Java)
- Garbage collection (generic)
- Manage Your App's Memory
- Investigating your RAM Usage
- Loading Large Bitmaps Efficiently
- Handling Bitmaps
- Android Performance Patterns video series
- Garbage Collection in Android
- Do Not Leak Views
- Memory and Threading
- Object Pools

# 4.3: Best practices: network, battery, compression

**Contents:**



Most users are on limited bandwidth with expensive data plans, and all users are unhappy when an app drains their battery. Networking is one of the biggest users of battery. To reduce battery drain and reduce the size and frequency of data transfers, follow best practices to optimize your networking.

Over half of users worldwide will experience your app over a 2G connection. Improve their experience by optimizing your app for low-speed connections and offline work. To do this, store data, queue requests, and handle images for optimal performance.

The general aim for developers is to:

- Reduce active radio time. You can do this by optimizing the frequency of network requests, and by batching requests.
- Reduce size of data per request.

Optimizing network and battery performance is a huge topic, and as devices and the Android OS change, so do some of the recommendations. The Android team constantly improves the framework and APIs to make it easier for you to write apps that perform well. To dive deeper and get the most up-to-date recommendations, make use of the extensive resources to which this document links.

This chapter presents a basic introduction to network and battery optimization, along with recommended practices.

## Radios

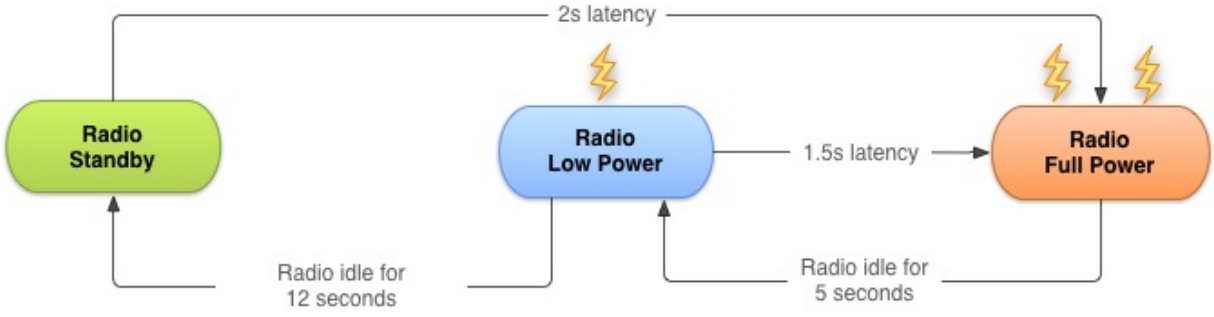
Inside your mobile device is a small piece of hardware that's essentially a *radio* . The purpose of this radio is to communicate with local cell towers and transmit data. However, this radio is not always active and drawing power. It starts in a powered down state, and when the device needs to send data, the radio turns on and transfers the data. After the data packet is sent, the radio stays awake for a while, in case the server sends a response. Eventually, the radio goes back to sleep to save battery.

**IMPORTANT:** The exact number of states and amounts of power drawn depend on the device and the Android version. The radio state machine on each device varies based on the mobile radio technology employed—2G, 3G, LTE, etc. This variation is particularly true of the transition delay ("tail time") and startup latency. The device's carrier network defines and configures these attributes.
The state machine for a typical 3G network radio consists of three energy states:

1. *Full power:* Used when a connection is active, allowing the device to transfer data at its highest possible rate.
2. *Low power:* An intermediate state that uses around 50% of the battery power of the full-power state.
3. *Standby:* The minimal energy state during which no network connection is active or required.

While the low and standby states drain a lot less battery than the full-power state, they also introduce significant latency to network requests. Moving from low power to full power takes around 1.5 seconds, and moving from standby to full power can take over 2 seconds.

To minimize latency, the state machine uses a delay to postpone the transition to lower energy states. The diagram below uses AT&T's timings for a typical 3G radio. The diagram shows how long it takes the radio to switch between standby, a low-power state, and a full-power state. The arrows show the direction of state changes and the latency in seconds between state changes.



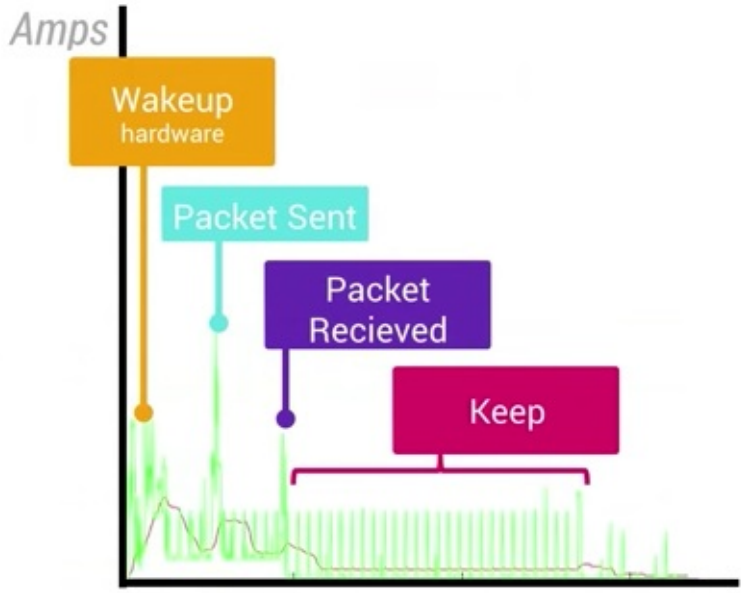

For more about Android radio states, see The Radio State Machine.

From the perspective of sending a request and receiving a response:

- There is a high drain on the battery when the radio first wakes up, as shown in yellow in

the image below (**Wakeup hardware**).

- There are some additional spikes when data packets are sent and received, as shown in blue and purple in the image (**Packet Sent** and **Packet Received**).
- There is a continuous draw when the radio is in the low-power state or the standby state, as shown in red in the image (**Keep awake**).




When the cellular radio starts, the radio hardware wakes up, and there is an associated battery cost. After a package is sent, the radio stays awake for another 20-30 seconds, waiting for server responses. Then the radio powers into a lower state, and then it turns off.

You need to minimize the number of times the radio is powered up. Send as much data as you can per cycle, and get a server response before the hardware is turned off. Doing all this requires some timing acrobatics, and there are several APIs, including `JobScheduler` and `Firebase JobDispatcher`, to help you. Even using these APIs, you should follow network and battery best practices.

# Network and battery best practices

Sending and receiving data packets over the network is one of the biggest users of battery. Your biggest gain in battery efficiency may come from improving how your networking interacts with the device's hardware.

There are three types of networking requests, and they need to be handled differently:

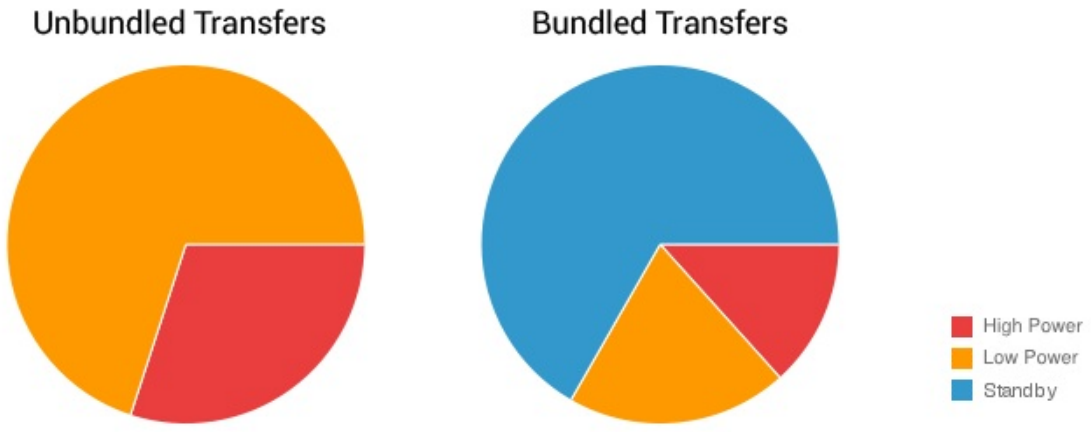
- *Do now.* These networking requests are mostly user-initiated, and the user expects a fast response. As such, there is not much to optimize here. Signing in and requesting a photo are examples of "do now" networking requests.
- *Server response.* The app makes a request to the server, and the server sends a

response. To improve resource usage for the request and reduce the size of the request and response, ask only for the data you need, and use compression.
- *Data push such as uploading analytics, saving state, or syncing.* You can optimize the timing of the push and the size of the received data.

Best practices for reducing network usage and battery consumption:

- *Never poll the server for updates.* An app that pings the server every 20 seconds, just to acknowledge that the app is running and visible to the user, keeps the radio powered on indefinitely. Keeping the radio on for this reason results in a significant battery cost for almost no actual data transfer. Instead, use a service such as Firebase Cloud Messaging, which allows the server to let your app know when new data is available.
- *Do not oversync.* Sync data only as often as you must. And ideally, sync when the phone is on Wi-Fi and plugged in.
- *Use an (exponential) back-off pattern when you sync or poll.* That is, extend the time between subsequent polls, as described in Exponential backoff. A service like Firebase Cloud Messaging will handle this task for your app.
- *Bundle requests.* Instead of sending one request at a time, bundle and send multiple requests together. This optimizes the way your app takes advantage of an active cell radio. Use the `JobScheduler` API or theFirebase `JobDispatcher` API. For more about bundling requests, see Batch Transfers and Connections.

  The diagram below shows relative mobile radio power use for unbundled data transfers (on the left) versus bundled data transfers (on the right). Red sections in the pie graphs indicate high power use, orange sections indicate low power use, and blue indicates standby state.




- *Defer non-immediate requests until the user is on Wi-Fi or the phone is plugged in.* The Wi-Fi radio uses significantly less battery than the mobile radio. See also Modifying your Download Patterns Based on the Connectivity Type.

- *Prefetch data.* Try to predict what the user might need in the next few minutes and download it ahead of time along with requested data. This requires some amount of

guessing, but if you predict correctly, you save. For example, if the user is listening to a playlist, it is likely they will listen to the next song after this one, and you can prefetch it. And for a news app, it might make sense to prefetch all the breaking news, or to prefetch the publications that the user prefers. See Prefetch Data.

- *Adapt to what the user is doing.* For example, if the user's phone has been inactive for 8 hours and suddenly becomes active, and it is morning, it is likely that the user just woke up. If the device is on Wi-Fi, sync the user's email while the device is still on Wi-Fi, instead of when the device is off Wi-Fi during the user's commute to work or school.
- *Fetch text before media.* Text requests tend to be smaller and compress better than media requests. That means text requests transfer faster, which means that your app can display useful content quickly. Use "lazy" loading of images. Cancel the image download if the user scrolls past the item.
- *Compress your data.* In general, it takes much less battery power for the CPU to compress and decompress data than it takes for the radio to transfer that same data over the network uncompressed. For more on image compression, see the Optimizing images section of this document. For a list of resources on data compression, see the Learn more section of this document.

Make your apps usable offline:

- *Bundle network requests when users are offline.* For example, let users compose messages when offline and send the mesages when online.
- *Store and cache data locally.* Caching is turned off by default for Android apps. To enable caching of all responses, use the `HttpResponseCache` class. (For details, see the #CACHEMATTERS for networking video.)

  Use Firebase Cloud Messaging in combination with local storage. You can use a database or similar structure so that it performs optimally regardless of network conditions, for example by using `SQLite` with a content provider).

- *Consider an offline-first architecture,* which initially tries to fetch data from local storage and, failing that, requests the data from the network. After being retrieved from the network, the data is cached locally for future use. This helps to ensure that network requests for the same piece of data only occur once—with subsequent requests satisfied locally. To achieve this, use a local database for long-lived data (usually `SQLite` or `SharedPreferences`).

- *The best way to save on networking performance is not to download or upload data at all.* Apps should cache content that is fetched from the network and likely to be used again. Before making subsequent requests, apps should display locally cached data. This ensures that the app is functional even if the device is offline or on a slow or unreliable network. See also ETag and HTTP 304 for conditional requests.

## Adapt to available connectivity and quality

Use the following methods to detect network status and capabilities. Use the data from these methods to tailor your app's use of the network so that your app provides timely responses to user actions:

- `ConnectivityManager` > `isActiveNetworkMetered()`
- `ConnectivityManager` > `getActiveNetworkInfo()`
- `ConnectivityManager` > `getNetworkCapabilities( Network network)`
- `ConnectivityManager` > `getNetworkInfo ( Network network)`
- `TelephonyManager` > `getNetworkType()`

Follow these practices:

- On slower connections, consider downloading only lower-resolution media, or perhaps no media at all.
- Adapt your app's behavior by checking for connectivity and network capabilities before making requests or sending data.
- If your app performs a lot of network operations, provide user settings that allow users to control your app's data habits. For example, give users control over how often your app syncs data, whether to perform uploads and downloads only when on Wi-Fi, whether to use data while roaming, etc.
- Fetch large data only if connected to a Wi-Fi network.
- Prefetch data when device is connected to Wi-Fi.

The following code snippet tests network connectivity for Wi-Fi and mobile. The code determines whether Wi-Fi or mobile network interfaces are available (that is, whether network connectivity is possible) and connected. That is, the code tests whether network connectivity exists, and whether it is possible to establish sockets and pass data.

You need the `ACCESS_NETWORK_STATE` permission to read info about a device's network connection. You need `INTERNET` permissions to connect to the internet or network.

```
ConnectivityManager cm =
    (ConnectivityManager)this.getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
boolean isConnected = activeNetwork != null &&
                      activeNetwork.isConnectedOrConnecting();
int connectionType = -1;
if(isConnected) {
    connectionType = activeNetwork.getType();
    if (connectionType == ConnectivityManager.TYPE_WIFI) {
        Log.d(DEBUG_TAG, "Mobile connected: to Wi-Fi");
    } else if (connectionType == ConnectivityManager.TYPE_MOBILE) {
        Log.d(DEBUG_TAG, "Mobile connected: to Cellular Network");
    }
}
else{
    Log.d(DEBUG_TAG, "Mobile connected: No active network connection");
}
```

See more at Managing Network Usage.

# Optimizing images

Most downloaded traffic in an app consists of images. As a result, the smaller you can make your downloadable images, the better the network experience that your app can provide for users. This section provides guidance on making image files smaller and more network-friendly.

There are a number of ways to make images faster to download. These include serving WebP images, dynamically sizing images, and using image-loading libraries.

Images for Android apps are typically JPG, PNG, or WebP format. If you support WebP, use WebP.

*WebP images*

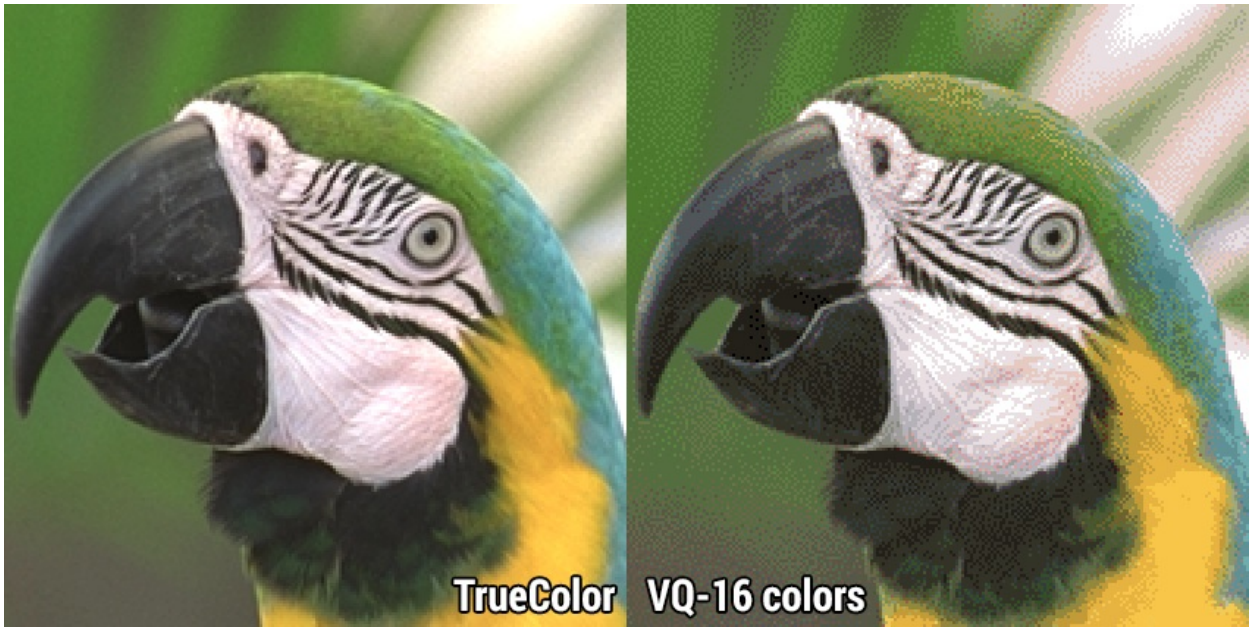
WebP is an image file format from Google. WebP provides lossy compression (as JPG does) and transparency (as PNG does), but WebP can provide better compression than either JPG or PNG.

Serve WebP files over the network to reduce image load times and save network bandwidth. A WebP file is often smaller in size than its PNG and JPG counterparts, with at least the same image quality. Even using lossy settings, WebP can produce a nearly identical image to the original. Android has included lossy WebP support since Android 4.0 (API 14) and support for lossless, transparent WebP since Android 4.2 (API 18).

*PNG images*

Optimize PNG images by reducing the number of unique colors. This is a preprocessing step that you apply to your images, and there are tools to help you. See Reducing Image Download Sizes.

In the image below, before reducing the number of unique colors (left) and after (right), you can see that there's a loss of quality. Most of the gradient colors have been replaced, imparting a banding effect to the image. Zoom in on the image to see the data loss even more clearly.




*JPG images*

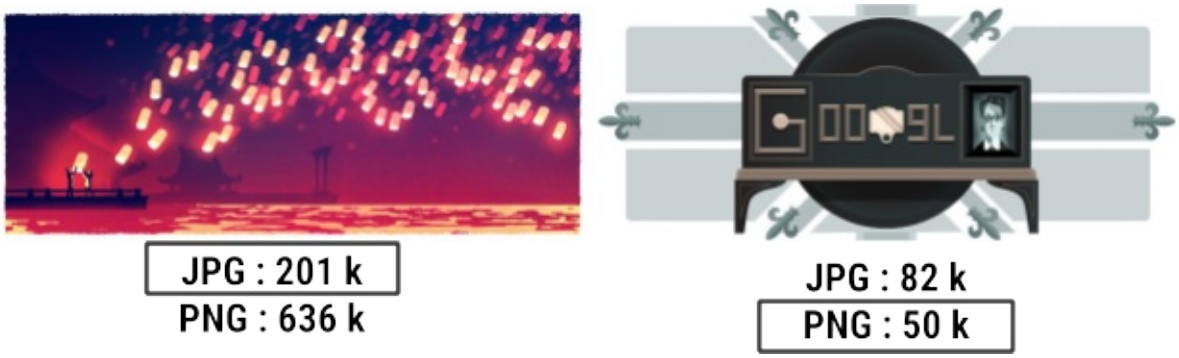
Adjust the quality of the image to the minimum required. For an average JPG there is a very minor, mostly insignificant change in *apparent* quality from 100 to 75, but a significant file size difference for each step down. This means that many images look good to the casual viewer at quality 75, but the images are half as large as they would be at quality 95. As quality drops below 75, there are larger apparent visual changes and reduced savings in file size.

## Choose the right image format

Different image formats are suitable for different types of images. JPG and PNG have very different compression processes, a nd they produce quite different results.

The decision between PNG and JPG often comes down to the complexity of the image itself. The figure below shows two images that come out quite differently depending on which compression scheme you use. The image on the left has many small details, and thus

compresses more efficiently with JPG. The image on the right, with runs of the same color, compresses more efficiently with PNG.




WebP as a format can support both lossy and lossless modes, making it an ideal replacement for both PNG and JPG. The only thing to keep in mind is that it only has native support on devices running Android 4.2.1 (API level 17) and higher. Fortunately, the large majority of devices satisfy that requirement.

Use the following algorithm to decide:

```
Do you support WebP?
        Yes: Use WebP
        No: Does it need transparency?
                Yes: Use PNG
                No: Is the image "simple" (colors, structure, subject?)
                       Yes: Use PNG
                        No: Use JPG
```

## Determine optimal quality values

There are several techniques you can use to achieve the right balance between compression and image quality. The technique described below uses scalar values, and therefore only works for JPG and WebP. Another technique takes advantage of the `Butteraugli` library, which estimates the *psychovisual* difference between two images (that is, it estimates the difference in how the user perceives the two images). The `Butteraugli` library is usable for all image formats.

## Use scalar values (JPG and WebP only)

The power of JPG and WebP comes from the fact that you can use a scalar value between 0 and 100 to balance quality against file size. The trick is finding out what the correct quality value is for your image. Too low a quality level produces a small file at the cost of image quality. Too high a quality level increases file size without providing a noticeable benefit to the user.

The most straightforward solution is to pick some non-maximum value, and use that value. However, be aware that the quality value affects every image differently. While a quality of 75%, for example, may look fine on most images, there may be some cases that do not fare as well. You should make sure to test your chosen maximum value against a representative sample of images. Also, make sure to perform all of your tests against the original images, and not on compressed versions.

For large media apps that upload and re-send millions of JPGs a day, hand-tuning for each asset is impractical. You might address this challenge by specifying several different quality levels, according to image category. For example, you might use 35% as the quality setting for thumbnails, since a smaller image hides more compression artifacts.

## Serve sizes

It is tempting to keep only a single resolution of an image on a server. When a device accesses the image, the server serves the image at that one resolution and leaves downscaling to the device.

This solution is convenient for you, but it might force the user to download more data than they need. Instead, store multiple sizes of each image and serve the size that is most appropriate for each use case. For example, for a thumbnail, serve an actual thumbnail image instead of serving a full-sized version of the image that is downscaled on the device.

Serving sized images is good for download speed, and it's less costly for users with limited or metered data plans. When you serve sized images, the images take less space on the device and in main memory. For large images, such as images with 4K resolution, this approach also saves the device from having to resize images before loading them.

To implement this approach, you need a backend image service that can provide images at various resolutions with proper caching. Services exist that can help with this task. For example, Google App Engine comes with image-resizing functionality already installed.

For more about image formats and how to choose the right ones, see Reducing Image Download Sizes.

## Dynamically resize images

When you download images from a server, request the size appropriate for the device. Adjust image size and quality based on the network bandwidth and quality. Dynamically resizing images in this way reduces transferred data size, improves download speed, and reduces the amount of memory needed on the device.

## Use image loading libraries

Your app should not download any image more than once. Image loading libraries such as `Glide` and `Picasso` fetch the image once, cache it, and provide hooks into your views to show placeholder images until the actual images are ready. Because images are cached, these libraries return the local copy of the image the next time an image is requested.

# Text data compression

In addition to optimizing images, you may be able to reduce the size of text-heavy pages. Start by testing how long a page takes to load on a mobile device, and if it is a problem, consider these remedies:

- *Make the page smaller.* Edit the page to use fewer words. Remove content that is not relevant. Break the page up into smaller pages.
- *Minify.* CSS and Javascript minifiers are powerful, easy to use, and fit into existing build pipelines.
- *Compress.* Ensure that your server has GZIP compression turned on. Generate better compressed GZIP data offline using `Zopfli` or `7-zip`.

See Text Compression for Web Developers.

# Serializing data

Serialization is the process of converting structured data into a format that can be stored and sent over the network. At the destination, the original data is reconstructed. JSON and XML are example serialization formats that are human-readable, but bulky and slow.

Use `FlatBuffers` to create a schema that describes your data, then compile the data into source code that can serialize and deserialize your data. It's smaller, faster, and less painful than using JSON or XML.

See the Serialization with Flatbuffers video and the `FlatBuffers` documentation.

# Related practical

The related exercises and practical documentation is in Optimizing network, battery, and image use.

# Learn more

Network:

- Reduced data cost for billions
- Connectivity for billions
- Network Performance 101
- Managing Network Usage
- Optimizing Networking Request Frequencies
- Effective Network Batching
- Effective Prefetching
- Efficient Data Transfers video series
- #CACHEMATTRS for networking
- Adapting to Latency
- Network Profiler
- Firebase Cloud Messaging

Battery:

- Transferring Data Without Draining the Battery: in-depth docs
- Optimizing Battery Life: in-depth docs
- Battery Performance 101
- Battery Drain and Wake Locks
- Battery Drain and Networking

WebP:

- Create WebP Images
- WebP website

Compression:

- Reducing JPG File Size (Colt McAnlis on Medium)
- Image Compression for Web Developers (good, comprehensive intro)
- Reducing Image Download Sizes
- Text Compression for Web Developers
- Understanding Compression, Data Compression for Modern Developers by Colt McAnlis and Aleks Haecky
- Compressor Head series

Flatbuffers:

- Serialization with Flatbuffers
- `FlatBuffers` documentation

# 5.1: Languages and layouts

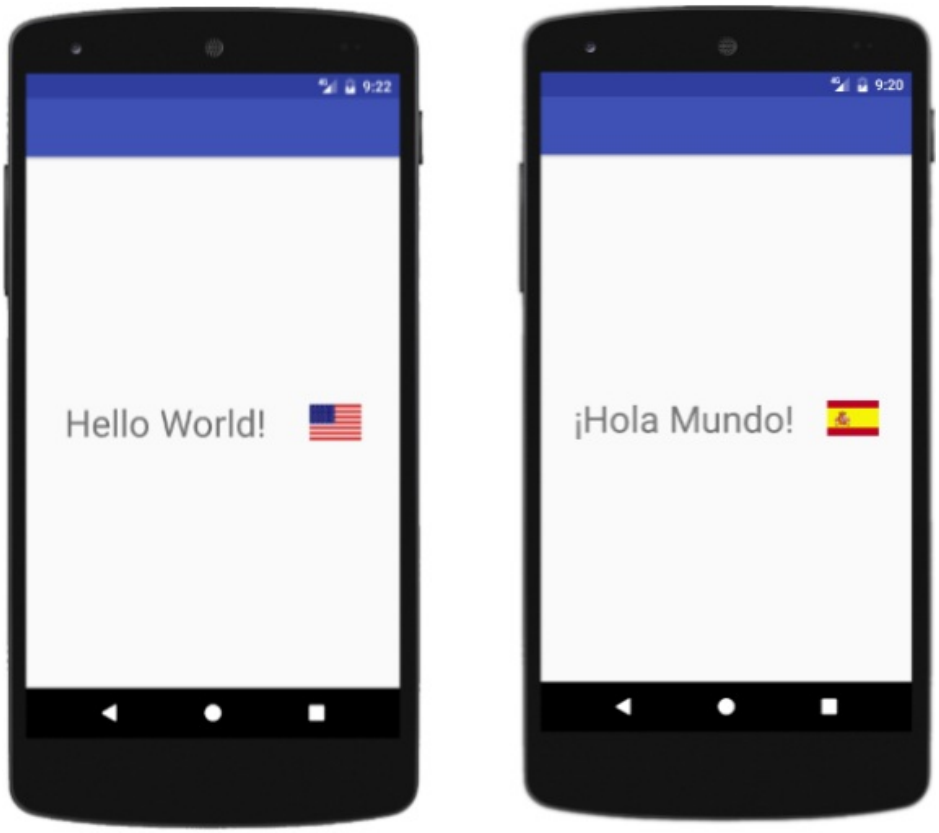
**Contents:**



Android runs on devices in different languages, and in different places (referred to as *locales* ). To reach the most users, your app should display information in ways appropriate to those locales. For an app developer, *localization* is about tailoring the entire app experience to a user's language and locale.

An app can include sets of resource directories customized for different languages and locales. When a user runs the app, Android automatically selects and loads the resources that best match the device.

(This chapter focuses on localization and locale. For a complete description of resource-switching and all the types of configurations that you can specify — screen orientation, touchscreen type, and so on — see Providing Alternative Resources.)




# Understanding language and locale settings

Android users can choose both the language and the locale for their devices in the Settings app. For example, a user can choose English (U.S.) for the United States locale, or English (U.K.) for the United Kingdom locale. You can add to your app support for dialects of a language for different locales, such as Français (France) and Français (Canada).

To localize an app for different countries, add a language for each country if possible. For each language, obtain a translation for *all* of the app's text—content, menu choices, navigation elements, notifications, alerts, and messages. You may also have to alter your layouts to accommodate languages that take up more space or read from right to left. If the user chooses a different language than the ones supported by your app, your app will use the default language. Your app also should handle numbers, dates, times, and currencies in formats appropriate to different locales.

This chapter focuses on language and layout, and the following chapter describes locale-specific issues such as formatting dates and currencies.

# Preparing an app for localization

To decide which languages to support in your app, you need to first know who will be using the app—which locales, and which languages *in* those locales. You may want to pick certain locales to receive your app first, identify the languages required for those locales, and add more languages over time as you enhance your app to support more locales.

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

1. Extract all strings as resources into `strings.xml`, including text content, menu choices, navigation tabs, messages, and any other UI elements that use text.
2. Use Android Studio's Translations Editor to add a language, and add translations for the strings. You may want to use a translation service, which sends back a `strings.xml` file with the translations.
3. If you are adding a right-to-left (RTL) language, change your layouts to add a "start" and "end" attribute for each "left" and right" attribute.
4. To you test your app, change the device's language in the Settings app before each test run.
5. When making your app available on Google Play, provide your app's store description, campaign ads, and in-app purchase products in multiple languages.

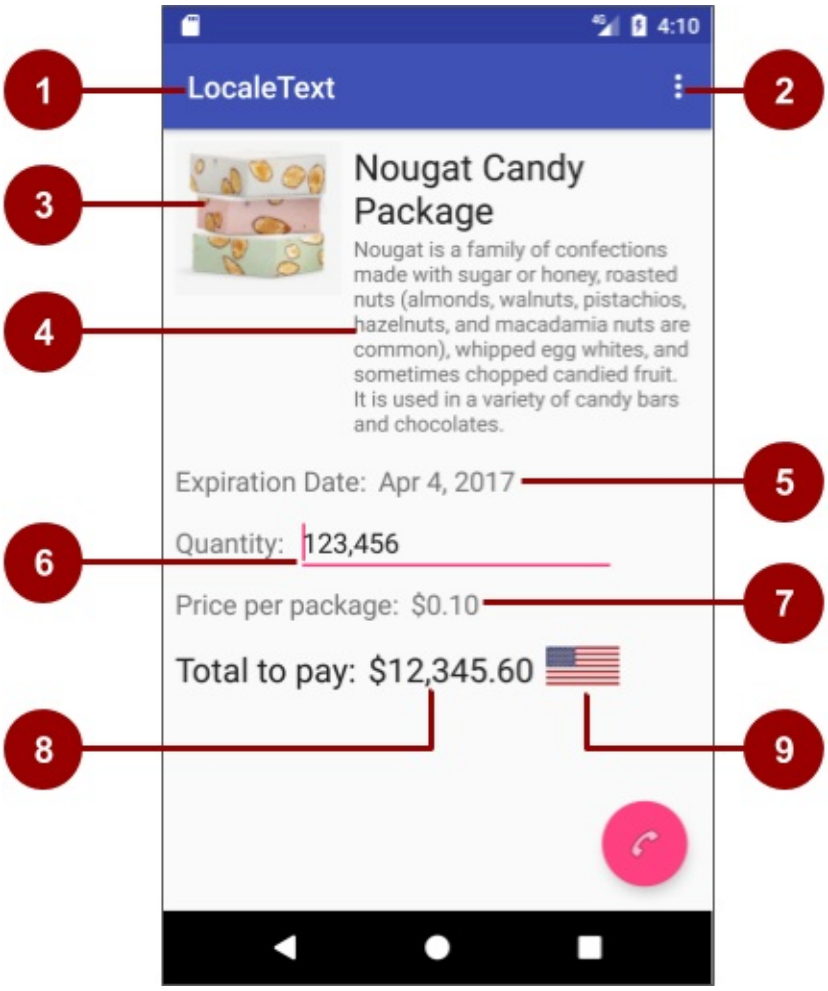
**Tip**: For details on getting a translation service and preparing an app for Google Play, see Translate & localize your app in the Play Console.

## Identifying UI elements to translate and format

In the app itself you need to identify *at a minimum* :

- All the UI text that needs to be translated, including menu choices, headings, and labels for entry fields.
- All the elements that need locale formatting, such as dates, times, numbers, and currencies.
- Layout changes to accommodate each language, such as leaving enough space for text elements to grow or shrink.
- Images you should replace or adjust in the layout for each locale.

You may also need to identify media resources, such as audio and video, that need captions or subtitles.




For example, in the figure above:

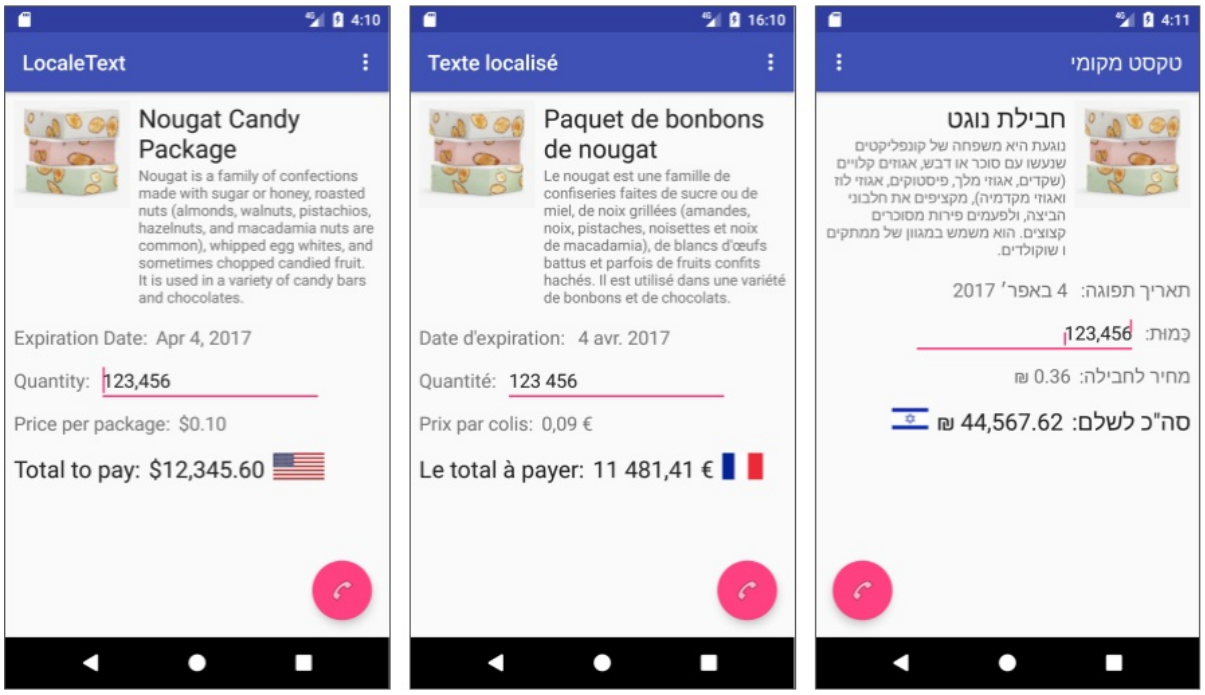
1. **App name:** The choice of translating the app name is up to you. Many apps add translated names so that the users in other languages understand the app's purpose and can find the app in Google Play in their own languages. Apps that use brand names for app names don't translate the app name.
2. **Options menu:** Translate all choices in the options menu.
3. **Images:** Adjust the layout so that the image appears on the opposite side of the screen for a right-to-left (RTL) language, such as Hebrew or Arabic. You may also want to use a different image for a specific language or locale based on cultural preference.
4. **Text:** All text in the UI should be translated. The sizes of text elements should be flexible to allow them to grow larger for a language that takes up more space.
5. **Date format:** Dates appear in different formats depending on the locale.
6. **Number format:** Numbers appear with different punctuation marks (such as commas or periods) depending on the locale.
7. **Currency format:** Each locale may have its own currency. It is up to you to decide how many different currencies you want to support in your app.
8. **Currency with amount calculations:** If you provide different currencies, you must also

use currency exchange rates.

9. **Image tied to a locale:** You can change an image depending on the locale. In this example, the locale country's flag appears.

**Tip**: The floating action button at the bottom of the screen in this example does not need to be adjusted for an RTL language, because it "floats" on the screen and is adjusted automatically by Android's layout mirroring feature.

The following figure shows the same app in the English language and U.S. locale (left), in French in the France locale (center), and in Hebrew, an RTL language, in the Israel locale (right).




## Best practices for localizing text

- Keep text separated from the rest of your app. Never embed text in code.
- To create string resources for other languages, use the Android Studio Translations Editor, which automatically places strings.xml files into appropriately named directories within the app project's *project_name* `/app/src/main/res` directory.
- Don't concatenate pieces of text to make a sentence or phrase. The pieces may not be in the proper order in the translated version of the 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. Format strings are a convenient way to build up one string that has variable content. For details on using format strings, see Android SDK Quick Tip: Formatting Resource Strings. To ensure that format strings are not translated, used the `<xliff:g>` placeholder tag to mark it, as described in the "Mark message parts that should not be translated" section of Localizing with Resources.

## Addressing design and layout issues

Fitting design elements around translated text is a challenge. The elements of the design may no longer fit around translated titles, and menus may appear to be wider or longer, because the translated words take up more or less space than the original words.

As you create your layouts, make sure that any UI elements that hold text are designed generously. It's good to allow more space than necessary—at least 30% more space. For example, if a heading is not wide enough, it may require two lines as shown below. The text body may shrink or grow depending on the language. Use layout attributes to constrain UI elements to other elements, so that if a text element grows in size, it does not overlap another element.


**Tip**: If you design your UI carefully, you can typically use a single set of layouts for all the languages you support. While you can also use the Layout Editor to create alternative layouts for specific locales, alternate layouts make your app harder to maintain.

## Communicating effectively

To be successful, apps need to communicate effectively. Translation may not be enough, and you may also need to adapt your app to local culture and customs.

Be especially careful with brand names and slogans. Get the writing right first, *before* translation. Consult a translator who knows the language. Keep in mind the following issues:

- Plurals and genders are handled differently in different languages.
- If you use an acronym, provide the full term as well so that it can be translated.
- Trademarked names are rarely if ever translated, unless they are also registered in the country you are targeting with the translation.
- Ensure correct grammatical structure and proper punctuation in the original language.
- Be consistent with word usage.
- Translation services often use memory to translate the same words and phrases from one project to the next. Consider reusing words and phrases that were translated previously.
- Avoid humor, which rarely translates well and may unintentionally offend someone in

another culture.
- Avoid jargon, regional phrases, and metaphors. For example, users may not know enough about American baseball to understand "knocking it out of the park" or a "grand slam."

The following are tips on how to prepare media, including graphics and images:

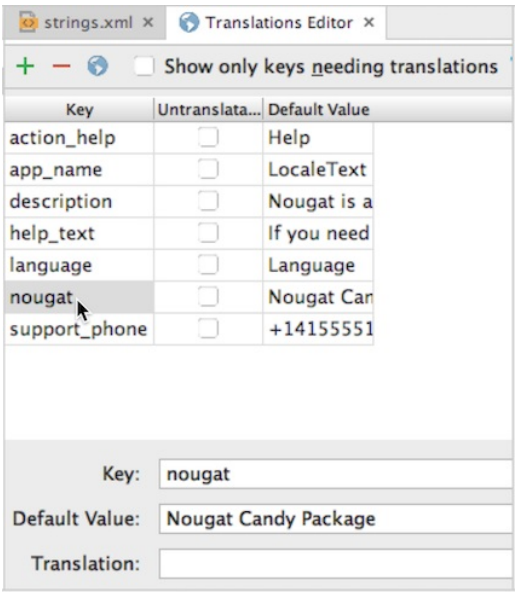
- Be aware of cross-cultural differences, as not all images and symbols carry the same meaning across borders. Maps, flags, location names, and cultural, historical, and political references may be offensive to users in another country.
- Do not include text within images because it is not easily translated.
- Include comments in the strings.xml file that you send out for translation, in order to describe the context in which each string is used. This information will be invaluable to the translator and result in better quality translation.

# Using the Translations Editor

You can use Android Studio's **Translations Editor** to add a language to your app and create the string resources for the language.

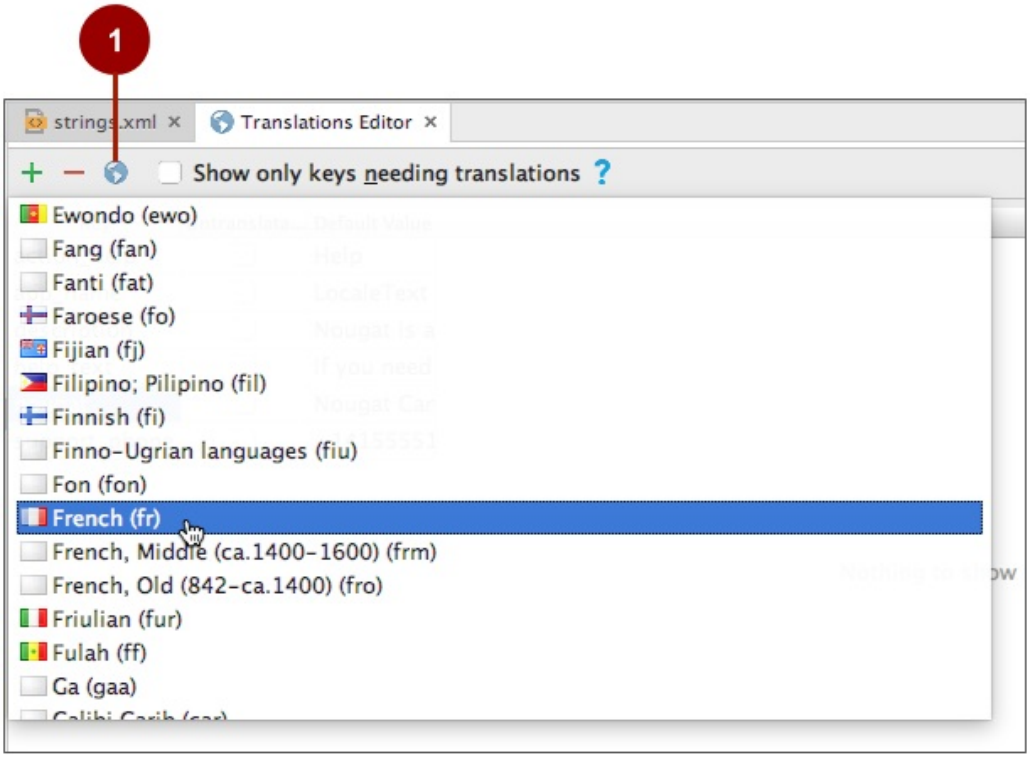
To use the **Translations Editor**, open the `strings.xml` file, and click the **Open editor** link in the top right corner, or you can use a shortcut: right-click the `strings.xml` file and select **Open Translations Editor**.

The **Translations Editor** opens with a row for each resource key and its default value.




## Adding a language to the app

To add a language, click the globe button in the top left corner of the **Translations Editor** ("1" in the following figure), and select the language in the dropdown menu.

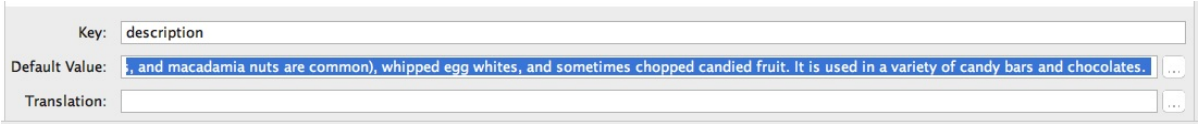



After you choose a language, a new column with blank entries appears in the Translation Editor for that language, and the keys that have not yet been translated appear in red.

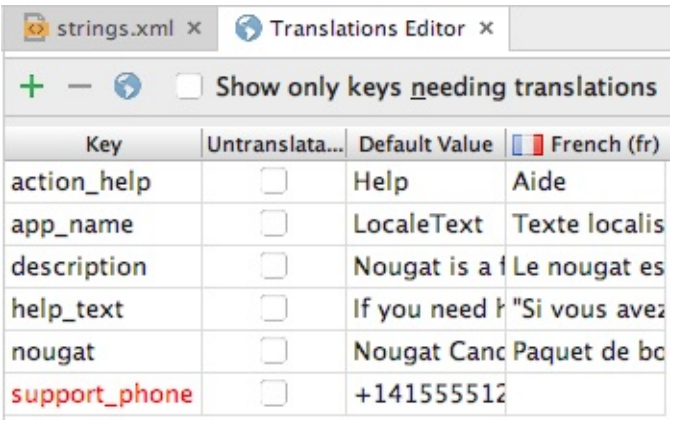
## Translating strings

To add translations to the string resource values, follow these steps:

1. Select a key. The **Key**, **Default Value**, and **Translation** fields appear at the bottom of the **Translations Editor** window.
2. For each key in the **Key** column of the **Translations Editor**, select the blank cell in the language column on the same row, and enter your translation in the empty **Translation** field at the bottom of the **Translations Editor** window.



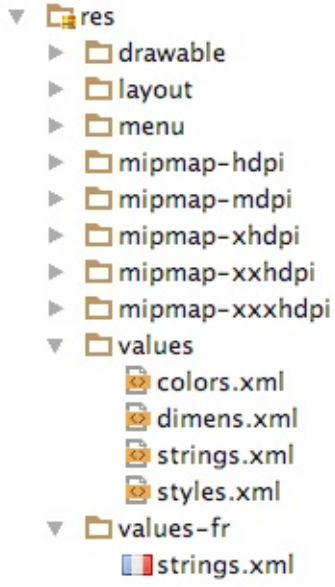
The translation appears in the language column for the key.




3. Select the **Untranslatable** checkbox for any key that should *not* be translated. For example, `support_phone` in the above figure should not be translated, and remains in red until you click **Untranslatable**. You should mark all untranslatable text so that the translator doesn't change it. The Translation Editor automatically adds the `translatable="false"` attribute to the string resource.

   To show only the keys that require translation, select the **Show only keys needing translation** checkbox option at the top of the **Translation Editor** window.

4. In the far left column of Android Studio, switch to **Project: Project Files** view, and expand the **res** directory. The Translations Editor created a new `values` directory for the language: `values-fr` (for French). The `values` directory is the default directory with colors, dimensions, strings, and styles, while the `values-fr` directory has only string resources translated into French.




An app can include multiple `values` directories, each customized for a different language and locale combination. When a user runs the app, Android automatically selects and loads the `values` directory that best matches the user's chosen language and locale.

# Adjusting layouts for RTL languages

Latin (or Roman) script for most Western and Central European alphabets, for example, is read from left to right. Devanagari script, the alphabet used for over 120 languages, including Hindi, is also read from left to right. However, Arabic script, used for several languages of Asia and Africa, reads right to left (RTL). Hebrew and Persian (Farsi) scripts are also RTL.

If you're distributing to countries where RTL languages are used, consider implementing support for RTL layouts.

## Using RTL layout mirroring

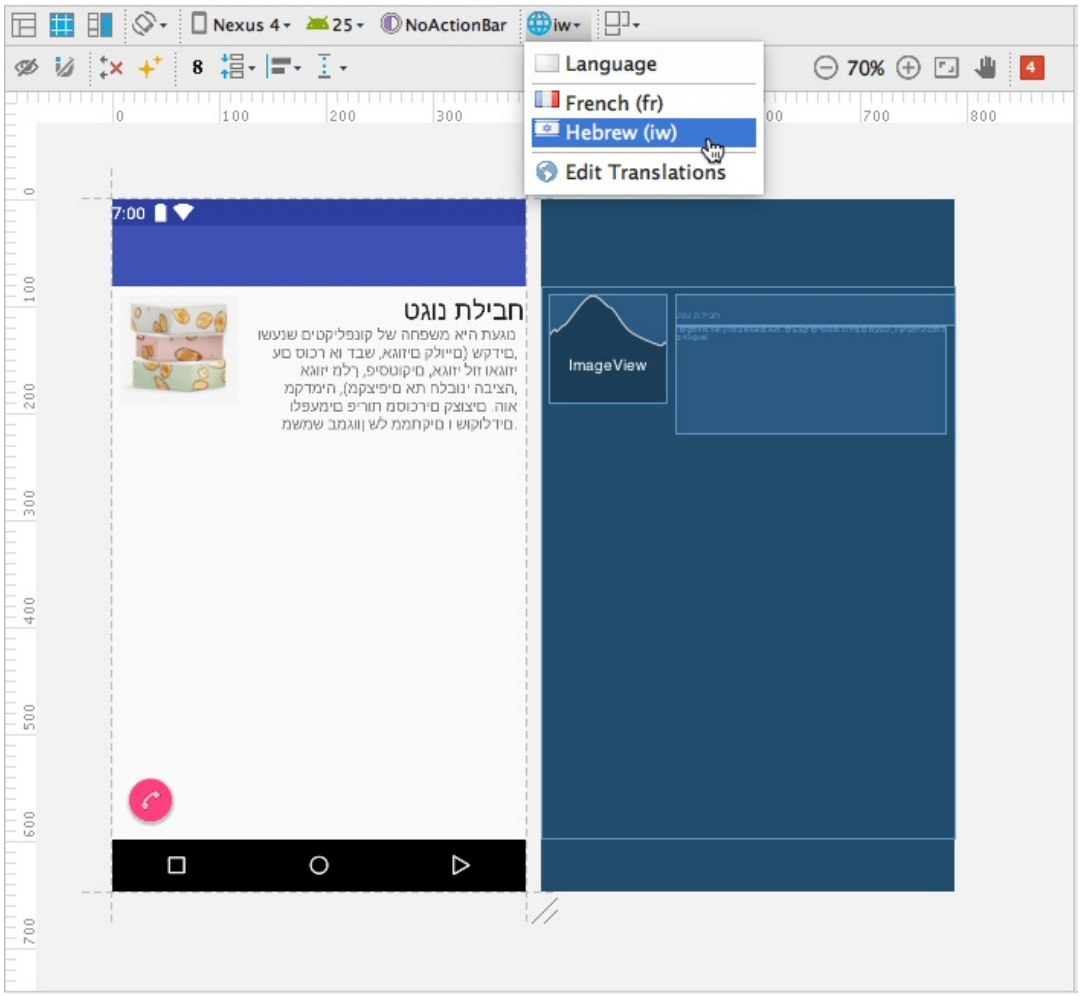
Android 4.2 and newer versions provide full native support for RTL layouts, including layout mirroring. *Layout mirroring* redraws the layout for a right-to-left orientation for RTL languages. As a result, UI elements are all automatically moved to the opposite side of the screen.

**Tip**: Use layout mirroring rather than alternate layouts for RTL languages, in order to avoid the overhead of managing the layouts. While you can build alternate layouts for languages, you may also want to build other layouts to support different screen sizes. (See Supporting Multiple Screens for a detailed explanation, and Build a UI with Layout Editor for instructions on using Android Studio to create the layouts.) RTL layout mirroring reduces the need for multiple layouts.

For each new project, Android Studio includes a declaration in the element in the AndroidManifest.xml file to support RTL layout mirroring:

```
android:supportsRtl="true"
```

After adding the RTL language (such as Hebrew), you can see a preview of layout mirroring. Click the **Design** tab, and choose a language from the **Language** menu at the top of the Layout Editor. The layout reappears with a preview of what it will look like in that language.




Changing the language can affect the layout. In this case, the RTL text is on the opposite side of the layout, as you would expect with layout mirroring. The floating action button has also been moved to the left side of the screen, as it should be for an RTL language. However, since the ImageView is still constrained to the left margin, it didn't move to the other side of the screen, as it should in an RTL layout.

## Adjusting the left/right positioning attributes to start/end

To adjust the layout so that it fully supports an RTL language, you need to do one of the following:

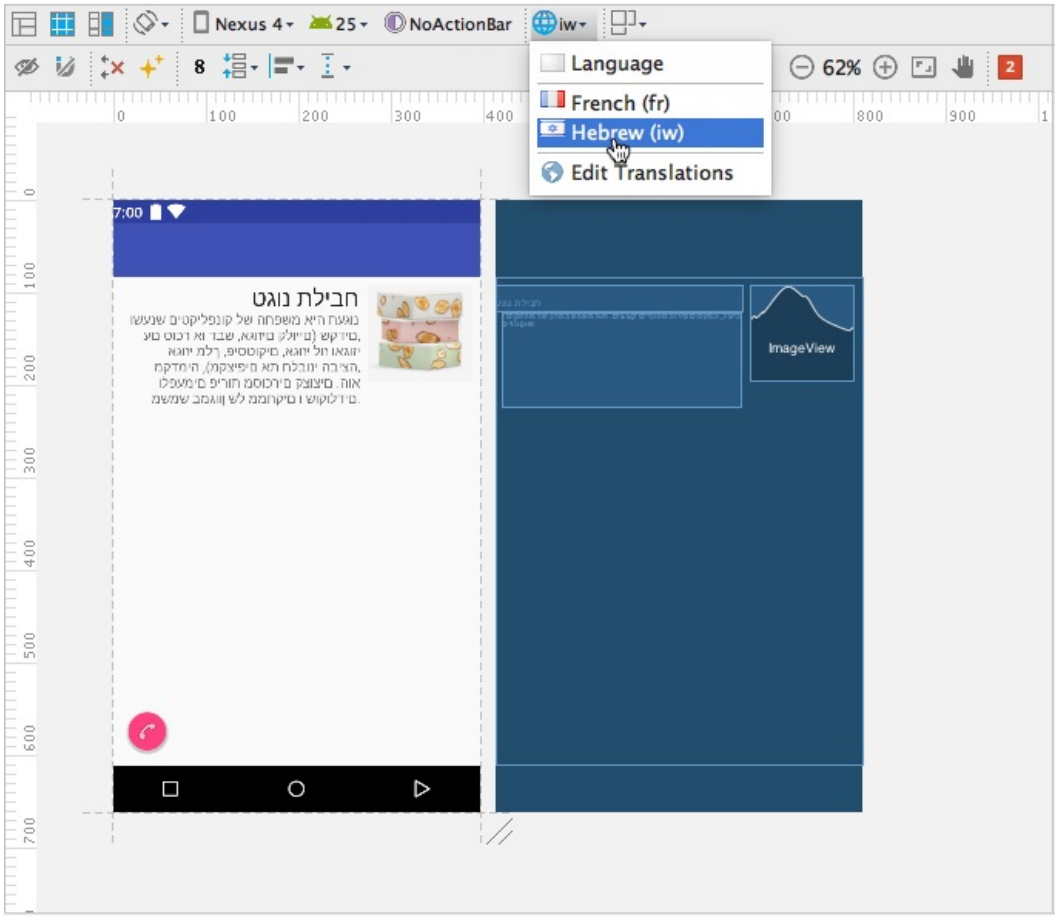
- If you are targeting your app to Android 4.2 (the app's `targetSdkVersion` or `minSdkVersion` is 17 or higher), use "start" and "end" layout attributes in place of "left" and "right" attributes. For example, replace `android:layout_marginLeft` with `android:layout_marginStart`.
- If you want your app also to work with versions earlier than Android 4.2 (the app's `targetSdkVersion` or `minSdkVersion` is 16 or less), add "start" and "end" layout attributes *in addition to* "left" and "right." For example, use both `android:layout_marginLeft` and `android:layout_marginStart`.

With the `ConstraintLayout` root view, add "start" and "end" constraint attributes to any "right" and "left" constraint attributes. For example:

```
app:layout_constraintLeft_toRightOf="@+id/product_image"
app:layout_constraintStart_toEndOf="@+id/product_image"
```

The `constraintStart_toEndOf` attribute is the same as the `constraintLeft_toRightOf` attribute for an LTR language, but is the reverse position for an RTL language.

When you preview the layout after adding the "start" and "end" attributes, the layout shows the image on the right side of the screen (the "start" side), and the text views are constrained to its left ("end") side.



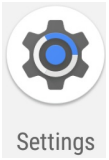

There are many other attributes with "right" and "left" positions. For example, in a RelativeLayout, you may be using the `android:layout_alignParentLeft` attribute. To support an RTL language, you would add the `android:layout_alignParentStart` attribute.

Margins and padding in text views are also important. The `android:layout_marginStart` and `android:layout_marginEnd` attributes are key to setting a proper margin for RTL languages.

# Testing languages and locales with apps

To test language and locale settings with an app, use an Android device or emulator. Choose both the language and the locale in the Settings app.

Users of Android 7.0 (Nougat) and newer versions can select multiple languages on a single device and assign preferences to them. If you don't provide language resources for the top preference in your app, the system switches to the next language preference in the list before choosing the default language resource.

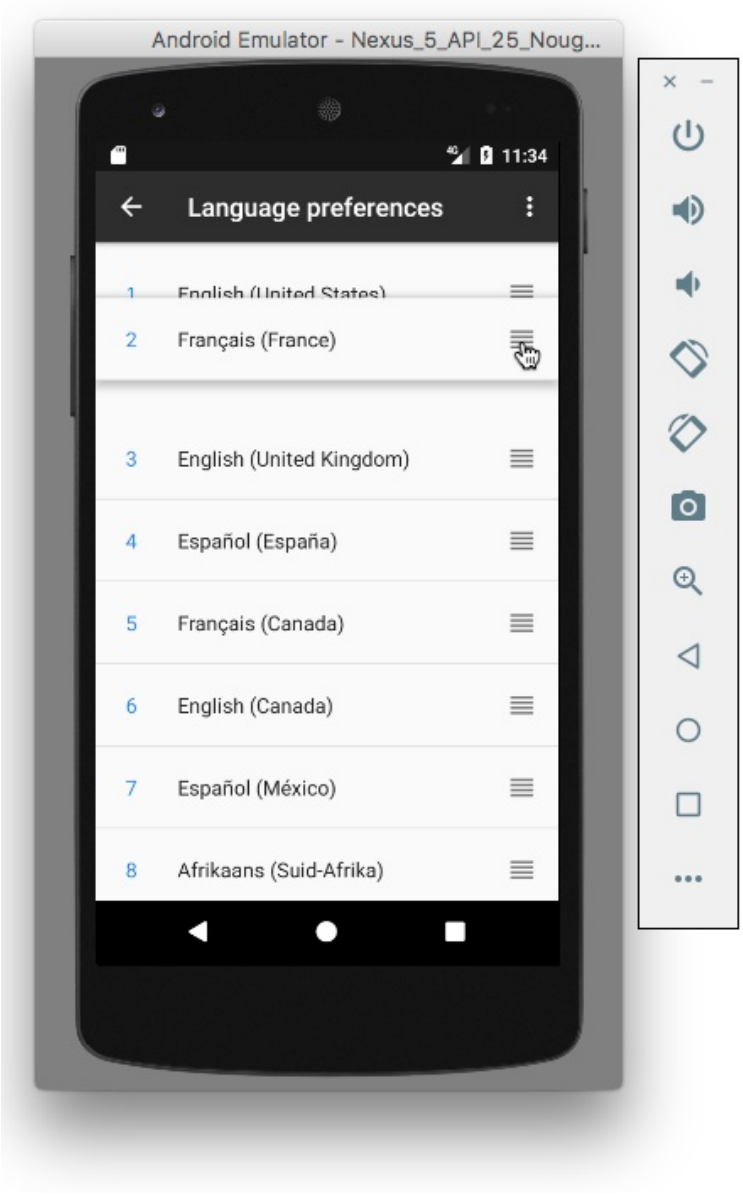
To switch the preferred language in your device or emulator, open the Settings app. If your Android device is in another language, look for the gear icon: 


Follow these steps in the Settings app for Android versions previous to Android 7.0 (Nougat):

1. Choose **Languages & input > Language**. The **Languages** choice is the first choice in the list for any language.
2. Select a language and locale such as **Français (France)**.

Follow these steps in the Settings app for Android 7.0 (Nougat) or newer versions:

1. Choose **Languages & input > Languages**. The **Languages** choice is the first choice in the list for any language.
2. To add a language not on the list, click **Add a language**, select the language (for example, **Français**), and then select the locale (such as **France**).
3. Click the move icon on the right side of the **Language preferences** screen, and drag the language to the top of the list. The language takes effect immediately after you drag




it to the top of the list.

**Tip**: When using the Settings app, be sure to remember the globe icon for the **Languages & input** choice, so that you can find it again if you switch to a language you do not understand.

# Related practical

The related practical documentation is Using resources for languages.

# Learn more

Android developer documentation:

- Supporting Different Languages and Cultures

- Localizing with Resources
- Localization checklist
- Language and Locale
- Testing for Default Resources

Material Design: Usability - Bidirectionality

Android Developers Blog:

- Android Design Support Library
- Native RTL support in Android 4.2

Android Play Console: Translate & localize your app

Video: Welcome to the World of Localization

Other:

- Android SDK Quick Tip: Formatting Resource Strings
- Language Subtag Registry - IANA
- Country Codes
- ISO 3166 Country Codes
- Android locale codes and variants

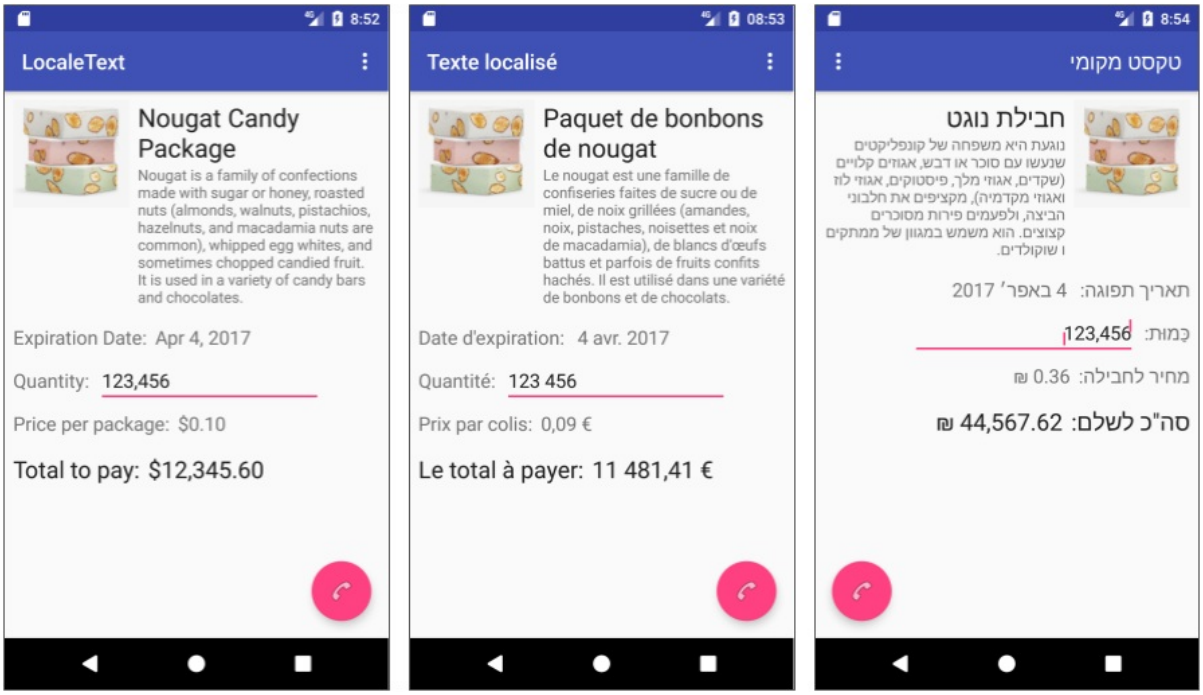
# 5.2: Locales

**Contents:**



Users run Android devices in many different parts of the world. To provide the best experience for users in different regions, your app should handle not only text but also numbers, dates, times, and currencies in ways appropriate to those regions.

When users choose a language, they also choose a locale for that language, such as English (United States) or English (United Kingdom). Your app should change to show the formats for that locale: for dates, times, numbers, currencies, and similar information. Dates can appear in different formats (such as *dd* / *mm* / *yyyy* or *yyyy* - *mm* - *dd* ), depending on the locale. Numbers appear with different punctuation, and currencies sometimes vary by locale.




Hardcoding your formats based on assumptions about the user's locale can result in problems when the user changes to another locale. For example, if your app guarantees shipping by 5/4/2017, that means May 4, 2017, in the U.S., but it means April 5, 2017, in the

U.K.

Android provides classes and methods you can use to apply the format of the user-chosen locale, or to use a format from another locale. Store all data for the app in a default format, and use these classes to localize the data for display.

For some aspects of localization you might need to add non-default resources, as described in Adding resources for different locales.

# Formatting the date and time

As a rule, store all data for the app in the format that's appropriate for the default language and locale, and then use a user-chosen format as needed.

Android provides the `DateFormat` class and methods to apply the date format of the user-chosen locale. `DateFormat` is an abstract class that can format a `Date` object to a string, and parse a date string, in a language-independent manner. For example, you can use the `DateFormat.getDateInstance()` method to get the format for the user's selected language and locale, and the `DateFormat.format()` method to prepare a formatted string for display:

```
final Date myDate = new Date();
// Format the date for the locale.
String myFormattedDate = DateFormat.getDateInstance().format(myDate);
// Display the formatted date.
TextView dateView = (TextView) findViewById(R.id.date);
dateView.setText(myFormattedDate);
```

When you use `DateFormat`, your code is independent of the locale conventions for formatting months, days of the week, or even the calendar format (lunar or solar). Use the following `DateFormat` factory methods:

- To get the date format for the user-chosen locale, use `getDateInstance()`. When used with `DateFormat.format`, the result is a string such as "Dec 31, 2016" (for the U.S. locale).
- To get the date format for any given `Locale` object (which can be different from user-chosen locale), use `getDateInstance( int style , Locale aLocale)`. The `style` is a formatting style such as `SHORT` for "M/d/yy" in the U.S. locale.
- To get the time format for a specific country, use `getTimeInstance()`. The result is a time format such as "4:00:00 PM" (for the U.S. locale).
- To get a combined date and time format, use `getDateTimeInstance()`. The result is a date/time format such as "Dec 31, 2016 4:00:00 PM" (for the U.S. locale).

To help you control how much space the localized date or time takes up, pass options (such as `style` with `getDateInstance()`) to these factory methods to control the length of the result. The exact result depends on the locale, but generally:

- `SHORT` is a numeric format, such as 12.13.52 or 3:30pm.
- `MEDIUM` abbreviates the month, such as Jan 12, 1952.
- `LONG` is the long version, such as January 12, 1952, or 3:30:32pm
- `FULL` is completely specified, such as Tuesday, April 12, 1952 AD, or 3:30:42pm PST.

**Tips**:

- To create strings for things like elapsed time and date ranges, days of the week, months, and AM/PM for time, use the `DateUtils` class.
- To write code that works with dates and a hybrid calendar that supports both the Julian and Gregorian calendar systems, use the `GregorianCalendar` class.

# Formatting numbers

Numbers appear with different punctuation in different locales. In U.S. English, the thousands separator is a comma; in France, the thousands separator is a space; and in Spain, the thousands separator is a period. The decimal separator also varies from period to comma.

`NumberFormat` is the abstract base class for all number formats. Your code can be completely independent of the locale conventions for decimal points and thousands separators.

For example, use the `NumberFormat.getInstance()` method to return a number format for the user-selected language and locale, and the `NumberFormat.format()` method to convert the formatted number to a string:

```
// Default quantity is 1.
int myQuantity = 1;
// Get the number format for this locale.
NumberFormat numberFormat = NumberFormat.getInstance();
// Format the number.
String myFormattedQuantity = numberFormat.format(myQuantity);
// Display the formatted number.
TextView quantityView = (TextView) findViewById(R.id.quantity);
quantityView.setText(myFormattedQuantity);
```

**Tip**: If you are formatting multiple numbers, it is more efficient to get the format and use it multiple times (such as `numberFormat`), so that Android does not need to fetch the user-chosen language and locale multiple times.

To format a number for a locale other than the device's current locale, specify the locale in the `NumberFormat.getInstance()` method. Android provides `Locale` constants for many countries:

```
NumberFormat numberFormat = NumberFormat.getInstance(Locale.FRANCE);
```

The `numberFormat` returned is for the specified locale, so that you can format the number without changing the user's chosen locale.

The `NumberFormat.getInstance()` method returns a general-purpose number format for the user-chosen locale. This is the same as `getNumberInstance()`. Use the following for more specific number formats:

- `getIntegerInstance()`: Returns an integer number format for the locale.
- `getPercentInstance()`: Returns a percentage format for the locale.
- `getCurrencyInstance()`: Returns a currency format for the locale. Currency formats are described in the next section.

Use `NumberFormat.parse()` to parse a given string, such as `EditText` field input, and use `intValue()` to return an integer:

```
// Parse contents of string in EditText view and return a number.
try {
    // Use numberFormat for the current locale.
    myQuantity =
              numberFormat.parse(qtyInput.getText().toString()).intValue();
} catch (ParseException e) {
    e.printStackTrace();
}
```

# Formatting currencies

Currencies sometimes vary by locale. Use the `NumberFormat` class to format currency numbers. The `getCurrencyInstance()` method returns the currency format for the user-selected language and locale, and the `format()` method applies the format to create a string, as shown in the following code:

```
// Fixed price in U.S. dollars and cents: ten cents.
double myPrice = 0.10;
// Get locale's currency.
NumberFormat currencyFormat = NumberFormat.getCurrencyInstance();
// Use the currency format for the locale.
String myFormattedPrice = currencyFormat.format(myPrice);
// Show the string.
TextView localePrice = (TextView) findViewById(R.id.price);
localePrice.setText(myFormattedPrice);
```

In the above code, the `NumberFormat` `format()` method applies the currency format of the locale to the `double` `myPrice` to create the `myFormattedPrice` string.

Currency exchange rates are in constant fluctuation. To offer multiple currencies in an app that calculates amounts, you may need to write code to retrieve the latest exchange rates every day from a web service.

**Tip**: You can quickly grab the latest exchange rate for a given currency by using the Google Finance Converter.

# Retrieving and using the locale

A `Locale` object represents a specific geographical, political, or cultural region. You can retrieve the country code of the user-chosen locale, compare the result to the locales supported by the app, then take actions depending on the locale.

## Using the country/region code

For example, you may want to support three currencies in your app for three locales (U.S. dollars for the U.S., euros for France, and new shekels for Israel), and default to U.S. dollars for all other locales. To get the country/region code for the user-chosen locale, use `Locale.getDefault().getCountry()`:

1. Use the `Locale.getDefault()` method to get the *value* of the user-chosen locale.
2. Use the `Locale.getCountry()` method to get the *country/region code* for the user-chosen locale.

In the following snippet, the country/region code `FR` is for France, and `IL` is for Israel. If the country/region code matches a country whose currency your app supports ( `FR` or `IL` ), then use the currency format for that country; otherwise, use the default currency format:

```
String myFormattedPrice;
// Check if locale is supported for currency.
String deviceLocale = Locale.getDefault().getCountry();
if (deviceLocale.equals("FR") || deviceLocale.equals("IL")) {
    // Use the currency format for France or Israel.
    myFormattedPrice = currencyFormat.format(myPrice);
} else {
    // Use the currency format for U.S.
    currencyFormat = NumberFormat.getCurrencyInstance(Locale.US);
    myFormattedPrice = currencyFormat.format(myPrice);
}
```

The code tests only for the `FR` or `IL` locales, because all other locales use the default currency format (U.S. dollars).

## Using a locale constant

The `Locale` class provides a number of convenient constants that you can use to create `Locale` objects for commonly used locales. For example, `Locale.US` creates a `Locale` object for the U.S. The following line in the above snippet uses `Locale.US` to apply the U.S. currency if the locale is not `FR` or `IL`:

```
currencyFormat = NumberFormat.getCurrencyInstance(Locale.US);
```

If a constant doesn't exist for a locale that you want to use, construct a `Locale` object using `Locale.Builder` (introduced in API Level 21). Once you've created a `Locale`, you can query it for information about itself:

- Use `getCountry()` to get the country (or region) code of the `Locale` object.
- Use `getLanguage()` to get the language code.
- Use `getDisplayCountry()` to get the name of the country suitable for displaying to the user.
- Use `getDisplayLanguage()` to get the name of the language suitable for displaying to the user.

**Tip**: Create a `Locale` object if you want to change the app's language dynamically from within the app at runtime.

# Adding resources for different locales

In addition to formatting information for specific locales, you can add language dialects for a locale, and change images, colors, dimensions, and styles for different locales.

An app can include multiple resource directories, each customized for a different language and locale. When a user runs the app, Android automatically selects and loads the resource directories that best match the user's chosen language and locale.

For example, the strings in the strings.xml file in the `values-fr` directory are in French. If the user chooses French as the language for the device, the French `strings.xml` file is used rather than the `strings.xml` file in the default `values` directory.

The same is true for other `values` files, such as `colors.xml` and `dimens.xml`. If the user chooses French as the language for the device, and if French `colors.xml` and `dimens.xml` files exist in `values-fr`, then these French resources are used rather than the resources in the default `values` directory.

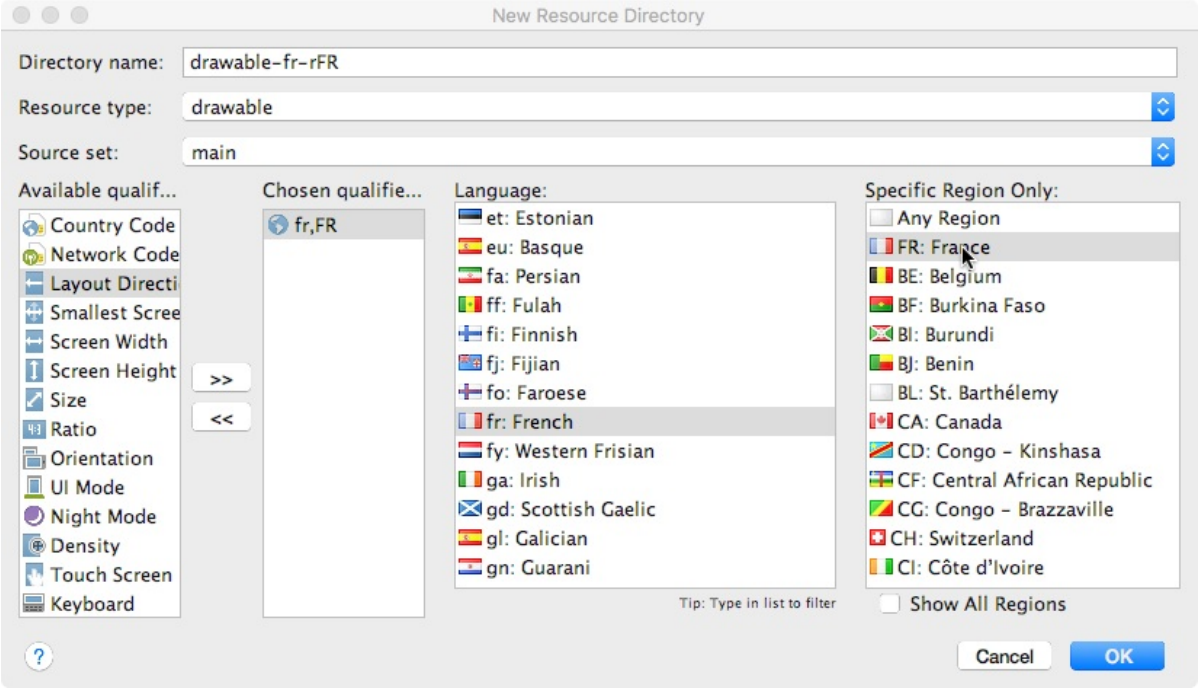
**Important:** Always provide default resources—don't rely on language- or locale-specific resources to cover all the use cases for your app. For details, see "Why default resources are important" in this section.
The previous chapter described how to add languages with the Translations Editor. You can also use Android Studio to create resource directories for strings, dimensions, colors, drawable images, and other resources, as described in this section.

## Adding a resource directory

Use Android Studio to add resource directories for a specific language and locale.

For example, you may want to substitute a specific image for "flag.png" in an app if the user chooses French as the language and France as the locale. To add a `drawable` folder with a replacement image, follow these steps:

1. Right-click the **res** directory and choosing **New > Android Resource Directory** to add a new resource directory.
2. Choose **drawable** from the **Resource type** dropdown menu. The **Directory name** changes to `drawable`. Note that you can add any type of resource directory, including `values` or `layout` directories.
3. Choose **Locale** in the left column and click to select a language and locale. Choose **fr: French** as the language, and **FR: France** as the locale. The **Directory name** changes to `drawable-fr-rFR`. Click **OK**.




The **Directory name** now includes a localization qualifier that specifies a language and, optionally, a region. The localization qualifier is a two-letter ISO 639-1 language code such as `fr` for French, optionally followed by a two letter ISO 3166-1-alpha-2 region code preceded by lowercase `r`, such as `rFR` for France.

The following is the general format for resource directory names that include localization qualifiers: *resource type* `-` *language code* `[-r` *country code* `]`

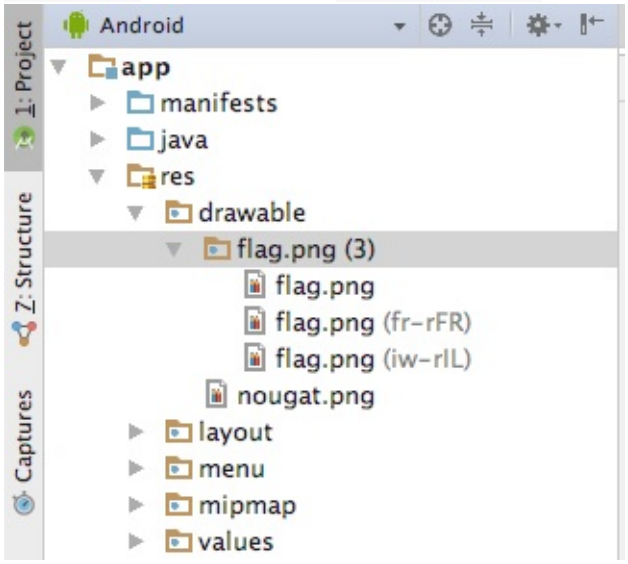
- *resource type* : The resource subdirectory, such as `values` or `drawable`.
- *language code* : The language code, such as `en` for English or `fr` for French.
- *country code* : Optional: The country code, such as `US` for the U.S. or `FR` for France.

After creating the `drawable-fr-rFR` directory, copy the replacement image, which should also be named "flag.png", into the directory. You must use the same filename, because that filename is defined for the `ImageView` in the layout, as shown below—Android simply swaps in the different image.

```
<ImageView
        android:id="@+id/flag"
        app:srcCompat="@drawable/flag"
```

In **Project: Android** view, the image files appear in the `drawable` directories with their identifiers in parentheses. The figure below shows the `flag.png` `drawable` in **Project: Android** view. The `flag.png (fr-rFR)` `drawable` is actually the `flag.png` file inside the

`values-fr-rFR` directory for French in France. The `flag.png (iw-rIL)` `drawable` is the `flag.png` file inside the `values-iw-rIL` directory for Hebrew in Israel.




## Changing colors and styles for different locales

You may not want to use the same design elements for each locale. For example, the color red in western countries indicates high energy or love, but in some eastern cultures, red is associated with prosperity, and in South Africa, mourning. Depending on your app, a design that incorporates a lot of red may work well in Peoria but not in Pretoria. You may also want to change the text size, font, and other style attributes for different languages.

To add color and style resources for different locales:

1. Add a resource directory for a language and locale.
2. Copy the files you need from the default resource directory, such as the `values` directory.
3. Paste the files into the new resource directory, such as `values-iw-rIL` (for Hebrew in Israel).
4. Edit the files that are associated with the language and locale identifiers—such as `styles.xml (iw-rIL)` or `colors.xml (iw-rIL)` .

For example, in some apps the text size is controlled by a style in `styles.xml` in the `values` directory. Copy that `styles.xml` file into the newly created `values-iw-rIL` directory, then open the `styles.xml (iw-rIL)` file to modify it for the Hebrew language in Israel.

In some apps, color for a button's background may be set in an XML file in the `drawable` directory. Copy the `drawable` directory to create a different language version, then change the color in the copied XML file.

## Why default resources are important

Do not assume that the device running your app is using one of the languages your app supports. The device might be set to a language or locale that you did not plan for, or that you didn't test. Design your app so that it will function no matter what language or locale the user chooses.

Android chooses the *default* resources if the language and locale is not specifically supported. That means if the user chooses **Español (España)** and your app does not provide the Spanish language (Español) for the Spain (España) locale, the app shows the language used in the *default* `values` resource directory.

**Note:** Make sure that your app includes a full set of default resources. If an app is missing a default resource (such as a string or a color setting), its behavior will be unpredictable. The default directories in the `res` directory, created automatically by Android Studio, must contain *all* resources for the app. The following default directories are the ones most often used for translation and customization for languages and locales:

- `drawable` : Drawables and ".png" files for images and logos in the app.
- `layout` : Layouts for the default language and locale.
- `values` : The resources for the default language and locale (such as English in the U.S.). The `values` directory contains the following *resource files:*
    - `colors.xml` : Resources for color choices.
    - `dimens.xml` : Resources for dimensions in the layout.
    - `strings.xml` : Resources for all strings in the app.
    - `styles.xml` : Resources for styles used in the layout.

The following default directories are rarely customized for localization:

- `menu` : Options and popup menu choices. Your app can offer different menus for different languages, but to reduce the overhead of managing them, translate only the strings for the menu choices that are defined in the `strings.xml` file (in the `values` directory).
- `xml` : Preferences and other XML collections of resources. To reduce the overhead of managing separate XML resources, translate only the strings for the preferences that are defined in the `strings.xml` file (in the `values` directory).
- `mipmap` : Pre-calculated, optimized collections of app icons used by the Launcher. You can offer different icons for different languages and locales, but it is not necessary if your icon images are acceptable to the cultures in other countries.

**Tip**: To test whether an app includes every string resource that it needs, set the emulator or device to a language that your app does not support. For step-by-step instructions, see Testing for Default Resources.

# Related practical

The related practical documentation is Using the locale to format information.

# Learn more

Android developer documentation:

- Supporting Different Languages and Cultures
- Localizing with Resources
- Localization checklist
- Language and Locale
- Testing for Default Resources

Android Developers Blog:

- Android Design Support Library
- Native RTL support in Android 4.2

Android Play Console: Translate & localize your app

Video: Welcome to the World of Localization

Other:

- Language Subtag Registry - IANA
- Country Codes
- ISO 3166 Country Codes
- Android Locale Codes and Variants

# 6.1: Accessibility

**Contents:**



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

Common disabilities that can affect a person's use of an Android device include blindness, low vision, color blindness, deafness or hearing loss, and restricted motor skills. When you develop your 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.

## About accessibility

The goal of accessibility is to enable all users, including those with disabilities, to use your app. To develop an accessible app, you must challenge the assumptions you have about your users. Tailor the design and code for your app to address those assumptions.

In most cases you don't need to rewrite or redesign your app to address accessibility—the Android framework provides many accessibility features that your app gets for free. To turn up further possible limitations, test your app with accessibility in mind. Usually you can make easy adaptations to fix the problems you find.

Most of Android's accessibility features for users are contained in the settings activity (**Android > Settings > Accessibility**). Android accessibility features include:

- *Google TalkBack* : To interact with your device using touch and spoken feedback, turn on the TalkBack screen reader. TalkBack describes your actions and tells you about alerts and notifications.

- *Select to Speak for Android* : If you want spoken feedback only at certain times, turn on Select to Speak. Select items on your screen to hear them read or described aloud.
- *Switch Access for Android* : For users with limited mobility, Switch Access is an alternative to using the touchscreen. Switch Access enables you to use a switch or keyboard to control your device.
- *Voice Access app for Android* : If using a touchscreen is difficult, the Voice Access app lets you control your device with spoken commands. Use your voice to open apps, navigate, and edit text hands-free. Voice Access is currently in a limited beta release in English only.
- *Google BrailleBack* : With this app you can connect a refreshable braille display to your device via Bluetooth. BrailleBack works with TalkBack for a combined speech and braille experience, allowing you to edit text and interact with your device. The BrailleBack app is available in the Play Store.
- *Display size and font size* : To change the size of items on your screen, adjust the display size or font size.
- *Magnification gestures* : To temporarily zoom or magnify your screen, use magnification gestures.
- *Contrast and color options* : To adjust the contrast or colors on your screen, use high-contrast text, color inversion, or color correction.
- *Captions* : Turn on captions for your device and specify options for closed captioning. Options include language, text, and style.

# App layout and organization

Accessibility begins with the basic design of your app. Designing clear layouts with distinct calls to action helps all your users, including users with disabilities.

## Layout design

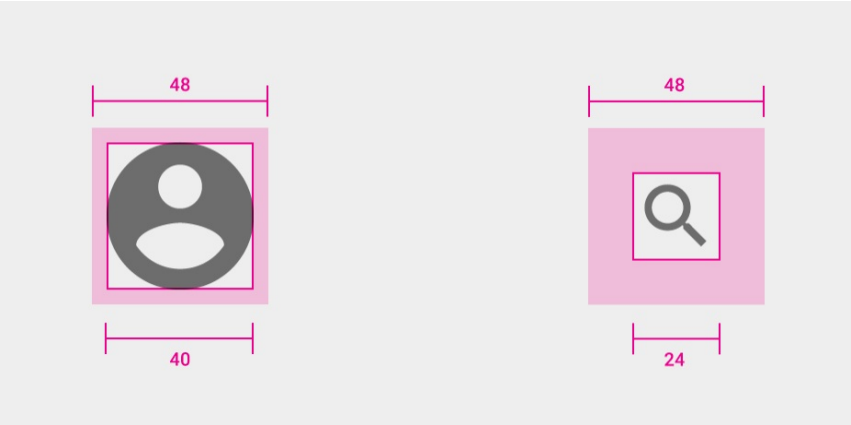
Follow these guidelines for layout design to improve the accessibility of your app:

- Avoid complex layouts and cluttered screens, where users with low vision may have trouble distinguishing between elements. Screens with too many views may be difficult to navigate with a screen reader, because each view's description must be read aloud.
- Place items on the screen according to their level of importance. Place important actions at the top or the bottom of the screen. Ensure that key items stand out structurally and visually.
- Use the standard Material Design guidelines for toolbars, menus, action buttons, and other elements. Apps with regular, consistent elements are easier to navigate.
- Group-related items together on the screen in proximity to each other. Keeping related

items together is helpful for users who have low vision or trouble focusing on the screen.

## Touch targets

*Touch targets* are the parts of the screen that respond to user input. They extend beyond the visual bounds of an element. For example, an icon may appear to be 24 x 24 dp, but the padding surrounding it comprises the full 48 x 48 dp touch target.




Touch targets should be at least 48 x 48 dp. Larger touch targets are easier for users to tap, which is helpful for users who can't see the screen and users who have motor-dexterity problems. Large touch targets may also help you to accommodate a larger spectrum of users, such as children with developing motor skills. For more information, see Touch target size.

## Selection and focus

Selection and focus are not concepts you normally run into with touch-based devices, because the user interacts directly with the views on the screen. To activate a button, for example, the user just touches it.

Users who cannot touch the screen or who have visual impairments may use a separate input device such as a keyboard, mouse, or the TalkBack screen reader to interact with your app. With a hardware input device, the user can move an indicator such as a highlight or colored box from view to view on the screen. The view that has the highlight is considered *selected* or *focussed* . The user can then *activate* the selected view by pressing a key or a button. With TalkBack, the user touches the screen to identify each view, and the selected view is read aloud. The selected item is activated by touching it a second time.

If you use standard Android views in your app, you do not need to do anything to enable selection and focus navigation for your app. The Android system automatically recognizes when a user is interacting with your app using a method other than the touch screen, and shows a highlight (a colored box) on selected views.

You should test your app to ensure that it behaves the way you expect for non-touch users. To test your app, connect an external input device to your phone or tablet, or use the Android emulator. Use the tab keys to move the selection from view to view, or the arrow keys to move left, right, up, and down. Press **Return** to activate the selection.

There are two things to watch out for when you test your app for focus and selection:

- Whether all the views can receive the focus (that is, whether all the views are focusable).
- The focus order of the views in an activity.

Views that can be selected or focussed are considered *focusable* . Most Android views are focusable by default. Some views, such as `ImageView` objects, are not focusable, can't be selected, and are not described in TalkBack. If you use an `ImageView` to convey important information, you may want this view to be focusable so it can be described in TalkBack. To make a view focusable, use the `focusable` attribute in your XML layout:

```
<ImageView
   android:id="@+id/image_partly_cloudy"
   android:layout_width="200dp"
   android:layout_height="160dp"
   app:srcCompat="@drawable/partly_cloudy"
   android:focusable="true" />
```

*Focus order* indicates which view next gets the selection and the focus when the user traverses your app from view to view. For focus order, Android automatically tries to figure out the right focus order for your views. Most often, the focus order is left to right, then top to bottom. To change the default focus order for your app, use the `nextFocusDown` , `nextFocusLeft` , `nextFocusRight` , and `nextFocusUp` XML attributes. Specify which view (by ID) should get the focus next, after the current view.

```
<Button
   android:id="@+id/button1"
   android:layout_alignParentTop="true"
   android:layout_alignParentRight="true"
   android:nextFocusForward="@+id/editText1"
   ... />
```

See Supporting Keyboard Navigation for more details on selection and focus order.

# Content labels and descriptions

Content labels describe the meaning and purpose of each element in your app. These labels allow screen readers to explain the function of each element accurately.

Views that don't otherwise contain text need labels. For example, `ImageView` and `ImageButton` views need labels. Buttons with text that doesn't fully describe the action for that view might also need labels.

Views with textual components such as `TextView` views or checkboxes might *not* need labels, because the actual text might be enough of a label.

## Providing labels

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

- When labeling a static element, add an attribute to the corresponding XML element within the activity's layout resource file. (Static elements don't change their appearance during an activity's lifecycle.)
- When labeling a dynamic element, set the element's label in the Java method that changes the element's appearance. (Dynamic elements change their appearance during an activity's lifecycle.)

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

```
<ImageButton
   android:id="@+id/button_image"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:contentDescription="Discard this message"
   app:srcCompat="@drawable/ic_action_discard" />
```

**Note:** Your content descriptions in XML layout files can (and should) be string resources, so that they can be translated.
When labeling editable elements, such as `EditTex t` objects, use the `android:hint` XML attribute for static elements and the `setHint()` method for dynamic elements to indicate each element's purpose.

```
<EditText
   android:id="@+id/edittext_message"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:hint="Enter your message" />
```

If you target your app to Android 4.2 (API level 17) or higher, use the `android:labelFor` attribute when labeling views that serve as content labels for other `View` objects. For example, the `TextView` view below serves as a content label for an `EditText` view with the ID `edittext_message`. The `TextView` uses the `android:labelFor` attribute with the ID of the `EditText` that the `TextView` labels.

```
<TextView
   android:id="@+id/textview_label"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="@string/label_message"
   android:labelFor="@+id/edittext_message"/>
```

## Creating useful descriptions

When adding content descriptions to your views, follow these best practices:

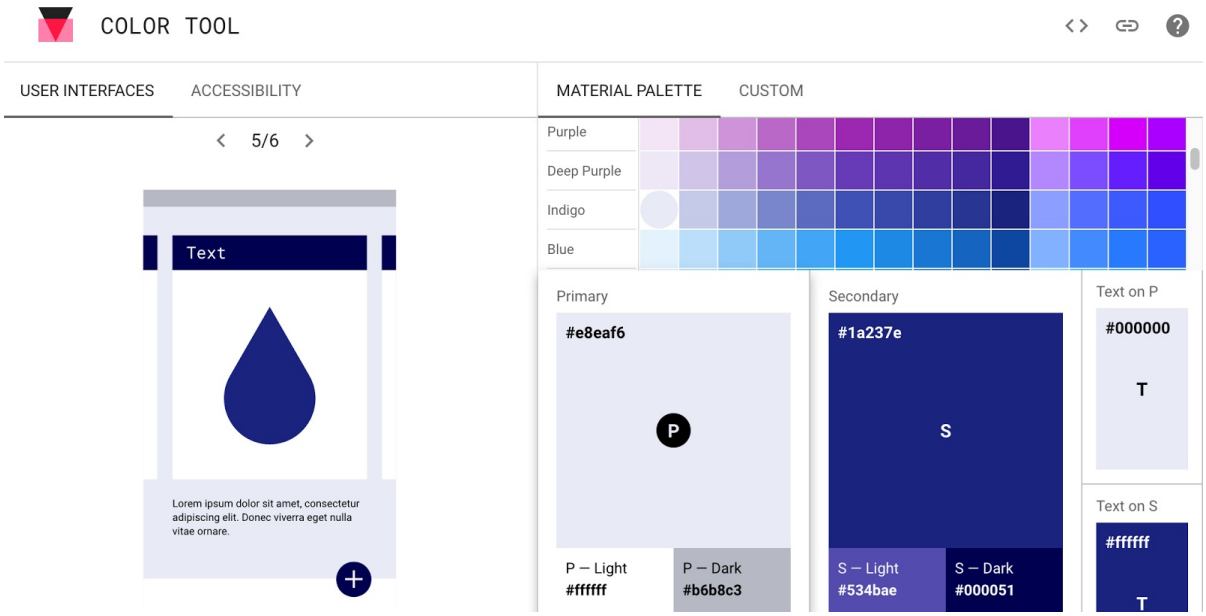
- Always add content descriptions to `ImageView` and `ImageButton` views that have no textual component.
- A button needs a content description if the button label does not fully describe the action, or to differentiate buttons with the same label.
- You usually don't need to add a content description to a view that has a textual component, such as a text view or a checkbox.
- Use action verbs to describe what an element does, not what the element looks like. For example, "Delete this message" is better than "Trash Can."
- Do not include the name of the UI element (such as "button") in the content description. Accessibility services such as TalkBack announce the view's type after announcing its label.
- Don't tell users how to physically interact with a control. Users may be navigating with a keyboard or by voice, not with their fingers or a mouse. Accessibility software will describe the correct interaction for the user. For example, "Voice Search" is a better label than "Tap to speak."
- Make sure that your content descriptions are unique so that your users can identify each element accurately. Unique content descriptions are especially important with collective views such as `RecyclerView` objects, where there may be many similar list elements.
- Conversely, make sure that the same view has the same description everywhere the description is used.

# Color and contrast

Your visual design choices can help users with low vision or color blindness see and interpret your app's content, interact with the right elements, and understand actions.

## Color

When choosing colors for your app, the Material Design color palettes have been designed to work harmoniously with each other. Google's Material Design color tool can help you create, share, and apply color palettes to your UI, as well as measure the accessibility level of any color combination. Choose colors for your app and then click the **Accessibility** tab to see legibility warnings. The tool can help you ensure that the majority of combinations you choose have the right amount of contrast.



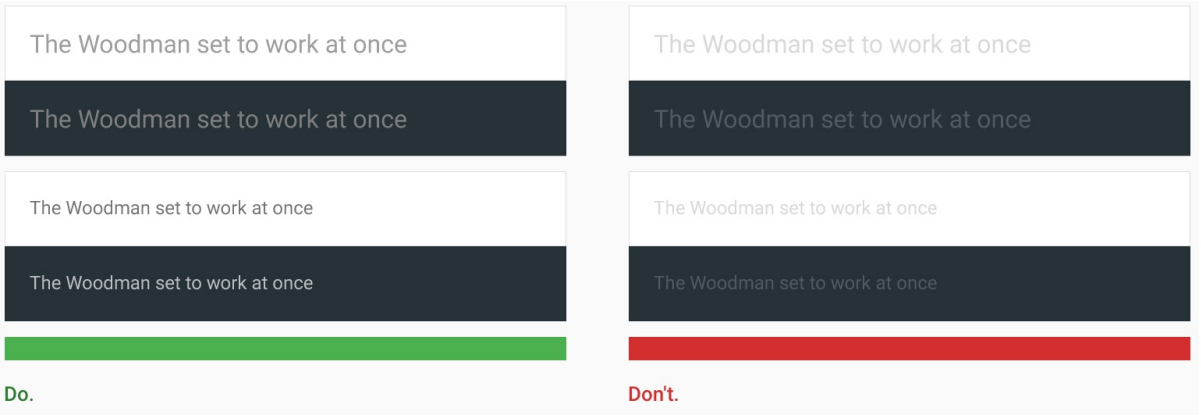

## Contrast

Even if you choose colors from the Material Design palettes, ensure that your app's colors have sufficient color contrast between elements. A *contrast ratio* represents how different a color is from another color. The higher the difference between the two numbers in the ratio, the greater the difference in relative luminance between the two colors.

The contrast ratio between a color and its background ranges from 1-21 based on its luminance, or intensity of light emitted, according to the World Wide Web Consortium (W3C). The W3C recommends the following contrast ratios for body text and image text:

- Small text should have a contrast ratio of at least 4.5:1 against its background.
- Large text (at 14 pt bold/18 pt regular and up) should have a contrast ratio of at least 3:1 against its background.

These two images show examples of text set against a black or white background. The text at the left show higher contrast. On the right, the text is set at a lower contrast, which is harder to read.

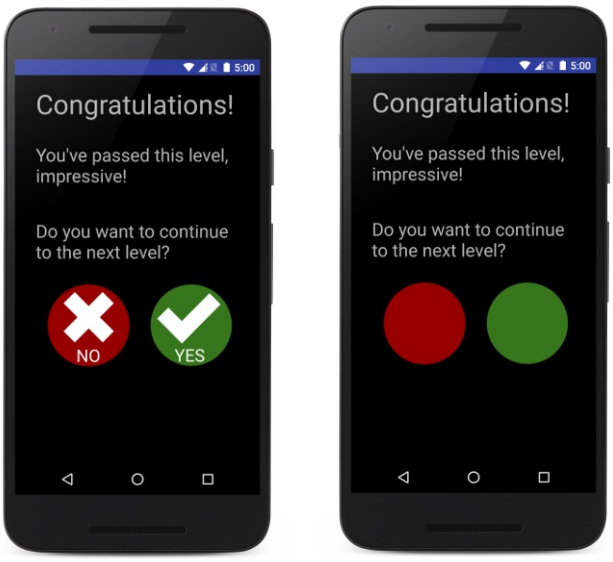



Declarative elements such as logos don't need to meet minimum contrast ratios. Even so, if declarative elements provide important information, make sure they are easy to distinguish.

## Other visual cues

Color blindness takes different forms, including red-green, blue-yellow, and monochromatic color blindness. To assist users with color vision deficiencies, use visual cues other than color to distinguish UI elements within your app's screens. To mark the differences between elements, use different shapes or sizes, provide text or visual patterns, or add audio- or touch-based (haptic) feedback.

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



# Type size

Users can change the system font size in the Android settings by selecting **Settings > Accessibility > Font Size**. Larger font sizes are easier to read for users with vision impairment, older adults, and on-the-go users who may need to access information at a glance.

An Android best practice is to specify all type in your app as scalable point (sp) units. Using sp units for font sizes enables your layout to scale based on the system font size.

Android also provides styles for small, medium, and large typefaces that you can use for your views:

```
<TextView
    android:id="@+id/labelSmall"
    style="@android:style/TextAppearance.Small"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Small Text" />
```

Specifying your type sizes with dynamic values (sp or styles) enables your layout to adjust to the user's type-size setting.

Test your app with a large font size enabled to ensure that your layout adapts for larger font sizes.

# Audio and video

Apps that play sound or video need to be accessible by people who are hard of hearing. Provide visual alternatives to audio content, and provide audio alternatives to visual content. Do not use audio-only feedback. For essential audio feedback and alerts, provide closed captions, a transcript, or another visual alternative.

# Testing for accessibility

Testing for accessibility lets you experience your app from the perspective of your users and find usability issues that you might otherwise miss. Accessibility testing can reveal opportunities to make your app more powerful and versatile for all your users, including users with disabilities.

For requirements, recommendations, and considerations to help you make sure that your app is accessible, see the tools described in this section and the Accessibility Developer Checklist.

## Manual testing with TalkBack

TalkBack is Android's built-in screen reader. When TalkBack is on, users can interact with their Android device without seeing the screen. Users with visual impairments may rely on TalkBack when they use your app.

Testing your app manually with TalkBack lets you to experience your app the same way your users do. TalkBack can help you quickly identify places with room for improvement.

To enable TalkBack and test your app:

1. On a device, navigate to **Settings > Accessibility > TalkBack**.
2. Tap the **On/Off** toggle button to turn on TalkBack.
3. Tap **OK** to confirm permissions.
4. Confirm your device password, if asked.

   If you have never run TalkBack, a tutorial launches. Use the tutorial to learn about:

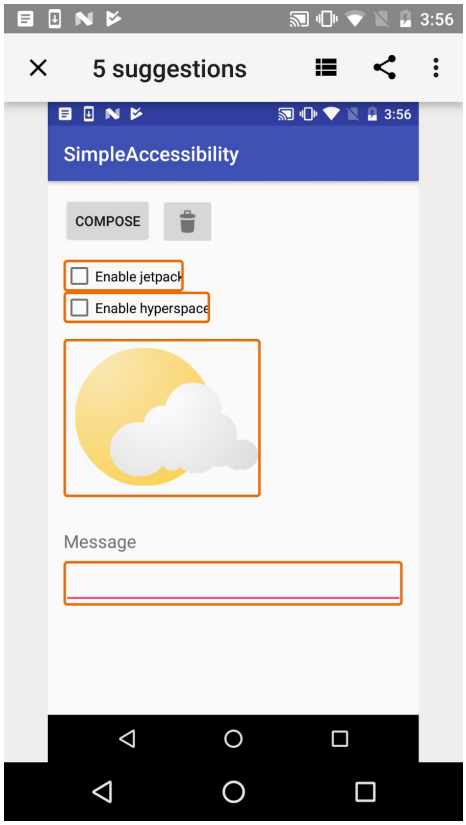
   - Explore by touch. TalkBack identifies every item you touch on the screen. You can touch items individually or move your finger over the screen. Swipe left or right to explore the items in tab (focus) order. To activate the last item heard, double-tap that item.
   - Scrolling. Lists can be scrolled with a two finger scroll, or you can jump forward or back in a list with a side-by-side swipe.
   - Using global and local TalkBack menus.
   - Setting the text navigation rate. Swipe up or down as TalkBack reads text aloud, one character, word, line, or paragraph at a time.
   - Activating `EditText` views and entering text.
5. Navigate to your app and test each view and activity.

## The Accessibility Scanner app

The Accessibility Scanner app scans your screen and provides suggestions for improving the accessibility of your app. To use Accessibility Scanner:

1. Install Accessibility Scanner on your device from the Google Play Store.
2. Navigate to **Settings > Accessibility >Accessibility Scanner** and click the toggle button to enable Accessibility Scanner. A button with a check mark appears and floats on the screen.

3. Navigate to your app on the device and to the activity you want to scan.
4. Tap the **Accessibility Scanner** (check mark) button to scan the screen.

A screenshot appears with problematic elements highlighted, and with notes about possible improvements. To get a full list of results, tap the list button, which is the third icon from the right. 


Accessibility Scanner saves the results of all past scans, and you can share the results of any scan.

## Accessibility testing framework

The Android platform supports several testing frameworks, including Espresso and Robolectric. These testing frameworks allow you to create and run automated tests that evaluate the accessibility of your app.

The Android Testing Support Library includes accessibility functions.You can enable and configure accessibility testing through the `AccessibilityChecks` class:

```
AccessibilityChecks.enable();
```

By default, the checks run when you perform any view action that's defined in the `ViewActions` class. The check includes the view on which the action is performed as well as all descendent views. You can check the entire view hierarchy of a screen with this code:

```
AccessibilityChecks.enable().setRunChecksFromRootView(true);
```

### Lint

Android Studio shows lint warnings for various accessibility issues and provides links to the places in the source code containing these issues. For example, if one of the views in your XML layout needs a content description and doesn't have one, that view name is highlighted in yellow. Android Studio displays a message such as the following:

```
[Accessibility] Missing 'contentDescription' attribute on image
```

## Related practicals

The related practical documentation is in:

- Explore accessibility in Android
- Creating accessible apps

## Learn more

Android support documentation:

- Android accessibility overview - Android Accessibility Help
- Get started on Android with TalkBack - Android Accessibility Help
- Editable View labels - Android Accessibility Help
- Content labels - Android Accessibility Help

Android developer documentation:

- Accessibility Overview
- Making Apps More Accessible
- Accessibility Developer Checklist
- Building Accessible Custom Views
- Developing Accessible Applications
- Designing Effective Navigation
- Testing Your App's Accessibility
- Developing an Accessibility Service

Material Design:

- Usability - Accessibility

- Color Tool

Accessibility testing:

- Testing your App's Accessibility
- Get Started with Accessibility Scanner
- Accessibility Testing on Android

Videos:

- GTAC 2015: Automated Accessibility Testing for Android Applications
- Google I/O 2015 - Improve your Android app's accessibility
- Google I/O 2016 - What's New in Android Accessibility
- Google I/O 2017 - What's New in Android Accessibility

Other:

- An Introduction to Android Accessibility Features - SitePoint
- Having Trouble Focusing? A Primer on Focus in Android
- Touch Mode
- Type Sizes for Every Device

# 7.1: Location services

**Contents:**



Using location in an app can greatly improve the user experience by providing contextual information. For example, you may want your app to be aware that the user is at the gym to start tracking their fitness, or to turn on Wi-Fi when they go to a friend's house.

The system obtains the device's location using a combination of GPS, Wi-Fi, and cell network technologies. Google Play services provides the `FusedLocationProviderClient` class, a convenient way to let the system make location requests on your behalf. A *fused* location service estimates device location by combining, or "fusing," all the available location providers, including GPS location and network location. It does this to balance fast, accurate results with minimal battery drain.

The fused location provider methods return a `Location` object, which contains geographic coordinates in the form of latitude and longitude. If your app requires a physical address, you can convert from latitude/longitude coordinates (and back) using the `Geocoder` class.

The user controls location settings on their device, including settings for location precision. The user can also turn off location services completely. If your app requires location services, you can detect the current location settings and prompt the user to change them in a system dialog.

## Setting up Google Play services

Before you can use the features provided by Google Play services, you must install the Google Repository, which includes the Google Play services SDK. To install the Google Repository and update the Android SDK Manager:

1. Open Android Studio.
2. Select **Tools > Android > SDK Manager**.
3. Select the **SDK Tools** tab.
4. Expand **Support Repository**, select **Google Repository**, and click **OK**.

To add Google Play services to your project, add the following line of code to the `dependencies` section in your app-level `build.gradle (Module: app)` file:

```
compile 'com.google.android.gms:play-services:XX.X.X'
```

Replace `XX.X.X` with the latest version number, for example `11.0.2`. Android Studio will let you know if you need to update it. For more information and the latest version number, see Add Google Play Services to Your Project.

**Note:** If your app references more than 65K methods, the app may fail to compile. To mitigate this problem, compile your app using only the Google Play services APIs that your app uses. To learn how to selectively compile APIs into your executable, see Set Up Google Play Services in the developer documentation. To learn how to enable an app configuration known as *multidex* , see Configure Apps with Over 64K Methods.
Now that you have Google Play services installed, you're ready to connect to the `LocationServices` API.

## Requesting location permissions

Apps that use location services must request location permissions. Android offers two location permissions that you can include in the manifest:

- `ACCESS_COARSE_LOCATION`: The API returns a location with an accuracy approximately equivalent to a city block.
- `ACCESS_FINE_LOCATION`: The API returns the most precise location data available.

Starting with Android 6.0 (API level 23), you also have to request "dangerous" permissions at runtime. The user can revoke permissions at any time, so each time you use location services, check for permissions. For more information, see Requesting Permissions at Runtime.

## Requesting the last known location

Once your app connects to the `LocationServices` API, here's how to get the last known location of a user's device:

1. Create an instance of a `FusedLocationProviderClient` using the `LocationServices.getFusedLocationProviderClient()` method.

2. Call the `FusedLocationProviderClient` `getLastLocation()` method to retrieve the device location. The precision of the location returned by this call is determined by the permission setting you put in your app manifest, as described above.

The `getLastLocation()` method returns a `Task` object. A `Task` object represents an asynchronous operation (usually an operation to fetch some value, in this case a `Location` object). You retrieve the latitude and longitude of a location from the `Location` object. In rare cases when the location is not available, the `Location` object is `null`.

The `Task` class supplies methods for adding success and failure listeners. If the operation is successful, the success listener delivers the desired object. If the operation is unsuccessful, the failure listener delivers an `Exception`:

```
mFusedLocationClient.getLastLocation().addOnSuccessListener(
        new OnSuccessListener<Location>() {
            @Override
            public void onSuccess(Location location) {
                if (location != null) {
                    mLastLocation = location;
                }
            }
        });

mFusedLocationClient.getLastLocation().addOnFailureListener(
        new OnFailureListener() {
            @Override
            public void onFailure(@NonNull Exception e) {
                Log.e(TAG, "onFailure: ", e.printStackTrace());

            }
        }
);
```

Note that the `getLastLocation()` method doesn't actually force the fused location provider to obtain a new location. Instead, it retrieves the most recently obtained location from a local cache. On a physical device, there is usually a service that fetches and caches the location right after the device is restarted.

An emulator, however, does not have such a service, so `getLastLocation()` is more likely to return `null`. To force the fused location provider to obtain a new location, start the Google Maps app and accept the terms and conditions (if you haven't already). Use the **Location** tab of the emulator settings to deliver a location to the fused location provider. This forces a value to be cached, and `getLastLocation()` no longer returns `null`.

## Geocoding and reverse geocoding

- *Geocoding* is the process of converting a street address into a set of coordinates (like latitude and longitude).
- *Reverse geocoding* is the process of converting a set of coordinates into a human-readable address.

The location services API methods provide information about the device's location using `Location` objects. A `Location` object contains a latitude, longitude, timestamp, and optional parameters such as bearing, speed, and altitude. Although latitude and longitude are useful for calculating distance or displaying a map position, in many cases the address of the location is more useful. For example, if you want to let your users know where they are or what is close by, a street address is more meaningful than the latitude and longitude.

You can use the `Geocoder` class to do both geocoding and reverse geocoding. The amount of detail in a reverse-geocoded location description varies; for example it might contain the full street address of the closest building, or only a city name and postal code.

The `Geocoder` class supplies the following methods:

- `getFromLocation(double latitude, double longitude, int maxResults)` : Use this method for reverse geocoding. The `getFromLocation()` method takes coordinates as arguments and returns a list of `Address` objects.

  ```
  List<Address> addresses = geocoder.getFromLocation(
       location.getLatitude(), location.getLongitude(), 1);
  ```

- `getFromLocationName(String locationName, int maxResults)` : Use this method to geocode a location name into a set of coordinates. Like the `getFromLocation()` method, the `getFromLocationName()` method returns a list of `Address` objects that contain latitude and longitude coordinates. It can also be used with additional parameter to specify a rectangle to search within.

  ```
  List<Address> addresses = geocoder
  .getFromLocationName("731 Market St, San Francisco, CA 94103", 1)
  Address firstAddress = addresses.get(0);
  double latitude = firstAddress.getLatitude();
  double longitude = firstAddress.getLongitude();
  ```

  Both `getFromLocation()` and `getFromLocationName()` perform a network lookup and therefore may take a while to complete. For this reason you should not call them on the main thread.

**Note**: The `Geocoder` class requires a backend service that is not included in the core Android framework. The `Geocoder` query methods return an empty list if there is no backend service in the platform. To determine whether a `Geocoder` implementation exists, use the `isPresent()` method.

## Creating a LocationRequest object

If your app requires precise tracking that involves periodic updates, you need to create a `LocationRequest` object that contains the requirements of the location updates. The fused location provider attempts to batch location requests from different apps together, in order to preserve battery.

The options you can set for a `LocationRequest` include:

- `setInterval()` : This method sets how frequently your app needs location updates, in milliseconds. If another app is receiving updates more frequently, location updates may be more frequent than this interval. Updates may also be less frequent than this interval, or there may be no updates at all, for example if the device has no connectivity.
- `setFastestInterval()` : This method sets a limit, in milliseconds, to the update rate. The location APIs send out updates at the fastest rate that any app has requested with `setInterval()` . If this rate is faster than your app can handle, you may encounter UI flicker or data overflow. To prevent these problems, call `setFastestInterval()` .
- `setPriority()` : This method sets the priority of the request, which gives the location APIs a strong hint about which location sources to use. The following values are supported:
- `PRIORITY_BALANCED_POWER_ACCURACY` : Use this setting to request location precision to within a city block, which is an accuracy of approximately 100 meters. This is considered a coarse level of accuracy, and is likely to consume less power. With this setting, the location services are likely to use Wi-Fi and cell-tower positioning to determine location. Note, however, that the choice of location provider depends on many other factors, such as which sources are available.
- `PRIORITY_HIGH_ACCURACY` : Use this setting to request the most precise location possible. With this setting, the location services are more likely to use GPS to determine location.
- `PRIORITY_LOW_POWER` : Use this setting to request city-level precision, which is an accuracy of approximately 10 kilometers. This is considered a coarse level of accuracy, and is likely to consume less power.
- `PRIORITY_NO_POWER` : Use this setting if you need zero additional power consumption, but want to receive location updates when available. With this setting, your app does not trigger any location updates, but receives locations triggered by other apps.

For a list of other options you can set, see `LocationRequest` .

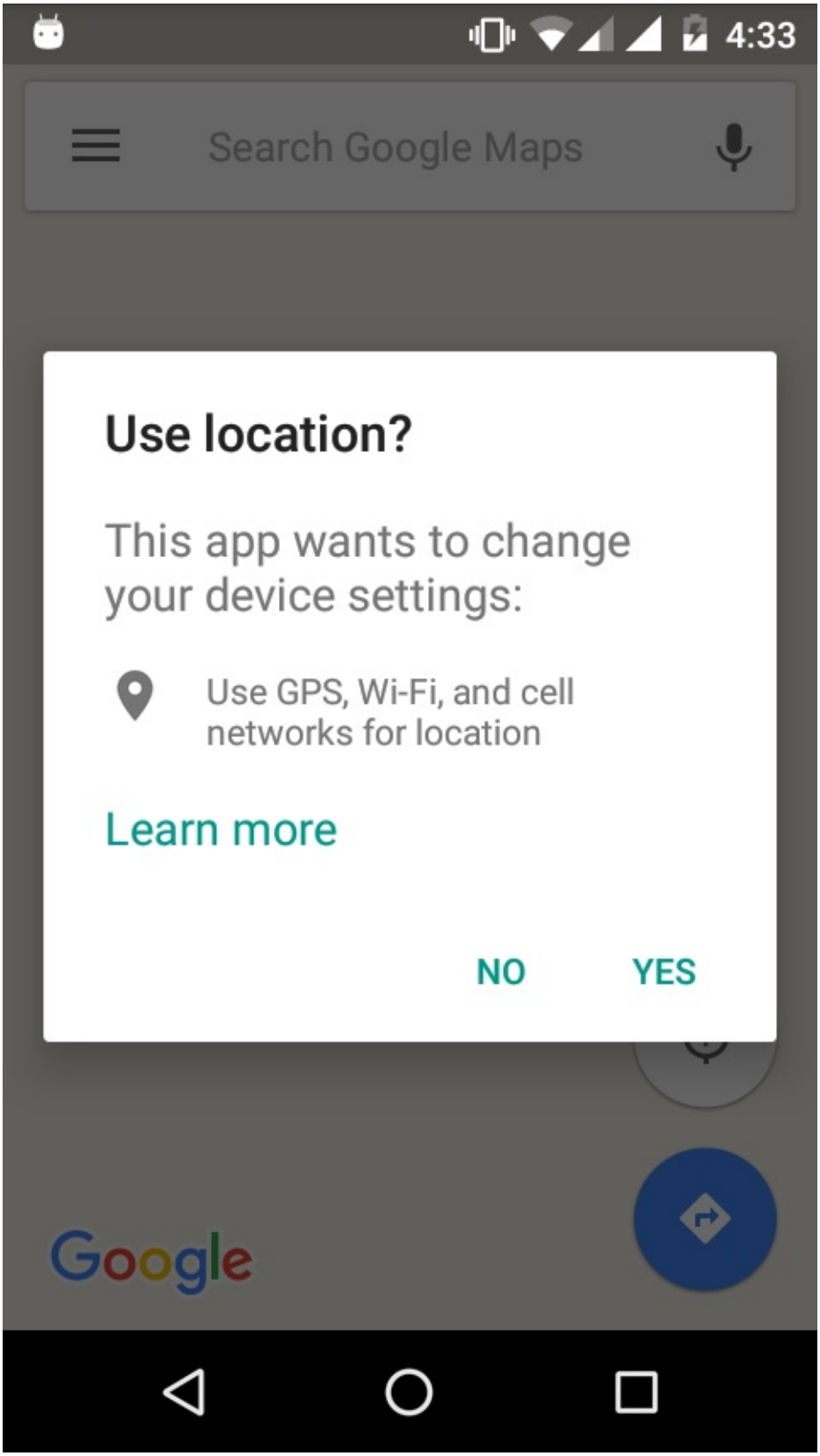
```
private LocationRequest getLocationRequest() {
   LocationRequest locationRequest = new LocationRequest();
   locationRequest.setInterval(10000);
   locationRequest.setFastestInterval(5000);
   locationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
   return locationRequest;
}
```

## Working with the user's location settings

Activating the GPS and network capabilities of a device uses significant power. Additionally, users may have privacy concerns about their location being tracked.This is why users have a device setting that lets them control the balance between accuracy and power for location requests. Users can also turn off location services completely.

The user's settings may reduce the accuracy of location tracking to the point where it reduces your app's usability. For example, if the user turns off location, it doesn't make sense to have a tracker app running and using system resources. You can detect the device settings and prompt the user to change them to match what you've set in your

The user's settings may reduce the accuracy of location tracking to the point where it reduces your app's usability. For example, if the user turns off location, it doesn't make sense to have a tracker app running and using system resources. You can detect the device settings and prompt the user to change them to match what you've set in your `LocationRequest`.




To check the device settings:

2. Create a `SettingsClient` object using the `LocationServices.getSettingsClient()` method, and pass in the context:

```
SettingsClient client = LocationServices.getSettingsClient(this);
```

3. Use the `SettingsClient` `checkLocationSettings()` method to see if the device settings match the `LocationRequest`. Pass in the `LocationSettingsRequest` that you created in step 1:

```
Task<LocationSettingsResponse> task = client
       .checkLocationSettings(settingsRequest);
```

   The `checkLocationSettings()` method returns a `Task` object, just like the `getLastLocation()` method discussed above.

4. Use the `OnFailureListener` to catch the case where the device settings do not match the `LocationRequest`.

   The `Exception` passed into the resulting `onFailure()` method contains a `Status` object with information about whether the device settings match the `LocationRequest`.

   If the device settings don't match the `LocationRequest`, the status code is `LocationSettingsStatusCodes.RESOLUTION_REQUIRED`. In this case, show the user a dialog that prompts them to change their settings, then handle the user's decision to either change their settings or back out:

5. Call the `startResolutionForResult()` method on the status.

```
task.addOnFailureListener(this, new OnFailureListener() {
   @Override
   public void onFailure(@NonNull Exception e) {
       int statusCode = ((ApiException) e).getStatusCode();
       if (statusCode
               == LocationSettingsStatusCodes
               .RESOLUTION_REQUIRED) {
           // Location settings are not satisfied, but this can
           // be fixed by showing the user a dialog
           try {
               // Show the dialog by calling
               // startResolutionForResult(), and check the
               // result in onActivityResult()
               ResolvableApiException resolvable =
                       (ResolvableApiException) e;
               resolvable.startResolutionForResult
                       (MainActivity.this,
                               REQUEST_CHECK_SETTINGS);
           } catch (IntentSender.SendIntentException sendEx) {
               // Ignore the error
           }
       }
   }
});
```

6. Override the `onActivityResult()` callback in your `Activity` to handle the user's decision (either to update their settings or to back out). Make sure the `requestCode` matches the constant that you used in the `startResolutionForResult()` method.

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
       Intent data) {
   if (requestCode == REQUEST_CHECK_SETTINGS) {
       if (resultCode == RESULT_CANCELED) {
           stopTrackingLocation();
       } else if (resultCode == RESULT_OK) {
           startTrackingLocation();
       }
   }

   super.onActivityResult(requestCode, resultCode, data);
}
```

## Requesting location updates

After checking the device settings and prompting the user to update them if they don't match your location request, you can start requesting periodic location updates using the `LocationRequest` and the `FusedLocationProvider Client` objects. The accuracy of the

location is determined by the available location providers (network and GPS), the location permissions you requested, and the options you set in the location request.

To request and start location updates:

1. Create a `LocationRequest` object with the desired parameters for your location updates, as described in Creating a LocationRequest object above.
2. The fused location provider invokes the `LocationCallback.onLocationResult()` callback method. Create an instance of `LocationCallback`, and override its `onLocationResult()` method:

```
mLocationCallback = new LocationCallback() {
        @Override
        public void onLocationResult(LocationResult locationResult) {
            for (Location location : locationResult.getLocations()) {
                // Update UI with location data
                // ...
            }
        };
}
```

3. To start the regular updates, call `requestLocationUpdates()` on the `FusedLocationProviderClient`. Pass in the `LocationRequest` and `LocationCallback`. This starts the location updates, which are delivered to the `onLocationResult()` method.

## Stopping location updates

When you no longer need location updates, stop them using the `FusedLocationProviderClient` `removeLocationUpdates()` method, passing it your instance of the `LocationCallback`.

You might want to stop location updates when the activity is no longer in focus, for example when the user switches to another app or to a different activity in the same app. Stopping location updates at these times can reduce power consumption, provided the app doesn't need to collect information even when it's running in the background.

## Related practical

The related practical documentation is in 7.1 P: Using the device location.

## Learn more

Android developer documentation:

- Making Your App Location-Aware

- `Location`
- `Geocoder`
- `FusedLocationProviderClient`

# 8.1: Places API

**Contents:**



The Location API is great for providing your app with the device's current location and requesting periodic updates. However, the Location API falls short when you need details about a location, or when you need to allow the user to search for a specific place that's not tied to the device location.

A *place* is defined as a physical space that has a name. Another way of thinking about a place is that it's anything you can find on a map. Examples include local businesses, points of interest, and geographic locations.

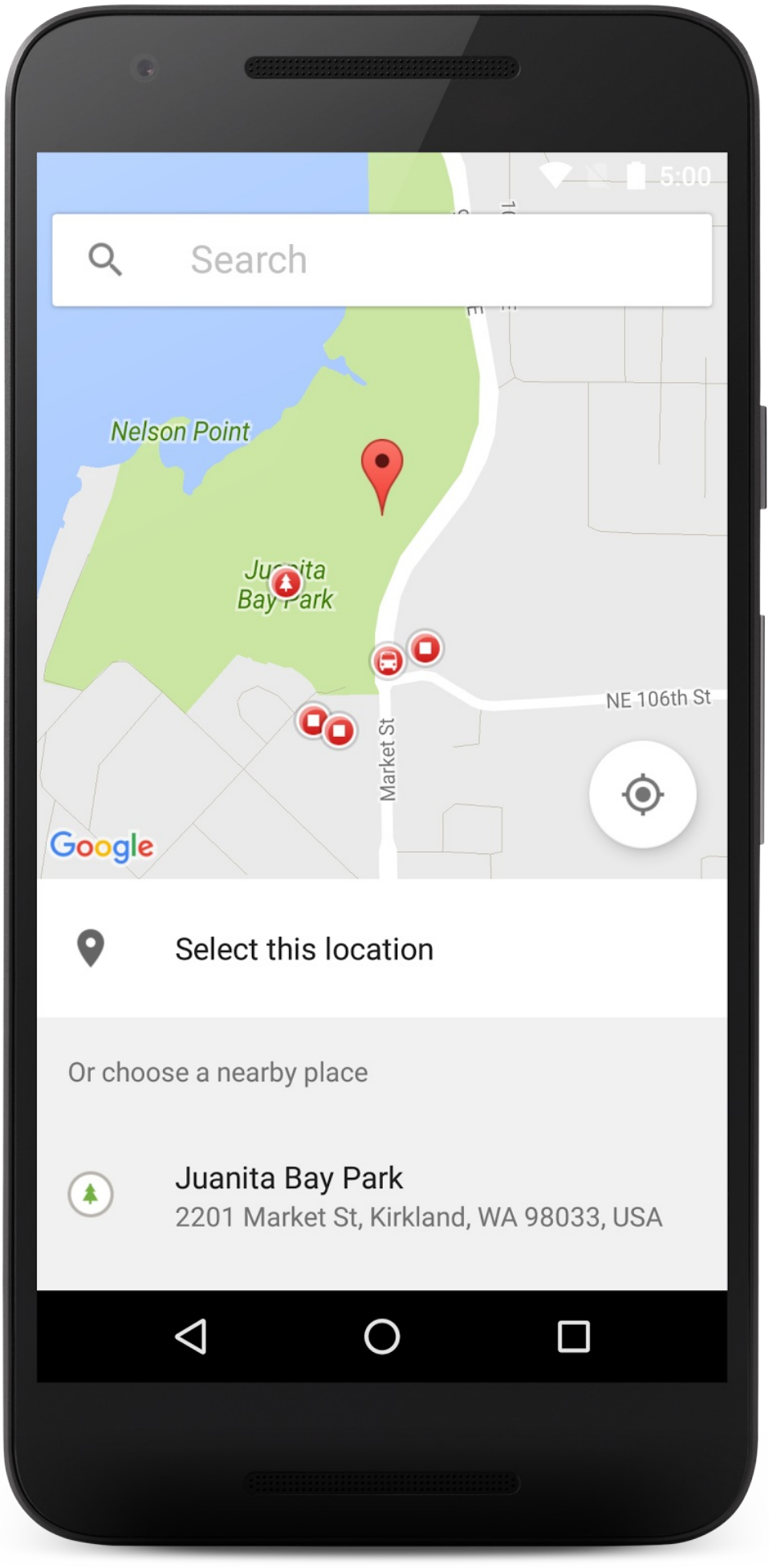
In the Google Places API for Android, a `Place` object represents a place. A `Place` object can include the place's name and address, geographical location, place ID, phone number, place type, website URL, and more.

There are several ways to access the Places API:

- The place-picker UI lets users select a place on an interactive map.
- The `PlaceDetectionClient` interface provides quick access to a device's current place and the place's details.
- The `GeoDataClient` interface provides access to Google's database of local place and business information, including high-quality images.
- The place-autocomplete service provides a UI element that lets a user search for a specific place with autocomplete suggestions. You can also build your own autocomplete UI and use place-autocomplete API calls.

To include the Places API in your app, you must add a dependency to your app-level `build.gradle` file and set up an API key. These steps are described in the related practical for this lesson.

# Using the place-picker UI


5:00
Search
Nelson Point
Juanita Bay Park
NE 106th St
Market St
Google
Select this location
Or choose a nearby place
Juanita Bay Park
2201 Market St, Kirkland, WA 98033, USA

The `PlacePicker` provides a UI dialog that displays an interactive map and a list of nearby places. Users choose a place by selecting the place on a map or by searching for the place. Your app can then retrieve the details of the selected place.

The place picker requires the `ACCESS_FINE_LOCATION` permission.

## Launching the place-picker UI

To use the place-picker UI, complete the following steps:

1. Use the `PlacePicker.IntentBuilder()` to construct an `Intent`.
2. If you want to change the default latitude and longitude bounds of the map that the place picker displays, call `setLatLngBounds()` on the builder. Pass in a `LatLngBounds` object.

   The bounds that you set in the `LatLngBounds` object define an area called the *viewport*. By default, the viewport is centered on the device's location, with the zoom at city-block level.

3. Call `startActivityForResult()`. Pass in the intent and a predefined request code. The request code lets you identify the request when the result is returned.

```
PlacePicker.IntentBuilder builder =
       new PlacePicker.IntentBuilder();
try {
   // Launch the PlacePicker.
   startActivityForResult(builder.build(MainActivity.this)
           , REQUEST_PICK_PLACE);
} catch (GooglePlayServicesRepairableException
       | GooglePlayServicesNotAvailableException e) {
   e.printStackTrace();
}
```

## Obtaining the selected place

Once the user has selected a place in the place-picker dialog, your activity's `onActivityResult()` method is called. The `onActivityResult()` method uses the integer request code you used in `startActivityForResult()`.

To retrieve the place, call `PlacePicker.getPlace()` in `onActivityResult()`. Pass in the activity context and the data `Intent`. If the user has not selected a place, the method returns `null`.

You can also retrieve the most recent bounds of the map by calling `PlacePicker.getLatLngBounds()`.

```
@Override
    protected void onActivityResult(int requestCode, int resultCode,
            Intent data) {
        if (resultCode == RESULT_OK) {
            Place place = PlacePicker.getPlace(this, data);
            setAndroidType(place);
            mLocationTextView.setText(
                    getString(R.string.address_text, place.getName(),
                            place.getAddress(), System.currentTimeMillis()));

        } else {
            mLocationTextView.setText(R.string.no_place);
        }

        super.onActivityResult(requestCode, resultCode, data);
    }
```

# Getting the device's current place

To get information about the device's current location, call the `PlaceDetectionClient.getCurrentPlace()` method. Optionally, pass in a `PlaceFilter` instance. (Filtering is described below, in Filtering the list of places.)

The `getCurrentPlace()` method returns a `Task` that contains a `PlaceLikelihoodBuffer`. The `PlaceLikelihoodBuffer` contains a list of `PlaceLikelihood` objects that represent likely `Place` objects. If no known place corresponds to the filter criteria, the buffer may be empty.

- To retrieve a `Place` object, call `PlaceLikelihood.getPlace()`.
- Each place result includes an indication of the likelihood that the place is the right one. To get the place's likelihood value, call `PlaceLikelihood.getLikelihood()`.

About the likelihood values:

- A place's likelihood value is the relative probability that this place is the best match within the list of returned places for a single request. You can't compare likelihoods across different requests.
- The value of the likelihood is between 0 and 1.0. A higher likelihood value means a greater probability that the place is the best match.
- The sum of the likelihoods in a given `PlaceLikelihoodBuffer` is always less than or equal to 1.0. The sum isn't necessarily 1.0.

For example, there might be a 55% likelihood that the correct place is place A and a 35% likelihood that the correct place is place B. To represent this scenario, the `PlaceLikelihoodBuffer` has two members: `Place` A with a likelihood of 0.55, and `Place` B with a likelihood of 0.35.

**Important:** To prevent a memory leak, release the `PlaceLikelihoodBuffer` object when your app no longer needs it. Read more about handling buffers.

```
Task<PlaceLikelihoodBufferResponse> placeResult =
       mPlaceDetectionClient.getCurrentPlace(null);
placeResult.addOnCompleteListener
       (new OnCompleteListener<PlaceLikelihoodBufferResponse>() {
   @Override
   public void onComplete(@NonNull
           Task<PlaceLikelihoodBufferResponse> task) {

       // If you get a result, get the most likely place and
       // update the place name.
       if (task.isSuccessful()) {
           PlaceLikelihoodBufferResponse likelyPlaces =
                   task.getResult();
           float maxLikelihood = 0;
           Place currentPlace = null;
           for (PlaceLikelihood placeLikelihood : likelyPlaces) {
               if (maxLikelihood < placeLikelihood.getLikelihood())
               {
                   maxLikelihood = placeLikelihood.getLikelihood();
                   currentPlace = placeLikelihood.getPlace();
               }
           }
           likelyPlaces.release();

           // Update the UI.
           if (currentPlace != null) {
               mLocationTextView.setText(
                       getString(R.string.address_text,
                               currentPlace.getName(), result,
                               System.currentTimeMillis()));
               setAndroidType(currentPlace);
           }
           // Otherwise, show an error.
       } else {
           mLocationTextView.setText(
                   getString(R.string.address_text,
                           getString(R.string.no_place),
                           result, System.currentTimeMillis()));
       }

   }
});
```

## Filtering the list of places

To limit the results that the `PlaceLikelihoodBuffer` includes when you request information about the device's current place, use a `PlaceFilter` .

- To limit the results to places that are currently open, pass `true` as the first parameter in the `PlaceFilter`.
- To limit the results to places that match certain place IDs, pass in the place IDs as the second parameter in the `PlaceFilter`. Provide the place IDs as a `Collection` of strings.

# Using the place-autocomplete service

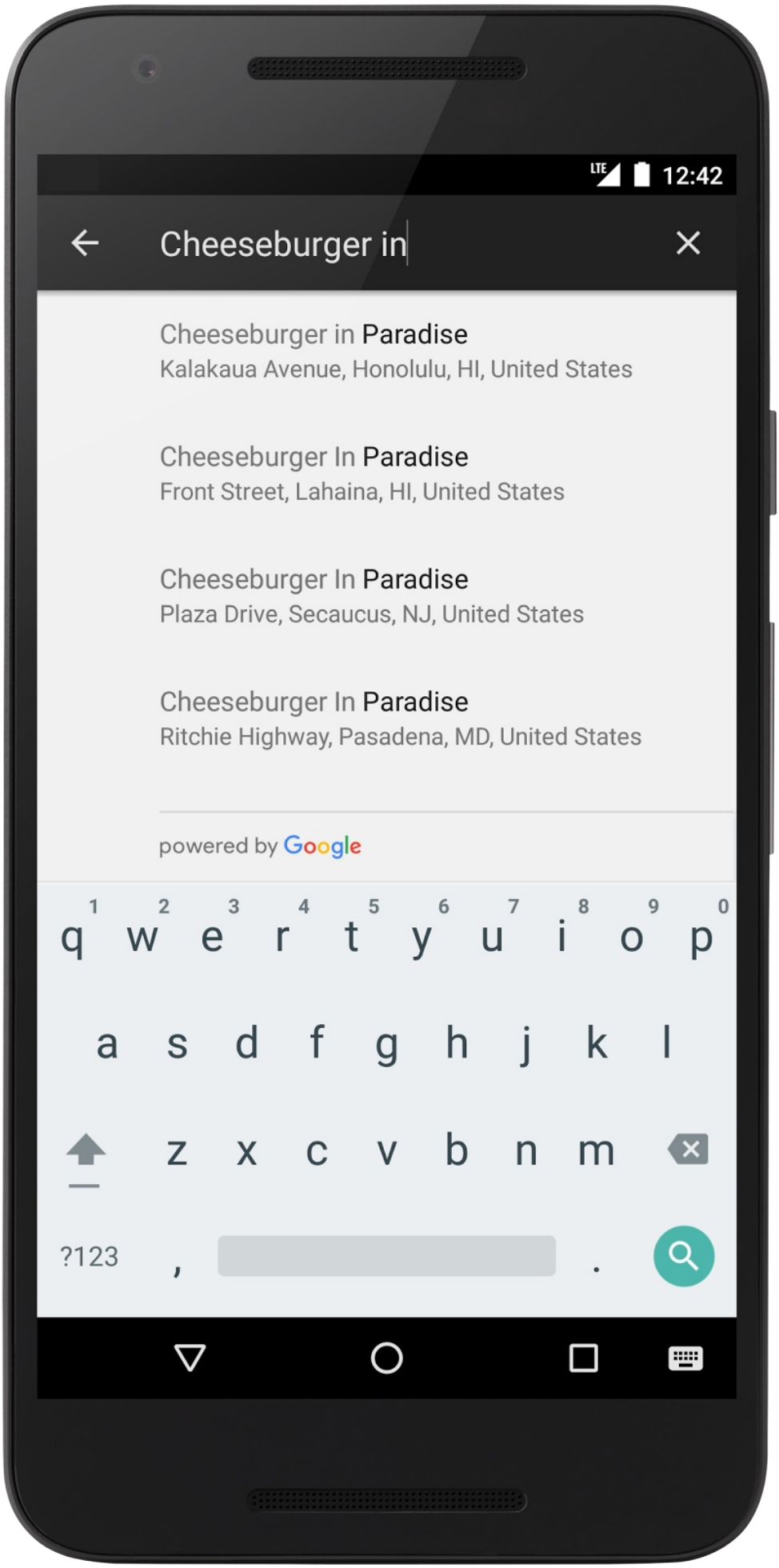
The Places API provides a `PlaceAutocomplete` service that returns `Place` predictions based on user search queries. As the user types, the autocomplete service returns suggestions for places such as businesses, addresses, and points of interest.

You can add the place-autocomplete service to your app in either of the following ways:

- Use the autocomplete UI, as described below. This approach saves development time and ensures a consistent user experience.
- Get place predictions programmatically, as described in Getting place predictions programmatically. With this approach, you can create a customized user experience.

## Adding an autocomplete UI element

The autocomplete UI is a search dialog with built-in autocomplete functionality. As a user enters search terms, the search dialog presents a list of predicted places to choose from. When the user makes a selection, a `Place` instance is returned. Your app can use the `Place` instance to get details about the selected place.


12:42
Cheeseburger in
Cheeseburger in Paradise
Kalakaua Avenue, Honolulu, HI, United States
Cheeseburger In Paradise
Front Street, Lahaina, HI, United States
Cheeseburger In Paradise
Plaza Drive, Secaucus, NJ, United States
Cheeseburger In Paradise
Ritchie Highway, Pasadena, MD, United States
powered by Google
?123

There are two ways to add the autocomplete UI to your app:

- Option 1: Embed a `PlaceAutocompleteFragment` fragment, as described below.
- Option 2: If you want to launch the autocomplete activity from someplace other than a search dialog, use an intent to launch the autocomplete activity, as described in Using an intent to launch the autocomplete activity.

## Embedding a PlaceAutocompleteFragment fragment

To add a `PlaceAutocompleteFragment` to your app, take the following steps:

1. Add a fragment to your activity's XML layout, with the `android:name` attribute set to `com.google.android.gms.location.places.ui.PlaceAutocompleteFragment`.

   ```
    <fragment
      android:id="@+id/place_autocomplete_fragment"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:name="com.google.android.gms.location.places.ui.PlaceAutocompleteFragme
   nt"
      />
   ```

   **Note:** By default, the fragment has no border or background. To provide a consistent visual appearance, nest the fragment within another layout element such as a `CardView`.
2. Add a listener to your activity or fragment.

   The `PlaceSelectionListener` handles returning a place in response to the user's selection. The following code creates a reference to the fragment and adds a listener to the `PlaceAutocompleteFragment`:

```
PlaceAutocompleteFragment autocompleteFragment =
      (PlaceAutocompleteFragment)getFragmentManager()
             .findFragmentById(R.id.place_autocomplete_fragment);
autocompleteFragment.setOnPlaceSelectedListener(new
                                PlaceSelectionListener() {
 @Override
 public void onPlaceSelected(Place place) {
     // TODO: Get info about the selected place.
     Log.i(TAG, "Place: " + place.getName());
 }

 @Override
 public void onError(Status status) {
     // TODO: Handle the error.
     Log.i(TAG, "An error occurred: " + status);
     }
 });
```

## Using an intent to launch the autocomplete activity

You might want your app to use a navigational flow that's different from what the autocomplete UI provides. For example, your app might trigger the autocomplete experience from an icon, rather than from a search field.

To launch the autocomplete activity from someplace other than the default autocomplete search dialog, use an intent:

1. Use `PlaceAutocomplete.IntentBuilder` to create an intent. Pass in the desired `PlaceAutocomplete` display mode—you can choose overlay or full-screen. The intent must call `startActivityForResult()`, passing in a request code that identifies your intent.

```
 int PLACE_AUTOCOMPLETE_REQUEST_CODE = 1;
 ...
 try {
     Intent intent =
             new PlaceAutocomplete.IntentBuilder(PlaceAutocomplete.MODE_FULLSCREEN
)
                     .build(this);
     startActivityForResult(intent, PLACE_AUTOCOMPLETE_REQUEST_CODE);
 } catch (GooglePlayServicesRepairableException e) {
     // TODO: Handle the error.
 } catch (GooglePlayServicesNotAvailableException e) {
     // TODO: Handle the error.
 }
```

2. Override the `onActivityResult()` method to receive the selected place. When a user has selected a place, check for the request code that you passed into your intent. For example:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == PLACE_AUTOCOMPLETE_REQUEST_CODE) {
        if (resultCode == RESULT_OK) {
            Place place = PlaceAutocomplete.getPlace(this, data);
            Log.i(TAG, "Place: " + place.getName());
        } else if (resultCode == PlaceAutocomplete.RESULT_ERROR) {
            Status status = PlaceAutocomplete.getStatus(this, data);
            // TODO: Handle the error.
            Log.i(TAG, status.getStatusMessage());

        } else if (resultCode == RESULT_CANCELED) {
            // The user canceled the operation.
        }
    }
}
```

## Restricting the search results

You can set the autocomplete search to bias results to a geographic region. You can also filter the results to one or more place types.

To bias autocomplete results to a geographic region, call `setBoundsBias()` off your app's `PlaceAutocompleteFragment` instance, or off your app's `IntentBuilder` instance. Pass in a `LatLngBounds` object.

The following code calls `setBoundsBias()` on a fragment instance. The code biases the fragment's autocomplete suggestions to a region of Sydney, Australia.

```
autocompleteFragment.setBoundsBias(new LatLngBounds(
        new LatLng(-33.880490, 151.184363),
        new LatLng(-33.858754, 151.229596)));
```

To filter autocomplete results to a specific place type, call `AutocompleteFilter.Builder` to create a new `AutocompleteFilter`. Call `setTypeFilter()` to set the filter to use. Then, pass the filter to the fragment or intent.

The following code sets an `AutocompleteFilter` on a `PlaceAutocompleteFragment`. The filter returns only results that have precise addresses.

```
AutocompleteFilter typeFilter = new AutocompleteFilter.Builder()
        .setTypeFilter(AutocompleteFilter.TYPE_FILTER_ADDRESS)
        .build();

autocompleteFragment.setFilter(typeFilter);
```

The following code sets the same `AutocompleteFilter` on an intent instead of on a fragment:

```
AutocompleteFilter typeFilter = new AutocompleteFilter.Builder()
        .setTypeFilter(AutocompleteFilter.TYPE_FILTER_ADDRESS)
        .build();

Intent intent =
        new PlaceAutocomplete.IntentBuilder(PlaceAutocomplete.MODE_FULLSCREEN)
                .setFilter(typeFilter)
                .build(this);
```

For information about place types, see the guide to place types and the `AutocompleteFilter.Builder` documentation.

To filter autocomplete results to a specific country:

1. Call `AutocompleteFilter.Builder` to create a new `AutocompleteFilte`r.
2. Call `setCountry()` to set the country code.
3. Pass the filter to a fragment or intent.

The following code sets an `AutocompleteFilter` on a `PlaceAutocompleteFragment`. The filter returns only results within the specified country.

```
AutocompleteFilter typeFilter = new AutocompleteFilter.Builder()
        .setCountry("AU")
        .build();

autocompleteFragment.setFilter(typeFilter);
```

## Getting place predictions programmatically

You can create a custom search UI instead of using the Places API UI. To do this, your app must get place predictions programmatically.

To get a list of predicted place names or addresses from the autocomplete service, call `GeoDataClient.getAutocompletePredictions()`. Pass in the following parameters:

- A query string containing the text typed by the user.
- A `LatLngBounds` object that biases the results to a specific area specified by latitude

and longitude bounds.
- *Optional:* To restrict the results to one or more types of place, include an `AutocompleteFilter` that contains a set of place types.

The API returns an `AutocompletePredictionBuffer` in a `Task`. The `AutocompletePredictionBuffer` contains a list of `AutocompletePrediction` objects that represent predicted places. If no known place corresponds to the query and the filter criteria, the buffer may be empty.

**Important:** To prevent a memory leak, release the `AutocompletePredictionBuffer` object when your app no longer needs it. Read more about handling buffers.
For each predicted place, you can call the following methods to retrieve place details:

- `getPrimaryText(CharacterStyle matchStyle)` returns the primary text that describes a place. This text is usually the name of the place, for example, "Eiffel Tower" or "123 Pitt Street."
- `getSecondaryText(CharacterStyle matchStyle)` returns the secondary text that describes a place. For example, "Avenue Anatole France, Paris, France," or "Sydney, New South Wales." Secondary text is useful as a second line of text when showing autocomplete predictions.
- `getFullText(CharacterStyle matchStyle)` returns the full text of a place description. This description combines the primary and secondary text. For example, "Eiffel Tower, Avenue Anatole France, Paris, France." To highlight sections of the description that match the search, include a `CharacterStyle` parameter. Set the parameter to `null` if you don't need any highlighting.
- `getPlaceId()` returns the place ID of the predicted place. A place ID is a textual identifier that uniquely identifies a place. You can use the place ID to retrieve the `Place` object later.
- `getPlaceTypes()` returns the list of place types associated with this place.

## About usage limits on place predictions

When you get a place prediction programmatically, usage limits apply. In particular, the `GeoDataClient.getAutocompletePredictions()` method is subject to tiered query limits. For details, see Usage Limits.

## Displaying attributions in your app

- If your app uses the autocomplete service programmatically, your UI must display a "Powered by Google" attribution, or your UI must appear within a Google-branded map.
- If your app uses the autocomplete UI, you don't need to do anything special, because the required attribution is displayed by default.

- If you retrieve and display additional place information after getting a place by ID, you must display third-party attributions.

For more details, see Displaying Attributions.

# Getting place photos

You can use the Places API to request place photos to display in your app. The photos come from several sources, including business owners and Google+ users. To retrieve photos for a place, take the following steps:

1. Call `GeoDataClient.getPlacePhotos()`, passing a string with a place ID. This method returns a `PlacePhotoMetadataResult` instance in a `Task`.
2. Call `getPhotoMetadata()` on the `PlacePhotoMetadataResult` instance. The method returns a `PlacePhotoMetadataBuffer` that holds a list of `PlacePhotoMetadata` instances. The list includes one `PlacePhotoMetadata` instance for each photo.
3. Call `get()` on the `PlacePhotoMetadataBuffer` instance. Pass in an integer to retrieve the `PlacePhotoMetadata` instance at the given index.
    - To return a bitmap image, call either `PlacePhotoMetadata.getPhoto()`, or `PlacePhotoMetadata.getScaledPhoto()` on a `PlacePhotoMetadata` instance.
    - To return attribution text, call `PlacePhotoMetadata.getAttributions()` on a `PlacePhotoMetadata` instance.

The Places API requires network access, therefore all calls must be made in the background, off the main UI thread. For this reason, retrieve the photos in the background asynchronously.

## About usage limits on place photos

When you get place photos programmatically, usage limits apply. Retrieving an image costs one unit of quota. There are no usage limits for retrieving photo metadata.

For more information, see Usage Limits.

## Displaying attributions in your app

In most cases, place photos can be used without attribution, or the required attribution is included as part of the image. However, if a `PlacePhotoMetadata` instance includes an attribution, you must include the attribution in your app wherever you display the image.

For more information, see Displaying Attributions.

# Using the place ID

A *place ID* is a textual identifier that uniquely identifies a place. To retrieve a place's ID:

- Call `Place.getId()`.
- The place-autocomplete service also returns a place ID for each place that matches the supplied search query and filter.

Store the place ID and use it to retrieve the `Place` object again later.

To get a place by its ID:

1. Call `GeoDataClient.getPlaceById()`, passing in one or more place IDs.
2. The API returns a `PlaceBuffer` in a `Task`. The `PlaceBuffer` data structure contains a list of `Place` objects that match the supplied place IDs.

```
 mGeoDataClient.getPlaceById(placeId).addOnCompleteListener(new OnCompleteListener
<PlaceBufferResponse>() {
     @Override
     public void onComplete(@NonNull Task<PlaceBufferResponse> task) {
         if (task.isSuccessful()) {
             PlaceBufferResponse places = task.getResult();
             Place myPlace = places.get(0);
             Log.i(TAG, "Place found: " + myPlace.getName());
             places.release();
         } else {
             Log.e(TAG, "Place not found.");
         }
     }
 });
```

# Getting place details

The `Place` object provides information about a specific place. You can get a `Place` object in the following ways:

- Call `PlaceDetectionClient.getCurrentPlace()`. For details about this process, see Current Place.
- Add the `PlacePicker` UI, then call `PlacePicker.getPlace()`.
- Call `GeoDataClient.getPlaceById()`. For more information, see Get a place by ID.

Use the following methods to retrieve data from a `Place` object:

- `getName()`: The place's name.
- `getAddress()`: The place's address, in human-readable format.
- `getID()`: The textual identifier for the place, as described in Using the place ID.

- `getPhoneNumber()` : The place's phone number.
- `getWebsiteUri()` : The URI of the place's website, if known. This website is maintained by the business or other entity associated with the place. Returns `null` if no website is known.
- `getLatLng()` : The geographical location of the place, specified as latitude and longitude coordinates.
- `getViewport()` : A viewport, returned as a `LatLngBounds` object, useful for displaying the place on a map. If the size of the place is not known, this method may return `null` .
- `getLocale()` : The locale for which the name and address are localized.
- `getPlaceTypes()` : A list of place types that characterize this place. For a list of available place types, see the `Place` constant documentation.
- `getPriceLevel()` : The price level for this place, returned as an integer with values ranging from 0 (cheapest) to 4 (most expensive).
- `getRating()` : An rating of the place, returned as a `float` with values ranging from 1.0 to 5.0, based on aggregated user reviews.

# Related practical

The related practical documentation is in 8.1: Using the Places API.

# Learn more

Android developer documentation:

- Getting Started with the Places API for Android
- `Place`
- `PlacePicker`
- `GeoDataClient`
- `PlaceDetectionClient`

# 9.1: Google Maps API

**Contents:**



The Google Maps API for Android lets you include a Google Map in your app. You can customize the look and feel of the map, as well as the map's behavior. For example, you can change the color of certain features to highlight them, limit the area a user can scroll in, display the user's location in real time, and much more.

To use the Google Maps API, you must register your app in the Google API Console and obtain an API key.

## GoogleMap objects

A Google Map is represented by a `GoogleMap` object, plus a `MapFragment` object or a `MapView` object. The `GoogleMap` object allows you to programmatically interact with the map. The `MapFragment` and `MapView` objects represent the Android view components into which the map is loaded.

The `GoogleMap` object handles the following operations for you:

- Connecting to the Google Maps service.
- Downloading map imagery.
- Displaying the map on the device screen.
- Displaying controls such as pan and zoom.
- Responding to pan and zoom gestures.

You can control the behavior of maps with objects and methods of the API. For example, `GoogleMap` has callback methods that respond to keystrokes and touch gestures on the map. You can set marker icons on your map and add overlays to the map, using objects you provide to `GoogleMap`.

Google Maps are loaded as a set of map "tiles," which are square pieces of imagery tied to geographic coordinates.

## The MapFragment class

`MapFragment`, a subclass of the Android `Fragment` class, allows you to place a map in an Android fragment. The `MapFragment` object acts as a container for the map and provides access to the `GoogleMap` object.

## The MapView class

`MapView`, a subclass of the `View` class, allows you to place a map in a view instead of placing the map in a fragment. The `MapView` object acts as a container for the map and provides access to the `GoogleMap` object.

Your code must override the following lifecycle methods for the activity:

- `onCreate()`
- `onStart()`
- `onResume()`
- `onPause()`
- `onStop()`
- `onDestroy()`
- `onSaveInstanceState()`
- `onLowMemory()`

The overridden methods should call the corresponding lifecycle methods of `MapView`. For example, you would override and forward the `onResume()` method of the activity as follows:

```
@Override
protected void onResume() {
    super.onResume();
    mMapView.onResume();
}
```

## Create the Google Map

To create a `GoogleMap` instance, you must do the following:

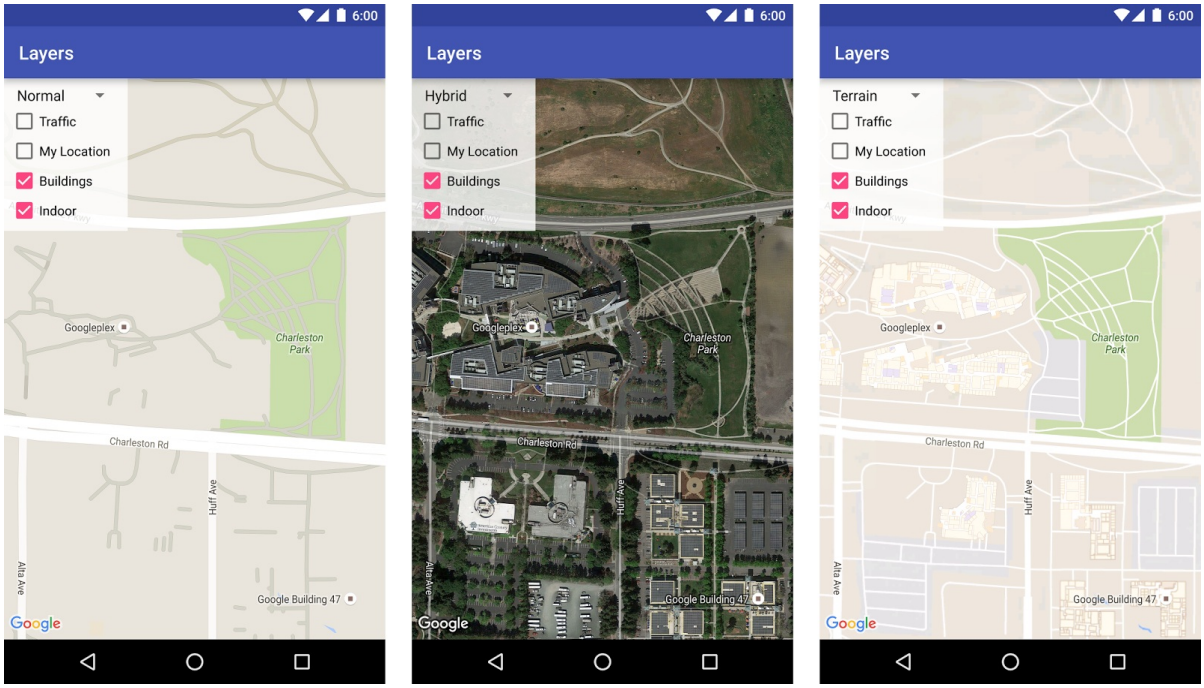
1. Make your activity implement the `OnMapReadyCallback` interface and override the required `onMapReady()` method.
2. Get a reference to the container (either a `MapFragment` or a `MapView`) and call `getMapAsync()` on it.

After the map is ready, the `onMapReady()` callback is triggered with a passed-in `GoogleMap` object, which you can then save in a member variable. This callback is where all map operations occur.

# Map types

The Google Maps Android API offers five types of maps:

- *Normal* : Typical road map. Shows roads, some features built by humans, and important natural features such as rivers. Road and feature labels are also visible.
- *Hybrid* : Satellite photograph data with road maps added. Road and feature labels are also visible.
- *Satellite* : Satellite photograph data. Road and feature labels are not visible.
- *Terrain* : Topographic data. The map includes colors, contour lines and labels, and perspective shading. Some roads and labels are also visible.
- *None* : No tiles. The map is rendered as an empty grid with no tiles loaded.




To set the type of a map, call the `GoogleMap` object's `setMapType()` method. Pass in one of the type constants defined in `GoogleMap`. For example, here's how to display a hybrid map:

```
GoogleMap map;
// Sets the map type to be "hybrid"
mMap.setMapType(GoogleMap.MAP_TYPE_HYBRID);
```

# Configuring the initial map state

The Maps API allows you to configure the initial state of the map. You can specify the following:

- The camera position, including location, zoom, bearing, and tilt. See the Camera and view section below.
- The map type.
- Whether the zoom buttons or compass appear on the screen.
- The gestures a user can use to manipulate the map.
- Whether lite mode is enabled or not.

If you add the map to your activity's layout file, you use XML to configure the map's initial state. If you add the map in your activity, you configure the map's initial state programmatically.

## Defining the initial state in XML

The Maps API defines a set of custom XML attributes for a `MapFragment` or a `MapView` object:

- `mapType` : Specify the type of map to display. Valid values include `none` , `normal` , `hybrid` , `satellite` , and `terrain` .
- `cameraTargetLat` , `cameraTargetLng` , `cameraZoom` , `cameraBearing` , `cameraTilt` : Specify the initial camera position. See Camera and View for more details on camera position and properties.
- `uiZoomControls` , `uiCompass` : Specify whether you want the zoom controls and compass to appear on the map. See `UiSettings` for more details.
- `uiZoomGestures` , `uiScrollGestures` , `uiRotateGestures` , `uiTiltGestures` : Specify which gestures are enabled or disabled for the user's interaction with the map. See `UiSettings` for more details.
- `zOrderOnTop` : User `zOrderOnTop` if you want the map to cover everything, including controls and other child views of the `MapView` . See `SurfaceView.setZOrderOnTop(boolean)` for more details.
- `useViewLifecycle` : Specifies whether the lifecycle of the map should be tied to the fragment's view, or tied to the fragment itself. Only valid with a `MapFragment` . See the `MapFragment` reference for more details on why you would use `useViewLifeCycle` .
- `liteMode` : A *lite-mode* map is a bitmap image of a map that supports a subset of the functionality supplied by the full API. The default value of this attribute is `false` .

To use these custom attributes within your XML layout file, add the following namespace declaration to the fragment or view:

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

The following XML code shows how to configure a `MapFragment` . These same attributes can be applied to a `MapView` .

```
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:map="http://schemas.android.com/apk/res-auto"
  android:name="com.google.android.gms.maps.SupportMapFragment"
  android:id="@+id/map"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  map:cameraBearing="112.5"
  map:cameraTargetLat="-33.796923"
  map:cameraTargetLng="150.922433"
  map:cameraTilt="30"
  map:cameraZoom="13"
  map:mapType="normal"
  map:uiCompass="false"
  map:uiRotateGestures="true"
  map:uiScrollGestures="false"
  map:uiTiltGestures="true"
  map:uiZoomControls="false"
  map:uiZoomGestures="true"/>
```

## Define the initial state programmatically

If you plan to add, remove, or replace fragments or views programmatically, create them in your activity as follows:

- If you are using a `MapFragment`, use the `MapFragment.newInstance(GoogleMapOptions options)` static factory method to construct the fragment.
- If you are using a `MapView`, use the `MapView(Context, GoogleMapOptions)` constructor.

To configure the initial state of the fragment or view, pass in a `GoogleMapOptions` object that specifies your options. The options available to you are the same as the options available via XML. Create a `GoogleMapOptions` object like this:

```
GoogleMapOptions options = new GoogleMapOptions();
```

And then configure the object as follows:

```
options.mapType(GoogleMap.MAP_TYPE_SATELLITE)
    .compassEnabled(false)
    .rotateGesturesEnabled(false)
    .tiltGesturesEnabled(false);
```

# Businesses and other points of interest

By default, points of interest (POIs) appear on the base map along with their corresponding icons. POIs include parks, schools, government buildings, and more. *Business* POIs represent businesses such as shops, restaurants, or hotels. When the map type is `normal`, business POIs appear on the map in addition to regular POIs.

A POI corresponds to a place, as defined in the Google Places API. For example, recreational parks are POIs, but things like water fountains are usually not POIs unless they're of national or historic significance.

POIs appear on the map by default, but there is no default on-click UI. That is, the API does not automatically display an info window or any other user interface when the user taps a POI.

If you want to respond to a user tapping on a POI, use an `OnPoiClickListener` as shown in the following code sample:

```
public class OnPoiClickDemoActivity extends FragmentActivity
    implements OnMapReadyCallback, GoogleMap.OnPoiClickListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.poi_click_demo);

        SupportMapFragment mapFragment = (SupportMapFragment)
                getSupportFragmentManager().findFragmentById(R.id.map);
        mapFragment.getMapAsync(this);
    }

    @Override
    public void onMapReady(GoogleMap map) {
        map.setOnPoiClickListener(this);
    }

    @Override
    public void onPoiClick(PointOfInterest poi) {
        Toast.makeText(getApplicationContext(), "Clicked: " +
                        poi.name + "\nPlace ID:" + poi.placeId +
                        "\nLatitude:" + poi.latLng.latitude +
                        " Longitude:" + poi.latLng.longitude,
                Toast.LENGTH_SHORT).show();
    }
}
```

As the above sample shows, you set the `OnPoiClickListener` on the map by calling `GoogleMap.setOnPoiClickListener(OnPoiClickListener)`. When a user taps on a POI, your app receives an `onPoiClick(PointOfInterest)` event indicating the point of interest (POI) that the user tapped.

The `PointOfInterest` object contains the latitude/longitude coordinates, place ID, and name of the POI.

# Lite mode

A lite-mode map supports all the map types (normal, hybrid, satellite, terrain) and a subset of the functionality supplied by the full API. Lite mode is useful when you want to provide several maps in a list, or a map that is too small to support meaningful interaction.

Users can't zoom or pan a lite-mode map. In a lite-mode map, icons let users access the full Google Maps mobile app and request directions.

Set a `GoogleMap` to lite mode in XML:

```
map:liteMode="true"
```

Or in the `GoogleMapOptions` object:

```
GoogleMapOptions options = new GoogleMapOptions().liteMode(true);
```

# Map styles

You can customize the presentation of the standard Google Map styles, changing the visual display of features like roads, parks, businesses, and other points of interest. This means that you can emphasize particular components of the map or make the map look good with your app.

Styling works only on the `normal` map type. Styling does not affect indoor maps.

To style your map, use a JSON file generated by the Google Maps APIs Styling Wizard. In addition to changing the appearance of features, you can also hide features completely.

```
[
  {
    "featureType": "all",
    "stylers": [
      { "color": "#C0C0C0" }
    ]
  },{
    "featureType": "road.arterial",
    "elementType": "geometry",
    "stylers": [
      { "color": "#CCFFFF" }
    ]
  },{
    "featureType": "landscape",
    "elementType": "labels",
    "stylers": [
      { "visibility": "off" }
    ]
  }
]
```

Styled maps use two concepts to apply colors and other style changes to a map:

- *Selectors* specify the geographic components that you can style on the map. These geographic components include roads, parks, bodies of water, and more, as well as their labels. The selectors include features and elements, specified as `featureType` and `elementType` properties.
- *Stylers* are color and visibility properties that you can apply to map elements. They define the displayed color through a combination of hue, color, and lightness/gamma values.

See the style reference for a detailed description of the JSON styling options.

To style your map, first save the JSON file in your raw resource directory ( `raw/res` ), which is used to store arbitrary files. Then call `GoogleMap.setMapStyle()` . Pass in a `MapStyleOptions` object that loads your resource from the raw directory:

```
try {
   // Customize the styling of the base map using a JSON object defined
   // in a raw resource file.
   boolean success = googleMap.setMapStyle(
           MapStyleOptions.loadRawResourceStyle(
                   this, R.raw.map_style));

   if (!success) {
       Log.e(TAG, "Style parsing failed.");
   }
} catch (Resources.NotFoundException e) {
   Log.e(TAG, "Can't find style file. Error: ", e);
}
```

# Camera and view

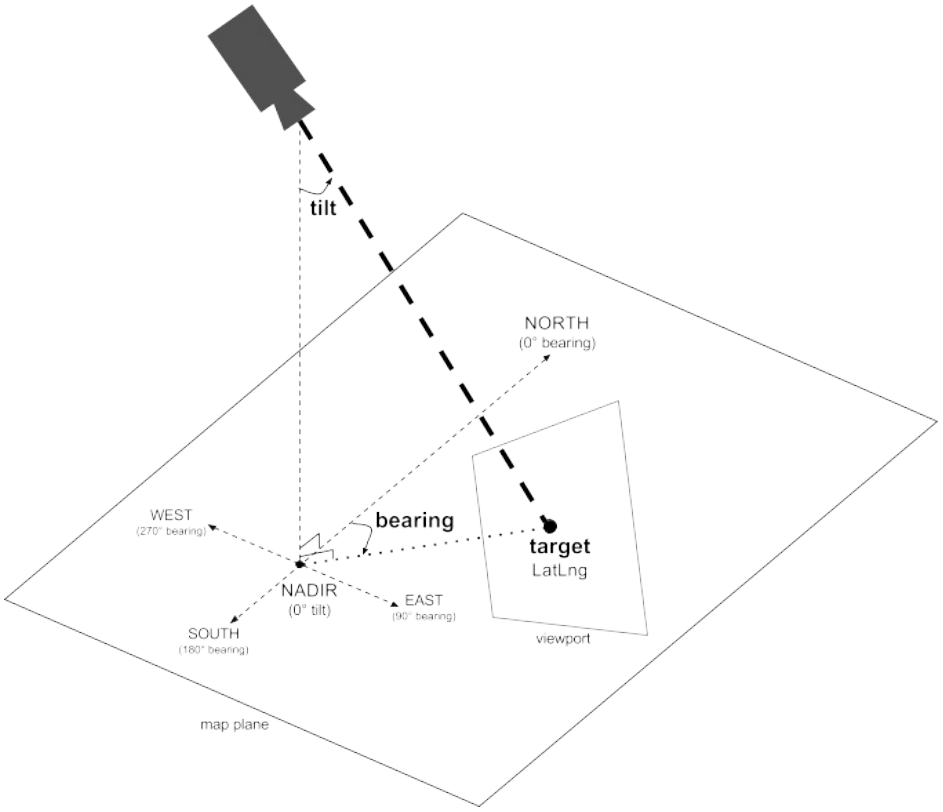
Like Google Maps on the web, the Google Maps Android API uses a Mercator projection to represent the world's curved surface on your device's flat screen. In the east and west directions, the map repeats infinitely as the Earth seamlessly wraps around on itself. In the north and south directions, the map is limited to approximately 85 degrees north and 85 degrees south. You can see parts of Antarctica, but not the North and South Poles.

The Google Maps Android API allows you to change the user's view of the map by modifying the map's camera. Changes to the camera don't change markers, overlays, or graphics that you add.

Because you can listen for user gestures on the map, you can change the map in response to user requests. For example, the callback method `OnMapClickListener.onMapClick()` responds to a single tap on the map. The method receives the latitude and longitude of the tap location, which you can use to pan or zoom the camera to that point. Similar methods are available for responding to taps on a marker's bubble or for responding to a drag gesture on a marker.

## The camera position

The map view is modeled as a camera looking down on a flat plane. The following properties specifiy the camera's position (and hence the rendering of the map): target (latitude/longitude location), bearing, tilt, and zoom.




## Target (location)

The camera target is the location of the center of the map, specified as latitude and longitude.

## Bearing (orientation)

The camera bearing is the direction in which a vertical line on the map points, measured in degrees clockwise from north. Someone driving a car often turns a road map to align it with their direction of travel, while hikers using a map and compass usually orient the map so that a vertical line points north. The Maps API lets you change a map's alignment or bearing. For example, a bearing of 90 degrees results in a map where the upwards direction points due east.

## Tilt (viewing angle)

By default, the camera points down onto the map and no perspective is applied. You can tilt the camera to change the viewing angle, which allows the map to appear in perspective, where more distant features appear smaller.

## Zoom

The zoom level of the camera determines the scale of the map. At higher zoom levels more detail can be seen on the screen, while at lower zoom levels more of the world can be seen on the screen.

At zoom level 0, the scale of the map is such that the entire world has a width of approximately 256 dp (density-independent pixels).

Increasing the zoom level by 1 doubles the width of the world on the screen. Hence at zoom level *N* , the width of the world is approximately 256 * 2 *N* dp. That is, at zoom level 2, the whole world is approximately 1024 dp wide.

The zoom level doesn't need to be an integer. The range of zoom levels that the map permits depends on factors including location, map type, and screen size. The following list shows the approximate level of detail you can expect to see at each zoom level:

- `1` : World
- `5` : Landmass/continent
- `10` : City
- `15` : Streets
- `20` : Buildings

The following images show different zoom levels:

A map at zoom level 5.

A map at zoom level 15.

A map at zoom level 20.

# Controls and gestures

Using the Google Maps Android API, you can customize how users can interact with your map, by specifying which of the built-in UI components appear on the map and which gestures are allowed.

## UI controls

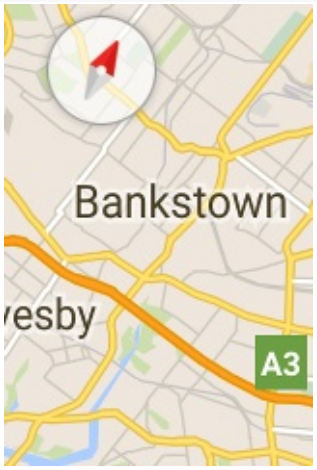
The Maps API offers UI controls that are similar to the UI controls in the Google Maps app on your Android phone. To toggle the visibility of these controls, use the `UiSettings` class. To obtain a `UiSettings` object from a `GoogleMap`, use the `GoogleMap.getUiSettings()` method. Changes made in the `UiSettings` class are immediately reflected on the map. For example, you may want to disable the compass icon that allows the users to toggle their bearing.

## Zoom controls

The Maps API provides zoom controls that appear in the bottom-right corner of the map. These controls are disabled by default. To enable the controls, call `UiSettings.setZoomControlsEnabled(true)`.

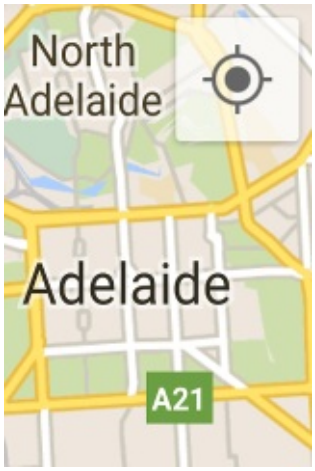

## Compass

The Maps API provides a compass graphic that sometimes appears in the top-left corner of the map. The compass only appears when the camera is oriented such that it has a non-zero bearing or non-zero tilt. The user can tap the compass to reset the camera's tilt and bearing to vertical and north-facing.

To disable the compass graphic from appearing altogether, call `UiSettings.setCompassEnabled(boolean)`. However, you can't force the compass to always be shown. 


## My Location button

The **My Location** button appears in the top-right corner of the screen only when the **My Location** layer is enabled. The **My Location** layer is covered later in this lesson.

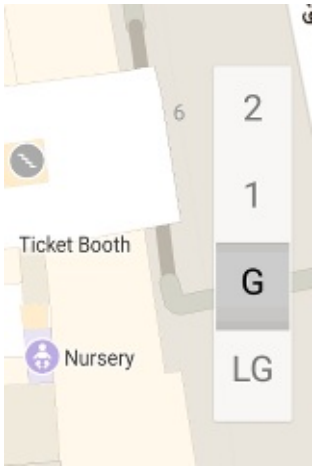



## Level picker

When the user views an indoor map, a level picker (floor picker) appears by default near the center-right edge of the screen. When two or more indoor maps are visible, the level picker applies to the building that's currently in focus, which is typically the building nearest the center of the screen. Each building starts out showing a default floor level, which is set when the building is added to Google Maps. Users can choose a different level by selecting it from the picker.

To disable or enable the level-picker control, call `GoogleMap.getUiSettings().setIndoorLevelPickerEnabled(boolean)`.

This is useful if you want to replace the default level picker with your own level picker, for example to create an augmented-reality scavenger hunt in your home.




## Map toolbar

By default, a toolbar appears at the bottom-right of the map when a user taps a marker. The toolbar gives the user quick access to the Google Maps mobile app.

In a lite-mode map, the toolbar persists independently of the user's actions. In a fully interactive map, the toolbar slides in when the user taps a marker and slides out again when the marker is no longer in focus.

The toolbar icons provide access to a map view or a direction-request in the Google Maps mobile app. When a user taps an icon on the toolbar, the API builds an intent to launch the corresponding activity in the Google Maps mobile app.

To enable and disable the toolbar, call `UiSettings.setMapToolbarEnabled(boolean)`.

The toolbar is visible on the right side of the above screenshot. Depending on the content of the map, zero, one, or both of the intent icons appear on the map (if the Google Maps mobile app supports the intents).

## Zoom gestures

The map responds to a variety of gestures that change the zoom level of the camera:

- Double-tap to increase the zoom level by 1 (zoom in).
- Two-finger tap to decrease the zoom level by 1 (zoom out).
- Two-finger pinch to zoom out, or two-finger stretch to zoom in.
- One-finger zoom: Double-tap but don't release on the second tap. Then slide the finger up to zoom out, or slide the finger down to zoom in.

To disable zoom gestures, call `UiSettings.setZoomGesturesEnabled(boolean)`. If zoom gestures are disabled, a user can still use the zoom controls to zoom in and out.

## Scroll (pan) gestures

A user can scroll (pan) around the map by dragging the map with their finger. To disable scrolling, call `UiSettings.setScrollGesturesEnabled(boolean)`. For example, you may want to limit the map to one specific area of interest.

## Tilt gestures

A user can tilt the map by placing two fingers on the map and moving them down or up together to increase or decrease the tilt angle respectively. To disable tilt gestures, call `UiSettings.setTiltGesturesEnabled(boolean)`.

## Rotate gestures

A user can rotate the map by placing two fingers on the map and applying a rotate motion. To disable rotation, call `UiSettings.setRotateGesturesEnabled(boolean)`.

# Indoor maps

At high zoom levels, the map shows floor plans for indoor spaces such as airports, shopping malls, large retail stores, and transit stations. These floor plans, called indoor maps, are displayed for the `normal` and `satellite` map types ( `GoogleMap.MAP_TYPE_NORMAL` and `GoogleMap.MAP_TYPE_SATELLITE` ). Floor plans are automatically enabled when the user zooms in, and they fade away when the map is zoomed out.

Here's a summary of the indoor maps functionality in the API:

- To disable indoor maps, call `setIndoorEnabled(false)` on the `GoogleMap` object. By default, indoor maps are enabled.
- To disable the level-picker control that appears when you zoom in on a building, call `getUiSettings().setIndoorLevelPickerEnabled(false)`.
- To set a listener that's called when the state of an indoor map changes, use the `OnIndoorStateChangeListener` interface. The state changes when a new building comes into focus (meaning that a new building is in the center of the screen and zoomed in). The state also changes when a user uses the level-picker control to switch to a new level in a building.
- To show the building that's in focus, use `GoogleMap.getFocusedBuilding()`. You can then find the currently active level by calling `IndoorBuilding.getActiveLevelIndex()`. Refer to the reference documentation to see all the information available in the `IndoorBuilding` and `IndoorLevel` objects.

Styling of the base map does not affect indoor maps.

Indoor maps (floor plans) are available in select locations. If floor-plan data is not available for a building that you would like to highlight in your app, you can:

- Add floor plans to Google Maps so that your floor plans are available to all users of Google Maps.
- Display a floor plan as a ground overlay or tile overlay on your map so that only users of your app can view your floor plans.

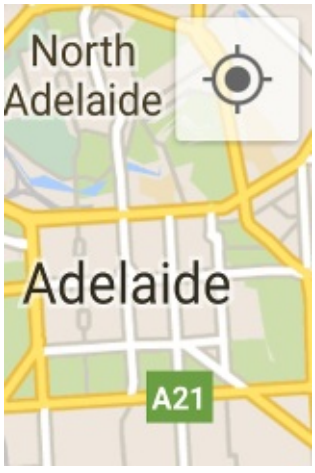
# The My Location layer

You can use the **My Location** layer and the **My Location** button to show your user their current position on the map.

**Note:** Before you enable the **My Location** layer, make sure you have the required runtime location permission. You need either `Manifest.permission.ACCESS_FINE_LOCATION` or `Manifest.permission.ACCESS_COARSE_LOCATION`, depending on your required precision. After the `onMapReady()` callback is triggered, enable the **My Location** layer on the map:

```
mMap.setMyLocationEnabled(true);
```

When the **My Location** layer is enabled, the **My Location** button appears in the top-right corner of the map. When a user taps the button, the camera centers the map on the current location of the device, if the location is known. The location is indicated on the map by a small blue dot if the device is stationary, or as a chevron if the device is moving.




A tap on the **My Location** button also triggers the `GoogleMap.OnMyLocationButtonClickListener`, which you can override.

To prevent the **My Location** button from appearing altogether, call `UiSettings.setMyLocationButtonEnabled(false)`.

**Note:** The **My Location** layer does not return any data. To access location data programmatically, use the Location API.

# Map markers

Markers identify locations on the map. The default marker uses the standard Google Maps icon:  .

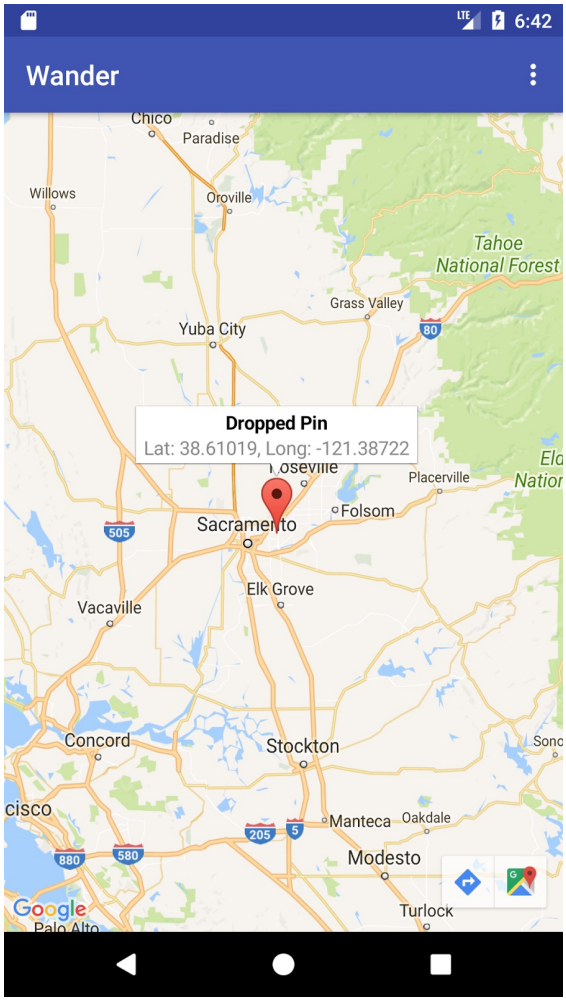
You can use the API to change the icon's color, image, or anchor point. Markers are objects of type `Marker`, and you add them to the map using the `GoogleMap.addMarker(markerOptions)` method.

Markers are designed to be interactive. They receive click-events by default, and are often used with event listeners to bring up info windows. Setting a marker's draggable property to `true` allows the user to change the position of the marker. (Touch & hold the marker to make it movable.)

By default, when a user taps a marker, the map toolbar appears at the bottom-right of the map, giving the user quick access to the Google Maps mobile app.

For more information on adding and customizing markers see Markers.

# Info windows




An info window displays text or images in a pop-up window that overlays the map. Info windows are always anchored to a marker. Their default behavior is to open when the marker is tapped. Only one info window is displayed at a time.

The default info window contains the title in bold, with the (optional) snippet text below the title.

To customize the content and design of info windows:

1. Create a concrete implementation of the `InfoWindowAdapter` interface, which is described below.
2. Call `GoogleMap.setInfoWindowAdapter()` with your `InfoWindowAdapter` implementation.

The `InfoWindowAdapter` interface contains two methods for you to implement:

- The `getInfoWindow(Marker)` method lets you provide a view that's used for the entire info window.
- The `getInfoContents(Marker)` method lets you customize the info-window content, while keeping the default frame and background.

The API first calls `getInfoWindow(Marker)`. If the method returns `null`, the API calls `getInfoContents(Marker)`. If this method also returns `null`, the API uses the default info window.

# Shapes

The Google Maps API for Android offers ways for you to draw shapes over your maps to customize them for your app.

- A *polyline* is a series of connected line segments that can form any shape you want. You can use polylines to mark paths and routes on the map.
- A *polygon* is an enclosed shape that can be used to mark areas on the map.
- A *circle* is a geographically accurate projection of a circle on the Earth's surface drawn on the map.

You can customize a shape's appearance by altering properties. For more information, see Shapes.

# Ground overlays

A ground overlay is an image that is fixed on the map at a given point. Unlike markers, ground overlays are oriented against the Earth's surface rather than against the screen. Rotating, tilting, or zooming the map changes the orientation of the image. Ground overlays are useful for adding single images. If you want to add extensive imagery that covers a large portion of the map, use a tile overlay, as described below.

For more information, see Ground Overlays.

# Tile overlays

A Google Map is split up into square tiles that are loaded as they are needed. The scale of the map loaded into each tile depends on the zoom level. A `TileOverlay` object defines a set of images that are added on top of the base map tiles.

Provide the tiles for each zoom level that you want to support. If you have enough tiles at multiple zoom levels, you can supplement Google's map data for the entire map.

Tile overlays are useful when you want to add extensive imagery to the map, typically covering large geographical areas. In contrast, ground overlays are useful when you wish to fix a single image at one area on the map.

You can use transparent tile overlays to add extra features to the map. To make tile overlays transparent, set a transparency factor on the tile overlay programmatically or provide transparent tile images.

For more information, see Tile Overlays.

# Listening to map events

Using the Google Maps Android API, you can listen to events on the map. The API includes support for the following types of map events:

- Click-events, touch & hold events
- Camera-change events
- Point of interest (POI) events, business POI events
- Indoor-map events
- Marker events, info-window events
- Shape events, overlay events

For more information on reacting to these events in your app, see Events.

# Street View

Google Street View provides panoramic 360-degree views from designated roads throughout its coverage area.

The `StreetViewPanorama` class models the Street View panorama in your app. Within your UI, a panorama is represented by a `StreetViewPanoramaFragment` object or `StreetViewPanoramaView` object.

Each Street View panorama is an image, or set of images, that provides a full 360-degree view from a single location. Images conform to the equirectangular (or "plate carrée") projection, which contains 360 degrees of horizontal view (a full wrap-around) and 180 degrees of vertical view (from straight up to straight down). The resulting 360-degree panorama defines a projection on a sphere with the image wrapped to the two-dimensional surface of that sphere.

The coverage available through the Google Maps Android API v2 is the same as the coverage available for the Google Maps app on your Android device. For an interactive map that shows areas that Street View supports, see About Street View.

Unlike with a map, it's not possible to configure the initial state of the Street View panorama via XML. However, you can configure the panorama programmatically by passing in a `StreetViewPanoramaOptions` object containing your specified options.

If you use a `StreetViewPanoramaFragment` object:

- Use the `StreetViewPanoramaFragment.newInstance(StreetViewPanoramaOptions options)` static factory method to construct the fragment and pass in your custom configured options.

If you use a `StreetViewPanoramaView` object:

- Use the `StreetViewPanoramaView(Context, StreetViewPanoramaOptions)` constructor and pass in your configuration options.


# Related practical

The related practical documentation is in 9.1 P: Adding a Google Map to your app.

# Learn more

Android developer documentation:

- Getting Started with the Google Maps Android API

- Adding a Map with a Marker
- Map Objects
- Adding a Styled Map
- Markers
- Shapes
- Ground Overlays
- Tile Overlays
- Events
- Street View

Reference documentation:

- `GoogleMap`
- `MapFragment`
- `StreetViewPanoramaFragment`

# 10.1: Custom views

**Contents:**



Android offers a large set of `View` subclasses that you can use to construct a UI that enables user interaction and displays information in your app. In the Android Developer Fundamentals course you learned how to use many of these subclasses, such as `Button`, `TextView`, `EditText`, `ImageView`, `CheckBox`, `RadioButton`, and `Spinner`.

However, if none of these meet your needs, you can create a `View` subclass known as a *custom* view. A custom view is a subclass of `View`, or of any `View` subclass (such as `Button`), that extends or replaces its parent's functionality.

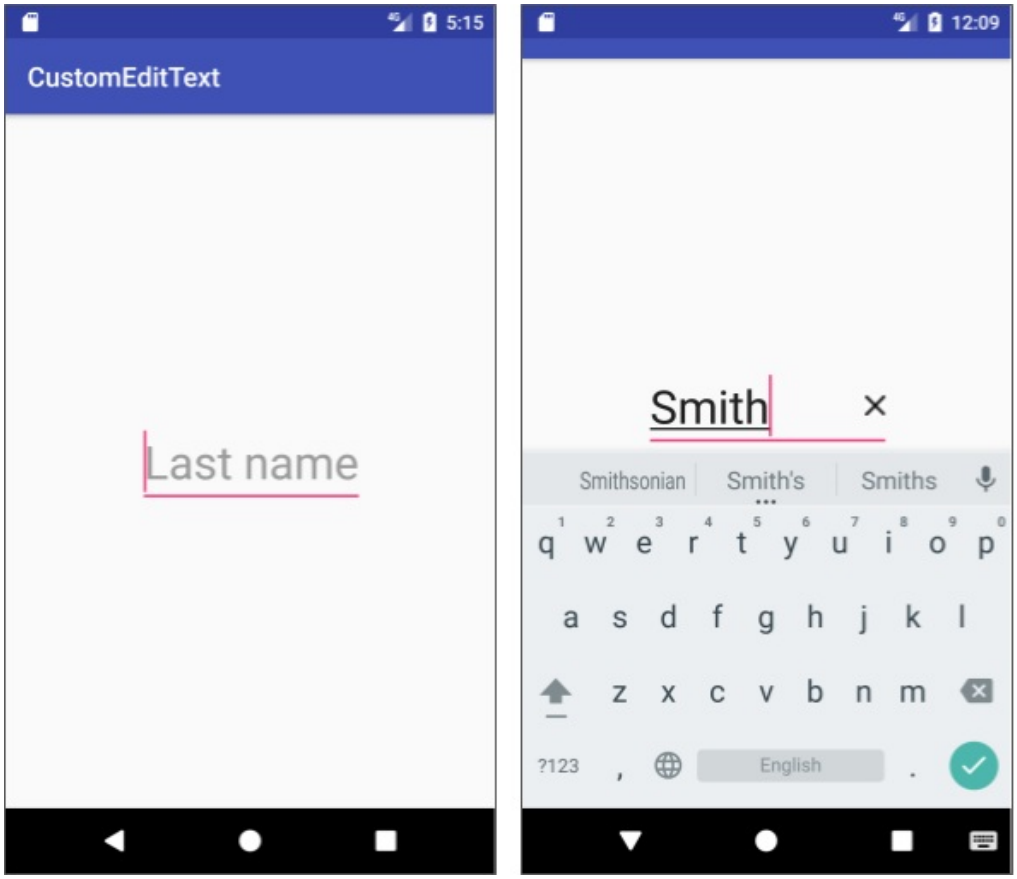
Once you have created a custom view, you can add it to different layouts in XML or programmatically. This chapter explains how to create and use custom views.

## Understanding custom views

Views are the basic building blocks of an app's UI. The `View` class provides many subclasses, referred to as *UI widgets*, that cover many of the needs of a typical Android app.

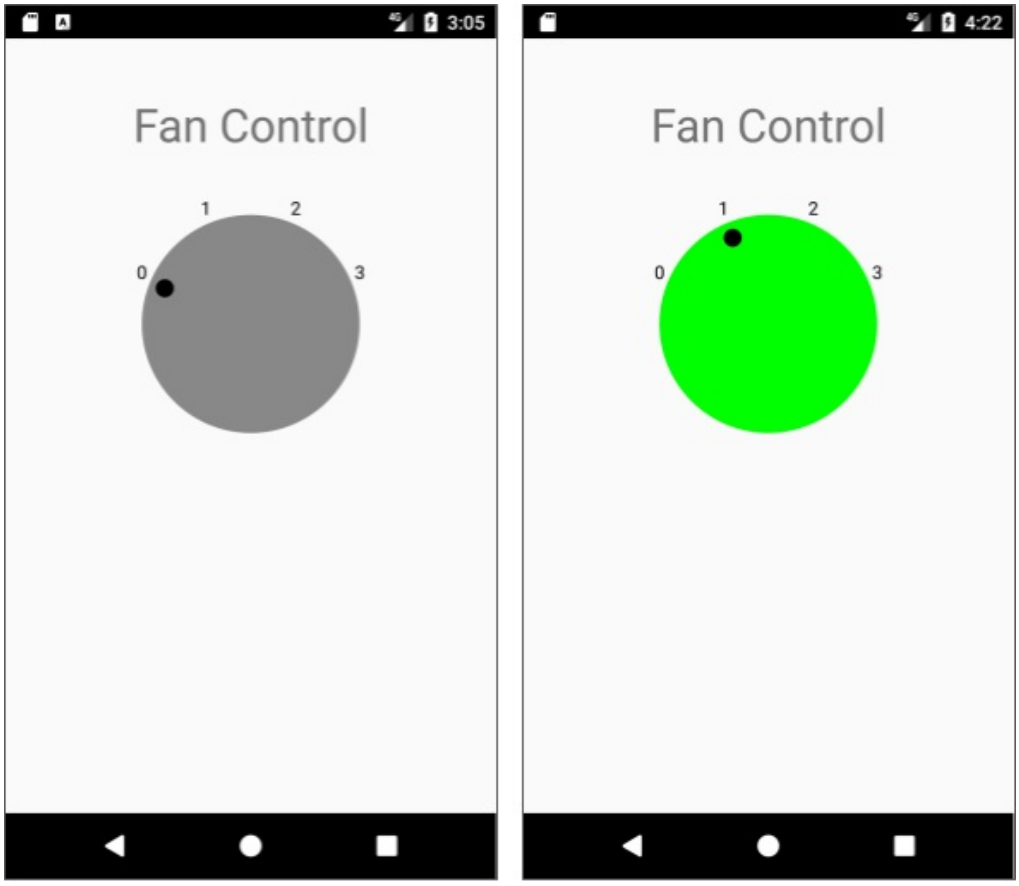
UI building blocks such as `Button` and `TextView` are subclasses that extend the `View` class. To save time and development effort, you can extend one of these `View` subclasses. The custom view inherits the look and behavior of the subclass, and you can override the behavior or aspect of the appearance that you want to change. For example, if you extend

`EditText` to create a custom view, the view acts just like an `EditText` view, but could also be customized to show, for example, an **X** button that clears text from the field.




You can extend any `View` subclass, such as `EditText`, to get a custom view—pick the one closest to what you want to accomplish. You can then use the custom view like any other `View` subclass in one or more layouts as an XML element with attributes. You can also extend the `View` class, and draw a UI element of any size and shape. Your code overrides `View` methods to define its appearance and functionality.

A well-designed custom view encapsulates all of its appearance and functionality within the view. For example, a dial or controller with selections, as shown below, includes all of the logic for making a selection. The user can tap the controller to switch selections.




With custom views, you can:

- Create your own shapes, user interaction, and behavior.
- Make your apps unique.
- Experiment with novel user interactions.

However, the challenge of extending `View` to create your own `View` subclass is the need to override `onDraw()`, which can cause performance issues. (You learn more about `onDraw()` and drawing on a `Canvas` object with a `Paint` object in another lesson.)

# Steps to creating a custom view

Follow these general steps, which are explained in more detail in this chapter:

1. Create a custom view class that extends `View`, or extends a `View` subclass (such as `Button` or `EditText`), with constructors to create an instance of the `View` or subclass.
2. Draw the custom view.
    - If you extend the `View` class, draw the custom view's shape and control its appearance by overriding `View` methods such as `onDraw()` and `onMeasure()` in the new class.
    - If you extend a `View` subclass, override only the behavior or aspects of the appearance that you want to change.

    - In either case, add code to respond to user interaction and, if necessary, redraw the custom view.
3. Use the new class as an XML element in the layout. You can define custom attributes for the view and apply them using getter and setter methods, so that you can use the same custom view with different activities.

# Creating a custom view class

To create a custom view, create a new class that either:

- Extends a `View` subclass, such as `Button`, `TextView`, or `EditText`.
- Extends the `View` class itself.

## Including constructors for the custom view

Include constructors to create an instance of the view. The steps for creating a new class and including its constructors are the same whether the class extends `View` or a `View` subclass.

For example, the following steps add a new subclass that extends `EditText` to provide a clear button (**X**) on the right side for clearing the text.


1. Create a new Java class. In the **Create New Class** dialog, Enter the class name (such as **EditTextWithClear**), and choose a superclass:
    - To extend `EditText`, choose **android.support.v7.widget.AppCompatEditText** as the superclass. `AppCompatEditText` is an `EditText` subclass that supports compatible features on older versions of the Android platform.
    - To extend `View`, choose **android.view.View** as the superclass.
2. Click on the **EditTextWithClear** class definition. In a moment, the red bulb appears, because the class is not complete—it needs constructors. Click the red bulb and choose **Create constructor matching super**. Select all three constructors in the popup menu to add them, and click **OK**:
    - **AppCompatEditText(context:Context)**: Required for creating an instance of a `View` from code.
    - **AppCompatEditText(context:Context, attrs:AttributeSet)**: Required to inflate the

view from an XML layout and apply XML attributes.
- **AppCompatEditText(context:Context, attrs:AttributeSet, defStyleAttr:int)**: Required to apply a default style to all UI elements without having to specify it in each layout file.

All views that extend `View` need at least the first constructor, which is called when the view is created from code. The second constructor is called when the view is inflated from a layout file.

## Adding behavior to the custom view

The clear (**X**) button should appear gray at first, and then turn black to highlight it when the user's finger is touching down on the button.

1. Add to the `drawables` folder the `ic_clear_black_24dp` vector asset twice—one as is for the black version, and one with an opacity of 50% (call `ic_clear_opaque_24dp` ) for the gray version.
2. Define the member variable for the `drawable` ( `Drawable mClearButtonImage` ), and create a helper method that initializes it.

```
private void init() {
    mClearButtonImage = ResourcesCompat.getDrawable(getResources(),
                            R.drawable.ic_clear_opaque_24dp, null);
    // TODO: If the X (clear) button is tapped, clear the text.
    // TODO: If the text changes, show or hide the X (clear) button.
}
```

   The code includes two `TODO` comments for adding touch and text listeners, described later in this chapter.

3. Call the helper method from each constructor, so that you don't have to repeat the same code in each constructor.

```
public EditTextWithClear(Context context) {
        super(context);
        init();
}
public EditTextWithClear(Context context, AttributeSet attrs) {
        super(context, attrs);
        init();
}
public EditTextWithClear(Context context,
                         AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
        init();
}
```

**Tip**: If you want to control access to a custom view class, you can define it as an inner class, inside the activity that wants to use it.

# Drawing the custom view

Once you have created a custom view, you need to be able to draw it.

- If your class is extending a `View` subclass, such as `EditText`, you don't need to draw the entire view—you can override some of its methods to customize the view. This is described in Customizing a View subclass, below.
- If your class is extending `View`, you must override the `onDraw()` method to draw the view. This is described in Drawing an extended View, below.

## Extending and customizing a View subclass

When you extend a `View` subclass such as `EditText`, you are using that subclass to define the view's appearance and attributes. Consequently, you don't have to write code to draw the view. You can override methods of the parent to customize your view.

For example, if you extend an `EditText` view, which is itself a subclass of `TextView`, you can use the `getCompoundDrawables()` and `setCompoundDrawables()` methods inherited from `TextView` to get and set `drawables` for the borders of the view.

The following code shows how you can add a `drawable` (`mClearButtonImage`) to EditTextWithClear and accommodate right-to-left (RTL) languages as well as left-to-right (LTR) languages. (For details on supporting RTL languages, see the chapter on localization.) The code uses `setCompoundDrawablesRelativeWithIntrinsicBounds()`, which sets the bounds of the `drawable` to its intrinsic bounds—the actual dimensions of the `drawable`—and places it in one or more of these locations:

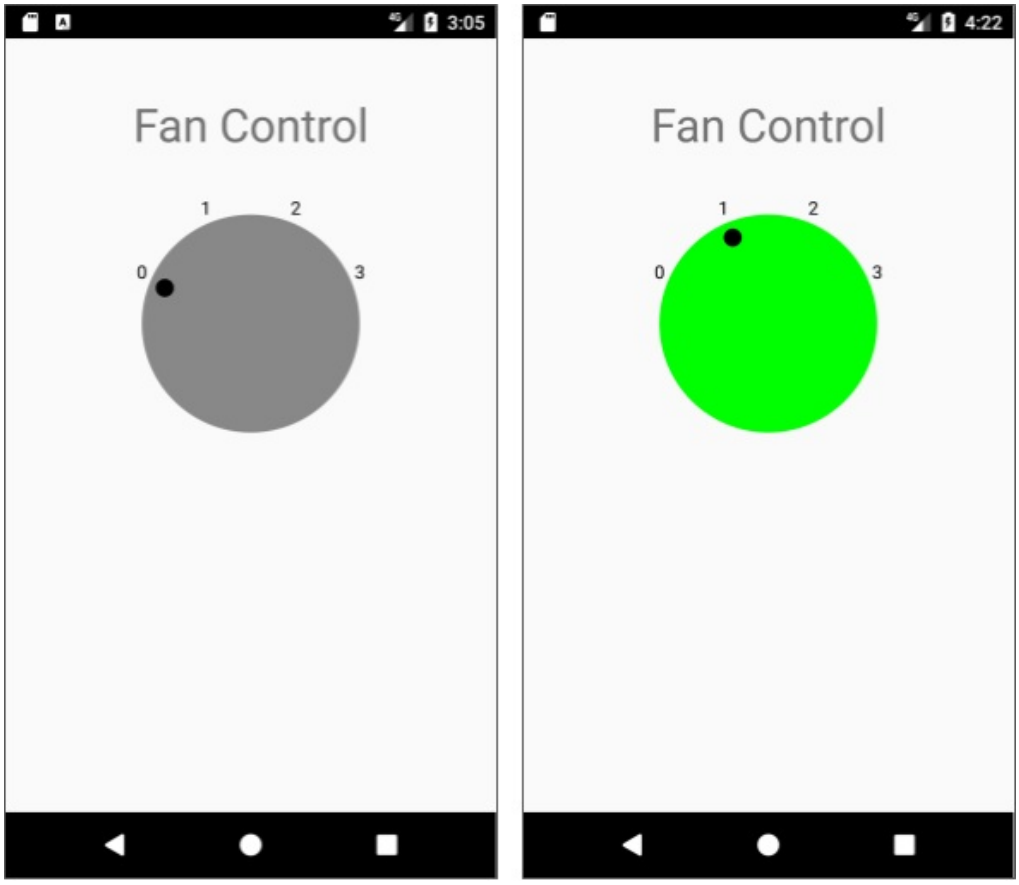
- At the *start* of the text, which is the left side for LTR languages or the right side for RTL languages.
- Above the text.
- At the *end* of the text, which is the right side for LTR languages or the left side for RTL languages.
- Below the text. 


For any of these positions, use `null` for no `drawable`.

```
setCompoundDrawablesRelativeWithIntrinsicBounds
                (null,                      // Start of text.
                        null,               // Above text.
                        mClearButtonImage,  // End of text.
                        null);              // Below text.
// ...
```

## Drawing an extended View

This section shows how to draw the `DialView` custom view that extends `View` to create the shape of a control knob for a fan, with selections for off (0), low (1), medium (2), and high (3).




In order to properly draw a custom view that extends `View`, you need to:

1. Override the `onDraw()` method to draw the custom view, using a `Canvas` object styled by a `Paint` object for the view.
2. Calculate the view's size when it first appears and when it changes by overriding the `onSizeChanged()` method.
3. Use the `invalidate()` method when responding to a user click to invalidate the entire view, thereby forcing a call to `onDraw()` to redraw the view.

In apps that have a deep view hierarchy, you can also override the `onMeasure()` method to accurately define how your custom view fits into the layout. That way, the parent layout can properly align the custom view. The `onMeasure()` method provides a set of `measureSpecs` that you can use to determine your view's height and width.

To understand how Android draws views, see How Android Draws Views. To learn more about overriding `onMeasure()`, see Custom Components.

## Draw with Canvas and Paint

Implement `onDraw()` to draw your custom view, using a `Canvas` as a background. `Canvas` defines shapes that you can draw, while `Paint` defines the color, style, font, and so forth of each shape you draw.

To optimize custom view drawing for better overall app performance, define the `Paint` object for the view in the `init()` helper method you call from the constructors, rather than in `onDraw()`. Don't place allocations in `onDraw()`, because allocations may lead to a garbage collection that would cause a stutter.

**Tip**: See the lesson on performance for a more complete description of optimizing performance with views.

For example, the following `init()` helper method defines the `Paint` object and other attributes for the `DialView` custom view shown in the previous figure:

```
private float mWidth;                   // Custom view width.
private float mHeight;                  // Custom view height.
private Paint mTextPaint;               // For text in the view.
private Paint mDialPaint;               // For dial circle in the view.
private float mRadius;                  // Radius of the circle.

private void init() {
    mTextPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    mTextPaint.setColor(Color.BLACK);
    mTextPaint.setStyle(Paint.Style.FILL_AND_STROKE);
    mTextPaint.setTextAlign(Paint.Align.CENTER);
    mTextPaint.setTextSize(40f);
    mDialPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    mDialPaint.setColor(Color.GRAY);
}
```

The following code snippet shows how you override `onDraw()` to create a `Canvas` object, and draw on the canvas. It uses the `Canvas` methods `drawCircle()` and `drawText()`.

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    // Draw the dial.
    canvas.drawCircle(mWidth / 2, mHeight / 2, mRadius, mDialPaint);

    // Draw the text labels.
    // ...
    canvas.drawText(Integer.toString(i), x, y, mTextPaint);

    // Draw the indicator mark.
    // ...
    canvas.drawCircle(x, y, 20, mTextPaint);
}
```

The `Canvas` and `Paint` classes offer a number of useful drawing shortcuts:

- Draw text using `drawText()`. Specify the typeface by calling `setTypeface()`, and the text color by calling `setColor()`.
- Draw primitive shapes using `drawRect()`, `drawOval()`, and `drawArc()`. Change whether the shapes are filled, outlined, or both by calling `setStyle()`.
- Draw bitmaps using `drawBitmap()`.

**Tip**: To learn more about drawing with on a `Canvas` with `Paint`, see the lesson on using the `Canvas`.

## Calculate the size in onSizeChanged()

When your custom view first appears, the `View` method `onSizeChanged()` is called, and again if the size of your view changes for any reason. Override `onSizeChanged()` in order to calculate positions, dimensions, and any other values related to your view's size, instead of recalculating them every time you draw:

```
@Override
protected void onSizeChanged(int w, int h, int oldw, int oldh) {
        // Calculate the radius from the width and height.
        mWidth = w;
        mHeight = h;
        mRadius = (float) (Math.min(mWidth, mHeight) / 2 * 0.8);
}
```

**Tip**: If you use padding values, include them when you calculate your view's size.

## Forcing a redraw after user interaction

To handle a custom view's behavior when a user interacts with it, add an event listener—an interface in the `View` class that contains a single callback method, such as `onClick()` from `View.OnClickListener`. When a change occurs to the custom view, use the `invalidate()` method of `View` to invalidate the entire view, thereby forcing a call to `onDraw()` to redraw the view.

For example, the following code snippet uses an `OnClickListener()` to perform an action when the user taps the view. Each tap moves the selection indicator to the next position: 0-1-2-3 and back to 0. Also, if the selection is 1 or higher, the code changes the background from gray to green, indicating that the fan power is on:

```
setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        // Rotate selection to the next valid choice.
        mActiveSelection = (mActiveSelection + 1) % SELECTION_COUNT;
        // Set dial background color to green if selection is >= 1.
        if (mActiveSelection >= 1) {
            mDialPaint.setColor(Color.GREEN);
        } else {
            mDialPaint.setColor(Color.GRAY);
        }
        // Redraw the view.
        invalidate();
    }
});
```

**Tip**: To learn more about event listeners, see Input Events.

# Using the custom view in a layout

To add a custom view to an app's UI, specify it as an element in the activity's XML layout. Control its appearance and behavior with XML attributes, as you would for any other UI element.

## Using standard attributes

A custom view inherits all the attributes of the class it extends. For example, if you are extending an `EditText` view to create a custom view, the attributes you would use for an `EditText` are the same as the ones you use for the custom view.

While developing, you can use the view you are extending as a placeholder for the custom view, and use standard attributes with the placeholder. After developing the custom view, replace the placeholder with the custom view.

For example, you might use an `ImageView` as a placeholder in the layout until you have designed the `DialView` custom view. The attributes you use for the `ImageView` are the same as the ones you need for the custom view. All you need to change is the element's tag from `ImageView` to `com.example.customfancontroller.DialView` in order to replace it with the custom view:

```
<com.example.customfancontroller.DialView
        android:id="@+id/dialView"
        android:layout_width="@dimen/dial_width"
        android:layout_height="@dimen/dial_height"
        <!-- More attributes ...  -->
/>
```

## Using custom attributes

In addition to the attributes available from the parent, you can also define custom attributes for a custom view. You define an attribute and its type in XML, set the attribute value, and retrieve the attribute value in the view's constructor.

To use a custom attribute:

1. Define the name and type of your custom attributes in a `<declare-styleable>` resource element in a `res/values/attrs.xml` file. For example, you could define a `boolean` attribute for the custom view that changes the custom view's appearance at night.

   ```
   <resources>
       <declare-styleable name="DialView">
           <attr name="nightAppearance" format="boolean" />
       </declare-styleable>
   </resources>
   ```

   This declares a `styleable` called `DialView` (it is typical to use the same name as the custom view class), and specifies the `nightAppearance` attribute to be a `boolean` (true or false).

2. Specify values for the attributes in your XML layout. For example, the `nightAppearance` attribute for `DialView` can be set to `true` or `false`:

   ```
   <com.example.customfancontroller.DialView
       android:id="@+id/dialView"
       app:nightAppearance="false"
       <!-- More attributes ...  -->
   />
   ```

3. Retrieve the attribute value in the view's constructor. When a view is created from an XML layout, all of the attributes are read from a resource `Bundle` and passed into the view's constructor as an `AttributeSet` ( `attrs` ). In the custom view class (in this example, `DialView.java` ), look for the constructor method that uses the `AttributeSet` `attrs` . To retrieve the attribute values, add code to this method to pass `attrs` to `obtainStyledAttributes()` :

```
public DialView(Context context, AttributeSet attrs) {
    super(context, attrs);
    // Collect the attributes into a typed array.
    TypedArray typedArray = context.obtainStyledAttributes
                            (attrs, R.styleable.DialView, 0, 0);
```

This `obtainStyledAttributes()` method passes back a `TypedArray` of attribute values, including the `nightAppearance` custom attribute, which is a `boolean`.

4. Adding to the above constructor, assign the `boolean` returned in the typed array to `mNightAppearance`, and recycle the typed array:

```
    // Use mNightAppearance to represent the attribute.
    mNightAppearance = typedArray.getBoolean
                    (R.styleable.DialView_nightAppearance, false);
    // Recycle the array.
    typedArray.recycle();
```

The `getBoolean()` method takes two arguments: an index to the attribute (`R.styleable.DialView_nightAppearance`), and the default value of the attribute (`false`) if the value is not set in the layout file. The method returns the `boolean` value of the attribute, or the default value if the attribute was not defined. `TypedArray` objects are a shared resource and must be recycled by the `recycle()` method so that they can be reused by a later caller. The following shows the entire constructor:

```
public DialView(Context context, AttributeSet attrs,
                                        int defStyleAttr) {
 super(context, attrs, defStyleAttr);
 // Collect the attributes into a typed array.
 TypedArray typedArray = context.obtainStyledAttributes(attrs,
                             R.styleable.DialView, 0, 0);
 // Use mNightAppearance to represent the attribute.
 mNightAppearance =
     typedArray.getBoolean(R.styleable.DialView_nightAppearance,
         false);
 // Recycle the array.
 typedArray.recycle();
 init();
}
```

5. Apply the retrieved attribute value to your custom view. In the above example, `mNightAppearance` is now `true` or `false` depending on the value of the `nightAppearance` custom attribute, so you can use `mNightAppearance` to set the `Paint` values for the text labels and the dial:

```
if (mNightAppearance) {
    mTextPaint.setColor(Color.RED);
    mDialPaint.setColor(Color.YELLOW);
} else {
    mTextPaint.setColor(Color.BLACK);
    mDialPaint.setColor(Color.GRAY);
}
```

# Using property accessors and modifiers

Attributes are a powerful way of controlling the behavior and appearance of views, but they can be read only when the view is initialized. To provide dynamic behavior, expose a property accessor ("getter") and modifier ("setter") in your custom view class for each attribute.

For example, you can design the custom view to use a different set of colors at night depending on the position of a switch ( `nightSwitch` ).

In the layout, you use the `nightAppearance` custom attribute for the custom view, and set its default to `false` . You would then implement a new getter and setter method in the `DialView` class to get and set the value of the `attribute` , and write code to decide which color set to use:

- Use night colors if `nightAppearance` is `true` (in position 0, yellow for the dial and red for the text).
- Use normal colors if `nightAppearance` is `false` (in position 0, gray for the dial and black for the text).

(When the user taps the controller to turn the fan on, the dial should turn green in either case.)

In the code snippet, the "getter" `getNightAppearance()` method returns the `boolean` `mNightAppearance` representing the current state of the custom view ( `true` if the night switch is on, `false` if not):

```
public boolean getNightAppearance() {
    // ...
    return mNightAppearance;
}
```

The "setter" `setNightColors()` method sets the colors based on the `nightSwitch` argument ( `true` or `false` ) passed to the method:

```
public void setNightColors(boolean nightSwitch) {
    // ...
    if (nightSwitch) {
        mTextPaint.setColor(Color.RED);
        mDialPaint.setColor(Color.YELLOW);
    } else {
        mTextPaint.setColor(Color.BLACK);
        mDialPaint.setColor(Color.GRAY);
    }
    invalidate();
    // ...
}
```

You must `invalidate()` the view after any change to its properties that might change its appearance so the system knows that the view needs to be redrawn. The system in turn calls `onDraw()` to redraw the view.

Likewise, if a property change affects the size or shape of the view, you need to request a new layout with `requestLayout()`. Note, however, that any time a view calls `requestLayout()`, the system traverses the entire view hierarchy to find out how big each view needs to be, which can affect overall app performance.

**Tip**: Always make `invalidate()` and `requestLayout()` calls from the UI thread. See the lesson on performance for a more complete description of optimizing performance with views.

The following code snippet shows how you would use the "getter" and "setter" methods in an `Activity`:

```
DialView myView;
// ...
    if (myView.getNightAppearance()) {
        myView.setNightColors(true);
    } else {
        myView.setNightColors(false);
    }
```

By exposing property accessors and modifiers in your custom view class for each attribute, you can control the values for those attributes in the `Activity` code. This technique is extremely valuable for controlling the appearance of a custom view based on user interaction.

# Related practicals

The related practical documentation:

- Creating a custom view from a View subclass
- Creating a custom view from scratch

# Learn more

Android developer documentation:

- Creating Custom Views
- Custom Components
- How Android Draws Views
- View
- Input Events
- onDraw()
- Canvas
- drawCircle()
- drawText()
- Paint

Video: Quick Intro to Creating a Custom View in Android

# 11.1: The Canvas class

**Contents:**



## Drawing in Android

In Android, you have several techniques available for implementing custom 2D graphics and animations. Which techniques you choose depends on the type of graphics you want to display, as well as the needs of your app.

- *Drawables*

  A `Drawable` is a graphic that can be drawn to the screen. You create `Drawable` objects for things such as textured buttons or frame-by-frame animations and display them into a standard or custom view. The drawing of your graphics is handled by the system's view hierarchy drawing process. Using a drawable is your best choice when you want to draw simple graphics that do not need to change dynamically and are not part of a performance-intensive game. For more informations, see `Drawables` and the chapter about drawables in Android Developer Fundamentals.

- *Canvas*

You can do your own custom 2D drawing using the drawing methods of the `Canvas` class. Draw to a `Canvas` when your app needs to regularly re-draw itself. In this chapter, you learn how to create and draw on a canvas.

- *SurfaceView*

An instance of `SurfaceView` is a drawing surface that is part of your view hierarchy and managed and rendered by the system along with the view hierarchy. One of the purposes of this class is to provide a surface in which a secondary thread can render into the screen.

The challenge is that you have to correctly manage your thread to only draw to the surface when it exists. The `SurfaceView` chapter includes an example of using a `SurfaceView`.

In addition, the following techniques are available but not covered in this course:

- Hardware Acceleration. Beginning in Android 3.0, you can hardware-accelerate most of the drawing done by the `Canvas` APIs to further increase their performance.
- `OpenGL`. Android supports `OpenGL ES 1.0` and `2.0`, with Android framework APIs as well as natively with the Native Development Kit (NDK). Using the framework APIs is desireable when you want to add a few graphical enhancements to your application that are not supported with the `Canvas` APIs, or if you desire platform independence and don't demand high performance.

# What is a Canvas object?

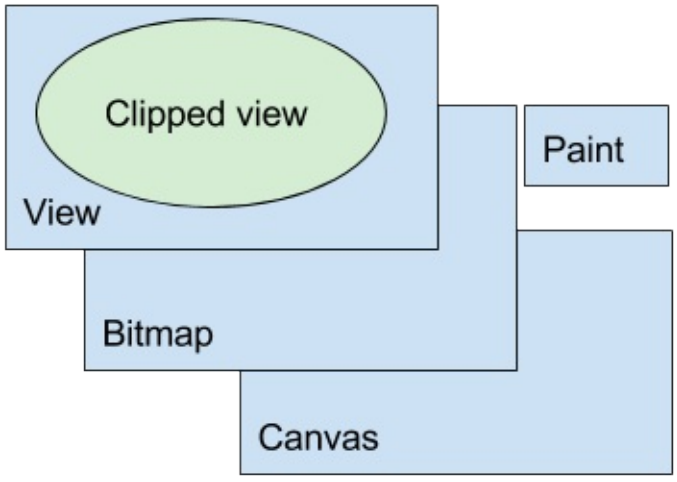
When you want to draw shapes or text into a view on Android, you can do so via a `Canvas` object. Very simplified:

- A `Canvas` is a 2D drawing surface that provides methods for drawing to a bitmap.
- The bitmap, an instance of the `Bitmap` class is associated with a `View` instance that displays it.
- A `Paint` object holds the style and color information about how to draw geometries (such as line, rectangle, oval, and paths), text, and bitmap.

`Canvas` defines shapes that you can draw on the screen, while `Paint` defines the color, style, font, and so forth, of each shape you draw.

The `Canvas` class also provides methods for clipping. Clipping is the action of defining geometrically what portion of the canvas is shown to the user in the view.

The figure below shows all the pieces required to draw.



The types of operations you can perform on a canvas include:

- Fill the whole canvas with color.
- Draw shapes, such as rectangles, arcs, paths styled as defined in a `Paint` object.
- Draw text styled by the properties in a `Paint` object.
- Save the current `Canvas` state and restore a previous state.
- Apply transformations, such as translation, scaling, or custom transformations.
- Clip, that is, apply a shape or path to the `Canvas` that defines its visible portions.

# Steps for creating and drawing on a Canvas object

After you have a project and activity, you need the following to work with a `Canvas` object.

1. Create a custom `View` class. You can extend any view that has the features you need. You do not need a custom `View` to draw, as discussed in the Simple Canvas practical. Typically you use a custom `View` so that you can draw by overriding the `onDraw()` method of the `View`.

   public class MyCanvasView extends View {...}

2. In `onCreate()` of the `MainActivity`, set the content `View` of the activity to be an instance of your custom `View`.

   myCanvasView = new MyCanvasView(this); setContentView(myCanvasView);

3. In the constructor of the custom `View`, create a `Paint` object and set initial `Paint` properties. Initialize member variables and get a reference to the context. You cannot create your `Canvas` here because the `View` has not been inflated yet and thus has no size.

```
MyCanvasView(Context context) {
   this(context, null);
}

public MyCanvasView(Context context, AttributeSet attributeSet) {
   super(context);

   int backgroundColor;

   mDrawColor = ResourcesCompat.getColor(getResources(),
                  R.color.opaque_orange, null);
   backgroundColor = ResourcesCompat.getColor(getResources(),
           R.color.opaque_yellow, null);

   // Holds the path we are currently drawing.
   mPath = new Path();
   // Set up the paint with which to draw.
   mPaint = new Paint();
   mPaint.setColor(backgroundColor);
   // Smoothes out edges of what is drawn without affecting shape.
   mPaint.setAntiAlias(true);
   // Dithering affects how colors with higher-precision than the device
   // are down-sampled.
   mPaint.setDither(true);
   mPaint.setStyle(Paint.Style.STROKE); // default: FILL
   mPaint.setStrokeJoin(Paint.Join.ROUND); // default: MITER
   mPaint.setStrokeCap(Paint.Cap.ROUND); // default: BUTT
   mPaint.setStrokeWidth(12); // default: Hairline-width (really thin)
}
```

4. In the custom `View`, override `onSizeChanged()` and create a `Bitmap`, then create a `Canvas` with the `Bitmap`. The `onSizeChanged()` method is called when your `View` is first assigned a size, and again if the size of your `View` changes for any reason. Calculate positions, dimensions, and any other values related to your `View`'s size in `onSizeChanged()`.

```
@Override
protected void onSizeChanged(int width, int height,
                             int oldWidth, int oldHeight) {
   super.onSizeChanged(width, height, oldWidth, oldHeight);
   // Create bitmap, create canvas with bitmap, fill canvas with color.
   mBitmap = Bitmap.createBitmap(width, height, Bitmap.Config.ARGB_8888);
   mCanvas = new Canvas(mBitmap);
   mCanvas.drawColor(mDrawColor);
}
```

5. In the custom `View`, override the `onDraw()` method. What happens in the `onDraw()` method can be as short as drawing a `Path` that was calculated in another method, or it can include drawing, transformations, and clipping, which are discussed below.

```
@Override
protected void onDraw(Canvas canvas) {
   super.onDraw(canvas);
   // First, draw the bitmap as created.
   canvas.drawBitmap(mBitmap, 0, 0, mPaint);
   // Then draw the path on top, styled by mPaint.
   canvas.drawPath(mPath, mPaint);
}
```

6. If any drawing is to happen in response to user motion, override `onTouchEvent()`. In this example, when the user drags their finger, and then lifts it off the screen, the x,y coordinates are handed off for some drawing action. You must call `invalidate()` every time the screen needs to be redrawn.

```
@Override
public boolean onTouchEvent(MotionEvent event) {
   float x = event.getX();
   float y = event.getY();

   switch (event.getAction()) {
       case MotionEvent.ACTION_DOWN:
           touchStart(x, y);
           // No need to invalidate because we are not drawing anything.
           break;
       case MotionEvent.ACTION_MOVE:
           touchMove(x, y);
           invalidate();
           break;
       case MotionEvent.ACTION_UP:
           touchUp();
           invalidate();
           break;
       default:
           // do nothing
   }
   return true;
}
```

**Note:**This work all happens on the UI thread, which is why you should not use this technique for lengthy operations. For more complex operations, manage a `SurfaceView` and perform draws to the `Canvas` as fast as your thread is capable. See the `SurfaceView` chapter.

# Drawing shapes and text

You can draw primitive shapes using predefined methods, and use a `Path` to create any shape you need.

- Draw primitive shapes using `drawRect()`, `drawOval()`, and `drawArc()`. Change whether the shapes are filled, outlined, or both by calling `setStyle()`.
- Draw more complex shapes using the `Path` class. Define a shape by adding lines and curves to a `Path` object, then draw the shape using `drawPath()`. As with primitive shapes, paths can be outlined, filled, or both, depending on the `setStyle` ().
- Draw text using `drawText()`. Specify the typeface by calling `setTypeface()`, and the text color by calling setColor().
- Define gradient fills by creating `LinearGradient` objects. Call `setShader()` to use your `LinearGradient` on filled shapes.
- Draw bitmaps using `drawBitmap()`.

See the documentation for details of each method, and the practicals for examples on how these methods are used.

# Transformations

Transformations are operations that change the canvas itself:

- Use `translate()` to move the origin of the canvas. For example, when you are drawing the same shape in multiple places, instead of recalculating the shape, you can move the origin of the canvas and draw the same shape again. See the Applying clipping to a canvas practical for an example.

  ```
  canvas.translate(dx, dy);
  ```

- Use rotate to turn the `Canvas` by a number of degrees.

  ```
  canvas.rotate(180);
  ```

- Skew the `Canvas` by calling the `skew()` method. You can achieve interesting effects with text using skewing.

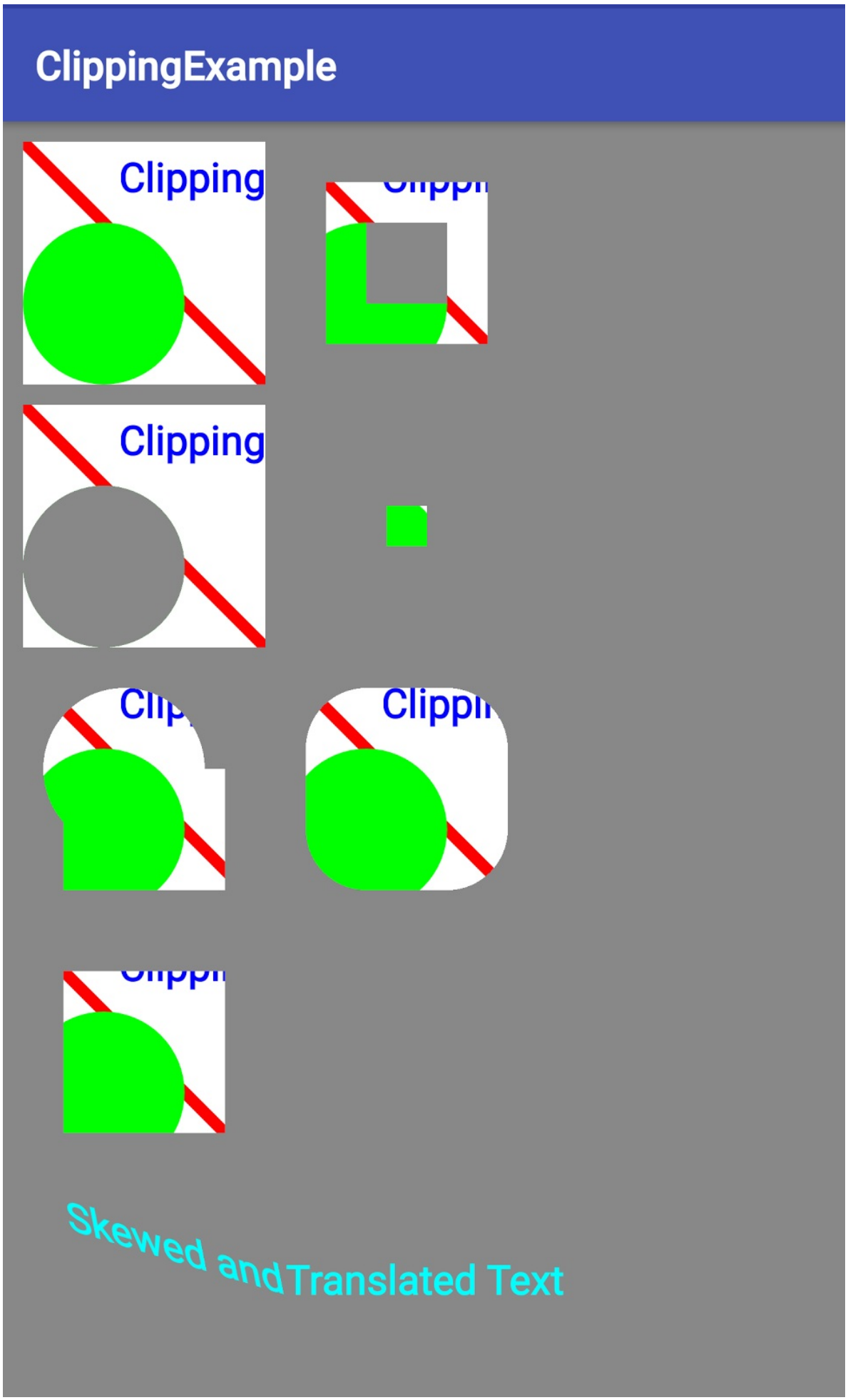
  ```
  mPaint.setTextSize(120);
  canvas.translate(100, 1800);
  canvas.skew(0.2f, 0.3f);
  canvas.drawText("Clipping", 400, 60, mPaint);
  ```

  See the `Canvas` documentation for details on these and additional methods.

# Clipping

`Clipping` is a method for defining regions of an image, canvas, or bitmap that are selectively drawn or not drawn onto the screen. One purpose of clipping is to reduce *overdraw* . When you reduce overdraw, you minimize the number of times a pixel or region of the display is drawn, in order to maximize drawing performance. You can also use clipping to create interesting effects in user interface design and animation.

The following screenshot from the Applying clipping to a canvas practical shows how you can clip and combine clipping regions.


ClippingExample
Clipping
Clipping
Skewed and Translated Text

`Canvas` provides methods for clipping, and you can also clip by defining a custom path.

- Use `canvas.clipRect()` to set a rectangular clipping region.
- Define a `Path` and apply it with `canvas.clipPath()` for a custom clipping region. For example, to create a circular clipping region.
    - `CCW` stands for counterclockwise and indicates in which direction the circle should be drawn.
    - `Region.Op.DIFFERENCE` specifies how the clipping region should be applied to the canvas. See the `Region.Op` documentation for other operators.

Here is the code:

```
mPath.addCircle(mCircleRadius, mClipRectBottom-mCircleRadius, mCircleRadius, Path.Dire
ction.CCW);
canvas.clipPath(mPath, Region.Op.DIFFERENCE);
```

For a list of all clipping methods and supported operators for different versions of Android, see the `Canvas` documentation.

## quickReject()

The `quickReject()` method allows you to check whether a specified rectangle or path would lie completely outside the currently visible regions, after all transformations have been applied.

The `quickReject()` method is incredibly useful when you are constructing more complex drawings and need to do so as fast as possible. With `quickReject()`, you can decide efficiently which objects you do not have to draw at all, and there is no need to write your own intersection logic.

- The method returns true if the rectangle or path would not be visible at all. For partial overlaps, you still have to do your own checking.
- The `EdgeType` is either `AA` (Antialiased: Treat edges by rounding-out, because they may be antialiased) or `BW` (Black-White: Treat edges by just rounding to nearest pixel boundary) for just rounding to the nearest pixel.

| | |
|---|---|
| `boolean` | quickReject `(float left, float top, float right, float bottom,` Canvas.EdgeType `type)` |
| `boolean` | quickReject `(` RectF `rect,` Canvas.EdgeType `type)` |
| `boolean` | quickReject `(` Path `path,` Canvas.EdgeType `type)` |

# Saving and restoring a canvas

The activity context maintains a stack of drawing states. Each state includes the currently applied transformations and clipping regions. You can't remove clipping regions. Undoing a transformation by reversing it is error-prone, as well as chaining too many transformations relative to each other. Translation is straightforward to reverse, but if you also stretch, rotate, or custom deform, it gets complex quickly. Instead, save the state of the canvas, apply your transformations, draw, and then restore the previous state.

```
canvas.save();
mPaint.setTextSize(120);
canvas.translate(100, 1800);
canvas.skew(0.2f, 0.3f);
canvas.drawText("Skewing", 400, 60, mPaint);
canvas.restore();

canvas.save();
mPaint.setColor(Color.CYAN);
canvas.translate(600, 1800);
canvas.drawText("Save/Restore", 400, 60, mPaint);
canvas.restore();
```

For a complete example, see the Clipping Canvas practical.

# Related practicals

The related exercises and practical documentation is in Advanced Android: Practicals.

- Create a simple Canvas
- Draw on a Canvas
- Apply clipping to a Canvas

# Learn more

Android developer docs:

- `Canvas` class
- `Bitmap` class
- `View` class
- `Paint` class
- `Bitmap.config` configurations
- `Region.Op` operators

- `Path` class
- `android.graphics` graphics tools
- `Bitmap.Config` `Canvas` configurations
- Canvas and Drawables
- Understanding save() and restore() for the Canvas context

Also see the Graphics Architecture series of articles for an in-depth explanation of how the Android framework draws to the screen.

# 11.2: The SurfaceView class

**Contents:**



When you create a custom view and override its `onDraw()` method, all drawing happens on the UI thread. Drawing on the UI thread puts an upper limit on how long or complex your drawing operations can be, because your app has to complete all its work for every screen refresh. Each screen refresh usually happens in 16 milliseconds or less, for a typical 60 frames-per-second mobile display. (See the lesson on Performance.)

One solution is to move some of the drawing work to a different thread. One way of doing that is with a `SurfaceView`.

A `SurfaceView` is a view in your app's view hierarchy that has its own separate `Surface`, as shown in the diagram below. This makes it possible to draw to the `Surface` from a separate thread.

In the context of the Android framework, *`Surface`* refers to a lower-level drawing surface whose contents are eventually displayed on the user's screen. To create what the user sees on the screen, the Android system processes your app's layouts and drawing instructions to compose one or more `Surfaces`, and renders them to the screen. All the views in your view hierarchy are rendered onto one `Surface` in the UI thread.

To draw onto a `Surface`:

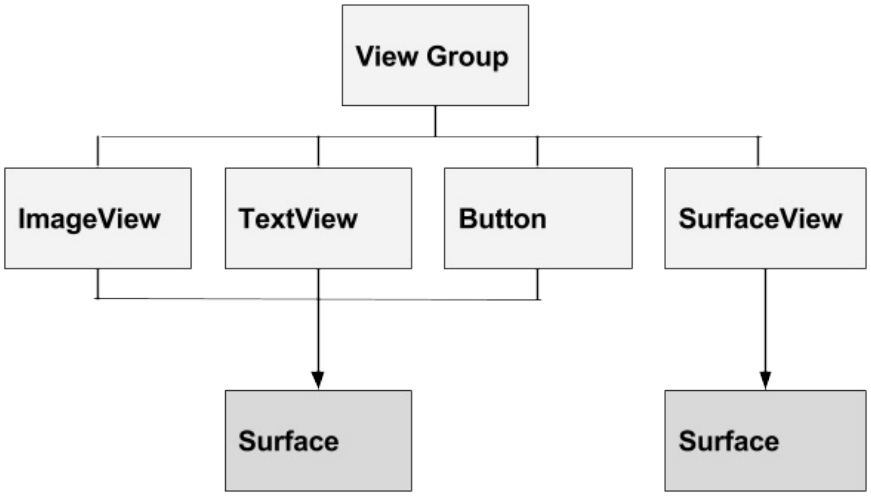
1. Start a thread.
2. Lock the `SurfaceView`'s canvas.
3. Do your drawing.
4. Post it to the `Surface`. You learn more about this below.

The Android system eventually combines all the `Surfaces` and renders them to the screen, with the `SurfaceView's` `Surface` behind the app's `Surface`.

To fully understand how `SurfaceView` works requires learning Android Graphics Architecture, which is beyond the scope of this course. The Android developer documentation provides an excellent and much more detailed technical overview in the Graphics Architecture documentation.

**Important:** `SurfaceView` is currently not hardware-accelerated. Depending on your app, you

may need to consider performance trade-offs. Starting with Android O, you can hardware-accelerate your SurfaceView, which eliminates this specific performance concern.
**Note:** Other classes are available for when you need more control of your drawing surfaces and operations. Their discussion is beyond the scope of this course. See Graphics architecture for a thorough introduction.




# Working with the SurfaceView class

To create a `SurfaceView` and draw to it in a separate thread, do the following. See the Creating a SurfaceView object practical for a detailed example.

1. Create a custom view that extends `SurfaceView` and implements `Runnable`. The `Runnable` class includes a `run()` method.Implementing the `run()` method allows a class to execute on a separate thread. Define you class as follows:

   ```
   public class GameView extends SurfaceView implements Runnable{}
   ```

2. Use the custom view's constructor to initialize your member variables and obtain a reference to the `SurfaceView` 's `SurfaceHolder`. Through the holder you can control the `Surface` size and format, edit the pixels in the `Surface`, and monitor changes to the `Surface`. Most importantly, the holder is persistent even when a `Surface` is not available (see next step).

   ```
   mSurfaceHolder = getHolder();
   ```

3. A `Surface` is only available while the `SurfaceView` 's window is visible. If your `SurfaceView` is not always visible, implement the `surfaceCreated(SurfaceHolder)` and `surfaceDestroyed(SurfaceHolder)` callback methods. Use these methods to discover when the `Surface` is created and destroyed, as the window is shown and hidden.

Now implement the `run()` method to do the following:

1. Always check whether a valid `Surface` is available.

```
if (mSurfaceHolder.getSurface().isValid()) {
```

2. Lock the canvas. If there is more than one thread drawing on this `Surface`, you must put this into a `try/catch` block to handle an `exception` when you can't acquire the lock. See the SurfaceView documentation for details and limitations.

```
mCanvas = mSurfaceHolder.lockCanvas();

// OR for Android O and later, you can also use lockHardwareCanvas()
mCanvas = mSurfaceHolder.lockHardwareCanvas();
```

3. Draw on the canvas.
4. Unlock the canvas and post its contents to the `Surface`. Minimize the amount of time the canvas is locked, as there is a performance penalty associated with this.

```
mSurfaceHolder.unlockCanvasAndPost(mCanvas);
```

Implement `pause()` and `resume()` to stop and start a new thread. For example:

```
public void pause() {
   mRunning = false;
   try {
       // Stop the thread (rejoin the main thread)
       mGameThread.join();
   } catch (InterruptedException e) {
   }
}

public void resume() {
   mRunning = true;
   mGameThread = new Thread(this);
   mGameThread.start();
}
```

Handle user-touches or any other data input that affects drawing.

Be aware of these threading semantics:

- Ensure that the drawing thread only touches the underlying `Surface` while it is valid; that is, between `SurfaceHolder.Callback.surfaceCreated()` and `SurfaceHolder.Callback.surfaceDestroyed()`.
- All `SurfaceView` and `SurfaceHolder.Callback` methods will be called from the thread

running the `SurfaceView`'s window (typically the main thread of the application). These methods need to correctly synchronize with any state that is also touched by the drawing thread.

**IMPORTANT:** This concept chapter provides only enough information for you to get started with `SurfaceView`. See the documentation and many publicly available examples if you are interested in learning more about lower-level drawing, game loops, and game development.

# Related practical

The related practical documentation is in Creating a SurfaceView object.

# Learn more

Android developer docs

- `SurfaceView` class
- `SurfaceHolder` class
- `ClippingBasic` code sample
- Graphics architecture article
- Sleeping your app is a bad idea
- `Choreographer` class
- Hardware acceleration
- Grafika collection of Android graphics tips

# 12.1: Animations

**Contents:**



## What is animation?

Animation is the technique for creating the illusion of a moving object by showing a series of discrete images that change over time, such as:

- A flip book, which has a different image on each page. By flipping through the pages fast enough, your eyes perceive it as motion.
- Claymation, which is a type of stop-motion animation.
- User interface animations, such as flinging a list item to move it off the screen.
- .. and millions of mobile games with character, environment, and UI animations.

Each image in an animated sequence is called a *frame* . The speed at which those images appear on the screen is called the *frame rate* . Ideally, your frame rate closely matches the screen's refresh rate.

- If the frame rate is slower than the refresh rate, animation may stutter.
- If the frame rate is faster than the refresh rate, the app is wasting resources.
- If the frame rate is the same as the screen refresh rate, animations are smooth and no resources are wasted.

Fortunately, the Android system manages frame rate for you, and in most situations, you do not need to manage the frame rate of your animations. If you want to go deep, see the Graphics Architecture series of articles for an in-depth explanation of how the Android framework draws to the screen.

## Types of animation for Android

The Android framework provides the following animation systems:

`View` animation

`View` animation, as the name implies, is a system for animating views. It is an older system and limited to `View` objects. It is relatively easy to set up and offers enough capabilities to meet many applications' needs.

`Property` animation

Introduced in Android 3.0 (API level 11), the property animation system lets you animate properties of any object, including ones that are not rendered to the screen. The system is extensible and lets you animate properties of custom types as well. Prefer property animation over `View` animation. See also the Property Animation practical.

`Drawable` animation

`Drawable` animation involves displaying `Drawable` resources one after another, like a roll of film. This method of animation is useful if you want to animate things that are easier to represent with `Drawable` resources, such as a progression of bitmaps.

Physics-based animation

Physics-based animation relies on the laws of physics to manifest a high degree of realism in animation. Physics-based animation uses the fundamentals of physics to build animations. For example, an animation is driven by force, and the animation comes to rest when the force reaches equilibrium.

# View animation

The `View` animation system provides the limited capability to only animate `View` objects. For example, you can scale or rotate a `View` object.

The view animation system only modifies where the view is drawn, and not the actual `View` object itself. For instance, if you animate a button to move across the screen, the button draws correctly, but the location where you can click the button does not change, and you have to implement your own logic to handle this.

The view animation system, however, takes less time to set up and requires less code to write than property animations. If view animation accomplishes everything that you need to do, there may be no need to use the property animation system.

# Property animation

The property animation system is a robust framework that allows you to animate almost anything. A property animation changes a property's value over a specified length of time. For example, you can animate a circle to grow bigger by increasing its radius over time. You can define an animation to change any object property (a field in an object) over time, regardless of whether the change draws to the screen or not.

With the property animation system, at a high level, you assign animators to the properties that you want to animate, such as color, position, or size. You also define aspects of the animation, such as interpolation. For example, you could create an animator for the radius of a circle you want to grow.

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

- *Duration:* You can specify how long an animation runs. The default length is 300 milliseconds.
- *Time interpolation:* You can specify how the values for the property are calculated as a function of the animation's current elapsed time. For example, you can move an object across the screen linearly, by the same amount for every time interval. Or you can increase or decrease the amount for every interval. You can choose from system-provided interpolators or create your own. See below for details.
- *Repeat count and behavior:* You can specify whether or not to have an animation repeat when it reaches the end of a duration, and how many times to repeat the animation. You can also specify whether you want the animation to play back in reverse. Setting it to reverse plays the animation forwards then backwards repeatedly, until the number of repeats is reached.
- *Animator sets:* You can group animations into logical sets that play together or sequentially or after specified delays. For example, you could coordinate several bouncing balls.
- *Frame-refresh delay:* You can specify how often to refresh frames of your animation. The default is set to refresh every 10 ms, but the speed in which your application can refresh frames ultimately depends on how busy the system is overall and how fast the system can service the underlying timer.

## Interpolators

Material Design encourages vivid animations that catch the user's attention while behaving in a more natural way. In nature, most motion is not linear but changes over time. For example, friction will slow down a rolling ball, or a vehicle gains speed as it moves forward.

A time interpolator defines how specific values in an animation change over time. For example, you can specify animations to happen linearly across the whole animation, meaning the animation moves evenly the entire time. You can also specify animations to use

nonlinear time, for example, accelerating at the beginning and decelerating at the end of the animation.

The following interpolators are predefined in the Android API and ready to use:

- `LinearInterpolator` : `An` interpolator whose rate of change is constant. The value changes at a constant rate over time. For example, you can move a box across the screen, taking a total time of three seconds, and every frame moves the box by the same amount of distance.
- `AccelerateInterpolator` : An interpolator whose rate of change starts out slowly and then accelerates. The value changes at an increasing or decreasing rate over time. For example, you can move a box across the screen, taking a total time of three seconds, and every frame moves the box by an increasing or decreasing amount of distance, creating an impression accelerating or decelerating.
- `AccelerateDecelerateInterpolator` : An interpolator whose rate of change starts and ends slowly but accelerates through the middle. For example, you can start a box moving at one end of the screen, let it accelerate, and come to slow stop at the other side of the screen.
- `AnticipateInterpolator` : An interpolator whose change starts backward then flings forward. Visualize this by thinking of snapping a rubber band.
- `OvershootInterpolator` : An interpolator whose change flings forward and overshoots the last value then comes back.
- `AnticipateOvershootInterpolator` : An interpolator whose change starts backward, flings forward and overshoots the target value, then finally goes back to the final value.
- `BounceInterpolator` : An interpolator whose change bounces at the end.
- `CycleInterpolator` : An interpolator whose animation repeats for a specified number of cycles.
- `DecelerateInterpolator` : An interpolator whose rate of change starts out quickly and and then decelerates.
- `PathInterpolator` : An interpolator that follows a `Path` that you specify.
- `TimeInterpolator` : An interface that allows you to implement your own interpolator.

## Creating a property animation

To animate the property of an object, you always need the following:

- An `Animator` that manages the animation for you. The `Animator` class provides the basic structure for creating animations. You normally do not use the `Animator` class directly, because it only provides minimal functionality that must be extended to fully support animating values. The `ObjectAnimator` subclass does a lot of work for you. For example, `ObjectAnimator` updates the property accordingly when it computes a new value for the animation. Most of the time, use `ObjectAnimator` .

- A `TypeEvaluator`. Evaluators tell the property animation system how to calculate values for a given property. They take the timing data that is provided by an `Animator` class, the animation's start and end value, and calculate the animated values of the property based on this data. You can use one of the provided `IntEvaluator`, `FloatEvaluator`, or `ArgbEvaluator` classes, or you can subclass `TypeEvaluator` to handle other types of data.
- A `TimeInterpolator`. You can use one of the provided interpolators listed above, such as `AccelerateInterpolator`, or you can create your own subclass of `TimeInterpolator`.

To create a basic property animation with `ObjectAnimator`:

1. Create an instance of `ObjectAnimator`. Choose the factory method and supply the appropriate arguments:
   - The object that is to be animated. This can be a `View` or any other object, including a custom object with custom properties.
   - The name of the property to be animated as a string.
   - A starting value.
   - The ending value.

   The following example uses the `ObjectAnimator` class's `ofFloat()` factory method. The `ofFloat()` method animates the radius property of a custom view (called `mView`) from 0 to 50.

   ```
   ObjectAnimator animator = ObjectAnimator.ofFloat(mView,"radius", 0, 50);
   ```

2. Optionally, set an interpolator.

   ```
   animator.setInterpolator(new AccelerateInterpolator());
   ```

3. Specify the duration of your animation in milliseconds.

   ```
   animator.setDuration(400);
   ```

4. Start the animation.

   ```
   animator.start();
   ```

## Required setter for the animated property

To make sure that the `ObjectAnimator` object correctly updates properties:

- The object property that you are animating must have a "setter" function in the form of `set` *`PropertyName`* `()` of a matching type. The `ObjectAnimator` automatically

updates the property during animation, so the `ObjectAnimator` must be able to access the property by using this setter method. For example, if the property name is `radius`, you need a `setRadius()` method.

- If you only specify an ending value for the animation, you need a "getter" function, which must be in the form of `get` *`PropertyName`* `()` of a matching type. For example, if the property name is `radius`, you need a `getRadius()` method.
- Depending on what property or object you are animating, you might need to call the `invalidate()` method on a `View` to force the `View` to redraw itself with the updated animated values. You call `invalidate()` in the `onAnimationUpdate()` callback. For example, animating the color property of a `Drawable` object only causes updates to the screen when that `Drawable` object redraws itself. All of the property setters on views, such as `setAlpha()` and `setTranslationX()`, invalidate the `View` properly, so you do not need to invalidate the `View` when calling these methods with new values.

To learn more about what you can do with property animation, see the Property Animation documentation.

## Animation listeners

For even more control over your animations, you can listen for important events during the animation process. For example, to do something after an animation ends, add a listener and callback for `onAnimationEnd()`.

```
ValueAnimator fadeAnim = ObjectAnimator.ofFloat(newBall, "alpha", 1f, 0f);
fadeAnim.setDuration(250);
fadeAnim.addListener(new AnimatorListenerAdapter() {
public void onAnimationEnd(Animator animation) {
    balls.remove(((ObjectAnimator)animation).getTarget());
}
```

See the `AnimationListener` documentation for details.

## Choreographing multiple animations with an AnimatorSet object

In many cases, you want to group animations together so that they run in relation to one another. For example, you might play an animation when another animation starts or finishes. The Android system lets you bundle animations together into an `AnimatorSet` object. The `AnimatorSet` lets you specify whether to start animations simultaneously, sequentially, or after a specified delay. You can also nest `AnimatorSet` objects within each other, as shown in the code below. The animations are played in the following order:

1. Play `bounceAnim`.
2. Play `squashAnim1`, `squashAnim2`, `stretchAnim1`, and `stretchAnim2` at the same time.
3. Play `bounceBackAnim`.
4. Play `fadeAnim`.

Here is the code:

```
// Create a new AnimatorSet.
AnimatorSet bouncer = new AnimatorSet();

// Specify when animations play relative to each other.
bouncer.play(bounceAnim).before(squashAnim1);
bouncer.play(squashAnim1).with(squashAnim2);
bouncer.play(squashAnim1).with(stretchAnim1);
bouncer.play(squashAnim1).with(stretchAnim2);
bouncer.play(bounceBackAnim).after(stretchAnim2);

// Create a fade animation
ValueAnimator fadeAnim = ObjectAnimator.ofFloat(newBall, "alpha", 1f, 0f);
fadeAnim.setDuration(250);

// Create second animator set that specifies to play the bouncer animation set
// before the fade animation.
AnimatorSet animatorSet = new AnimatorSet();
animatorSet.play(bouncer).before(fadeAnim);
animatorSet.start();
```

See the `AnimatorSet` documentation for more information.

## Property animations in XML

The property animation system lets you declare property animations with XML instead of doing it programmatically. By defining your animations in XML, you can reuse your animations in multiple activities.

Save the XML files for property animations in the `res/animator/` directory.

The following property animation classes have XML declaration support with the following XML tags:

| Class | XML tag |
|---|---|
| ValueAnimator | `animator` |
| ObjectAnimator | `objectAnimator` |
| AnimatorSet | `set` |

To find the attributes that you can use in your XML declaration, see Animation Resources. The following example plays the two sets of object animations sequentially. The first nested set plays two object animations together:

```
<set android:ordering="sequentially">
    <set>
        <objectAnimator
            android:propertyName="x"
            android:duration="500"
            android:valueTo="400"
            android:valueType="intType"/>
        <objectAnimator
            android:propertyName="y"
            android:duration="500"
            android:valueTo="300"
            android:valueType="intType"/>
    </set>
    <objectAnimator
        android:propertyName="alpha"
        android:duration="500"
        android:valueTo="1f"/>
</set>
```

In order to run this animation, you must inflate the XML resources in your code to an `AnimatorSet` object, then set the target objects for all of the animations, and then start the animation set. Call the `setTarget ()` method to set a single target object for all children of the `AnimatorSet`, as a convenience. The following code shows how to do this:

```
AnimatorSet set = (AnimatorSet) AnimatorInflater.loadAnimator(myContext,
    R.anim.property_animator);
set.setTarget(myObject);
set.start();
```

# Drawable animation

`Drawable` animation lets you load a series of `Drawable` resources one after another to create an animation. This is a traditional animation in the sense that it is created with a sequence of different images, played in order, like a roll of film.

The easiest way to create drawable animations is from an XML file. Save the file in `res/drawable`.

The XML file consists of an `<animation-list>` element as the root node and a series of child `<item>` nodes that each define a frame: a drawable resource for the frame and the frame duration. Here's an example XML file for a `Drawable` animation:

```
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="true">
    <item android:drawable="@drawable/rocket_thrust1" android:duration="200" />
    <item android:drawable="@drawable/rocket_thrust2" android:duration="200" />
    <item android:drawable="@drawable/rocket_thrust3" android:duration="200" />
</animation-list>
```

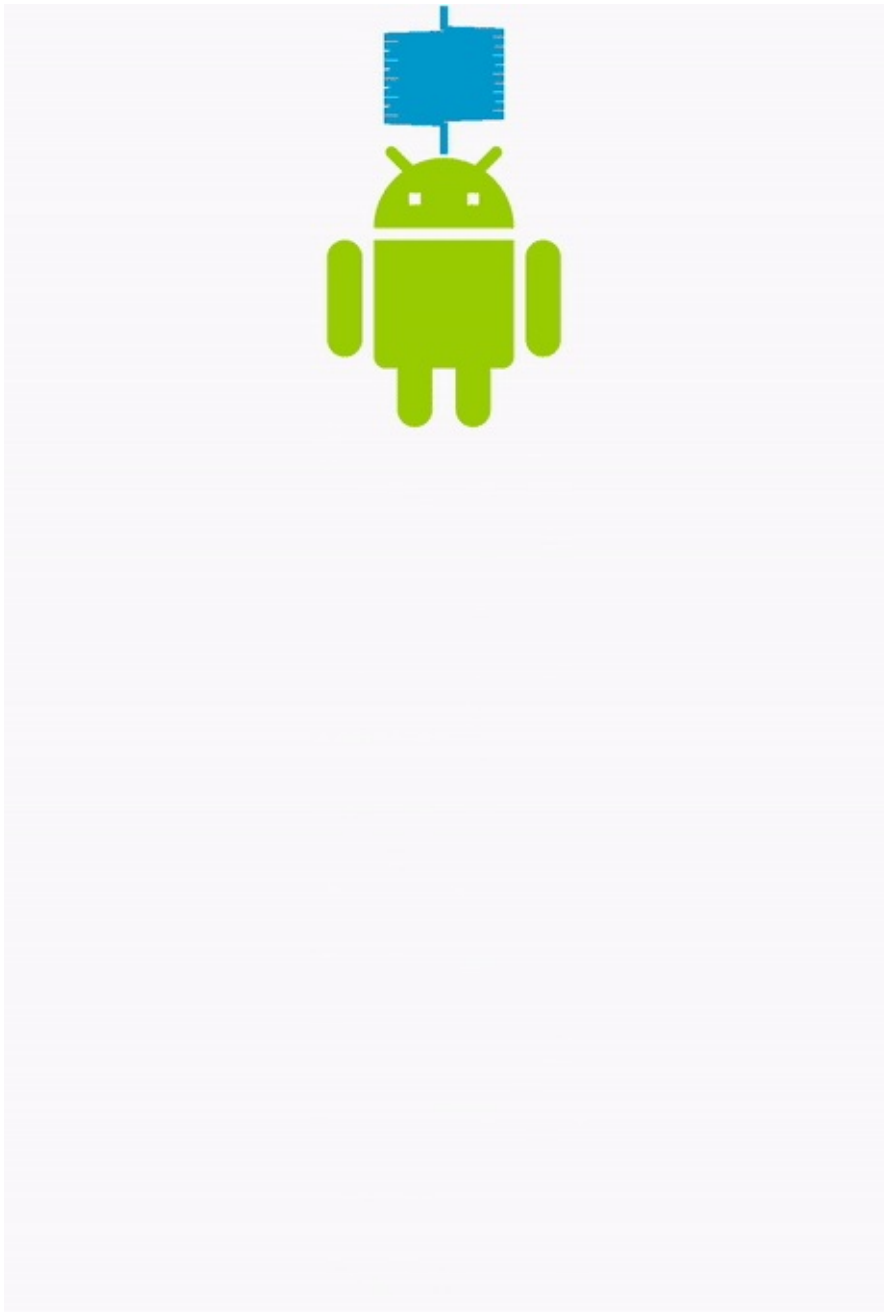
This animation runs for three frames. With the `android:oneshot` attribute of the list to true, it will cycle just once then stop and hold on the last frame. If `android:oneshot` is set `false`, the animation will loop. With this XML saved as `rocket_thrust.xml` in the `res/drawable/` directory of the project, it can be added as the background image to a `View` and then called to play. Here's an example activity, in which the animation is added to an `ImageView` object and animated when the screen is touched:

```
AnimationDrawable rocketAnimation;

public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  ImageView rocketImage = (ImageView) findViewById(R.id.rocket_image);
  rocketImage.setBackgroundResource(R.drawable.rocket_thrust);
  rocketAnimation = (AnimationDrawable) rocketImage.getBackground();
}

public boolean onTouchEvent(MotionEvent event) {
  if (event.getAction() == MotionEvent.ACTION_DOWN) {
    rocketAnimation.start();
    return true;
  }
  return super.onTouchEvent(event);
}
```

See Drawable Animation for more information.

# Physics-based animation

Physics-based animation relies on the laws of physics to manifest a high degree of realism in animation. In our day-to-day life, when a change occurs, it comes with a physical transition that is natural for us to recognize.

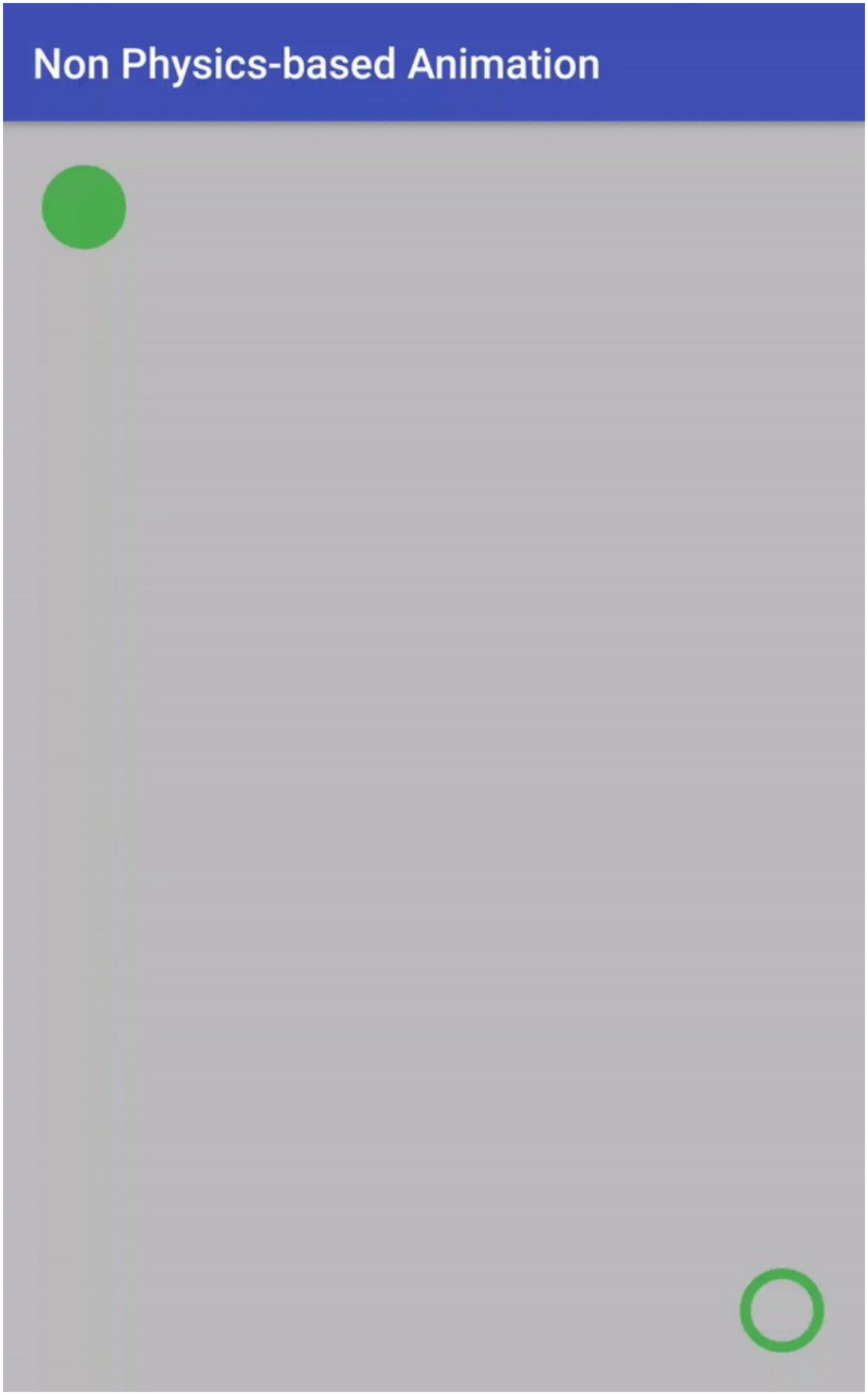


Physics-based animations are driven by force instead of being driven by fixed durations and changes in animation values. The animation comes to rest when the force reaches equilibrium. This results in more natural looking animations and the ability to change an animation during its run without introducing visual disruption.

In summary, physics-based animations have these benefits:

- *Natural-looking.* Physics-based animations are flexible and mimic real time movements. Drawing influence from physics creates motion that is easy to understand and works holistically.
- *Course correction.* Animations keep momentum when their target changes, and end with a smoother motion.

- *Reduced visual janks.* Animations appear more responsive and smooth, and reduce overall visual disruptions.



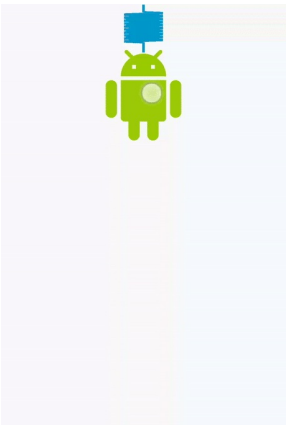

## Creating a physics-based animation

The Android framework supports physics-based animations through the `android.support.animation` API, which was introduced with version 25.3.0 of the Android Support Library. To add the physics-based support library to your project, open the `build.gradle` file of your application and add the library to the dependencies section, replacing the *N* placeholder in the dependency shown below with the current version number:

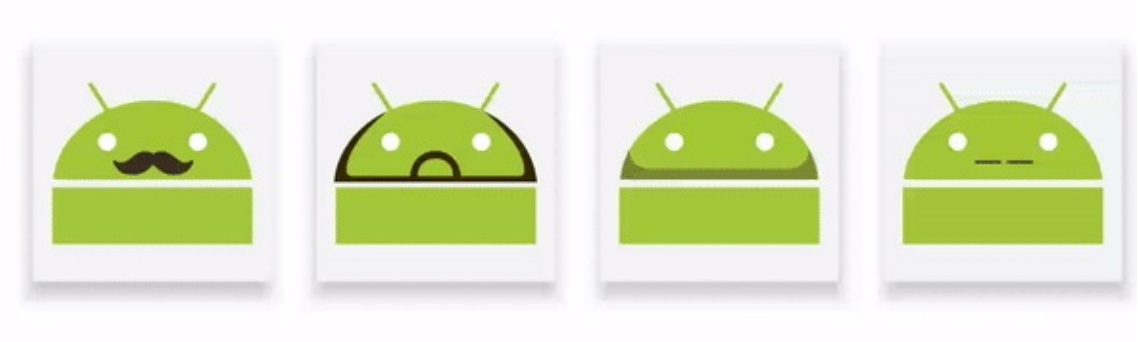
```
dependencies {
    ...
    compile "com.android.support:support-dynamic-animation:N"
}
```

As of this writing, the library has classes for *spring* and *fling* animations.

- Spring animation. In a spring-based animation, you customize the spring's stiffness, its damping ratio, and its final position. When the animation begins, the spring force updates the animation value and the velocity on each frame. The animation continues until the spring force reaches an equilibrium. A spring is the obvious example for this.



- Fling animation. Fling-based animation uses a friction force that is proportional to an object's velocity. Use it to animate a property of an object when you want to end the animation gradually. It has an initial momentum, which is mostly received from the gesture velocity, and gradually slows down. The animation comes to an end when the velocity of the animation is low enough that it makes no visible change on the device screen.



Here is a code snippet for a simple vertical spring animation:

```
final SpringAnimation anim = new SpringAnimation(this, DynamicAnimation.Y, 10)
   .setStartVelocity(10000);
anim.getSpring().setStiffness(STIFFNESS_LOW);
anim.start();
```

See the Spring Animation documentation and `SpringAnimation` class for more information.

Here is a code snippet for a simple rotation fling animation:

```
FlingAnimation fling = new FlingAnimation(this, DynamicAnimation.ROTATION_X);
fling.setStartVelocity(150)
       .setMinValue(0)
       .setMaxValue(1000)
       .setFriction(0.1f)
       .start();
```

See the Fling Animation documentation and `FlingAnimation` class for more information.

# Related practicals

The related exercises and practical documentation is in Advanced Android – Practicals:

- Property Animations

# Learn more

Android developer docs:

- View Animation
- Property Animation
- Drawable Animation
- Physics-based Animations
- `AnimatedVectorDrawable` class
- See the Graphics Architecture series of articles for an in-depth explanation of how the Android framework draws to the screen.