https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

Lesson 2: Activities and intents

2.1: Activities and intents

Contents:

Introduction

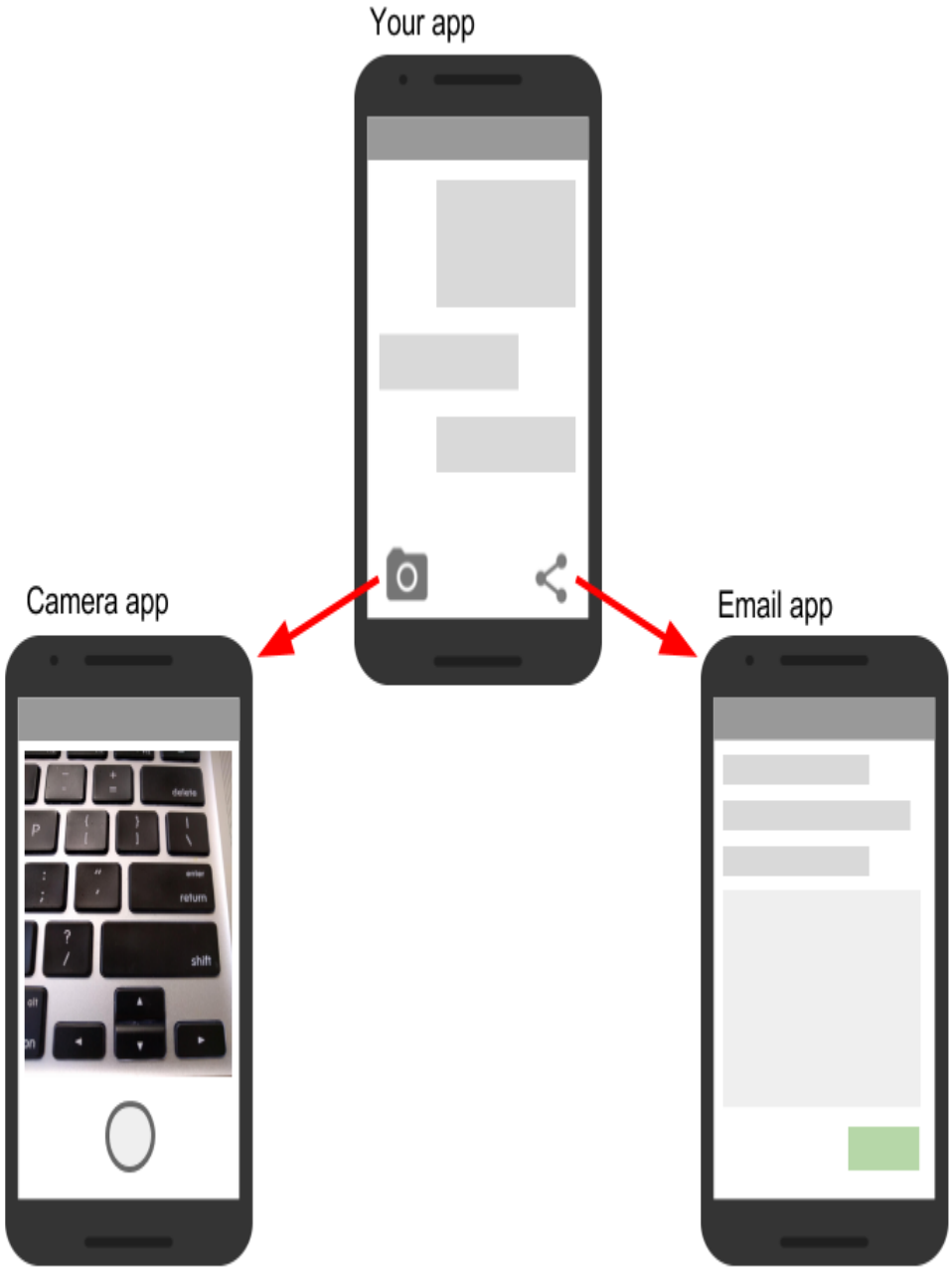
In this chapter you learn about the `Activity` class, the major building block of your app's user interface (UI). You also learn about using an `Intent` to communicate from one activity to another.

About activities

An *activity* represents a single screen in your app with an interface the user can interact with. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading individual messages. Your app is probably a collection of activities that you create yourself, or that you reuse from other apps.

Although the activities in your app work with each other to form a cohesive user experience, each activity is independent of the others. This enables your app to start an activity in another app, and it enables other apps to start activities in your app (if your app allows this). For example, a messaging app could start an activity in a camera app to take a picture, then start an activity in

an email app to let the user share the picture in email.



Typically, one `Activity` in an app is specified as the "main" activity, for example `MainActivity`. The user sees the main activity when they launch the app for the first time. Each activity can start other activities to perform different actions.
Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When the user is done with the current activity and presses the Back button, the activity is popped from the stack and destroyed, and the previous activity resumes.

When an activity is stopped because a new activity starts, the first activity is notified by way of the activity lifecycle callback methods. The *activity lifecycle* is the set of states an `Activity` can be in: when the activity is first created, when it's stopped or resumed, and when the system destroys it. You learn more about the activity lifecycle in a later chapter.

Creating an Activity

To implement an `Activity` in your app, do the following:

- Create an `Activity` Java class.
- Implement a basic UI for the `Activity` in an XML layout file.
- Declare the new `Activity` in the `AndroidManifest.xml` file.

When you create a new project for your app, or add a new `Activity` to your app by choosing **File > New > Activity**, the template automatically performs the steps listed above.

Create the Activity

When you create a new project in Android Studio and choose the **Backwards Compatibility (AppCompat)** option, the `MainActivity` is, by default, a subclass of the `AppCompatActivity` class. The `AppCompatActivity` class lets you use up-to-date Android app features such as the app bar and Material Design, while still enabling your app to be compatible with devices running older versions of Android.
Here is a skeleton subclass of `AppCompatActivity`:

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

The first task for you in your `Activity` subclass is to implement the standard `Activity` lifecycle callback methods (such as `onCreate()`) to handle the state changes for your `Activity`. These state changes include things such as when the `Activity` is created, stopped, resumed, or destroyed. You learn more about the `Activity` lifecycle and lifecycle callbacks in a different chapter.
The one required callback your app must implement is the `onCreate()` method. The system calls this method when it creates your `Activity`, and all the essential components of your `Activity` should be initialized here. Most importantly, the `onCreate()` method calls `setContentView()` to create the primary layout for the `Activity`.
You typically define the UI for your `Activity` in one or more XML layout files. When the `setContentView()` method is called with the path to a layout file, the system creates all the initial views from the specified layout and adds them to your `Activity`. This is often referred to as *inflating* the layout.
You may often also want to implement the `onPause()` method in your `Activity`. The system calls this method as the first indication that the user is leaving your `Activity` (though it does not always mean that the `Activity` is being destroyed). This is usually where you should commit any changes that should be persisted beyond the current user session (because the user might not come back). You learn more about `onPause()` and all the other lifecycle callbacks in a later chapter.
In addition to lifecycle callbacks, you may also implement methods in your `Activity` to handle other behavior such as user input or button clicks.

Implement the activity's UI

The UI for an activity is provided by a hierarchy of `View` elements, which controls a particular space within the activity window and can respond to user interaction.
The most common way to define a UI using `View` elements is with an XML layout file stored as part of your app's resources. Defining your layout in XML enables you to maintain the design of your UI separately from the source code that defines the activity behavior.
You can also create new `View` elements directly in your activity code by inserting new `View` objects into a `ViewGroup`, and then passing the root `ViewGroup` to `setContentView()`. After your layout has been inflated—regardless of its source—you can add more `View` elements anywhere in the `View` hierarchy.

Declare the Activity in AndroidManifest.xml

Each `Activity` in your app must be declared in the `AndroidManifest.xml` file with the `<activity>` element, inside the `<application>` section. When you create a new project or add a new `Activity` to your project in Android Studio, the `AndroidManifest.xml` file is created or updated to include skeleton declarations for each `Activity`. Here's the declaration for `MainActivity`:

```
<activity android:name=".MainActivity" >
   <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
   </intent-filter>
</activity>
```

The `<activity>` element includes a number of attributes to define properties of the `Activity` such as its label, icon, or theme. The only required attribute is `android:name`, which specifies the class name for the `Activity` (such as `MainActivity`). See the `<activity>` element reference for more information on `Activity` declarations.
The `<activity>` element can also include declarations for `Intent` filters. The `Intent` filters specify the kind of `Intent` your `Activity` will accept.

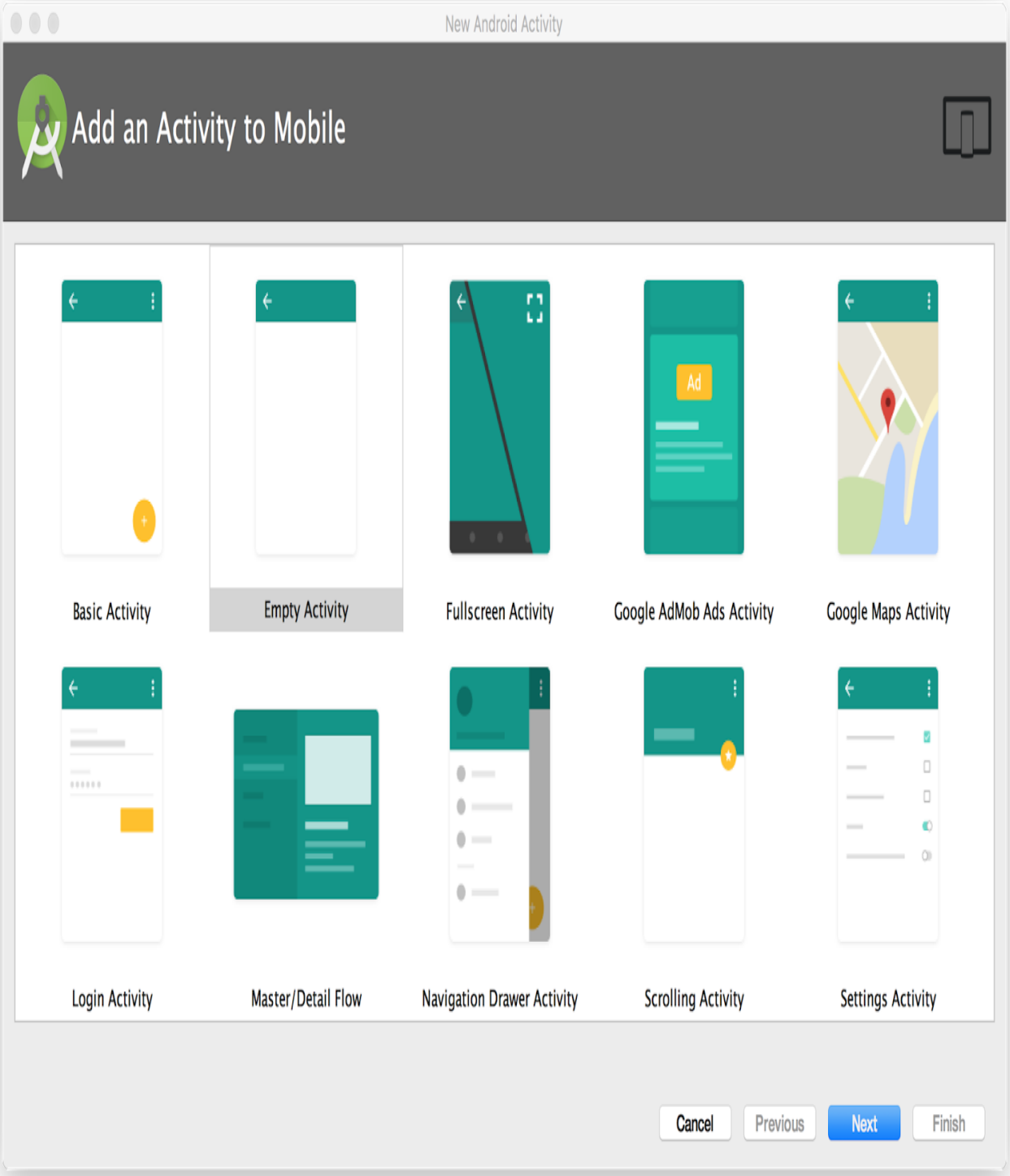
```
<intent-filter>
   <action android:name="android.intent.action.MAIN" />
   <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

`Intent` filters must include at least one `<action>` element, and can also include a `<category>` and optional `<data>`. The `MainActivity` for your app needs an `Intent` filter that defines the "main" action and the "launcher" category so that the system can launch your app. Android Studio creates this `Intent` filter for the `MainActivity` in your project.
The `<action>` element specifies that this is the "main" entry point to the app.
The `<category>` element specifies that this `Activity` should be listed in the system's app launcher (to allow users to launch this `Activity`).
Each `Activity` in your app can also declare `Intent` filters, but only your `MainActivity` should include the "main" action. You learn more about how to use an implicit `Intent` and `Intent` filters in a later section.

Add another Activity to your project

The `MainActivity` for your app and its associated layout file is supplied by an `Activity` template in Android Studio such as Empty Activity or Basic Activity. You can add a new `Activity` to your project by choosing **File > New > Activity**. Choose the `Activity` template you want to use, or

open the Gallery to see all the available templates.



When you choose an `Activity` template, you see the same set of screens for creating the new activity that you did when you created the project. Android Studio provides three things for each new activity in your app:

- A Java file for the new `Activity` with a skeleton class definition and `onCreate()` method. The new `Activity`, like `MainActivity`, is a subclass of `AppCompatActivity`.
- An XML file containing the layout for the new activity. Note that the `setContentView()` method in the `Activity` class inflates this new layout.
- An additional `<activity>` element in the `AndroidManifest.xml` file that specifies the new activity. The second `Activity` definition does not include any `Intent` filters. If you plan to use this activity only within your app (and not enable that activity to be started by any other app), you do not need to add filters.

About intents

Each activity is started or activated with an `Intent`, which is a message object that makes a request to the Android runtime to start an activity or other app component in your app or in some other app.
When your app is first started from the device home screen, the Android runtime sends an `Intent` to your app to start your app's main activity (the one defined with the MAIN action and the LAUNCHER category in the `AndroidManifest.xml` file). To start another activity in your app, or to request that some other activity available on the device perform an action, you build your own intent and call the `startActivity()` method to send the intent.
In addition to starting an activity, an intent can also be used to pass data between one activity and another. When you create an intent to start a new activity, you can include information about the data you want that new activity to operate on. So, for example, an email `Activity` that displays a list of messages can send an `Intent` to the `Activity` that displays that message. The display activity needs data about the message to display, and you can include that data in the intent.
In this chapter you learn about using intents with activities, but intents can also be used to start services or broadcast receivers. You learn how to use those app components in another practical.

Intent types

Intents can be *explicit* or *implicit*:

- *Explicit intent:* You specify the receiving activity (or other component) using the activity's fully qualified class name. You use explicit intents to start components in your own app (for example, to move between screens in the UI), because you already know the package and class name of that component.
- *Implicit intent:* You do *not* specify a specific activity or other component to receive the intent. Instead, you declare a general action to perform, and the Android system matches your request to an activity or other component that can handle the requested action. You learn more about using implicit intents in another practical.

Intent objects and fields

For an explicit `Intent`, the key fields include the following:

- The `Activity` *class* (for an explicit `Intent`). This is the class name of the `Activity` or other component that should receive the `Intent`; for example, `com.example.SampleActivity.class`. Use the `Intent` constructor or the `setComponent()`, `setComponentName()`, or `setClassName()` methods to specify the class.
- The `Intent` *data.* The `Intent` data field contains a reference to the data you want the receiving `Activity` to operate on as a Uri object.
- `Intent` *extras.* These are key-value pairs that carry information the receiving `Activity` requires to accomplish the requested action.
- `Intent` *flags.* These are additional bits of metadata, defined by the `Intent` class. The flags may instruct the Android system how to launch an `Activity` or how to treat it after it's launched.

For an implicit `Intent`, you may need to also define the `Intent` action and category. You learn more about `Intent` actions and categories in another chapter.

Starting an Activity with an explicit Intent

To start a specific `Activity` from another `Activity`, use an explicit `Intent` and the `startActivity()` method. An explicit `Intent` includes the fully qualified class name for the `Activity` or other component in the `Intent` object. All the other `Intent` fields are optional, and `null` by default.
For example, if you want to start the `ShowMessageActivity` to show a specific message in an email app, use code like this:

```
Intent messageIntent = new Intent(this, ShowMessageActivity.class);
startActivity(messageIntent);
```

The intent constructor takes two arguments for an explicit `Intent`:

- An application context. In this example, the `Activity` class provides the context (`this`).
- The specific component to start (`ShowMessageActivity.class`).

Use the `startActivity()` method with the new `Intent` object as the only argument.
The `startActivity()` method sends the `Intent` to the Android system, which launches the `ShowMessageActivity` class on behalf of your app. The new `Activity` appears on the screen, and the originating `Activity` is paused.
The started `Activity` remains on the screen until the user taps the Back button on the device, at which time that `Activity` closes and is reclaimed by the system, and the originating `Activity` is resumed. You can also manually close the started `Activity` in response to a user action (such as a `Button` click) with the `finish()` method:

```
public void closeActivity (View view) {
    finish();
}
```

Passing data from one Activity to another

In addition to simply starting one `Activity` from another `Activity`, you also use an `Intent` to *pass information* from one `Activity` to another. The `Intent` object you use to start an `Activity` can include `Intent` *data* (the URI of an object to act on), or `Intent` *extras*, which are bits of additional data the `Activity` might need.
In the first (sending) `Activity`, you:

1. Create the `Intent` object.
2. Put data or extras into that `Intent`.
3. Start the new `Activity` with `startActivity()`.

In the second (receiving) `Activity`, you:

1. Get the `Intent` object the `Activity` was started with.
2. Retrieve the data or extras from the `Intent` object.

When to use Intent data or Intent extras

You can use either `Intent` data or `Intent` extras to pass data from one `Activity` to another. There are several key differences between data and extras that determine which you should use.
The `Intent` *data* can hold only one piece of information: a URI representing the location of the data you want to operate on. That URI could be a web page URL (`http://`), a telephone number (`tel://`), a geographic location (`geo://`) or any other custom URI you define.
Use the `Intent` data field:

- When you only have one piece of information you need to send to the started `Activity`.
- When that information is a data location that can be represented by a URI.

Intent *extras* are for any other arbitrary data you want to pass to the started `Activity`. `Intent` extras are stored in a `Bundle` object as key and value pairs. A `Bundle` is a map, optimized for Android, in which a key is a string, and a value can be any primitive or object type (objects must implement the `Parcelable` interface). To put data into the `Intent` extras you can use any of the `Intent` class `putExtra()` methods, or create your own `Bundle` and put it into the `Intent` with `putExtras()`.
Use the `Intent` extras:

- If you want to pass more than one piece of information to the started `Activity`.
- If any of the information you want to pass is not expressible by a URI.

`Intent` data and extras are not exclusive; you can use data for a URI and extras for any additional information the started `Activity` needs to process the data in that URI.

Add data to the Intent

To add data to an explicit `Intent` from the originating `Activity`, create the `Intent` object as you did before:

```
Intent messageIntent = new Intent(this, ShowMessageActivity.class);
```

Use the `setData()` method with a Uri object to add that URI to the `Intent`. Some examples of using `setData()` with URIs:

```
// A web page URL
messageIntent.setData(Uri.parse("http://www.google.com"));
// a Sample file URI
messageIntent.setData(Uri.fromFile(new File("/sdcard/sample.jpg")));
// A sample content: URI for your app's data model
messageIntent.setData(Uri.parse("content://mysample.provider/data"));
// Custom URI
messageIntent.setData(Uri.parse("custom:" + dataID + buttonId));
```

Keep in mind that the data field can only contain a single URI; if you call `setData()` multiple times only the last value is used. Use `Intent` extras to include additional information (including URIs.)

After you've added the data, you can start the `Activity` with the `Intent` as usual:

```
startActivity(messageIntent);
```

Add extras to the Intent

To add `Intent` extras to an explicit `Intent` from the originating `Activity`:

1. Determine the keys to use for the information you want to put into the extras, or define your own. Each piece of information needs its own unique key.
2. Use the `putExtra()` methods to add your key/value pairs to the `Intent` extras. Optionally you can create a `Bundle` object, add your data to the `Bundle`, and then add the `Bundle` to the `Intent`.

The `Intent` class includes extra keys you can use, defined as constants that begin with the word `EXTRA_`. For example, you could use `Intent.EXTRA_EMAIL` to indicate an array of email addresses (as strings), or `Intent.EXTRA_REFERRER` to specify information about the originating `Activity` that sent the `Intent`.

You can also define your own `Intent` extra keys. Conventionally you define `Intent` extra keys as static variables with names that begin with `EXTRA_`. To guarantee that the key is unique, the string value for the key itself should be prefixed with your app's fully qualified class name. For example:

```
public final static String EXTRA_MESSAGE =
                                    "com.example.mysampleapp.MESSAGE";
public final static String EXTRA_POSITION_X = "com.example.mysampleapp.X";
public final static String EXTRA_POSITION_Y = "com.example.mysampleapp.Y";
```

Create an `Intent` object (if one does not already exist):

```
Intent messageIntent = new Intent(this, ShowMessageActivity.class);
```

Use a `putExtra()` method with a key to put data into the `Intent` extras. The `Intent` class defines many `putExtra()` methods for different kinds of data:

```
messageIntent.putExtra(EXTRA_MESSAGE, "this is my message");
messageIntent.putExtra(EXTRA_POSITION_X, 100);
messageIntent.putExtra(EXTRA_POSITION_Y, 500);
```

Alternately, you can create a new `Bundle` and populate that `Bundle` with your `Intent` extras. `Bundle` defines many "put" methods for different kinds of primitive data as well as objects that implement Android's `Parcelable` interface or Java's `Serializable`.

```
Bundle extras = new Bundle();
extras.putString(EXTRA_MESSAGE, "this is my message");
extras.putInt(EXTRA_POSITION_X, 100);
extras.putInt(EXTRA_POSITION_Y, 500);
```

After you've populated the `Bundle`, add it to the `Intent` with the `putExtras()` method (note the "s" in `Extras`):

```
messageIntent.putExtras(extras);
```

Start the `Activity` with the `Intent` as usual:

```
startActivity(messageIntent);
```

Retrieve the data from the Intent in the started Activity

When you start an `Activity` with an `Intent`, the started `Activity` has access to the `Intent` and the data it contains.

To retrieve the `Intent` the `Activity` (or other component) was started with, use the `getIntent()` method:

```
Intent intent = getIntent();
```

Use `getData()` to get the URI from that `Intent`:

```
Uri locationUri = intent.getData();
```

To get the extras out of the `Intent`, you need to know the keys for the key/value pairs. You can use the standard `Intent` extras if you used those, or you can use the keys you defined in the originating `Activity` (if they were defined as public.)

Use one of the `getExtra()` methods to extract extra data out of the `Intent` object:

```
String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
int positionX = intent.getIntExtra(MainActivity.EXTRA_POSITION_X);
int positionY = intent.getIntExtra(MainActivity.EXTRA_POSITION_Y);
```

Or you can get the entire extras `Bundle` from the `Intent` and extract the values with the various `Bundle` methods:

```
Bundle extras = intent.getExtras();
String message = extras.getString(MainActivity.EXTRA_MESSAGE);
```

Getting data back from an Activity

When you start an `Activity` with an `Intent`, the originating `Activity` is paused, and the new `Activity` remains on the screen until the user clicks the Back button, or you call the `finish()` method in a click handler or other function that ends the user's involvement with this `Activity`.

Sometimes when you send data to an `Activity` with an `Intent`, you would like to also get data back from that `Intent`. For example, you might start a photo gallery `Activity` that lets the user pick a photo. In this case your original `Activity` needs to receive information about the photo the user chose back from the launched `Activity`.

To launch a new `Activity` and get a result back, do the following steps in your originating `Activity`:

1. Instead of launching the `Activity` with `startActivity()`, call `startActivityForResult()` with the `Intent` and a request code.
2. Create a new `Intent` in the launched `Activity` and add the return data to that `Intent`.
3. Implement `onActivityResult()` in the originating `Activity` to process the returned data.

You learn about each of these steps in the following sections.

Use startActivityForResult() to launch the Activity

To get data back from a launched `Activity`, start that `Activity` with the `startActivityForResult()` method instead of `startActivity()`.

```
startActivityForResult(messageIntent, TEXT_REQUEST);
```

The `startActivityForResult()` method, like `startActivity()`, takes an `Intent` argument that contains information about the `Activity` to be launched and any data to send to that `Activity`.
The `startActivityForResult()` method, however, also needs a request code.
The request code is an integer that identifies the request and can be used to differentiate between results when you process the return data. For example, if you launch one `Activity` to take a photo and another to pick a photo from a gallery, you need different request codes to identify which request the returned data belongs to.
Conventionally you define request codes as static integer variables with names that include `REQUEST`. Use a different integer for each code. For example:

```
public static final int PHOTO_REQUEST = 1;
public static final int PHOTO_PICK_REQUEST = 2;
public static final int TEXT_REQUEST = 3;
```

Return a response from the launched Activity

The response data from the launched `Activity` back to the originating `Activity` is sent in an `Intent`, either in the data or the extras. You construct this return `Intent` and put the data into it in much the same way you do for the sending `Intent`. Typically your launched `Activity` will have an `onClick()` or other user input callback method in which you process the user's action and close the `Activity`. This is also where you construct the response.
To return data from the launched `Activity`, create a new empty `Intent` object.

```
Intent returnIntent = new Intent();
```

Note: To avoid confusing sent data with returned data, use a new `Intent` object rather than reusing the original sending `Intent` object.

A return result `Intent` does not need a class or component name to end up in the right place. The Android system directs the response back to the originating `Activity` for you.
Add data or extras to the `Intent` the same way you did with the original `Intent`. You may need to define keys for the return `Intent` extras at the start of your class.

```
public final static String EXTRA_RETURN_MESSAGE =
                                "com.example.mysampleapp.RETURN_MESSAGE";
```

Then put your return data into the `Intent` as usual. In the following, the return message is an `Intent` extra with the key `EXTRA_RETURN_MESSAGE`.

```
messageIntent.putExtra(EXTRA_RETURN_MESSAGE, mMessage);
```

Use the `setResult()` method with a response code and the `Intent` with the response data:

```
setResult(RESULT_OK,replyIntent);
```

The response codes are defined by the `Activity` class, and can be

- `RESULT_OK`: The request was successful.
- `RESULT_CANCELED`: The user canceled the operation.
- `RESULT_FIRST_USER`: For defining your own result codes.

You use the result code in the originating `Activity`.
Finally, call `finish()` to close the `Activity` and resume the originating `Activity`:

```
finish();
```

Read response data in onActivityResult()

Now that the launched `Activity` has sent data back to the originating `Activity` with an `Intent`, that first `Activity` must handle that data. To handle returned data in the originating `Activity`, implement the `onActivityResult()` callback method. Here is a simple example.

```
public void onActivityResult(int requestCode, int resultCode,  Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == TEXT_REQUEST) {
        if (resultCode == RESULT_OK) {
            String reply =
               data.getStringExtra(SecondActivity.EXTRA_RETURN_MESSAGE);
               // process data
        }
    }
}
```

The three arguments to `onActivityResult()` contain all the information you need to handle the return data.

- *Request code:* The request code you set when you launched the `Activity` with `startActivityForResult()`. If you launch a different `Activity` to accomplish different operations, use this code to identify the specific data you're getting back.
- *Result code:* the result code set in the launched `Activity`, usually one of `RESULT_OK` or `RESULT_CANCELED`.
- *Intent data:* the `Intent` that contains the data returned from the launch `Activity`.

The example method shown above shows the typical logic for handling the request and response codes. The first test is for the `TEXT_REQUEST` request, and that the result was successful. Inside the body of those tests you extract the return information out of the `Intent`. Use `getData()` to get the `Intent` data, or `getExtra()` to retrieve values out of the `Intent` extras with a specific key.

Activity navigation

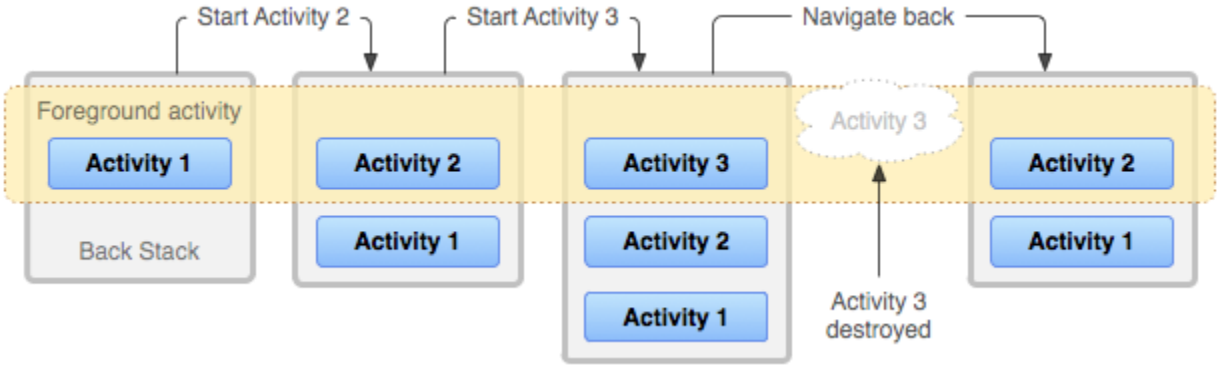
Any app of any complexity that you build will include more than one `Activity`. As your users move around your app and from one `Activity` to another, consistent navigation becomes more important to the app's user experience. Few things frustrate users more than basic navigation that behaves in inconsistent and unexpected ways. Thoughtfully designing your app's navigation will make using your app predictable and reliable for your users.
Android system supports two different forms of navigation strategies for your app.

- Back (temporal) navigation, provided by the device Back button, and the back stack.
- Up (ancestral) navigation, provided by you as an option in the app bar.

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

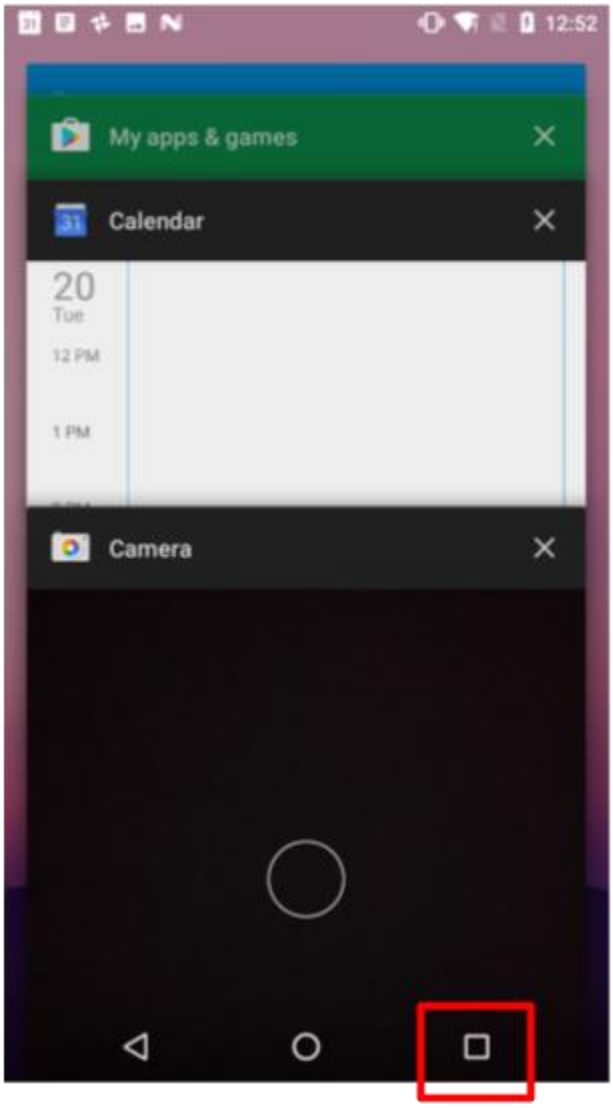
Back navigation, tasks, and the back stack

Back navigation allows your users to return to the previous `Activity` by tapping the device back button  . Back navigation is also called *temporal* navigation because the back button navigates the history of recently viewed screens, in reverse chronological order.
The *back stack* is the set of each `Activity` that the user has visited and that can be returned to by the user with the back button. Each time a new `Activity` starts, it is pushed onto the back stack and takes user focus. The previous `Activity` is stopped but is still available in the back stack. The back stack operates on a "last in, first out" mechanism, so when the user is done with the current `Activity` and presses the Back button, that `Activity` is popped from the stack (and destroyed) and the previous `Activity` resumes.
Because an app can start an `Activity` both inside and outside a single app, the back stack contains each `Activity` that has been launched by the user in reverse order. Each time the user presses the Back button, each `Activity` in the stack is popped off to reveal the previous one, until the user returns to the Home screen.



Android provides a back stack for each *task*. A task is an organizing concept for each `Activity` the user interacts with when performing an operation, whether they are inside your app or across multiple apps. Most tasks start from the Android home screen, and tapping an app icon starts a task (and a new back stack) for that app. If the user uses an app for a while, taps home, and starts a new app, that new app launches in its own task and has its own back stack. If the user returns to the first app, that first task's back stack returns. Navigating with the Back button returns only to the `Activity` in the current task, not for all tasks running on the device. Android enables the user to navigate between tasks with the overview or recent tasks screen,

accessible with the square button on lower right corner of the device  .

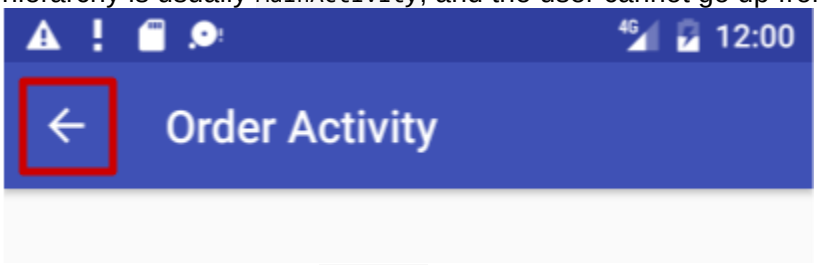


In most cases you don't have to worry about managing either tasks or the back stack for your app—the system keeps track of these things for you, and the back button is always available on the device.

There may, however, be times where you may want to override the default behavior for tasks or for the back stack. For example, if your screen contains an embedded web browser where users can navigate between web pages, you may wish to use the browser's default back behavior when users press the device's *Back* button, rather than returning to the previous `Activity`. You may also need to change the default behavior for your app in other special cases such as with notifications or widgets, where an `Activity` deep within your app may be launched as its own task, with no back stack at all. You learn more about managing tasks and the back stack in a later section.

Up navigation

Up navigation, sometimes referred to as ancestral or logical navigation, is used to navigate within an app based on the explicit hierarchical relationships between screens. With Up navigation, each `Activity` is arranged in a hierarchy, and each "child" `Activity` shows a left-facing arrow in the app bar  that returns the user to the "parent" `Activity`. The topmost `Activity` in the hierarchy is usually `MainActivity`, and the user cannot go up from there.



For instance, if the main `Activity` in an email app is a list of all messages, selecting a message launches a second `Activity` to display that single email. In this case the message `Activity` would provide an Up button that returns to the list of messages.
The behavior of the Up button is defined by you in each `Activity` based on how you design your app's navigation. In many cases, Up and Back navigation may provide the same behavior: to just return to the previous `Activity`. For example, a Settings `Activity` may be available from any `Activity` in your app, so "up" is the same as back—just return the user to their previous place in the hierarchy.
Providing Up behavior for your app is optional, but a good design practice, to provide consistent navigation for your app.

Implement Up navigation with a parent Activity

With the standard template projects in Android Studio, it's straightforward to implement Up navigation. If one `Activity` is a child of another `Activity` in your app's `Activity` hierarchy, specify the parent of that other `Activity` in the `AndroidManifest.xml` file.
Beginning in Android 4.1 (API level 16), declare the logical parent of each `Activity` by specifying the `android:parentActivityName` attribute in the `<activity>` element. To support older versions of Android, include `<meta-data>` information to define the parent `Activity` explicitly. Use both methods to be backwards-compatible with all versions of Android.

The following are the skeleton definitions in `AndroidManifest.xml` for both a main (parent) `Activity` (`MainActivity`) and a second (child) `Activity` (`SecondActivity`):

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <!-- The main activity (it has no parent activity) -->
    <activity android:name=".MainActivity">
       <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER" />
       </intent-filter>
    </activity>
    <!-- The child activity) -->
    <activity android:name=".SecondActivity"
       android:label = "Second Activity"
       android:parentActivityName=".MainActivity">
       <meta-data
          android:name="android.support.PARENT_ACTIVITY"
          android:value="com.example.android.twoactivities.MainActivity" />
       </activity>
</application>
```

You learn more about Up navigation and other user navigation features in another practical.

Related practical

The related practical is 2.1: Activities and intents.

Learn more

Android Studio documentation:

- Meet Android Studio

Android developer documentation:

- Application Fundamentals
- Activities
- Intents and Intent Filters
- Designing Back and Up navigation
- `Activity`
- `Intent`
- `ScrollView`
- `View`

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

2.2: Activity lifecycle and state

Contents:

Introduction

In this chapter you learn about the activity lifecycle, the callback events you can implement to perform tasks in each stage of the lifecycle, and how to handle `Activity` instance states throughout the activity lifecycle.

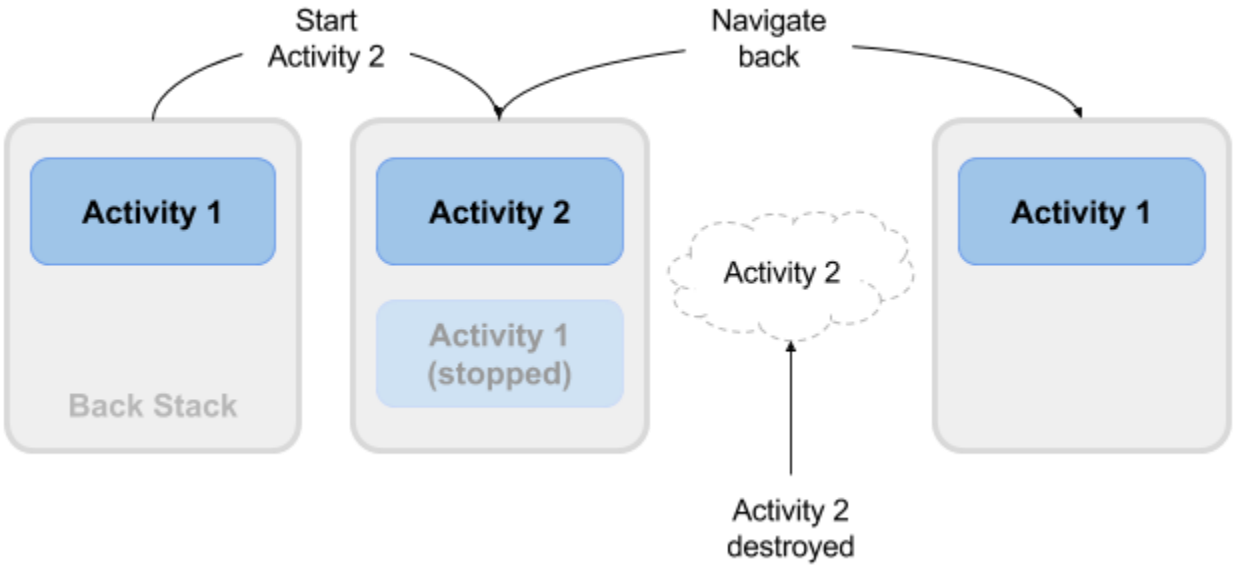
About the activity lifecycle

The activity lifecycle is the set of states an activity can be in during its entire lifetime, from the time it's created to when it's destroyed and the system reclaims its resources. As the user interacts with your app and other apps on the device, activities move into different states.

For example:

- When you start an app, the app's main activity ("Activity 1" in the figure below) is started, comes to the foreground, and receives the user focus.
- When you start a second activity ("Activity 2" in the figure below), a new activity is created and started, and the main activity is stopped.
- When you're done with the Activity 2 and navigate back, Activity 1 resumes. Activity 2 stops and is no longer needed.

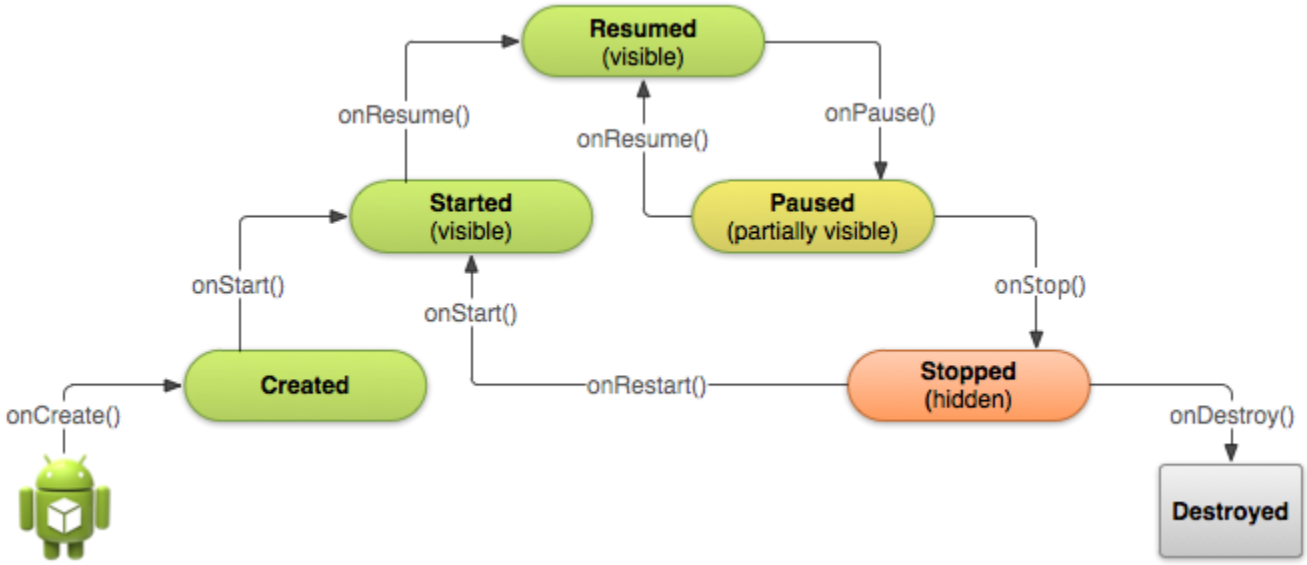
- If the user doesn't resume Activity 2, the system eventually destroys it.



Activity states and lifecycle callback methods

When an `Activity` transitions into and out of the different lifecycle states as it runs, the Android system calls several lifecycle callback methods at each stage. All of the callback methods are hooks that you can override in each of your Activity classes to define how that `Activity` behaves when the user leaves and re-enters the `Activity`. Keep in mind that the lifecycle states (and callbacks) are per `Activity`, not per app, and you may implement different behavior at different points in the lifecycle of each `Activity`.

This figure shows each of the `Activity` states and the callback methods that occur as the `Activity` transitions between different states:



Depending on the complexity of your `Activity`, you probably don't need to implement all the lifecycle callback methods in an `Activity`. However, it's important that you understand each one and implement those that ensure your app behaves the way users expect. Managing the lifecycle of an `Activity` by implementing callback methods is crucial to developing a strong and flexible app.

Activity created: the onCreate() method

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // The activity is being created.
}
```

Your `Activity` enters into the created state when it is started for the first time. When an `Activity` is first created, the system calls the `onCreate()` method to initialize that `Activity`. For example, when the user taps your app icon from the Home screen to start that app, the system calls the `onCreate()` method for the `Activity` in your app that you've declared to be the "launcher" or "main" `Activity`. In this case the main `Activity` `onCreate()` method is analogous to the `main()` method in other programs.
Similarly, if your app starts another `Activity` with an `Intent` (either explicit or implicit), the system matches your `Intent` request with an `Activity` and calls `onCreate()` for that new `Activity`.
The `onCreate()` method is the only required callback you must implement in your `Activity` class. In your `onCreate()` method you perform basic app startup logic that should happen only once, such as setting up the user interface, assigning class-scope variables, or setting up background tasks.
Created is a transient state; the `Activity` remains in the created state only as long as it takes to run `onCreate()`, and then the `Activity` moves to the started state.

Activity started: the onStart() method

```
@Override
protected void onStart() {
    super.onStart();
    // The activity is about to become visible.
}
```

After your `Activity` is initialized with `onCreate()`, the system calls the `onStart()` method, and the `Activity` is in the started state. The `onStart()` method is also called if a stopped `Activity` returns to the foreground, such as when the user clicks the Back button or the Up button to navigate to the previous screen.

While `onCreate()` is called only once when the `Activity` is created, the `onStart()` method may be called many times during the lifecycle of the `Activity` as the user navigates around your app.

When an `Activity` is in the started state and visible on the screen, the user cannot interact with it until `onResume()` is called, the `Activity` is running, and the `Activity` is in the foreground.

Typically you implement `onStart()` in your `Activity` as a counterpart to the `onStop()` method. For example, if you release hardware resources (such as GPS or sensors) when the `Activity` is stopped, you can re-register those resources in the `onStart()` method.

Started, like created, is a transient state. After starting, the `Activity` moves into the resumed (running) state.

Activity resumed/running: the onResume() method

```
@Override
protected void onResume() {
    super.onResume();
    // The activity has become visible (it is now "resumed").
}
```

Your `Activity` is in the resumed state when it is initialized, visible on screen, and ready to use. The resumed state is often called the running state, because it is in this state that the user is actually interacting with your app.

The first time the `Activity` is started the system calls the `onResume()` method just after `onStart()`. The `onResume()` method may also be called multiple times, each time the app comes back from the paused state.

As with the `onStart()` and `onStop()` methods, which are implemented in pairs, you typically only implement `onResume()` as a counterpart to `onPause()`. For example, if in the `onPause()` method you halt any animations, you would start those animations again in `onResume()`.

The `Activity` remains in the resumed state as long as the `Activity` is in the foreground and the user is interacting with it. From the resumed state the `Activity` can move into the paused state.

Activity paused: the onPause() method

```
@Override
protected void onPause() {
    super.onPause();
    // Another activity is taking focus
    // (this activity is about to be "paused").
}
```

The paused state can occur in several situations:

- The `Activity` is going into the background, but has not yet been fully stopped. This is the first indication that the user is leaving your `Activity`.
- The `Activity` is only partially visible on the screen, because a dialog or other transparent `Activity` is overlaid on top of it.
- In multi-window or split screen mode (API 24), the `Activity` is displayed on the screen, but some other `Activity` has the user focus.

The system calls the `onPause()` method when the `Activity` moves into the paused state. Because the `onPause()` method is the first indication you get that the user may be leaving the `Activity`, you can use `onPause()` to stop animation or video playback, release any hardware-intensive resources, or commit unsaved `Activity` changes (such as a draft email).
The `onPause()` method should execute quickly. Don't use `onPause()` for CPU-intensive operations such as writing persistent data to a database. The app may still be visible on screen as it passes through the paused state, and any delays in executing `onPause()` can slow the user's transition to the next `Activity`. Implement any heavy-load operations when the app is in the stopped state instead.
Note that in multi-window mode (API 24), your paused `Activity` may still fully visible on the screen. In this case you do not want to pause animations or video playback as you would for a partially visible `Activity`. You can use the `inMultiWindowMode()` method in the `Activity` class to test whether your app is running in multi-window mode.
Your `Activity` can move from the paused state into the resumed state (if the user returns to the `Activity`) or to the stopped state (if the user leaves the `Activity` altogether).

Activity stopped: the onStop() method

```
@Override
protected void onStop() {
    super.onStop();
    // The activity is no longer visible (it is now "stopped")
}
```

An `Activity` is in the stopped state when it's no longer visible on the screen. This is usually because the user started another activity or returned to the home screen. The Android system retains the activity instance in the back stack, and if the user returns to the activity, the system restarts it. If resources are low, the system might kill a stopped activity altogether.
The system calls the `onStop()` method when the activity stops. Implement the `onStop()` method to save persistent data and release resources that you didn't already release in `onPause()`, including operations that may have been too heavyweight for `onPause()`.

Activity destroyed: the onDestroy() method

```
@Override
protected void onDestroy() {
    super.onDestroy();
    // The activity is about to be destroyed.
}
```

When your `Activity` is destroyed it is shut down completely, and the `Activity` instance is reclaimed by the system. This can happen in several cases:

- You call `finish()` in your `Activity` to manually shut it down.
- The user navigates back to the previous `Activity`.
- The device is in a low memory situation where the system reclaims any stopped `Activity` to free more resources.
- A device configuration change occurs. You learn more about configuration changes later in this chapter.

Use `onDestroy()` to fully clean up after your `Activity` so that no component (such as a thread) is running after the `Activity` is destroyed.
Note that there are situations where the system will simply kill the hosting process for the `Activity` without calling this method (or any others), so you should not rely on `onDestroy()` to save any required data or `Activity` state.
Use `onPause()` or `onStop()` instead.

Activity restarted: the onRestart() method

```
@Override
protected void onRestart() {
    super.onRestart();
    // The activity is about to be restarted.
}
```

The restarted state is a transient state that only occurs if a stopped `Activity` is started again. In this case the `onRestart()` method is called between `onStop()` and `onStart()`. If you have resources that need to be stopped or started you typically implement that behavior in `onStop()` or `onStart()` rather than `onRestart()`.

Configuration changes and Activity state

Earlier in the section on `onDestroy()` you learned that your `Activity` may be destroyed when the user navigates back, or when your code executes the `finish()` method, or when the system needs to free resources. Another way an `Activity` can be destroyed is when the device undergoes a *configuration change*.

Configuration changes occur on the device, in runtime, and invalidate the current layout or other resources in your `Activity`. The most common form of a configuration change is when the device is rotated. When the device rotates from portrait to landscape, or from landscape to portrait, the layout for your app needs to change. The system recreates the `Activity` to help that `Activity` adapt to the new configuration by loading alternative resources (such as a landscape-specific layout). Other configuration changes can include a change in locale (the user chooses a different system language), or the user enters multi-window mode (Android 7). In multi-window mode, if you have configured your app to be resizeable, Android recreates the `Activity` to use a layout definition for the new, smaller size.

When a configuration change occurs, the Android system shuts down your activity, calling `onPause()`, `onStop()`, and `onDestroy()`. Then the system restarts the activity from the beginning, calling `onCreate()`, `onStart()`, and `onResume()`.

Activity instance state

When an `Activity` is destroyed and recreated, there are implications for the runtime state of that `Activity`. When an `Activity` is paused or stopped, the state of the `Activity` is retained because that `Activity` is still held in memory. When an `Activity` is recreated, the state of the `Activity` and any user progress in that `Activity` is lost, with these exceptions:

- Some `Activity` state information is automatically saved by default. The state of `View` elements in your layout with a unique ID (as defined by the `android:id` attribute in the layout) are saved and restored when an `Activity` is recreated. In this case, the user-entered values in `EditText` elements are usually retained when the `Activity` is recreated.

- The `Intent` that was used to start the `Activity`, and the information stored in the data or extras for that `Intent`, remains available to that `Activity` when it is recreated.

The `Activity` state is stored as a set of key/value pairs in a `Bundle` object called the `Activity` *instance state* . The system saves default state information to instance state `Bundle` just before the `Activity` is stopped, and passes that `Bundle` to the new `Activity` instance to restore.
You can add your own instance data to the instance state `Bundle` by overriding the `onSaveInstanceState()` callback. The state `Bundle` is passed to the `onCreate()` method, so you can restore that instance state data when your `Activity` is created. There is also a corresponding `onRestoreInstanceState()` callback you can use to restore the state data.
Test that your `Activity` behaves correctly when the user rotates the device, because device rotation is a common use case. Implement instance state if you need to.
Note: The `Activity` instance state is particular to a specific instance of an `Activity`, running in a single task. If the user force-quits the app or reboots the device, or if the Android system shuts down the app process to preserve memory, the `Activity` instance state is lost. To keep state changes across app instances and device reboots, you need to write that data to shared preferences. You learn more about shared preferences in another chapter.

Saving Activity instance state

To save information to the instance state `Bundle`, use the `onSaveInstanceState()` callback. This is not a lifecycle callback method, but it is called when the user is leaving your `Activity` (sometime before the `onStop()` method).

```
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    super.onSaveInstanceState(savedInstanceState);
    // save your state data to the instance state bundle
}
```

The `onSaveInstanceState()` method is passed a `Bundle` object (a collection of key/value pairs) when it is called. This is the instance state `Bundle` to which you will add your own `Activity` state information.
You learned about `Bundle` in a previous chapter when you added keys and values to the `Intent` extras. Add information to the instance state `Bundle` in the same way, with keys you define and the various "put" methods defined in the `Bundle` class:

```
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    super.onSaveInstanceState(savedInstanceState);

    // Save the user's current game state
    savedInstanceState.putInt("score", mCurrentScore);
    savedInstanceState.putInt("level", mCurrentLevel);
}
```

Don't forget to call through to the superclass, to make sure the state of the `View` hierarchy is also saved to the `Bundle`.

Restoring Activity instance state

Once you've saved the `Activity` instance state, you also need to restore it when the `Activity` is recreated. You can do this one of two places:

- The `onCreate()` callback method, which is called with the instance state `Bundle` when the `Activity` is created.
- The `onRestoreInstanceState()` callback, which is called after `onStart()` after the `Activity` is created.

Most of the time the better place to restore the `Activity` state is in `onCreate()`, to ensure that your UI, including the state, is available as soon as possible.
To restore the saved instances state in `onCreate()`, test for the existence of a state `Bundle` before you try to get data out of it. When your `Activity` is started for the first time there will be no state and the `Bundle` will be `null`.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    // Always call the superclass first
    super.onCreate(savedInstanceState);

    // Check if recreating a previously destroyed instance.
    if (savedInstanceState != null) {
        // Restore value of members from saved state.
        mCurrentScore = savedInstanceState.getInt("score");
        mCurrentLevel = savedInstanceState.getInt("level");
    } else {
        // Initialize members with default values for a new instance.
        // ...
    }
    // ... Rest of code
}
```

Related practical

The related practical is 2.2: Activity lifecycle and state.

Learn more

Android Studio documentation:

- Meet Android Studio

Android developer documentation:

- Application Fundamentals
- Activities
- Understand the Activity Lifecycle
- Intents and Intent Filters
- Handle configuration changes
- `Activity`
- `Intent`
- `Bundle`

2.3: Implicit intents

Contents:

Introduction

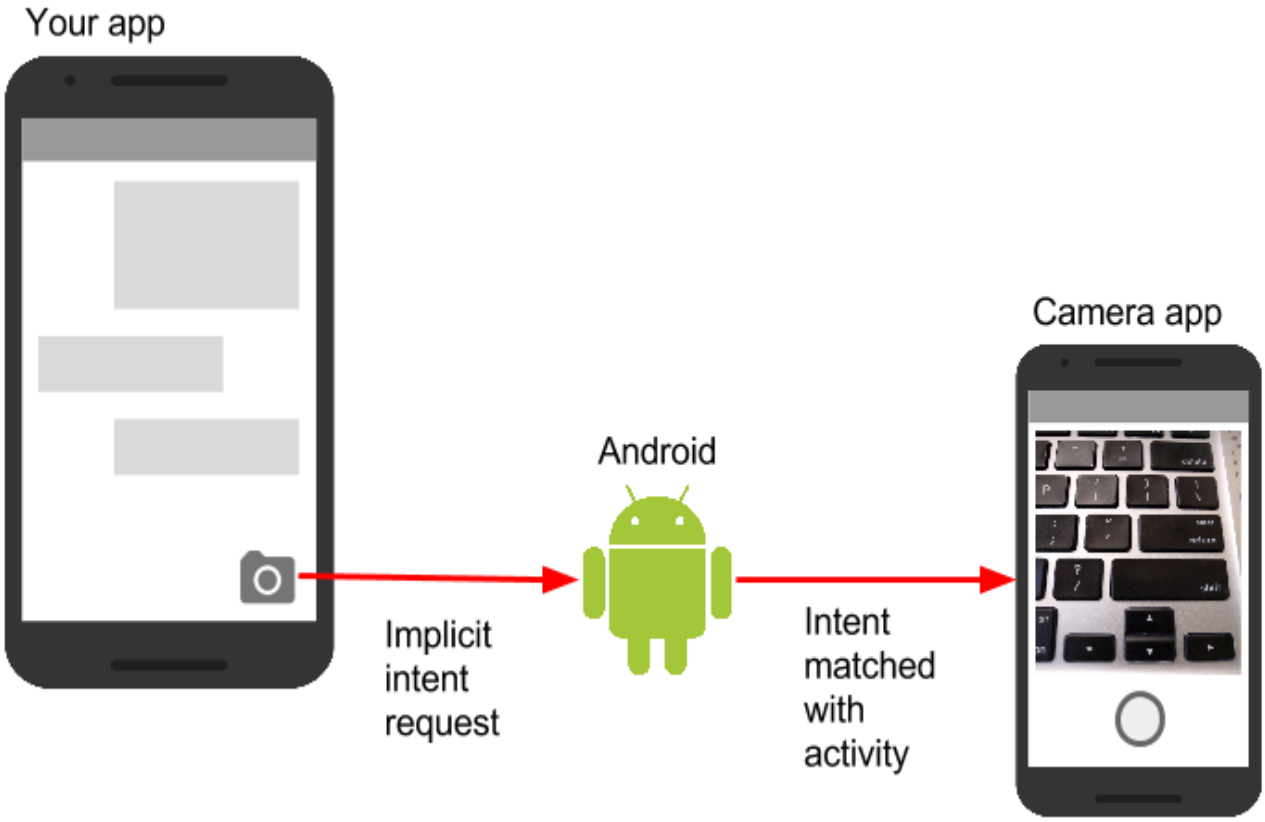
In a previous chapter you learned how to launch a specific activity in your app with an explicit intent. In this chapter you learn how to send and receive an *implicit* intent. In an implicit intent, you declare a general action to perform, and the system matches your request with an activity.

You also learn more about Android *tasks*, and how you can configure your apps to associate new activities with different tasks.

Understanding an implicit Intent

In an earlier chapter you learned how to start an activity from an activity by specifying the class name of the activity to start in an *explic*it intent. This is the most basic way to use an intent: To start an `Intent` or other app component and pass data to it (and sometimes receive data).
A more flexible use of an `Intent` is the *implicit* intent. You don't specify the exact activity (or other component) to run—instead, you include just enough information in the intent about the task you want to perform. The Android system matches the information in your request intent with any activity available on the device that can perform that task. If there's only one activity that matches, that activity is launched. If more than one activity matches the intent, the user is presented with an app chooser

and picks which app they would like to perform the task.



For example, you have an app that lists available snippets of video. If the user touches an item in the list, you want to play that video snippet. Rather than implementing an entire video player in your own app, you can launch an `Intent` that specifies the task as "play an object of type video." The Android system then matches your request with an `Activity` that has registered itself to play objects of type video. An `Activity` registers itself with the system as being able to handle an implicit `Intent` with `Intent` *filters*, declared in the `AndroidManifest.xml` file. For example, the main `Activity` (and only the main `Activity`) for your app has an `Intent` filter that declares it the main `Activity` for the launcher category. This `Intent` filter is how the Android system knows to start that specific `Activity` in your app when the user taps the icon for your app on the device home screen.

Intent actions, categories, and data

An implicit `Intent`, like an explicit `Intent`, is an instance of the `Intent` class. In addition to the parts of an `Intent` you learned about in an earlier chapter (such as the `Intent` data and extras), these fields are used by an implicit `Intent`:

- The `Intent` *action*, which is the generic action the receiving `Activity` should perform. The available `Intent` actions are defined as constants in the `Intent` class and begin with the word `ACTION_`. A common `Intent` action is `ACTION_VIEW`, which you use when you have some information that an `Activity` can show to the user, such as a photo to view in a gallery app, or an address to view in a map app. You can specify the action for an `Intent` in the `Intent` constructor, or with the `setAction()` method.
- An `Intent` *category*, which provides additional information about the category of component that should handle the `Intent`. `Intent` categories are optional, and you can add more than one category to an `Intent`. `Intent` categories are also defined as constants in the `Intent` class and begin with the word `CATEGORY_`. You can add categories to the `Intent` with the `addCategory()` method.
- The data *type*, which indicates the MIME type of data the `Activity` should operate on. Usually, the data type is inferred from the URI in the `Intent` data field, but you can also explicitly define the data type with the `setType()` method.

`Intent` actions, categories, and data types are used both by the `Intent` object you create in your sending `Activity`, as well as in the `Intent` filters you define in the `AndroidManifest.xml` file for the receiving `Activity`. The Android system uses this information to match an implicit `Intent` request with an `Activity` or other component that can handle that `Intent`.

Sending an implicit Intent

Starting an `Activity` with an implicit `Intent`, and passing data from one `Activity` to another, works much the same way as it does for an explicit `Intent`:

1. In the sending `Activity`, create a new `Intent` object.
2. Add information about the request to the `Intent` object, such as data or extras.
3. Send the `Intent` with `startActivity()` (to just start the `Activity`) or `startActivityforResult()` (to start the `Activity` and expect a result back).

When you create an implicit `Intent` object, you:

- Do *not* specify the specific `Activity` or other component to launch.
- Add an `Intent` action or `Intent` categories (or both).
- Resolve the `Intent` with the system before calling `startActivity()` or `startActivityforResult()`.
- Show an app chooser for the request (optional).

Create implicit Intent objects

To use an implicit `Intent`, create an `Intent` object as you did for an explicit `Intent`, only without the specific component name.

```
Intent sendIntent = new Intent();
```

You can also create the `Intent` object with a specific action:

```
Intent sendIntent = new Intent(Intent.ACTION_VIEW);
```

Once you have an `Intent` object you can add other information (category, data, extras) with the various `Intent` methods. For example, this code creates an implicit `Intent` object, sets the `Intent` action to `ACTION_SEND`, defines an `Intent` extra to hold the text, and sets the type of the data to the MIME type `"text/plain"`.

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType("text/plain");
```

Resolve the Activity before starting it

When you define an implicit `Intent` with a specific action and/or category, there is a possibility that there won't be *any* `Activity` on the device that can handle your request. If you just send the `Intent` and there is no appropriate match, your app will crash.

To verify that an `Activity` or other component is available to receive your `Intent`, use the `resolveActivity()` method with the system package manager like this:

```
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(chooser);
}
```

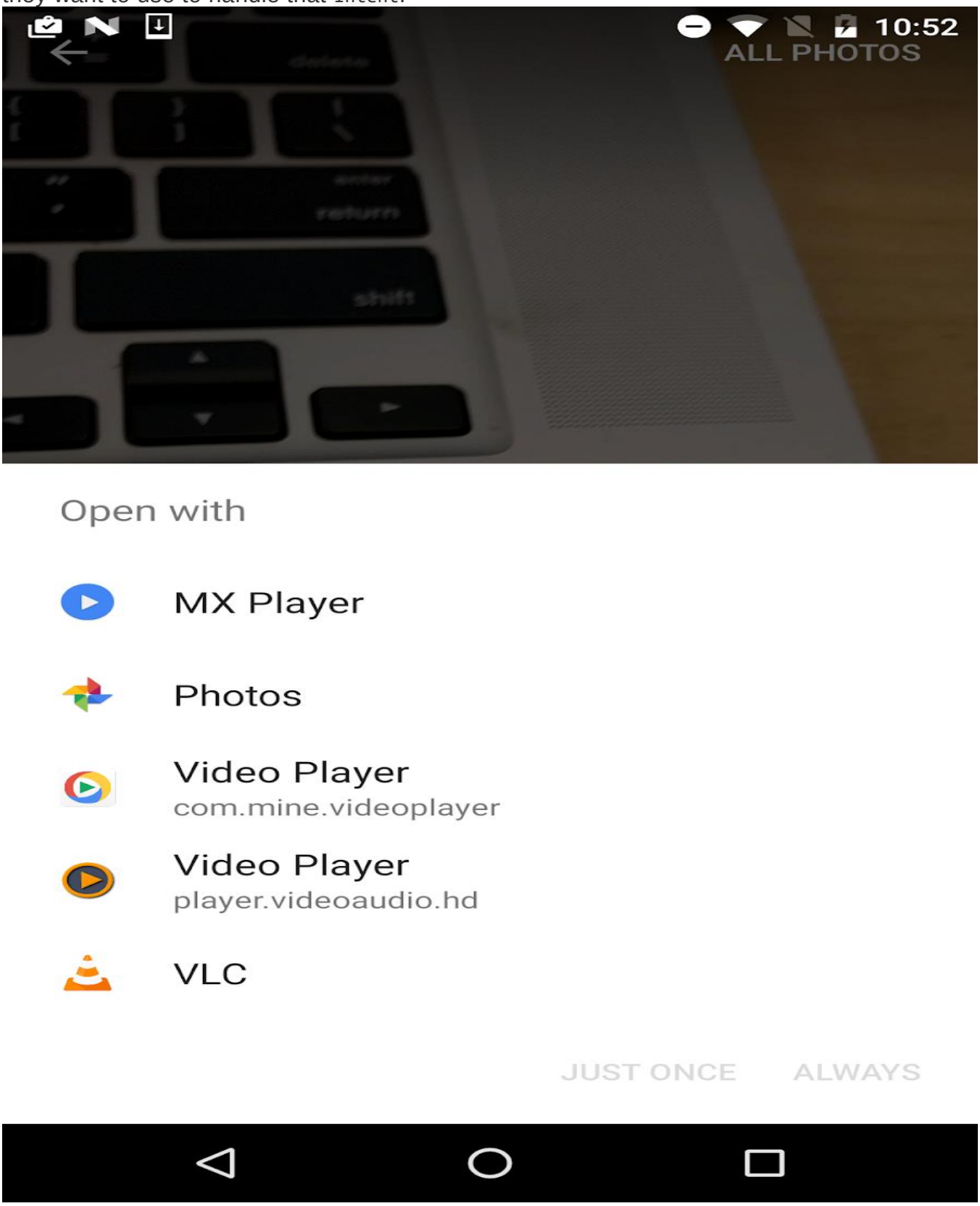
If the result of `resolveActivity()` is not `null`, then there is at least one app available that can handle the `Intent`, and it's safe to call `startActivity()`. Do not send the `Intent` if the result is `null`.

If you have a feature that depends on an external `Activity` that may or may not be available on the device, a best practice is to test for the availability of that external `Activity` before the user tries to use it. If there is no `Activity` that can handle your request (that is, `resolveActivity()` returns `null`), disable the feature or provide the user an error message for that feature.

Show the app chooser

To find an `Activity` or other component that can handle your `Intent` requests, the Android system matches your implicit `Intent` with an `Activity` whose `Intent` filters indicate that they can perform that action. If there are multiple apps installed that match, the user is presented with an app chooser that lets them select which app

they want to use to handle that `Intent`.



In many cases the user has a preferred app for a given task, and they will select the option to always use that app for that task. However, if multiple apps can respond to the `Intent` and the user might want to use a different app each time, you can choose to explicitly show a chooser dialog every time. For example, when your app performs a "share this" action with the `ACTION_SEND` action, users may want to share using a different app depending on the current situation.
To show the chooser, you create a wrapper `Intent` for your implicit `Intent` with the `createChooser()` method, and then resolve and call `startActivity()` with that wrapper `Intent`. The `createChooser()` method also requires a string argument for the title that appears on the chooser. You can specify the title with a string resource as you would any other string.
For example:

```
// The implicit Intent object
Intent sendIntent = new Intent(Intent.ACTION_SEND);
// Always use string resources for UI text.
String title = getResources().getString(R.string.chooser_title);
// Create the wrapper intent to show the chooser dialog.
Intent chooser = Intent.createChooser(sendIntent, title);
// Resolve the intent before starting the activity
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(chooser);
}
```

Receiving an implicit Intent

If you want an `Activity` in your app to respond to an implicit `Intent` (from your own app or other apps), declare one or more `Intent` filters in the `AndroidManifest.xml` file. Each `Intent` filter specifies the type of `Intent` it accepts based on the action, data, and category for the `Intent`. The system will deliver an implicit `Intent` to your app component only if that `Intent` can pass through one of your `Intent` filters.
Note: An explicit `Intent` is always delivered to its target, regardless of any `Intent` filters the component declares. Conversely, if an `Activity` does not include `Intent` filters, it can only be launched with an explicit `Intent`.

Once your `Activity` is successfully launched with an implicit `Intent`, you can handle that `Intent` and its data the same way you did an explicit `Intent`, by:

1. Getting the `Intent` object with `getIntent()`.
2. Getting `Intent` data or extras out of that `Intent`.
3. Performing the task the `Intent` requested.
4. Returning data to the calling `Activity` with another `Intent`, if needed.

Intent filters

Define `Intent` filters with one or more `<intent-filter>` elements in the `AndroidManifest.xml` file, nested in the corresponding `<activity>` element. Inside `<intent-filter>`, specify the type of intent your activity can handle. The Android system matches an implicit intent with an activity or other app component only if the fields in the `Intent` object match the `Intent` filters for that component.
An `Intent` filter may contain the following elements, which correspond to the fields in the `Intent` object described above:

- `<action>`: The `Intent` action that the activity accepts.
- `<data>`: The type of data accepted, including the MIME type or other attributes of the data URI (such as scheme, host, port, and path).
- `<category>`: The `Intent` category.

For example, the main `Activity` for your app includes this `<intent-filter>` element, which you saw in an earlier chapter:

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

This `Intent` filter has the action `MAIN` and the category `LAUNCHER`. The `<action>` element specifies that this is the app's "main" entry point. The `<category>` element specifies that this activity should be listed in the system's app launcher (to allow users to launch the activity). Only the main activity for your app should have this `Intent` filter.
Here's another example for an implicit `Intent` that shares a bit of text. This `Intent` filter matches the implicit `Intent` example from the previous section:

```
<activity android:name="ShareActivity">
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
</activity>
```

You can specify more than one action, data, or category for the same `Intent` filter, or have multiple `Intent` filters per `Activity` to handle each different kind of `Intent`.
The Android system tests an implicit `Intent` against an `Intent` filter by comparing the parts of that `Intent` to each of the three `Intent` filter elements (action, category, and data). The `Intent` must pass all three tests or the Android system won't deliver the `Intent` to the component. However, because a component may have multiple `Intent` filters, an `Intent` that does not pass through one of a component's filters might make it through on another filter.

Actions

An `Intent` filter can declare zero or more `<action>` elements for the `Intent` action. The action is defined in the name attribute, and consists of the string `"android.intent.action."` plus the name of the `Intent` action, minus the `ACTION_` prefix. So, for example, an implicit `Intent` with the action `ACTION_VIEW` matches an `Intent` filter whose action is `android.intent.action.VIEW`.
For example, this `Intent` filter matches either `ACTION_EDIT` and `ACTION_VIEW`:

```
<intent-filter>
    <action android:name="android.intent.action.EDIT" />
    <action android:name="android.intent.action.VIEW" />
</intent-filter>
```

To get through this filter, the action specified in the incoming `Intent` object must match at least one of the actions. You must include at least one `Intent` action for an incoming implicit `Intent` to match.

Categories

An `Intent` filter can declare zero or more `<category>` elements for `Intent` categories. The category is defined in the name attribute, and consists of the string `"android.intent.category."` plus the name of the `Intent` category, minus the `CATEGORY` prefix.
For example, this `Intent` filter matches either `CATEGORY_DEFAULT` and `CATEGORY_BROWSABLE`:

```
<intent-filter>
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
</intent-filter>
```

Note that any `Activity` that you want to accept an implicit `Intent` must include the `android.intent.category.DEFAULT` `Intent` filter. This category is applied to all implicit `Intent` objects by the Android system.

Data

An `Intent` filter can declare zero or more `<data>` elements for the URI contained in the `Intent` data. As the `Intent` data consists of a URI and (optionally) a MIME type, you can create an `Intent` filter for various aspects of that data, including:

- URI Scheme
- URI Host
- URI Path
- Mime type

For example, this `Intent` filter matches any data `Intent` with a URI scheme of `http` and a MIME type of either `"video/mpeg"` or `"audio/mpeg"`.

```
<intent-filter>
    <data android:mimeType="video/mpeg" android:scheme="http" />
    <data android:mimeType="audio/mpeg" android:scheme="http" />
</intent-filter>
```

Sharing data using ShareCompat.IntentBuilder

Share actions are an easy way for users to share items in your app with social networks and other apps. Although you can build a share action in your own app using an implicit `Intent` with the `ACTION_SEND` action, Android provides the `ShareCompat.IntentBuilder` helper class to easily implement sharing in your app.
Note: For apps that target Android releases after API 14, you can use the `ShareActionProvider` class for share actions instead of `ShareCompat.IntentBuilder`. The `ShareCompat` class is part of the V4 support library, and allows you to provide share actions in apps in a backward-compatible fashion. `ShareCompat` provides a single API for sharing on both old and new Android devices. You learn more about the Android support libraries in another chapter.

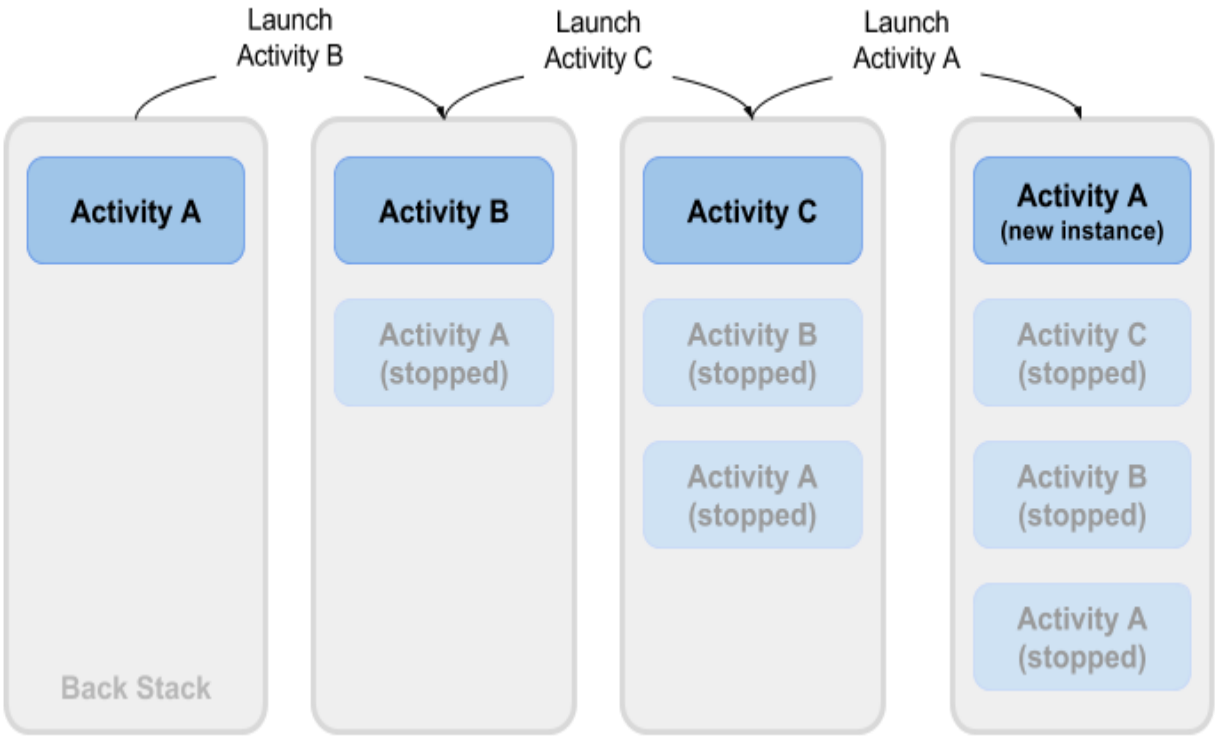
With the `ShareCompat.IntentBuilder` class you do not need to create or send an implicit `Intent` for the share action. Use the methods in `ShareCompat.IntentBuilder` to indicate the data you want to share as well as any additional information. Start with the `from()` method to create a new `Intent` builder, add other methods to add more data, and end with the `startChooser()` method to create and send the `Intent`. You can chain the methods together like this:

```
ShareCompat.IntentBuilder
    .from(this)         // information about the calling activity
    .setType(mimeType)  // mime type for the data
    .setChooserTitle("Share this text with: ") //title for the app chooser
    .setText(txt)       // intent data
    .startChooser();    // send the intent
```

Managing tasks

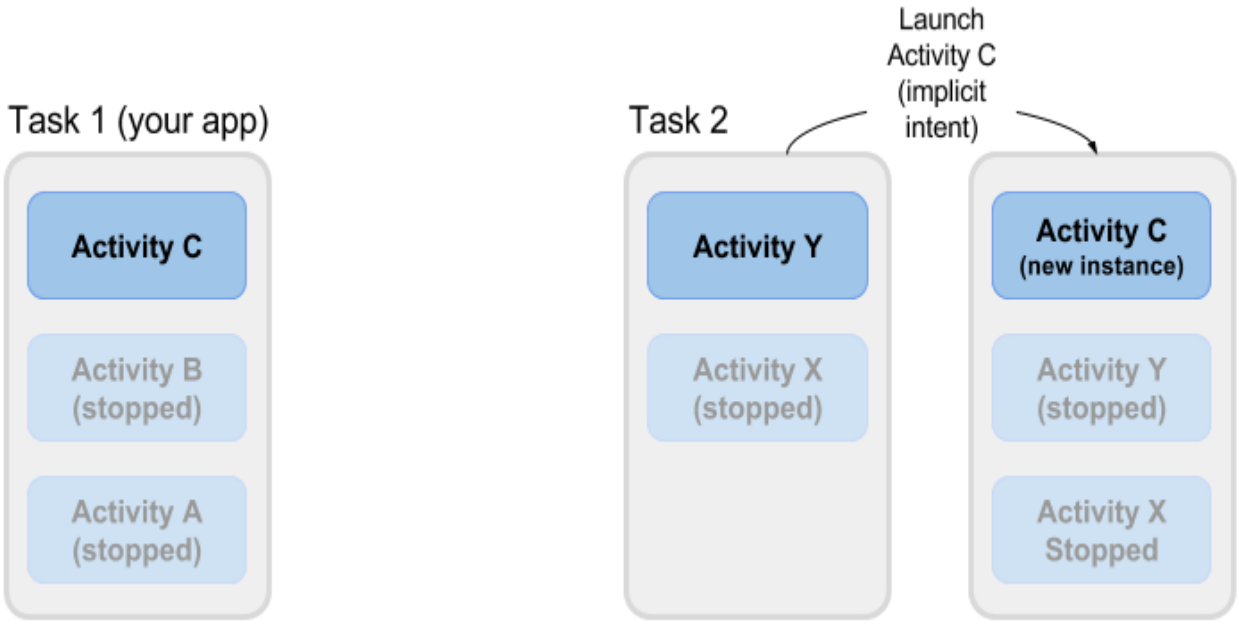
In a previous chapter you learned about tasks and the back stack. The task for your app contains its own stack that contains each `Activity` the user has visited while using your app. As the user navigates around your app, `Activity` instances for that task are pushed and popped from the stack for that task.
Most of the time the user's navigation from one `Activity` to another `Activity` and back again through the stack is straightforward. Depending on the design and navigation of your app there may be complications, especially with an `Activity` started from another app and other tasks.
For example, say you have an app with three `Activity` objects: A, B, and C. A launches B with an `Intent`, and B launches C. C, in turn sends an `Intent` to launch A. In this case the system creates a *second instance* of A on the top of the stack, rather than bringing the already-running instance to the foreground. Depending on how you

implement each `Activity`, the two instances of A can get out of sync and provide a confusing experience for a user navigating back through the stack.



Or, say your `Activity` C can be launched from a second app with an implicit `Intent`. The user runs the second app, which has its own task and its own back stack. If that app uses an implicit `Intent` to launch your `Activity` C, a new instance of C is created and placed on the back stack for that second app's task. Your app still has its own

task, its own back stack, and its own instance of C.



Much of the time the Android's default behavior for tasks works fine and you don't have to worry about how each `Activity` is associated with tasks, or how they exist in the back stack. If you want to change the normal behavior, Android provides a number of ways to manage tasks and each `Activity` within those tasks, including:

- `Activity` launch modes, to determine how an `Activity` should be launched.
- Task affinities, which indicate which task a launched `Activity` belongs to.

Activity launch modes

Use `Activity` launch modes to indicate how each new `Activity` should be treated when launched—that is, if the `Activity` should be added to the current task, or launched into a new task. Define launch modes for the `Activity` with attributes on the `<activity>` element of the `AndroidManifest.xml` file, or with flags set on the `Intent` that starts that `Activity`.

Activity attributes

To define a launch mode for an `Activity` add the `android:launchMode` attribute to the `<activity>` element in the `AndroidManifest.xml` file. This example uses a launch mode of `"standard"`, which is the default.

```
<activity
   android:name=".SecondActivity"
   android:label="@string/activity2_name"
   android:parentActivityName=".MainActivity"
   android:launchMode="standard">
   <!-- More attributes ... -->
</activity>
```

There are four launch modes available as part of the `<activity>` element:

- `"standard"` (the default): A new `Activity` is launched and added to the back stack for the current task. An `Activity` can be instantiated multiple times, a single task can have multiple instances of the same `Activity`, and multiple instances can belong to different tasks.
- `"singleTop"`: If an instance of an `Activity` exists at the top of the back stack for the current task and an `Intent` request for that `Activity` arrives, Android routes that `Intent` to the existing `Activity` instance rather than creating a new instance. A new `Activity` is still instantiated if there is an existing `Activity` anywhere in the back stack other than the top.
- `"singleTask"`: When the `Activity` is launched the system creates a new task for that `Activity`. If another task already exists with an instance of that `Activity`, the system routes the `Intent` to that `Activity` instead.
- `"singleInstance"`: Same as single task, except that the system doesn't launch any other `Activity` into the task holding the `Activity` instance. The `Activity` is always the single and only member of its task.

The vast majority of apps will only use the standard or single top launch modes. See the `android:launchMode` attribute for more detailed information on launch modes.

Intent flags

`Intent` flags are options that specify how the activity (or other app component) that receives the intent should handle that intent. `Intent` flags are defined as constants in the `Intent` class and begin with the word `FLAG_`. You add `Intent` flags to an `Intent` object with `setFlag()` or `addFlag()`.
Three specific `Intent` flags are used to control activity launch modes, either in conjunction with the `launchMode` attribute or in place of it. `Intent` flags always take precedence over the launch mode in case of conflicts.

- `FLAG_ACTIVITY_NEW_TASK`: start the `Activity` in a new task. This behavior is the same as for `singleTask` launch mode.
- `FLAG_ACTIVITY_SINGLE_TOP`: if the `Activity` to be launched is at the top of the back stack, route the `Intent` to that existing `Activity` instance. Otherwise create a new `Activity` instance. This is the same behavior as the `singleTop` launch mode.
- `FLAG_ACTIVITY_CLEAR_TOP`: If an instance of the `Activity` to be launched already exists in the back stack, destroy any other `Activity` on top of it and route the `Intent` to that existing instance. When used in conjunction with `FLAG_ACTIVITY_NEW_TASK`, this flag locates any existing instances of the `Activity` in any task and brings it to the foreground.

See the `Intent` class for more information about other available `Intent` flags.

Handle a new Intent

When the Android system routes an `Intent` to an existing `Activity` instance, the system calls the `onNewIntent()` callback method (usually just before the `onResume()` method). The `onNewIntent()` method includes an argument for the new `Intent` that was routed to the `Activity`. Override the `onNewIntent()` method in your class to handle the information from that new `Intent`.
Note that the `getIntent()` method—to get access to the `Intent` that launched the `Activity`—*always* retains the original `Intent` that launched the `Activity` instance.
Call `setIntent()` in the `onNewIntent()` method:

```
@Override
public void onNewIntent(Intent intent) {
    super.onNewIntent(intent);
    // Use the new intent, not the original one
    setIntent(intent);
}
```

Any call to `getIntent()` after this returns the new `Intent`.

Task affinities

Task affinities indicate which task an `Activity` prefers to belong to when that `Activity` instance is launched. By default each `Activity` belongs to the app that launched it. An `Activity` from outside an app launched with an implicit `Intent` belongs to the app that sent the implicit `Intent`.
To define a task affinity, add the `android:taskAffinity` attribute to the `<activity>` element in the `AndroidManifest.xml` file. The default task affinity is the package name for the app (declared in `<manifest>`). The new task name should be unique and different from the package name. This example uses `"com.example.android.myapp.newtask"` for the affinity name.

```
<activity
   android:name=".SecondActivity"
   android:label="@string/activity2_name"
   android:parentActivityName=".MainActivity"
   android:taskAffinity="com.example.android.myapp.newtask">
   <!-- More attributes ... -->
</activity>
```

Task affinities are often used with the `singleTask` launch mode or the `FLAG_ACTIVITY_NEW_TASK` `Intent` flag to place a new `Activity` in its own named task. If the new task already exists, the `Intent` is routed to that task and that affinity.
Another use of task affinities is reparenting, which enables a task to move from the `Activity` in which it was launched to the `Activity` it has an affinity for. To enable task reparenting, add a task affinity attribute to the `<activity>` element and set `android:allowTaskReparenting` to true.

```
<activity
   android:name=".SecondActivity"
   android:label="@string/activity2_name"
   android:parentActivityName=".MainActivity"
   android:taskAffinity="com.example.android.myapp.newtask"
   android:allowTaskReparenting="true" >
   <!-- More attributes ... -->
</activity>
```

Related practical

The related practical is 2.3: Implicit intents.

Learn more

Android Studio documentation:

- Meet Android Studio

Android developer documentation:

- Application Fundamentals
- Activities
- Understand the Activity Lifecycle
- Intents and Intent Filters
- Handle configuration changes
- Allowing Other Apps to Start Your Activity
- Understand Tasks and Back Stack
- `Activity`
- `Intent`
- `<intent-filter>`
- `<activity>`
- `Uri`
- `ShareCompat.IntentBuilder`

Other:

- Manipulating Android tasks and back stack
- Stack Overflow: Android Task Affinity Explanation
- The Cheese Factory: Understand Android Activity's launchMode