

2.1: Understanding Activities and Intents

Contents:

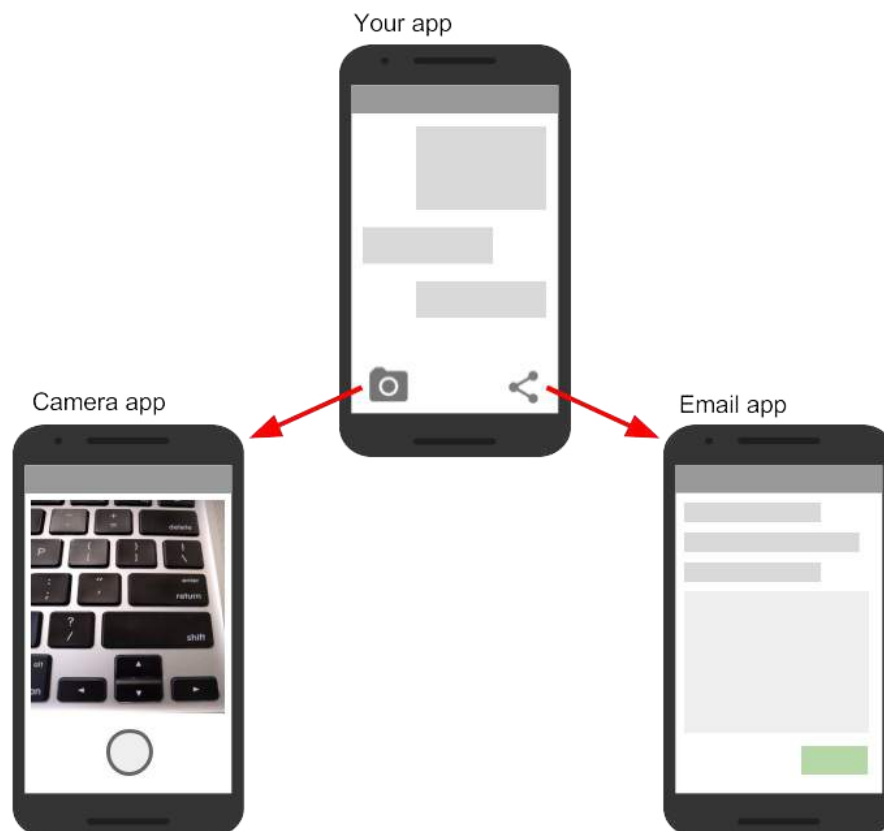
- Introduction
- About activities
- Creating activities
- About intents
- Starting an activity with an explicit intent
- Passing data between activities with intents
- Getting data back from an activity
- Activity navigation

In this chapter you'll learn about *activities*, the major building blocks of your app's user interface, as well as using *intents* to communicate between activities.

About activities

A single screen with a user-interactive interface constitutes an activity in your app. For instance, an email app might offer three distinct features: the ability to write emails, read individual messages, and display a list of new emails. Your app consists of numerous tasks that you either create yourself or steal from other apps.

Each activity in your app is independent of the others even though they all work together to create a seamless user experience. As a result, if your app permits it, other apps can start your activities as well as those in other apps. For instance, the messaging app you create might initiate a photo-taking activity in the camera app, followed by an email-sharing activity in the email app



A typical app designates one activity as the "main" activity, which is the one that the user sees when they first open the app. Then, each activity can launch additional activities to carry out various tasks.

The previous activity is stopped each time a new one begins, but the system keeps it in a stack (the "back stack"). When the user clicks the Back button to end the current activity, it is removed from the stack (and destroyed), and the previous activity continues.

The first activity is informed of the change by the activity's lifecycle callback methods when an activity is stopped because a new activity begins. The set of states that an activity can be in from the time it is first created through each time it is stopped or resumed and until the system destroys it is known as the activity lifecycle. The following chapter will cover the activity lifecycle in more detail..

Creating activities

To implement an activity in your app, do the following:

- Create an activity Java class.
- Implement a user interface for that activity.
- Declare that new activity in the app manifest.

When you create a new project for your app, or add a new activity to your app, in Android Studio (with File > New > Activity), template code for each of these tasks is provided for you.

Create the activity class

Activities are either one of, or a subclass of, the Activity class. Your projects' activities are by default subclasses of the AppCompatActivity class when you start an Android Studio project. With the help of the AppCompatActivity class, a subclass of Activity, you can continue to use modern Android app features like the action bar and material design older Android-powered devices should be able to use your app.

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);  
    }  
}
```

You must first implement the common activity lifecycle callback methods (like onCreate()) in your activity subclass in order to handle state changes for your activity. Such events as when the activity is created, stopped, resumed, or destroyed are included in these state changes. In the following chapter, you'll learn more about the activity lifecycle and lifecycle callbacks.

The onCreate() method is the only callback that must be implemented in your project. When the system generates your activity, it calls this method, and all the crucial parts of your activity should be initialized here. Most notably, the activity's principal layout is created by calling setContentView() in the onCreate() method.

Typically, you utilize one or more XML layout files to define the user interface for your activity. The system builds all of the initial views from the supplied layout and adds them to your activity when the setContentView() method is called with the path to a layout file. This practice is known as inflating the layout.

onPause() method implementation in your activity class is something you might want to do frequently. Although it does not necessarily imply that the activity is being terminated, the system uses this approach as the first indicator that the user is quitting your activity. Normally, you should commit any changes that need to last past the current user session here (because the user might not return). In the following chapter, you'll learn more about onPause() and every other lifecycle callback.

You can include methods in your activity to handle additional functionality, such as user input or button clicks, in addition to

lifecycle callbacks.

Implement a user interface

the user interface for an activity is provided by a hierarchy of views, which controls a particular space within the activity's window and can respond to user interaction.

The most common way to define a user interface using views 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 user interface separately from the source code that defines the activity's behavior.

You can also create new views 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 views in Java anywhere in the view hierarchy.

Declare the activity in the manifest

Each activity in your app must be declared in the Android app manifest with the `<activity>` element, inside `<application>`. When you create a new project or add a new activity to your project in Android Studio, your manifest is created or updated to include skeleton activity declarations for each activity. Here's the declaration for the main 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 `<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 intents your activity will accept.

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

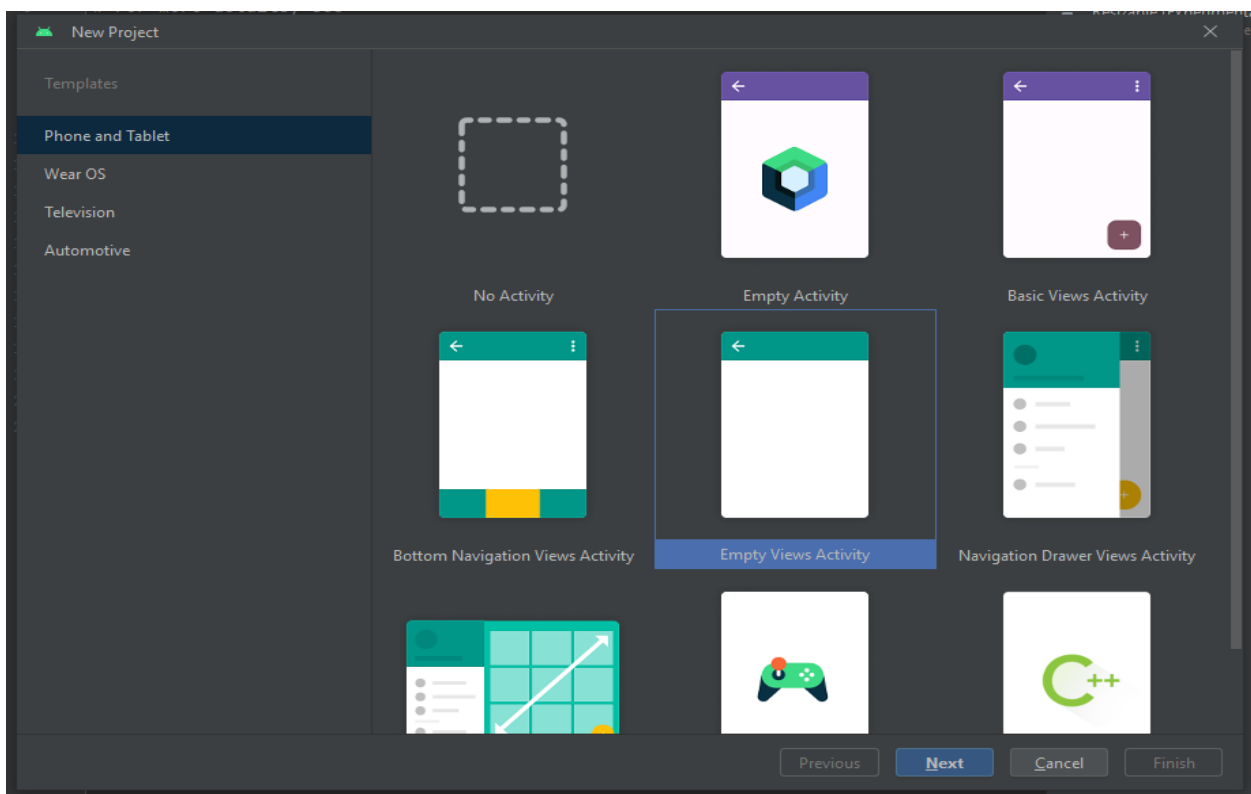
Intent filters must include at least one `<action>` element, and can also include a `<category>` and optional `<data>`. The main activity for your app needs an intent filter that defines the "main" action and the "launcher" category so that the system can launch your app. Android Studio creates this intent filter for the main activity in your project:

The `<action>` element specifies that this is the "main" entry point to the application. The `<category>` element specifies that this activity should be listed in the system's application launcher (to allow users to launch this activity).

Other activities in your app can also declare intent filters, but only your main activity should include the "main" action.. You'll learn more about implicit intents and intent filters in a later section.

Add more activities to your project

The main activity for your app and its associated layout file comes with your project when you create it. You can add new activities to your project in Android Studio with the **File > New > Activity** menu. 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'll see the same set of screens for creating the new activity that you did when you initially created the project. Android Studio provides these 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 the main activity, 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 Android manifest that specifies the new activity. The second activity definition does not include any intent filters. If you intend 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

All Android activities are started or activated with an *intent*. Intents are message objects that make a request to the Android runtime to start an activity or other app component in your app or in some other app. You don't start those activities yourself;

The Android runtime sends an intent to your app to launch its primary activity (the one specified with the MAIN action and the LAUNCHER category in the Android Manifest) when it is launched for the first time from the device's home screen. You can create your own intents with the Intent class and send them by calling the `startActivity()` function to start additional activities in your app or request that actions be taken by another activity that is currently running on the device.

Intentions are used to start actions as well as transfer data between them. You can provide the data that the new activity should operate on when you create an intent to start a new activity. As a result, an intent can be sent to an activity that shows a message from an email activity, for instance, when it displays a list of messages. You can provide the information needed for the display activity to display the message in the intent.

You will learn how to utilize intentions with activities in this chapter, but you can also use intents to launch services and broadcast receivers. Both of those app features will be covered in more detail later in the book.

Intent types

There are two types of intents in Android:

- The fully-qualified class name of the receiving activity (or other component) is specified in **explicit intents**. Use an explicit purpose to launch a component in your own app since you already know its package and class name (for example, to switch between screens in the user interface).
- **Implicit intents** do not name a particular action or other component that will be used to fulfill the intent. Instead, you state a broad action that will be carried out in the intent. The Android system matches your request to a component or activity that can carry out the task you've asked it to. In a later chapter, you'll discover more about implicit intents.

Intent objects and fields

An Intent object is an instance of the Intent class. For explicit intents, the key fields of an intent include the following:

- The activity *class* (for explicit intents). 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 intent's `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 implicit intents, you may need to also define the intent action and category. You'll learn more about intent actions and categories in section 2.3.

Starting an activity with an explicit intent

To start a specific activity from another activity, use an explicit intent and the `startActivity()` method. Explicit intents include 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 wanted 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 content (here, `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 between activities with intents

In addition to simply starting one activity from another, you also use intents to pass information between activities. 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 and intent extras to pass data between the activities. 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. Bundles are a map, optimized for Android, where the keys are strings, and the values 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's `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 several intent 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 = getData();
```

To get the extras out of the intent, you'll 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

The originating activity is halted when you start an activity with an intent, and the new activity takes its place on the screen until the user clicks the back button, you use the `finish()` method in a click handler, or you perform another action that stops the user's interaction with this activity.

When sending data to an activity with an intent, there are situations when you would also like to receive data from that intent. For instance, you might begin a photo gallery activity where the user can select a picture. In this instance, the information regarding the photo the user selected must be returned to your original activity from the one that was launched.

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'll 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'll 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

In an intent, either in the data or the extras, the response data from the launched activity back to the originating activity is sent. Similar to how you create the transmitting intent, you construct the return intent and add the data to it. The `onClick` or other user input callback method that your launched activity typically has allows you to process the user's action and end the activity. Additionally, this is where you put together your answer.

To return data from the launched activity, create a new empty intent object.

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

Note: Use a new intent object rather than reusing the previous transmitting intent object to prevent mixing up sent and returned data.

A class or component name is not necessary for return result intentions to land in the appropriate location. The Android system does this for you by rerouting the response to the original activity.

Similar to how you did with the first intent, add information or extras to the intent. At the beginning of your class, you might need to define keys for the return intent extras.

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

Then put your return data into the intent as usual. Here 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 cancelled the operation.
- RESULT_FIRST_USER. for defining your own result codes.

You'll 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 the 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 different activities 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 complicated program you create will have a number of activities that you planned, implemented, and maybe even found in other apps. Consistent navigation becomes more crucial to the app's user experience as users travel within your app and switch between activities. Few things annoy users more than straightforward navigation that acts erratically and unexpectedly. The navigation of your app should be carefully designed to give customers a predictable and dependable experience.

Android system supports two different forms of navigation strategies for your app.

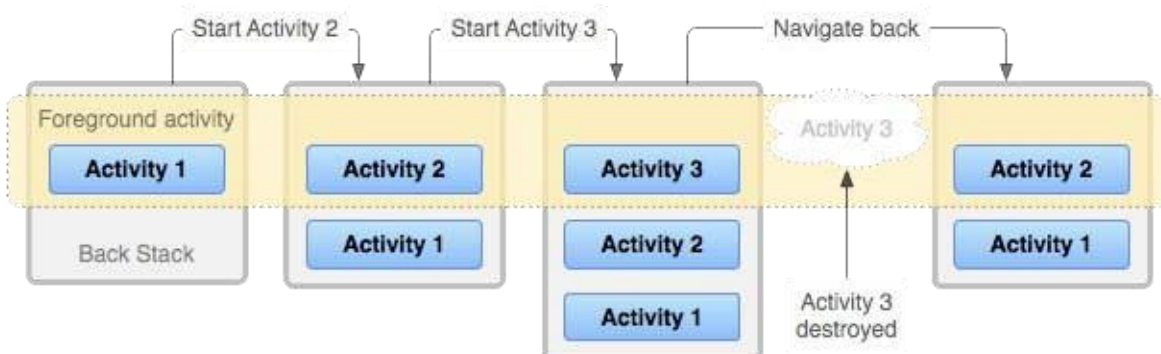
- Temporal or Back navigation, provided by the device back button, and the back stack.
- Ancestral, or Up navigation, provided by you as an option in the app's action bar.

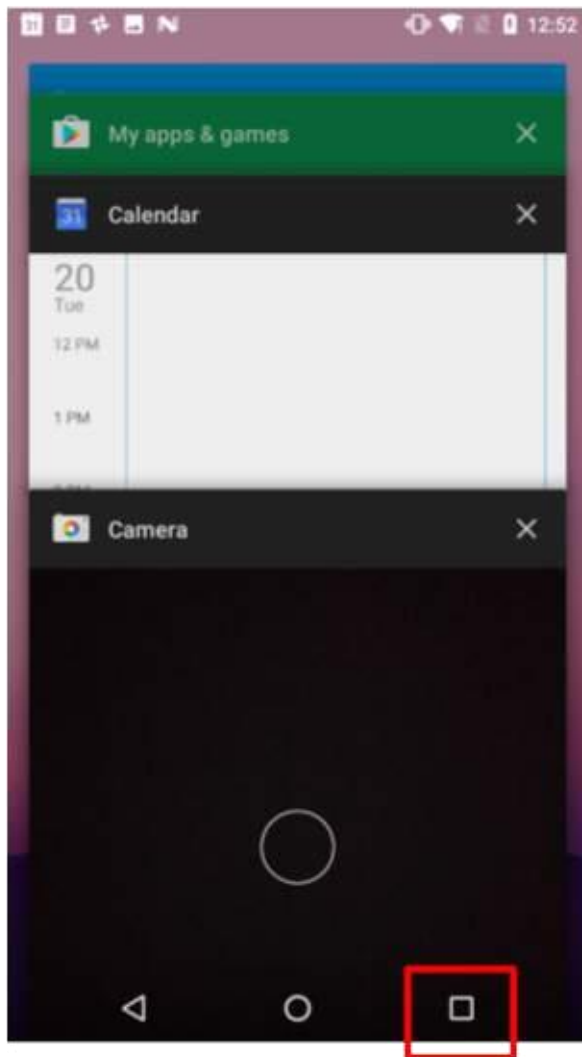
Back navigation, tasks, and the back stack

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 set of activities the user has visited and can access by pressing the back button is known as the **back stack**. When a new activity begins, it is pushed to the back stack and commands the user's attention. Although the prior activity has been halted, it is still present in the back stack. When a user finishes an activity and pushes the Back button, that activity is removed from the stack (and destroyed) and the prior activity picks up where it left off. This is known as the "last in, first out" process used by the back stack.

The back stack contains all of the user-initiated actions in reverse order because an app might start activities both within and outside of another app. Until the user returns to the Home screen, each activity in the stack is revealed when the Back button is pressed, revealing the previous one.






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 activities deep within your app may be launched as their own tasks, with no back stack at all. You'll 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, your activities are arranged in a hierarchy, and

"child" activities show a left-facing arrow in the action bar  that returns the user to the "parent" activity. The topmost activity in the hierarchy is usually your main activity, and the user cannot go up from there.

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, your activities are arranged in a hierarchy, and

"child" activities show a left-facing arrow in the action bar  that returns the user to the "parent" activity. The topmost activity in the hierarchy is usually your main activity, 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 the activities in your app.

Implement up navigation with parent activities

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 that activity's parent in the Android Manifest.

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.

Here are the skeleton definitions for both a main (parent) activity and a second (child) activity:

```
<application ... >
  <!-- The main/home 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>
  <!-- A child of the main activity -->
  <activity android:name=".SecondActivity"
    android:label="@string/activity2_name"
    android:parentActivityName=".MainActivity">
    <meta-data
      android:name="android.support.PARENT_ACTIVITY"
      android:value="com.example.android.twoactivities.MainActivity" />
    </activity>
</application>
```

2.2: The Activity Lifecycle and Managing State

Contents:

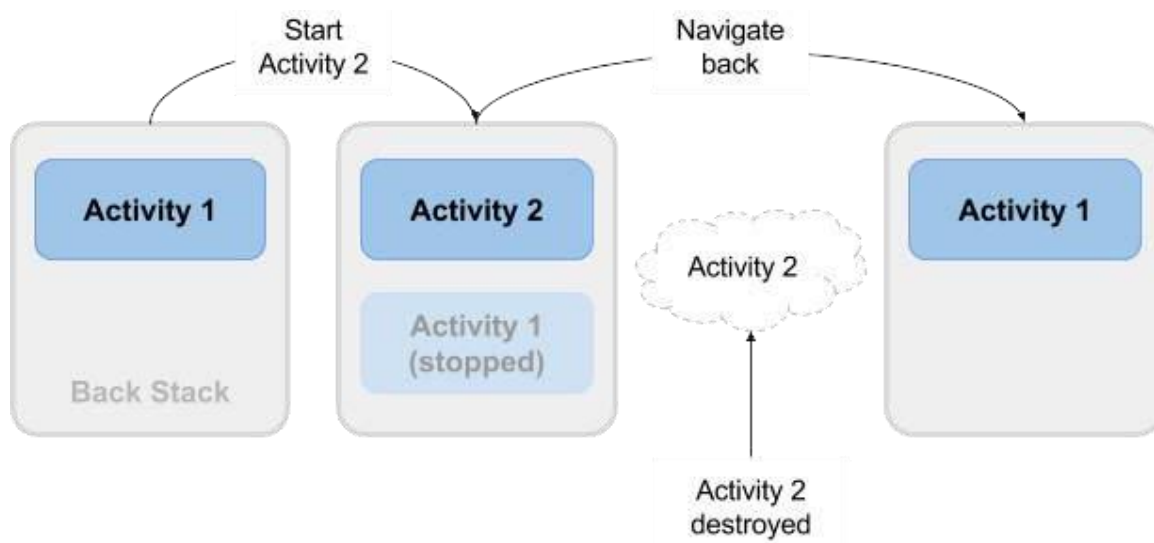
- Introduction
- About the activity lifecycle
- Activity states and lifecycle callback methods
- Configuration changes and activity state

In this chapter you'll 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 set of states that an activity can be in from the moment it is first formed until it is killed and the system reclaims its resources is known as the activity lifecycle. The many activities shift into various states when the user interacts with your app and other apps on the device.

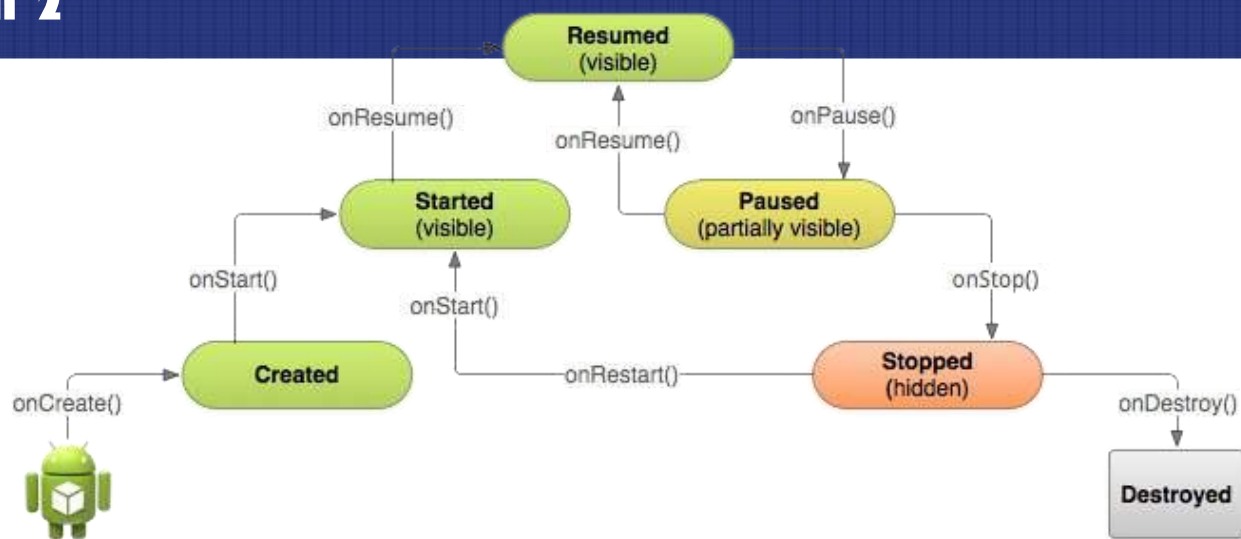
For instance, when you launch an app, Activity 1—the app's primary activity—begins, moves to the front of the screen, and gets the user's attention. When you start a second activity (Activity 2), the primary activity is halted and Activity 2 is likewise generated and started. The first action resumes after you finish the second activity and navigate back. The second activity ends and is no longer required; if the user does not pick it back up again, the system eventually destroys it.



Activity states and lifecycle callback methods

The Android system calls a number of lifecycle callback methods at each stage as an activity moves between the various lifecycle phases while it runs. Each of the callback methods can be overridden in each of your Activity classes to specify how that activity should operate when the user exits and re-enters it. Remember that the lifecycle states (and callbacks) are per activity, not per app, and thus you can implement various behaviors for various activities inside your app at various lifecycle points.

The callback methods that are used as the activity switches between various states are shown in this figure along with each activity state:



You may not need to include all of the lifecycle callback methods in your activities, depending on their complexity. However, it's crucial that you comprehend each one and put it into practice in order to make sure that your app behaves as customers would anticipate. To build a robust and adaptable application, callback methods must be used to manage the lifespan of your operations.

Activity created (onCreate() method)

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

When your activity is launched for the first time, it enters the created state. The system invokes the `onCreate()` method to initialize an activity when it is first created. For instance, the system executes the `onCreate()` function for the activity in your app that you've designated as the "launcher" or "main" activity when the user presses the app's icon from the Home screen to launch it. The `onCreate()` method of the main activity in this situation is comparable to the `main()` method in other programs.

The system matches your intent request with an activity and runs `onCreate()` for that new activity if your app launches another activity with an intent (explicit or implicit).

The only callback that needs be implemented in your activity class is the `onCreate()` method. You set up the user interface, assign class-scope variables, or create background tasks as part of your application's basic startup logic in the `onCreate()` method.

The state of an activity is temporary; it only lasts as long as it takes to run `onCreate()` before it transitions to the started state.

Activity started (onStart() method)

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


The system invokes the `onStart()` method once your activity is initialized using `onCreate()`, at which point it is in the started state. When a halted activity comes back to the foreground, such as when a user clicks the back or up arrows to move to a previous screen, the `onStart()` method is also triggered. While the `onStart()` method may be called numerous times throughout the activity's lifecycle as the user moves around your app, `onCreate()` is only called once when the activity is created.

The user cannot interact with an activity that is in the begun state and is visible on the screen until `onResume()` is called, the activity is running, and the activity is in the foreground.

`onStart()` is typically implemented in your activity as an alternative to the `onStop()` method. For instance, you can re-register hardware resources in the `onStart()` method if you release hardware resources (such as GPS or sensors) after the activity is terminated.

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

Activity resumed/running (`onResume()` method)

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

When your activity has been initialized, is visible on screen, and is prepared for usage, it is in the resumed state. Because the user is really engaging with your program in this state, the resumed state is also known as the running state.

The system invokes the `onResume()` method immediately following `onStart()` when an activity is launched for the first time. Every time the app resumes from a paused state, the `onResume()` method may be invoked more than once.

`onResume()` is normally only implemented as a counterpart to `onPause()`, unlike the pair-implemented `onStart()` and `onStop()` methods. For instance, if you stop any on-screen animations in the `onPause()` method, you would resume those animations in the `onResume()` method.

As long as the activity is in the foreground and the user is interacting with it, it stays in the resumed state. The activity might enter the stopped state from the resumed state.

Activity paused (`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:

- Though it is fading into the background, the action has not yet completely stopped. This is the first cue that the user is ceasing their participation in 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 passed 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 be 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 multiwindow 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 (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 is no longer visible on the screen at all. This is usually because the user has started another activity, or returned to the home screen. The system retains the activity instance in the back stack, and if the user returns to that activity it is restarted again. Stopped activities may be killed altogether by the Android system if resources are low.

The system calls the `onStop()` method when the activity stops. Implement the `onStop()` method to save any persistent data and release any remaining resources you did not already release in `onPause()`, including those operations that may have been too heavyweight for `onPause()`.

Activity destroyed (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 stopped activities to free more resources.
- A device configuration change occurs. You'll 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 activity's hosting process 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 (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 in 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 `onDestroy()` you learned that your activities may be destroyed when the user navigates back, by you with the `finish()` method, or by the system when it needs to free resources. The fourth time your activities are 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 vice versa, the layout for your app also 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 resizable, Android recreates your activities to use a layout definition for the new, smaller activity size.

When a configuration change occurs Android system shuts down your activity (calling `onPause()`, `onStop()`, and `onDestroy()`), and then starts it over again from the start (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 views 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` views are usually retained when the activity is recreated.
- The intent that was used to start the activity, and the information stored in that intent's data or extras, 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.

By replacing the `onSaveInstanceState()` callback, you can add your own instance data to the instance state bundle. When your activity is created, you may restore the instance state data because the state bundle is supplied to the `onCreate()` method. You can use the appropriate `onRestoreInstanceState()` callback to retrieve the state information as well.

Make sure you test that your activity responds to this configuration change successfully because device rotation is a frequent use case for your app. If necessary, implement instance state.

Note: The state of an activity operating in a single task is unique to that activity's individual instance. The activity instance state is lost if the user force-quits the app, the device reboots, or if the Android system terminates the app process entirely to conserve memory. You must write that information to shared preferences in order to maintain state changes across app instances and device reboots. In a subsequent chapter, you'll discover more information on shared preferences.

Saving activity instance state

Use the `onSaveInstanceState()` callback to save data to the instance state bundle. Although it is not a lifecycle callback function, this method is called just before the `onStop()` method when the user exits your activity.

```
@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 bundles 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 user interface 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) {
    super.onCreate(savedInstanceState); // Always call the superclass first

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

2.3: Activities and Implicit Intents

Contents:

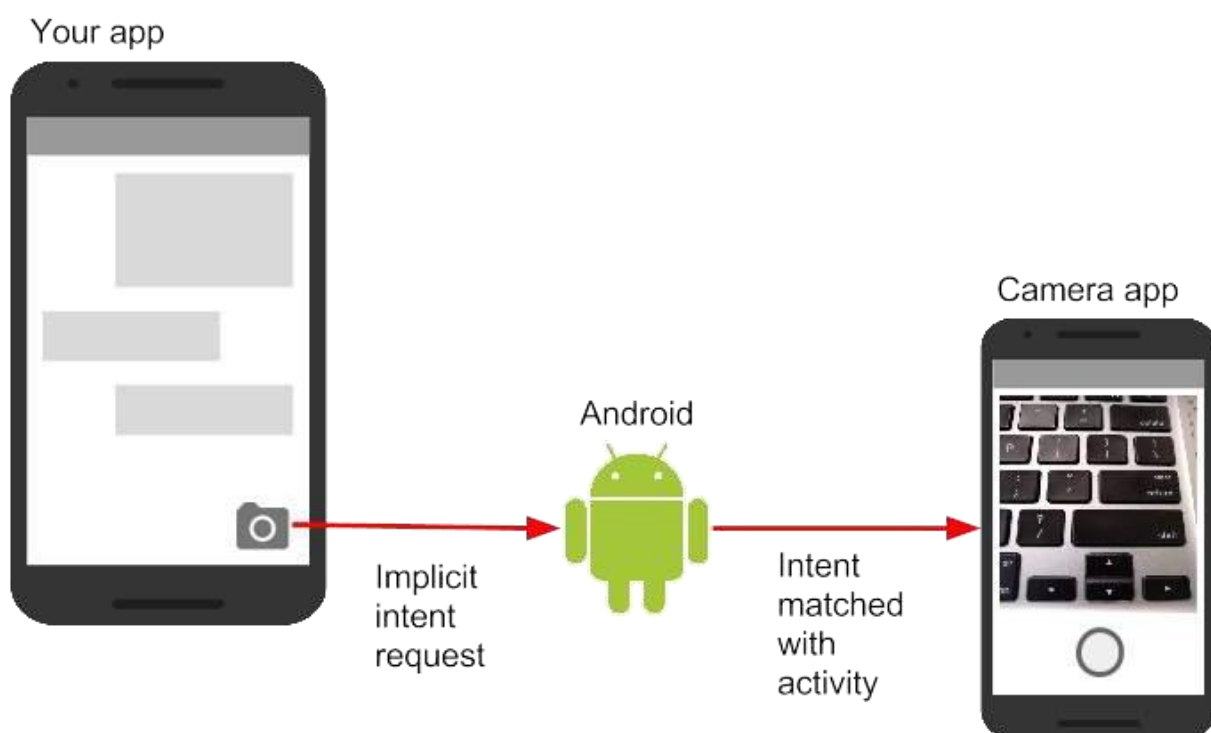
- Introduction
- About implicit intents
- Sending implicit intents
- Receiving implicit intents
- Sharing data with `ShareCompat.IntentBuilder`
- Managing tasks and activities
- Activity Launch Modes
- Task Affinities

You learned about intents and how to use them to launch specified app operations in a previous chapter. You will learn how to send and receive implicit intents in this chapter. In an implicit intent, you define a broad action to be carried out, and the system matches your request with a particular activity. You'll also learn more about Android tasks and how to set up your apps so that they may link new activities to various tasks.

About implicit intents

In an earlier chapter you learned about explicit intents, where you can start one activity from another by specifying the class name of that activity. This is the most basic way to use intents, to start an activity or other app component and pass data to it (and sometimes pass data back.)

A more flexible use of intents is the *implicit intent*. With implicit intents you do not 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 activities available on the device that can perform that task. If there's only one activity that matches, that activity is launched. If there are multiple matching activities, the user is presented with an app chooser that enables them to pick 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.

Activities register themselves with the system as being able to handle implicit intents with intent *filters*, declared in the Android manifest. 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

Implicit intents, like explicit intents, are instances 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 intent extras), these fields are used by implicit intents:

- 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, this 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 Android manifest 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 implicit intents

Starting activities with implicit intents, and passing data between those activities, works much the same way as it does for explicit intents:

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* activities 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:

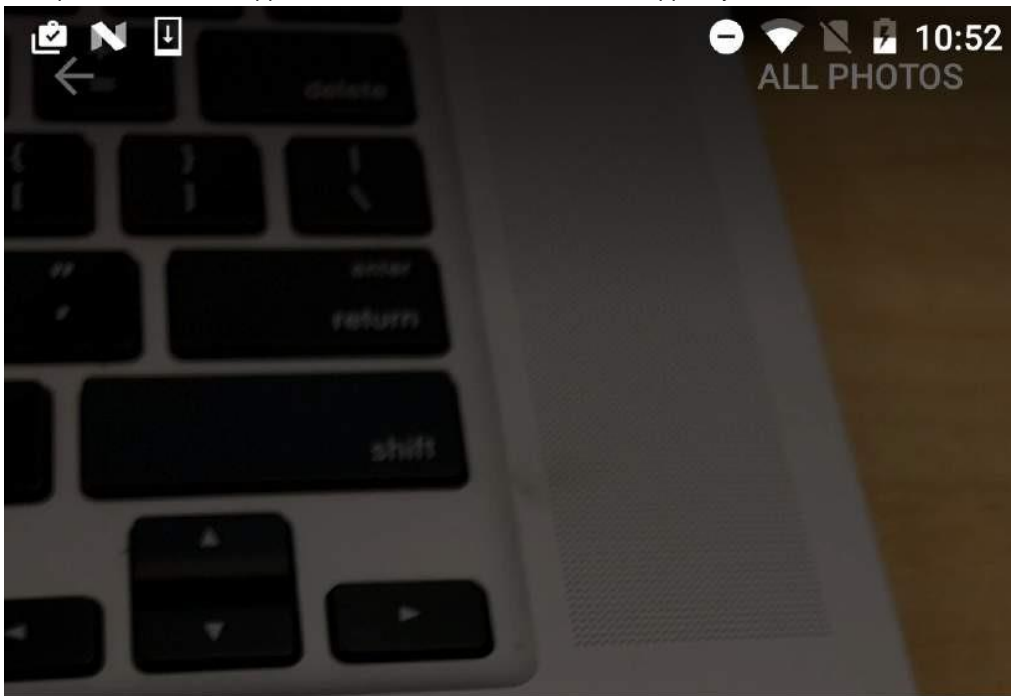
```
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(chooser);
}
```

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.



Open with

-  MX Player
-  Photos
-  Video Player
com.mine.videoplayer
-  Video Player
player.videoaudio.hd
-  VLC
-

JUST ONCE ALWAYS



The user will frequently choose the option to always utilize their preferred app for a certain job if they have one. However, you can decide to explicitly display a chooser dialog each time if numerous apps can react to the intent and the user would wish to select a different app each 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 implicit intents

Declare one or more intent filters in the Android manifest if you want an activity in your app to respond to implicit intents (from your own app or from other apps). Based on the action, data, and category of the intent, each intent filter specifies the kinds of intentions it accepts. Only if the implicit intent can get past one of your intent filters will the system give it to your app component.

Note: No matter what intent filters the component declares, an explicit intent is always sent to its recipient. On the other hand, if your activities do not specify any intent filters, then only an explicit intent can be used to start them.

After successfully launching your action with an implicit intent, you may handle that intent and its data in the same manner that you did with an explicit intent by:

Getting the Intent object with `getIntent()`.

1. Getting intent data or extras out of that intent.
2. Performing the task the intent requested.
3. 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 app's manifest file, nested in the corresponding `<activity>` element. Inside `<intent-filter>`, specify the type of intents 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 these elements, which correspond to the fields in the Intent object described above:

- `<action>` : The intent action.
- `<data>` : The type of data accepted, including the MIME type or other attributes of the data URI (such as scheme, host, port, path, and so on).
- `<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 "main" entry point to the application. The `<category>` element specifies that this activity should be listed in the system's application launcher (to allow users to launch this activity). Only the main activity for your app should have this intent filter.

Here's another example for an implicit intent to share 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 different kinds of intents.

The Android system compares the components of an implicit intent to each of the three intent filter elements (action, category, and data) before running it against the intent filter. The Android system will not provide the intent to the component if the intent fails any one of the three conditions. An intent that is rejected by one of a component's intent filters might be accepted by another filter since a component may have several intent filters.

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 all activities that you want to accept implicit intents 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 data intents 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 with `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 implicit intents with the `ACTION_SEND` action, Android provides the `ShareCompat.IntentBuilder` helper class to easily implement sharing in your app.

Note: Use `ShareActionProvider` instead of `ShareCompat.IntentBuilder` for share actions in apps that target Android releases after API 14. The `ShareCompat` class, which is a component of the V4 support library, enables you to offer backward-compatible sharing actions in apps. A single API is offered by `ShareCompat` for sharing across both old and new Android devices. In a later chapter, you'll discover more information about the Android support libraries.

You do not need to construct or submit an implicit intent for the share action when using the `ShareCompat.IntentBuilder` class. To identify the data you want to transmit as well as any other information, use the methods in `ShareCompat.IntentBuilder`. Start by creating a new intent builder using the `from()` method, then add other methods to add more data before finishing with the `startChooser()` method to construct and deliver the intent. The approaches can be linked together as follows:

```
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
```

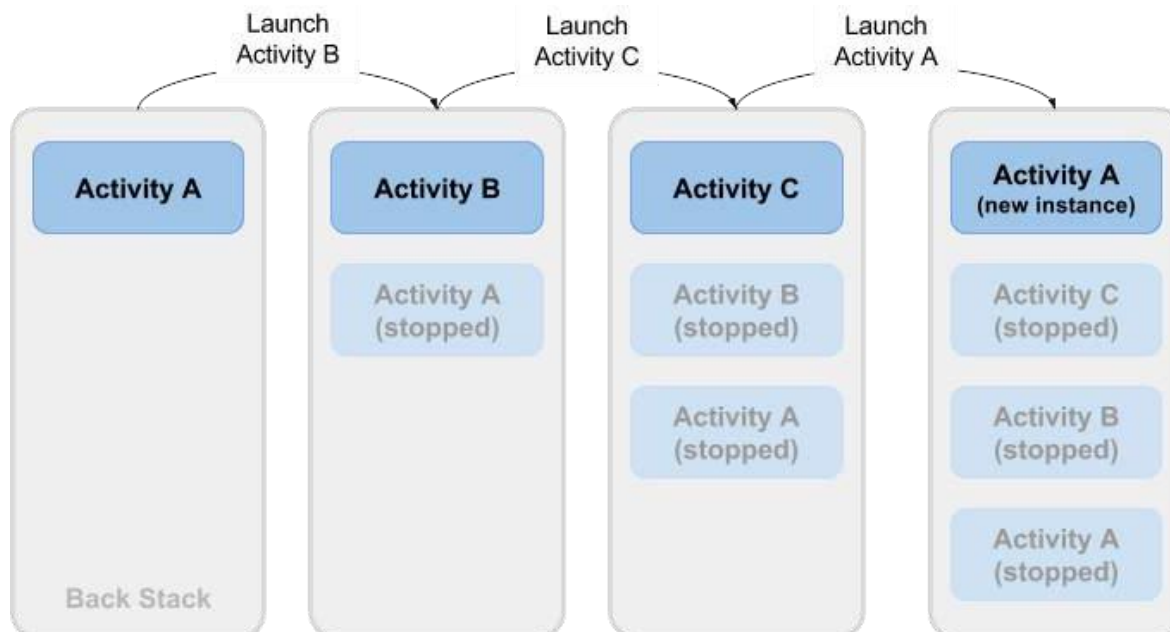
Managing tasks and activities

A task for your app contains its own stack for the activities the user has visited while using your app, as you learned about tasks and the back stack in a previous chapter. Activity objects for that task are pushed and popped from the stack as the user moves about your program.

The user's movement through the stack from one activity to the next and back again is typically simple. There can be issues depending on the layout and navigation of your app, particularly with tasks and activities that are initiated from other apps.

Let's take an app with three activities as an example: A, B, and C. B launches C after being intentionally launched by A. C then signals to A that it wants to launch. Instead of bringing the instance that is now running to the foreground in this situation, the system creates a second instance of A at the top of the stack. Depending on how you implement your activities, the user

may encounter confusion as they traverse back through the stack if the two instances of A become out of sync.



Or, let's imagine that your activity C can be started with an implicit intent from a different app. The second app, which has its own job and back stack, is run by the user. A new instance of your activity C is created and added to the back stack for the job of the second app if it utilizes an implicit intent to launch it. Your application still has a separate job, back stack, and C instance.



The default behavior of tasks and activities on Android generally works as intended, so you usually don't need to worry about how your activities are connected to tasks or how they are organized in the back stack. Android offers a variety of ways to handle tasks and the activities contained within those tasks if you want to alter the default behavior, 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 new activities should be treated when they're launched—that is, if they 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 Android manifest, 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 Android manifest. 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">
    ...
</activity>
```

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

- "standard" (the default): New activities are 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 activities 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 `launchMode` attribute documentation 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 is the same behavior as the `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 activities 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 new intents

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

When an activity instance is launched, task affinities show which task it wishes to be a part of. By default, each activity is the property of the app that initiated it. Activities from a different app that is launched with an implied intent are the responsibility of the sending app.

To define a task affinity, add the `android:taskAffinity` attribute to the `<activity>` element in the Android manifest. The default task affinity is the package name for the app (declared in `AndroidManifest.xml`). The new task name should be unique and different from the package name. This example uses `"com.example.android.myapplication.newtask"` for the affinity name.

```
<activity
    android:name=".SecondActivity"
    android:label="@string/activity2_name"
    android:parentActivityName=".MainActivity"
    android:taskAffinity="com.example.android.myapplication.newtask">
    ...
</activity>
```

Task affinities are often used in conjunction 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.myapplication.newtask"
    android:allowTaskReparenting="true" >
    ...
</activity>
```