Android Developer Fundamentals Course

Learn to develop Android Applications

Practical Workbook

Developed by Google Developer Training Team

December 2016

Table of Contents

Android Apps – Phone Calls and SMS

The Android Apps – Phone Calls and SMS course includes two lessons created by the Google Developer Training team. The lessons are designed to be used in instructor-led training, but the materials are also available online for self-study.

Prerequisites

- Completion of the entire Android Developer Fundamentals course, or equivalent knowledge
- Java programming experience
- Knowledge of Android programming fundamentals

Course materials

The course materials include the conceptual lessons and practical exercises in this GitBook. Slide decks are also available for optional use by instructors.

What do the lessons cover?

In Android Phone Calls, students learn how to use the telephony features in Android. In the practical, they create an app that launches the Phone app with a phone number to make a call, and another app that checks for needed permissions and services, and then lets the user make a phone call from within the app.

In Android SMS Messages, students learn how to use the SMS features in Android. In the practical, they create an app that launches an SMS messaging app with a message, and another app that checks for needed permissions, sends an SMS message, and receives SMS messages.

Developed by the Google Developer Training Team

Last updated: March 2017

1: Phone Calls

Contents:

Android mobile devices with telephone/cellular service are supplied with a Phone app for making calls, which includes a dialer for dialing a phone number. This chapter describes the Android telephony features you can use from within *your* app by launching the Phone app with an implicit intent. You can add code to your app to:

- Dial: Launch the Phone app's dialer with a phone number to dial a call. This is the preferred technique for apps that don't need to monitor the phone's state.
- Call: Request the user's permission if necessary, and make a phone call from within the app, with the ability to monitor the phone's state. This technique keeps the user within your app, without having to navigate back to the app. It also enables phone calls if the Phone app has been disabled in Settings.

Android's Phone app automatically receives incoming phone calls. You can use the PhoneStateListener class to monitor the phone's ringing state and show the incoming phone number.

Tip: You can also use a broadcast receiver in your app to detect an incoming call or SMS message. Broadcast receivers are described in Broadcast Receivers in the Android Developer Fundamentals Course.

The difference between dialing and calling

You use an implicit intent to launch the Phone app from your app. You can do this in two ways:

- Use an implicit Intent and `ACTION_DIAL` to launch the Phone app and display the phone number in the dialer.
  - This is the simplest way, with no need to request permission from the user (the Phone app asks for user permission if needed).

    - The user can change the phone number before dialing the call.
    - The user navigates back to your app using the **Back** button after the call is completed.
- Use an implicit Intent and `ACTION_CALL` to make the phone call directly from within your app.
    - This action keeps the user within your app, without having to navigate back from the Phone app.
    - Your code must ask the user for permission before making the call if the user hasn't already granted permission. Just as your app needs the user's permission to access the contacts or use the built-in camera, it needs the user's permission to directly use the phone.
    - Your app can monitor the state of the phone call.

Formatting a phone number

To use an intent to launch the Phone app with a phone number to dial, your app needs to prepare a Uniform Resource Identifier (URI) for the phone number. The URI is a string prefixed by "tel:", for example, `tel:14155551212` for a U.S. number (use the complete number as you would enter it on a phone keypad). You can hard-code a phone number inside your app, or provide an EditText field where the user can enter a phone number.

The PhoneNumberUtils class provides utility methods for normalizing and formatting phone number strings. For example, to remove extraneous characters such as dashes and parentheses, use the normalizeNumber() method, which removes characters other than digits from a phone number string. For example, the statement below normalizes a phone number entered into an EditText view named `editText`:

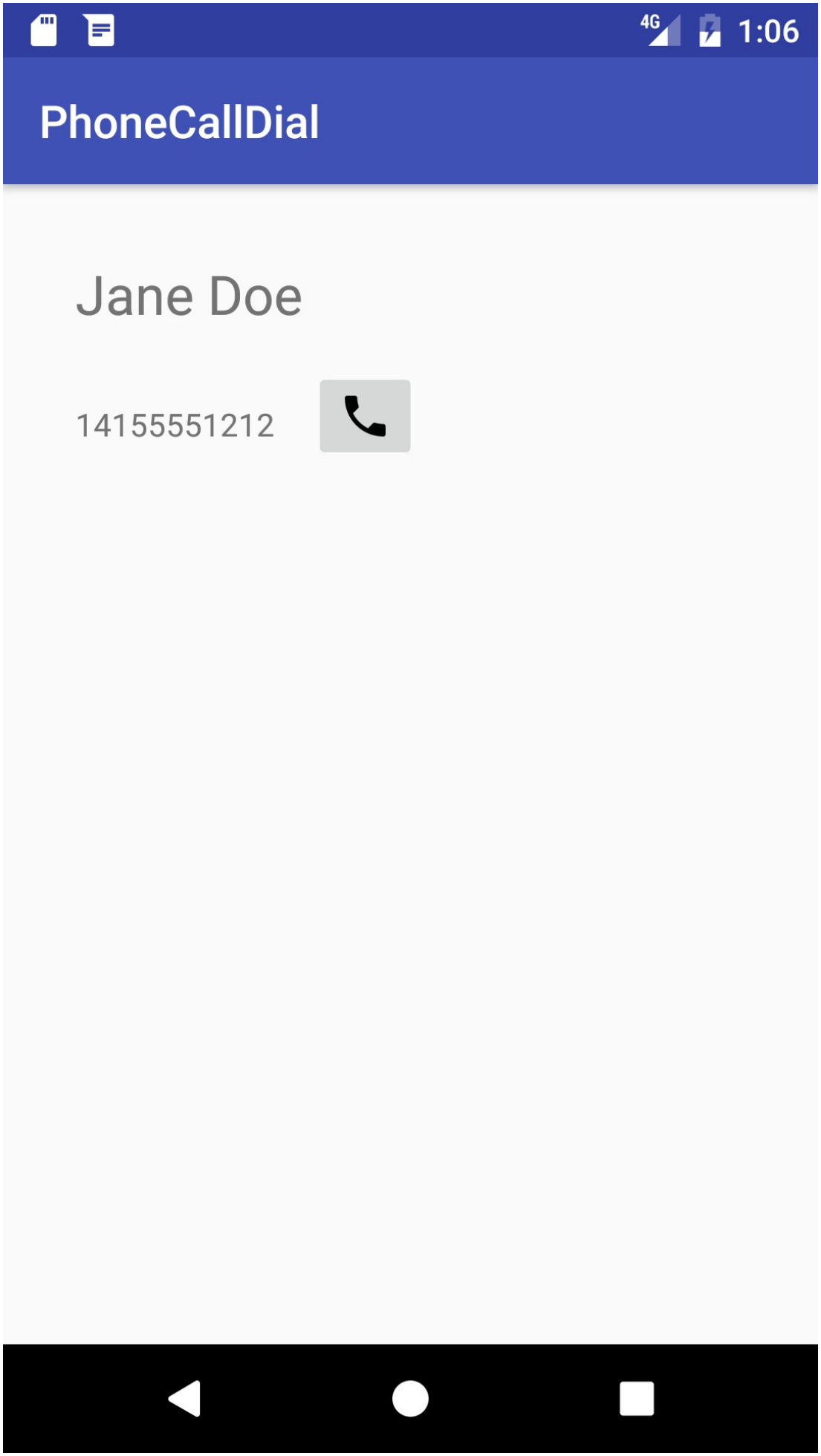
```
String normalizedPhoneNumber =
            PhoneNumberUtils.normalizeNumber(editText.getText().toString());
```

Note: The normalizeNumber() method, added to Android API level 21 (corresponding to Lollipop), is not available in older versions of the API.
Use the formatNumber() method to format a phone number string for a specific country if the given number doesn't include a country code. You can use `Locale.getDefault().getCountry()` to get the current country setting, or use the ISO 3166-1 two-letter country code to specify a country to use:

```
String formattedNumber = PhoneNumberUtils.formatNumber(normalizedPhoneNumber,
                    Locale.getDefault().getCountry());
```

Using an intent with the phone number to dial

To use an intent to launch the Phone app, use a button to let the user start the call. When the user taps the button, its click handler initiates the call. For example, a simple layout could provide an ImageButton like the phone icon in the figure below.


4G
1:06
PhoneCallDial
Jane Doe
14155551212

The Phone app opens with the number to be dialed. The user can change the number and initiate the call. The Phone app then makes the call.

```
<ImageButton
    android:id="@+id/phone_icon"
    ...
    android:onClick="dialNumber"/>
```

In the `dialNumber()` method, use an implicit intent with the intent action `ACTION_DIAL` to pass the phone number to the Phone app as a URI.

```
public void dialNumber() {
    TextView textView = (TextView) findViewById(R.id.number_to_call);
    // Use format with "tel:" and phone number to create phoneNumber.
    String phoneNumber = String.format("tel: %s",
                                       textView.getText().toString());
    // Create the intent.
    Intent dialIntent = new Intent(Intent.ACTION_DIAL);
    // Set the data for the intent as the phone number.
    dialIntent.setData(Uri.parse(phoneNumber));
    // If package resolves to an app, send intent.
    if (dialIntent.resolveActivity(getPackageManager()) != null) {
        startActivity(dialIntent);
    } else {
        Log.e(TAG, "Can't resolve app for ACTION_DIAL Intent.");
    }
}
```

In the above example, the code does the following:

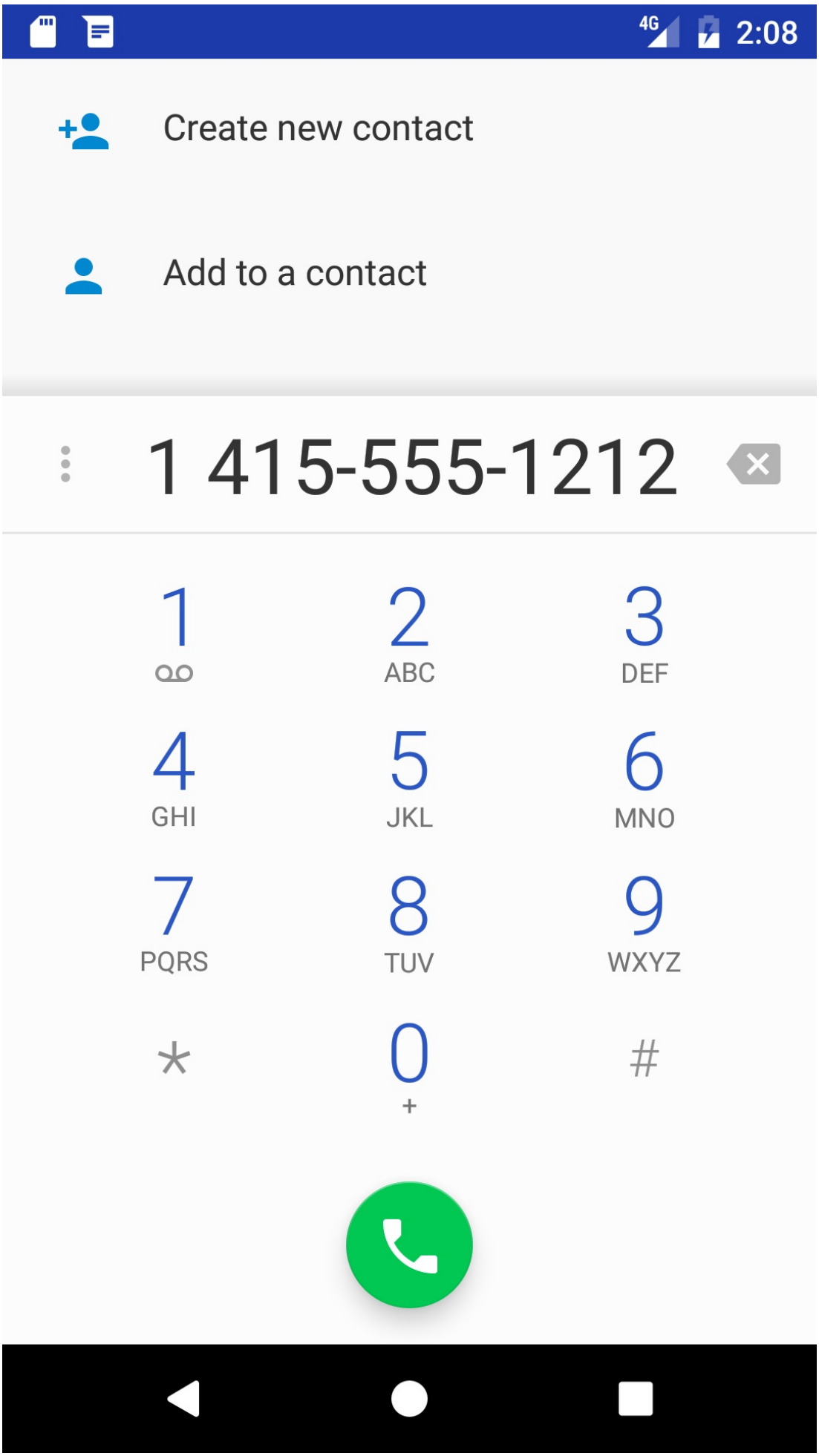
- Gets the text of the phone number from `textView` with `getText()` and uses it with `String.format` to include the `tel:` prefix (for example `tel:14155551212`):

  ```
  ...
    String phoneNumber = String.format("tel: %s",
                                       textView.getText().toString());
  ...
  ```

- Creates an implicit intent with the intent action `ACTION_DIAL` , and sets the phone number as the data for the intent using setData():

  ```
  …
  Intent intent = new Intent(Intent.ACTION_DIAL);
  // Set the data for the intent as the phone number.
  intent.setData(Uri.parse(phoneNumber));
  ...
  ```

- Checks if the implicit intent resolves to an app that is installed on the device.
    1. If it does, the code sends the intent with `startActivity()`, and the system launches the Phone app, as shown in the figure below.


2:08
Create new contact
Add to a contact
1 415-555-1212
1
2 ABC
3 DEF
4 GHI
5 JKL
6 MNO
7 PQRS
8 TUV
9 WXYZ
*
0 +
#

2. If it does not, an error is logged.

If there are no apps on the device that can receive the implicit intent, your app will crash when it calls `startActivity()`. To first verify that an app exists to receive the intent, call resolveActivity() on your Intent object with getPackageManager() to get a PackageManager instance for finding package information. The `resolveActivity()` method determines the best action to perform for a given intent. If the result is non-null, there is at least one app that can handle the intent and it's safe to call `startActivity()`.

```
…
if (intent.resolveActivity(getPackageManager()) != null) {
    startActivity(intent);
} else {
    Log.e(TAG, "Can't resolve app for ACTION_DIAL Intent.");
}
...
```

This example uses a hard-coded phone number, which is a useful technique for providing a support hotline number, or the selected phone number for a contact. In the next example, users can enter their own numbers to make calls.

Making a call from within your app

To make a phone call directly from your app, do the following:

1. Add permissions that enable making a call and reading the phone activity.
2. Check to see if telephony is enabled; if not, disable the phone feature.
3. Check to see if the user continues to grant permission, or request permission if needed.
4. Extend PhoneStateListener, and register the listener using the TelephonyManager class.
5. Use an implicit intent with `ACTION_CALL` to make the phone call.

Adding permissions to the manifest

Access to telephony information is permission-protected. In order to make a call, your app needs the `CALL_PHONE` permission. In addition, in order to monitor the phone state, your app needs the `READ_PHONE_STATE` permission. Both must be declared in the apps's AndroidManifest.xml file.

Add the following to your app's AndroidManifest.xml file after the first line (with the `package` definition) and before the `<application>` section:

```
...
<uses-permission android:name="android.permission.CALL_PHONE" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />

<application
...
```

Checking for TelephonyManager

Not all devices are enabled to use TelephonyManager. To check to see if telephony is enabled, follow these steps:

1. Retrieve a TelephonyManager using getSystemService() with the string constant TELEPHONY_SERVICE in the onCreate() method of the activity:

   ```
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      ...
      // Create a telephony manager.
      mTelephonyManager = (TelephonyManager)
                                getSystemService(TELEPHONY_SERVICE);
      ...
   }
   ```

2. Create a method to ensure that `mTelephonyManager` is not null, and that the SIM state is ready:

   ```
   private boolean isTelephonyEnabled() {
      if (mTelephonyManager != null) {
          if (mTelephonyManager.getSimState() ==
                                TelephonyManager.SIM_STATE_READY) {
              return true;
          }
      }
      return false;
   }
   ```

   The `getSimState()` method returns a constant indicating the state of the SIM card.

   The above `return` statement first checks if `telephonyManager` is not `null`, and if it is not, it returns `true` if the state of the SIM is "ready".

3. Call the above method in the `onCreate()` method of your activity. If telephony is not enabled, your code should disable the feature. The example below displays a toast message, logs a debug message, and disables the call button, effectively disabling the phone feature:

```
...
if (isTelephonyEnabled()) {
   Log.d(TAG, getString(R.string.telephony_enabled));
   // Todo: Register the PhoneStateListener.
   ...
   // Todo: Check for permission here.
   ...
} else {
  Toast.makeText(this,
              R.string.telephony_not_enabled,
              Toast.LENGTH_LONG).show();
  Log.d(TAG, getString(R.string.telephony_not_enabled));
  // Disable the call button.
  disableCallButton();
}
...
```

Checking and requesting user permission

Beginning in Android 6.0 (API level 23), users grant permissions to apps while the app is running, not when they install the app. This approach streamlines the app install process, since the user does not need to grant permissions when they install or update the app. It also gives the user more control over the app's functionality. However, your app must check for permission every time it does something that requires permission (such as making a phone call). If the user has used the Settings app to turn off Phone permissions for the app, your app can display a dialog to request permission.

Tip: For a complete description of the request permission process, see Requesting Permissions at Run Time.

Follow these steps to check for and request user permission to make phone calls:

1. At the top of the activity that makes a phone call, and below the activity's class definition, define a constant variable to hold the request code, and set it to `1`:

   ```
   private static final int MY_PERMISSIONS_REQUEST_CALL_PHONE = 1;
   ```

   Why the integer 1? Each permission request needs three parameters: the `context`, a string array of permissions, and an integer `requestCode`. The `requestCode` is an integer attached to the request, and it can be any integer that suits your use case. When a result returns in the activity, it contains this code and uses it to differentiate multiple permission results from each other.

2. In the activity that makes a phone call, create a private method `checkForPhonePermission()` that returns `void`.

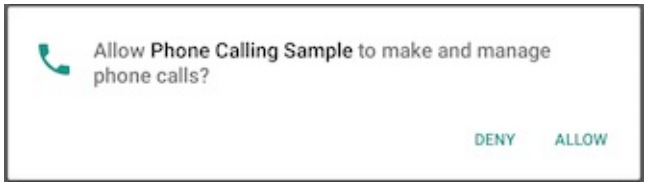
```
private void checkForPhonePermission() {
if (ActivityCompat.checkSelfPermission(this,
                    Manifest.permission.CALL_PHONE) !=
                    PackageManager.PERMISSION_GRANTED) {
    // Permission not yet granted. Use requestPermissions().
    Log.d(TAG, getString(R.string.permission_not_granted));
    ActivityCompat.requestPermissions(this,
                    new String[]{Manifest.permission.CALL_PHONE},
                    MY_PERMISSIONS_REQUEST_CALL_PHONE);
} else {
    // Permission already granted.
}
```

3. In the activity's `onCreate()` method, call the method to perform the telephony check:

```
...
if (isTelephonyEnabled()) {
   // Check for phone permission.
   checkForPhonePermission();
   // Todo: Register the PhoneStateListener.
} else {
...
```

Note the following about the `checkForPhonePermission()` method:

- The code uses `checkSelfPermission()` to determine whether your app has been granted a particular permission by the user.
- If permission has *not* been granted, the code uses the `requestPermissions()` method to display a standard dialog for the user to grant permission.
- The `requestPermissions()` method needs three parameters: the context, a string array of permissions, and the predefined integer `requestCode` .
- When your app calls `requestPermissions()` , the system shows a standard permission dialog to the user, as shown in the figure below. Your app can't configure or alter the dialog.



Tip: If you want to provide any information or explanation to the user, you must do that before you call `requestPermissions()` , as described in Explain why the app needs permissions.

When the user responds to the request permission dialog, the system invokes your app's onRequestPermissionsResult() method, passing it the user response. Override that method to find out whether the permission was granted:

```
@Override
public void onRequestPermissionsResult(int requestCode,
                    String permissions[], int[] grantResults) {
    switch (requestCode) {
        case MY_PERMISSIONS_REQUEST_CALL_PHONE: {
            if (permissions[0].equalsIgnoreCase
                        (Manifest.permission.CALL_PHONE)
                        && grantResults[0] ==
                        PackageManager.PERMISSION_GRANTED) {
                // Permission was granted.
            } else {
                // Permission denied. Stop the app.
                Log.d(TAG, getString(R.string.failure_permission));
                Toast.makeText(this,
                            getString(R.string.failure_permission),
                            Toast.LENGTH_SHORT).show();
                // Disable the call button
                disableCallButton();
            }
        }
    }
}
```

The above code snippet shows a `switch` statement based on the value of `requestCode`, with one `case` to check if the permission is the one you defined as `MY_PERMISSIONS_REQUEST_CALL_PHONE`.

Tip: It helps to use a `switch` statement so that you can add more requests to the app.

The user's response to the request dialog is returned in the `permissions` array (index `0` if only one permission is requested in the dialog). Compare this to the corresponding grant result, which is either `PERMISSION_GRANTED` or `PERMISSION_DENIED`.

If the user denies a permission request, your app should take appropriate action. For example, your app might disable the functionality that depends on this permission (such as the call button) and show a dialog explaining why it could not perform it.

Extending and registering PhoneStateListener

PhoneStateListener monitors changes in specific telephony states such as ringing, off-hook, and idle. To use PhoneStateListener:

- Implement a listener class that extends PhoneStateListener, and override its

onCallStateChanged() method.
- In the onCallStateChanged() method, provide actions based on the phone state.
- Register the listener using the TelephonyManager class, which provides access to information about the telephony services on the device.

Implementing a listener

Create a private class in your activity that extends PhoneStateListener:

```
private class MyPhoneCallListener extends PhoneStateListener {
    ...
}
```

Within this class, implement the `onCallStateChanged()` method of PhoneStateListener to take actions based on the phone state. The code below uses a `switch` statement with constants of the TelephonyManager class to determine which of three states the phone is in: `CALL_STATE_RINGING`, `CALL_STATE_OFFHOOK`, and `CALL_STATE_IDLE`:

```
@Override
public void onCallStateChanged(int state, String incomingNumber) {
    switch (state) {
        case TelephonyManager.CALL_STATE_RINGING:
            // Incoming call is ringing.
            break;
        case TelephonyManager.CALL_STATE_OFFHOOK:
            // Phone call is active -- off the hook.
            break;
        case TelephonyManager.CALL_STATE_IDLE:
            // Phone is idle before and after phone call.
            ...
            break;
        default:
            // Must be an error. Raise an exception or just log it.
            break;
    }
}
```

Provide actions for phone states

Add the actions you want to take based on the phone states. For example, your code can log a message for testing purposes, and display a toast to the user. The `CALL_STATE_RINGING` state includes the incoming phone number, which your code can show in a toast.

The phone is in the `CALL_STATE_IDLE` state until a call is started. The phone state changes to `CALL_STATE_OFFHOOK` in order to make the connection and stays in the state for the duration of the call. The phone state returns to the `CALL_STATE_IDLE` state after the call finishes or if the call is denied or not completed.

Your app resumes when the state changes back to the `CALL_STATE_IDLE` state.

Tip: An app running on Android versions prior to KitKat (version 19) doesn't resume when the phone state returns to `CALL_STATE_IDLE` from `CALL_STATE_OFFHOOK` at the end of a call. The code below sets the flag `returningFromOffHook` to `true` when the state is `CALL_STATE_OFFHOOK`, so that when the state is back to `CALL_STATE_IDLE`, you can use the flag to catch the end-of-call and restart the app's activity.

```
private class MyPhoneCallListener extends PhoneStateListener {
    private boolean returningFromOffHook = false;

    @Override
    public void onCallStateChanged(int state, String incomingNumber) {
        // Define a string for the message to use in a toast.
        String message = getString(R.string.phone_status);
        switch (state) {
            case TelephonyManager.CALL_STATE_RINGING:
                // Incoming call is ringing
                message = message +
                        getString(R.string.ringing) + incomingNumber;
                Toast.makeText(MainActivity.this, message,
                        Toast.LENGTH_SHORT).show();
                Log.i(TAG, message);
                break;
            case TelephonyManager.CALL_STATE_OFFHOOK:
                // Phone call is active -- off the hook
                message = message + getString(R.string.offhook);
                Toast.makeText(MainActivity.this, message,
                                    Toast.LENGTH_SHORT).show();
                Log.i(TAG, message);
                returningFromOffHook = true;
                break;
            case TelephonyManager.CALL_STATE_IDLE:
                // Phone is idle before and after phone call.
                // If running on version older than 19 (KitKat),
                // restart activity when phone call ends.
                message = message + getString(R.string.idle);
                Toast.makeText(MainActivity.this, message,
                                   Toast.LENGTH_SHORT).show();
                Log.i(TAG, message);
                if (returningFromOffHook) {
                    // No need to do anything if >= version K
                    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.KITKAT) {
                        Log.i(TAG, getString(R.string.restarting_app));
                        // Restart the app.
```

```
                        Intent intent = getPackageManager()
                                    .getLaunchIntentForPackage(
                                    .getPackageName());
                        i.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
                        startActivity(i);
                    }
                }
                break;
            default:
                message = message + "Phone off";
                Toast.makeText(MainActivity.this, message,
                                        Toast.LENGTH_SHORT).show();
                Log.i(TAG, message);
                break;
        }
    }
}
```

Registering your listener

Register the listener object using the TelephonyManager class:

- Use getSystemService() with `Context.TELEPHONY_SERVICE`.
- Use telephonyManager.listen() with the PhoneStateListener set to the `LISTEN_CALL_STATE`.

A good place to do this is in the activity's `onCreate()` method right after checking for phone permission, which is after ensuring that telephony is enabled. Follow these steps:

1. At the top of the activity, define a variable for the PhoneStateListener:

   ```
   private MyPhoneCallListener mListener;
   ```

2. In the `onCreate()` method, add the following code after checking for telephony and permission:

   ```
   ...
   if (isTelephonyEnabled()) {
      ...
      checkForPhonePermission();
      // Register the PhoneStateListener to monitor phone activity.
      mListener = new MyPhoneCallListener();
      telephonyManager.listen(mListener,
                  PhoneStateListener.LISTEN_CALL_STATE);
   } else { ...
   ```

3. You must also unregister the listener in the activity's onDestroy() method. This method is usually implemented to free resources like threads that are associated with an

activity, so that a destroyed activity does not leave such things around while the rest of its application is still running. Override the `onDestroy()` method by adding the following code:

```
@Override
protected void onDestroy() {
   super.onDestroy();
   if (isTelephonyEnabled()) {
       telephonyManager.listen(mListener,
                               PhoneStateListener.LISTEN_NONE);
   }
}
```

Using an implicit intent to make the call

To launch the Phone app and start the call:

- Use a button in the layout with an `onClick` method such as `callNumber()`.
- In the `callNumber()` method, create an implicit intent with the intent action `ACTION_CALL`.
- Set the phone number as the data for the intent with `setData()`.

The following is an sample `callNumber()` method:

```
public void callNumber(View view) {
    EditText editText = (EditText) findViewById(R.id.editText_main);
    // Use format with "tel:" and phone number to create phoneNumber.
    String phoneNumber = String.format("tel: %s",
                                       editText.getText().toString());
    // Create the intent.
    Intent callIntent = new Intent(Intent.ACTION_CALL);
    // Set the data for the intent as the phone number.
    callIntent.setData(Uri.parse(phoneNumber));
    // If package resolves to an app, check for phone permission,
    // and send intent.
    if (callIntent.resolveActivity(getPackageManager()) != null) {
        checkForPhonePermission();
        startActivity(callIntent);
    } else {
        Log.e(TAG, "Can't resolve app for ACTION_CALL Intent.");
    }
}
```

As in the previous example for passing a number for dialing, you use a string for the phone number with the `tel:` prefix.

If the implicit intent resolves to an installed app, use the `checkForPhonePermission()` method you created previously to check to see if the app still has permission to make the call. You must check for that permission every time you perform an operation that requires it, because the user is always free to revoke the permission. Even if the app used the phone a minute ago, it can't assume it still has that permission a minute later.

Using emulators to test phone call functionality

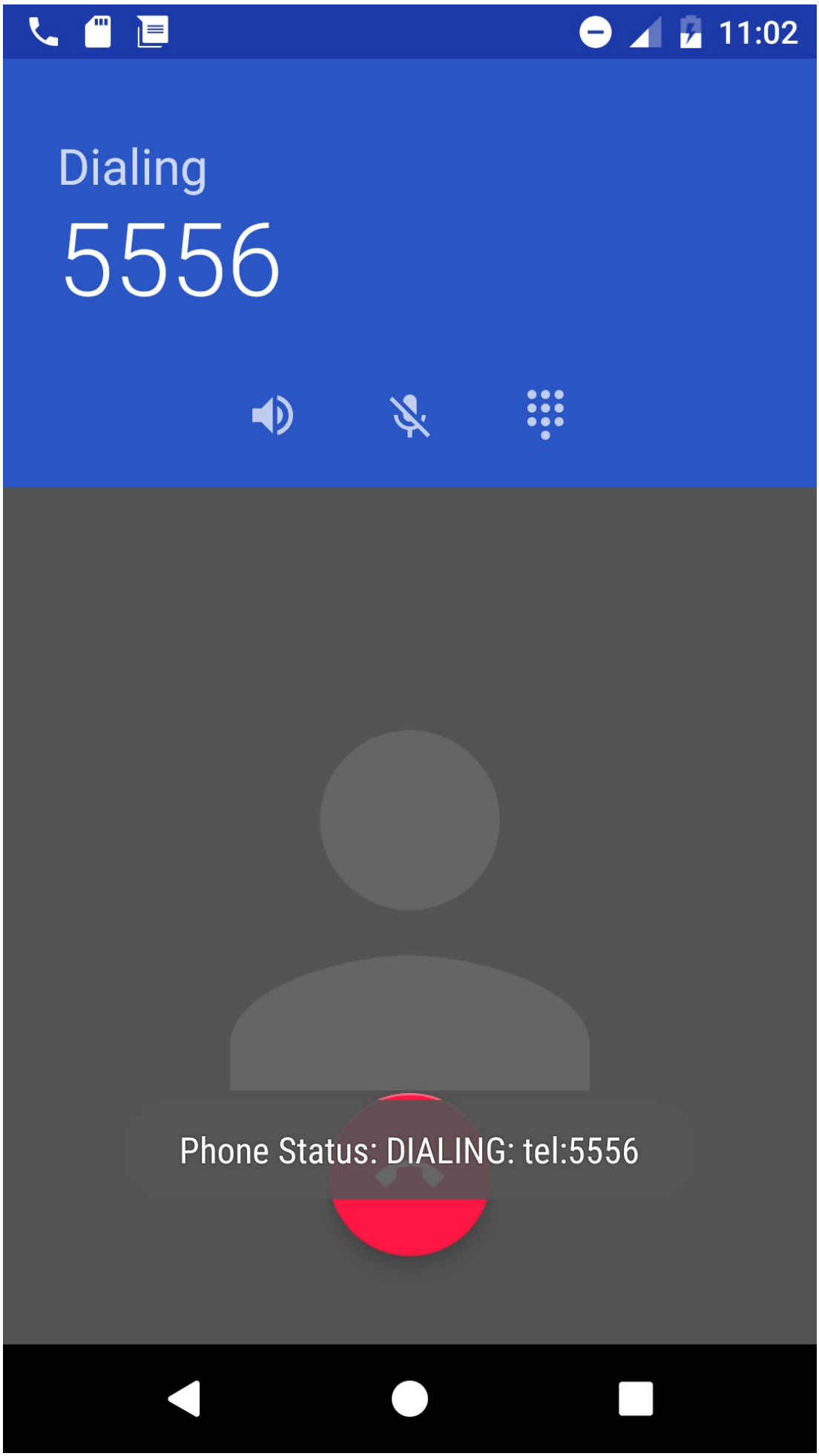
If you don't have cellular service on your device, or telephony is not enabled, you can test the app using two emulator instances—one emulator instance calls the other one.

1. Launch an emulator directly from the AVD Manager by choosing **Tools > Android > AVD Manager**.
2. Double-click a predefined device. Note the number that appears in the emulator's window title on the far right, as shown in the figure below as #1 (5556). This is the port number of the emulator instance.

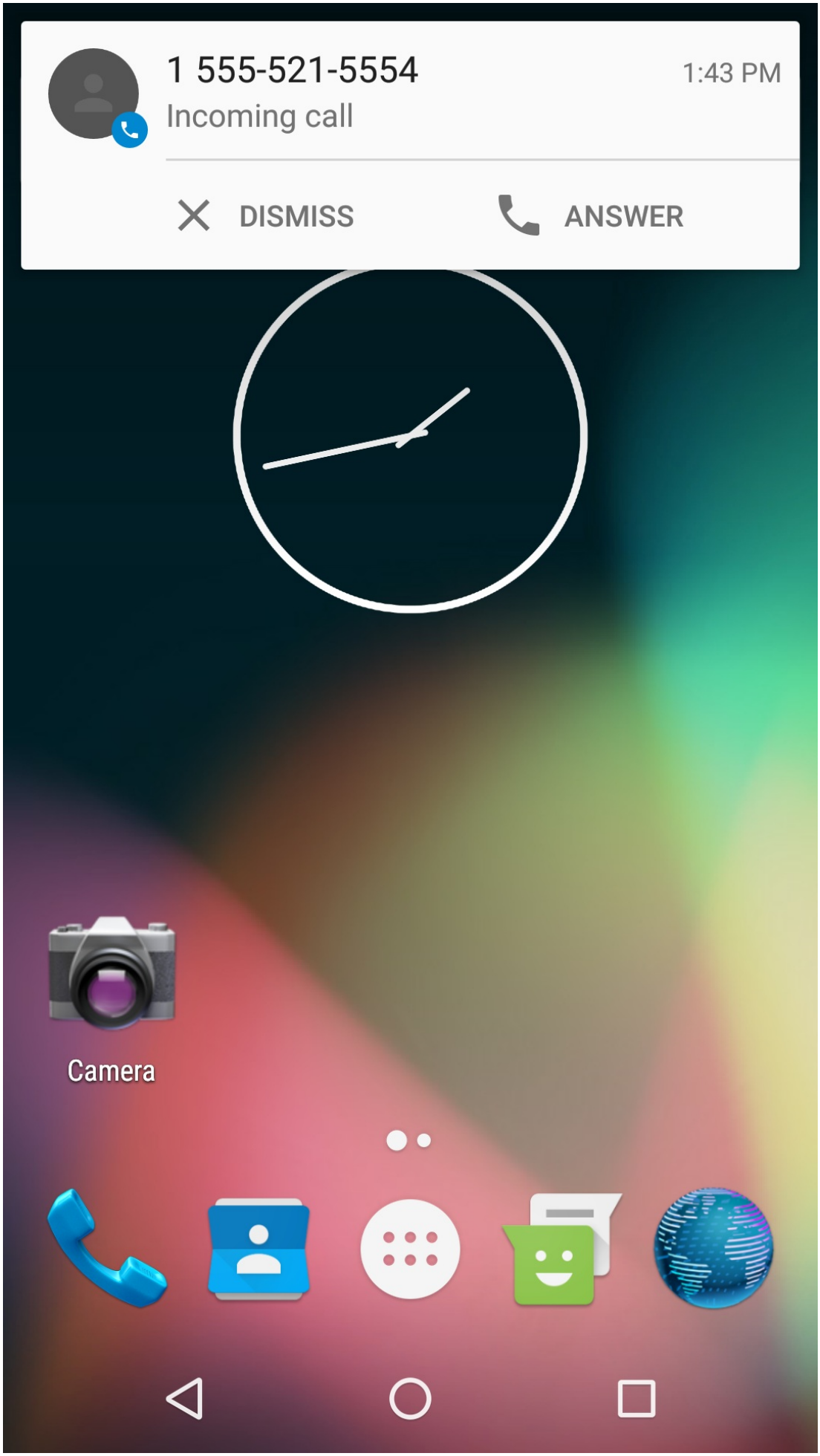
3. Open the Android Studio project for the phone-calling app.
4. Run the phone-calling app, but choose *another* emulator from the list—*not* the one that

is already running.

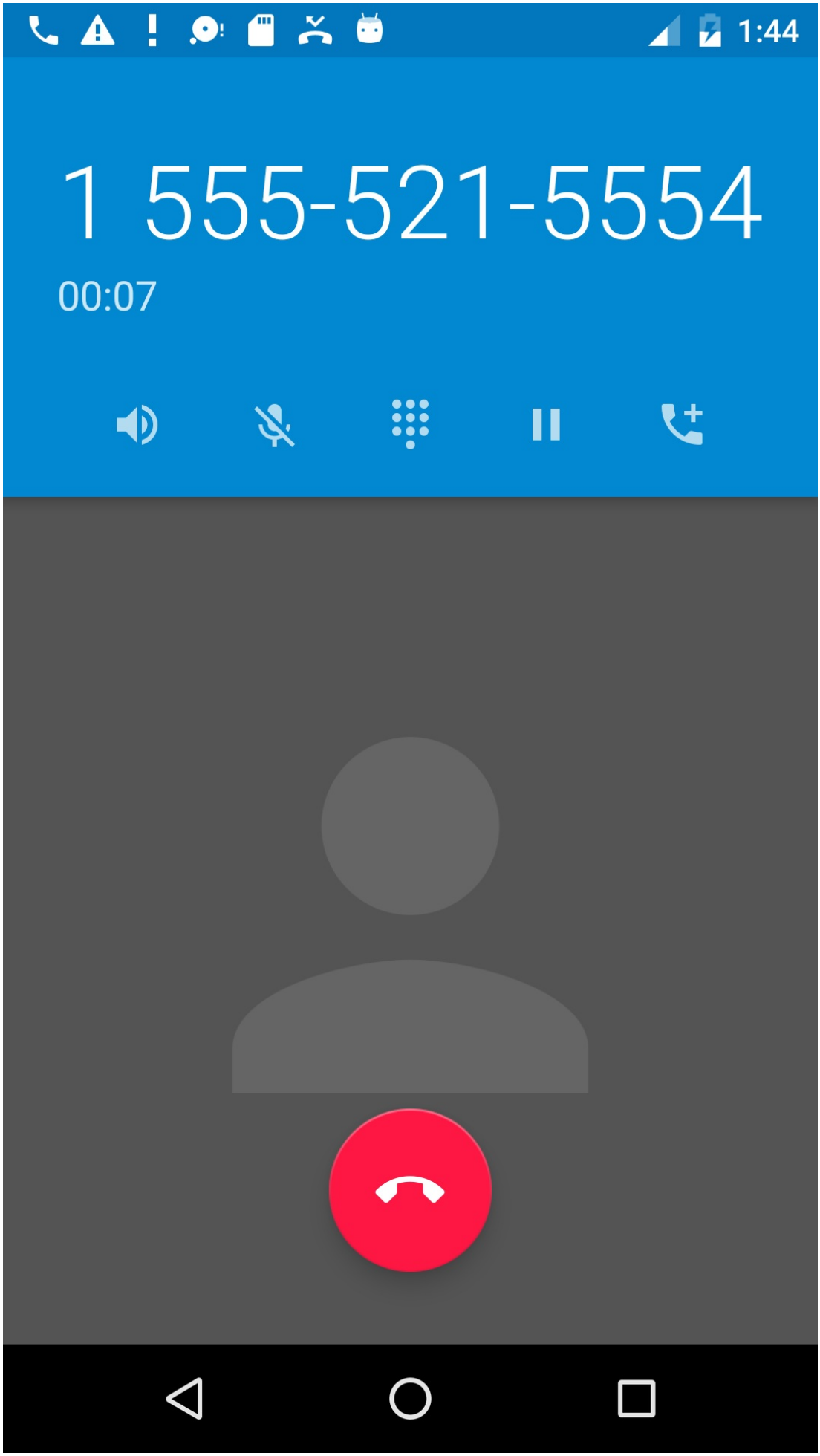
5. In the phone-calling app, instead of a real phone number, enter the port number (as in 5556), and click the call button. The emulator shows the phone call starting up, as shown in the figure below.


11:02
Dialing
5556
Phone Status: DIALING: tel:5556

The other emulator instance should now be receiving the call, as shown in the figure below:


1 555-521-5554
1:43 PM
Incoming call
DISMISS
ANSWER
Camera

6. Click **Answer** or **Dismiss** on the emulator receiving the call,. After you click **Answer**, also click the red **Hang-up** button to end the call.


1:44
1 555-521-5554
00:07

Related practical

- Making Phone Calls - Part 1
- Making Phone Calls - Part 2

Learn more

- Android Developer Reference:
  - Common Intents
  - TelephonyManager
  - PhoneStateListener
  - Requesting Permissions at Run Time
  - checkSelfPermission
  - Run Apps on the Android Emulator
  - Intents and Intent Filters
  - Intent
- Stack Overflow:
  - How to format a phone number using PhoneNumberUtils?
  - How to make phone call using intent in android?
  - Ringing myself using android emulator
  - Fake Incoming Call Android
  - Simulating incoming call or sms in Android Studio
- Other
  - User (beginner) tutorial: How to Make Phone Calls with Android
  - Developer Video: How to Make a Phone Call

1.1: Part 1 - Making Phone Calls

Contents:

Android mobile devices with telephone/cellular service are pre-installed with a Phone app for making calls, which includes a dialer for dialing any phone number. You use an implicit Intent to launch the Phone app from your app. You have two choices:

- Use `ACTION_DIAL` to launch the Phone app independently from your app with the phone number displayed in the dialer. The user then makes the call in the Phone app. This is the preferred action for apps that don't have to monitor the phone's state.
- Use `ACTION_CALL` to launch the Phone app in the context of your app, making the call directly from your app, and monitoring the phone state. This action keeps the user within your app, without having to navigate back to the app. Your app must request permission from the user before making the call if the user hasn't already granted permission.

What you should already KNOW

From the previous chapters, you should be able to:

- Create and run interactive apps in Android Studio.
- Work with XML layouts.
- Create an implicit intent to perform an action using another app.

What you will LEARN

In this practical, you will learn to:

- Pass a phone number to the Phone app's dialer.
- Perform a phone call within your app.
- Test to see if telephony services are enabled.

- Check for calling permission, and request permission if required.

What you will DO

In this practical, you will:

- Create an app that uses an implicit intent to launch the Phone app.
- Create another app that makes phone calls from within the app.
- Test to see if telephony services are enabled before enabling the app.
- Check for calling permission, which can change at any time.
- Request permission from the user, if necessary, to make the call.

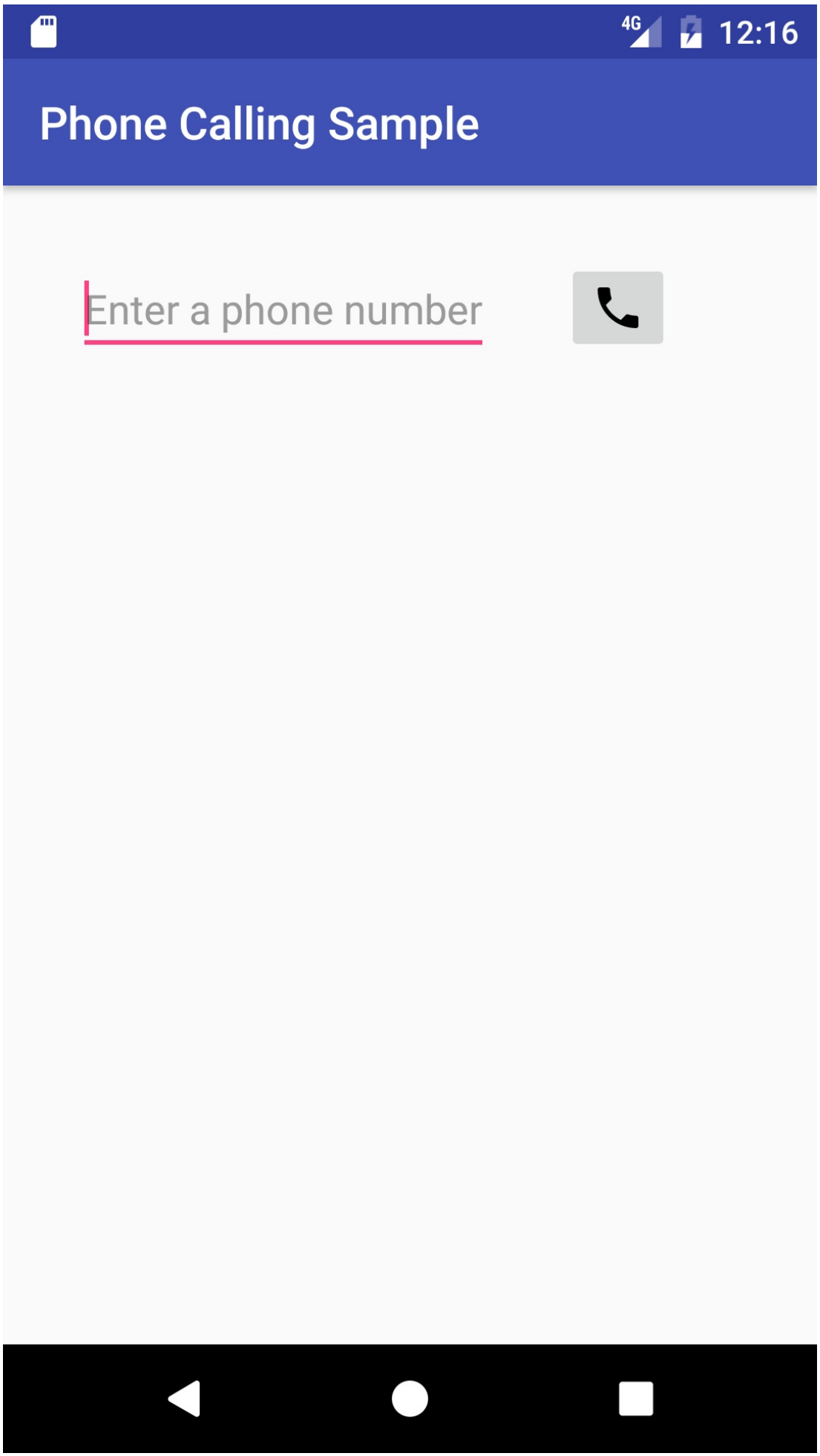
App overview

You will create two apps:

- **PhoneCallDial**: A basic app that uses an implicit intent to launch the Phone app with a hard-coded phone number for dialing. The Phone app makes the call. You could use this technique to provide a one-button dialer to custom support. In this lesson you will build a layout, shown in the figure below. It includes a TextView with a hard-coded phone number, and an ImageButton with an icon to launch the Phone app with that phone number in its dialer.


4G
1:06
PhoneCallDial
Jane Doe
14155551212

- **Phone Calling Sample**: An app that secures permission, uses an implicit intent to make a phone call from the app, and uses the TelephonyManager class to monitor the phone's state. You would use this technique if you want to keep the user within your app, without having to navigate back to the app. In this lesson, you modify the above layout to use an EditText so that users can enter the phone number. The layout looks like the figure below:


12:16
Phone Calling Sample
Enter a phone number

Task 1. Send an intent with the phone number to dial

In this task you will create an app that uses an implicit intent to launch the Phone app to dial a given phone number. To send that intent, your app needs to prepare a Uniform Resource Identifier (URI) that is prefixed by "tel:" (for example `tel:14155551212`).

1.1 Create the app and layout

1. Create a project using the Empty Activity template and call it **PhoneCallDial**.
2. Add an icon for the call button by following these steps:
    i. Select the drawable/ folder in the Project: Android view and choose **File > New > Vector Asset**.
    ii. Click the Android icon next to "Icon:" to choose an icon. To find a handset icon, choose **Communication** in the left column.
    iii. Select the icon, click **OK**, click **Next**, and then click **Finish**.
3. Open the activity_main.xml layout file.
    i. Change the root view to RelativeLayout.
    ii. In the "Hello World" TextView element, remove the `layout_constraint` attributes, if they are present.
    iii. Change the TextView to show a dummy contact name, as if the app had retrieved the name from a contacts database and assigned it to a TextView:

```
<TextView
android:id="@+id/contact_name"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_margin="@dimen/activity_horizontal_margin"
android:textSize="24sp"
android:text="Jane Doe" />
```

4. Extract the strings and dimensions into resources:
    - `24sp` : `contact_text_size` for the text size.
    - `Jane Doe` : `contact_name` for the text.
5. Add another TextView for the phone number:

```
<TextView
    android:id="@+id/number_to_call"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="@dimen/activity_horizontal_margin"
    android:layout_below="@id/contact_name"
    android:text="14155551212" />
```

You will use the `android:id` `number_to_call` to retrieve the phone number.

6. After adding a hard-coded phone number string, extract it into the resource `phone_number`.
7. Add an ImageButton for initiating the call:

```
<ImageButton
    android:id="@+id/phone_icon"
    android:contentDescription="Make a call"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/contact"
    android:layout_toRightOf="@id/number_to_call"
    android:layout_toEndOf="@id/number_to_call"
    android:src="@drawable/ic_call_black_24dp"
    android:onClick="dialNumber"/>
```

Use the vector asset you added previously (for example `ic_call_black_24dp` for a phone handset icon) for the `android:src` attribute. You will use the `android:id` `@phone_icon` to refer to the button for dialing the phone.

The `dialNumber` method referred to in the `android:onClick` attribute remains highlighted until you create this method in the MainActivity, which you do in the next step.

8. After adding a hard-coded content description, extract it into the string resource `make_a_call`.
9. Click `dialNumber` in the `android:onClick` attribute, click the red light bulb that appears, and then select **Create dialNumber(View) in 'MainActivity'**. Android Studio automatically creates the `dialNumber()` method in MainActivity as `public`, returning `void`, with a `View` parameter. This method is called when the user taps the ImageButton.

```
public void dialNumber(View view) {
}
```

The layout should look something like the figure below.


4G
1:06
PhoneCallDial
Jane Doe
14155551212

The following is the complete code for the XML layout in activity_main.xml, including comments:

```
<RelativeLayout …
    <!-- TextView for a dummy contact name from a contacts database -->
    <TextView
        android:id="@+id/contact_name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/activity_horizontal_margin"
        android:textSize="@dimen/contact_text_size"
        android:text="@string/contact" />

    <!-- TextView for a hard-coded phone number  -->
    <TextView
        android:id="@+id/number_to_call"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/activity_horizontal_margin"
        android:layout_below="@id/contact_name"
        android:text="@string/phone_number" />

    <!-- The dialNumber() method will be called by this button.  -->
    <ImageButton
        android:id="@+id/phone_icon"
        android:contentDescription="@string/make_a_call"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/contact_name"
        android:layout_toRightOf="@id/number_to_call"
        android:layout_toEndOf="@id/number_to_call"
        android:src="@drawable/ic_call_black_24dp"
        android:onClick="dialNumber"/>
</RelativeLayout>
```

1.2 Edit the onClick method in MainActivity

1. In MainActivity, define a constant for the log tag.

   ```
   public static final String TAG = MainActivity.class.getSimpleName();
   ```

2. Inside the `dialNumber()` method created in the previous section, create a reference to the `number_to_call` TextView:

   ```
   public void dialNumber(View view) {
      TextView textView = (TextView) findViewById(R.id.number_to_call);
      ...
   ```

3. To create the phone number URI string `phoneNumber`, get the phone number from `textView` and use it with `String.format` to include the `tel:` prefix:

```
...
// Use format with "tel:" and phone number to create mPhoneNum.
String phoneNumber = String.format("tel: %s",
                                   textView.getText().toString());
...
```

4. Define an implicit intent (`dialIntent`) with the intent action `ACTION_DIAL`, and set the `phoneNumber` as data for the intent:

```
...
// Create the intent.
Intent dialIntent = new Intent(Intent.ACTION_DIAL);
// Set the data for the intent as the phone number.
dialIntent.setData(Uri.parse(phoneNumber));
...
```

5. To verify that an app exists to receive the intent, call resolveActivity() on your Intent object with getPackageManager() to get a PackageManager instance for finding package information. The `resolveActivity()` method determines the best action to perform for a given intent. If the result is non-null, there is at least one app that can handle the intent and it's safe to call `startActivity()`.

```
...
// If package resolves to an app, send intent.
if (dialIntent.resolveActivity(getPackageManager()) != null) {
     startActivity(dialIntent);
} else {
     Log.e(TAG, "Can't resolve app for ACTION_DIAL Intent.");
}
...
```

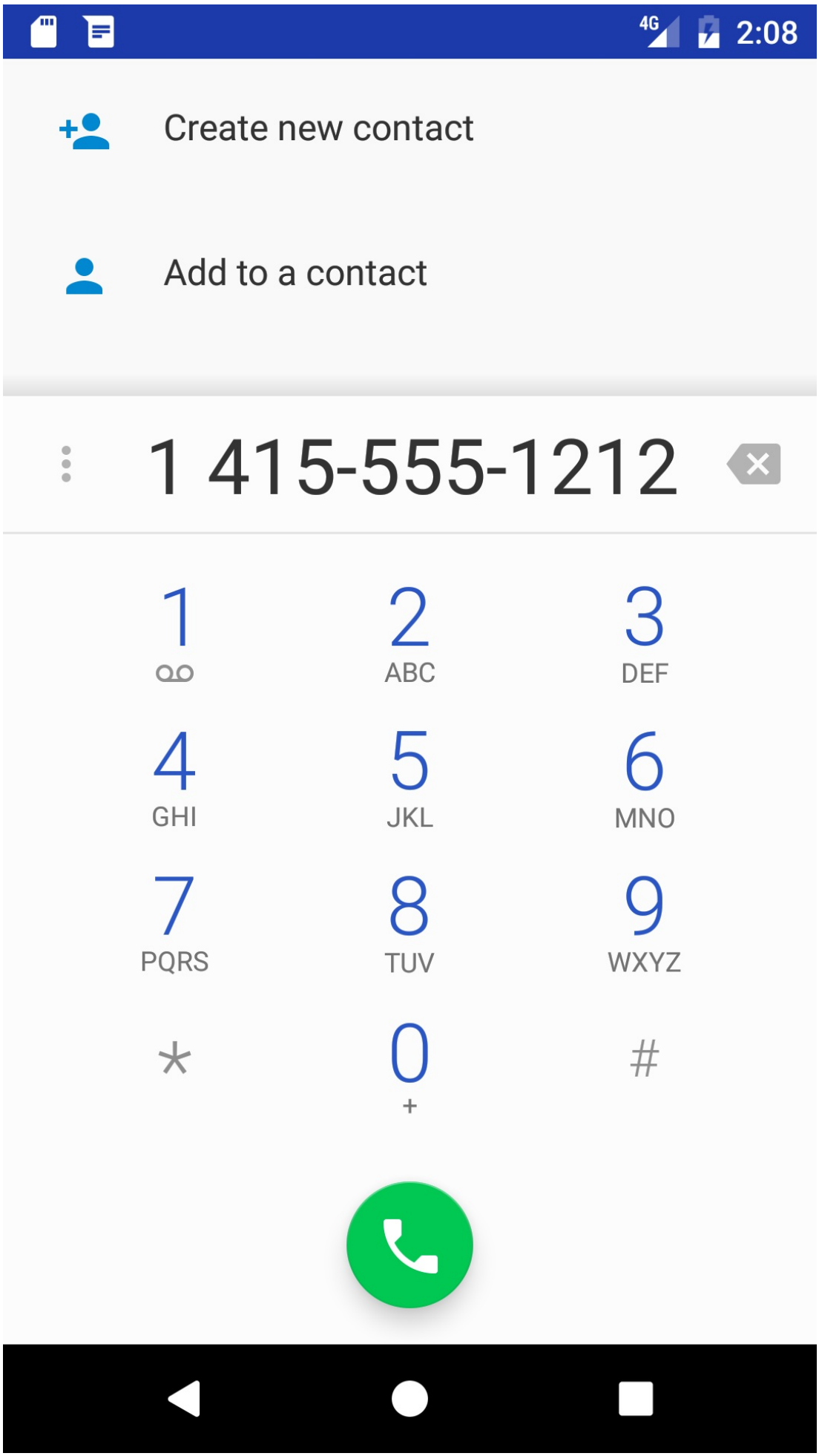
The full method should now look like the following:

```
public void dialNumber() {
    TextView textView = (TextView) findViewById(R.id.number_to_call);
    // Use format with "tel:" and phone number to create phoneNumber.
    String phoneNumber = String.format("tel: %s",
                                       textView.getText().toString());
    // Create the intent.
    Intent dialIntent = new Intent(Intent.ACTION_DIAL);
    // Set the data for the intent as the phone number.
    dialIntent.setData(Uri.parse(phoneNumber));
    // If package resolves to an app, send intent.
    if (dialIntent.resolveActivity(getPackageManager()) != null) {
        startActivity(dialIntent);
    } else {
        Log.e(TAG, "Can't resolve app for ACTION_DIAL Intent.");
    }
}
```

1.3 Run the app

You can run the app on either an emulator or a device:

1. Click or tap the phone icon. The dialer should appear with the phone number ready to use, as shown in the figure below:


4G
2:08
Create new contact
Add to a contact
1 415-555-1212
1
2
ABC
3
DEF
4
GHI
5
JKL
6
MNO
7
PQRS
8
TUV
9
WXYZ
*
0
+
#

2. The `phone_number` string holds a fixed number (1-415-555-1212). You can change the number in the Phone app's dialer before calling.
3. Use the **Back** button to return to the app. You may need to tap or click it two or three times to navigate backwards from the Phone app's dialer and Favorites list.

Solution code

Android Studio project: PhoneCallDial

Task 2. Make a phone call from within an app

In this task you will copy the PhoneCallDial app from the previous task, refactor and rename it to **PhoneCallingSample**, and modify its layout and code to create an app that enables a user to enter a phone number and perform the phone call from within your app.

In the first step you will add the code to make the call, but the app will work only if telephony is enabled, and if the app's permission for Phone is set manually in Settings on the device or emulator.

In subsequent steps you will do away with setting this permission manually by requesting phone permission from the user if it is not already granted. You will also add a telephony check to display a message if telephony is not enabled and code to monitor the phone state.

2.1 Create the app and add permission

1. Copy the **PhoneCallDial** project folder, rename the folder to **PhoneCallingSample**, and refactor the app to populate the new name throughout the app project. (See the Appendix for instructions on copying a project.)
2. Add the following permission to the AndroidManifest.xml file after the first line (with the `package` definition) and before the `<application>` section:

```
<uses-permission android:name="android.permission.CALL_PHONE" />
```

   Your app can't make a phone call without the `CALL_PHONE` permission line in AndroidManifest.xml. This permission line enables a setting for the app in the Settings app that gives the user the choice of allowing or disallowing use of the phone. (In the next task you will add a way for the user to grant that permission from within the app.)

2.2 Create the app layout

1. Open **activity_main.xml** to edit the layout.
2. Remove the `contact_name` TextView, and replace the `number_to_call` TextView with the following EditText view:

```
<EditText
    android:id="@+id/editText_main"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="@dimen/activity_horizontal_margin"
    android:inputType="phone"
    android:hint="Enter a phone number" />
```

3. After adding a hard-coded string for the `android:hint` attribute, extract it into the string resource `enter_phone`, and note the following:
    - You will use the `android:id` for the EditText view in your code to retrieve the phone number.
    - The EditText view uses the `android:inputType` attribute set to `"phone"` for a phone-style numeric keypad.
4. Change the ImageButton as follows:
    i. Change the `android:layout_below`, `android:layout_toRightOf`, and `android:layout_toEndOf` attributes to refer to `editText_main`.
    ii. Add the `android:visibility` attribute set to `visible`. You will control the visibility of this ImageButton from your code.
    iii. Change the android:onClick method to `callNumber`. This will remain highlighted until you add that method to MainActivity.

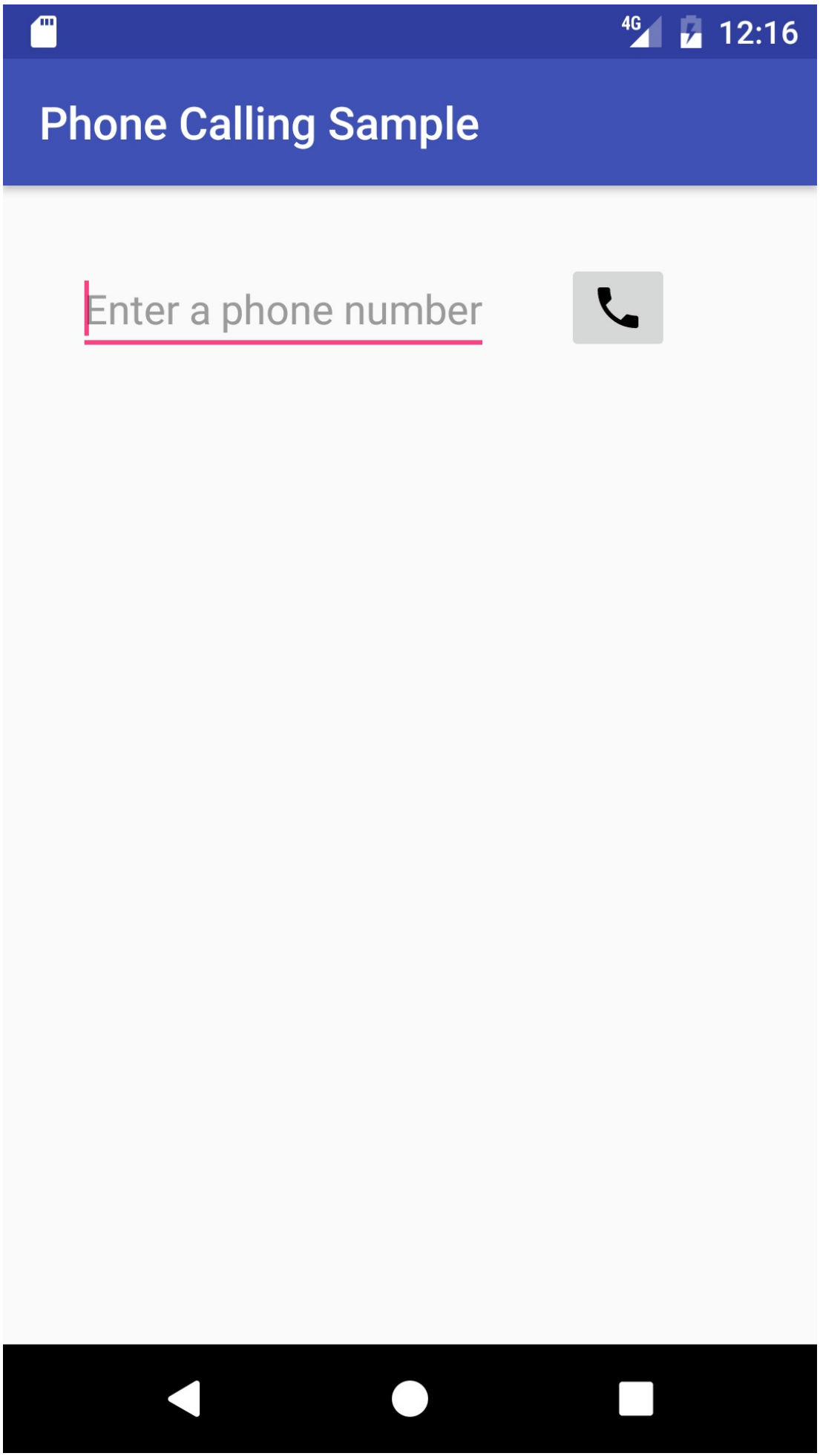
```
<ImageButton
  android:id="@+id/phone_icon"
  android:contentDescription="@string/make_a_call"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_margin="@dimen/activity_horizontal_margin"
  android:layout_toRightOf="@id/editText_main"
  android:layout_toEndOf="@id/editText_main"
  android:src="@drawable/ic_call_black_24dp"
  android:visibility="visible"
  android:onClick="callNumber"/>
```

5. Add the following Button at the end of the layout, before the ending `</RelativeLayout>` tag:

```
<Button
    android:id="@+id/button_retry"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:contentDescription="Retry"
    android:layout_below="@id/editText_main"
    android:text="Retry"
    android:visibility="invisible"/>
</RelativeLayout>
```

6. After adding a hard-coded string "Retry" for the `android:contentDescription` attribute, extract it into the string resource `retry`, and then replace the hard-coded string in the `android:text` attribute to `"@string/retry"`.
7. Note the following:
    - You will refer to the the `android:id` `button_retry` in your code.
    - Make sure you include the `android:visibility` attribute set to `"invisible"`. It should appear only if the app detects that telephony is not enabled, or if the user previously denied phone permission when the app requested it.

Your app's layout should look like the following figure (the `button_retry` Button is invisible):


12:16
Phone Calling Sample
Enter a phone number

2.3 Change the onClick method in MainActivity

1. In MainActivity, refactor and rename the `dialNumber()` method to call it `callNumber()`.
2. Change the first statement, which referred to a TextView, to use an EditText view:

```
public void callNumber(View view) {
  EditText editText = (EditText) findViewById(R.id.editText_main);
  ...
}
```

3. Change the next statement to get the phone number from the EditText view (`editText`) to create the phone number URI string `phoneNumber`:

```
// Use format with "tel:" and phone number to create phoneNumber.
String phoneNumber = String.format("tel: %s",
                                   editText.getText().toString());
```

4. Before the intent, add code to show a log message and a toast message with the phone number:

```
// Log the concatenated phone number for dialing.
Log.d(TAG, "Phone Status: DIALING: " + phoneNumber);
Toast.makeText(this,
               "Phone Status: DIALING: " + phoneNumber,
               Toast.LENGTH_LONG).show();
```

5. Extract `"Phone Status: DIALING: "` to a string resource (`dial_number`). Replace the second use of the string in the `Toast.makeText()` statement to `getString(R.string.dial_number)`.
6. Refactor and rename the `dialIntent` implicit intent to `callIntent`, and replace `ACTION_DIAL` with `ACTION_CALL`. As a result, the statements should now look like this:

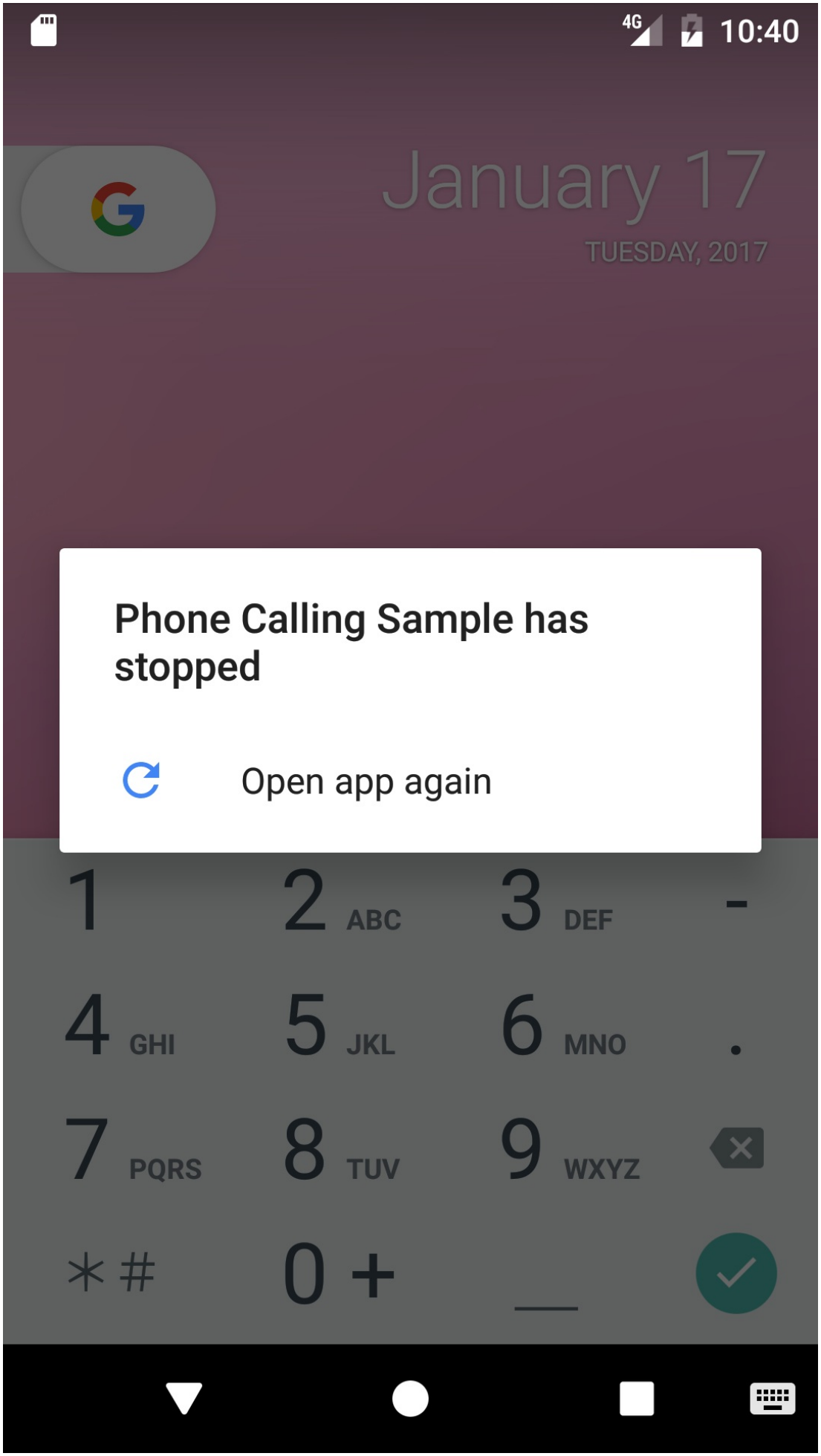
```
...
// Create the intent.
Intent callIntent = new Intent(Intent.ACTION_CALL);
// Set the data for the intent as the phone number.
callIntent.setData(Uri.parse(phoneNumber));
// If package resolves to an app, send intent.
if (callIntent.resolveActivity(getPackageManager()) != null) {
   startActivity(callIntent);
} else {
   Log.e(TAG, "Can't resolve app for ACTION_CALL Intent.");
}
...
```

The full method should now look like the following:

```
public void callNumber() {
    EditText editText = (EditText) findViewById(R.id.editText_main);
    // Use format with "tel:" and phone number to create phoneNumber.
    String phoneNumber = String.format("tel: %s",
                                       editText.getText().toString());
    // Log the concatenated phone number for dialing.
    Log.d(TAG, getString(R.string.dial_number) + phoneNumber);
    Toast.makeText(this,
                   getString(R.string.dial_number) + phoneNumber,
                   Toast.LENGTH_LONG).show();
    // Create the intent.
    Intent callIntent = new Intent(Intent.ACTION_CALL);
    // Set the data for the intent as the phone number.
    callIntent.setData(Uri.parse(phoneNumber));
    // If package resolves to an app, send intent.
    if (callIntent.resolveActivity(getPackageManager()) != null) {
        startActivity(callIntent);
    } else {
        Log.e(TAG, "Can't resolve app for ACTION_CALL Intent.");
    }
}
```

2.4 Run the app

When you run the app, the app may crash with the following screen depending on whether the device or emulator has been previously set to allow the app to make phone calls:


10:40
January 17
TUESDAY, 2017
Phone Calling Sample has stopped
Open app again
1
2 ABC
3 DEF
-
4 GHI
5 JKL
6 MNO
.
7 PQRS
8 TUV
9 WXYZ
* #
0 +

In some versions of Android, this permission is turned on by default. In others, this permission is turned *off* by default.

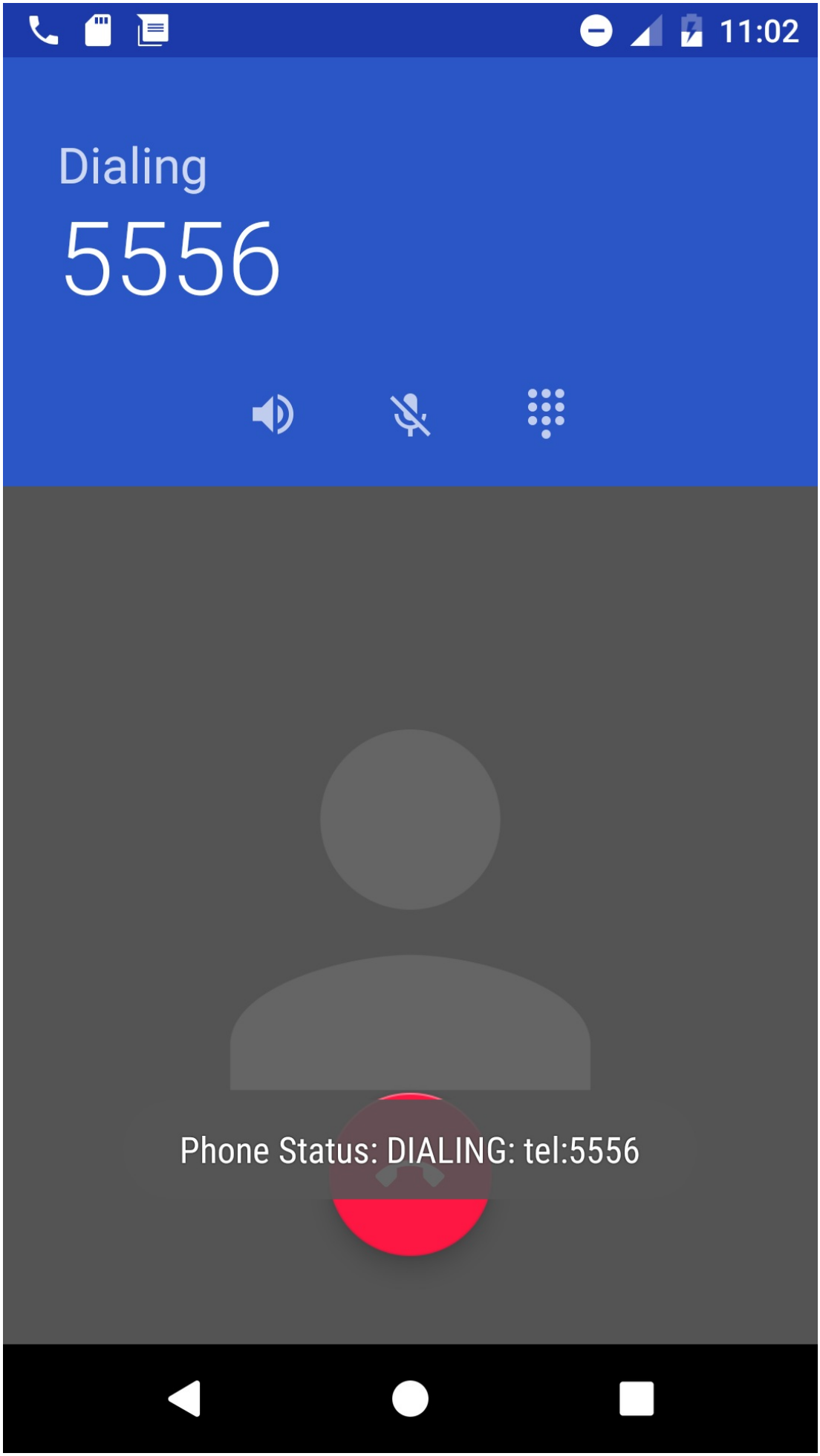
To set the app's permission on a device or emulator instance, perform the function that a user would perform: choose **Settings > Apps > Phone Calling Sample > Permissions** on the device or emulator, and turn on the Phone permission for the app. Since the user can turn on or off Phone permission at any time, you have to add a check in your app for this permission, and request it from the user if required. You will do this in the next task.

If you don't have cellular service on your device, or if telephony is not enabled, you can test the app using two emulator instances—one emulator instance calls the other one. Follow these steps:

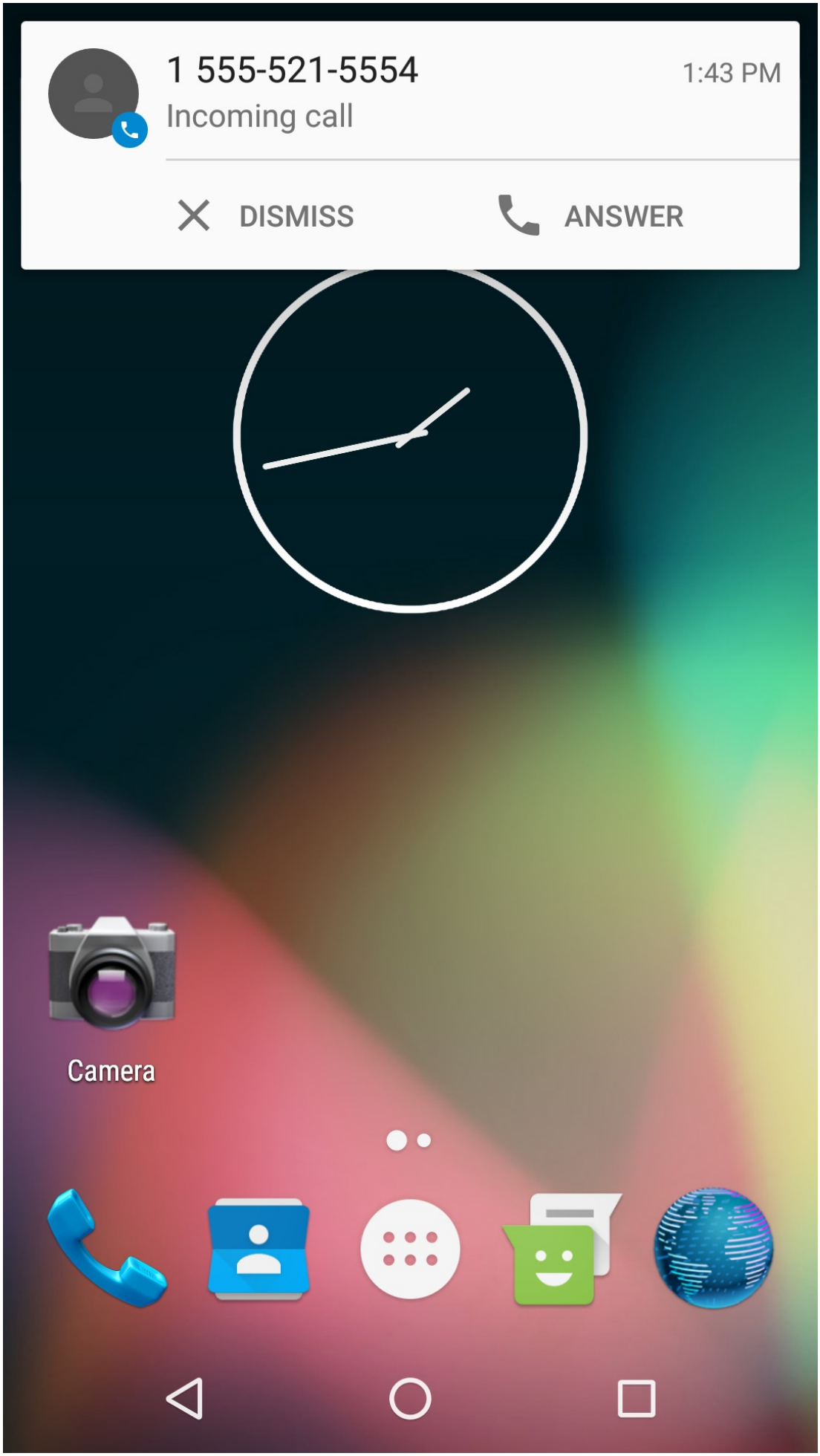
1. To launch an emulator directly from the AVD Manager, choose **Tools > Android > AVD Manager**.
2. Double-click a predefined device. Note the number that appears in the emulator's window title on the far right, as shown in the figure below as #1 (5556). This is the port number of the emulator instance.

3. Open the Android Studio project for the app, if it isn't already open.

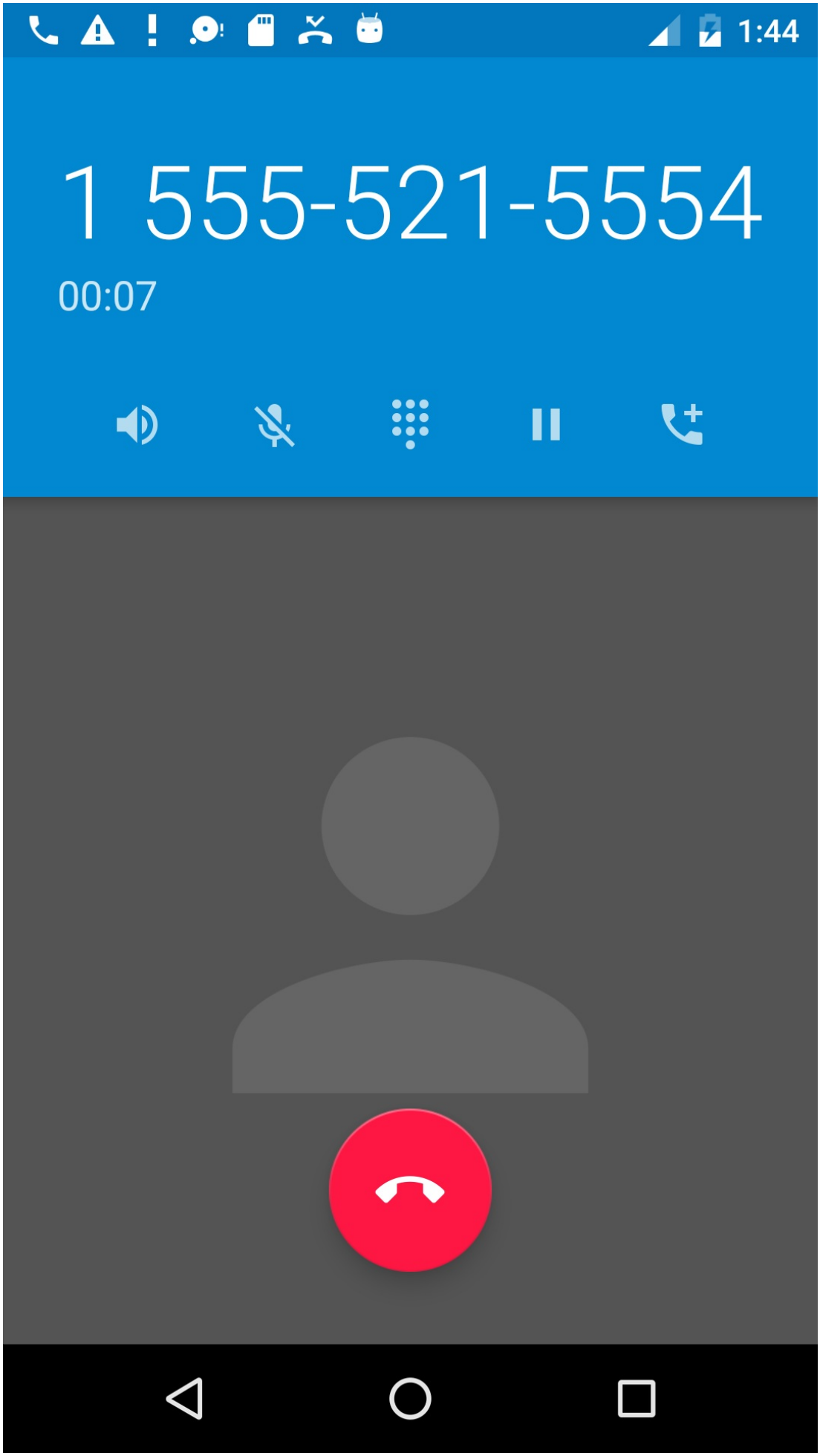
4. Run the app, but choose *another* emulator—*not* the one that is already running. Android Studio launches the other emulator.
5. In the app, enter the port number of the other emulator rather than a real phone number.
6. Click the call button in the app. The emulator shows the phone call starting up, as shown in the figure below.


11:02
Dialing
5556
Phone Status: DIALING: tel:5556

The other emulator instance should now be receiving the call, as shown in the figure below:


1 555-521-5554
1:43 PM
Incoming call
DISMISS
ANSWER
Camera

7. Click **Answer** or **Dismiss** on the emulator receiving the call. If you click **Answer**, also click the red **Hang-up** button to end the call.


1:44
1 555-521-5554
00:07

End of Part 1 - Continue with Part 2

1.2: Part 2 - Making Phone Calls

Contents:

Task 3. Check for telephony service and request permission

If telephony features are not enabled for a device, your app should detect that and disable the phone features.

In addition, your app must always get permission to use anything that is not part of the app itself. In the previous task you added the following permission to the AndroidManifest.xml file:

```
<uses-permission android:name="android.permission.CALL_PHONE" />
```

This statement enables a permission setting for this app in Settings. The user can allow or disallow this permission at any time in Settings. You can add code to request permission if the user has turned off phone permission.

3.1 Check if telephony services are enabled

1. Open the Android Studio project for the **PhoneCallingSample** app, if it isn't already open.
2. At the top of MainActivity below the class definition, define a variable for the TelephonyManager class object:

   ```
   private TelephonyManager mTelephonyManager;
   ```

3. Add the following statement to onCreate() method in MainActivity to use the string constant `TELEPHONY_SERVICE` with `getSystemService()` and assign it to

`mTelephonyManager`. This gives you access to some of the telephony features of the device.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
   // Create a telephony manager.
   mTelephonyManager = (TelephonyManager)
                               getSystemService(TELEPHONY_SERVICE);
```

4. Create a method in MainActivity to check if telephony is enabled:

```
  private boolean isTelephonyEnabled() {
      if (mTelephonyManager != null) {
          if (mTelephonyManager.getSimState() ==
                                TelephonyManager.SIM_STATE_READY) {
              return true;
          }
      }
      return false;
  }
```

5. Call the above method in the `onCreate()` method, right after assigning `mTelephonyManager`, in an `if` statement to take action if telephony is enabled. The action should be to log a message (to show that telephony is enabled), and include a comment about checking permission, which you will add in the next step. If telephony is not enabled, display a toast message, log a message, and disable the call button:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  ...
  mTelephonyManager = (TelephonyManager)
                    getSystemService(TELEPHONY_SERVICE);
  if (isTelephonyEnabled()) {
       Log.d(TAG, "Telephony is enabled");
       // ToDo: Check for phone permission.
       // ToDo: Register the PhoneStateListener.
  } else {
       Toast.makeText(this,
                    "TELEPHONY NOT ENABLED! ",
                    Toast.LENGTH_LONG).show();
       Log.d(TAG, "TELEPHONY NOT ENABLED! ");
       // Disable the call button
       disableCallButton();
  }
}
```

6. Extract the hard-coded strings in the above code to string resources:
   - "`Telephony is enabled`": `telephony_enabled`
   - `"TELEPHONY NOT ENABLED! "`: `telephony_not_enabled`
7. Create the `disableCallButton()` method in MainActivity, and code to:
   - Display a toast to notify the user that the phone feature is disabled.
   - Find and then set the call button to be invisible so that the user can't make a call.
   - If telephony is enabled (but the phone permission had not been granted), set the **Retry** button to be visible, so that the user can start the activity again and allow permission.

```
private void disableCallButton() {
  Toast.makeText(this,
            "Phone calling disabled", Toast.LENGTH_LONG).show();
  ImageButton callButton = (ImageButton) findViewById(R.id.phone_icon);
  callButton.setVisibility(View.INVISIBLE);
  if (isTelephonyEnabled()) {
      Button retryButton = (Button) findViewById(R.id.button_retry);
      retryButton.setVisibility(View.VISIBLE);
  }
}
```

8. Extract a string resource (`phone_disabled`) for the hard-coded string "Phone calling disabled" in the toast statement.
9. Create an `enableCallButton()` method in MainActivity that finds and then sets the call button to be visible:

```
private void enableCallButton() {
  ImageButton callButton = (ImageButton) findViewById(R.id.phone_icon);
  callButton.setVisibility(View.VISIBLE);
}
```

10. Create the `retryApp()` method in MainActivity that will be called when the user clicks the visible **Retry** button. Add code to:
    - Call `enableCallButton()` to enable the call button.
    - Create an intent to start (in this case, restart) the activity.

```
public void retryApp(View view) {
   enableCallButton();
   Intent intent = getPackageManager()
            .getLaunchIntentForPackage(getPackageName());
   startActivity(intent);
}
```

11. Add the `android:onClick` attribute to the **Retry** button to call retryApp:

```
<Button
   ...
   android:id="@+id/button_retry"
   ...
   android:onClick="retryApp"/>
```

3.2 Request permission for phone calling

1. At the top of MainActivity below the class definition, define a global constant for the call-phone permission request code, and set it to 1:

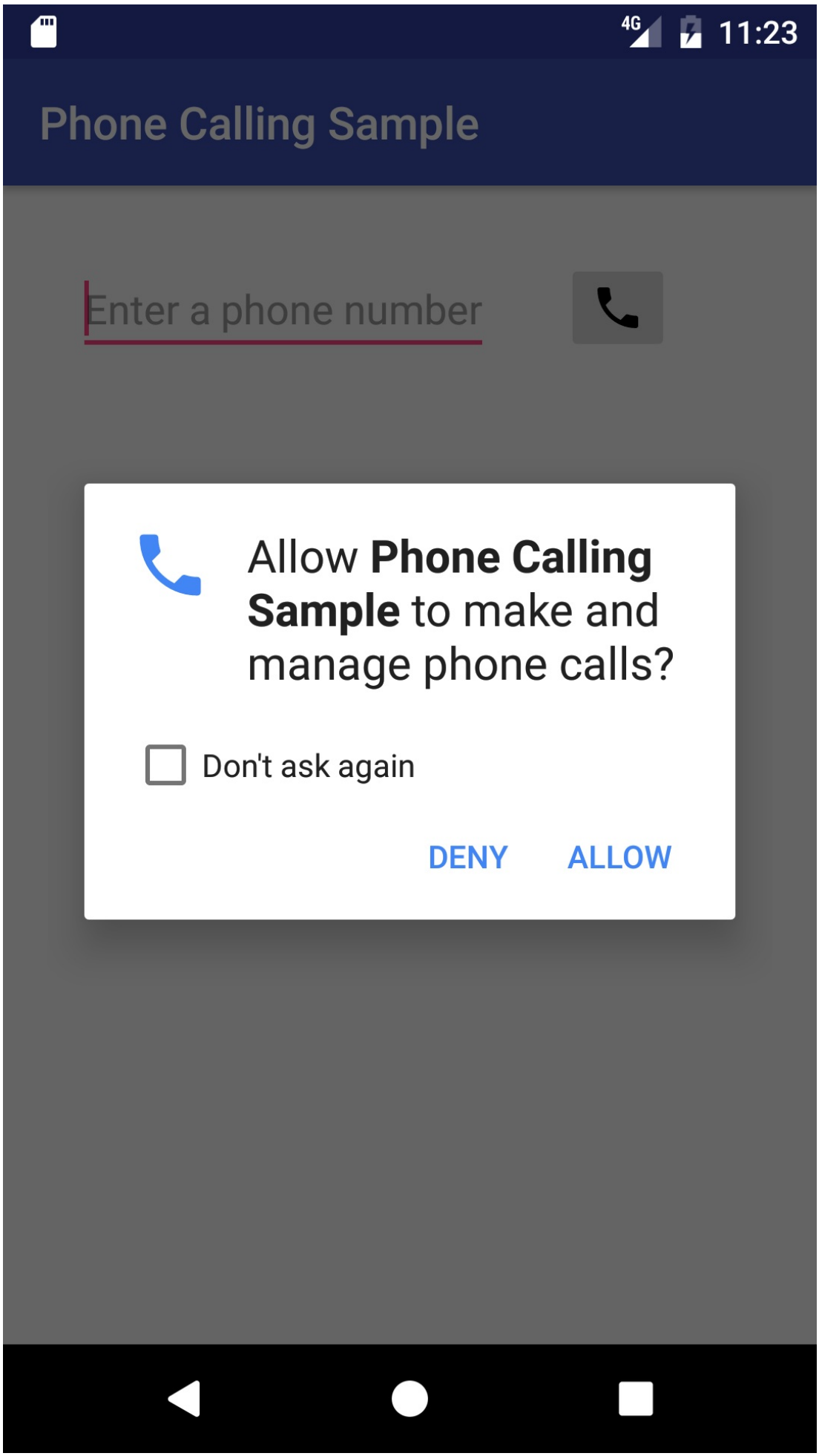
   ```
   private static final int MY_PERMISSIONS_REQUEST_CALL_PHONE = 1;
   ```

   Why the integer 1? Each permission request needs three parameters: the `context`, a string array of permissions, and an integer `requestCode`. The `requestCode` is a code attached to the request, and can be any integer that suits your use case. When a result returns back to the activity, it contains this code and uses it to differentiate multiple permission results from each other.

2. In MainActivity, create a private method called `checkForPhonePermission` to check for `CALL_PHONE` permission, which returns `void`. You put this code in a separate method because you will use it more than once:

   ```
   private void checkForPhonePermission() {
     if (ActivityCompat.checkSelfPermission(this,
                      Manifest.permission.CALL_PHONE) !=
                      PackageManager.PERMISSION_GRANTED) {
     Log.d(TAG, "PERMISSION NOT GRANTED!");
     ActivityCompat.requestPermissions(this,
                      new String[]{Manifest.permission.CALL_PHONE},
                      MY_PERMISSIONS_REQUEST_CALL_PHONE);
     } else {
         // Permission already granted. Enable the call button.
         enableCallButton();
     }
   }
   ```

3. Use `checkSelfPermission()` to determine whether your app has been granted a particular permission by the user. If permission has *not* been granted by the user, use the `requestPermissions()` method to display a standard dialog for the user to grant permission.
4. When your app calls `requestPermissions()`, the system shows a standard dialog to the user, as shown in the figure below.


4G
11:23
Phone Calling Sample
Enter a phone number
Allow **Phone Calling Sample** to make and manage phone calls?
Don't ask again
DENY
ALLOW

5. Extract the hard-coded string `"PERMISSION NOT GRANTED!"` in the above code to the string resource `permission_not_granted`.
6. In the `onCreate()` method after checking to see if telephony is enabled, add a call to `checkForPhonePermission()`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
...
if (isTelephonyEnabled()) {
    // Check for phone permission.
    checkForPhonePermission();
    // ToDo: Register the PhoneStateListener.
    ...
```

7. When the user responds to the request permission dialog, the system invokes your app's `onRequestPermissionsResult()` method, passing it the user response. Override that method to find out whether the permission was granted:

```
@Override
public void onRequestPermissionsResult(int requestCode,
                String permissions[], int[] grantResults) {
   // Check if permission is granted or not for the request.
   ...
}
```

8. For your implementation of `onRequestPermissionsResult()`, use a `switch` statement with each `case` based on the value of `requestCode`. Use one `case` to check if the permission is the one you defined as `MY_PERMISSIONS_REQUEST_CALL_PHONE`:

```
...
// Check if permission is granted or not for the request.
switch (requestCode) {
    case MY_PERMISSIONS_REQUEST_CALL_PHONE: {
        if (permissions[0].equalsIgnoreCase
                    (Manifest.permission.CALL_PHONE)
                    && grantResults[0] ==
                    PackageManager.PERMISSION_GRANTED) {
            // Permission was granted.
        } else {
            // Permission denied.
            Log.d(TAG, "Failure to obtain permission!");
            Toast.makeText(this,
                        "Failure to obtain permission!",
                        Toast.LENGTH_LONG).show();
            // Disable the call button
            disableCallButton();
        }
    }
}
```

9. Extract the hard-coded string `"Failure to obtain permission!"` in the above code to the string resource `failure_permission`, and note the following:
    - The user's response to the request dialog is returned in the `permissions` array (index `0` if only one permission is requested in the dialog). The code snippet above compares this to the corresponding grant result, which is either `PERMISSION_GRANTED` or `PERMISSION_DENIED`.
    - If the user denies a permission request, your app should take appropriate action. For example, your app might disable the functionality that depends on this permission and show a dialog explaining why it could not perform it. For now, log a debug message, display a toast to show that permission was not granted, and disable the call button with `disableCallButton()`.

3.3 Run the app and test permission

1. Run the app once. After running the app, turn *off* the Phone permission for the app on your device or emulator so that you can test the permission-request function:
    i. Choose **Settings > Apps > Phone Calling Sample > Permissions** on the device or emulator.
    ii. Turn *off* the Phone permission for the app.
2. Run the app again. You should see the request dialog in the figure in the previous section.

   i. Tap **Deny** to deny permission. The app should display a toast message showing the failure to gain permission, and the **Retry** button. The phone icon should disappear.

   ii. Tap **Retry**, and when the request dialog appears, tap **Allow**. The phone icon should reappear. Test the app's ability to make a phone call.

3. Since the user might turn off Phone permission while the app is still running, add the same permission check method to the `callNumber()` method—after the intent resolves to a package, as shown below—to check for permission right before making a call:

```
// If package resolves to an app, check for phone permission,
// and send intent.
if (callIntent.resolveActivity(getPackageManager()) != null) {
   checkForPhonePermission();
   startActivity(callIntent);
} else {
   Log.e(TAG, "Can't resolve app for ACTION_CALL Intent");
}
```

4. Run the app. If the user changes the Phone permission for the app while the app is running, the request dialog appears again for the user to **Allow** or **Deny** the permission.

   i. Click **Allow** to test the app's ability to make a phone call. The app should make the call without a problem.

   ii. Jump to the Settings app to turn off Phone permission for the app (the app should still be running):

      i. Choose **Settings > Apps > Phone Calling Sample > Permissions** on the device or emulator.

      ii. Turn *off* the Phone permission for the app.

   iii. Go back to the app and try to make a call. The request dialog should appear again. This time, Click **Deny** to deny permission to make a phone call. The app should display a toast message showing the failure to gain permission, and the **Retry** button. The phone icon should disappear.

Task 4. Monitor the phone state

You can monitor the phone state with PhoneStateListener, which monitors changes in specific telephony states. You can then show the user the state in a toast message, so that the user can see if the phone is idle or off the hook.

When the phone call finishes and the phone switches to the idle state, your app's activity resumes if the app is running on KitKat (version 19) or newer versions. However, if the app is running on a version of Android older than KitKat (version 19), the Phone app remains active. You can check the phone state and restart the activity if the state is idle.

To use PhoneStateListener, you need to register a listener object using the TelephonyManager class, which provides access to information about the telephony services on the device. Create a new class that extends PhoneStateListener to perform actions depending on the phone state. You can then register the listener object in the `onCreate()` method of the activity, using the TelephonyManager class.

4.1 Set the permission and logging tag

1. Open the Android Studio project for the **PhoneCallingSample** app, if it isn't already open.
2. Add the following `READ_PHONE_STATE` permission to the AndroidManifest.xml file after after the CALL_PHONE permission, and before the `<application>` section:

   ```
   <uses-permission android:name="android.permission.CALL_PHONE" />
   <uses-permission android:name="android.permission.READ_PHONE_STATE" />
   ```

   Monitoring the state of a phone call is permission-protected. This permission is *in addition to* the `CALL_PHONE` permission.

4.2 Create a class that extends PhoneStateListener

1. To create a listener object and listen to the phone state, create a private inner class called `MyPhoneCallListener` in MainActivity that extends PhoneStateListener.

   ```
   private class MyPhoneCallListener extends PhoneStateListener {
   ...
   }
   ```

2. Within this class, implement the `onCallStateChanged()` method of PhoneStateListener to take actions based on the phone state. The code below uses a `switch` statement with constants of the TelephonyManager class to determine which of three states the phone is in: `CALL_STATE_RINGING`, `CALL_STATE_OFFHOOK`, and `CALL_STATE_IDLE`:

```
@Override
public void onCallStateChanged(int state, String incomingNumber) {
  switch (state) {
      case TelephonyManager.CALL_STATE_RINGING:
          // Incoming call is ringing (not used for outgoing call).
          break;
      case TelephonyManager.CALL_STATE_OFFHOOK:
          // Phone call is active -- off the hook.
          break;
      case TelephonyManager.CALL_STATE_IDLE:
          // Phone is idle before and after phone call.
          break;
      default:
          // Must be an error. Raise an exception or just log it.
          break;
  }
}
```

3. Just above the `switch (state)` line, create a `String` called `message` to use in a toast as a prefix for the phone state:

```
...
// Define a string for the message to use in a toast.
String message = "Phone Status: ";
switch (state) { ...
```

4. Extract the string `"Phone Status: "` to the string resource `phone_status`.
5. For the `CALL_STATE_RINGING` state, assemble a message for logging and displaying a toast with the incoming phone number:

```
...
switch (state) {
  case TelephonyManager.CALL_STATE_RINGING:
       // Incoming call is ringing (not used for outgoing call).
       message = message + "RINGING, number: " + incomingNumber;
       Toast.makeText(MainActivity.this, message,
                      Toast.LENGTH_SHORT).show();
       Log.i(TAG, message);
       break;
  ...
```

6. Extract `"RINGING, number: "` to the string resource `ringing`.
7. Add a boolean `returningFromOffHook`, set to `false`, at the top of the `MyPhoneCallListener` declaration, in order to use it with the the `CALL_STATE_OFFHOOK` state:

```
private class MyPhoneCallListener extends PhoneStateListener {
   private boolean returningFromOffHook = false;
   ...
}
```

Tip: An app running on Android versions prior to KitKat (version 19) doesn't resume when the phone state returns to `CALL_STATE_IDLE` from `CALL_STATE_OFFHOOK` at the end of a call. The boolean `returningFromOffHook` is used as a flag, and set to `true` when the state is `CALL_STATE_OFFHOOK`, so that when the state is back to `CALL_STATE_IDLE`, the flag designates an end-of-call in order to restart the app's activity.

8. For the `CALL_STATE_OFFHOOK` state, assemble a message for logging and displaying a toast, and set the `returningFromOffHook` boolean to `true`.

```
...
switch (state) {
     case TelephonyManager.CALL_STATE_OFFHOOK:
               // Phone call is active -- off the hook.
               message = message + "OFFHOOK";
               Toast.makeText(MainActivity.this, message,
                                    Toast.LENGTH_SHORT).show();
               Log.i(TAG, message);
               returningFromOffHook = true;
               break;
     ...
```

9. Extract `"OFFHOOK"` to the string resource `offhook`.
10. For the `CALL_STATE_IDLE` state, log and display a toast, and check if `returningFromOffHook` is `true`; if so, restart the activity if the version of Android is earlier than KitKat.

```
...
switch (state) {
        case TelephonyManager.CALL_STATE_IDLE:
           // Phone is idle before and after phone call.
           // If running on version older than 19 (KitKat),
           // restart activity when phone call ends.
           message = message + "IDLE";
           Toast.makeText(MainActivity.this, message,
                          Toast.LENGTH_SHORT).show();
           Log.i(TAG, message);
           if (returningFromOffHook) {
               // No need to do anything if >= version KitKat.
               if (Build.VERSION.SDK_INT < Build.VERSION_CODES.KITKAT) {
                   Log.i(TAG, "Restarting app");
                   // Restart the app.
                   Intent intent = getPackageManager()
                       .getLaunchIntentForPackage(
                       .getPackageName());
                    intent.addFlags
                       (Intent.FLAG_ACTIVITY_CLEAR_TOP);
                    startActivity(intent);
               }
            }
            break;
          ...
```

If the app is running on KitKat (version 19) or newer versions, there is no need to restart the activity after the phone call ends. But if the app is running on a version of Android older than KitKat (version 19), the code must restart the current activity so that the user can return to the app after the call ends.

Tip: The code also sets `FLAG_ACTIVITY_CLEAR_TOP` so that instead of launching a new instance of the current activity, any other activities on top of the current activity are closed and an intent is delivered to the (now on top) current activity. This flag helps you manage a stack of activities in an app.

11. Extract `"IDLE"` to the string resource `idle`, and extract `"Restarting app"` to the string resource `restarting_app`.

The code below shows the entire `onCallStateChanged()` method:

```
...
@Override
public void onCallStateChanged(int state, String incomingNumber) {
    // Define a string for the message to use in a toast.
    String message = getString(R.string.phone_status);
    switch (state) {
        case TelephonyManager.CALL_STATE_RINGING:
            // Incoming call is ringing (not used for outgoing call).
```

```
            message = message +
                            getString(R.string.ringing) + incomingNumber;
            Toast.makeText(MainActivity.this, message,
                            Toast.LENGTH_SHORT).show();
            Log.i(TAG, message);
            break;
        case TelephonyManager.CALL_STATE_OFFHOOK:
            // Phone call is active -- off the hook.
            message = message + getString(R.string.offhook);
            Toast.makeText(MainActivity.this, message,
                            Toast.LENGTH_SHORT).show();
            Log.i(TAG, message);
            returningFromOffHook = true;
            break;
        case TelephonyManager.CALL_STATE_IDLE:
            // Phone is idle before and after phone call.
            // If running on version older than 19 (KitKat),
            // restart activity when phone call ends.
            message = message + getString(R.string.idle);
            Toast.makeText(MainActivity.this, message,
                            Toast.LENGTH_SHORT).show();
            Log.i(TAG, message);
            if (returningFromOffHook) {
                // No need to do anything if >= version KitKat.
                if (Build.VERSION.SDK_INT < Build.VERSION_CODES.KITKAT) {
                    Log.i(TAG, getString(R.string.restarting_app));
                    // Restart the app.
                    Intent intent = getPackageManager()
                                    .getLaunchIntentForPackage(
                                    .getPackageName());
                    intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
                    startActivity(intent);
                }
            }
            break;
        default:
            message = message + "Phone off";
            Toast.makeText(MainActivity.this, message,
                                    Toast.LENGTH_SHORT).show();
            Log.i(TAG, message);
            break;
    }
}
...
```

4.3 Register the PhoneStateListener

1. At the top of MainActivity below the class definition, define a variable for the PhoneStateListener:

```
private MyPhoneCallListener mListener;
```

2. In the `onCreate()` method, add the following code after checking for telephony and permission:

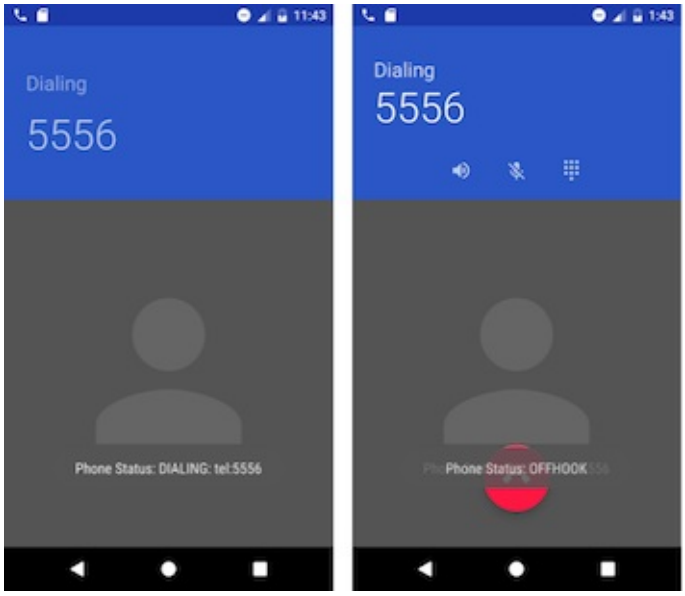
```
...
if (isTelephonyEnabled()) {
   ...
   checkForPhonePermission();
   // Register the PhoneStateListener to monitor phone activity.
   mListener = new MyPhoneCallListener();
   telephonyManager.listen(mListener,
                PhoneStateListener.LISTEN_CALL_STATE);
} else { ...
```

3. You must also unregister the listener in the activity's onDestroy() method. Override the `onDestroy()` method by adding the following code:

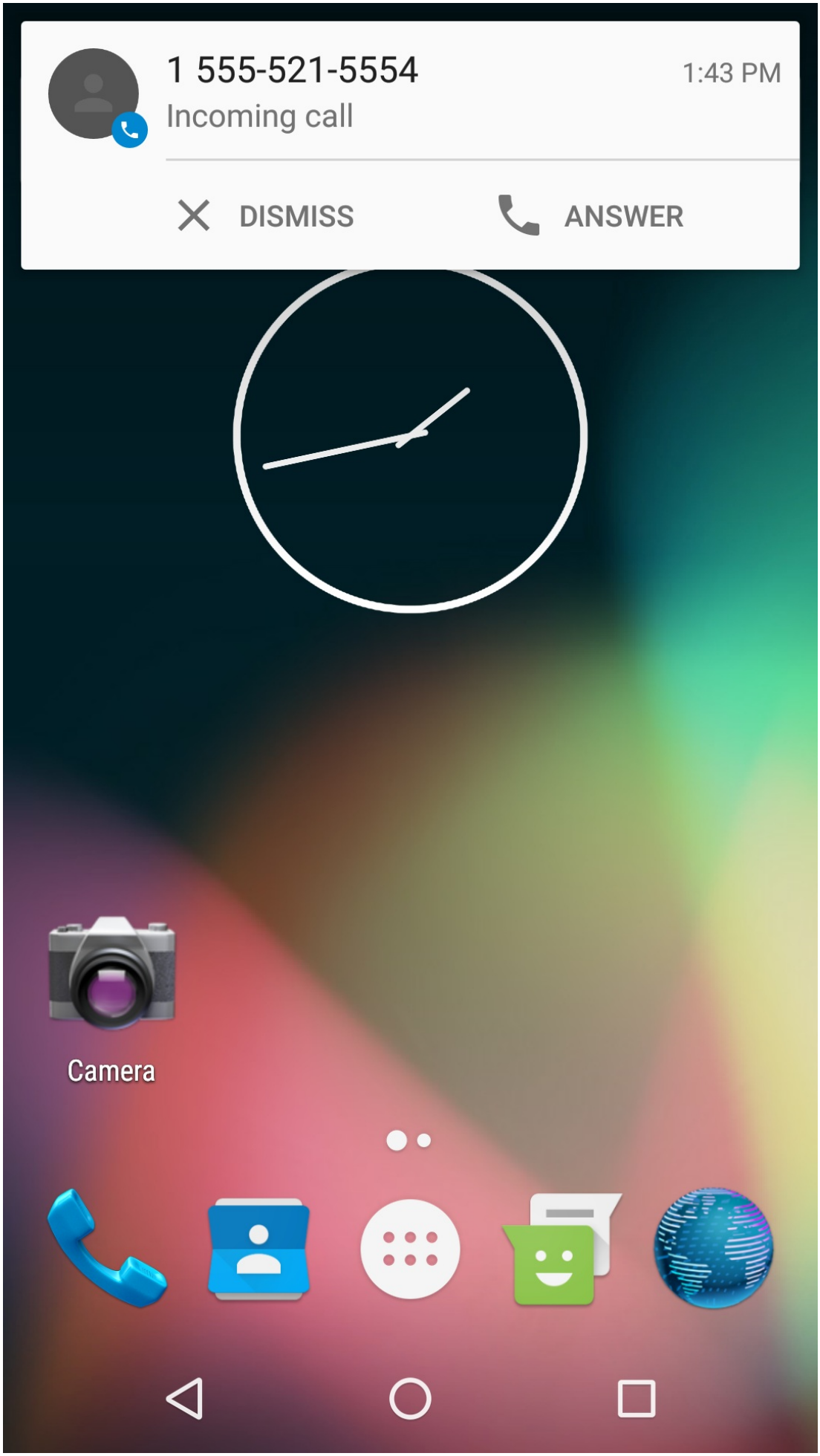
```
@Override
protected void onDestroy() {
   super.onDestroy();
   if (isTelephonyEnabled()) {
       telephonyManager.listen(mListener,
                           PhoneStateListener.LISTEN_NONE);
   }
}
```

4.4 Run the app

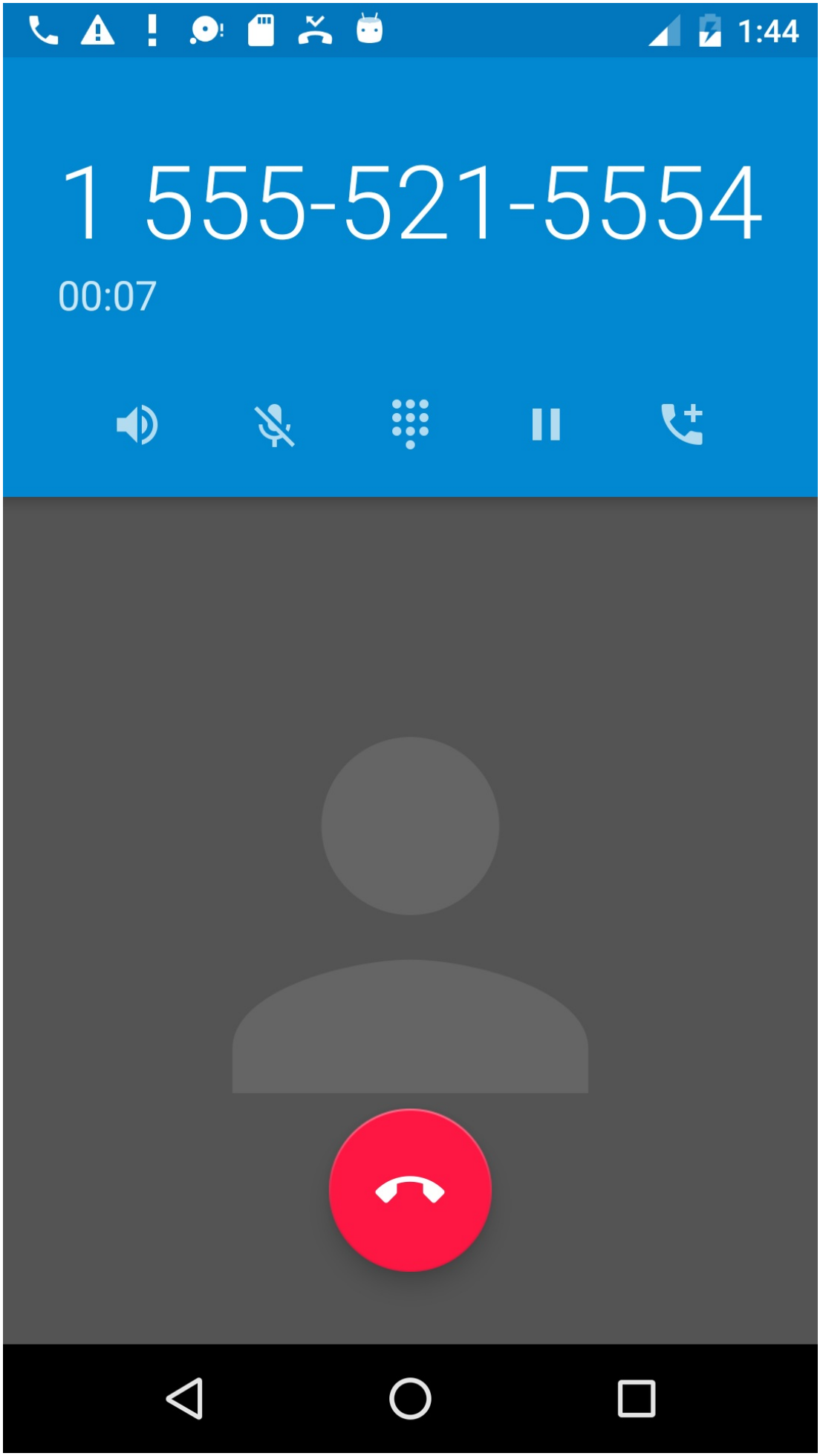
1. Run the app. If the user changes the Phone permission for the app while the app is running, the request dialog appears again for the user to **Allow** or **Deny** the permission. Click **Allow** to test the app's ability to make a phone call.
2. After entering a phone number and clicking the call button, the emulator or device shows the phone call starting up, as shown in the figure below. A toast message appears showing the phone number (left side of figure), and the toast message changes to show a new status of "OFFHOOK" (right side of figure) after the call has started.



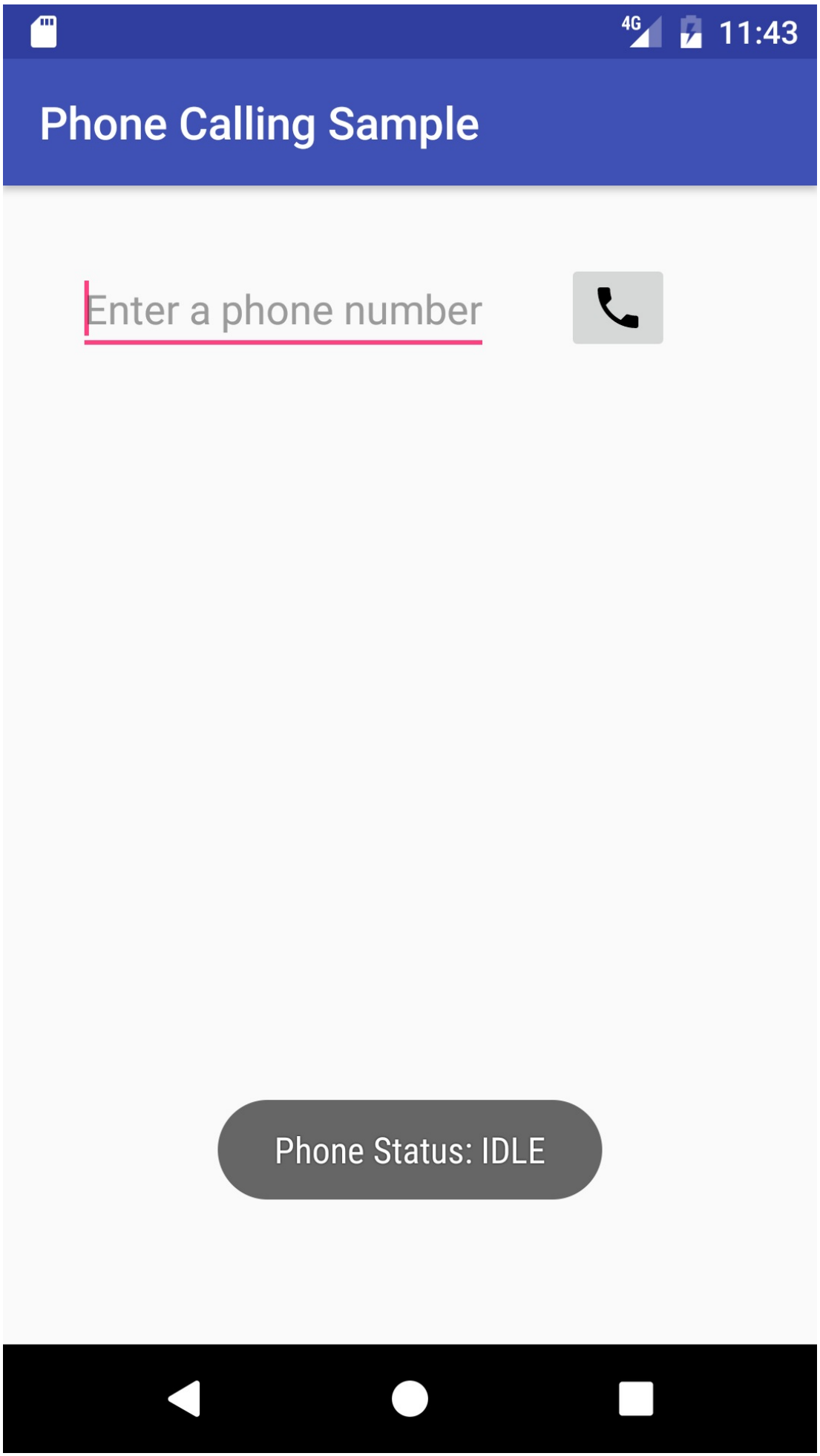
3. The other emulator instance or device should now be receiving the call, as shown in the figure below. Click **Answer** or **Dismiss** on the device or emulator receiving the call.


1 555-521-5554
1:43 PM
Incoming call
DISMISS
ANSWER
Camera

4. If you click **Answer**, be sure to also click the red **Hang-up** button to finish the call, as shown in the figure below.


1:44
1 555-521-5554
00:07

After you hang up, the app should reappear with a toast message showing that the phone is now in the idle state, as shown in the figure below.


4G
11:43
Phone Calling Sample
Enter a phone number
Phone Status: IDLE

Solution code

Android Studio project: PhoneCallingSample

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.
Challenge:

1. Use the normalizeNumber() method in the PhoneNumberUtils class to remove characters other than digits from the phone number after the user has entered it. This method was added to API level 21. If you need your app to run on older versions, include a check for the version that uses the normalizeNumber() method only if the version is older than Lollipop. Your app already uses a log statement to show the phone number as dialed, so if the user enters "1-415-555-1212" the log message should show that the number was normalized:

   ```
   D/MainActivity: Phone Status: DIALING: tel: 14155551212
   ```

2. Add an invisible TextView to the PhoneCallingSample app. This TextView should appear below the invisible **Retry** button, but *only* when the phone is ringing (indicating an incoming call), and it should show the phone number of the caller.

If you have both emulators open as described previously, install the app on both emulators. You can then test an incoming call by using the app on one emulator to call the other emulator.

Tip: You can also emulate receiving a call by clicking the **…** (More) icon at the bottom of the emulator's toolbar on the right side. Click **Phone** in the left column to see the extended phone controls, and click **Call Device** to call the emulator.

Android Studio project: PhoneCallingSampleChallenge

Summary

- To send an intent to the Phone app with a phone number, your app needs to prepare a URI for the phone number as a string prefixed by "tel:" (for example tel:14155551212).
- To dial a phone number, create an implicit intent with `ACTION_DIAL`, and set the phone number URI as the data for the intent with `setData()`:

```
Intent callIntent = new Intent(Intent.ACTION_DIAL);
callIntent.setData(Uri.parse(phoneNumber));
```

- For phone permission, add the following to the AndroidManifest.xml file:

```
<uses-permission android:name="android.permission.CALL_PHONE" />
```

- To make a phone call, create an implicit intent with `ACTION_CALL`, and set the phone number URI as the data for the intent with `setData()`:

```
Intent callIntent = new Intent(Intent.ACTION_CALL);
callIntent.setData(Uri.parse(phoneNumber));
```

- To check if telephony is enabled, use the string constant `TELEPHONY_SERVICE` with `getSystemService()` to retrieve a TelephonyManager, which gives you access to telephony features.
- Use `checkSelfPermission()` to determine whether your app has been granted a particular permission by the user. If permission has *not* been granted, use the `requestPermissions()` method to display a standard dialog for the user to grant permission.
- To monitor the phone state with PhoneStateListener, register a listener object using the TelephonyManager class.
- For phone monitoring permission, add the following to the AndroidManifest.xml file:

```
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
```

- To monitor phone states, create a private class that extends PhoneStateListener, and override the `onCallStateChanged()` method of PhoneStateListener to take different actions based on the phone state: `CALL_STATE_RINGING`, `CALL_STATE_OFFHOOK`, or `CALL_STATE_IDLE`.

Related concept

- Phone Calls

Learn more

- Android developer documentation:
  - Common Intents
  - TelephonyManager

    - PhoneStateListener
    - Requesting Permissions at Run Time
    - checkSelfPermission
    - Run Apps on the Android Emulator
    - Intents and Intent Filters
    - Intent
- Stack Overflow:
    - How to format a phone number using PhoneNumberUtils?
    - How to make phone call using intent in android?
    - Ringing myself using android emulator
    - Fake Incoming Call Android
    - Simulating incoming call or sms in Android Studio
- Other:
    - User (beginner) tutorial: How to Make Phone Calls with Android
    - Developer Video: How to Make a Phone Call

2: SMS Messages

Contents:

Android devices can send and receive messages to or from any other phone that supports Short Message Service (SMS). Android offers the Messenger app that can send and receive SMS messages. A host of third-party apps for sending and receiving SMS messages are also available in Google Play.

This chapter describes how to use SMS in your app. You can add code to your app to:

- Launch an SMS messaging app from your app to handle all SMS communication.
- Send an SMS message from within your app.
- Receive SMS messages in your app.

Note: The SMS protocol was primarily designed for user-to-user communication and is not well-suited for apps that want to transfer data. You should not use SMS to send data messages from a web server to your app on a user device. SMS is neither encrypted nor strongly authenticated on either the network or the device.

Sending and receiving SMS messages

Access to the SMS features of an Android device is protected by user permissions. Just as your app needs the user's permission to use phone features, so also does an app need the user's permission to directly use SMS features.

However, your app doesn't need permission to pass a phone number to an installed SMS app, such as Messenger, for sending the message. The Messenger app itself is governed by user permission.

You have two choices for *sending* SMS messages:

- Use an implicit Intent to launch a messaging app such as Messenger, with the ACTION_SENDTO action.

    - This is the simplest choice for sending messages. The user can add a picture or other attachment in the messaging app, if the messaging app supports adding attachments.
    - Your app doesn't need code to request permission from the user.
    - If the user has multiple SMS messaging apps installed on the Android phone, the App chooser will appear with a list of these apps, and the user can choose which one to use. (Android smartphones will have at least one, such as Messenger.)
    - The user can change the message in the messaging app before sending it.
    - The user navigates back to your app using the **Back** button.
- Send the SMS message using the sendTextMessage() method or other methods of the SmsManager class.
    - This is a good choice for sending messages from your app without having to use another installed app.
    - Your code must ask the user for permission before sending the message if the user hasn't already granted permission.
    - The user stays in your app during and after sending the message.
    - You can manage SMS operations such as dividing a message into fragments, sending a multipart message, get carrier-dependent configuration values, and so on.

To *receive* SMS messages, the best practice is to use the onReceive() method of the BroadcastReceiver class. The Android framework sends out system broadcasts of events such as receiving an SMS message, containing intents that are meant to be received using a BroadcastReceiver. Your app receives SMS messages by listening for the SMS_RECEIVED_ACTION broadcast.

Most smartphones and mobile phones support what is known as "PDU mode" for sending and receiving SMS. *PDU* (protocol data unit) contains not only the SMS message, but also metadata about the SMS message, such as text encoding, the sender, SMS service center address, and much more. To access this metadata, SMS apps almost always use PDUs to encode the contents of a SMS message. The sendTextMessage() and sendMultimediaMessage() methods of the SmsManager class encode the contents for you. When receiving a PDU, you can create an SmsMessage object from the raw PDU using createFromPdu().

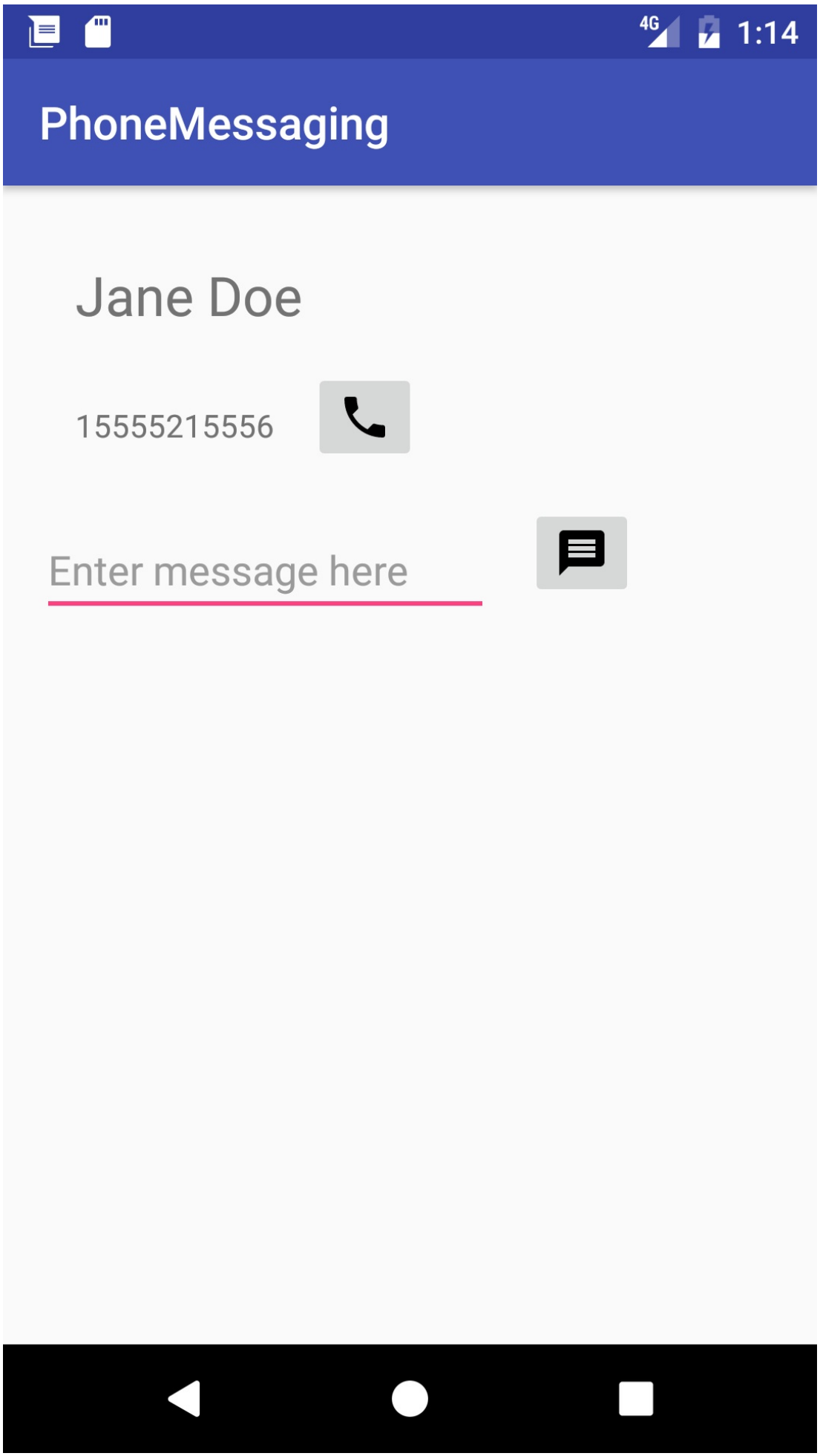
Using an intent to launch an SMS app

To use an Intent to launch an SMS app, your app needs to prepare a Uniform Resource Identifier (URI) for the phone number as a string prefixed by "smsto:" (as in `smsto:14155551212`). You can use a hardcoded phone number, such as the phone number of

a support message center, or provide an EditText field in the layout to enable the user to enter a phone number.

Tip: For details about using methods in the PhoneNumberUtils class to format a phone number string, see the related concept Phone Calls.

Use a button (such as an ImageButton) that the user can tap to pass the phone number to the SMS app. For example, an app that enables a user make a phone call and/or send a message to the phone number might offer a simple layout with a phone icon button for calling, and a messaging icon button for sending a message, as shown in the figure below.


4G
1:14
PhoneMessaging
Jane Doe
15555215556
Enter message here

To call a method such as `smsSendMessage()` that would launch a messaging app with a phone number, you can add the `android:onClick` attribute to the button for sending a message:

```
<ImageButton
    ...
    android:onClick="smsSendMessage"/>
```

In the `smsSendMessage()` method, you would convert the phone number to a string prefixed by "smsto:" (as in `smsto:14155551212`). Use an implicit intent with `ACTION_SENDTO` to pass the phone number to the SMS app, and set the phone number and message for the intent with `setData()` and `putExtra`.

Tip: The "smsto:" prefix with `ACTION_SENDTO` ensures that your intent is handled only by a text messaging app (and not by other email or social apps).

If the user has several SMS messaging apps, the user can choose which one to open. The SMS app opens with the supplied phone number and message, enabling the user to tap a button to send the message, or change the number and message before sending. The SMS app then sends the message.

The following code demonstrates how to perform an implicit intent to send a message:

```
public void smsSendMessage(View view) {
    // Find the TextView number_to_call and assign it to textView.
    TextView textView = (TextView) findViewById(R.id.number_to_call);
    // Concatenate "smsto:" with phone number to create smsNumber.
    String smsNumber = "smsto:" + textView.getText().toString();
    // Find the sms_message view.
    EditText smsEditText = (EditText) findViewById(R.id.sms_message);
    // Get the text of the sms message.
    String sms = smsEditText.getText().toString();
    // Create the intent.
    Intent smsIntent = new Intent(Intent.ACTION_SENDTO);
    // Set the data for the intent as the phone number.
    smsIntent.setData(Uri.parse(smsNumber));
    // Add the message (sms) with the key ("sms_body").
    smsIntent.putExtra("sms_body", sms);
    // If package resolves (target app installed), send intent.
    if (smsIntent.resolveActivity(getPackageManager()) != null) {
        startActivity(smsIntent);
    } else {
        Log.e(TAG, "Can't resolve app for ACTION_SENDTO Intent.");
    }
}
```

Note the following in the above code:

- The method gets the phone number from the `number_to_call` TextView, and concatenates it with the `smsto:` prefix (as in `smsto:14155551212`) before assigning it to `smsNumber`. It also gets the message entered into the EditText view.
- To launch an SMS messaging app, use an implicit intent (`smsIntent`) with `ACTION_SENDTO`, and set the phone number and message for the intent with `setData()` and `putExtra`.

  The putExtra() method needs two strings: the key identifying the type of data (`"sms_body"`) and the data itself, which is the text of the message (`sms`). For more information about common intents and the `putExtra()` method, see Common Intents: Text Messaging.

- You need to supply a check to see if the implicit intent resolves to a package (an app), and if it does, you need to start the `smsIntent` activity. If it doesn't, display a log message about the failure.

Sending SMS messages from your app

To send an SMS message from your app, use the sendTextMessage() method of the SmsManager class. Perform these steps to enable sending messages from within your app:

1. Add the `SEND_SMS` permission to send SMS messages.
2. Check to see if the user continues to grant permission. If not, request permission.
3. Use the `sendTextMessage()` method of the SmsManager class.

Checking for user permission

Beginning in Android 6.0 (API level 23), users grant permissions to apps while the app is running, not when they install the app. This approach streamlines the app install process, since the user does not need to grant permissions when they install or update the app. It also gives the user more control over the app's functionality. However, your app must check for permission every time it does something that requires permission (such as sending an SMS message). If the user has used the Settings app to turn off SMS permissions for the app, your app can display a dialog to request permission.

Tip: For a complete description of the request permission process, see Requesting Permissions at Run Time.

Add the `SEND_SMS` permission to the AndroidManifest.xml file after the first line (with the `package` definition) and before the `<application>` section:

```
<uses-permission android:name="android.permission.SEND_SMS" />
```

Because the user can turn permissions on or off for each app, your app must check whether it still has permission every time it does something that requires permission (such as sending an SMS message). If the user has turned SMS permission off for the app, your app can display a dialog to request permission.

Follow these steps:

1. At the top of the activity that sends an SMS message, and below the activity's class definition, define a constant variable to hold the request code, and set it to an integer:

   ```
   private static final int MY_PERMISSIONS_REQUEST_SEND_SMS = 1;
   ```

   Why the integer 1? Each permission request needs three parameters: the `context`, a string array of permissions, and an integer `requestCode`. The `requestCode` is the integer attached to the request. When a result returns in the activity, it contains this code and uses it to differentiate multiple permission results from each other.

2. In the activity that makes a phone call, create a method that uses the checkSelfPermission() method to determine whether your app has been granted the permission:

   ```
   private void checkForSmsPermission() {
      if (ActivityCompat.checkSelfPermission(this,
                      Manifest.permission.SEND_SMS) !=
                      PackageManager.PERMISSION_GRANTED) {
          Log.d(TAG, getString(R.string.permission_not_granted));
          // Permission not yet granted. Use requestPermissions().
          // MY_PERMISSIONS_REQUEST_SEND_SMS is an
          // app-defined int constant. The callback method gets the
          // result of the request.
          ActivityCompat.requestPermissions(this,
                      new String[]{Manifest.permission.SEND_SMS},
                      MY_PERMISSIONS_REQUEST_SEND_SMS);
      } else {
          // Permission already granted. Enable the message button.
          enableSmsButton();
      }
   }
   ```

   The code uses `checkSelfPermission()` to determine whether your app has been granted a particular permission by the user. If permission has *not* been granted, the code uses the requestPermissions() method to display a standard dialog for the user to grant permission.

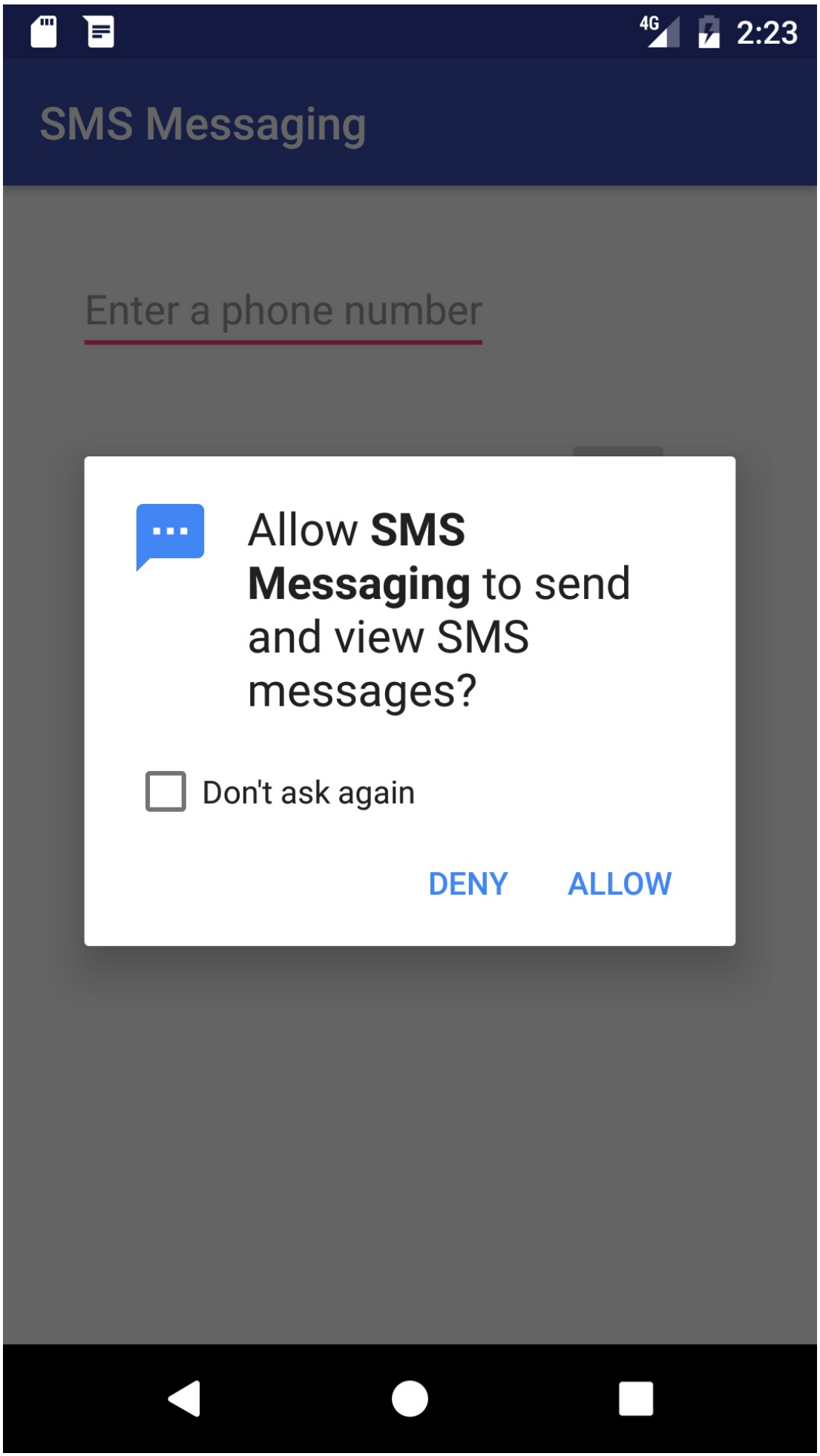
Use your `checkForSmsPermission()` method to check for permission at the following times:

- When the activity starts—in its `onCreate()` method.
- Every time before sending a message. Since the user might turn off the SMS permission while the app is still running, call the `checkForSmsPermission()` method in the `smsSendMessage()` method before using the `SmsManager` class.

Requesting user permission

If permission has *not* been granted by the user, use the requestPermissions() method of the `ActivityCompat` class. The `requestPermissions()` method needs three parameters: the context (`this`), a string array of permissions (`new String[] {Manifest.permission.SEND_SMS}`), and the predefined integer `MY_PERMISSIONS_REQUEST_SEND_SMS` for the `requestCode` .

When your app calls `requestPermissions()` , the system shows a standard dialog to the user, as shown in the figure below.


2:23
SMS Messaging
Enter a phone number
Allow SMS Messaging to send and view SMS messages?
Don't ask again
DENY
ALLOW

When the user responds to the request permission dialog by tapping **Deny** or **Allow**, the system invokes the onRequestPermissionsResult() method, passing it the user response. Your app has to override that method to find out whether the permission was granted.

The following code demonstrates how you can use a `switch` statement in your implementation of `onRequestPermissionsResult()` based on the value of `requestCode`. The user's response to the request dialog is returned in the `permissions` array (index `0` if only one permission is requested in the dialog). This is compared to the corresponding grant result, which is either `PERMISSION_GRANTED` or `PERMISSION_DENIED`.

If the user denies a permission request, your app should disable the functionality that depends on this permission and show a dialog explaining why it could not perform it. The code below logs a debug message, displays a toast to show that permission was not granted, and disables the message icon used as a button.

```
@Override
public void onRequestPermissionsResult(int requestCode,
                    String permissions[], int[] grantResults) {
    switch (requestCode) {
        case MY_PERMISSIONS_REQUEST_SEND_SMS: {
            if (permissions[0].equalsIgnoreCase(Manifest.permission.SEND_SMS)
                        && grantResults[0] ==
                        PackageManager.PERMISSION_GRANTED) {
                // Permission was granted.
            } else {
                // Permission denied.
                Log.d(TAG, getString(R.string.failure_permission));
                Toast.makeText(MainActivity.this,
                            getString(R.string.failure_permission),
                                  Toast.LENGTH_SHORT).show();
                // Disable the message button.
                disableSmsButton();
            }
        }
    }
}
```

Using SmsManager to send the message

Use the sendTextMessage() method of the SmsManager class to send the message, which takes the following parameters:

- `destinationAddress`: The string for the phone number to receive the message.
- `scAddress`: A string for the service center address, or `null` to use the current default SMSC. A Short Message Service Center (SMSC) is a network element in the mobile telephone network. The mobile network operator usually presets the correct service

center number in the default profile of settings stored in the device's SIM card.

Tip: You can find the default SMSC in an Android smartphone's hidden menu. Open the Phone app and dial **##4636##** to open the testing menu. Tap **Phone information**, and scroll to the bottom. The SMSC number should appear blank. Tap **Refresh** to see the number.

- `smsMessage` : A string for the body of the message to send.
- `sentIntent` : A PendingIntent. If not `null` , this is broadcast when the message is successfully sent or if the message failed.
- `deliveryIntent` : A PendingIntent. If not `null` , this is broadcast when the message is delivered to the recipient.

Follow these steps to use the sendTextMessage() method:

1. Create an onClick handler for a button that sends the message.
2. Get the strings for the phone number (`destinationAddress)` and the message (`smsMessage)` .
3. Declare additional string and `PendingIntent` parameters:

```
...
// Set the service center address if needed, otherwise null.
String scAddress = null;
// Set pending intents to broadcast
// when message sent and when delivered, or set to null.
PendingIntent sentIntent = null, deliveryIntent = null;
...
```

4. Use the SmsManager class to create `smsManager` , which automatically imports `android.telephony.SmsManager` , and use `sendTextMessage()` to send the message:

```
...
// Use SmsManager.
SmsManager smsManager = SmsManager.getDefault();
smsManager.sendTextMessage
       (destinationAddress, scAddress, smsMessage,
                       sentIntent, deliveryIntent);
...
```

The following code snippet shows a sample onClick handler for sending a message:

```
public void smsSendMessage(View view) {
    EditText editText = (EditText) findViewById(R.id.editText_main);
    // Set the destination phone number to the string in editText.
    String destinationAddress = editText.getText().toString();
    // Find the sms_message view.
    EditText smsEditText = (EditText) findViewById(R.id.sms_message);
    // Get the text of the SMS message.
    String smsMessage = smsEditText.getText().toString();
    // Set the service center address if needed, otherwise null.
    String scAddress = null;
    // Set pending intents to broadcast
    // when message sent and when delivered, or set to null.
    PendingIntent sentIntent = null, deliveryIntent = null;
    // Check for permission first.
    checkForSmsPermission();
    // Use SmsManager.
    SmsManager smsManager = SmsManager.getDefault();
    smsManager.sendTextMessage
                (destinationAddress, scAddress, smsMessage,
                            sentIntent, deliveryIntent);
}
```

Receiving SMS messages

To receive SMS messages, use the onReceive() method of the BroadcastReceiver class. The Android framework sends out system broadcasts of events such as `SMS_RECEIVED` for receiving an SMS message. You must also include `RECEIVE_SMS` permission in your project's AndroidManifest.xml file:

```
<uses-permission android:name="android.permission.RECEIVE_SMS" />
```

To use a broadcast receiver:

1. Add the broadcast receiver by choosing **File > New > Other > Broadcast Receiver**. The `<receiver...</receiver>` tags are automatically added to the AndroidManifest.xml file.
2. Register the receiver by adding an *intent filter* within the `<receiver...</receiver>` tags to specify the type of broadcast intent you want to receive.
3. Implement the `onReceive()` method.

Adding a broadcast receiver

You can perform the first step by selecting the package name in the Project:Android: view and choosing **File > New > Other > Broadcast Receiver**. Make sure "Exported" and "Enabled" are checked. The "Exported" option allows your app to respond to outside

broadcasts, while "Enabled" allows it to be instantiated by the system.

Android Studio automatically generates a `<receiver>` tag in the app's AndroidManifest.xml file, with your chosen options as attributes:

```
<receiver
    android:name="com.example.android.phonecallingsms.MySmsReceiver"
    android:enabled="true"
    android:exported="true">
</receiver>
```

Registering the broadcast receiver

In order to receive any broadcasts, you must register for specific broadcast intents. In the Intent documentation, under "Standard Broadcast Actions", you can find some of the common broadcast intents sent by the system.

The following intent filter registers the receiver for the `android.provider.Telephony.SMS_RECEIVED` intent:

```
<receiver
    android:name="com.example.android.smsmessaging.MySmsReceiver"
    android:enabled="true"
    android:exported="true">
    <intent-filter>
         <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
    </intent-filter>
</receiver>
```

Implementing the onReceive() method

Once your app's BroadcastReceiver intercepts a broadcast it is registered for (`SMS_RECEIVED`), the intent is delivered to the receiver's `onReceive()` method, along with the context in which the receiver is running.

The following shows the first part of the `onReceive()` method, which does the following:

- Retrieves the extras (the SMS message) from the intent.
- Stores it in a `bundle` .
- Defines the `msgs` array and `strMessage` string.
- Gets the `format` for the message from the `bundle` in order to use it with createFromPdu() to create the SmsMessage.

The `format` is the message's mobile telephony system format passed in an `SMS_RECEIVED_ACTION` broadcast. It is usually "3gpp" for GSM/UMTS/LTE messages in the 3GPP format, or "3gpp2" for CDMA/LTE messages in 3GPP2 format.

```
@Override
public void onReceive(Context context, Intent intent) {
    // Get the SMS message.
    Bundle bundle = intent.getExtras();
    SmsMessage[] msgs;
    String strMessage = "";
    String format = bundle.getString("format");
    // Retrieve the SMS message received.
    ...
}
```

The `onReceive()` method then retrieves from the bundle one or more pieces of data in the PDU:

```
...
// Retrieve the SMS message received.
Object[] pdus = (Object[]) bundle.get("pdus");
if (pdus != null) {
    // Fill the msgs array.
    msgs = new SmsMessage[pdus.length];
    for (int i = 0; i < msgs.length; i++) {
        ...
```

Use createFromPdu(byte[] pdu, String format) to fill the `msgs` array for Android version 6.0 (Marshmallow) and newer versions. For earlier versions of Android, use the deprecated signature createFromPdu(byte[] pdu):

- For Android version 6.0 (Marshmallow) and newer versions, use the following, which includes `format`:

  ```
  msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i], format);
  ```

- For earlier versions of Android, use the following:

  ```
  msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
  ```

The `onReceive()` method then builds the `strMessage` to show in a toast message. It gets the originating address using the `getOriginatingAddress()` method, and the message body using the `getMessageBody()` method.

```
...
// Build the message to show.
strMessage += "SMS from " + msgs[i].getOriginatingAddress();
strMessage += " :" + msgs[i].getMessageBody() + "\n";
...
```

The following shows the complete `onReceive()` method for SMS messages:

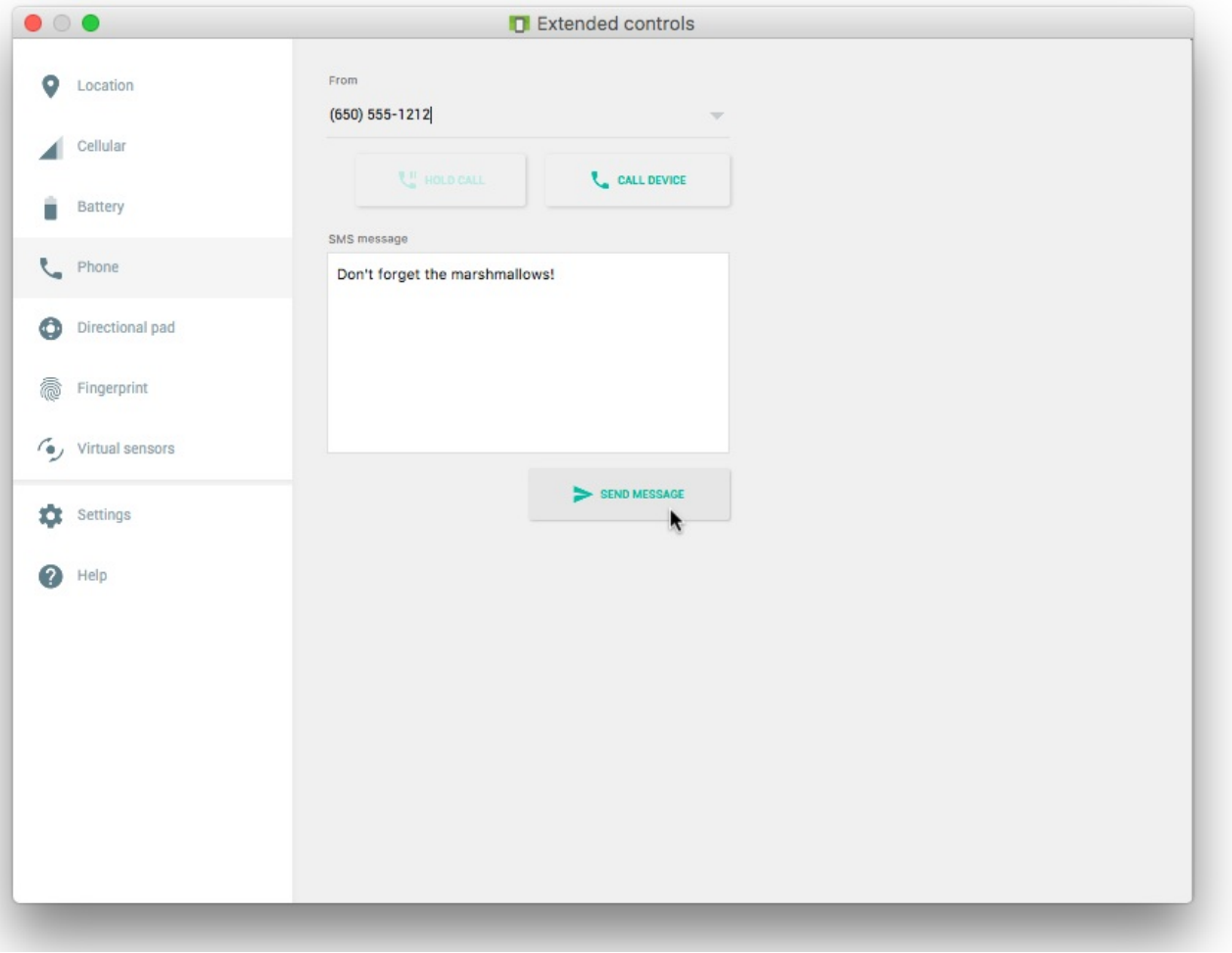
```
@TargetApi(Build.VERSION_CODES.M)
@Override
public void onReceive(Context context, Intent intent) {
    // Get the SMS message.
    Bundle bundle = intent.getExtras();
    SmsMessage[] msgs;
    String strMessage = "";
    String format = bundle.getString("format");
    // Retrieve the SMS message received.
    Object[] pdus = (Object[]) bundle.get("pdus");
    if (pdus != null) {
        // Check the Android version.
        boolean isVersionM =
                        (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M);
        // Fill the msgs array.
        msgs = new SmsMessage[pdus.length];
        for (int i = 0; i < msgs.length; i++) {
            // Check Android version and use appropriate createFromPdu.
            if (isVersionM) {
                // If Android version M or newer:
                msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i], format);
            } else {
                // If Android version L or older:
                msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
            }
            // Build the message to show.
            strMessage += "SMS from " + msgs[i].getOriginatingAddress();
            strMessage += " :" + msgs[i].getMessageBody() + "\n";
            // Log and display the SMS message.
            Log.d(TAG, "onReceive: " + strMessage);
            Toast.makeText(context, strMessage, Toast.LENGTH_LONG).show();
        }
    }
}
```

Tip: To build an SMS app with more features, see Telephony in the Android Developer's Documentation, and read the blog post Getting Your SMS Apps Ready for KitKat.

Receiving test messages in the emulator

You can emulate receiving a call or an SMS text message by clicking the ... (More) icon at the bottom of the emulator's toolbar on the right side, as shown in the figure below.

The extended controls for the emulator appear. Click **Phone** in the left column to see the extended phone controls:



Enter a message (or use the default "marshmallows" message) and click **Send Message** to send an SMS message to your emulator.

The emulator essentially sends itself the message, and responds as if receiving a call or receiving an SMS message.

Related practical

- Sending and Receiving SMS Messages - Part 1
- Sending and Receiving SMS Messages - Part 2

Learn more

- Android Developer Reference:
  - Intent
  - Common Intents: Text Messaging

    - Intents and Intent Filters
    - SmsManager
    - SmsMessage
    - Requesting Permissions at Run Time
    - checkSelfPermission
    - Run Apps on the Android Emulator
- Stack Overflow: Simulating incoming call or sms in Android Studio
- Android blog: Getting Your SMS Apps Ready for KitKat

2.1 - Part 1: Sending and Receiving SMS Messages

Contents:

Android smartphones can send and receive messages to or from any other phone that supports Short Message Service (SMS). You have two choices for *sending* SMS messages:

- Use an implicit Intent to launch a messaging app with the ACTION_SENDTO intent action.
  - This is the simplest choice for sending messages. The user can add a picture or other attachment in the messaging app, if the messaging app supports adding attachments.
  - Your app doesn't need code to request permission from the user.
  - If the user has multiple SMS messaging apps installed on the Android phone, the App chooser will appear with a list of these apps, and the user can choose which one to use. (Android smartphones will have at least one, such as Messenger.)
  - The user can change the message in the messaging app before sending it.
  - The user navigates back to your app using the **Back** button.
- Send the SMS message using the sendTextMessage() method or other methods of the SmsManager class.
  - This is a good choice for sending messages from your app without having to use another installed app.
  - Your app must ask the user for permission before sending the SMS message, if the user hasn't already granted permission.
  - The user stays in your app during and after sending the message.
  - You can manage SMS operations such as dividing a message into fragments, sending a multipart message, get carrier-dependent configuration values, and so on.

To *receive* SMS messages, use the onReceive() method of the BroadcastReceiver class.

What you should already KNOW

You should already be able to:

- Create an onClick method for a button with the `android:onClick` attribute.
- Use an implicit intent to perform a function with another app.
- Use a broadcast receiver to receive system events.

What you will LEARN

In this practical, you will learn to:

- Launch an SMS messaging app from your app with a phone number and message.
- Send an SMS message from within an app.
- Check for the SMS permission, and request permission if necessary.
- Receive SMS events using a broadcast receiver.
- Extract an SMS message from an SMS event.

What you will DO

In this practical, you will:

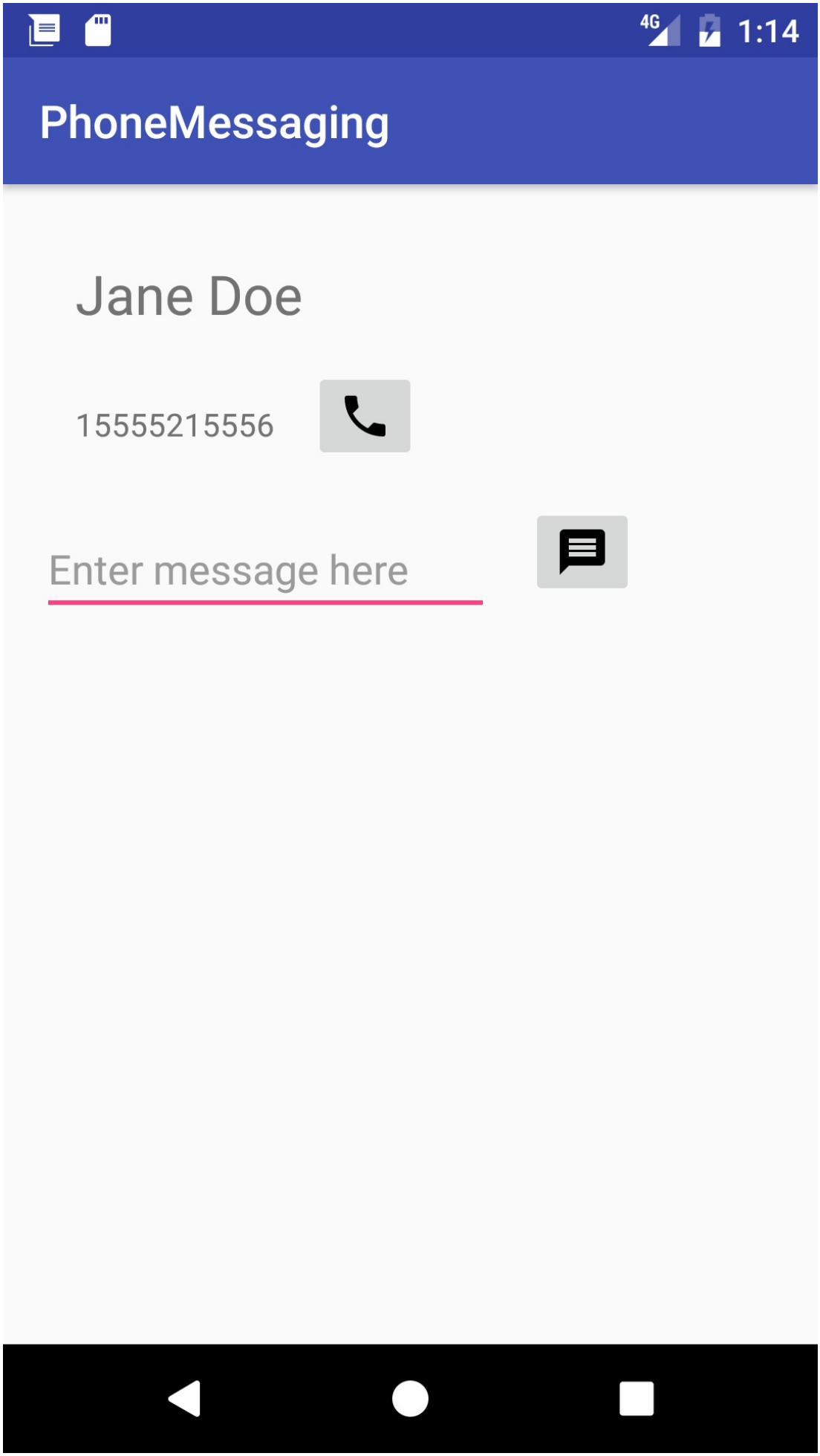
- Create an app that uses an implicit intent to launch a messaging app.
- Pass data (the phone number) and the message with the implicit intent.
- Create an app that sends SMS messages using the SmsManager class.
- Check for the SMS permission, which can change at any time.
- Request permission from the user, if necessary, to send SMS messages.
- Receive and process an SMS message.

App overview

You will create two new apps based on apps you created previously for the lesson about making phone calls:

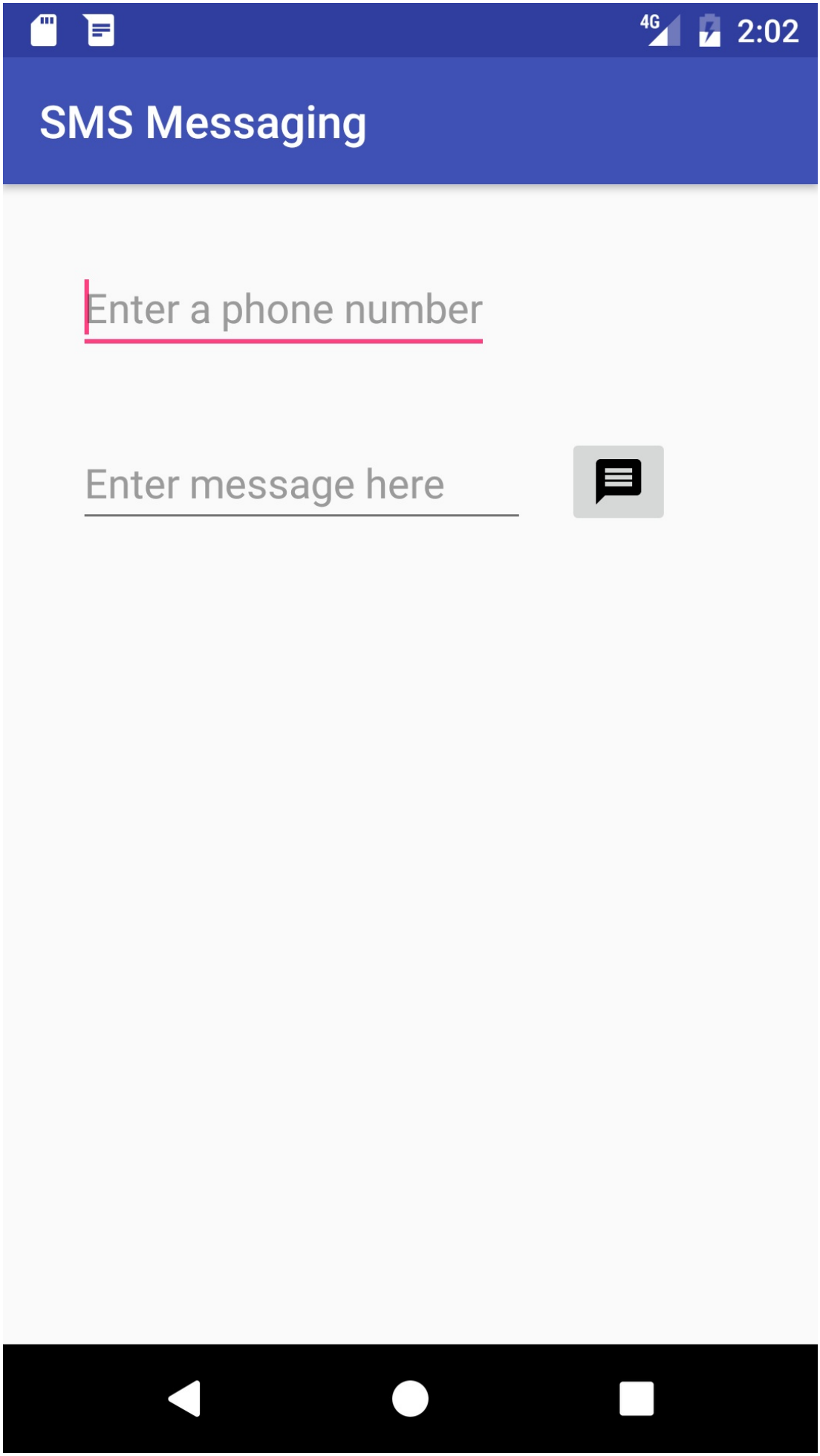
- **PhoneMessaging**: Rename and refactor the PhoneCallDial app from the previous chapter, and add code to enable a user to not only dial a hard-coded phone number but also send an SMS message to the phone number. It uses an implicit intent using `ACTION_SENDTO` and the phone number to launch a messaging app to send the message.

As shown in the figure below, the PhoneCallDial app already has TextEdit views for the contact name and the hard-coded phone number, and an ImageButton for making a phone call. You will copy the app, rename it to PhoneMessaging, and modify the layout to include an EditText for entering the message, and another ImageButton with an icon that the user can tap to send the message.


4G
1:14
PhoneMessaging
Jane Doe
15555215556
Enter message here

- **SMS Messaging**: Change the PhoneCallingSample app from the previous chapter to enable a user to enter a phone number, enter an SMS message, and send the message from within the app. It checks for permission and then uses the SmsManager class to send the message.

  As shown in the figure below, the PhoneCallingSample app already has an EditText view for entering the phone number and an ImageButton for making a phone call. You will copy the app, rename it to **SmsMessaging**, and modify the layout to include another EditText for entering the message, and change the ImageButton to an icon that the user can tap to send the message.


2:02
SMS Messaging
Enter a phone number
Enter message here

Task 1. Launch a messaging app to send a message

In this task you create an app called PhoneMessaging, a new version of the PhoneCallDial app from a previous lesson. The new app launches a messaging app with an implicit intent, and passes a fixed phone number and a message entered by the user.

The user can tap the messaging icon in your app to send the message. In the messaging app launched by the intent, the user can tap to send the message, or change the message or the phone number before sending the message. After sending the message, the user can navigate back to your app using the **Back** button.

1.1 Modify the app and layout

1. Copy the **PhoneCallDial** project folder, rename it to **PhoneMessaging**, and refactor it to populate the new name throughout the app project. (See the Appendix for instructions on copying a project.)
2. Add an icon for the messaging button by following these steps:
    i. Select drawable in the Project: Android view and choose **File > New > Vector Asset**.
    ii. Click the Android icon next to "Icon:" to choose an icon. To find a messaging icon, choose **Communication** in the left column.
    iii. Select the icon, click **OK**, click **Next**, and then click **Finish**.
3. Add the following EditText to the existing layout after the `phone_icon` ImageButton:

```
...
<ImageButton
    android:id="@+id/phone_icon"
    ... />

<EditText
    android:id="@+id/sms_message"
    android:layout_width="200dp"
    android:layout_height="wrap_content"
    android:layout_below="@id/number_to_call"
    android:layout_marginTop="@dimen/activity_vertical_margin"
    android:layout_marginRight="@dimen/activity_horizontal_margin"
    android:hint="Enter message here"
    android:inputType="textMultiLine"/>
```

You will use the `android:id` `sms_message` to retrieve the message in your code. You can use `@dimen/activity_horizontal_margin` and `@dimen/activity_vertical_margin` for the EditText margins because they are already defined in the dimens.xml file. The EditText view uses the `android:inputType` attribute set to `"textMultiLine"` for entering multiple lines of text.

4. After adding hard-coded strings and dimensions, extract them into resources:
   - `android:layout_width="@dimen/edittext_width"` : The width of the EditText message (200dp).
   - `android:hint="@string/enter_message_here"` : The hint for the EditText ("Enter message here").
5. Add the following ImageButton to the layout after the above EditText:

```
<ImageButton
    android:id="@+id/message_icon"
    android:contentDescription="Send a message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/activity_vertical_margin"
    android:layout_toRightOf="@id/sms_message"
    android:layout_toEndOf="@id/sms_message"
    android:layout_below="@id/phone_icon"
    android:src="@drawable/ic_message_black_24dp"
    android:onClick="smsSendMessage"/>
```

You will use the `android:id` `message_icon` to refer to the ImageButton for launching the messaging app. Use the vector asset you added previously (such as `ic_message_black_24dp` for a messaging icon) for the ImageButton.

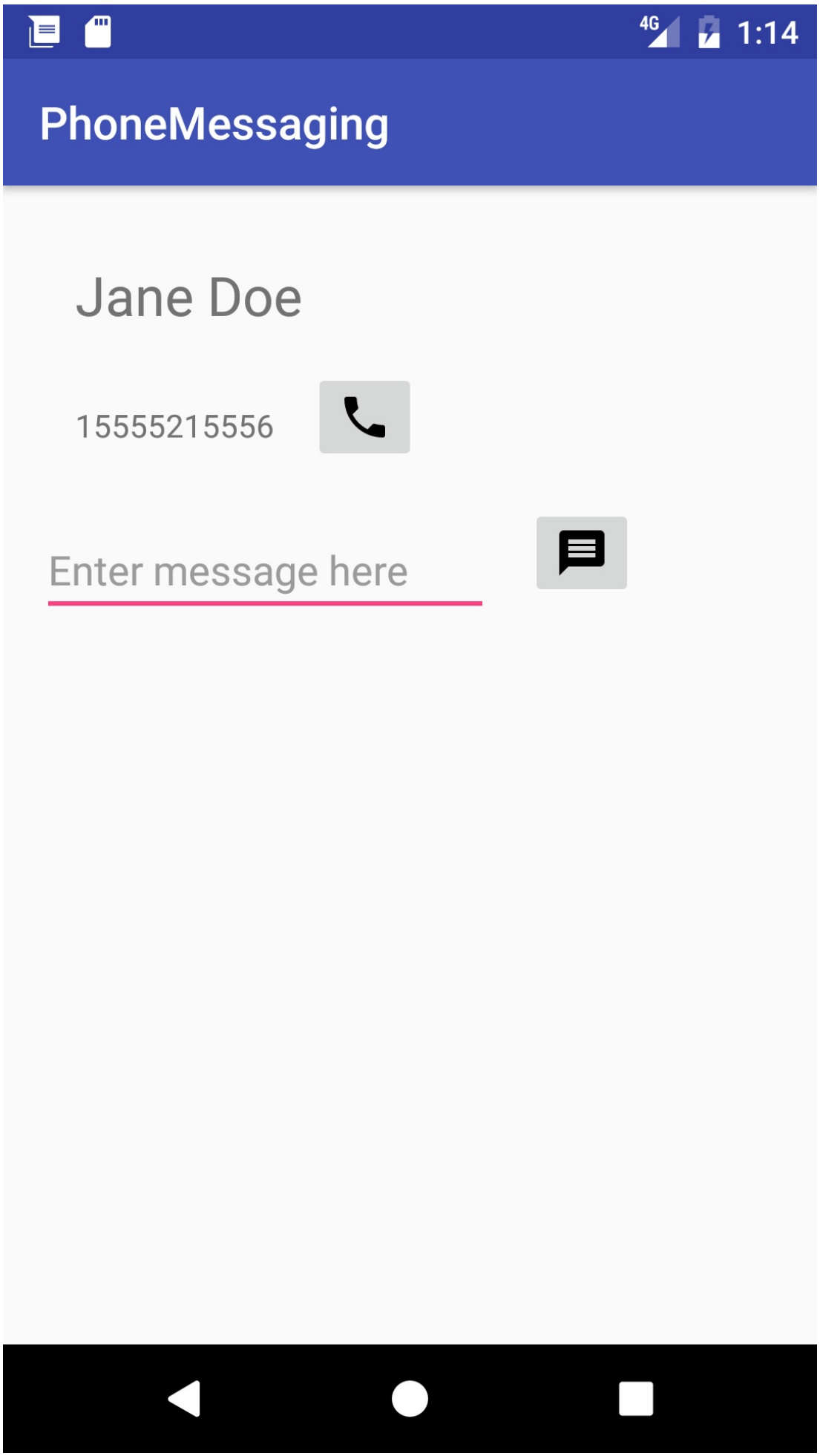
6. After adding the hard-coded string for the `android:contentDescription` attribute, extract it into the resource `send_a_message` .

   The `smsSendMessage()` method referred to in the `android:onClick` attribute remains highlighted until you create this method in the MainActivity, which you will do in the next step.

7. Click `smsSendMessage` in the `android:onClick` attribute, click the red light bulb that appears, and then select **Create smsSendMessage(View) in 'MainActivity'**. Android Studio automatically creates the `smsSendMessage()` method in MainActivity as `public` , returning `void` , with a `View` parameter. This method is called when the user taps the `message_icon` ImageButton.

```
public void smsSendMessage(View view) {
}
```

Your app's layout should now look like the following figure:


4G
1:14
PhoneMessaging
Jane Doe
15555215556
Enter message here

1.2 Edit the onClick method in MainActivity

1. Inside the `smsSendMessage()` method in MainActivity, get the phone number from the `number_to_call` TextView, and concatenate it with the `smsto:` prefix (as in `smsto:14155551212`) to create the phone number URI string `smsNumber` :

```
...
TextView textView = (TextView) findViewById(R.id.number_to_call);
// Use format with "smsto:" and phone number to create smsNumber.
String smsNumber = String.format("smsto: %s",
                                 textView.getText().toString());
...
```

2. Get the string of the message entered into the EditText view:

```
...
// Find the sms_message view.
EditText smsEditText = (EditText) findViewById(R.id.sms_message);
// Get the text of the SMS message.
String sms = smsEditText.getText().toString();
...
```

3. Create an implicit intent (`smsIntent`) with the intent action `ACTION_SENDTO` , and set the phone number and text message as intent data and extended data, using `setData()` and `putExtra` :

```
...
// Create the intent.
Intent smsIntent = new Intent(Intent.ACTION_SENDTO);
// Set the data for the intent as the phone number.
smsIntent.setData(Uri.parse(smsNumber));
  // Add the message (sms) with the key ("sms_body").
smsIntent.putExtra("sms_body", sms);
...
```

   The putExtra() method needs two strings: the key identifying the type of data (`"sms_body"`) and the data itself, which is the text of the message (`sms`). For more information about common intents and the `putExtra()` method, see Common Intents: Text Messaging.

4. Add a check to see if the implicit intent resolves to a package (a messaging app). If it does, send the intent with `startActivity()` , and the system launches the app. If it does not, log an error.

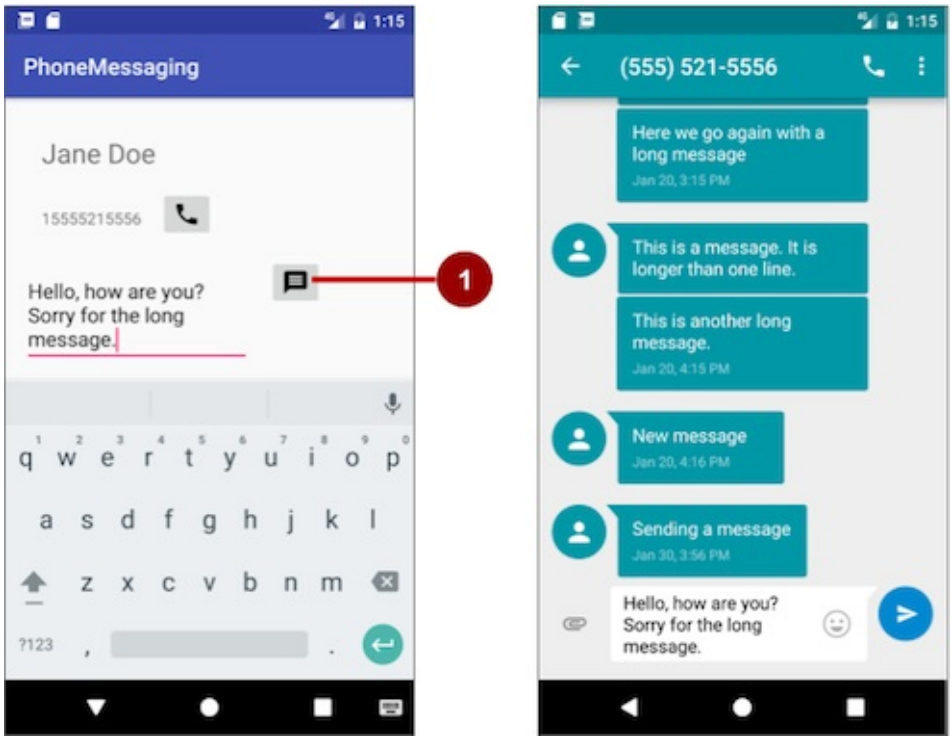
```
...
// If package resolves (target app installed), send intent.
if (smsIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(smsIntent);
} else {
    Log.e(TAG, "Can't resolve app for ACTION_SENDTO Intent");
}
...
```

The full method should now look like the following:

```
public void smsSendMessage(View view) {
    TextView textView = (TextView) findViewById(R.id.number_to_call);
    // Use format with "smsto:" and phone number to create smsNumber.
    String smsNumber = String.format("smsto: %s",
                                     textView.getText().toString());
    // Find the sms_message view.
    EditText smsEditText = (EditText) findViewById(R.id.sms_message);
    // Get the text of the sms message.
    String sms = smsEditText.getText().toString();
    // Create the intent.
    Intent smsIntent = new Intent(Intent.ACTION_SENDTO);
    // Set the data for the intent as the phone number.
    smsIntent.setData(Uri.parse(smsNumber));
    // Add the message (sms) with the key ("sms_body").
    smsIntent.putExtra("sms_body", sms);
    // If package resolves (target app installed), send intent.
    if (smsIntent.resolveActivity(getPackageManager()) != null) {
        startActivity(smsIntent);
    } else {
        Log.d(TAG, "Can't resolve app for ACTION_SENDTO Intent");
    }
}
```

1.3 Run the app

1. Run the app on either an emulator or a device.
2. Enter a message, and tap the messaging icon (marked "1" in the left side of the figure below). The messaging app appears, as shown on the right side of the figure below.



3. Use the **Back** button to return to the PhoneMessaging app. You may need to tap or click it more than once to leave the SMS messaging app.

Solution code

Android Studio project: PhoneMessaging

Task 2. Send an SMS message from within an app

In this task you will copy the **PhoneCallingSample** app from the lesson on making a phone call, rename and refactor it to **SmsMessaging**, and modify its layout and code to create an app that enables a user to enter a phone number, enter an SMS message, and send the message from within the app.

In the first step you will add the code to send the message, but the app will work only if you first turn on SMS permission manually for the app in Settings on your device or emulator.

In subsequent steps you will do away with setting this permission manually by requesting SMS permission from the app's user if it is not already set.

2.1 Create the app and layout and add permission

1. Copy the **PhoneCallingSample** project folder, rename it to **SmsMessaging**, and refactor it to populate the new name throughout the app project. (See the Appendix for

instructions on copying a project.)

2. Open **strings.xml** and change the `app_name` string resource to `"SMS Messaging"`.
3. Add the `android.permission.SEND_SMS` permission to the **AndroidManifest.xml** file, and remove the `CALL_PHONE` and `READ_PHONE_STATE` permissions for phone use, so that you have only one permission:

```
<uses-permission android:name="android.permission.SEND_SMS" />
```

Sending an SMS message is permission-protected. Your app can't use SMS without the `SEND_SMS` permission line in AndroidManifest.xml. This permission line enables a setting for the app in the Settings app that gives the user the choice of allowing or disallowing use of SMS. (In the next task you will add a way for the user to grant that permission from within the app.)

4. Add a messaging icon as you did in the previous task, and remove the phone icon from the drawable folder.
5. Open activity_main.xml and edit the EditText view and replace the `android:layout_margin` attribute with the following:

```
...
android:layout_marginTop="@dimen/activity_vertical_margin"
android:layout_marginRight="@dimen/activity_horizontal_margin"
...
```

You can use `@dimen/activity_horizontal_margin` and `@dimen/activity_vertical_margin` because they are already defined in the dimens.xml file.

6. Add the following EditText to the layout after the first EditText (for an image of the layout, see the figure at the end of these steps):

```
...
<EditText
    android:id="@+id/sms_message"
    android:layout_width="@dimen/edittext_width"
    android:layout_height="wrap_content"
    android:layout_below="@id/editText_main"
    android:layout_margin="@dimen/activity_horizontal_margin"
    android:hint="Enter message here"
    android:inputType="textMultiLine"/>
```

You will use the `android:id` attribute to `sms_message` to identify it as the EditText for the message. The EditText view uses the `android:inputType` attribute set to `"textMultiLine"` for entering multiple lines of text.

7. After adding the hard-coded string "Enter message here" for the `android:hint` attribute, extract it into the text resource `"enter_message_here"`.

8. Change the `android:layout_below` attribute for the `button_retry` Button to refer to the `sms_message` EditText view. The Button should appear below the SMS message in the layout if it becomes visible:

   ```
   android:layout_below="@id/sms_message"
   ```

   The `button_retry` Button is set to invisible. It appears only if the app detected that telephony is not enabled, or if the user previously denied phone permission when the app requested it.

9. Replace the `phone_icon` ImageButton from the existing layout with the following:

   ```
   <ImageButton
       android:id="@+id/message_icon"
       android:contentDescription="Send a message"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_marginTop="@dimen/activity_vertical_margin"
       android:layout_toRightOf="@id/sms_message"
       android:layout_toEndOf="@id/sms_message"
       android:layout_below="@id/editText_main"
       android:src="@drawable/ic_message_black_24dp"
       android:visibility="visible"
       android:onClick="smsSendMessage"/>
   ```

   You will use the `android:id` `message_icon` in your code to refer to the ImageButton for sending the message. Use the vector asset you added previously (such as `ic_message_black_24dp` for a messaging icon) for the ImageButton. Make sure you include the `android:visibility` attribute set to `"visible"`. You will control the visibility of this ImageButton from your code.

10. After adding a hard-coded string for the `android:contentDescription` attribute, extract it to the `send_a_message` string resource.

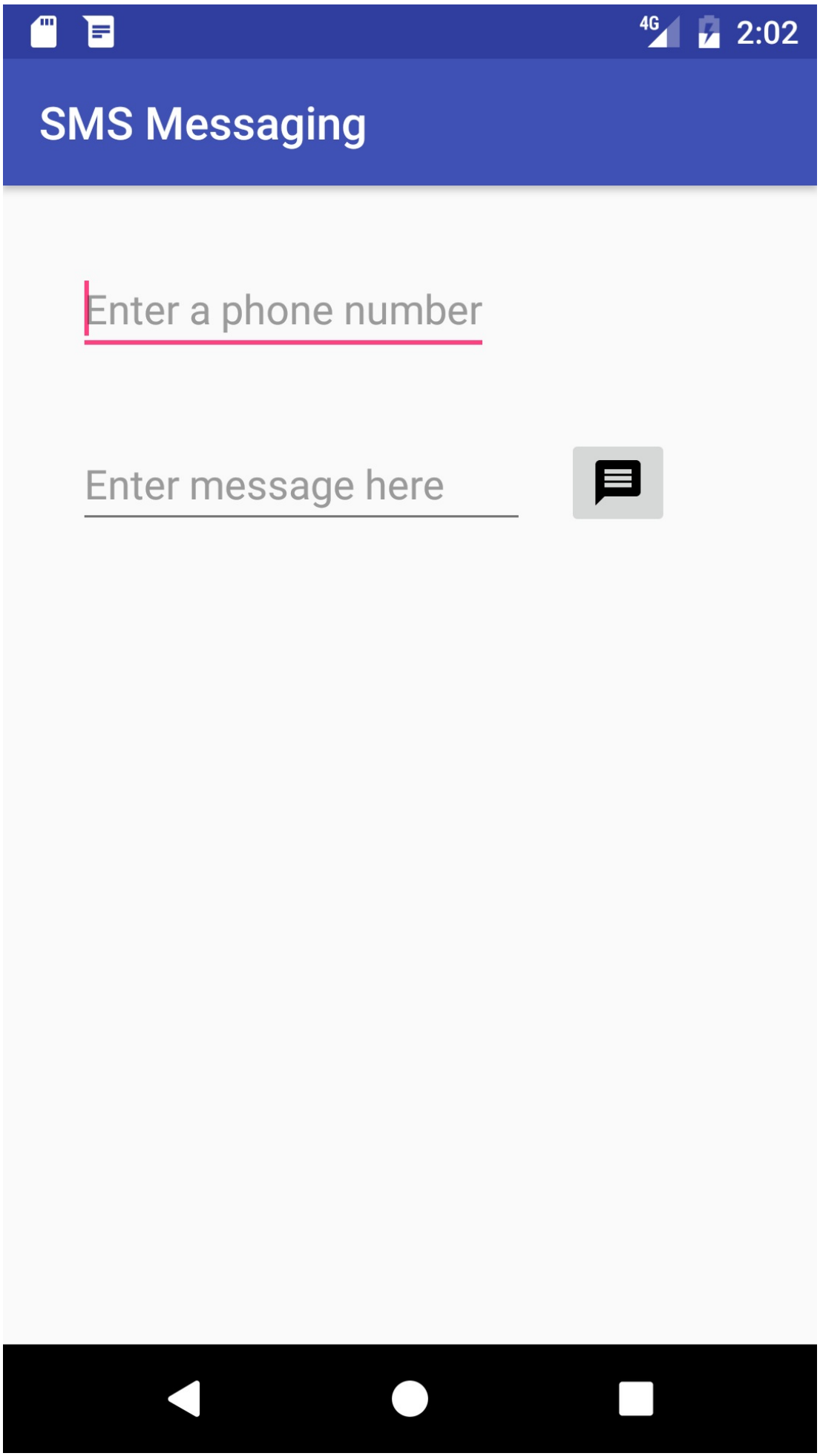
    The `smsSendMessage()` method referred to in the `android:onClick` attribute for the ImageButton remains highlighted until you create this method in the MainActivity, which you will do in the next step.

11. Click `smsSendMessage` in the `android:onClick` attribute, click the red light bulb that appears, and then select **Create smsSendMessage(View) in 'MainActivity'**. Android Studio automatically creates the `smsSendMessage()` method in MainActivity as `public`,

returning `void`, with a `View` parameter. This method is called when the user taps the `message_icon` ImageButton.

```
public void smsSendMessage(View view) {
}
```

Your app's layout should look like the following figure (the `button_retry` Button is invisible):


4G
2:02
SMS Messaging
Enter a phone number
Enter message here

2.2 Edit the onClick method in MainActivity

1. Open MainActivity and find the new `smsSendMessage()` method you created in the last step.
2. Add statements to the method to get the string for the phone number from the `editText_main` view, and get the string for the SMS message from the `sms_message` view:

```
public void smsSendMessage(View view) {
   EditText editText = (EditText) findViewById(R.id.editText_main);
   // Set the destination phone number to the string in editText.
   String destinationAddress = editText.getText().toString();
   // Find the sms_message view.
   EditText smsEditText = (EditText) findViewById(R.id.sms_message);
   // Get the text of the sms message.
   String smsMessage = smsEditText.getText().toString();
   ...
}
```

3. Declare additional string and `PendingIntent` parameters for the sendTextMessage() method, which will send the message (`destinationAddress` is already declared as the string for the phone number to receive the message):
   - `scAddress` : A string for the service center address, or `null` to use the current default SMSC. A Short Message Service Center (SMSC) is a network element in the mobile telephone network. The mobile network operator usually presets the correct service center number in the default profile of settings stored in the device's SIM card.
   - `smsMessage` : A string for the body of the message to send.
   - `sentIntent` : A `PendingIntent` . If not `null` , this is broadcast when the message is successfully sent or if the message failed.
   - `deliveryIntent` : A `PendingIntent` . If not `null` , this is broadcast when the message is delivered to the recipient.

```
...
// Set the service center address if needed, otherwise null.
String scAddress = null;
// Set pending intents to broadcast
// when message sent and when delivered, or set to null.
PendingIntent sentIntent = null, deliveryIntent = null;
...
```

4. Use the SmsManager class to create `smsManager` , which automatically imports `android.telephony.SmsManager` , and use `sendTextMessage()` to send the message:

```
...
// Use SmsManager.
SmsManager smsManager = SmsManager.getDefault();
smsManager.sendTextMessage
       (destinationAddress, scAddress, smsMessage,
                       sentIntent, deliveryIntent);
...
```

The full method should now look like the following:

```
public void smsSendMessage(View view) {
    EditText editText = (EditText) findViewById(R.id.editText_main);
    // Set the destination phone number to the string in editText.
    String destinationAddress = editText.getText().toString();
    // Find the sms_message view.
    EditText smsEditText = (EditText) findViewById(R.id.sms_message);
    // Get the text of the SMS message.
    String smsMessage = smsEditText.getText().toString();
    // Set the service center address if needed, otherwise null.
    String scAddress = null;
    // Set pending intents to broadcast
    // when message sent and when delivered, or set to null.
    PendingIntent sentIntent = null, deliveryIntent = null;
    // Use SmsManager.
    SmsManager smsManager = SmsManager.getDefault();
    smsManager.sendTextMessage
                  (destinationAddress, scAddress, smsMessage,
                                  sentIntent, deliveryIntent);
}
```

If you run the app now, on either a device or an emulator, the app may crash depending on whether the device or emulator has been previously set to allow the app to use SMS. In some versions of Android, this permission is turned on by default. In other versions, this permission is turned off by default.

To set the app's permission on a device or emulator instance, choose **Settings > Apps > SMS Messaging > Permissions**, and turn on the SMS permission for the app. Since the user can turn on or off SMS permission at any time, you need to add a check in your app for this permission, and request it from the user if necessary. You will do this in the next step.

2.3 Check for and request permission for SMS

Your app must always get permission to use anything that is not part of the app itself. In Step 2.1 you added the following permission to the AndroidManifest.xml file:

```
<uses-permission android:name="android.permission.SEND_SMS" />
```

This permission enables a permission setting in the Settings app for your app. The user can allow or disallow this permission at any time from the Settings app. You can add code to request permission from the user if the user has turned off SMS permission for the app. Follow these steps:

1. At the top of MainActivity, below the class definition, change the global constant for the `MY_PERMISSIONS_REQUEST_CALL_PHONE` request code to the following:

   ```
   private static final int MY_PERMISSIONS_REQUEST_SEND_SMS = 1;
   ```

   When a result returns in the activity, it will contain the `MY_PERMISSIONS_REQUEST_SEND_SMS` `requestCode` so that your code can identify it.

2. Remove the constant declarations for `mTelephonyManager` and `MyPhoneCallListener`.

3. Remove the `isTelephonyEnabled()` method, and remove all of the code in the `onCreate()` method that starts with the `mTelephonyManager` assignment, leaving only the first two lines:

   ```
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
   }
   ```

4. Refactor/rename the existing `disableCallButton()` method to `disableSmsButton()` and edit the method to do the following:

   i. Display a toast to notify the user that SMS usage is disabled.

   ii. Find and then set the `smsButton` (the message icon) to be invisible so that the user can't send a message.

   iii. Set the **Retry** button to be visible, so that the user can restart the activity and allow permission.

```
private void disableSmsButton() {
   Toast.makeText(this, "SMS usage disabled", Toast.LENGTH_LONG).show();
   ImageButton smsButton = (ImageButton) findViewById(R.id.message_icon);
   smsButton.setVisibility(View.INVISIBLE);
   Button retryButton = (Button) findViewById(R.id.button_retry);
   retryButton.setVisibility(View.VISIBLE);
}
```

Extract a string resource (`sms_disabled`) for the hard-coded string "SMS usage disabled" in the toast statement.

5. Refactor/rename the existing `enableCallButton()` method to `enableSmsButton()` to set the SMS message icon button to be visible:

```
private void enableSmsButton() {
   ImageButton smsButton = (ImageButton) findViewById(R.id.message_icon);
   smsButton.setVisibility(View.VISIBLE);
}
```

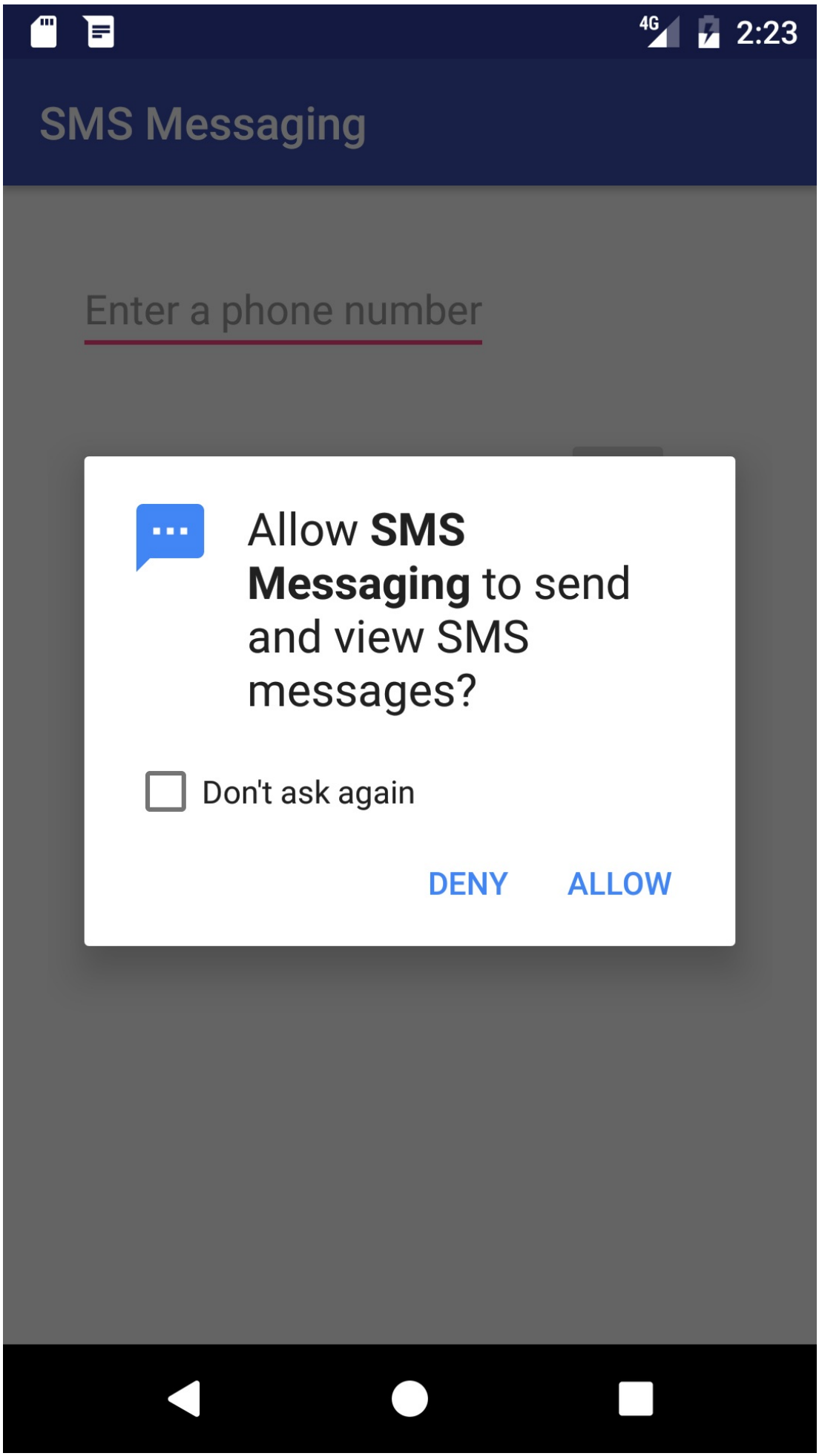
6. Modify the existing `retryApp()` method in MainActivity to remove the call to `enableCallButton()` .

7. In MainActivity, rename and refactor the `checkForPhonePermission()` method to `checkForSmsPermission()` , and change the code to the following:

```
private void checkForSmsPermission() {
   if (ActivityCompat.checkSelfPermission(this,
               Manifest.permission.SEND_SMS) !=
               PackageManager.PERMISSION_GRANTED) {
       Log.d(TAG, getString(R.string.permission_not_granted));
       // Permission not yet granted. Use requestPermissions().
       // MY_PERMISSIONS_REQUEST_SEND_SMS is an
       // app-defined int constant. The callback method gets the
       // result of the request.
       ActivityCompat.requestPermissions(this,
               new String[]{Manifest.permission.SEND_SMS},
               MY_PERMISSIONS_REQUEST_SEND_SMS);
   } else {
       // Permission already granted. Enable the SMS button.
       enableSmsButton();
   }
}
```

Use checkSelfPermission() to determine whether your app has been granted a particular permission by the user. If permission has *not* been granted by the user, use the requestPermissions() method to display a standard dialog for the user to grant permission.

When your app calls requestPermissions(), the system shows a standard dialog for each permission to the user, as shown in the figure below.


2:23
SMS Messaging
Enter a phone number
Allow SMS Messaging to send and view SMS messages?
Don't ask again
DENY
ALLOW

8. When the user responds to the request permission dialog, the system invokes your app's onRequestPermissionsResult() method, passing it the user response. Find the `onRequestPermissionsResult()` method you created for the previous version of this app.

   Your implementation of `onRequestPermissionsResult()` already uses a `switch` statement based on the value of `requestCode`. A `case` for checking phone permission is already implemented using `MY_PERMISSIONS_REQUEST_CALL_PHONE`. Replace `MY_PERMISSIONS_REQUEST_CALL_PHONE` with `MY_PERMISSIONS_REQUEST_SEND_SMS`, and replace `CALL_PHONE` with `SEND_SMS`. The switch block should now look like the following:

   ```
   ...
   switch (requestCode) {
       case MY_PERMISSIONS_REQUEST_SEND_SMS: {
           if (permissions[0].equalsIgnoreCase
               (Manifest.permission.SEND_SMS)
               && grantResults[0] ==
               PackageManager.PERMISSION_GRANTED) {
               // Permission was granted. Enable sms button.
               enableSmsButton();
           } else {
               // Permission denied.
               Log.d(TAG, getString(R.string.failure_permission));
               Toast.makeText(this,
                           getString(R.string.failure_permission),
                           Toast.LENGTH_LONG).show();
               // Disable the sms button.
               disableSmsButton();
           }
       }
   }
   ```

   If the user allows the permission request, the message button is re-enabled with `enableSmsButton()` in case it was made invisible by lack of permission.

   If the user denies the permission requests, your app should take appropriate action. For example, your app might disable the functionality that depends on a specific permission and show a dialog explaining why it could not perform it. For now, log a debug message, display a toast to show that permission was not granted, and disable the message button with `disableSmsButton()`.

9. In the `onCreate()` method of MainActivity, add a call to the `checkForSmsPermission()` method:

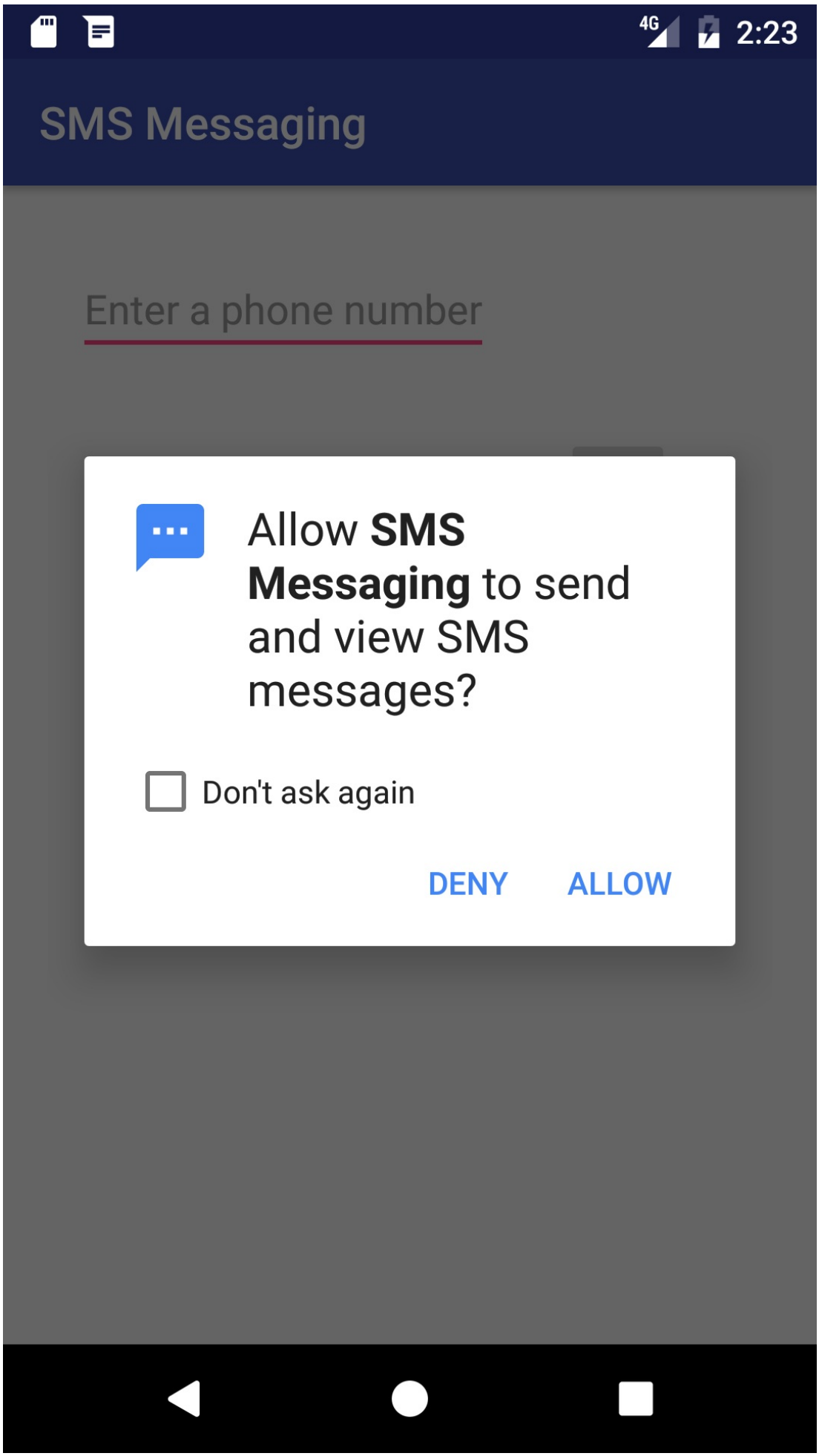
```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
   checkForSmsPermission();
}
```

10. Remove the `callNumber()` method and the `MyPhoneCallListener` inner class (including the `onCallStateChanged()` method, as you are no longer using the Telephony Manager).
11. Remove the `onDestroy()` method since you are no longer using a listener.
12. Since the user might turn off SMS permission while the app is still running, add a check for SMS permission in the `smsSendMessage()` method after setting the `sentIntent` but before using the `SmsManager` class:

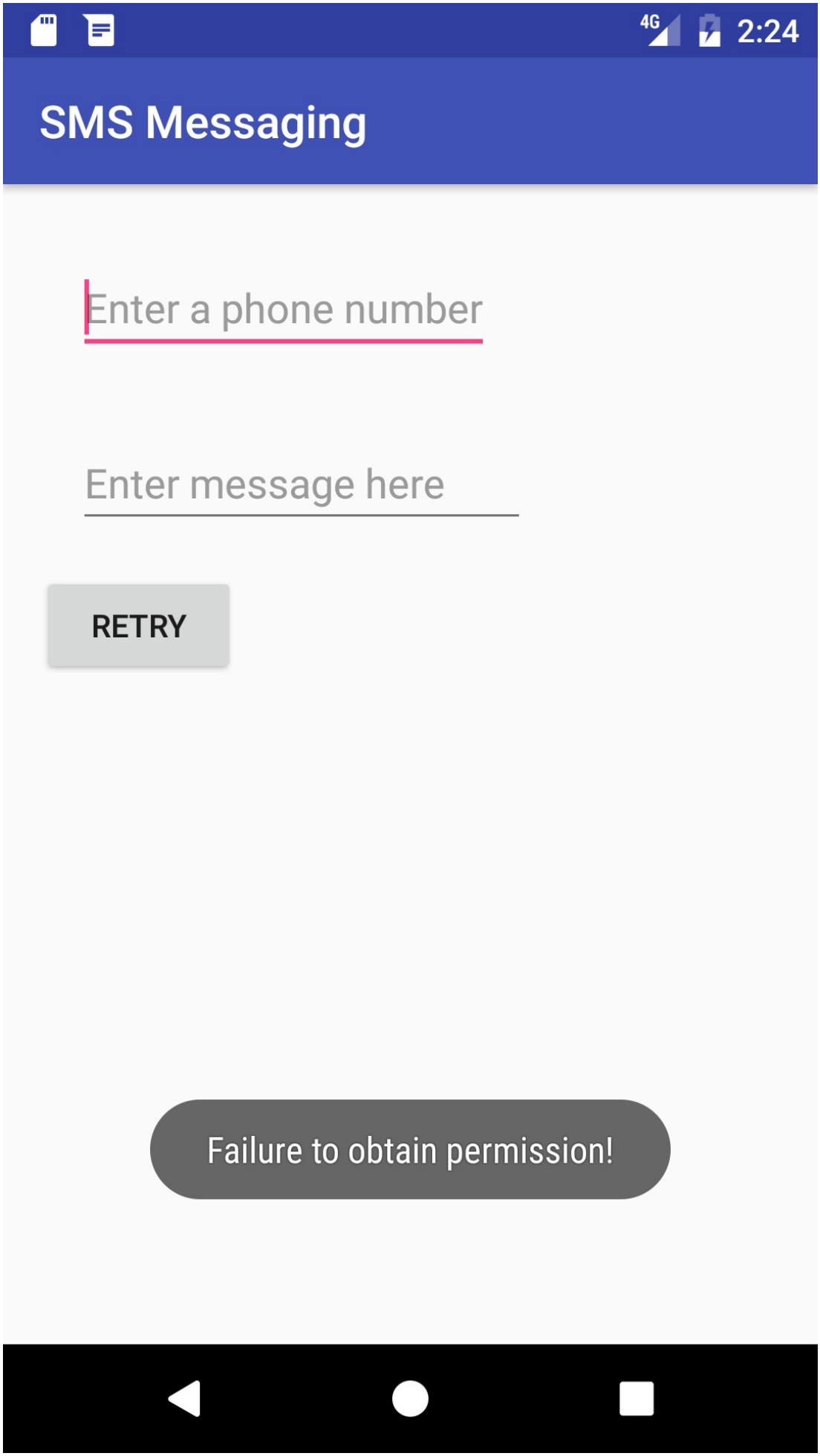
```
...
PendingIntent sentIntent = null, deliveryIntent = null;
// Check for permission first.
checkForSmsPermission();
// Use SmsManager.
...
```

2.4 Run the app and test permissions

1. Run your app. Enter a phone number (or the emulator port number if using emulators), and enter the message to send. Tap the messaging icon to send the message.
2. After running the app, choose **Settings > Apps > SMS Messaging > Permissions** and turn *off* SMS permission for the app.
3. Run the app again. You should see the SMS permission request dialog as shown below.


2:23
SMS Messaging
Enter a phone number
Allow SMS Messaging to send and view SMS messages?
Don't ask again
DENY
ALLOW

4. Click **Deny**. In the app's UI, the message icon button no longer appears, and a **Retry** button appears, as shown below.

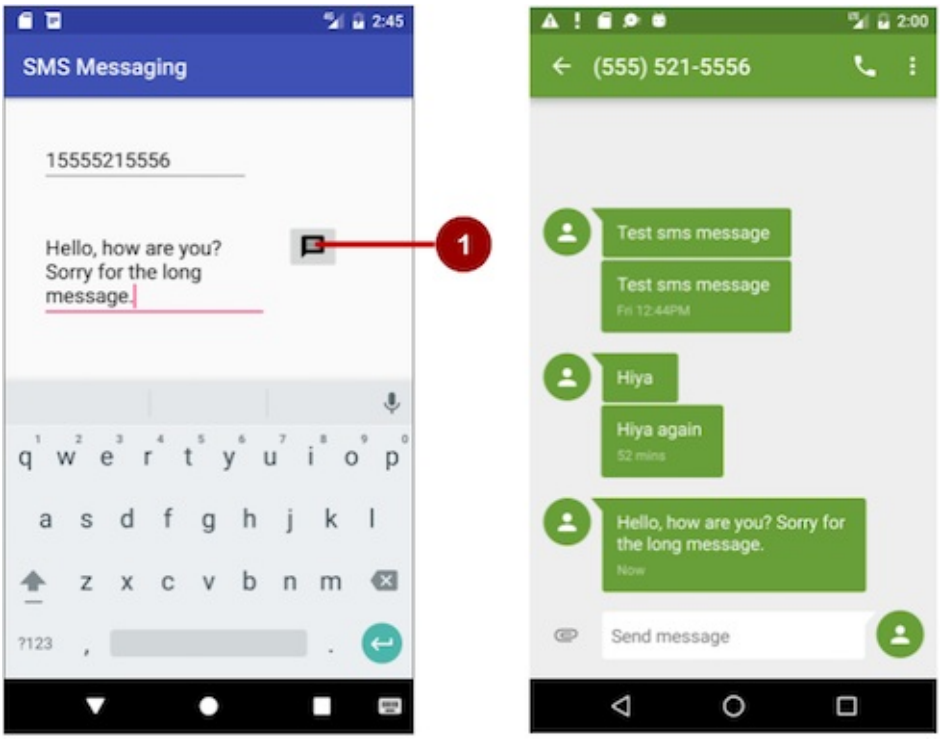

2:24
SMS Messaging
Enter a phone number
Enter message here
RETRY
Failure to obtain permission!

5. Click **Retry**, and then click **Allow** for SMS permission.
6. Test the app's ability to send a message:

    i. Enter a phone number.

    ii. Enter a message.

    iii. Tap the messaging icon.



End of Part 1 - Continue with Part 2

2.2: Part 2 - Sending and Receiving SMS Messages

Contents:

Task 3. Receive SMS messages with a broadcast receiver

To receive SMS messages, use the onReceive() method of the BroadcastReceiver class. The Android framework sends out system broadcasts of events such as receiving an SMS message, containing intents that are meant to be received using a BroadcastReceiver. You need to add the `RECEIVE_SMS` permission to your app's AndroidManifest.xml file.

3.1 Add permission and create a broadcast receiver

To add `RECEIVE_SMS` permission and create a broadcast receiver, follow these steps:

1. Open the AndroidManifest.xml file and add the `android.permission.RECEIVE_SMS` permission below the other permission for SMS use:

   ```
   <uses-permission android:name="android.permission.SEND_SMS" />
   <uses-permission android:name="android.permission.RECEIVE_SMS" />
   ```

   Receiving an SMS message is permission-protected. Your app can't receive SMS messages without the `RECEIVE_SMS` permission line in AndroidManifest.xml.

2. Select the package name in the Project:Android: view and choose **File > New > Other > Broadcast Receiver**.
3. Name the class "MySmsReceiver" and make sure "Exported" and "Enabled" are checked.

   The "Exported" option allows your app to respond to outside broadcasts, while "Enabled" allows it to be instantiated by the system.

4. Open the AndroidManifest.xml file again. Note that Android Studio automatically generates a `<receiver>` tag with your chosen options as attributes:

```
<receiver
    android:name=
        "com.example.android.smsmessaging.MySmsReceiver"
    android:enabled="true"
    android:exported="true">
</receiver>
```

3.2 Register the broadcast receiver

In order to receive any broadcasts, you must register for specific broadcast intents. In the Intent documentation, under "Standard Broadcast Actions", you can find some of the common broadcast intents sent by the system. In this app, you use the `android.provider.Telephony.SMS_RECEIVED` intent.

Add the following inside the `<receiver>` tags to register your receiver:

```
<receiver
    android:name="com.example.android.smsmessaging.MySmsReceiver"
    android:enabled="true"
    android:exported="true">
    <intent-filter>
        <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
    </intent-filter>
</receiver>
```

3.3 Implement the onReceive() method

Once the BroadcastReceiver intercepts a broadcast for which it is registered (`SMS_RECEIVED`), the intent is delivered to the receiver's `onReceive()` method, along with the context in which the receiver is running.

1. Open MySmsReceiver and add under the class declaration a string constant `TAG` for log messages and a string constant `pdu_type` for identifying PDUs in a bundle:

```
public class MySmsReceiver extends BroadcastReceiver {
   private static final String TAG =
                          MySmsReceiver.class.getSimpleName();
   public static final String pdu_type = "pdus";
   ...
```

2. Delete the default implementation inside the supplied `onReceive()` method.
3. In the blank `onReceive()` method:

i. Add the `@TargetAPI` annotation for the method, because it performs a different action depending on the build version.

ii. Retrieve a map of extended data from the intent to a `bundle`.

iii. Define the `msgs` array and `strMessage` string.

iv. Get the `format` for the message from the `bundle`.

```
@TargetApi(Build.VERSION_CODES.M)
@Override
public void onReceive(Context context, Intent intent) {
   // Get the SMS message.
   Bundle bundle = intent.getExtras();
   SmsMessage[] msgs;
   String strMessage = "";
   String format = bundle.getString("format");
   ...
```

As you enter `SmsMessage[]`, Android Studio automatically imports `android.telephony.SmsMessage`.

4. Retrieve from the bundle one or more pieces of data in the protocol data unit (PDU) format, which is the industry-standard format for an SMS message:

```
...
// Retrieve the SMS message received.
Object[] pdus = (Object[]) bundle.get(pdu_type);
...
```

5. If there are messages (`pdus`), check for Android version 6.0 (Marshmallow) and newer versions. You will use this boolean to check if your app needs the deprecated signature createFromPdu(byte[] pdu) for earlier versions of Android:

```
...
if (pdus != null) {
   // Check the Android version.
   boolean isVersionM = (Build.VERSION.SDK_INT >=
                         Build.VERSION_CODES.M);
   ...
```

6. Initialize the `msgs` array, and use its length in the `for` loop:

```
...
// Fill the msgs array.
  msgs = new SmsMessage[pdus.length];
      for (int i = 0; i < msgs.length; i++) {
          // Check Android version and use appropriate createFromPdu.
          if (isVersionM) {
              // If Android version M or newer:
              msgs[i] =
                   SmsMessage.createFromPdu((byte[]) pdus[i], format);
          } else {
              // If Android version L or older:
              msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
          }
          ...
```

Use createFromPdu(byte[] pdu, String format) to fill the `msgs` array for Android version 6.0 (Marshmallow) and newer versions. For earlier versions of Android, use the deprecated signature createFromPdu(byte[] pdu).

7. Build the `strMessage` to show in a toast message:
   i. Get the originating address using the `getOriginatingAddress()` method.
   ii. Get the message body using the `getMessageBody()` method.
   iii. Add an ending character for an end-of-line.

```
...
// Build the message to show.
strMessage += "SMS from " + msgs[i].getOriginatingAddress();
strMessage += " :" + msgs[i].getMessageBody() + "\n";
...
```

8. Log the resulting `strMessage` and display a toast with it:

```
...
// Log and display the SMS message.
Log.d(TAG, "onReceive: " + strMessage);
Toast.makeText(context, strMessage, Toast.LENGTH_LONG).show();
...
```

The complete `onReceive()` method is shown below:

```
@TargetApi(Build.VERSION_CODES.M)
@Override
public void onReceive(Context context, Intent intent) {
    // Get the SMS message.
    Bundle bundle = intent.getExtras();
    SmsMessage[] msgs;
    String strMessage = "";
    String format = bundle.getString("format");
    // Retrieve the SMS message received.
    Object[] pdus = (Object[]) bundle.get(pdu_type);
    if (pdus != null) {
        // Check the Android version.
        boolean isVersionM =
                        (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M);
        // Fill the msgs array.
        msgs = new SmsMessage[pdus.length];
        for (int i = 0; i < msgs.length; i++) {
            // Check Android version and use appropriate createFromPdu.
            if (isVersionM) {
                // If Android version M or newer:
                msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i], format);
            } else {
                // If Android version L or older:
                msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
            }
            // Build the message to show.
            strMessage += "SMS from " + msgs[i].getOriginatingAddress();
            strMessage += " :" + msgs[i].getMessageBody() + "\n";
            // Log and display the SMS message.
            Log.d(TAG, "onReceive: " + strMessage);
            Toast.makeText(context, strMessage, Toast.LENGTH_LONG).show();
        }
    }
}
```

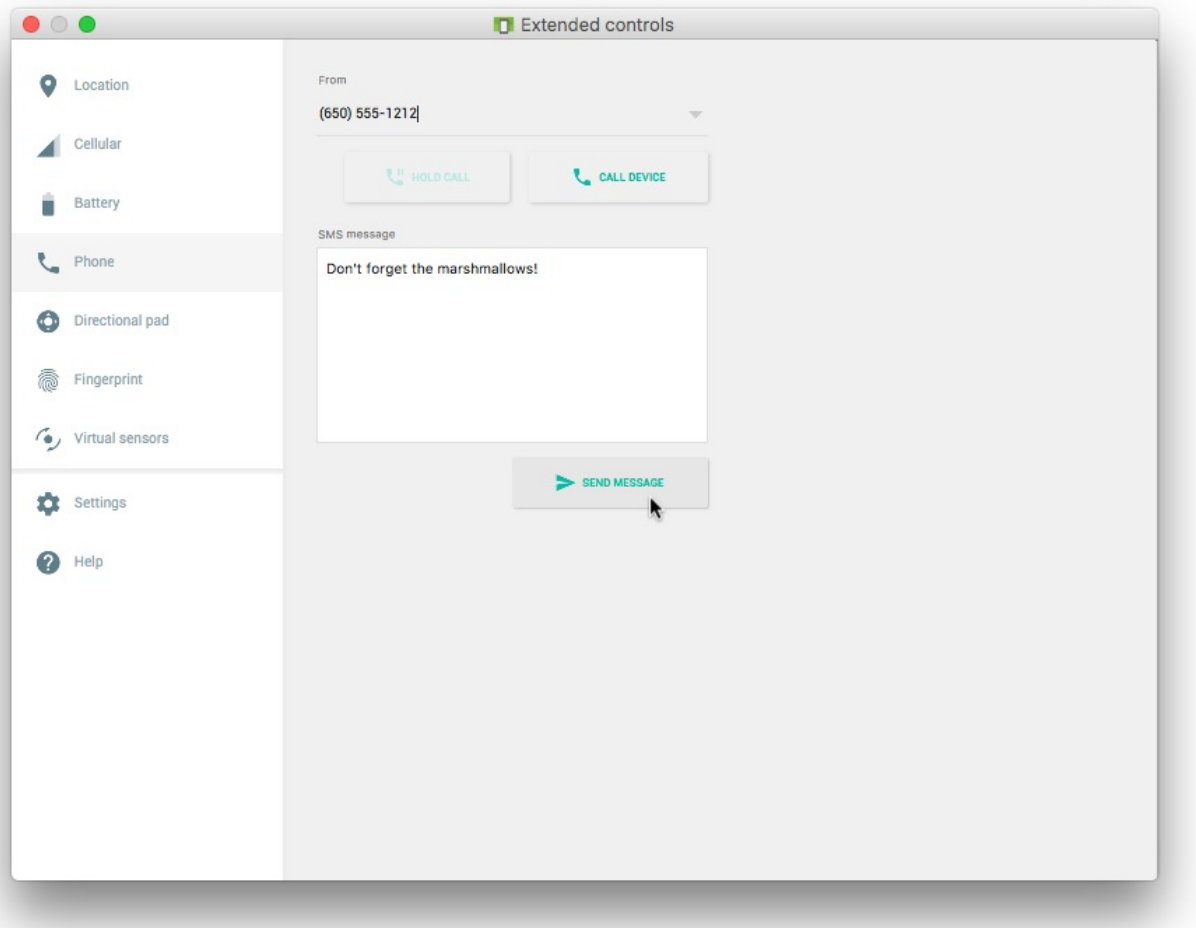
3.4 Run the app and send a message

Run the app on a device. If possible, have someone send you an SMS message from a different device.

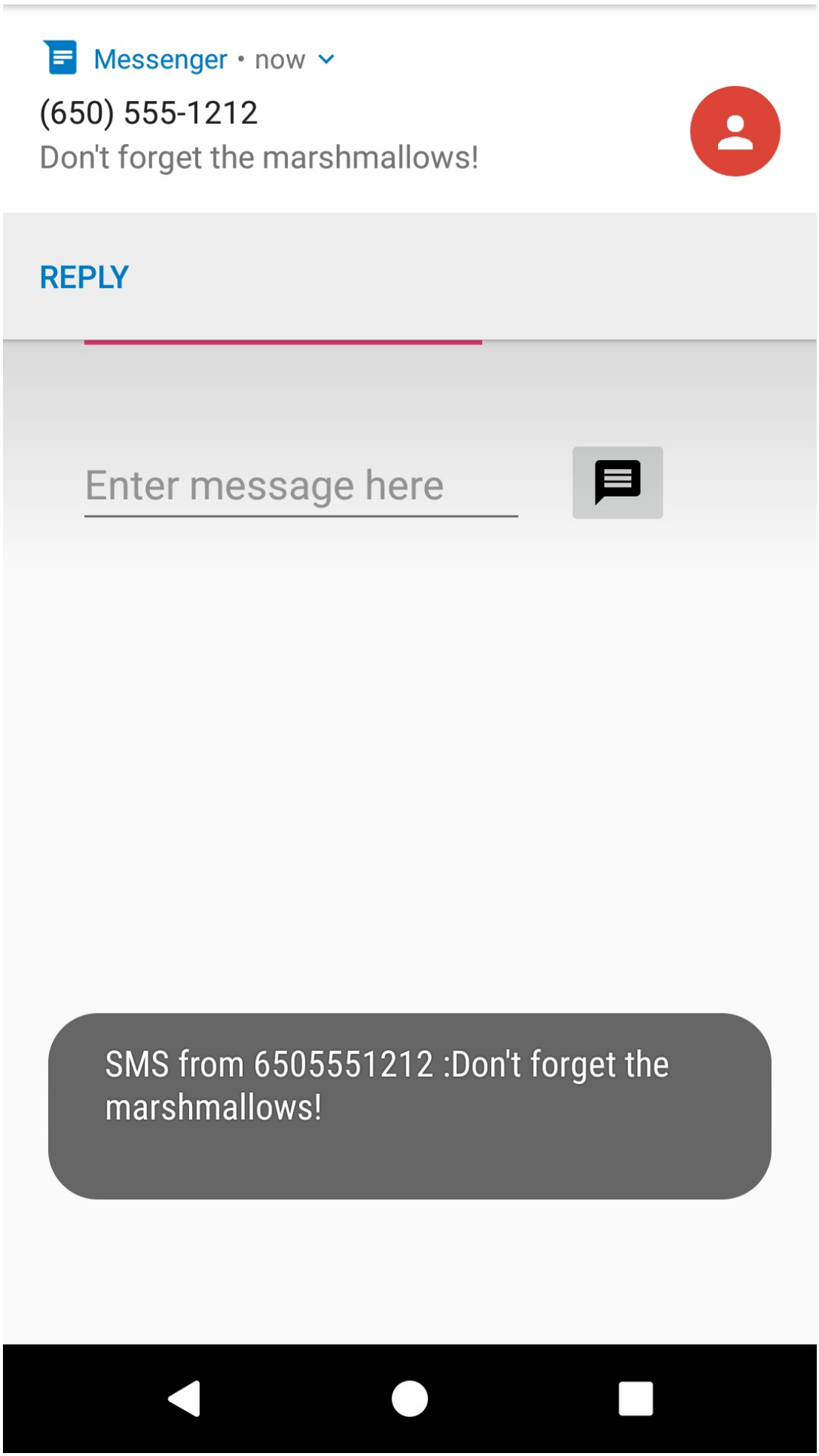
You can also receive an SMS text message when testing on an emulator. Follow these steps:

1. Run the app on an emulator.
2. Click the **...** (More) icon at the bottom of the emulator's toolbar on the right side, as shown in the figure below:

3. The extended controls for the emulator appear. Click **Phone** in the left column to see the extended phone controls:



4. You can now enter a message (or use the default "marshmallows" message) and click **Send Message** to have the emulator send an SMS message to itself.
5. The emulator responds with a notification about receiving an SMS message. The app should also display a toast message showing the message and its originating address, as shown below:


Messenger • now
(650) 555-1212
Don't forget the marshmallows!
REPLY
Enter message here
SMS from 6505551212 :Don't forget the marshmallows!

Solution Code

Android Studio project: SmsMessaging

Coding challenge

Note: All coding challenges are optional.
Challenge: Create a simple app with one button, **Choose Picture and Send**, that enables the user to select an image from the Gallery and send it as a Multimedia Messaging Service (MMS) message. After tapping the button, a choice of apps may appear, including the Messenger app. The user can select the Messenger app, and select an existing conversation or create a new conversation, and then send the image as a message.

The following are hints:

- To access and share an image from the Gallery, you need the following permission in the AndroidManifest.xml file:

```
  <uses-permission
         android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

- To enable the above permission, follow the model shown previously in this chapter to check for the `READ_EXTERNAL_STORAGE` permission, and request permission if necessary.
- Use the following intent for picking an image:

```
Intent galleryIntent = new Intent(Intent.ACTION_PICK,
   android.provider.MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
startActivityForResult(galleryIntent, IMAGE_PICK);
```

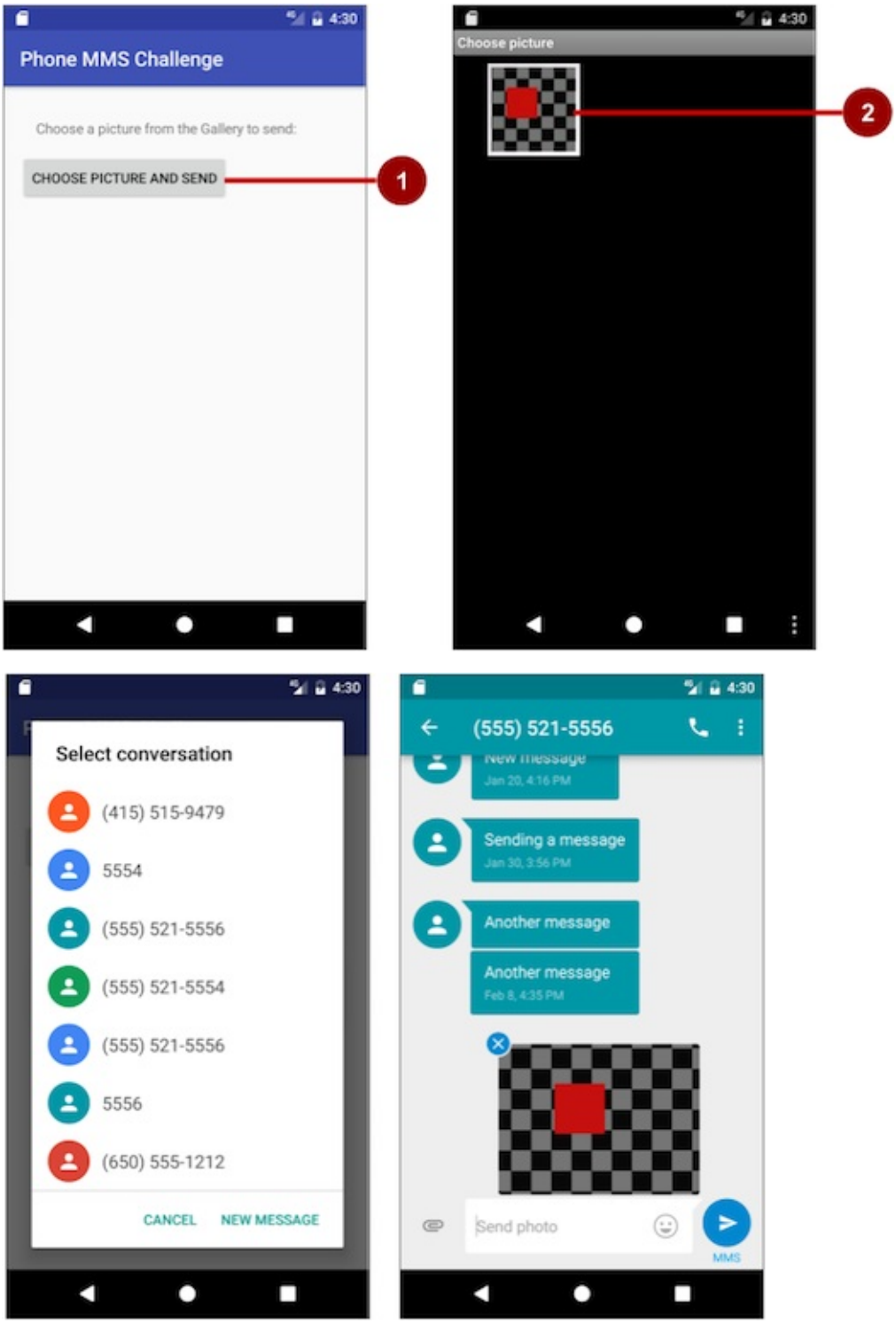
- Override the onActivityResult method to retrieve the intent result, and use `getData()` to get the Uri of the image in the result:

```
protected void onActivityResult
    (int requestCode, int resultCode, Intent imageReturnedIntent) {
    ...
    Uri mSelectedImage = imageReturnedIntent.getData();
}
```

- Set the image's Uri, and use an intent with `ACTION_SEND`, `putExtra()`, and `setType()`:

```
Intent smsIntent = new Intent(Intent.ACTION_SEND);
smsIntent.putExtra(Intent.EXTRA_STREAM, mSelectedImage);
smsIntent.setType("image/*");
```

- Android Studio emulators can't pass MMS messages to and from each other. You must test this app on real Android devices.
- For more information about sending multimedia messages, see Sending MMS with Android.



Android Studio project: MMSChallenge

Summary

- To send an intent to an SMS messaging app with a phone number, your app needs to prepare a URI for the phone number as a string prefixed by "smsto:" (as in `smsto:14155551212`).
- Use an implicit intent with `ACTION_SENDTO` to launch an SMS app, and set the phone number and message for the intent with `setData()` and `putExtra`.
- To send SMS messages from within your app, add the `"android.permission.SEND_SMS"` permission to the AndroidManifest.xml file:
- Use the sendTextMessage() method of the SmsManager class to send the message, which takes the following parameters:
  - `destinationAddress`: The string for the phone number to receive the message.
  - `scAddress`: A string for the service center address, or `null` to use the current default Short Message Service Center (SMSC).
  - `smsMessage`: A string for the body of the message to send.
  - `sentIntent`: A `PendingIntent`. If not `null`, this is broadcast when the message is successfully sent or if the message failed.
  - `deliveryIntent`: A `PendingIntent`. If not `null`, this is broadcast when the message is delivered to the recipient.
- Use checkSelfPermission() to determine whether your app has been granted a particular permission by the user. If permission has *not* been granted, use the requestPermissions() method to display a standard dialog for the user to grant permission.
- Create a broadcast receiver to receive SMS messages using the onReceive() method of the BroadcastReceiver class.
- Add the `"android.provider.Telephony.SMS_RECEIVED"` intent filter between the `<receiver>` tags in AndroidManifest.xml to register your receiver for SMS messages.
- Use `getExtras()` to get the message from the intent:

```
Bundle bundle = intent.getExtras();
```

- Retrieve the messages from the PDU format:

```
Object[] pdus = (Object[]) bundle.get("pdus");
```

- Use the following `createFromPdu()` signature for Android version 6.0 (Marshmallow) and newer versions:

```
msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i], format);
```

- Use the following `createFromPdu()` signature for versions older than Android version 6.0:

```
msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
```

- To send an SMS message to an app running in an emulator, click the **…** (More) icon at the bottom of the emulator's toolbar on the right side, choose **Phone**, enter a message (or use the default "marshmallows" message), and click **Send Message**.

Related concept

- SMS Messages

Learn more

- Android Developer Reference:
  - Intent
  - Common Intents: Text Messaging
  - Intents and Intent Filters
  - SmsManager
  - Requesting Permissions at Run Time
  - checkSelfPermission
  - Run Apps on the Android Emulator
- Stack Overflow: Simulating incoming call or sms in Android Studio
- Android blog: Getting Your SMS Apps Ready for KitKat