## Run the app

Now run the app again by clicking **Apply Changes** https://developer.android.com/studio/images/buttons/toolbar-apply-changes.png in the toolbar. When it opens, type a message in the text field, and tap **Send** to see the message appear in the second activity.



## Add up navigation

Each screen in your app that is not the main entry point (all screens that are not the "home" screen) should provide navigation so the user can return to the logical parent screen in the app hierarchy by tapping the Up button in the app bar.

All you need to do is declare which activity is the logical parent in the AndroidManifest.xml file. So open the file at **app > manifests > AndroidManifest.xml**, locate the <activity> tag for DisplayMessageActivity and replace it with the following:

<activity android:name=".DisplayMessageActivity"  
          android:parentActivityName=".MainActivity">  
    <!-- The meta-data tag is required if you support API level 15 and lower -->  
    <meta-data  
        android:name="android.support.PARENT\_ACTIVITY"  
        android:value=".MainActivity" />  
</activity>

## Respond to the send button

Add a method to the MainActivity class that's called by the button as follows:

1. In the file **app > java > com.example.myfirstapp > MainActivity**, add the sendMessage() method stub as shown below:

### KOTLIN

### JAVA

public class MainActivity extends AppCompatActivity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity\_main);  
    }  
  
    **/\*\* Called when the user taps the Send button \*/  
    public void sendMessage(View view) {  
        // Do something in response to button  
    }**  
}

You may see an error because Android Studio cannot resolve the View class used as the method argument. So click to place your cursor on the View declaration, and then perform a Quick Fix by pressing Alt + Enter (or Option + Enter on Mac). (If a menu appears, select **Import class**.)

1. Now return to the **activity\_main.xml** file to call this method from the button:
   1. Click to select the button in the Layout Editor.
   2. In the **Attributes** window, locate the **onClick** property and select **sendMessage [MainActivity]** from the drop-down list.

Now when the button is tapped, the system calls the sendMessage() method.

Take note of the details in this method that are required in order for the system to recognize it as compatible with the android:onClick attribute. Specifically, the method has the following characteristics:

* Public access
* A void or, in Kotlin, an implicit unit return value
* A View as the only parameter (it is the View object that was clicked)

Next, you’ll fill in this method to read the contents of the text field and deliver that text to another activity.

## Build an Intent

An Intent is an object that provides runtime binding between separate components, such as two activities. TheIntent represents an app’s "intent to do something." You can use intents for a wide variety of tasks, but in this lesson, your intent starts another activity.

In MainActivity, add the EXTRA\_MESSAGE constant and the sendMessage() code, as shown here:

### KOTLIN

### JAVA

public class MainActivity extends AppCompatActivity {  
    **public static final String EXTRA\_MESSAGE = "com.example.myfirstapp.MESSAGE";**  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity\_main);  
    }  
  
    /\*\* Called when the user taps the Send button \*/  
    public void sendMessage(View view) {  
        **Intent intent = new Intent(this, DisplayMessageActivity.class);  
        EditText editText = (EditText) findViewById(R.id.editText);  
        String message = editText.getText().toString();  
        intent.putExtra(EXTRA\_MESSAGE, message);  
        startActivity(intent);**  
    }  
}

Android Studio again encounters **Cannot resolve symbol** errors, so press Alt + Enter (or Option + Return on Mac). Your imports should end up as the following:

### KOTLIN

### JAVA

import android.content.Intent;  
import android.support.v7.app.AppCompatActivity;  
import android.os.Bundle;  
import android.view.View;  
import android.widget.EditText;

An error remains for DisplayMessageActivity, but that's okay; you'll fix that in the next section.

Here's what's going on in sendMessage():

* The Intent constructor takes two parameters:
  + A Context as its first parameter (this is used because the Activity class is a subclass of Context)
  + The Class of the app component to which the system should deliver the Intent (in this case, the activity that should be started).
* The putExtra() method adds the EditText's value to the intent. An Intent can carry data types as key-value pairs called extras. Your key is a public constant EXTRA\_MESSAGE because the next activity uses the key to retrieve the text value. It's a good practice to define keys for intent extras using your app's package name as a prefix. This ensures the keys are unique, in case your app interacts with other apps.
* The startActivity() method starts an instance of the DisplayMessageActivity specified by the Intent. Now you need to create that class.

Create the second activity

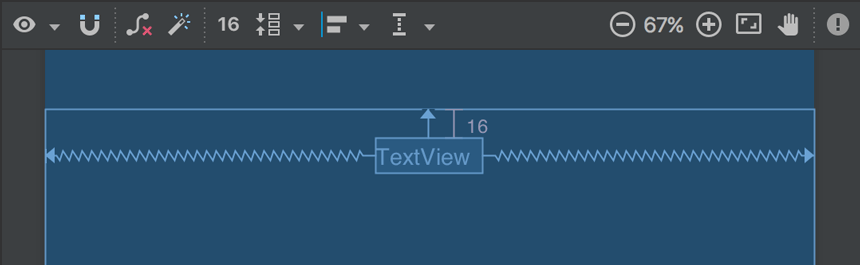
1. In the **Project** window, right-click the **app** folder and select **New > Activity > Empty Activity**.
2. In the **Configure Activity** window, enter "DisplayMessageActivity" for **Activity Name** and click **Finish** (leave all other properties set to the defaults).

Android Studio automatically does three things:

* Creates the DisplayMessageActivity file.
* Creates the corresponding activity\_display\_message.xml layout file.
* Adds the required <activity> element in AndroidManifest.xml.

If you run the app and tap the button on the first activity, the second activity starts but is empty. This is because the second activity uses the empty layout provided by the template.

Add a text view



**Figure 1.** The text view centered at the top of the layout

The new activity includes a blank layout file, so now you'll add a text view where the message will appear.

1. Open the file **app > res > layout > activity\_display\_message.xml**.
2. Click **Turn On Autoconnect** https://developer.android.com/studio/images/buttons/layout-editor-autoconnect-on.png in the toolbar (it should then be enabled, as shown in figure 1).
3. In the **Palette** window, click **Text** and then drag a **TextView** into the layout—drop it near the top-center of the layout so that it snaps to the vertical line that appears. Autoconnect adds left and right constraints to place the view in the horizontal center.
4. Create one more constraint from the top of the text view to the top of the layout, so it appears as shown in figure 1.

Optionally, make some adjustments to the text style by expanding **textAppearance** in the **Attributes** window and change attributes such as **textSize** and **textColor**.

## Display the message

Now you will modify the second activity to display the message that was passed by the first activity.

1. In DisplayMessageActivity, add the following code to the onCreate() method:

### JAVA

@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity\_display\_message);  
     **// Get the Intent that started this activity and extract the string  
    Intent intent = getIntent();  
    String message = intent.getStringExtra(MainActivity.EXTRA\_MESSAGE);  
  
    // Capture the layout's TextView and set the string as its text  
    TextView textView = findViewById(R.id.textView);  
    textView.setText(message);**  
}

1. Press Alt + Enter (or Option + Return on Mac) to import missing classes. Your imports should end up as the following:

import android.content.Intent;  
import android.support.v7.app.AppCompatActivity;  
import android.os.Bundle;  
import android.widget.TextView;

Activity-lifecycle concepts

callbacks:onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestroy(). The system invokes each of these callbacks as an activity enters a new state.



**Figure 1.** A simplified illustration of the activity lifecycle.

As the user begins to leave the activity, the system calls methods to dismantle the activity. In some cases, this dismantlement is only partial; the activity still resides in memory (such as when the user switches to another app), and can still come back to the foreground. If the user returns to that activity, the activity resumes from where the user left off. The system’s likelihood of killing a given process—along with the activities in it—depends on the state of the activity at the time. Activity state and ejection from memory provides more information on the relationship between state and vulnerability to ejection.

Depending on the complexity of your activity, you probably don't need to implement all the lifecycle methods. However, it's important that you understand each one and implement those that ensure your app behaves the way users expect.

The next section of this document provides detail on the callbacks that you use to handle transitions between states.

## Lifecycle callbacks

This section provides conceptual and implementation information about the callback methods used during the activity lifecycle.

Some actions, such as calling setContentView(), belong in the activity lifecycle methods themselves. However, the code implementing the actions of a dependent component should be placed in the component itself. To achieve this, you must make the dependent component lifecycle-aware. See Handling Lifecycles with Lifecycle-Aware Components to learn how to make your dependent components lifecycle-aware.

### onCreate()

You must implement this callback, which fires when the system first creates the activity. On activity creation, the activity enters the Created state. In the onCreate() method, you perform basic application startup logic that should happen only once for the entire life of the activity. For example, your implementation of onCreate() might bind data to lists, associate the activity with a ViewModel, and instantiate some class-scope variables. This method receives the parameter savedInstanceState, which is a Bundle object containing the activity's previously saved state. If the activity has never existed before, the value of the Bundle object is null.

If you have a lifecycle-aware component that is hooked up to the lifecycle of your activity it will receive the ON\_CREATEevent. The method annotated with @OnLifecycleEvent will be called so your lifecycle-aware component can perform any setup code it needs for the created state.

The following example of the onCreate() method shows fundamental setup for the activity, such as declaring the user interface (defined in an XML layout file), defining member variables, and configuring some of the UI. In this example, the XML layout file is specified by passing file’s resource ID R.layout.main\_activity to setContentView().

TextView mTextView;  
  
// some transient state for the activity instance  
String mGameState;  
  
@Override  
public void onCreate(Bundle savedInstanceState) {  
    // call the super class onCreate to complete the creation of activity like  
    // the view hierarchy  
    super.onCreate(savedInstanceState);  
  
    // recovering the instance state  
    if (savedInstanceState != null) {  
        mGameState = savedInstanceState.getString(GAME\_STATE\_KEY);  
    }  
  
    // set the user interface layout for this activity  
    // the layout file is defined in the project res/layout/main\_activity.xml file  
    setContentView(R.layout.main\_activity);  
  
    // initialize member TextView so we can manipulate it later  
    mTextView = (TextView) findViewById(R.id.text\_view);  
}  
  
// This callback is called only when there is a saved instance that is previously saved by using  
// onSaveInstanceState(). We restore some state in onCreate(), while we can optionally restore  
// other state here, possibly usable after onStart() has completed.  
// The savedInstanceState Bundle is same as the one used in onCreate().  
@Override  
public void onRestoreInstanceState(Bundle savedInstanceState) {  
    mTextView.setText(savedInstanceState.getString(TEXT\_VIEW\_KEY));  
}  
  
// invoked when the activity may be temporarily destroyed, save the instance state here  
@Override  
public void onSaveInstanceState(Bundle outState) {  
    outState.putString(GAME\_STATE\_KEY, mGameState);  
    outState.putString(TEXT\_VIEW\_KEY, mTextView.getText());  
  
    // call superclass to save any view hierarchy  
    super.onSaveInstanceState(outState);  
}