Extracted from:

Seven Mobile Apps in Seven Weeks

Native Apps, Multiple Platforms

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Seven Mobile Apps in Seven Weeks

Native Apps, Multiple Platforms

Tony Hillerson

Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

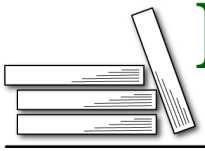
Seven Mobile Apps in Seven Weeks

Native Apps, Multiple Platforms

Tony Hillerson

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Candace Cunningham, Molly McBeath (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-148-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2016

Day 2: What Can I Get for a Buck?

Or, Building a Conversion Interface

We've got a solid foundation in place for getting data to and from the API. Now let's get into building an interface for the app.

Building an Interface

To allow the user to be able to do the conversion of one currency to another, we'll need an interface. Android's interface definition is done mostly in XML, with sensible layout language and a very powerful system for dealing with device differences. Let's start by building a form for the user to specify two currencies and convert an amount in one to an amount in the other.

Creating a Linear Layout

The activity_convert.xml layout is the view that ConvertActivity manages. Here's the first line of input fields for currencies.

Android/android_02_01_simple_phone_ui/CurrencyConverter/app/src/main/res/layout/activity_convert.xml

```
<TextView
    android:id="@+id/currency_label"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/currencies"
    />
<LinearLayout
    android:id="@+id/currencies"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <EditText
        android:id="@+id/from_currency"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1.0" />

    <EditText
        android:id="@+id/to_currency"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1.0" />
</LinearLayout>
```

A TextView simply displays text, an EditText is a text field, and a LinearLayout is a layout container that arranges views on the screen linearly. The ID in each tag allows the view to be referenced in code, and the @+id/foo syntax creates a new ID. We'll see how these IDs are made referenceable in just a bit.

Linear layouts are the most basic type of layout. I generally find myself converting linear layouts to the more powerful RelativeLayout, so keep that in mind for further research. However, this view will work fine as it is here.

Notice there is another really great feature of the platform in the text attribute of the TextView tag. Localization and internationalization are built right in—@string/currencies references a *string resource*, which you can see in this string resource file.

Android/android_02_01_simple_phone_ui/CurrencyConverter/app/src/main/res/values/strings.xml

```
<resources>
    <string name="app_name">Currency Converter</string>

    <string name="hello_world">Hello world!</string>
    <string name="action_settings">Settings</string>
    <string name="currencies">Currencies</string>
    <string name="amounts">Amounts</string>
    <string name="convert">Convert</string>
</resources>
```

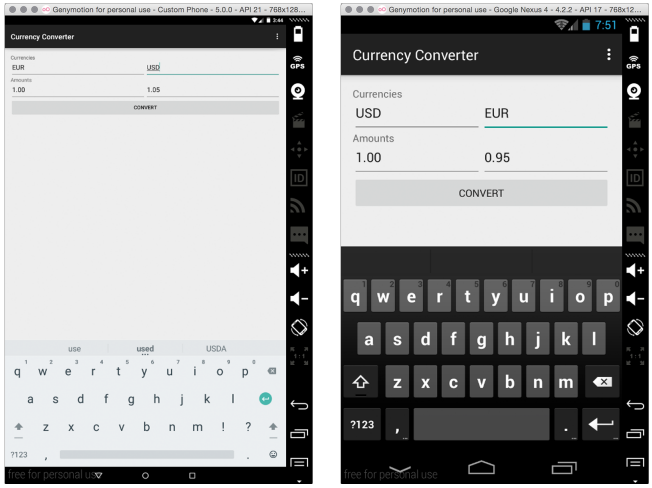
Android resources are important to understand because they're the method for dealing with not only localization but also layouts for different screen sizes and orientations and a lot of device differences and configurations.

Finally, to capture the user's intention to perform a conversion, we have a button in the UI.

Android/android_02_01_simple_phone_ui/CurrencyConverter/app/src/main/res/layout/activity_convert.xml

```
<Button
    android:id="@+id/convert_button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/convert"
    />
```

Again, note the definition of the ID and the use of the string resource. Here's how that UI looks on a large Android version 5.0 phone and on a regular version 4.2.2 phone.



Now let's look at how to get a handle on the view in the activity.

Controlling the View from the Activity

A lot of setup happens in onCreate(). For the activity to be able to control the view, it generally has to first grab on to a bunch of components in the view, so we'll do this in onCreate().

Android/android_02_02_edit_text_events/Currency … sevenapps/currencyconverter/ConvertActivity.java

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

  IntentFilter intentFilter =
    new IntentFilter(ConversionService.CONVERSION_RESULT_ACTION);
  LocalBroadcastManager.getInstance(this).
    registerReceiver(conversionReceiver, intentFilter);

  fromCurrencyField = (EditText) findViewById(R.id.from_currency);
  toCurrencyField = (EditText) findViewById(R.id.to_currency);
  fromAmountField = (EditText) findViewById(R.id.from_amount);
  toAmountField = (EditText) findViewById(R.id.to_amount);
  convertButton = (Button) findViewById(R.id.convert_button);

  fromAmountField.setText("1.00");
  toAmountField.setText("1.00");

  convertButton.setOnClickListener(new View.OnClickListener() {
    @Override public void onClick(View v) {
      convert();
    }
  });
}
```

The method setContentView() takes a layout ID and sets the activity's view to be the view described in the layout file. Notice this class, R, which is an autogenerated class holding named data about resources, such as layouts. Next, to wire up variables to components from the layout, we use findViewById(), which again uses R. The components are now accessible from the variable assignments. Next we set some default values in the amount fields.

Finally, we add a click listener to the conversion button.

The Convert button click handler calls convert(), which checks currenciesChanged() with some rudimentary caching to see if we need to pull down a rate again.

Android/android_02_02_edit_text_events/Currency ... sevenapps/currencyconverter/ConvertActivity.java

```
private void convert() {
  if (currenciesChanged()) {
    getRate();
  } else {
    calculateToAmount();
  }
}

private boolean currenciesChanged() {
  if (currentRate != null) {
    String from = fromCurrencyField.getText().toString().toLowerCase();
    String to = toCurrencyField.getText().toString().toLowerCase();
    if (from.equals(currentRate.from.toLowerCase()) &&
        to.equals(currentRate.to.toLowerCase())) {
      return false;
    }
  }
  return true;
}
```

That caching solution isn't great, we'll look deeper at it shortly. If the cached rate object turns out not to match the currency values, we load a new rate.

Android/android_02_02_edit_text_events/Currency ... sevenapps/currencyconverter/ConvertActivity.java

```
private void getRate() {
  String from = fromCurrencyField.getText().toString();
  String to = toCurrencyField.getText().toString();
  if (from != null && to != null && from.length() == 3 && to.length() == 3) {
    getRate(from, to);
  }
}

private void getRate(String from, String to) {
  Intent convertIntent = new Intent(this, ConversionService.class);
  convertIntent.putExtra(ConversionService.FROM, from);
  convertIntent.putExtra(ConversionService.TO, to);
  startService(convertIntent);
}

private void rateLoaded(ConversionRate newRate) {
  currentRate = newRate;
  calculateToAmount();
}

private void calculateToAmount() {
  if (currentRate != null) {
    Float toAmount = currentRate.convert(fromAmountField.getText().toString());
    String formattedToAmount = String.format("%.2f", toAmount);
    toAmountField.setText(formattedToAmount);
  }
}
```

This method uses the currency-conversion service, as we saw yesterday; there's nothing different here except for organization. The calculateToAmount() method uses the rate object to calculate a converted amount and then formats a string to put into the UI. Pretty straightforward. Let's add one more thing to make the caching solution work.

Saving Instance Data

This point about caching the rate object on the activity is related to something we saw yesterday: whenever a configuration change, such as a device rotation, occurs, the activity is destroyed and a new one is created. That means any data stored in instance variables won't be present in the new activity. Android has a process for dealing with this issue. Before an activity is destroyed, saveInstanceState() is called with a Bundle.

Android/android_02_03_save_instance_state/Curre ... sevenapps/currencyconverter/ConvertActivity.java

```
@Override protected void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);
  outState.putSerializable(CURRENT_RATE, currentRate);
}
```

The system will save the Bundle object and then hand it back into the onCreate() of the new activity. Then we need to check to see if we have some saved state and grab what we need out of it.

Android/android_02_03_save_instance_state/Curre ... sevenapps/currencyconverter/ConvertActivity.java

```
if (savedInstanceState != null) {
  currentRate = (ConversionRate) savedInstanceState.getSerializable(CURRENT_RATE);
} else {
  fromAmountField.setText("1.00");
  toAmountField.setText("1.00");
}
```

Now we've got a view in place that works pretty well on phones. It doesn't look the best on tablet-size screens, though. It's very easy to fix this using Android's resource system, so let's do that now.