



Building an Android User Interface

www.hyperiondev.com



Introduction

Welcome to the Building an Android User Interface Task!

Overview:

In this lesson, you'll be creating a layout in XML (eXtensive Markup Language) that includes a text field, which is used to display text on the screen, and a button. In the next lesson, we will build an app that responds when the button is pressed by sending the content of the text field to another activity.

Connect with your mentor



CONNECT

Remember that with our courses - you're not alone! You can contact your mentor to get support on any aspect of your course.

The best way to get help is to login to www.hyperiondev.com/support to start a chat with your mentor. You can also schedule a call or get support via email.



Your mentor is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!



A note from the Hyperion Team...

"I believe the mobile OS market will play out very similarly to Windows and Macintosh, with Android in the role of Windows. And so, if you want to be in front of the largest number of users, you need to be on Android." - Fred Wilson, co-founder of Union Square Ventures.

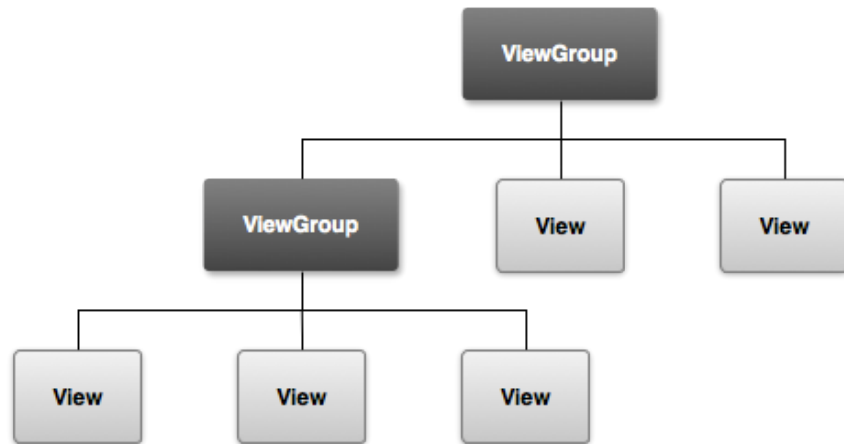
Android is currently the largest shareholder in the world of smartphones, so you can be confident that learning Android development really is a step towards reaching one of the largest software markets in the world. Once you start becoming proficient in Android development, you'll have the virtual world at your fingertips!

-The Hyperion Team

User Interfaces

The graphical user interface (GUI) for an Android app is built using a hierarchy of View and ViewGroup objects. If you remember from earlier in the course, a View is an object that displays images/information to the screen. ViewGroup objects are invisible containers that define the structure of how visible View objects are laid out, such as grid or vertical list.

Android provides an XML vocabulary that corresponds to the subclasses of View and ViewGroup so you can define your UI in XML using a hierarchy of UI elements, as illustrated in the image below. Layouts are subclasses of the ViewGroup. In this exercise, you'll work with a LinearLayout.



Instructions

Note: This task will be building onto the previous task's application.

Create a **LinearLayout**:

1. From the **res/layout/** directory, open the **activity_main.xml** file.
2. This XML file defines the layout of your activity. It contains the default "Hello Hyperion" text view from the *Creating your first Android App* task.
3. Open a layout file. You are first shown the design editor in the Layout Editor. For this lesson, you work directly with the XML, so click the **Text** tab at the bottom to switch to the text editor.
4. Replace the contents of the file with the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">
</LinearLayout>
```

LinearLayout is a view group (a subclass of `ViewGroup`) that lays out child views in either a vertical or horizontal orientation, as specified by the **android:orientation** attribute. Each child of a `LinearLayout` appears on the screen in the order in which it appears in the XML.

Two other attributes, **android:layout_width** and **android:layout_height**, are required for all views in order to specify their size.

Because the `LinearLayout` is the base view in the layout, it should fill the entire screen area that's available to the app. We do this by setting the width and height to **"match_parent"**. This value declares that the view should expand its width or height to match the width or height of the parent view.

- In the `activity_main.xml` file, within the **<LinearLayout>** element, add the following **<EditText>** element:

```
<EditText
    android:id="@+id/edit_message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:hint="@string/edit_message" />
```

Here is a description of the attributes in the **<EditText>** you added:

android:id

This provides a unique identifier for the view, which you can use to reference the object from your app code, such as to read and manipulate the object (you'll see this in the next lesson).

The at sign (**@**) is required when you're referring to any resource object, like the `TextView` from the previous task, from XML. It is followed by the resource type (**id** in this case), a slash, then the resource name (**edit_message**).

The plus sign (**+**) before the resource type is needed only when you're defining a resource ID for the first time. When you compile the app, the SDK tools use the ID name to create a new resource ID in your project's **gen/R.java** file that refers to the `EditText` element.

With the resource ID declared once this way, other references to the ID do not need the plus sign. Using the plus sign is necessary only when specifying a new resource ID and is not needed for concrete resources such as strings or layouts. See the side box for more information about resource objects.

android:layout_width and **android:layout_height**:

Instead of using specific sizes for the width and height, the "**wrap_content**" value specifies that the view should be only as big as needed to fit the contents of the view. If you were to instead use "**match_parent**", then the EditText element would fill the screen, because it would match the size of the parent LinearLayout.

android:hint

This is a default string to display when the text field is empty. Instead of using a hard-coded string as the value, the "**@string/edit_message**" value refers to a string resource defined in a separate file. Because this refers to a concrete resource (not just an identifier), it does not need the plus sign. However, because you haven't defined the string resource yet, you'll see a compiler error at first. You'll fix this in the next section by defining the string.

Note: This string resource has the same name as the element ID: **edit_message**. However, references to resources are always scoped by the resource type (such as **id** or **string**), so using the same name does not cause any problems.

Adding String Resources

By default, your Android project includes a string resource file at **res/values/strings.xml**. Here, you'll add a new string named "**edit_message**" and set the value to "Edit Message."

- From the res/values directory, open **strings.xml**.
- Add a line for a string named "**edit_message**" with the value, "**Edit Message**".
- Add a line for a string named "**button_send**" with the value, "**Send**".

The result for **strings.xml** looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">My First App</string>
    <string name="edit_message">Edit Message</string>
    <string name="button_send">Send</string>
</resources>
```

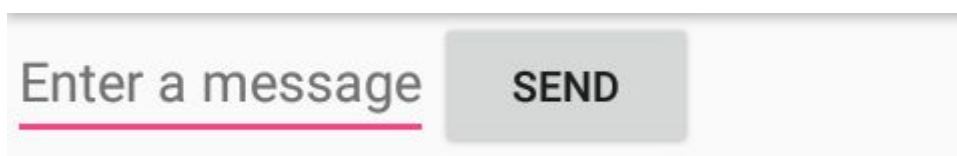
- In **activity_main.xml** file add a button after the **EditText**.

```
<Button
    android:id="@+id/send_button"
```

```
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="@string/button_send" />
```

The attributes you just added have the same description as the ones on **EditText**, except for **android:text** attribute, which represents what should be written on the button we just created.

The layout is currently designed so that both the **EditText** and **Button** widgets are only as big as necessary to fit their content as follows:



Your code at this point should look like this:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <EditText android:id="@+id/edit_message"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:hint="@string/edit_message" />
    <Button
        android:id="@+id/send_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_send" />
</LinearLayout>
```

This works fine for the button, but not as well for the text field, because the user might type something longer. It would be nice to fill the unused screen width with the text field. You can do this inside a **LinearLayout** with the **weight** property, which you can specify using the **android:layout_weight** attribute.

The weight value is a number that specifies the amount of remaining space each view should consume, relative to the amount consumed by sibling views. This works kind of like the amount of ingredients in a drink recipe: "2 parts soda, 1 part syrup" means two-thirds of the drink is soda. For example, if you give one view a weight of 2 and

another one a weight of 1, the sum is 3, so the first view fills 2/3 of the remaining space and the second view fills the rest. If you add a third view and give it a weight of 1, then the first view (with weight of 2) now gets 1/2 the remaining space, while the remaining two each get 1/4.

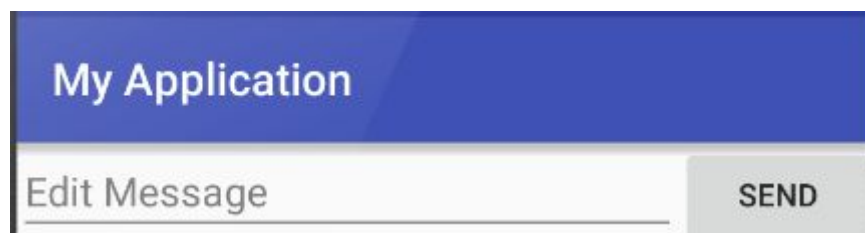
The default weight for all views is 0, so if you specify any weight value greater than 0 to only one view, then that view fills whatever space remains after all views are given the space they require. Setting the width to zero (0dp) improves layout performance because using "**wrap_content**" as the width requires the system to calculate a width. This is irrelevant because the weight value requires another width calculation to fill the remaining space.

Making the Input Box Fill in the Screen Width:

In **activity_main.xml**, modify the **EditText** so that the attributes look like this:

```
<EditText android:id="@+id/edit_message"
    android:layout_weight="1"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:hint="@string/edit_message" />
```

Your layout should look like this:



This is how your code should look:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <EditText android:id="@+id/edit_message"
        android:layout_weight="1"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:hint="@string/edit_message" />
```



```
<Button
    android:id="@+id/send_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_send" />
</LinearLayout>
```

Creating Customised View Styles

Sometimes android studio may not necessarily have the desired theme that you wish to implement for some of your Views, and we'd generally like to avoid having to retype the same attributes over and over. Luckily, you're given the ability to create your own customised styles which can then be applied to any View quickly and easily.

To create a set of styles, save an XML file in the **res/values/** directory of your project. You can name this file as you wish, but it must use the **.xml** extension.

The base node of the XML file must be **<resources>**. This is somewhat different to what you've been accustomed to when working the XML files, but is necessary for the style files.

For each style you want to add, create a **<style>** element in the file with a name that uniquely identifies the style (note that this attribute is **compulsory**). Then add an **<item>** element for each property you want as part of that style, with a name that declares the style property and a value to go with it. The value for the **<item>** can be a keyword string, a hex color, a reference to another resource type, or other value depending on the style property.

We can also make our styles extend other styles that already exist. This is useful if we want to use some attributes from an already existing style, and adjust or add to the existing attributes without recreating the entire style. We do this by using the parent attribute and setting it to the name of the style to extend.

Let's look at an example of a basic style which has been created below:

```
<?xml version="1.0" encoding="utf-8" ?>
<resources>
```

```

<style name="Orig" parent="@android:style/TextAppearance.Medium">
  <item name="android:layout_width">fill_parent</item>
  <item name="android:layout_height">wrap_content</item>
  <item name="android:textColor">#00FF00</item>
  <item name="android:typeface">monospace</item>
</style>
</resources>

```

Next, we'll see how it would be appended to a TextView:

```

<TextView
  style="@style/Orig"
  android:text="@string/hello" />

```

Using styles can be very useful not only for making your app appear more attractive while saving you typing and from possible debugging, it also makes it very easy to bundle different elements of your UI together. Applying the same style to various UI elements can make it easier for your user to see what elements are functionally similar or connected at a simple glance, making your app more functional and user friendly.

Applying Content Descriptions to Views for Accessibility

Content Descriptions are used to make your application accessible to people with disabilities.. They are used in the **TextToSpeech** functionality for those who can't see well or at all and verbally informs the user of what they are currently touching on the screen. Thus you are given the tools to create an application that caters for all audiences.


Implementing a Content Description is simple and implemented by a single line of code through the **activity.xml** since it is a UI functionality:

```

<Button
  android:id="@+id/send_button"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="@string/button_send"
  android:contentDescription="@string/send"
/>

```

We have now added a content descriptor to the send button which will inform the user through audio communication what the button does..

This layout is applied by the **MainActivity.java** class that was generated when you created the project, click **Run**  **'app'** in the toolbar. This will take a couple of minutes depending on your system.

Create a RelativeLayout

1. Open **activity_main.xml**
2. Replace the contents of the file with the following code:

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <EditText android:id="@+id/edit_message"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/edit_message" />
    <Button
        android:id="@+id/send_button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/button_send" />
    <TextView
        android:id="@+id/messageTV"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/app_name" />
</RelativeLayout>
```

What we just created above is a **RelativeLayout**, a ViewGroup that displays child views in relative positions. The position of each view can be specified as relative to sibling elements (views to the left-of or below another view) or in positions relative to the parent RelativeLayout area (such as aligned to the bottom, left or center).

A RelativeLayout is a very powerful utility for designing a user interface because it can eliminate nested view groups and keep your layout hierarchy flat.

Let's add a TextView just below the button. Add this attribute to the Button and the TextView respectively;

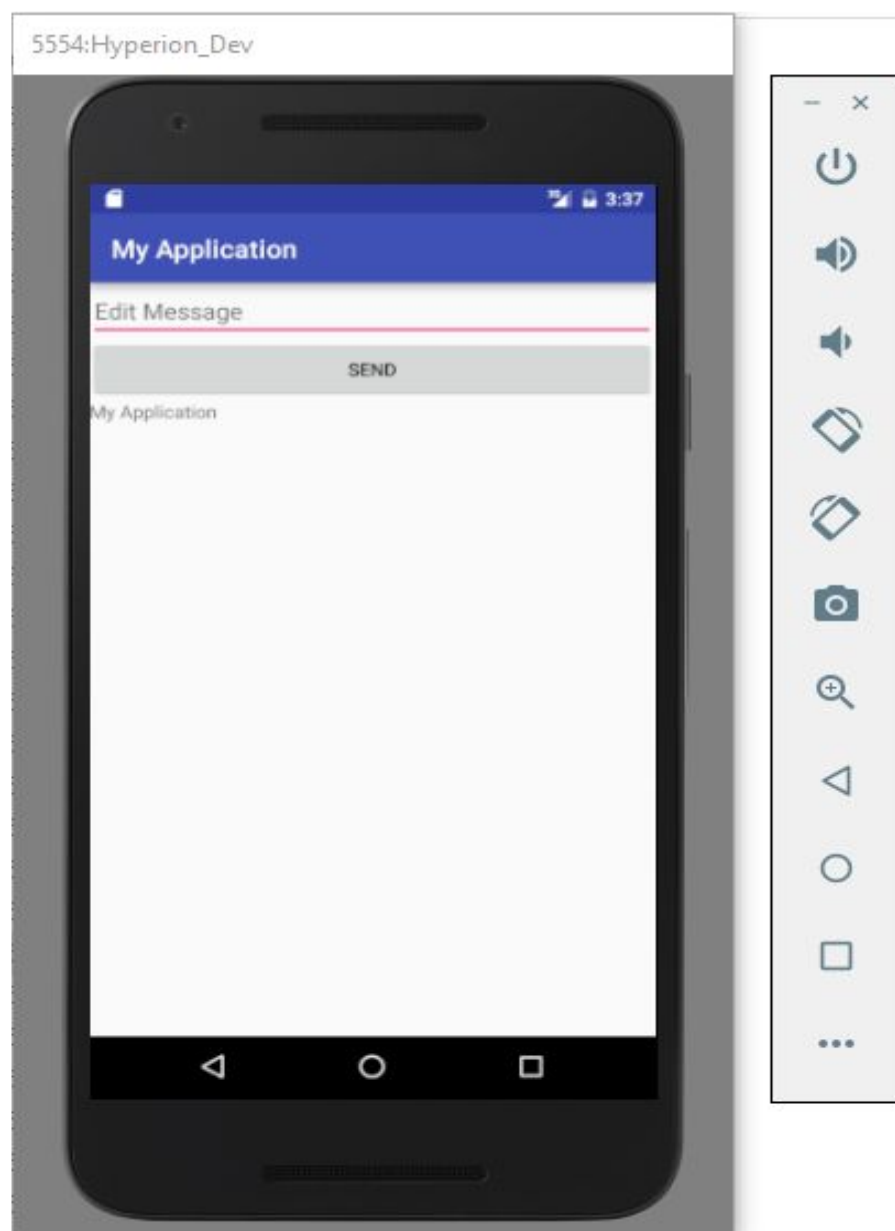
```
//This one goes on the Button  
android:layout_below="@id/edit_message"
```

```
//This one goes on the TextView  
android:layout_below="@id/edit_message"
```

Your code should now look like this:

```
<RelativeLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:tools="http://schemas.android.com/tools"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
    <EditText android:id="@+id/edit_message"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:hint="@string/edit_message"  
    />  
  
    <Button  
        android:id="@+id/send_button"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:text="@string/button_send"  
        android:layout_below="@id/edit_message"  
    />  
  
    <TextView  
        android:id="@+id/messageTV"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:text="@string/app_name"  
    />  
  
</RelativeLayout>
```

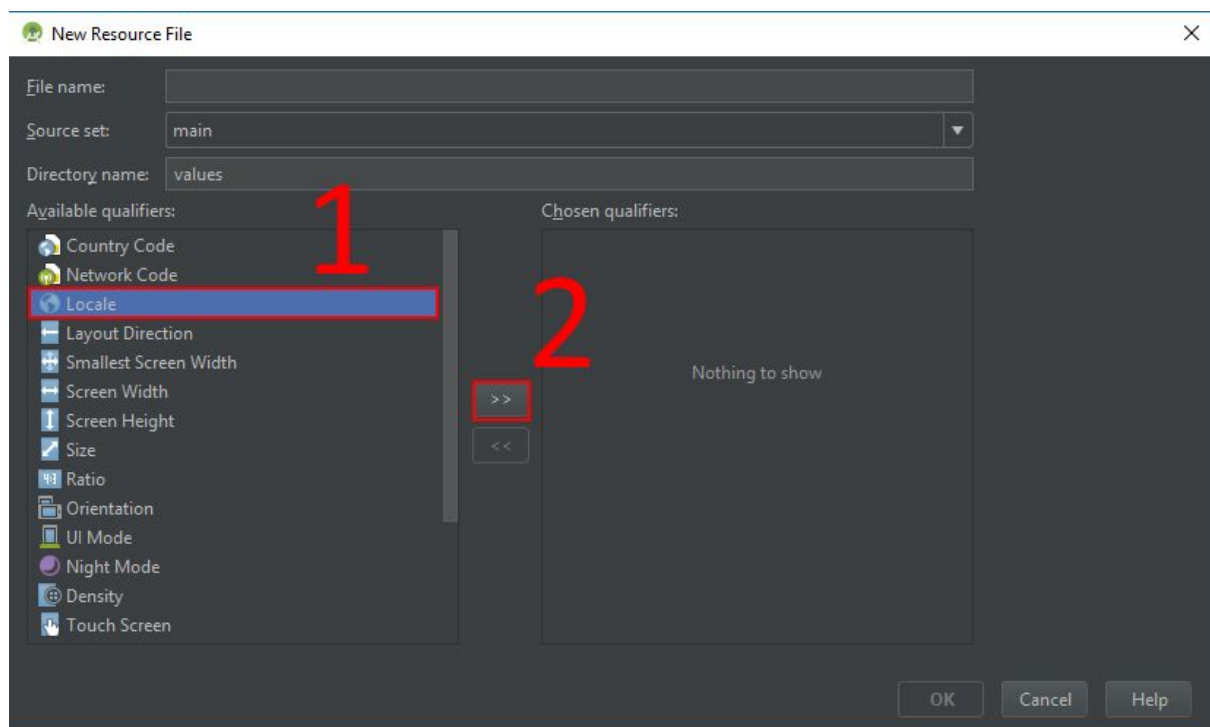
Now run you app to see the difference. This is how it should look::



Localising User Interface Text

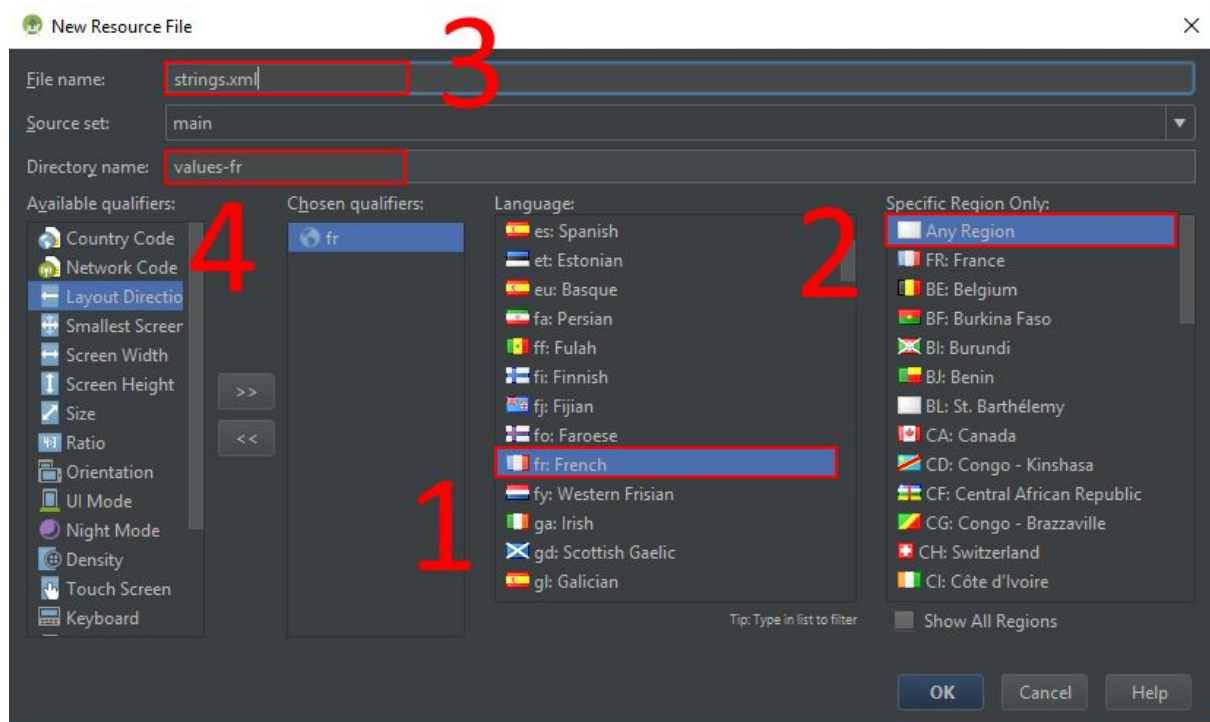
An important characteristic of many widely used apps is that they are able to run in multiple regions of the world. In order to accomplish this you also need to ensure that your app can be used in multiple languages around that world. While this may seem daunting, it is actually simply a matter of translating your app and storing the different translations for each language in their own independent value's XML file.

Up to now we've been using **strings.xml** in the **res/values/** folder to add strings we use in our user interface. This is actually just the default file however. If we want to add support for multiple localities, we will need to create a new resource file for that specific region or language. Suppose we wanted to now allow French users to use our app, let's create a new **strings.xml** file for the French language. To do this, right click the **values** directory in your Project window. Then select **New > Values Resource File**. This will bring up a window allowing you to create a new resource file. Firstly, we want to create a file for a new locale, so select the **locale** (1) under the "Available Qualifiers" list, and press the right facing arrows (2)



After this, look for **French** under "Languages" and select it (1). Because we do not care which region the language is used in, we will simply select any region (2). The region specifier allows us to have even more control over the language, as we can use different value files for a French device in France to that of a French device in Belgium etc. Next because we are creating a new strings.xml file, we will name the file "**strings.xml**" (3). Ensure that it is spelled correctly, as otherwise Android will not detect it. The directory

name should already have changed to “values-fr” automatically, however if it has not, change it now (4). This signifies to Android that this is the directory it should look under for the French locality.



After this, select “OK” and you should now see a new file called strings.xml (fr) under the new strings.xml folder. Open up this file and it should look the same as the default strings.xml file appeared initially (without the app name included). Any string you would now like to add a translation for can now be placed inside the strings.xml (fr) file. Ensure that the name in the fr file exactly matches the name of the string in the default file, otherwise Android will not use it correctly.

We now only need to add any strings we need to this file with the appropriate translation, however we do not need to translate every string. If we have 500 strings in the default file, but would only like to translate 50, we only need to add those respective strings to our new XML file. We also do not need to worry about changing anything in our actual Activity XML or Java files. Android will automatically find the correct string depending on the locality using the string name we are trying to access:

- If the string for that locality exists, Android will use it.
- If the string does not exist for that locality, Android will use the string from the default strings.xml file.
- If the string does not exist in the strings.xml file, it will cause an error.

This localization functionality works with all value types, not only strings. This means you can also display different images, colours, styles and layouts for different localities around the world easily.

You now know how to implement a custom user interface for your Android app. Here is challenge to test the skills you've acquired.

Compulsory Task

Follow these steps:

- Modify EditText hint to say "Type your Message here"
- Add a Content Description for the EditText which notifies them of what they should do
- Modify the Button Text to say "Submit"
- Create a custom style for the Button
- Try the `android:layout_alignParentBottom="true"` on the TextView, Remove the `android:layout_align_below` attribute
- Run your app to see the difference

Give your thoughts:



RATE

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes. Think there's something we can change or improve? Want to tell us about something you've enjoyed? We're listening to what you have to say!

[Click here](#) to share your thoughts anonymously.