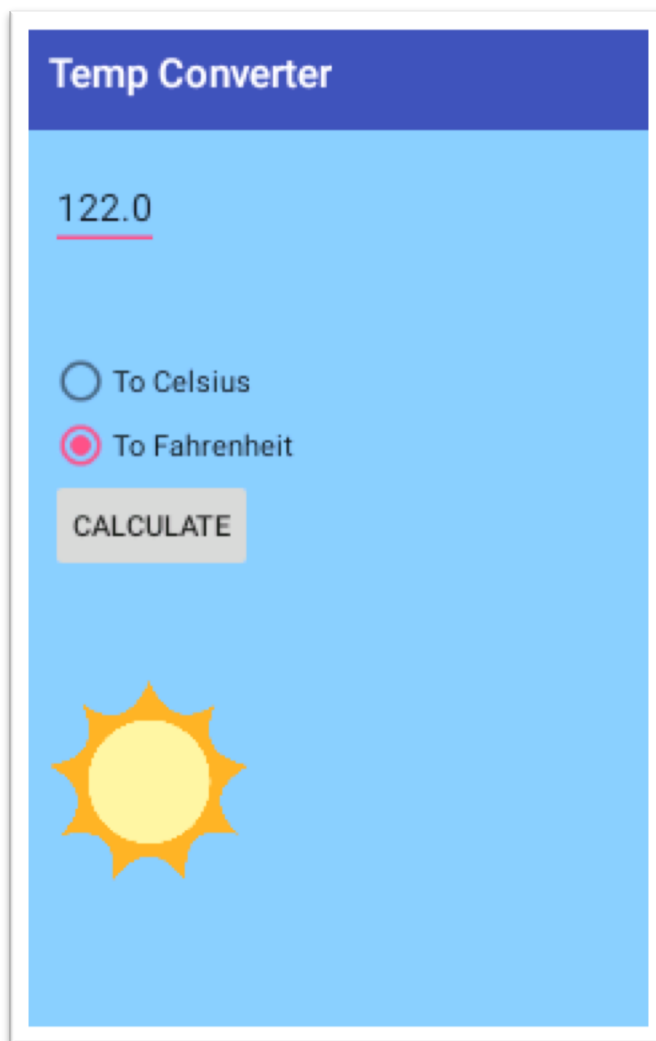


**Temperature Converter App- 50 points**

**Introduction.** This lab will have you create a simple temperature conversion app! Instructions include how to drag and drop into a layout view, to add User Interface (UI) components to the view, add/set properties for your components as well as manually add and edit various files. Also included is the functionality of the app that will be applied with an added Java class.

Controls for this app include EditText, Button, RadioGroup, RadioButtons and an ImageView. Interface of the app at runtime shown below, is what you will be similarly building for this lab.

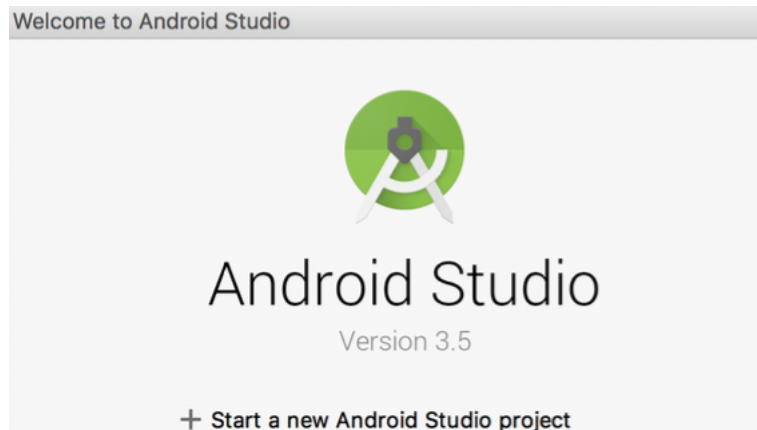


## STEP 1 Creating a New Android Project

Startup Android Studio.

Use the [Create New Project](#) wizard to create your project by doing the following:

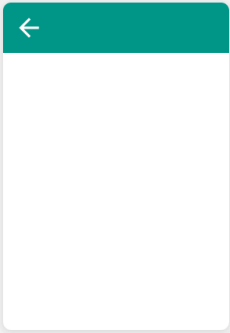
Click on the option '**Start a new Android Studio project**' seen on the Welcome screen if you are confronted with a Splash screen, otherwise just go to your menu and choose **New > New Project**.



Here a New Project dialog box appears where you can configure your new app. At the **Choose your project** screen, highlight **Empty Activity** and press [Next](#).

Enter in the Application name as shown below, and a project location of your choice, keep all other settings as defaulted then chose [Next](#). Then click [Finish](#) when complete.

### Configure your project



Empty Activity


Name

Package name

Save location

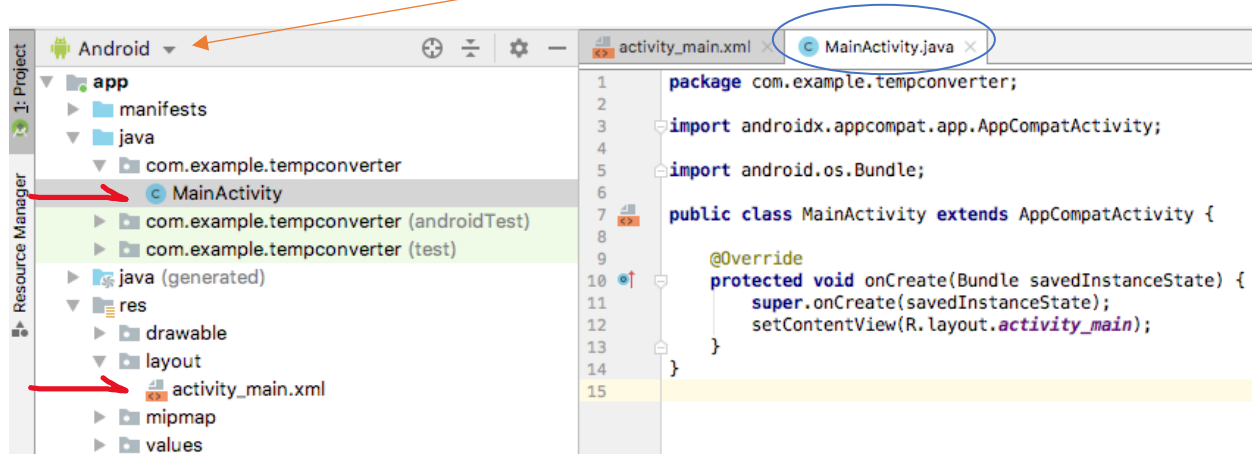
Language

Minimum API level

 Your app will run on approximately 6.0% of devices.

(*Note*- the Name will serve as the title of your app when it runs)

Now as your project starts, a sync of a 'gradle' building process occurs for a few seconds. Every starter project starts off with a few files to work with namely a **MainActivity.java** file and a *corresponding* layout file named **activity\_main.xml**. Sample folder tree follows showing file locations within your navigation tree. Familiarize yourself with the IDE a moment, its layout, menus, values, etc. Notice the nice tab structure (known as the Navigation Bar) tool for quick navigation to your files. MainActivity should be now displaying on your screen.



## STEP 2 Creating project attributes (with a common xml file)

We will start our project showing how Android allows you to create static resources that define attributes, e.g., for Strings or colors, etc. These attributes can be defined in XML files or by Java source code.

Select (double click) from your project tree, the **res/values/strings.xml** file to open the editor for this file.

As a start, add in a color tag by adding the following line to your strings.xml file within your <resources> root tag


```
<color name="myColor">#FFE4E1</color>
```

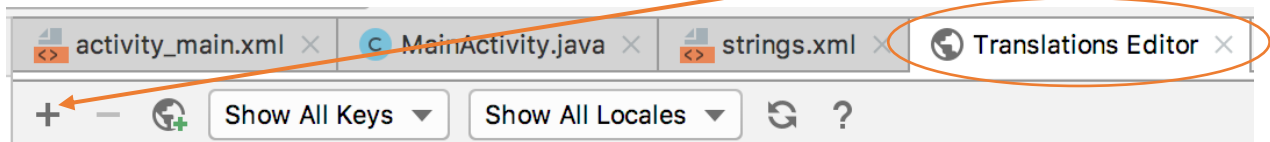
Notice what has been entered. **myColor** value has been included for the attribute name and the inner XML value shows the Hex color of **#FFE4E1**. This will eventually be used to serve as a nice *mistyrose* background color to your app.

Next add in some strings that will contain attribute values for common temperature displays. To add in various string attributes quickly without typing in any XML, you can simply click on the [Open editor](#) link towards the right top side of your IDE.




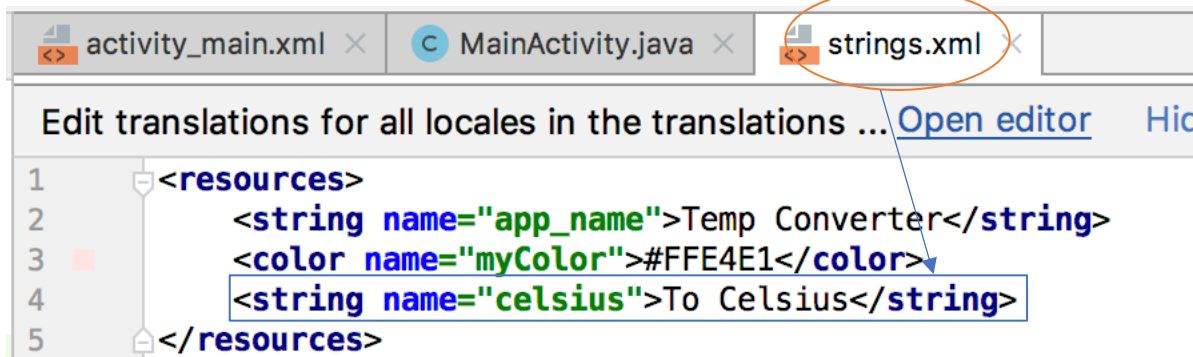
This will allow you to simply and rapidly add in a key/value pair denoting programmer defined values.

At this point, enter in your key/value data by pressing the Add Key  symbol within your Translations Editor tab...



and enter the following into your pop up.

Click on  to commit your information. Your resource file (strings.xml) now has been updated to include your new string data as shown below in the rectangle symbol.

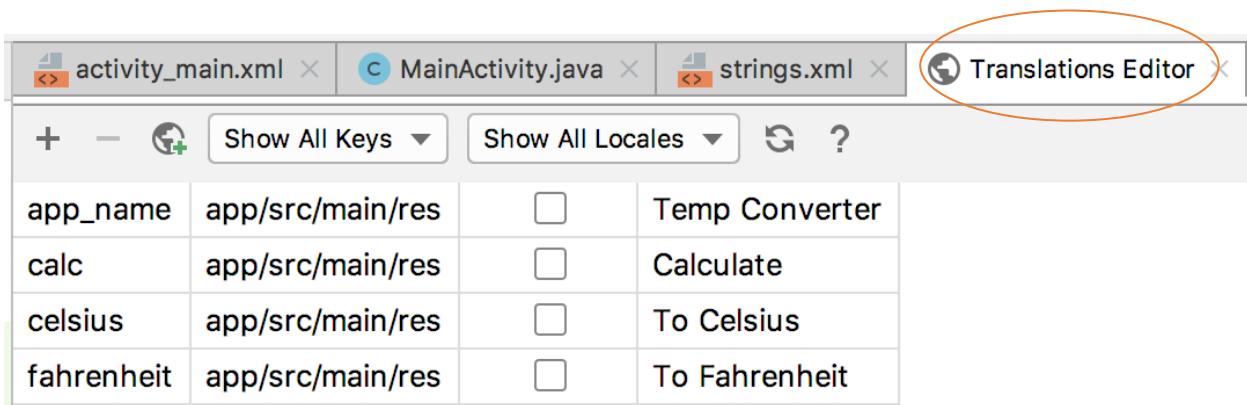


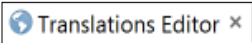
Do the same action and add in the following two additional attributes to your colors xml file that follow

### String Attributes add-ons

Key	Default Value
fahrenheit	To Fahrenheit
calc	Calculate

Notice the Translation Editor view now depicts all your new entries as well as the default key namely app\_name! Snapshot follows. Note your order by Keys shown next, may vary.



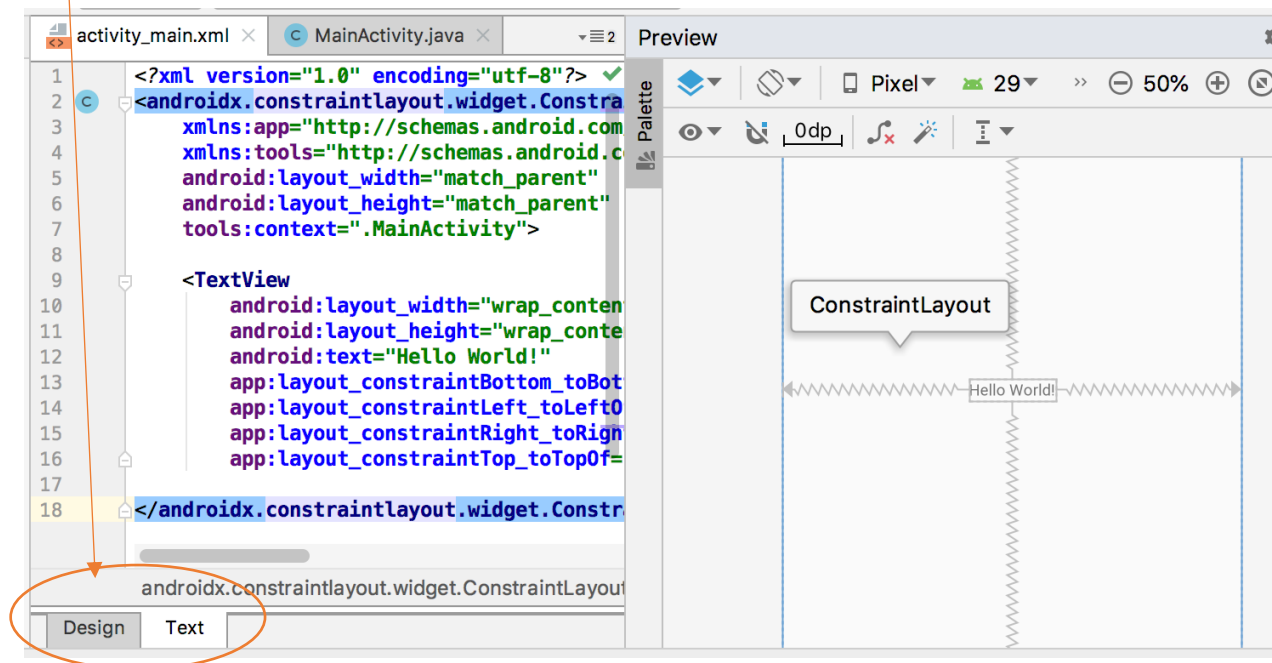
Once you update your XML settings, close the Translations Editor tab  to view and validate your edited XML file. It should now resemble something like this.

```
<resources>
  <string name="app_name">Temp Converter</string>
  <color name="myColor">#FFE4E1</color>
  <string name="celsius">To Celsius</string>
  <string name="fahrenheit">To Fahrenheit</string>
  <string name="calc">Calculate</string>
</resources>
```

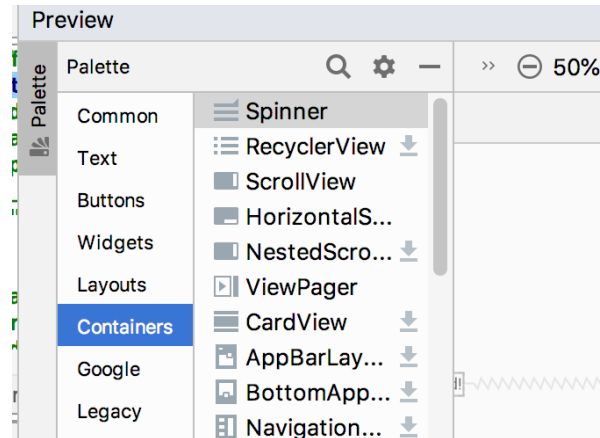
If something is off, feel free to edit your results right in the XML file. A good idea would be to save your files at this point and any time really when editing, updating or coding any files.

### STEP 3 Using the layout editor for setting a UI

Select (double-click) the **res/layout/activity\_main.xml** file. The associated Android editor allows you to create the physical layout via drag and drop in **Design** view and/or via the XML source code inside your **Text** view! You can switch between both representations via the tabs at the bottom of the editor.



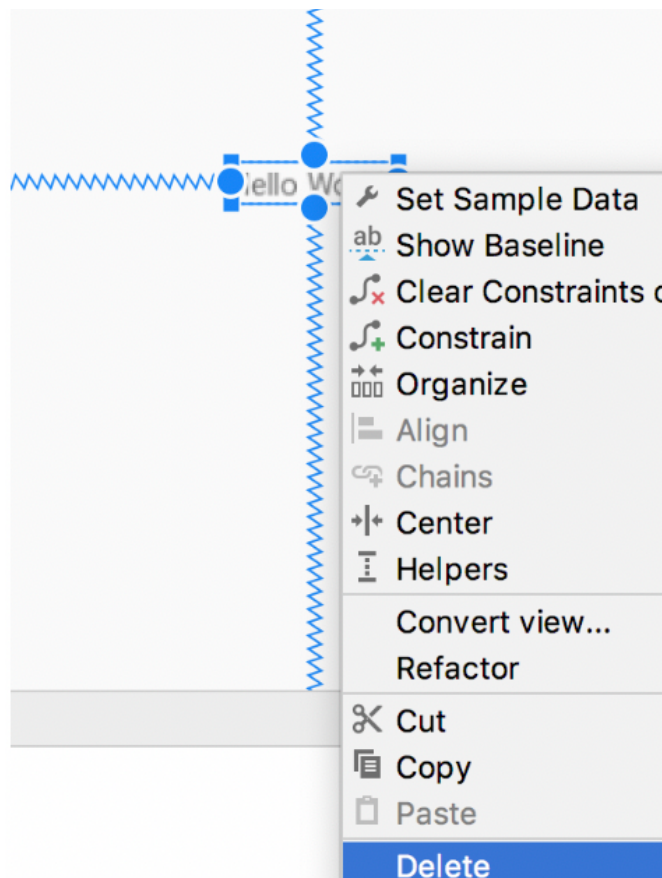
The following shows a snapshot of a Palette when in Design mode. This feature allows you to drag and drop new **View** elements to your layout. Simply click on any of the Palette items to view it contents.



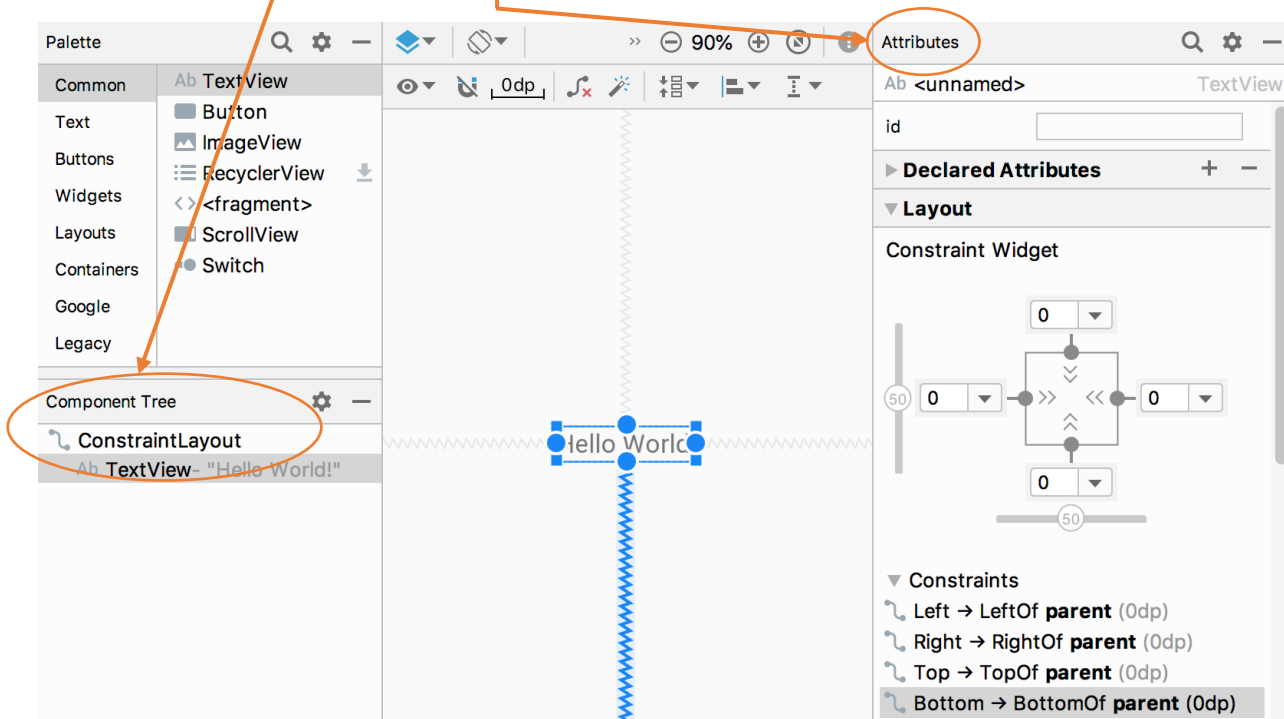
#### STEP 4 Adding View elements to your layout file

Layout views in Android allows for the addition of user interface components. Here you will create the base user interface (UI) for your application.

From your Design view, right-click on the existing Hello World! text object in the layout. Select Delete from the popup menu to remove the text object. Snapshot follows.

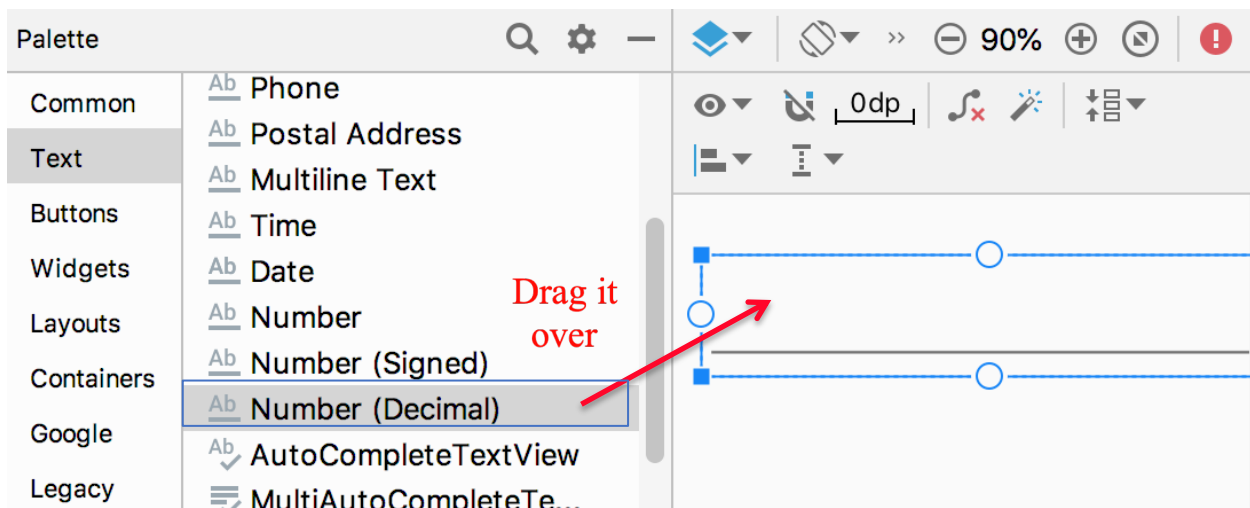


Get familiar with your surroundings concerning your Layout views! For example, in **Design** only view, check your Component Tree and Attributes (or properties) sections to the left/right of your Design interface.



Use the Component Tree section to quickly 'pinpoint' elements from your Tree layout or even add elements from your **Palette** directly by dragging objects from the Palette right within your tree! Very cool indeed. Also use the Attributes section to quickly 'tweak' elements for such things as height or width adjustments to your layout design elements, etc. in a snap! No fooling around with XML. You'll see working these sections next.

Now from your Palette, select the **Text** fields section and note all the text field choices. Locate the **Number (Decimal)** choice and drag it onto your layout (interface) thus creating an editable text input field as your first element added to your interface.



If you view your XML on this, you will notice that the Text Field element has been assigned a new **android:id** attribute as follows:

**android:id="@+id/editText"**

---

## Keynote



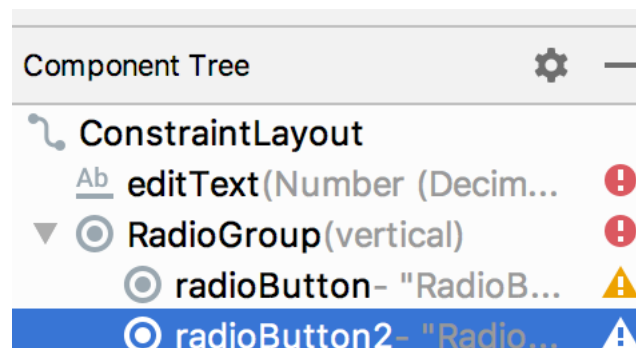
All entries in the Text Fields section of your XML file you'll see, define text fields. Various entries define "*added*" attributes for them, ex. if a text field should allow input only for numbers, or allow for negative values as well, etc. Check it:

```
<EditText
    android:id="@+id/editText"
    ::
    android:inputType="numberDecimal"
    tools:layout_editor_absoluteX="48dp"
    tools:layout_editor_absoluteY="34dp" />
```

---

in design view, select the **Buttons** section in the Palette and drag a **RadioGroup** entry into your layout and place it directly under your text view (it should snap right in). Notice how your Component Tree area tree looks each time your adding elements in. You can always click on an element in your tree to quickly focus on it to delete it or view its specific property attributes.

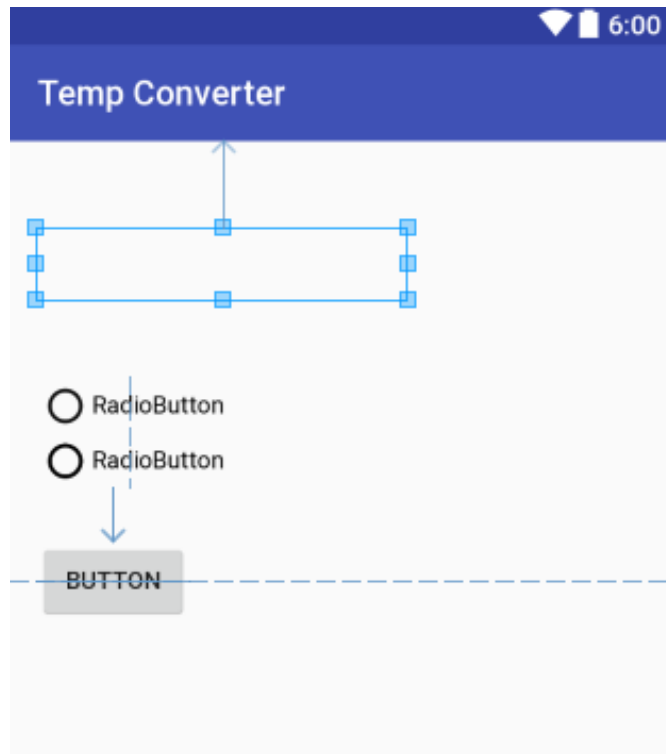
Next grab two **RadioButton** objects from your Buttons section from your Palette and drag each button directly into your RadioGroup within your **Component Tree** (as shown below) to make it easier for direct placement of these elements. You want RadioButtons to be part of your RadioGroup making coding easier to see what a user has selected.



Notice that the id's are given (as shown above) automatically when you've dragged them into your Tree, namely **radioButton** and **radioButton2**. Id's are always important so you can reach out or refer to the elements in code later on. Notice also default text is given to the radio buttons. That will be adjusted soon in your Attributes view.

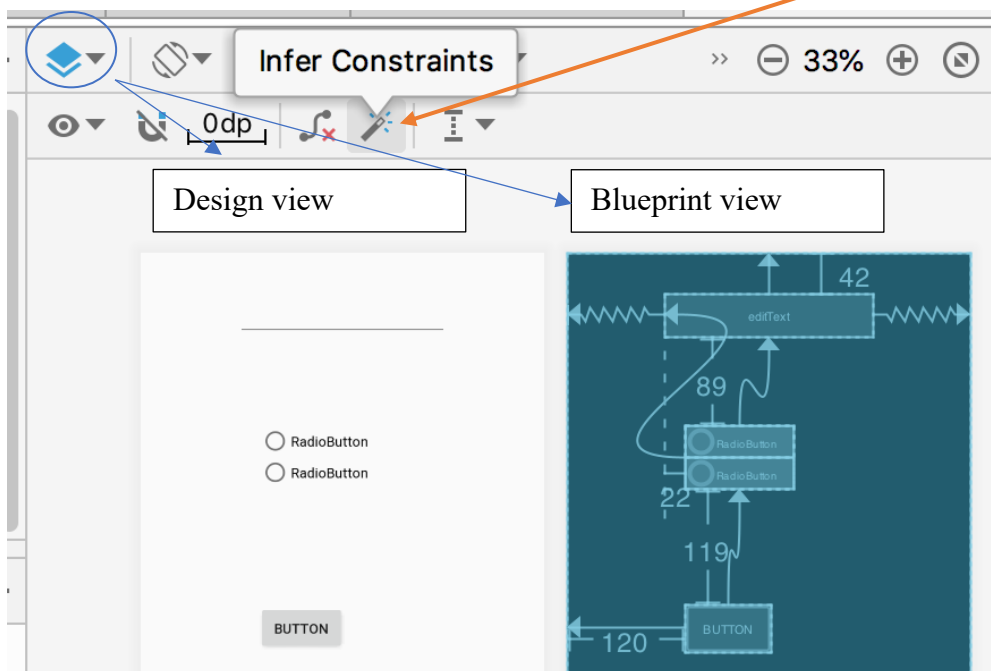
Finally drag a Button from the Buttons section onto the screen layout area under your RadioGroup. The button should now sit nicely right under RadioGroup element as shown next!





Make sure to include adequate spacing between your widgets to ensure your controls don't bunch up in your layout. You can move your elements around free as you wish in your Design view at this point.

Now a cool thing about the constraint layout is that it can infer positioning between elements. To ensure a nice layout at runtime to conform to your Design view, click on the "Magic Wand" to set your constraints automatically! Notice the update to your Blueprint view as well.



The resulting XML layout should look something similar like the following. Notice some of the constraints added in shown below bounded with a rectangle! NOTE: Feel free to use the magic wand anytime adding any additional constraints as well to autofit your elements relative to one another! If necessary "manually" edit tag information so constraints look fitting for the activity screen.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <EditText
        android:id="@+id/editText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="42dp"
        android:ems="10"
        android:inputType="numberDecimal"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <RadioGroup
        android:id="@+id/radioGroup"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="22dp"
        android:layout_marginTop="89dp"
        app:layout_constraintStart_toStartOf="@+id/editText"
        app:layout_constraintTop_toBottomOf="@+id/editText">

        <RadioButton
            android:id="@+id/radioButton"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="RadioButton" />

        <RadioButton
            android:id="@+id/radioButton2"
            :: />
    </RadioGroup>

    <Button
        android:id="@+id/button"
        :: />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Pause a moment here please to study the XML tree listing. Check all the attributes for each given element starting from the root of the tree! A great learning experience indeed! **Note if you run your app at this time, you should see a perfect layout according to your design!**

## STEP 5 Editing your view properties

Time to do some tweaks to your view to make everything look perfect. Here you'll start off doing some more work (edits) in your XML file to get used to things.

Switch to the XML file Text view and override the **android:text** attribute value of the first radio button and assign the **@string/celsius** value to the attribute. Assign also the Fahrenheit string attribute to the text property of the second radio button. Good idea to use **IntelliSense** when assigning values to properties to ensure correct values are assigned! Sample XML element appearances follows...

```
<RadioButton
    android:id="@+id/radioButton"
    ::
    android:text="@string/celsius" />

<RadioButton
    android:id="@+id/radioButton2"
    ::
    android:text="@string/fahrenheit" />
```

Note a few things here. Again we surround our RadioButtons within a RadioGroup so the app only allows for one selection at a time and therefore the options are not mutually exclusive which is what we want in this case.

**Note also that XML like HTML, need tag terminations ( /> ) for each element defined, so please be aware of that special syntax needed.**

Further note that your XML attributes properties may sometimes differ from your view objects versus with what is shown in the lab examples as is the case perhaps when you drag and drop controls around especially as you may be picking up some slightly differing layout height, width, and alignment attribute settings, etc. thus making it seem that your settings are a bit different. That's okay as it's your design so not to worry!!



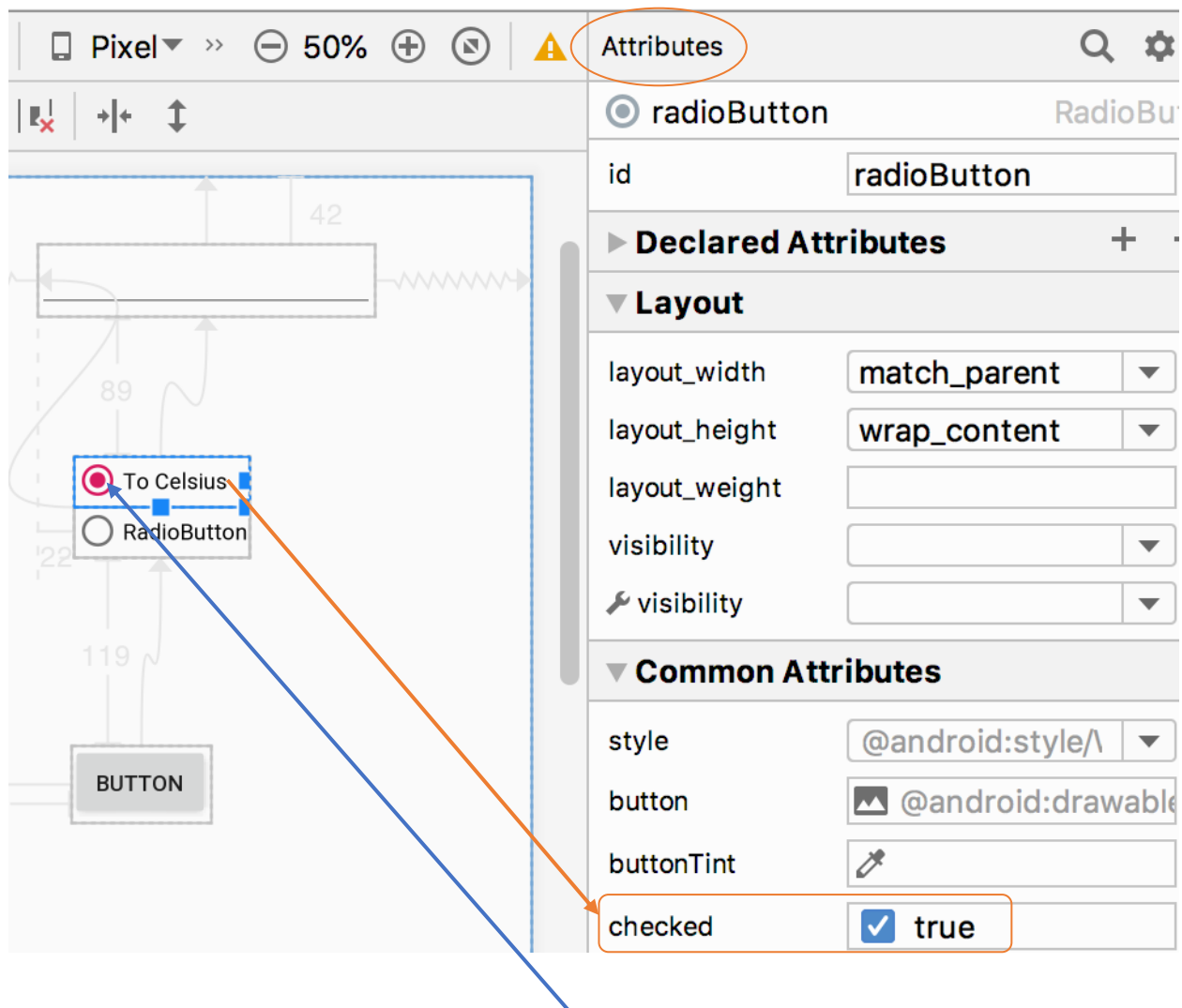
### More element tweaks!

Tweak each element listed below similarly to how you just tweaked your text values above for your radio buttons, as described next. Note, where you add in any attributes for a tag are arbitrary.

#### Radiobuttons

In Design view, select your first radio button and set the **checked** property of your first RadioButton in your group, this time in **Attributes** View under **Common Attributes** section, by actually clicking on the check box for the **checked** attribute, to ensure the first button for celsius will be the default RadioButton already checked on when your app starts.

Snapshot on next page illustrates setting action areas.



You will now notice in Design view your first Radio button (as shown above) is automatically checked or set as selected! Notice too that the attribute of "true" has been added to the checked property for your radio button.

### <RadioButton

```

::
    android:checked="true" />

```

### Button

Next assign `@string/calculator` to the `text` property of your **button** similarly what you did for your radio buttons plus add a tag (new attribute) which assigns the value `onClick` to the `onClick` property. This process eliminates the need for a listener to be added in code, so really you will just create a method called `onClick` in your `MainActivity` java file to handle any request by the user later in your Java code. So for now ignore the warning for a handler needed.

### EditText

Set your `InputType` property values to include both `numberSigned` and `numberDecimal` to allow for decimal input entry as well as the ability to enter negative values for temperatures. Use the pipe `|` on your keyboard by using your **Shift + \** key to set two values for the same attribute (sample tweak follows).

```

<EditText
    android:id="@+id/editText"
    ::
    android:inputType="numberDecimal|numberSigned"
    ::

```

Finally adjust the EditText to allow for the element to automatically *receive* focus at run time. Tweak the tag as follows. Make sure to include a separate close tag ending to allow for the `<requestFocus />` tag completion. Notice your EditText element has changed from a closed tag ending to just an open tag that just closes.

```

<EditText
    android:id="@+id/editText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="48dp"
    android:ems="10"
    android:inputType="numberDecimal|numberSigned"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/radioGroup2" >

    <requestFocus />

```

```

</EditText>

```

Make sure to close out beginning tag

### ConstraintLayout

All your user interface components are now contained in a layout. You can assign the background color to this Layout. For your opening ConstraintLayout tag, select the **background** attribute and add in `@color/myColor` to pickup your attribute value you previously defined in your **strings.xml** file. Voila! Your background now has changed immediately to a nice Mistyrose look n feel!

Your opening tag should now reflect the following line added in

```

<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    ::
    android:background="@color/myColor"
    ::

```

### Keynote

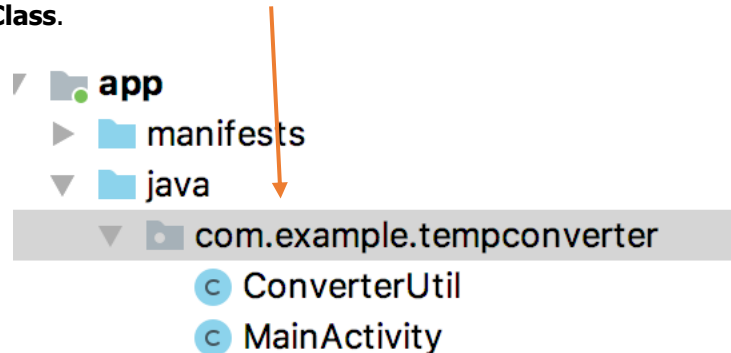


Note that sometimes XML element nodes or roots can end simply with a `/>` at the end of a closing attribute or a standalone ending with the element name. Ex., `</EditText>`. Either way is acceptable.

## STEP 6 Creating a utility class

Create the following utility class to enable the conversion of celsius to fahrenheit and vice versa.

To add in a class to your project, go to your project tree view, click open your app/java folder, then right click on your package (same package as your MainActivity.java class file) and select **New > Java Class**.



Name your class **ConverterUtil** within the Create New Class dialog box (shown next) and then click  to close out the dialog box.

Enter in the following code verbatim as you see below (note your package name may differ for all the code logic for this lab, which is okay!!!).

```
package com.example.tempconverter;
```

```
public class ConverterUtil {
```

```
    /**
     * @param fahrenheit
     * @return
     */
    // converts to celsius
    public static double convertFahrenheitToCelsius(float fahrenheit)
    {
        return ((fahrenheit - 32) * 5.0 / 9.0);
    }

    /**
     * @param celsius
     * @return
     */
    // converts to fahrenheit
    public static double convertCelsiusToFahrenheit(float celsius) {
        return (celsius * (9 / 5.0)) + 32;
    }
}
```

## STEP 7 Updating your MainActivity code

At startup of your project, Android Studio project wizard created the corresponding **MainActivity** class for your activity code. Adjust this class with the code below. Note, put in all the needed import statements below and code logic in your file. You'll notice that some code wraps to a new line due to the constraint set by margins.

```

package com.example.tempconverter;

import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.RadioButton;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    private EditText text;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        text = findViewById(R.id.editText);
    }

    /* this method is called when user clicks the button and is handled
    because we assigned the name to the "OnClick property" of the
    button */
    public void onClick(View view) {
        switch (view.getId()) {
            case R.id.button:
                RadioButton celsiusButton =
                    findViewById(R.id.radioButton);
                RadioButton fahrenheitButton =
                    findViewById(R.id.radioButton2);
                if (text.getText().length() == 0) {
                    Toast.makeText(this, "Please enter a valid number",
                        Toast.LENGTH_LONG).show();
                    return;
                }
                float inputValue =
                    Float.parseFloat(text.getText().toString());
                if (celsiusButton.isChecked()) {

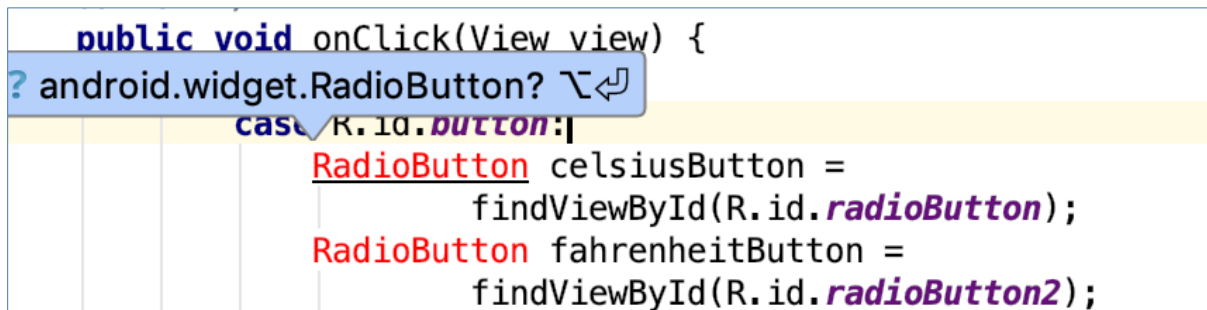
                    text.setText(String.valueOf(ConverterUtil.convertCelsiusToFahrenheit(inputValue)));
                    celsiusButton.setChecked(false);
                    fahrenheitButton.setChecked(true);
                }
                else {

                    text.setText(String.valueOf(ConverterUtil.convertFahrenheitToCelsius(inputValue)));
                    fahrenheitButton.setChecked(false);
                    celsiusButton.setChecked(true);
                }
                break;
            }
        }
    }
}

```

Note that whenever in the future you add in some code logic such as assignment statements especially involving some object creation or reference, etc., you probably will need some import statement(s) otherwise the system will flag you if not. A quick fix is to go to the end of your line that is flagged and press **Alt+Enter**.

Sample snapshot shows a pop-up error message for a given widget that has no import statement.



If you're used to shortcuts, please visit this [link](#) which show shortcuts operable in Studio.

## STEP 8 Running your application

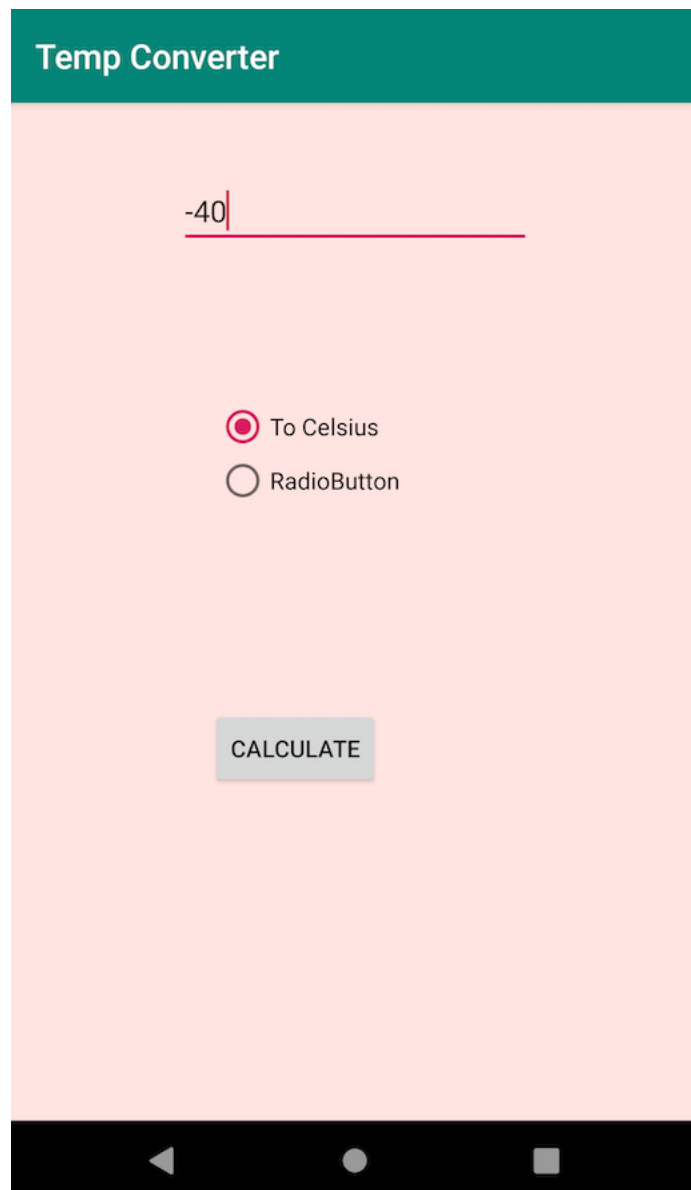
Whew! Time to run this pup. Assuming you have set up a Virtual Device in Studio, press **Run > Run 'MainActivity'** from your menu or merely right click on your project's MainActivity file and choose **Run 'MainActivity'**.

### Test your running app

Click once inside your text field (cursor should be showing as a prompt) and type in a temperature value, select your conversion and press the **Calculate** button. The result should be displayed back onto the text field and the other option button should get selected designating the resultant converted degree type. Try also pressing Calculate without any value in the text field, you should get a warning Toast pop up message!

Sample emulator at runtime snapshot follows.





## STEP 9      Modifying your Android application

Okay you have come this far! Congratulations! Time now to modify what you've done to include some extra nice features for your app. Modifications will include changing the background color of your app depending on the temperature. Perform the following tweaks.

In your layout view file, `activity_main`, give an id to your constraint layout's opening tag, called `activity_main`, as follows:

```
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    ::
    android:id="@+id/activity_main">
```

Save your file. Now you will be able to change colors of your layout background by referring to the id of the layout in code!

To change the background colors for various temperatures either being too hot or too cold in code, open up your MainActivity file and add in the following code enhancements.

Right towards the beginning of your class declaration, after the opening tag add, in the following line of code to create a view object

```
View view; //create object to manipulate background color
```

If you like, this could be placed directly after your declaration of your text object namely,

```
private EditText text;
```

Next in your **onClick** method, add in the following code logic after the last if/else statement but before the **break**; line, as follows

```
//grab CURRENT result value now in Text Field
inputValue = Float.parseFloat(text.getText().toString());
view = findViewById(R.id.activity_main);
if (inputValue>90){
//set hex color to skyblue
    view.setBackgroundColor(Color.parseColor("#87ceff"));
}
else
{
    view.setBackgroundColor(Color.YELLOW);
}
```

You'll notice in the code above, that a view object was created to point to the id of app's ConstraintLayout element, namely **activity\_main** to allow the altering of the app's background programmatically. Note the **setBackgroundColor** method which takes in a color of our choice. You can either choose the intellisense offerings for some standard colors (Yellow) as in our case or include for example some hex values (**#87ceff**), aka *skyblue*, as our alternative choice.

For more hex color choices visit- <http://cloford.com/resources/colours/500col.htm> .

Notice in the if conditional logic, it states to set a background color of Skyblue if the temp result in the Text field yields a temp greater than 90, otherwise it defaults to Yellow.

Finally make sure now to have the following import statement amongst your list of imports.

```
import android.graphics.Color;
```

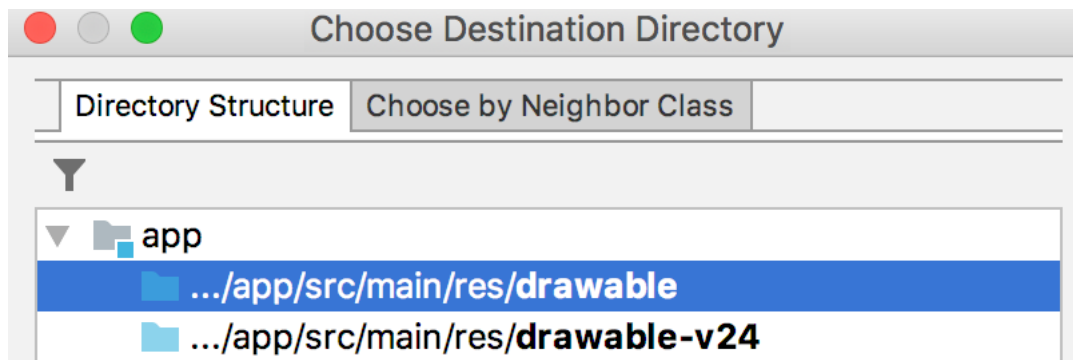
Modify your code further such that the background color turns red if the temp drops below zero (like our Chicago temps at times).

Run and test code to see if things work out with various temperatures. App lookin' pretty cool now huh?

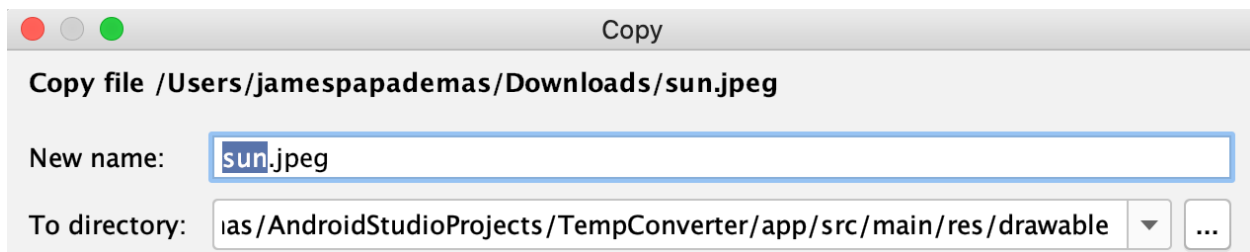
### Grad credit

Adding in images to your app. This last part will be similar this step 9 and work from other prior steps. Here you will add in two images to your app and then display them based on certain temp results.

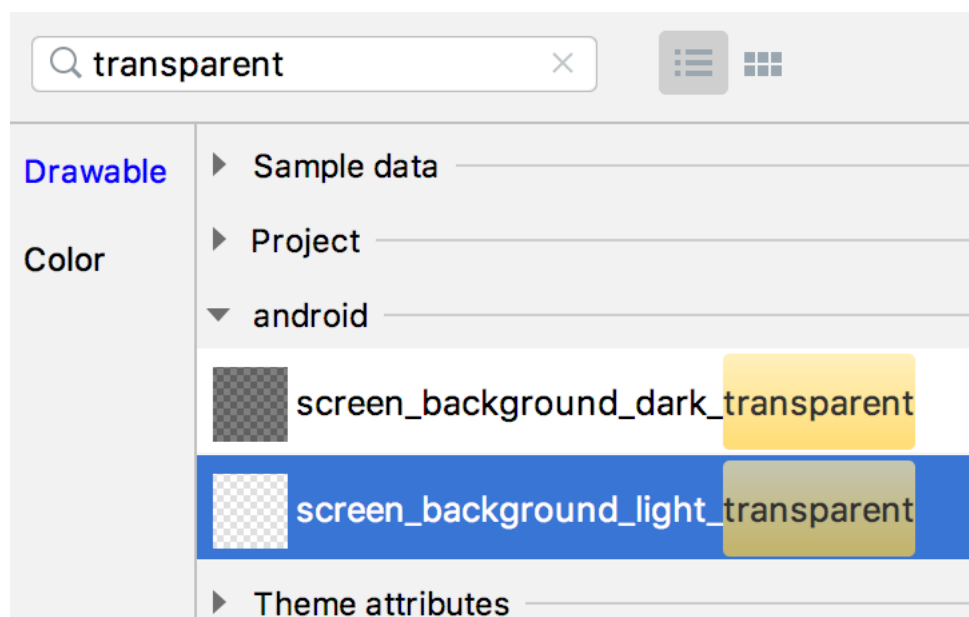
Grab 2 picture files of reasonable size from the web. One picture can represent warmth like a sun picture and the other to represent coldness like a frosty picture, etc. Copy and paste your images one at a time into your **res/drawable** folder. When prompted for a destination directory you can just choose the first option as shown below.



For the next pop up just make sure the name of the file and chosen directory to save the file to is correct. Example follows.



Now add in an **ImageView** from your Palette Widgets container choices, after your button in design view. You should now be prompted with a screen to add in a new resource such as an image to include for your object. For a default setting, type in the word **transparent** at the search area on top and choose under the android pull down choice, the **screen\_background\_light\_transparent** option and click **OK** as shown next.



Now open your MainActivity file, and in your code, declare right after your class starts, another object, this time an ImageView object as follows.

```
ImageView iv; //create iv object to manipulate image view
```

Next within your onClick method, after your first if/else statement, add in the assignment statement for **iv** object shown in **red** below.

```
//grab CURRENT result value now in Text Field
inputValue = Float.parseFloat(text.getText().toString());
view = findViewById(R.id.activity_main);
iv= findViewById(R.id.imageView);
```

Lastly “adjust” your last conditional logic block by adding in the **red** highlighted code shown below which renders a “sunny” image if temperature exceeds 90 degrees.

```
if (inputValue>90){
    //set hex color to skyblue
    view.setBackgroundColor(Color.parseColor("#87ceff"));
    iv.setVisibility(View.VISIBLE);
    //clear any prior image
    ((ImageView) iv.findViewById(R.id.imageView)).setImageResource(0);
    iv.setImageResource(R.drawable.sun); //show sun on image
}
else {
    view.setBackgroundColor(Color.YELLOW);
    iv.setVisibility(View.INVISIBLE);
    ((ImageView) iv.findViewById(R.id.imageView)).setImageResource(0);
}
}
```

Notice the added code will either set the visibility of the image to **VISIBLE** or **INVISIBLE** depending if you want the image to show at various temperatures. Also your image source can be set programmatically by the `setImageResource` method to point to any existing image in your drawable folder you specify (just the word sun in this case) and you can also set the clearing up of any resources used by the `ImageView` (any existing image in memory) by passing a **0** in the `setImageResource` parameter via the `iv` object.

Finally make sure to add in the following import statement if necessary.

```
import android.widget.ImageView;
```

Run your app and test it now thoroughly for temps above and below 90. If your image does not display, check your layout configuration settings and/or the size of your image if some settings are causing rendering issues. *You can always adjust your Attribute settings for your component if the image is not appearing in a desired location, by tweaking settings such as height/width setting them to a desired **dp**, as well as checking any other settings such as the `scaleType` property, or adjust positioning (ex. to center) or even a `layout_alignParentLeft` setting, etc. Check any IntelliSense choices for more desired settings in Text view!*

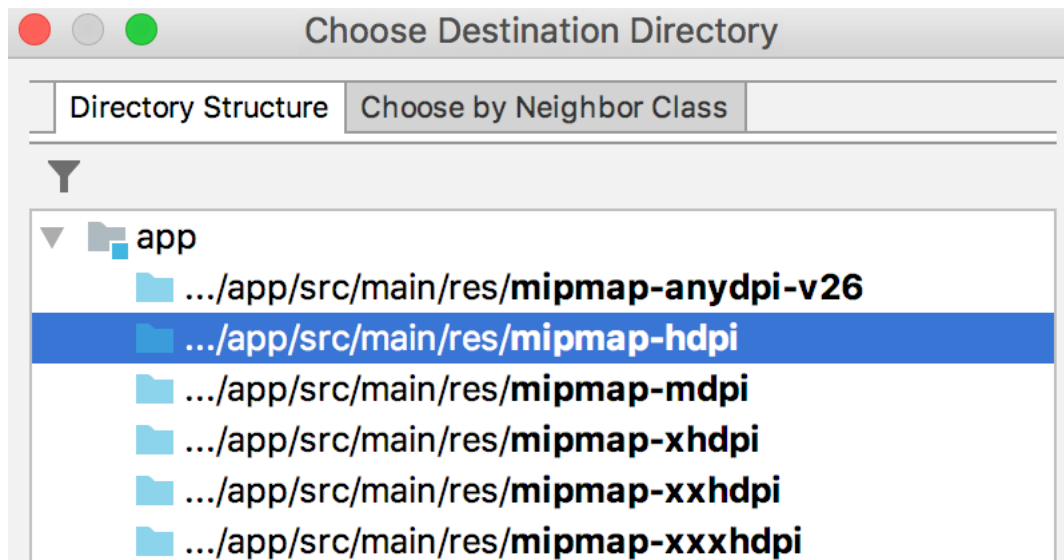
More on dpi's/unit measures- <https://material.io/design/layout/understanding-layout.html#usage>  
For more on constraint settings- <https://medium.com/exploring-android/exploring-the-new-android-constraintlayout-eed37fe8d8f1>

Modify your code to include an image of your choice when the temperature drops below zero. Note if there are any images between 0 and 90 showing as a result, something in your logic needs tweaking! Test it out.

### Extra credit (all)

Adding in a **Home Screen** desktop icon for your app.

Grab a suitable icon (.png or .jpg file formats and small to medium file sizes will do just fine) from your favorite search engine. Download an icon that relates perhaps to weather or temperatures and copy in the icon into your **res/mipmap** folder under the **root** of your project. Name your file to whatever you like. Note to copy files into your folder you can right click on your folder and choose paste. When prompted for a Destination directory, choose the following.




Click **OK** to continue. Click **OK** to finish at the next pop up message.

Next go to your **manifests/AndroidManifest.xml** file and go to the line under the application root and set the **android:icon** property value to include just your file name you have given your icon.

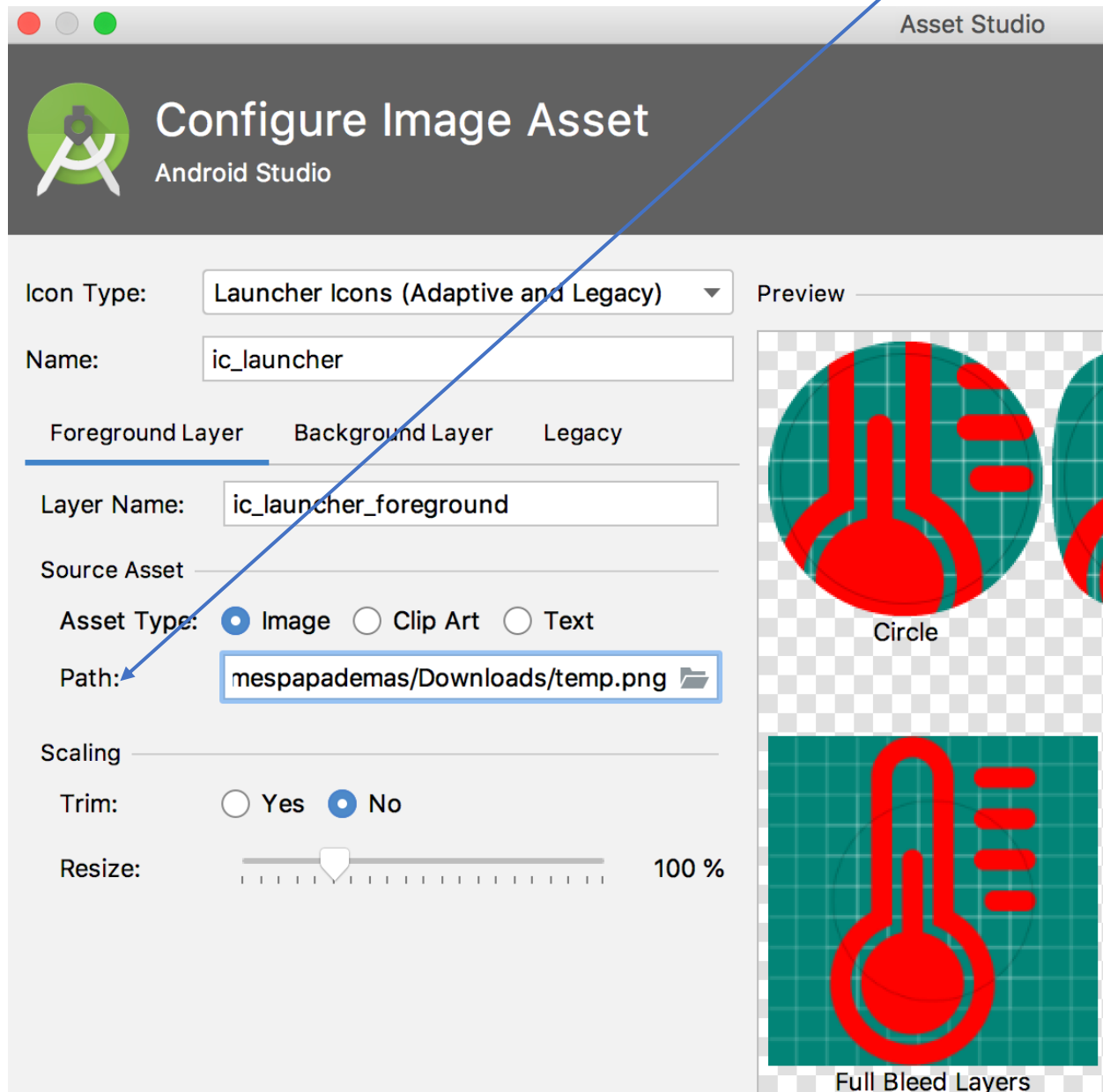
Ex. I have a file icon called **temp** with an extension of .png.

I would include the following tweak to the **AndroidManifest** file...

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/temp"
    ::
```

Basically you are now telling Android to start up not with the default desktop Android icon  but your own!

Next right click on your **mipmap** folder and choose New > Image Asset and where it says Path: point to your current image file which is probably in your downloads folder. Click **Open** then **Next**, then **Finish**.



You can try and run your app to see if this kicks in for you. Check your actual desktop on your phone to see the resulting new icon. To view your desktop apps/icons, just swipe upwards towards the bottom of your visible emulator screen.

## STEP 10 Submitting Your Program Code and Your Run Time Output

For full credit make sure to include any source code and edited xml files into a pdf file as well as snapshots in a separate pdf for a completed submission. Make sure your code is complete with comment statements where necessary and include a brief program description at the top area of your MainActivity.java file.



### **For full credit!**

**Take the following snapshots of your app in action. LABEL your snapshots accordingly. Save your document file as pdf.**

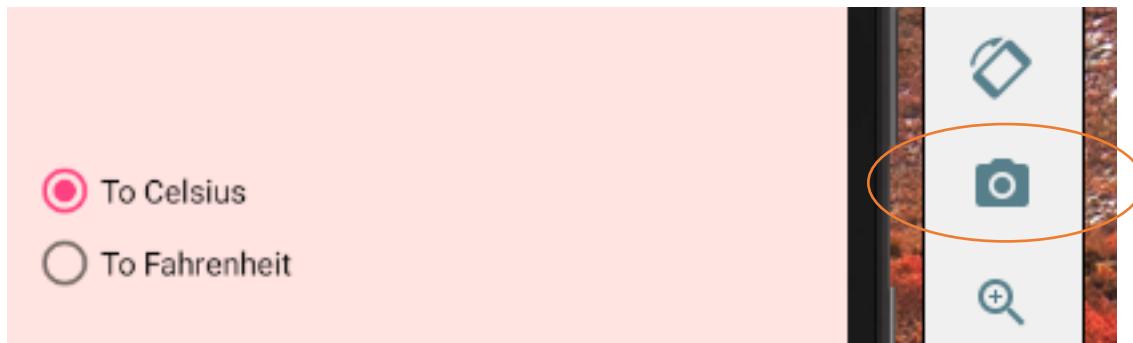
1. Take a snapshot of your running application, which will show the result of 40 degrees Celsius converted to Fahrenheit.
2. Take snapshots showing each background color that triggers for a very warm temp and a very cold temp. Include snapshots for red, skyblue and yellow colors.

Grads. Make sure to show images for temps greater than 90 or below 0 and a temperature in between like 30 degrees Fahrenheit.

3. Extra Credit.
  1. Show snapshot of your Home screen showing your custom icon display.
  2. Show snapshot of your activity screens in landscape view as well for each of the above requested snapshots. Sample follows...

To easily create snapshots, follow these basic suggestions if you will.

1. For Mac/Windows, just press the camera icon on the extended controls of your emulator to have a snapshot saved to your Desktop. Preview your snapshot and get it copied your Word Document file. Snapshot follows showing camera icon.



2. Use the Windows Snipping Tool built in the Accessories group in Windows 7, 8 & 10 or Mac people can use Skitch, which is a free download.

**Zip up your project files into Blackboard for credit. Also include a pdf of your snapshots.**

Include for your pdf file, your **name, lab number, course number/section** and the **date** at the top of your Word doc. Please do this for all labs. Name your Word doc file to include your first initial followed by an underscore, then your first 4 letters of your last name followed by another underscore and the lab number (ex. **s\_polk\_lab1**).

Proper documentation including descriptions and comments in your java source files and any error traps will always be thoroughly checked as being part of your grade so make sure to include these items where applicable!



### General App Troubleshooting Tips

1. Clean your project to clear errors by going to Build > Clean Project then if necessary run your application to see if everything also clears!
2. Do a **Save All** perhaps to update changes in your file(s).
3. Shutdown/restart Studio – if don't see an error go away after a clean!
4. Shutdown your running AVD and restart app from scratch.
5. Reset your AVD or create a new one (to handle differing Api's)  
Experiment with different RAM settings, VM bytes for added acceleration/performance.
6. Check any code that maybe in question along the way, especially when your Build your project (do **Build > APKs** to rebuild your APK file). With a lot of code and dark backgrounds fonts in your editor it may be hard sometimes to detect. Check for valid import statements!!!  
Use the Messages view in Studio if it's not visible, **View > Tool Windows > Event Log**, to help pinpoint gradle builds.
7. Check your XML properties & attributes!
8. Check log files/Gradle Console/Problem Windows, etc.
9. Desktop maintenance – deleting old/similar apps by project name.
10. Use various Studio window views when your app starts up or is running to detect your background processes, app build intel, logs (via Logcat view) etc.  
Always have these open at runtime to monitor / debug things!

What are your tips/errors? Please **share** at the Discussion forum in Blackboard!