ITMD 455/555 *Intelligent Device Applications* Lab 1

#### 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.

A screenshot of a cell phone

Description automatically generated

**STEP 1 Creating a New Android Project**

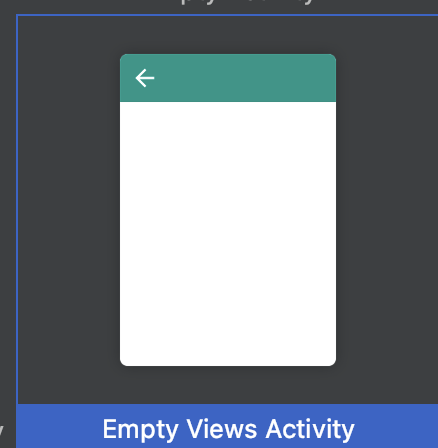
Startup Android Studio.

To start an app, click on the option ‘**New Project’** or just go to your menu and choose **File >** **New > New Project**.

Graphical user interface, application

Description automatically generated

Here a New Project dialog box appears where you can configure your new app. For your new app creation, highlight **Empty Views Activity** as shown below and press Next.



Enter in the Application name as shown below, a project location of your choice, use **Java** as your Language plus a **Minimum API level of 28**. For your Build configuration language, please choose

**Kotlin DSL (build.gradle.kts) [Recommended]**. Leave any other options as default. Click Finish when complete.



 (*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 files 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.

A screenshot of a cell phone

Description automatically generated



Notice from the lastes stable build, Android has added some bloated code in onCreate, which I elimated as shown in the code window view above.

**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 **#FFE41.** 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.

A screenshot of a cell phone

Description automatically generated

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...

A screenshot of a cell phone

Description automatically generated

…and enter EXACTLY the following into your pop up.

A screenshot of a cell phone

Description automatically generated

Click on OK 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.

A screenshot of a cell phone

Description automatically generated

Repeat the same action working your Translations Editor 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.

A close up of text on a blue background

Description automatically generated

Once your 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 all 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 the **Design** view or strictly be in the XML **Code** or even the **Split** view as shown below. You can switch between both representations via the tabs at the top right of the editor.

A screenshot of a computer

Description automatically generated

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 drag on any of the Palette items onto the activity screen to view its properties and its starter contents when in **Code** mode.

*A screenshot of text

Description automatically generated*

**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.

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 respectfully, of your Design interface.

­­

Graphical user interface, application

Description automatically generated

From your Design view, WITHIN your Component Tree, click on the existing TextView component and press your Delete key to remove the textview object/associated XML code.

A screenshot of a computer

Description automatically generated

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. Move the widget towards the center and down somewhat from the top of the screen as shown next.

Graphical user interface

Description automatically generated with low confidence

If you now view your XML source code, you will notice that the Text Field element has been assigned a new **android:id** attribute by default as follows: **android:id="@+id/editTextNumberDecimal"**

**Keynote**

A picture containing clipart

Description automatically generated All entries in the Text Fields section of your XML file define text fields. Various entries   
 define “added” attributes for them, example. If a text field should allow input only for   
 numbers, or allow for negative values as well, etc. Check it:

<EditText  
 android:id="@+id/editTextNumberDecimal"  
 ::

android:inputType="numberDecimal"  
 tools:layout\_editor\_absoluteX="48dp" />

Now as the text element has been added, think of this element as the leader where placements of other elements will be relative too! This will make working the layout way easy. Let’s concentrate now getting constraints then set up initially for our text view.

Notice from the snapshot below, there are no Constraints set like for horizontal / vertical constraints, you will get the red exclamation error icon! Click on the error icon as shown below and if necessary, click on the **Attributes** view on the right side to see the error messages as shown below.

A screenshot of a computer program

Description automatically generated

No worries here on the errors, we’ll get that fixed next to ensure a smooth layout process.

In **Design** view, click on your text element to grab focus of the element, and in your **Attributes** area locate the **Layout > Constraint Widget** section.

A screenshot of a cell phone

Description automatically generated

Focus in on the constraint settings area depicted below and notice you can click on the  symbol to auto set constraints for top, bottom, left and right positionings.

A picture containing clock

Description automatically generated

Merely click the top and left symbols as shown below to kick in gear, some much needed constraints. Yours of course will be different no doubt (mine shows 100 dps for the left margining setting and 89 for the top alignment to the parent). This is perfectly fine as now you can freely move the element around and the contraints are automatically adjusted and the constraint errors are gone as well!

A picture containing clock

Description automatically generated

Notice now that your contraints are now relative to the parent (activity screen)! Sweet!

A close up of a logo

Description automatically generated

Note if you still see other possible warnings/errors, feel free to adjust your attribute settings. Sometimes you may need to raise the height of the element as an exception, so make adjustments so as to remove a particular error. Example that follows, shows you can even choose to ignore errors at times as well.

Example of seeing viewable messages in the **Layout and Qualifiers** tab by clicking the error icon follows. Note even though there are layout errors, you can still run your app as a workable one.

A screenshot of a computer

Description automatically generated

Ok now onto getting the rest of your elements onboard so it looks like the activity screen on page 1. Go to **Buttons** in your Palette and select the **RadioGroup** element and drag in under your text field aligned leftward as shown next.

A screenshot of a cell phone

Description automatically generated

Alrighty then. We see a constraint error. We’ll get that fixed in a sec. as well. Drag two **Radio** **Buttons** into your **Component Tree** area beneath your Radio Group. You want RadioButtons to be part of your RadioGroup making coding easier to ascertain what a user has selected. You’ll notice Warning icons which are good to always note, but for now it won’t affect your work.

A screenshot of a cell phone

Description automatically generated

Warning icons

Next click on your **RadioGroup** element in your tree to grab focus.

A screenshot of a cell phone

Description automatically generated

Then perform the following settings to set positioning as seen in the animated gif that follows!

A screenshot of a cell phone

Description automatically generated

As you can see from the above animation, we merely “hook up” the needed contraints of the RadioGroup itself to parallel some alignment features relative to its parent, namely the text view. Note there is simplicity now to move the text field (the leader) to desired areas as a group (slowly and carefully) around the screen to any desired areas. Your ending constraints and positioning should look something like the snapshot that follows.

A screenshot of a cell phone

Description automatically generated

Ok almost finished with this layout stuff. Again, from your Design view, select the **Buttons** section in the Palette and drag a Button underneath your Radio Buttons group and align it vertically/horizontally to your Radio Group similarly to how you adjusted contraints for your Radio Group in the step above.

Your result Tree and Layout should resemble something similarly to the below snapshot.

A screenshot of a cell phone

Description automatically generated

Notice that the id’s are given (as shown above) automatically for elements under a Constraint style layout. Example ids are **radioButton** and **radioButton2**. Ids are always important so you can reach out or refer to the elements in code later on. Id’s can always quickly be changed in code if need be. Notice also default text is given to the radio buttons. That will be adjusted soon in your Attributes view.

Now if you run your app, you should see output that is perfectly according to your design!

Graphical user interface, application

Description automatically generated

Pause a moment 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!

**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 shown highlighted and in bold below, to get used to things.

Switch to the XML file Code view and **override** the **android:text** attribute **value** of your 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 hence 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!!

https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcTWTRuocSpcmMJvvPH3Lwcjo7MBM3MGtoiTfmhoRPj_DMCa_F_O9gMore element tweaks!

Tweak each element that follows similarly to how you just tweaked your text values for your radio buttons, as described next. Note where you add in attributes for a tag (i.e. the ordering) are arbitrary.

Radiobutton

In Design view, select your first radio button and set the **checked** property of your first RadioButton in your group, this time from your **Attributes** view under **Common Attributes** section, by actually clicking the check box for the **checked** attribute, to ensure the default radio button at run time will be the **first** button (for celsius) checked by default.

A screenshot of a cell phone

Description automatically generated

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/calc** 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. Added attributes/values follow.

<**Button**

**::  
 android:text="@string/calc"  
 android:onClick="onClick"**

**::**

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 your EditText tag 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 must change from a closed tag ending to just an open tag that just closes.

**<EditText  
 android:id="@+id/editText"  
 ::  
 android:inputType="numberDecimal|numberSigned"  
 app:layout\_constraintStart\_toStartOf="parent"  
 app:layout\_constraintTop\_toTopOf="parent"  
 tools:ignore="SpeakableTextPresentCheck"** >  
  
 <**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 pick up your attribute value you previously defined in your **strings.xml** file. Voila! Your background now has changed immediately to a nice Misty rose 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

A picture containing clipart

Description automatically generated 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) as shown next and select

**New > Java Class**.

A screenshot of a cell phone

Description automatically generated

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

A screenshot of a computer

Description automatically generated

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 that follows. 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 MS Word’s 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.***editTextNumberDecimal***);  
 }  
  
 */\* 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) {  
 **if** (view.getId() == 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**);

}

}

}

} *// end MainActivity*

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 hover over the flagged item and press **Alt+Enter**.

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

A screenshot of a computer

Description automatically generated

If you’re used to shortcuts, please visit this [link](https://developer.android.com/studio/intro/keyboard-shortcuts) 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, various ways to run and test your app is to either press **Run > Run** **‘MainActivity’ fr**om your menu, right clicking on your project’s MainActivity file and choose **Run ‘MainActivity’** or just pressing the Run icon  at the top which is your toolbar.

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!

Runtime snapshot of sample emulator follows.

Graphical user interface, text, application

Description automatically generated

**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 or modify an existing 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 “programmatically” 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 place 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 lines, 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 choice. You can either choose the intellisense offerings for some standard colors (Yellow) as in this 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 to **STEP 9** and 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 (pngs or jpgs) 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 or Drag and Drop to copy or move 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 next.

Copying to drawable directory.

A close up of a sign

Description automatically generated

At the pop up just make sure the name of the file and chosen directory to save the file to is correct.

Click OK to accept the choice destination. Example follows.

A screenshot of a computer

Description automatically generated

Now add in an **ImageView** from your Palette Widgets container choices (under Widgets), after your button in design view. You should now be immediately 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.

Graphical user interface, application

Description automatically generated

Choose the **screen\_background\_light\_transparent** option when it appears as shown next.

A screenshot of a cell phone

Description automatically generated

Click OK to continue. Set any constraints, even height and width settings you deem necessary for the ImageView control.

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*

Make sure to add in the following import statement if necessary.

**import** android.widget.ImageView;

Next within your onClick method, after your first if/else statements, add in the assignment statement for your 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. Also add in the code in the else part highlighted in red to reset the imageView back to an Invisible state.

**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. Note your image source can be set programmatically by the setImageResource method to point to any existing image in your drawable folder you specify (ex. code above shows **sun**, which is the choice file name). Further note you can also set the clearing up of any resources used by the ImageView (any existing image in memory) by passing a zero ( **0)** in the setImageResource parameter via the iv object.

Run your app and test it now thoroughly for temps above and below 90. If your image does not display properly, check your layout configuration constraint settings relative to your other elements and/or the size of your image if some settings are causing rendering issues. *For example, height and width settings can easily be adjusted by tweaking parameter settings such as height/width, setting them to a desired* ***dp(density pixels)****, as well as checking any other settings such as*,the android:scaleType *property for centering, etc.*

More on **dp**’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 like some frosty image. 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 (ex.,**.png** or**.jpg)** file format (small to medium file sizes will do just fine) from your favorite search engine. Download an icon that relates perhaps to weather or temperatures. Name your file to whatever you like.

To redefine the app icon go to your **res/mipmap** folder which points to a ‘starter’ icon to launch located under the root of your project and right click on your 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. This is needed to override the default Android desktop icon.

For good knowledge, go to your **manifests/AndroidManifest.xml** file and go to the line under the application to note the icon property for mipmap.

Snapshot follows…

Graphical user interface, application

Description automatically generated

You can try and run your app to see if this kicks in for you. Check your actual desktop on your emulator to see the resulting new icon. To view your desktop apps/icons, just click the Desktop or 'Home' view button on your emulator on the extended controls area



**STEP 10 Submitting your Program Code and your Run Time Output**



**For full credit!**

For full credit make sure to include any java source code and edited xml files into a pdf file as well as snapshots in a separate pdf for a completed submission. Also submit a separate zip file of your project. 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.

**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, i.e., temps greater than 90 or below 0. Thus, include snapshot results for skyblue and red

colored background appearances.

Grads. Make sure to show images for a temperature in temps greater than 90 and one for temps

below 0. Also snapshot a result of 86 degrees Fahrenheit to Celsius.

1. Extra Credit (choose either number 1 or 2 below as a choice). Label snapshot as Extra Credit.

1. Show snapshot of your Home screen showing your custom icon display. (+5 points)

2. Show snapshot of your activity screen in landscape view to be shown in an organized “neat” UI as well for each of the above requested snapshots. (+5 points)

Landscape sample follows…

A screenshot of a cell phone

Description automatically generated

To easily create snapshots, see the basic suggestions that follow.

For Mac/Windows, just press the camera icon on the extended controls of your emulator

to have a snapshot saved to your Desktop or to a desired location. Preview your snapshot and get it copied your Word Document file. Snapshot follows showing camera icon location.

A screenshot of a cell phone

Description automatically generated

Alternatively you can use the Windows Snipping Tool built in the Accessories group in Windows 7, 8,   
 10, 11 or for the Mac, users can use [Skitch](https://www.macupdate.com/app/mac/27134/skitch), a free download or alternatively do a **Command A picture containing text, brass knucks, weapon, window

Description automatically generated** plus  
 **Shift 4** to copy areas.

Include for your pdf files, your **name**, **lab number**, **course number/section** and the **date** at the

top of your document. Please do this for all labs. Name your document 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, file descriptions   
 and file management 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 & Smart 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 you’re 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 you Build

your project (from your menu choose Build > Build Bundle(s) / APK(s) > Build APK(s) to

rebuild your APK (aka Android Package Kit) file). With a lot of code and dark background fonts in   
 your editor, errors may be hard sometimes to detect. Also check for valid import statements!!!

7. Check your XML properties & attributes!

8. Check Logcat file, Gradle files (i.e., module & Project files), Problems window views, 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 Builds etc. Always have these views open at runtime to

monitor / debug things!