**Te121`**

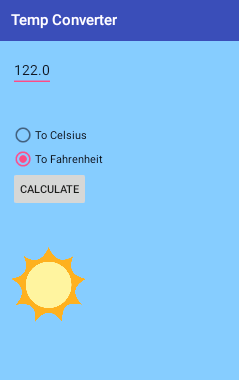
ITMD 455/555 *Intelligent Device Applications* Lab 1

#### 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, radio buttons and an imageview.

Interface of the app at runtime shown below is what you will be similarly building for this lab.

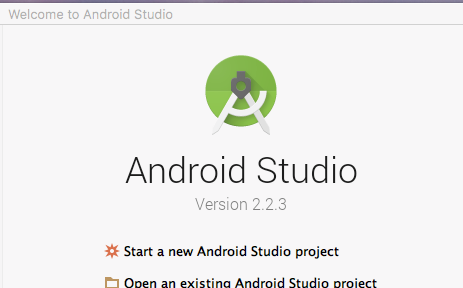


**STEP 1 Creating a New Android Project**

Start up Android Studio.

Use the Create New Project wizard to create your project by doing the following:

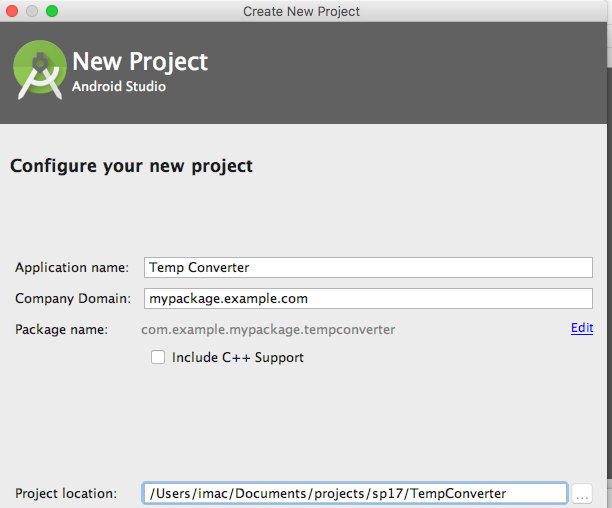
1. Clcik on the option Start a new Android Studio project seen on the Welcome screen.



Here a New Project dialog box appears where you can configure your new app.

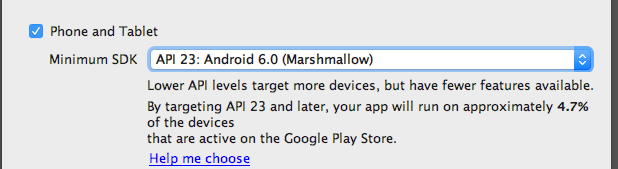
Enter in the Application name as shown below, and a project location of your choice then

chose Next.



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

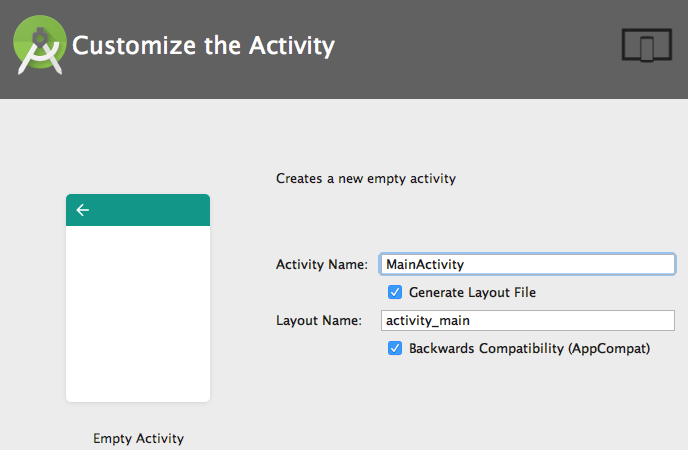
1. In the Target android Devices screen that appears next, keep all defaults and once again press Next.



1. For the next screen namely Add an activity to Mobile, keep or select the default

**Empty Activity** choice and press Next.

1. For your last screen, Customize the Activity, keep the default settings (as shown below) and press Finish when complete.

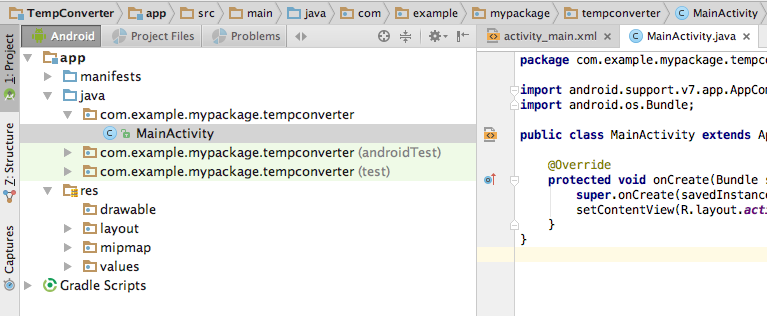


(*Note*- the name of the activity class is given as MainActivity with a correspnding layout

name of activty\_main)

Your project now creates itself.

Take a moment to familiarize yourself with your IDE, its layouts, menus, values that are part of your project tree you see below. Notice the nice tab structure (known as the Navigation Bar) tool for quick navigation to your files.



**STEP 2 Creating project attributes**

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.

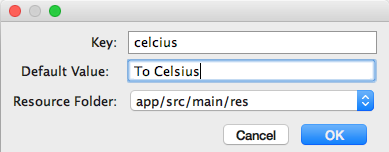
As a start, add in a color tag by adding the following line to your strings.xml file within your <resources> root tag

<**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 attribute 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 add in a key/value pair denoting programmer defined values. At this point you can 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 OK to commit your information. Your resource file now has been updated to include your new string data.

Do the same action and add in the following two additional atrributes that follow

**String Attributes add-ons**

|  |  |
| --- | --- |
| **Name** | **Value** |
| fahrenheit | To Fahrenheit |
| calc | Calculate |

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.

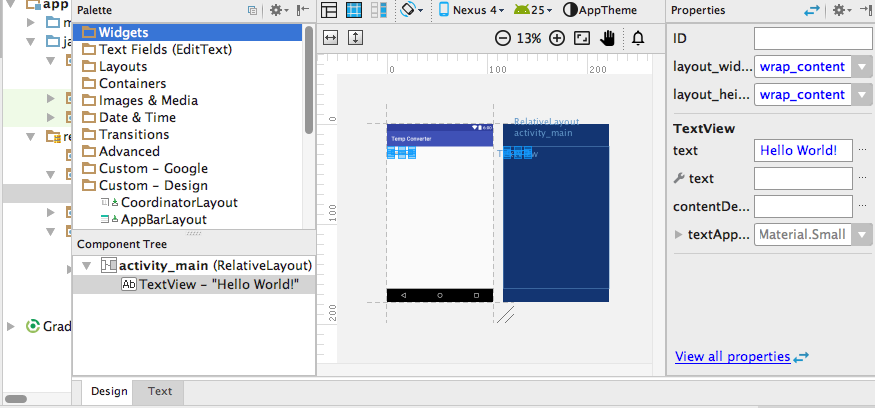
<**resources**>  
 <**string name="app\_name"**>Temp Converter</**string**>  
 <**color name="myColor"**>#FFE4E1</**color**>  
 <**string name="celsius"**>To Celsius</**string**>  
 <**string name="fahrenheit"**>To Fahrenheit</**string**>  
 <**string name="calc"**>Calculate</**string**>  
</**resources**>

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

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

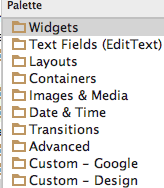
Select (double-click) the ***res/layout/activity\_main.xml*** file. The associated Android editor allows you to create the physical layout via drag and drop in **Design** view and/or via the XML source code inside your **Text** view! You can switch between both representations via the tabs at the bottom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

View Choices



‘

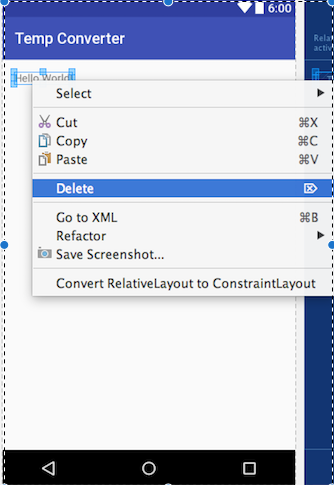
The following shows a snapshot of the *Palette* side of this editor. This feature allows you to drag and drop new **View** elements to your layout. Simply double click on any of the folders to view it contents.



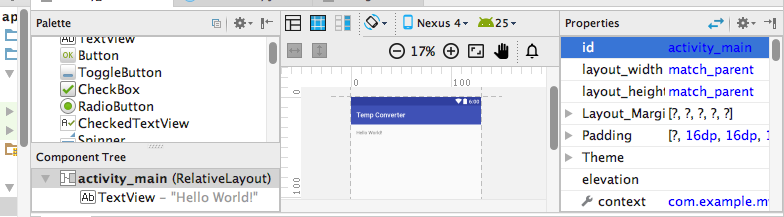
**STEP 4 Adding View elements to your layout file**

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.

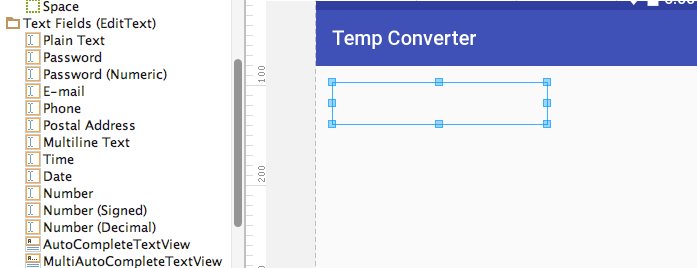


Get familiar with your surroundings concerning your Layout views! For example, check your Component Tree and Properties sections to the left and right of your Design interface.



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 Properties 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 click onto the section header to see all the text field choices . Locate the **Number (Decimal)** choice and drag it onto your layout (interface) to create a *editable* text input field.



Drag it over

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

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

**Keynote**

All entries in the Text Fields section of your XML file you’ll see, define

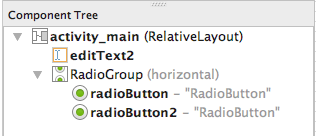
text fields. Various entries define *“added*” attributes for them, ex. if

a text field should allow input only for numbers, or allow for negative values as

well, etc..

Next select the *Containers* section in the *Palette* and drag a ***RadioGroup*** entry into your layout and place it directly under your text view. Notice in your Component Tree area, how your 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 properties in Properties view, etc..

Next grab two **RadioButton** objectsfrom your *Widgets* 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 buttons. That will be adjusted soon in Properties.

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



Make sure to include adequate spacing between your widgets to ensure your controls don’t bunch up in your layout. The resulting XML layout should look something like the following.

*<?***xml version="1.0" encoding="utf-8"***?>*<**RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:id="@+id/activity\_main"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:paddingBottom="@dimen/activity\_vertical\_margin"  
 android:paddingLeft="@dimen/activity\_horizontal\_margin"  
 android:paddingRight="@dimen/activity\_horizontal\_margin"  
 android:paddingTop="@dimen/activity\_vertical\_margin"  
 tools:context="com.example.mypackage.tempconverter.MainActivity"**>  
 <**EditText  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:inputType="numberDecimal"  
 android:ems="10"  
 android:layout\_alignParentTop="true"  
 android:layout\_alignParentStart="true"  
 android:layout\_marginTop="34dp"  
 android:id="@+id/editText"** />  
 <**Button  
 android:text="Button"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:id="@+id/button"  
 android:layout\_centerVertical="true"** />  
 <**RadioGroup  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_above="@+id/button"  
 android:layout\_alignParentStart="true"  
 android:layout\_marginBottom="30dp"**>  
 <**RadioButton  
 android:text="RadioButton"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:id="@+id/radioButton"  
 android:layout\_weight="1"** />  
 <**RadioButton  
 android:text="RadioButton"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:id="@+id/radioButton2"  
 android:layout\_weight="1"** />  
 </**RadioGroup**>  
 </**RelativeLayout**>

**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!**

**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 view and override the *android:text* property value of the first radio button and assign the **@string/celsius** value to the property. 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 follows...

<**RadioButton  
 android:text="@string/celsius"**

**::  
 android:id="@+id/radioButton"** />  
  
<**RadioButton  
 android:text="@string/fahrenheit"  
 ::**

**android:id="@+id/radioButton2"** />

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

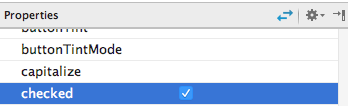
https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcTWTRuocSpcmMJvvPH3Lwcjo7MBM3MGtoiTfmhoRPj_DMCa_F_O9g

More element tweaks!

Tweak each element listed below similarl to how you just tweaked your text values above for your radiobuttons, as described next.

Radiobuttons

Select your first radiobutton and set the *Checked* property of your first RadioButton in your group this time in Properties View, 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.



You will now notice in design view your first radiobutton is automatically checked or set as selected! Notice too that the attribute of “true” has been added to the checked property for your radiobutton.

<**RadioButton  
 ::  
 android:checked="true"** />

Button

Next assign ***@string/calc*** to the text property of your **button** and 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 to handle any request by the user later in your Java code.

EditText

Set your *InputType* property values to included 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 code follows below).

Finally add the line below to allow for automatic focus to the EditText element on start up.

**android:focusable="true"**

Your EditText XML part should now resemble something like this

<EditText  
 ::  
 android:layout\_alignParentStart="true"  
 **android:inputType="numberDecimal|numberSigned"  
 android:focusable="true"** />

RelativeLayout

All your user interface components are now contained in a layout. You can assign the background color to this Layout. For your RelativeLayout 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

<**RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 ::**

**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.

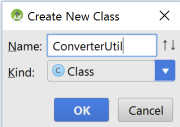
**STEP 6 Creating a utility class**

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, and right click on your app/java folder, then within your package (same package as your MainActivity.java class file) select New > Java Class.



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



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.mypackage.tempconverter;  
  
*/\*\*  
 \* Created by minimac on 1/9/2017.  
 \*/***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**

The Android Studio project wizard created the corresponding **MainActivity** class for your activity code. Adjust this class with the code below. Note, put in all import statements below *after you add in the needed code,* to ensure you won’t get any warnings (shown in red) from statements you will be adding in. You’ll notice that some code wraps to a new line for ease of readability here.

**import** android.os.Bundle;  
**import** android.support.v7.app.AppCompatActivity;  
**import** android.view.View;  
**import** android.widget.EditText;  
**import** android.widget.RadioButton;  
**import** android.widget.Toast;  
  
  
**public class** MainActivity **extends** AppCompatActivity {  
  
 **private** EditText **text**;  
  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
 **text** = (EditText) 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 = (RadioButton)  
 findViewById(R.id.***radioButton***);  
 RadioButton fahrenheitButton = (RadioButton)  
 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**;  
 }  
 }  
}

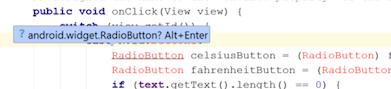
Note that whenever in the future you add in algorithms / code logic, assignment statements especially involving some control, etc., you probably will need some import statement(s) asscociated with your actions. The system will flag you will an error with a message stating

‘Can’t resolve symbol ….’ or ‘Can’t resolve method ….’, etc. if there is a missing import.

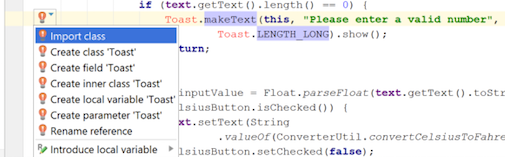
To add in the necessary imports you can of course code the needed import at the top or hover over any gray areas, underlined warnings, etc. where you will probably eventually see a message stating to press **Alt+Enter** for the fix. Or if you perhaps see a pop up icon  shown towards the left side line of the code error suggestiing fixes such as **Import class** which immediately will add your import in for you!

Note you can always also right click on your **app** folder or your **MainActivity** file and choose **Optimize Imports...** or from your menu choose **Code > Optimize Imports.** Be forewarned when using such an option though as sometimes imports can be seen as not to be inclusive as a suggestion and yet you may deem that a particular import(s) is perhaps necessary to keep.

Sample snapshots follow showing these pop up tips & icons offering help by Studio:







If you’re used to Eclipse style shortcuts, please visit the link below which show that some of the Eclipse shortcuts may be also operable in Studio.

<http://stackoverflow.com/questions/27657025/android-studio-shortcuts-like-eclipse?lq=1>

**STEP 8 Running your application**

Whew! Time to run this pup.

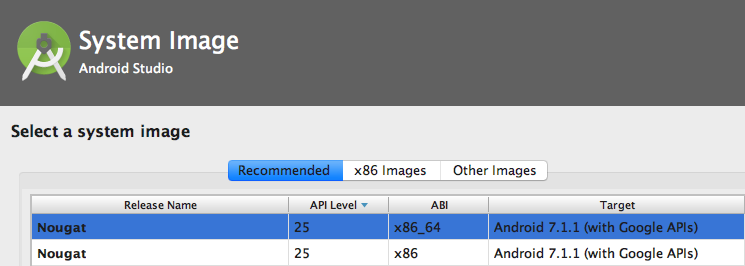
Since this your time time running an app in Studio you will want to configure a device image emulator for viewing your app at runtime.

Before even setting the emalator it is good to make sure your SDK is up to date with certain option settings and features so your emulator can run smoothly without any errors at runtime, etc.

If this is the first time you are running an app in Studio an emulator is needed to be set up. A few ways this can be accomplished. To set up your emulator merely hit the run  icon on your menu toolbar (or go to Tools>Android>AVD Manager). Assuming you go to run your app, click on the run icon and at the Select Deployment Target  at the bottom left of your Virtual Device Config screen to set up a sample Android Device emulation! Choose the following settings when setting up your droid emulation:



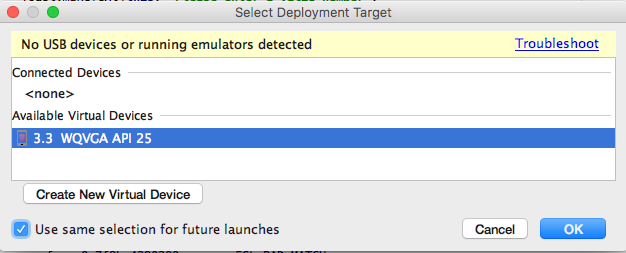
Choose Next and choose the following setting which will be the system image.



Press Next once again. At this verify your configuration choose an appropriate name or keep the default AVD name if you so desire. I usually choose something like Android AVD to know this is an Android image and not a Google one for API purposes! Keep all the other default settings at this point. Press Finish when complete. Then click on OK on the Select Deployment Target screen. You should now see your AVD Emulator loading and your app finally running!

Running you app!

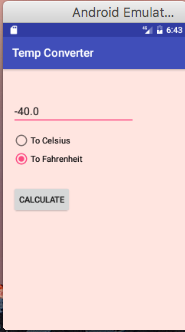
If at any time you wish to run your app (if it isn’t running initialy) merely right click on your project MainActivity file and choose Run ‘MainActivity’ or select the Run icon on the toolbar. Accept the default AVD to Launch your emulator that you configured previously as shown below.



Press OK to accept your choose to let your emulator run.

Test your running app

Click once inside your text field (cursor should be showing as a pompt) 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. Try also pressing Calculate without any value in the text field. You should get a warning Toast pop up message!





**For credit!**

Take a snapshot of your running application, which will show the result of 30

degrees Celsius converted to Fahrenheit. A few ways to take snapshots in Windows

1. You can snapshot your app easily by pressing **Alt + PrtSc** keys in Windows and pasting your

result into Word or

2. Use the Windows Snipping Tool built in the Accessories group in Windows 7,8 & 10!

3. In Studio you can press the Camera icon (to enact, go to View > Tool Windows > Android

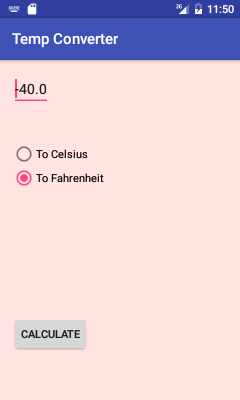
Monitor)  located in your Android Monitor view. Once you click the icon, right click on the

emulator and choose copy then paste result into Word (simpliest method).

Snapshot follows.

|  |  |
| --- | --- |
| From Studio | From Emulator strip |
|  |  |

Sample emulator output snapshot



If you are having problems with errors, go to the end of this lab for tips in the

Troubleshooting section.

**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 adding an interesting and catchy app icon to your app/Home Screen, changing the background color of your app depending on the temperature and the addition of certain images based on the temperature.

1. Adding in a **Home Screen** desktop icon for your app. This is a no-brainer.

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 where you will copy in the icon into your **res/drawables** folder under the root of your project. Note to copy files into your folder you can right click on your folder and choose paste.

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 for 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=*"****@drawable/temp****"*

::

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

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.



**For credit!**

Take a snapshot of your Home screen showing your icon display.

2. Adding in various background colors depending on the temperatures. This should be rather easy too, but will rely on some file tweaking and coding logic. Here you will allow for a background color change if the temperature display result is either too hot or too cold.

Open up your MainActivity.java file and tweak it as follows.Right towards the beginning of your class declaration, after the opening tag add in the following line of code

**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 (what’s in the red highlighted area) *after* the last if/else statements but *before* the **break**; line as follows

**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**);

}

//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*);

**break**;

You’ll notice in the code above, that a view object was created to point to the default id of app’s RelativeLayout element, namely **activity\_main** to allow for us to *alter* 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) in our case or include some hex values (#87ceff),aka *skyblue* like, 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 choose 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 to have the following import statement amongst your list of imports.

**import android.graphics.Color;**

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

Run and test code to see if things work out. App lookin’ pretty cool now huh?

3. Adding in images to your app. Ok let’s finish up. This last part will be similar to

your parts 1 & 2 above. 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. Place your picture files once again into your res/drawables folder.

Next open up your activity\_main.xml file in Design view mode. Under your Palette grab this time in a similar fashion as before, an **ImageView** component under the **Images & Media** section. Drag the component below your button. This component will help render your images when certain temperatures hit certain degrees. At the Resources screen, under Drawable selections, choose **** to set as your default opening image.

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 after your second if statement, add in the assigment statement for iv object shown in red below.

//grab result value now in Text Field

inputValue = Float.*parseFloat*(text.getText().toString());

view = findViewById(R.id.*mainLayout*);

iv=(ImageView) 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 (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.

**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 setings are causing rendering issues. *You can always adjust your Property settings for your component to tweak settings such as height/width setting them to a desired dp, as well as checking any other settings such as* the *scaletype property to adjust positioning (ex. layout\_alignParentLeft setting). Check the pull down choices for more info on a desired setting…*

For more on dpi’s and unnit measurements visit the following site:

<https://www.google.com/design/spec/layout/units-measurements.html>

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!

**STEP 10 Submit Your Program Code and Your Run Time Output**

For full credit make sure to include the following snapshots for a completed submission. Also 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 running application showing *each* background color that

triggers for a very warm temp and a very cold temp. Take a snapshot also when the temperature is between 0 and 90.

So in total you are turning in *five* snapshots for this lab!

Place all your snapshots into a Word document for credit. Label your snapshots accordingly.

Include 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 the first 7 letters of your last name followed by an underscore and the lab number

(ex. **sstudent\_lab1**).

Submit also as well a copy of all .java source files, and all your XML files you have created or

tweaked after your snapshots into your Word document as well. Label your files accordingly.

Ex. AndroidManifest.xml, Strings.xml, ConvertUtil.java, etc. Also include your entire app project

files zipped as well into BB. Proper documentation including comments in your java source files

will be throughly checked as being part of your grade so make sure to include this 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, aka Google vs. Android)

Experiment with different RAM settings, VM bytes for added acceleration/performance.

6. Check any code that maybe highlighted in red along the way, especailly when your Build

your project (do Build > Make Project as a check for errors). With a lot of code and small

fonts in your editor it may be hard sometimes to detect. Also if there are highlighted

errors, your file name iteslf maybe underlined in red indicating there are errors within your

code. Check for valid import statements!!!

Use the Messages view in Studio if it’s not visible, View > Tool Windows > Messages, to help pinpoint exact error details to check out, click on, etc., which can help you to go to the line in question.

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, etc. View > Tool Windows > Android Monitor for

 features especially or View >Tool Windows > Gradle Console.

Always have these open at runtime to monitor / debug things!

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