

Workout app

*** The most important note in this review is the flow that the application will have. As a programmer pay a lot of attention on the flow of application functionality ***

Functionality 1



The following interface was achieved by following these steps:

For the main logo the method is very straight forward as shown below:

```
<ImageView
    android:id="@+id/mainIv"
    android:layout_width="match_parent"
    android:layout_height="220dp"
```

```
android:src="@drawable/img_main_page"
app:layout_constraintTop_toTopOf="parent"
app:layout_constraintBottom_toTopOf="@+id/flStart"/!
```

There is no android inbuilt widget that can produce the shape of the start button. This means that we have to combine many programmer defined xml properties to achieve the desired shape. The layout best suited for this function is the frame layout since it has inbuilt functions that support layering. The code below shows how the design was achieved:

```
<FrameLayout
    android:id="@+id/flStart"
    app:layout_constraintTop_toBottomOf="@+id/mainIv"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    android:layout_width="150dp"
    android:layout_height="150dp"
    android:background="@drawable/ripple_effect_border"
    |

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="START"
        android:layout_gravity="center"
        android:textColor="@color/colorPrimary"
        android:textSize="22sp"
        android:textStyle="bold"/!
</FrameLayout|
```

The important thing to note here is the line :

```
android:background="@drawable/ripple_effect_border"
```

This is the line that converts the normal frame layout to the circular frame layout with a ripple effect. The drawable file **ripple_effect_border** is the custom xml file that has the design properties requires shown below

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Ripple effect has to have a color attribute. 040404-->
<ripple xmlns:android="http://schemas.android.com/apk/res/android"
    android:color="#E37654">

    <item android:id="@android:id/mask">
        <!-- Shape and color of the background that is to be masked -->
        <shape android:shape="oval">
            <solid android:color="@color/white"/>
        </shape>
    </item>

    <item android:drawable="@drawable/circular_border"/>
</ripple>
```

The first line defines the color that the ripple effect will have. The **android:color** tag is where the color is stated as shown below:

```
<ripple xmlns:android="http://schemas.android.com/apk/res/android"
    android:color="#E37654" |
```

Next is to define the area that will experience the ripple effect. The default color and shape of the area should be defined within the item tag identified by the **android:id="@android:id/mask"**. This id is an inbuilt feature of android studio. The code segment is shown below:

```
<item android:id="@android:id/mask" |
    <!-- Shape and color of the background that is to be masked -->
    <shape android:shape="oval" |
        <solid android:color="@color/white" /|
    </shape|
</item|
```

Lastly the frame layout was seen having a green border and circular shape with a white background. That is achieved by creating a separate xml file that will draw that characteristic and defining it as a drawable in this **ripple_effect_border.xml** file. This is done as follows:

The separate xml file that draws a green circular border with a white background

```

<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="oval">
    <stroke
        android:width="5dp"
        android:color="@color/colorAccent"/>
    <solid android:color="@color/white"/>
</shape>

```

Even though the shape is oval the fact that we defined the dimension of the frame layout equally as shown below:

```

android:layout_width='150dp'
android:layout_height='150dp'

```

The result will be a circular shape.

Defining this xml file as a drawable within the ripple_effect_border

```

<item android:drawable="@drawable/circular_border"/>

```

Functionality 2 : Using viewbinding and getting rid of an action bar

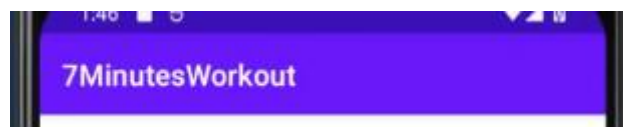


Image : This is an action bar

This is done by changing the theme that the project is using. The file is located in the res folder. By changing the theme to:

```

parent="Theme.MaterialComponents.DayNight.NoActionBar"

```

The action bar will be removed.

Viewbinding

For on to use viewbinding to locate screen widgets some changes have to be made to the gradle file. In the gradle file, specifically the **build.gradle(Module : Workout.app)**, within the android object the following should be added

```
buildFeatures{  
    viewBinding true  
}
```

Within the activity file view binding is used by first instantiating a variable called binding. After creating the binding variable it will be assigned a layout based on an inflated xml file. Lastly the binding variable will be used by the **setContentView()** method as shown below:

```
private var binding : ActivityMainBinding? = null
```

The above line is done outside the onCreate method while below is done within onCreate

```
super.onCreate(savedInstanceState)  
binding = ActivityMainBinding.inflate(layoutInflater)  
setContentView(binding?.root)
```

The **binding?.root** corresponds to this container in activity_main.xml:

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android" ...>
```

When using view binding remember to use the onDestroy() method to set the binding variable to null and prevent problems as shown below:

```
override fun onDestroy(){  
    super.onDestroy()  
    binding = null  
}
```

Functionality 3: Adding the exercise screen : Intents and customizing the tool bar



Image : Goal functionality

Creating tool bar

The first step will be to create the tool bar in the xml file with the below code.

```
<androidx.appcompat.widget.Toolbar
    android:id="@+id/tbExercise"
    android:layout_width="match_parent"
    android:layout_height="?android:attr/actionBarSize"
    android:theme="@style/Toolbar_theme"
    android:background="@color/white"
    app:titleTextColor="@color/colorPrimary"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    />
```

The most important item to consider here is the theme that the tool bar used. The theme used was programmer defined and how that came to be is as follows:

This is where we defined the theme of the tool bar. The characteristics of this theme was defined in the **theme.xml** file.

```
android:theme="@style/Toolbar_theme"
```

This is how it was defined in the **themes.xml** file:

```
<style name='Toolbar_theme' parent="@style/ThemeOverlay.MaterialComponents.ActionBar" |  
    <item name='colorControlNormal' | @color/toolbar_color</item |  
</style |
```

Now for setting up the tool bar using code. The code is self explanatory:

```
//Setting up the toolbar  
setSupportActionBar(binding?.tbExercise)  
//Adding a back button  
if(supportActionBar != null){  
    supportActionBar?.setDisplayHomeAsUpEnabled(true)  
}  
binding?.tbExercise?.setNavigationOnClickListener{  
    //This is the back button functionality  
    onBackPressed()  
}
```

Functionality 4 : Implementing custom UI elements for a progress bar to tick counterclockwise



Image : Timer progress bar

The goal of this is to create a progress bar with the above style and have it work as a timer. The progress bar will be within a frame layout since we will have to layer many xml files on top of each other. Below is the structure of the above widget:

```
▼ □ FrameLayout @+id/flProgressBar
  ▼ ↻ ProgressBar @+id/progressBar
    ▼ □ LinearLayout
      ▼ Ab TextView @+id/tvTimer
```

And here is the code for the framelayout. The background for the frame layout will be the **circular_border.xml**. It was used in the 1st functionality and the code is as follows:

Circular_border.xml


```

<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="oval">
    <stroke
        android:width="5dp"
        android:color="@color/colorAccent"/>
    <solid android:color="@color/white"/>
</shape>

```

The frame layout code is as follows:

```

<FrameLayout
    android:id="@+id/flProgressBar"
    android:layout_width="100dp"
    android:layout_height="100dp"
    android:layout_marginTop="10dp"
    android:background="@drawable/circular_border"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintTop_toBottomOf="@id/tbExercise"

```

Next is the progress bar. The progress bar will have the functionality of 'reducing' its green color as it counts down as shown below:



This is achieved by using the below code:

```

<ProgressBar
    android:id="@+id/progressBar"
    style="?Workout app
android:attr/progressBarStyleHorizontal"
    android:layout_width="100dp"
    android:layout_height="100dp"
    android:background="@drawable/circular_progress_bar_grey"
    android:progressDrawable="@drawable/circular_progress_bar"
    android:indeterminate="false"
    android:layout_gravity="center"

```

```
android:max='10'  
android:progress='100'  
android:rotation='-90'/>
```

We will begin by explaining the code **android:rotation = - 90**. If you imagine a compass put on top of the circle you will realize that by default the progress bar's starting point is in the east. For our case we want the start of the progress bar to be at the top or north is we use our metaphor. This therefore means that we have to turn 90 degrees back.

Now for the ring. There will be two different rings, a green one and a grey one. The green one is what will be on the foreground and as the timer starts counting down the grey ring will start appearing. The xml file used to draw the green ring is under the label **android:progressDrawable = @drawable/circular_progress_bar** whose code is as follows:

```
1  <?xml version="1.0" encoding="utf-8"?>  
2  <layer-list xmlns:android="http://schemas.android.com/apk/res/android">  
3      <item>  
4          <shape  
5              android:innerRadiusRatio="2.7"  
6              android:shape="ring"  
7              android:thicknessRatio="50.0"  
8              android:useLevel="true">  
9              <solid android:color="@color/colorAccent"/>  
10         </shape>  
11     </item>  
12 </layer-list>
```

This is simply the outer green ring. A layer list has been used since it has the capability of combining multiple drawables at once. This xml file has been used under the **android:progressDrawable** tag since it is the drawable that we want to use to display the progress of the horizontal progress bar. The grey background uses the following code:

```
1  <?xml version="1.0" encoding="utf-8"?>  
2  <layer-list xmlns:android="http://schemas.android.com/apk/res/android">  
3      <item>  
4          <shape  
5              android:innerRadiusRatio="2.7"  
6              android:shape="ring"  
7              android:thicknessRatio="50.0"  
8              android:useLevel="false">  
9              <solid android:color="@color/light_grey"/>  
10         </shape>  
11     </item>  
12 </layer-list>
```

Lastly is the green circle that also contains a text view. This is actually a linear layout which has a drawable of a circle as the background. The linear layout contains a textview which displays the number. A layout is used to ensure that the text view is in the middle of the custom timer. The code is as shown below:

```
59      <LinearLayout
60          android:layout_width="60dp"
61          android:layout_height="60dp"
62          android:layout_gravity="center"
63          android:background="@drawable/circular_color_background"
64          android:gravity="center">
65
66          <TextView
67              android:id="@+id/tvTimer"
68              android:layout_width="wrap_content"
69              android:layout_height="wrap_content"
70              android:textColor="@color/white"
71              android:textSize="25sp"
72              android:textStyle="bold"
73              tools:text="10"/>
74
75      </LinearLayout>
```

Functionality 5 : Using countdown timers to display seconds remaining

Here we are implementing the functionality of the timer to actually count down. We begin by creating 2 variables **restTimer (How long the rest is going to be)** and **restProgress (How much the rest is progressing)** as shown below:

```
private var restTimer : CountdownTimer? = null
private var restProgress : Int = 5
```

We will now create a method called **setRestProgressBar()** which takes as a parameter the amount of seconds resting will take place. The method simply creates a **CountDownTimer** objects that counts down and pastes the value of seconds to the textview. The code is as shown below:

```

private fun setRestProgressBar(totalRestTime : Int){
    restTimer = object : CountdownTimer((totalRestTime * 1000).toLong(), countdownInterval: 1000){
        //What should happen every tick
        override fun onTick(p0: Long) {
            binding?.progressBar?.max = totalRestTime
            restProgress++
            binding?.progressBar?.progress = totalRestTime - restProgress
            binding?.tvTimer?.text = "${totalRestTime - restProgress}"
        }
        //What should happen at the end
        override fun onFinish() {
            Toast.makeText(context: this@ExerciseActivity, text: "We will now start the exercise", Toast.LENGTH_LONG).show()
        }
    }.start() //Don't forget to start
}

```

In order to ensure that the timer object is destroyed after its done the following code is added. The method **onDestroy()** resets every thing back to default both the timer and the view binding as shown below:

```

override fun onDestroy() {
    super.onDestroy()
    if(restTimer != null){
        restTimer?.cancel()
        restProgress = 0
    }
    binding = null
}

```

Lastly we will also create another method called **setUpRestView()** which also takes as a parameter the total amount of rest time. The function of this method will be to set up the rest functionality of the work out app including the rest progress bar. For now the work of this method would be to reset the rest timer as shown by the code below:

```

private fun setUpRestView(totalRestTimeI : Int){

    if(restTimer != null){
        restTimer?.cancel()
        restProgress= 0
    }

    setRestProgressBar(totalRestTimeI)
}

```

Functionality 6 : Adding the exercise timer * xml similar to rest timer *

The point of this functionality is to start the exercise timer after the rest timer has ended. This will be achieved by playing with the visibility of the different timers. The code is very self explanatory but in a nutshell the process is as follows.

The rest timer is set up. At this point every widget of the exercise timer has the visibility of **gone**. This is to prevent them from taking any space. Once the Rest timer is over the method **setUpExerciseView()** is called from the **onFinish()** of the rest progress bar count down timer. This method simply makes the widgets of the rest view to become invisible and the widgets of the exercise view to become visible. The code is as shown below:

onFinish() of restTimer

```
//What should happen at the end
override fun onFinish() {
    setUpExerciseView(exerciseTime)
}
```

setUpExerciseView()

```
private fun setUpExerciseView(totalExerciseTimeI : Int){
    if(exerciseTimer != null){
        exerciseTimer?.cancel()
        exerciseProgress = 0
    }
    //Rest view widgets visibility = invisible. This is because other widgets position are relative to rest view widgets and therefore their
    existence cant be completely erased
    binding?.flProgressBar?.visibility = View.INVISIBLE
    binding?.tvTitle?.visibility = View.INVISIBLE
    binding?.restViewText?.visibility = View.INVISIBLE
    binding?.restViewExerciseName?.visibility = View.INVISIBLE

    binding?.tvExerciseName?.visibility = View.VISIBLE
    binding?.flProgressBarExercise?.visibility = View.VISIBLE
    binding?.ivImage?.visibility = View.VISIBLE
    //Position starts from zero but the ID for this exercise is 1
    binding?.ivImage?.setImageResource(exerciseList!![currentExercise].getImage())
    binding?.tvExerciseName?.text = exerciseList!![currentExercise].getName()

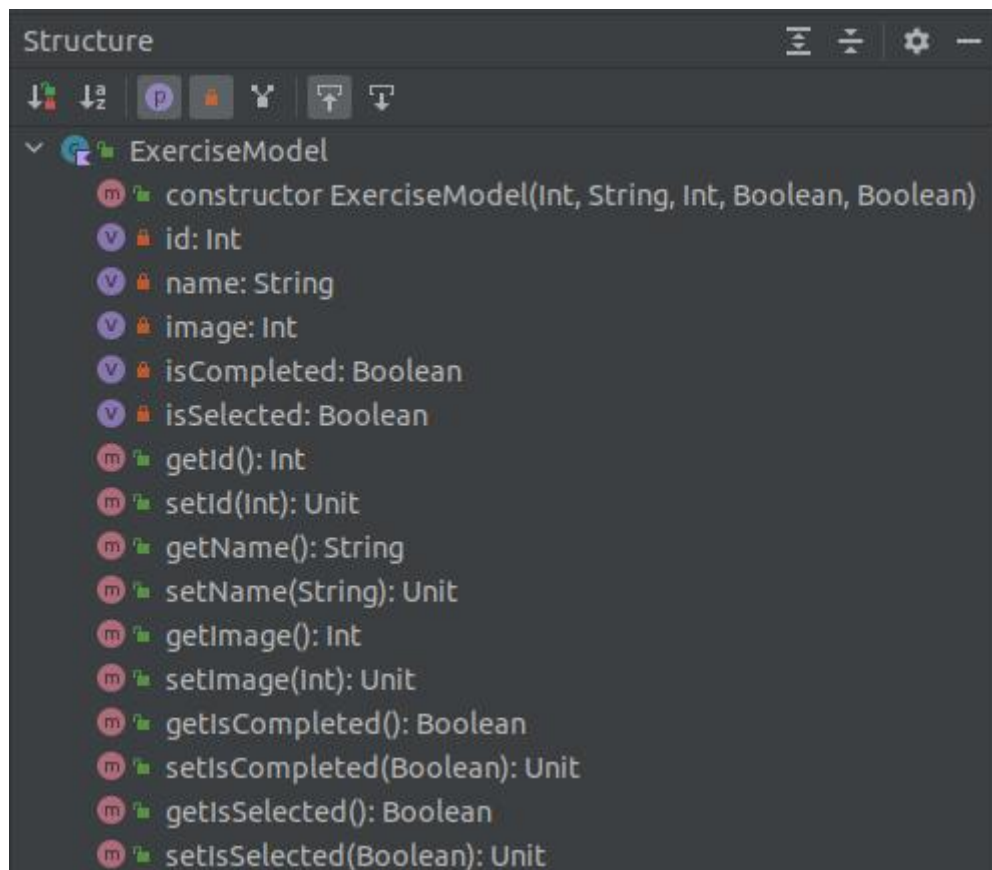
    //Manifest queries added
    speakOut( text: "Start ${exerciseList!![currentExercise].getName()}")
    setExerciseProgressBar(totalExerciseTimeI)
}
```

setUpExerciseProgressBar()

```
private fun setExerciseProgressBar(totalExerciseTime : Int){
    exerciseTimer = object : CountdownTimer((totalExerciseTime * 1000).toLong(), countDownInterval: 1000){
        override fun onTick(p0: Long) {
            binding?.progressBarExercise?.max = totalExerciseTime
            this@ExerciseActivity.exerciseProgress++
            binding?.progressBarExercise?.progress = totalExerciseTime - this@ExerciseActivity.exerciseProgress
            binding?.tvTimerExercise?.text = "${totalExerciseTime - this@ExerciseActivity.exerciseProgress}"
        }
        override fun onFinish() {...}
    }.start()
}
```

Functionality 7 : Adding the model, constants and exercises

This includes adding images for the different exercises and the model of what each exercise should comprises of. We will create a new class called **ExerciseModel**. The variables defined in this model each have a getter and a setter. The structure of the class should be enough to describe its contents



Next is to create a constants companion object which would mean that the variables stored here would be accessible through out the project. This is where all the exercises will be stored. An arraylist with all the exercises will be returned by the method **defaultExerciseList()** as shown below:

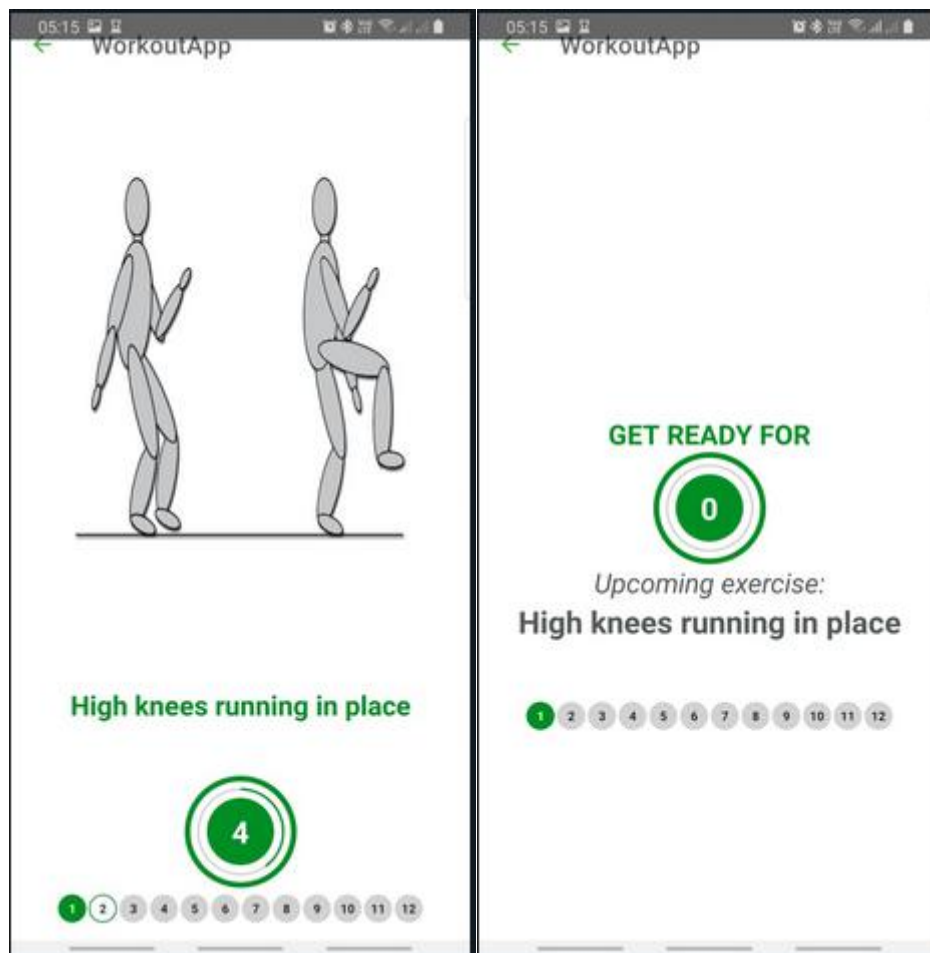
```
fun defaultExerciseList(): ArrayList<ExerciseModel>
```

In order to access the exercise list from the **exerciseactivity** we will first create an arraylist variable and then call the **defaultExerciseList()** method to return the list as shown below:

```
private var exerciseList : ArrayList<ExerciseModel>? = null
exerciseList = Constants.defaultExerciseList()
```

This is the list that is used to setup the exercises in the app.

Functionality 8 : Preparing a recyclerview at the bottom to display exercise progress



This is the recycler view we are trying to create with the functionality of highlighting the executed exercise.

We will start by defining how one item will look like. One item will be a simple text view with a circular background as shown below:

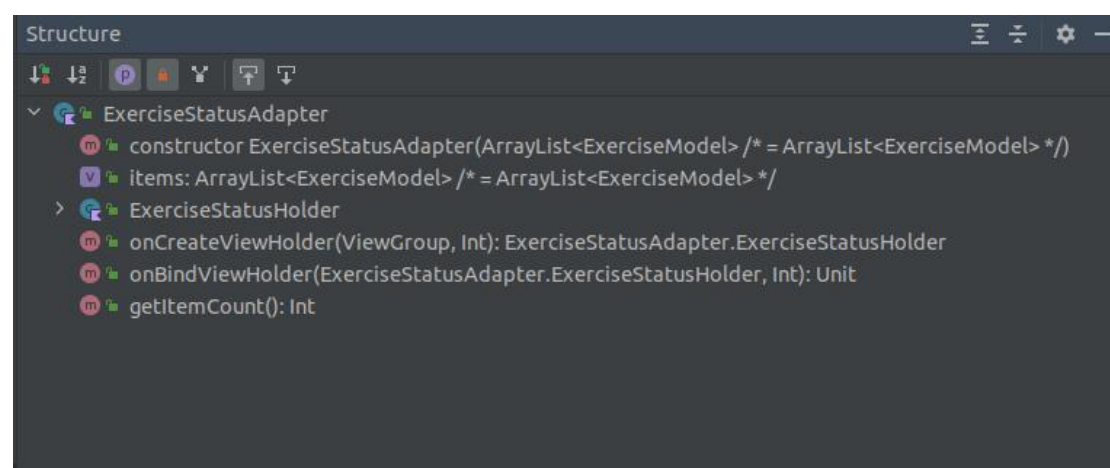
MainActivity

```
2 <TextView xmlns:android="http://schemas.android.com/apk/res/android"
3     android:id="@+id/tvItem"
4     android:layout_width="25dp"
5     android:layout_height="25dp"
6     android:layout_margin="1dp"
7     android:background="@drawable/item_circular_background"
8     android:gravity="center"
9     android:textAlignment="center"
10    android:padding="5dp"
11    android:textColor="#212121"
12    android:textStyle="bold"
13    android:text="1"
14    android:textSize="10sp"
15 />
```

The background drawable is a circle with a grey background as shown below:

```
<?xml version='1.0' encoding='utf-8' ?I
<shape xmlns:android='http://schemas.android.com/apk/res/android'
    android:shape='oval' I
    <solid android:color='@color/light_grey' /I
</shape I
```

The next step is creating an adapter for the recyclerview. We now create a new class called **ExerciseStatusAdater** whose structure is as shown below:



```
Structure
ExerciseStatusAdapter
  constructor ExerciseStatusAdapter(ArrayList<ExerciseModel> /* = ArrayList<ExerciseModel> */)
  items: ArrayList<ExerciseModel> /* = ArrayList<ExerciseModel> */
  > ExerciseStatusHolder
    onCreateViewHolder(ViewGroup, Int): ExerciseStatusAdapter.ExerciseStatusHolder
    onBindViewHolder(ExerciseStatusAdapter.ExerciseStatusHolder, Int): Unit
    getItemCount(): Int
```

This tells us that the main class **ExerciseStatusAdapter** takes has a constructor that takes as a parameter an ArrayList which stores elements of type **ExerciseModel**. The arraylist that is passed is called **items**. Next is the the same class has an inner class

called **ExerciseStatusHolder** which is the viewholder class of this adapter. This is the area where the variables **isCompleted** and **isSelected** are used since a completed exercise will have a different color from a finished routine. Routines that are yet to be done will also have a different color from the previous two routines.

Below is the first part of this class where the constructor and the inner class code have been defined:

```
class ExerciseStatusAdapter(val items : ArrayList<ExerciseModel>) : RecyclerView.Adapter<ExerciseStatusAdapter.ExerciseStatusHolder>() {
    //The first line has inflated the item view(item_exercise_status.xml) meaning that its components are now accessible
    inner class ExerciseStatusHolder(val itemBinding : ItemExerciseStatusBinding) : RecyclerView.ViewHolder(itemBinding.root){

        //Accessing the textview from the item_exercise_status.xml
        val tvItem : TextView = itemBinding.tvItem

        //User defined method that places the id of the current exercise at the text to be seen in the textview
        fun bindExerciseStatus(exerciseModel: ExerciseModel){
            itemBinding.tvItem.text = exerciseModel.getId().toString()
        }
    }
}
```

Note that the data is not bound at this point. The data will always be bound in the **onBindViewHolder()** when using recycler view. At this point we are simply creating a method that will bind the data to the recycler view item.

Next are the methods that have to be implemented when a recycler view is used. The **onCreateViewHolder()** and **getItemCount()** methods are self explanatory as shown below:

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ExerciseStatusHolder {
    return
    ExerciseStatusHolder( ItemExerciseStatusBinding.inflate(LayoutInflater.from(parent.context),parent,false))
}
```

```
override fun getItemCount(): Int {
    return items.size
}
```

The interesting bit is the **onBindViewHolder()** which is where the functionality of different color change is implemented. The first thing is binding the data to the recycler view item. The method **bindExerciseStatus()** will handle that functionality as shown below:

```
override fun onBindViewHolder(holder: ExerciseStatusHolder, position: Int) {
    //Get current exercise
    val model : ExerciseModel = items[position]
    //Binding the data to the recycler view item
    holder.bindExerciseStatus(model)
}
```

This is where the variable items was defined:

```
class ExerciseStatusAdapter(val items : ArrayList<ExerciseModel>)
```

*** note that the holder will always have the type of the defined inner class ***

For the color change the status of the particular exercise, whether it is selected or completed, is retrieved using getters that were defined in the **ExerciseModel** class. How the colors are set is self explanatory as shown below:

```
when{
    model.isSelected() -> {
        //itemView is an internal variable representing a single item in the ViewHolder
        holder.itemView.background = ContextCompat.getDrawable(holder.itemView.context,R.drawable.item_exercise_status_selected)
        holder.itemView.setTextColor(Color.parseColor( "colorString: "#212121"))
    }
    model.isCompleted() -> {
        holder.itemView.background = ContextCompat.getDrawable(holder.itemView.context,R.drawable.circular_color_background)
        holder.itemView.setTextColor(Color.parseColor( "colorString: "#ffffff"))
    }
    else -> {
        holder.itemView.background = ContextCompat.getDrawable(holder.itemView.context,R.drawable.item_circular_background)
        holder.itemView.setTextColor(Color.parseColor( "colorString: "#212121"))
    }
}
```

Now the next step is to display the recycler view in the exercise activity. We begin by creating an adapter object as shown below:

```
private var exerciseAdapter : ExerciseStatusAdapter? = null
```

We will then create a function to complete setting up the adapter by setting functionalities such as layout manager property and assign the adapter variable to the recyclerView widget in the xml as shown below:

```
private fun setUpExerciseStatusRecyclerView(){
    binding?.rvExerciseStatus?.layoutManager = LinearLayoutManager( context: this,LinearLayoutManager.HORIZONTAL, reverseLayout: false)
    //This therefore means that the method will be called after the exerciseList has been instantiated
    exerciseAdapter = ExerciseStatusAdapter(exerciseList!!)
    binding?.rvExerciseStatus?.adapter = exerciseAdapter
}
```

The exercise list being instantiated means that the below code exists before the method is written in the onCreate function.

```
exerciseList = Constants.defaultExerciseList()
```

Next thing we have to do is display on the recycler view the exercise that is being currently done as shown below. The current exercise number should have a different background:



How this was achieved in terms of the layout is described in the **onBindView()** method. In the exercise activity the **onFinish()** in the **setRestProgressBar()** will set **isSelected** variable to true so that when heading to the exercise it is already chosen as shown below:

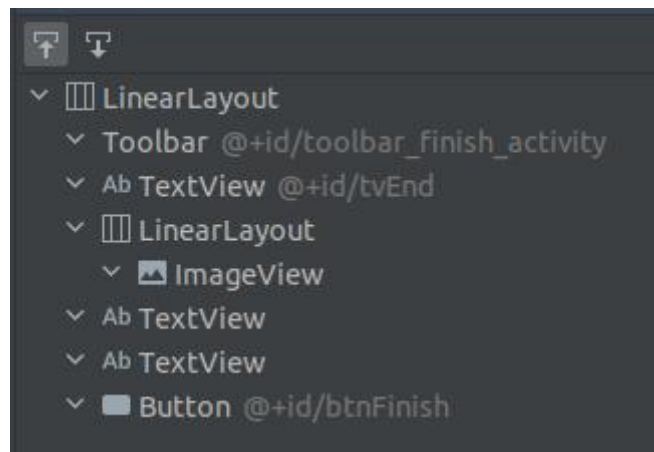
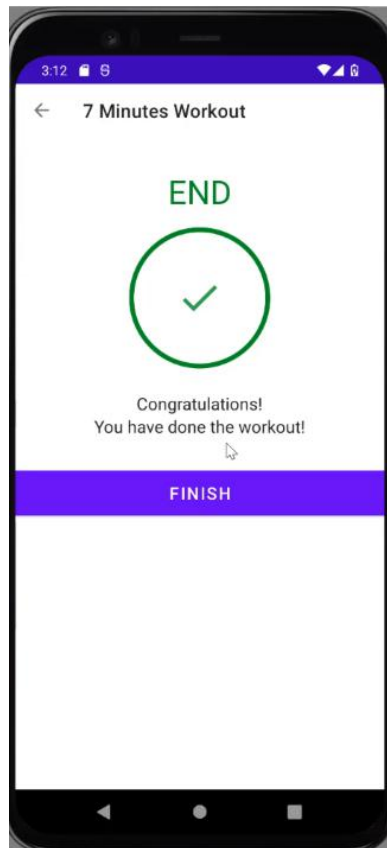
```
//What should happen at the end
override fun onFinish() {
    currentExercise++
    //When the rest timer is over the next exercise is automatically selected
    exerciseList!![currentExercise].setIsSelected(true)
    //To change recyclerView appearance
    exerciseAdapter!!.notifyDataSetChanged()
    setUpExerciseView(exerciseTime)
}
```

For the is complete section the **isSelected** is set as false and **isCompleted** changed to true in the **setUpExerciseProgressBar()** as shown below:

```
override fun onFinish() {
    exerciseList!![currentExercise].setIsSelected(false)
    exerciseList!![currentExercise].setIsCompleted(true)
    //To change recyclerView appearance
    exerciseAdapter!!.notifyDataSetChanged()
}
```

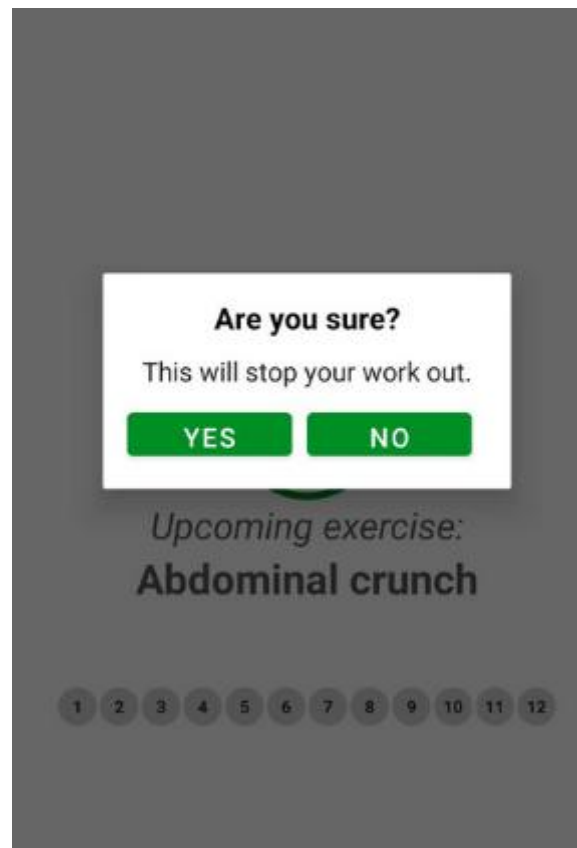
In both instances the line **exerciseAdapter!!.notifyDataSetChanged()** is used in order to inform the adapter that a change in data has occurred and therefore the layout of the recycler view has to be changed. If this line is not used there will be issues with the recycler view.

Functionality 9 : Creating the finish activity

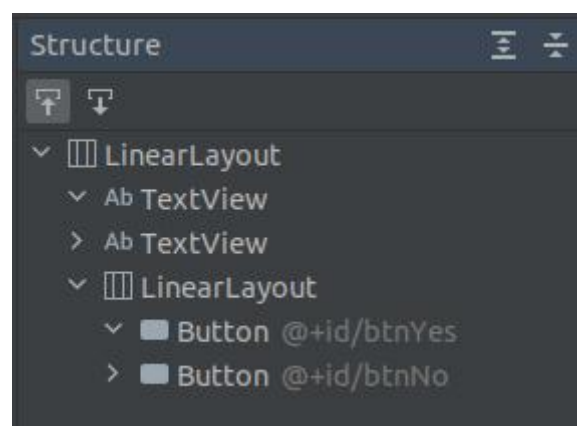


Above is the structure of the xml file used to create the finish activity layout. The most important thing to note is that the circle shape is as a result of the inner linear layout having the background of a circular drawable.

Functionality 10: Setting up a custom dialog for the back button



We begin by building a custom dialog layout file. The layout file we have created looks as shown below:



This functionality will now be implemented under the method **customBackButton()**. This method inflates the custom layout file enabling us to access its specific widgets. All other aspects of the intended functionality is self explanatory as shown in the code below:

```
private fun customBackButton() {  
    val customDialog = Dialog(this)  
    //Inflate the custom dialog layout file  
    val dialogBinding = CustomDialogBackButtonBinding.inflate(layoutInflater)  
    customDialog setContentView(dialogBinding.root)  
    //can not cancel dialog by clicking outside of it  
    customDialog.setCanceledOnTouchOutside(false)  
  
    dialogBinding.btnYes.setOnClickListener{  
        //The activity to be closed must be specified  
        this@ExerciseActivity.finish()  
        customDialog.dismiss()  
    }  
    dialogBinding.btnNo.setOnClickListener{  
        customDialog.dismiss()  
    }  
    customDialog.show()  
}
```

Functionality 11: Having a similar user interface for metric and imperial units

METRIC UNITS	IMPERIAL UNITS
WEIGHT (in kg)	WEIGHT (in pounds)
HEIGHT (in cm)	Feet Inches
CALCULATE	CALCULATE

The widget above is simply a customized radio group with customized radio buttons. Customization is done using xml files. We will begin by analyzing the code structure of the radio group which is shown below:

```
<RadioGroup
    android:id="@+id/rgUnits"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_below="@id/tb_BMI"
    android:layout_margin="10dp"
    android:background="@drawable/radio_group_tab_background"
    android:orientation="horizontal">
    <!-- -->
```

The important thing to note here is the use of a drawable resource to create the customized background. The drawable resource is simply a rectangle which has corner as shown in the code snippet below:

```
<?xml version='1.0' encoding='utf-8'?>
<shape android:shape='rectangle' xmlns:android='http://schemas.android.com/apk/res/android' >
    <solid android:color='#ECEDEF' />
    <stroke android:width='0.5dp'
        android:color='#9AA2AF' />
    <corners android:radius='30dp' />
</shape>
```

Next are the individual radio buttons. The xml for the radio buttons is also straight forward except for a few instances. The code for the metric radio button is as shown below:

```
<RadioButton
    android:id="@+id/rbMetric"
    android:layout_width='0dp'
    android:layout_height='35dp'
    android:layout_weight='1'
    android:background="@drawable/tab_selector"
    android:button="@null"
    android:checked='true'
    android:gravity='center'
    android:text='METRIC UNITS'
    android:textColor="@drawable/units_tab_color_selector"
```

```
android:textSize='15sp'  
android:textStyle='bold'/>
```

The first exception is obviously the background. The xml file of the background is as shown below:

```
1 <?xml version="1.0" encoding="utf-8"?>  
2 <selector xmlns:android="http://schemas.android.com/apk/res/android">  
3   <item android:state_checked="true">  
4     <shape android:shape="rectangle">  
5       <solid android:color="@color/colorAccent"/>  
6       <corners android:radius="30dp"/>  
7     </shape>  
8   </item>  
9 </selector>
```

The root tag is a selector due to the use of the radio button widget. Another important tag is the item tag with **android:state_checked**. This means that the xml definitions in this tag will only apply to a radio button if its state is checked. That is why by default the metric radio button has the tag **android:checked** that has been set to true.

Now we have also realized that the text color can also be defined using an xml file. The xml file defining the text color of the radio button is as shown below:

```
1 <?xml version="1.0" encoding="utf-8"?>  
2 <!-- Colors the layout based on the state without the need for an OnClickListener -->  
3 <selector xmlns:android="http://schemas.android.com/apk/res/android">  
4   <item android:color="#FFFFFF" android:state_checked="true"/>  
5   <item android:color="#ADB3B3" android:state_checked="false"/>  
6 </selector>
```

The selector tag is also used as the root tag. When the state of the radio button is true the text color will be set to white and when the state of the radio button is false the text will set to the color grey. The last important tag is the **android:button** tag. It is used to determine whether the default radio button drawable will be present as shown below:



Without android:button = "@null"



With android:button = "@null"

Now we have to create the layout for the US view elements. Again the code is pretty straight forward apart from how the feet and inches textinput layout was created. A view which is under the metric weight textinputlayout is created. It has a width and height of 0dp. Its centered horizontally so the end of the inches textinputlayout and start of the feet textinputlayout are manipulated accordingly to enable the layout seen. The feet input text box will be to the left of the view(**android:layout_toStartOf**) and the inches textinput will be to the right of the view widget(**android:layout_toEndOf**).The code that enables that is as shown below:

```
<View
    android:id="@+id/view"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_below="@id/tilMetricWeight"
    android:layout_centerHorizontal="true"/>
```

For the feet input notice the layout tag android:layout_toStartOf

```
<com.google.android.material.textfield.TextInputLayout
    android:id="@+id/tilImperialHeightFeet"
    android:layout_below="@id/tilMetricWeight"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_toStartOf="@id/view">
```

For the inches input notice the tag android:layout_toEndOf

```
<com.google.android.material.textfield.TextInputLayout
    android:id="@+id/tilImperialHeightInch"
    android:layout_below="@id/tilMetricWeight"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_toEndOf="@id/view">
```

That is it for the layout, now for the code that enables the functionality of only one of the two measurements to be seen. We will begin by creating a companion object that stores text that will be used as a flag to tell which layout is active as shown below:

```
companion object{
    private const val METRIC_UNITS_VIEW = 'METRIC_UNITS_VIEW'
    private const val IMPERIAL_UNITS_VIEW = 'IMPERIAL_UNITS_VIEW'
}
```

Next we will create methods that make either metric or imperial inputs visible based on the choice picked. Below is the method that will make metric inputs visible:

```
private fun makeMetricVisible(){
    currentView = METRIC_UNITS_VIEW
    binding?.etMetricWeight?.hint = "WEIGHT (in kg)"
    binding?.etMetricHeight?.hint = "HEIGHT (in cm)"
    binding?.tilMetricHeight?.visibility = View.VISIBLE
    binding?.tilImperialHeightFeet?.visibility = View.GONE
    binding?.tilImperialHeightInch?.visibility = View.GONE

    //Clear values
    binding?.etMetricWeight?.text!!.clear()
    binding?.etMetricHeight?.text!!.clear()
    binding?.llDisplayBmiResult?.visibility = View.INVISIBLE
}
```

And here is the view that will make imperial inputs visible:

```
private fun makeImperialVisible(){
    currentView = IMPERIAL_UNITS_VIEW
    binding?.etMetricWeight?.hint = "WEIGHT (in pounds)"
    binding?.tilMetricHeight?.visibility = View.INVISIBLE
    binding?.tilImperialHeightFeet?.visibility = View.VISIBLE
    binding?.tilImperialHeightInch?.visibility = View.VISIBLE

    //Clear values
    binding?.etMetricWeight?.text!!.clear()
    binding?.etImperialHeightFeet?.text!!.clear()
    binding?.etImperialHeightInch?.text!!.clear()
    binding?.llDisplayBmiResult?.visibility = View.INVISIBLE
}
```

The similarity in the two methods is that the linear layout which has the textviews that report the result of the calculations are made invisible. This is because the functionality we wish to have is that when the measurement type is changed for example from metric to imperial, the result of the metric measurements should not appear in the imperial state.

The main function of the **currentView** variable is that it is used during calculation of the BMI result as shown below:

```

private fun calculateBMI(){
    if(currentView == METRIC_UNITS_VIEW){
        if(validateMetricUnits()){
            val heightValue : Float = binding?.etMetricHeight?.text?.toString().toFloat() / 100
            val weightValue : Float = binding?.etMetricWeight?.text?.toString().toFloat()
            val BMI : Float = weightValue / (heightValue*heightValue)
            displayBMIResults(BMI)
        }else{
            Toast.makeText( context: this, text: "Please enter valid entries",Toast.LENGTH_LONG).show()
        }
    }
    else if(currentView == IMPERIAL_UNITS_VIEW){
        if(validateImperialUnits()){
            val weightValue : Float = binding?.etMetricWeight?.text?.toString().toFloat()
            val heightValueFeet : Float = binding?.etImperialHeightFeet?.text?.toString().toFloat()
            val heightValueInches : Float = binding?.etImperialHeightInch?.text?.toString().toFloat()
            val heightValue : Float = heightValueFeet*12 + heightValueInches
            val BMI : Float = 703 * (weightValue / (heightValue*heightValue))
            displayBMIResults(BMI)
        }else{
            Toast.makeText( context: this, text: "Please enter valid entries",Toast.LENGTH_LONG).show()
        }
    }
}
}

```

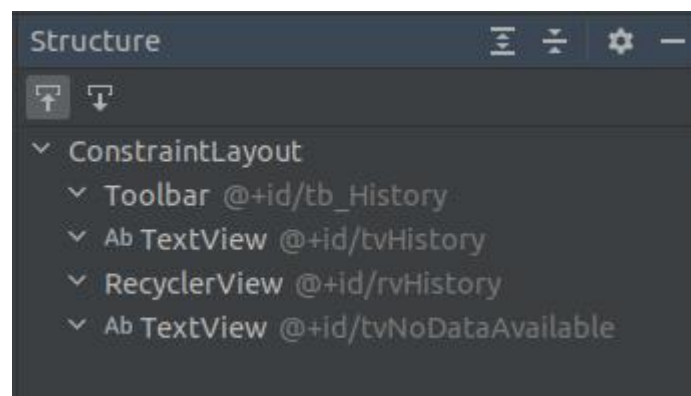
Finally we have to access the radio group widget to implement the functionality as it is being clicked. We will not use the normal `onClickListener` rather we will take the **`setOnCheckedChangeListener`** which is a lambda that takes the radio group as a parameter and returns automatically the id of the item that was clicked enabling access and consequently the ability to change the view:

```

binding?.rgUnits?.setOnCheckedChangeListener{ _,checkedId : Int ->
    if(checkedId == R.id.rbMetric){
        makeMetricVisible()
    }else if(checkedId == R.id.rbImperial){
        makeImperialVisible()
    }
}
}

```

Functionality 12: History of completed workouts



EXERCISE COMPLETED

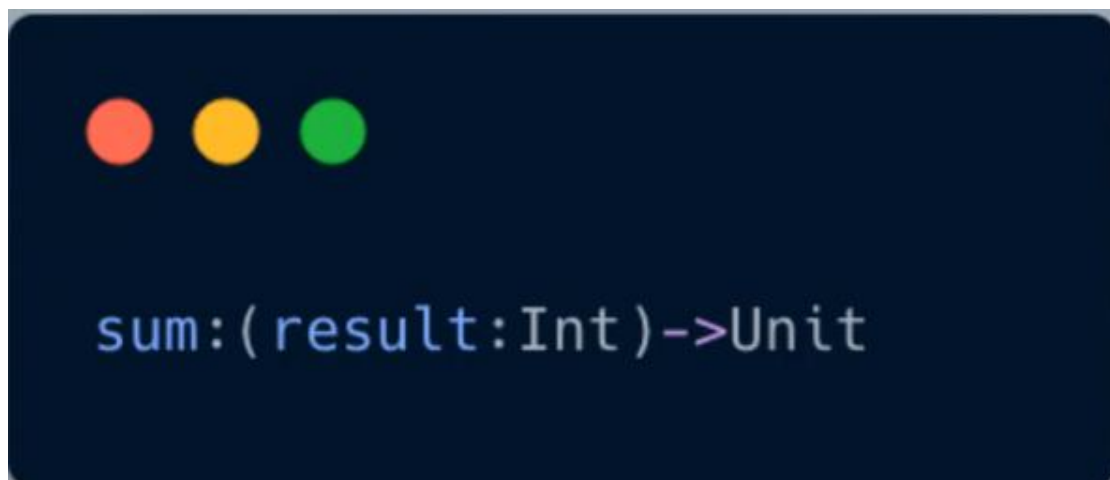
item 0
item 1
item 2
item 3
item 4
item 5
item 6
item 7
item 8
item 9

NO DATA AVAILABLE

One text view for showing that there is no data available if there are no exercises present.

Lambdas.

These are functions that have no method name as part of their method signature. They have the unique ability to take functions as well as variables as parameters: An example of a lambda that takes a variable as a parameter is as follows:



The lambda name is sum, result is a variable parameter and it returns nothing. Lambdas can also be passed as constructor parameters. Below is a snippet that shows how this is done:

```
class Arithmetic(private val sum:(result : Int) -> Boolean){
```

The lambda variable can then be used during initialization as seen in the init tag

Lambdas can also be called using an invoke tag as shown below:

```
class Arithmetic(private val sum:(result : Int) -> Boolean){
    //First we have to invoke the lambda so that we can pass values to it
    init{
        //Once the lambda is used this is the value that will be passed as a parameter
        sum.invoke( result: 5);
    }
}
```

Surrounding a lambda tag with {} enables a lambda to have a method body. Notice in the image below that the lambda that has been passed begins with a curly bracket and ends with one. To increase readability the () that is usually present when instantiating a class can be removed :

```

fun main(args: Array<String>) {
    println("Hello World!")

    Arithmetic { result ->
        if (result > 5) {
            print(result)
            true ^Arithmetic
        } else {
            print(false)
            false ^Arithmetic
        }
    }
}

```

The method body of a lambda can also be external in form of another method. The external method that will be used as the method body of the lambda must take the same parameters and have the same return type as shown below:

```

//Using a function as the method body of the lambda
Arithmetic {result -> operation( result: 9)}

fun operation(result: Int) : Boolean{
    if(result > 5) {
        //Lambda for printing
        print(result)
        return true
    } else {
        print(false)
        return false
    }
}

```

Roomdatabase for storing workouts

The first step is to create a history entity which is where the table that is going to be stored in the database is created. Our history entity is going to store the date as shown below:

```

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "history_table")
data class HistoryEntity(
    @PrimaryKey
    val date : String
)

```

Next is the DAO that will just insert the data as shown below:

```

@Dao
interface HistoryDao {

    @Insert
    suspend fun insert(historyEntity: HistoryEntity)

    @Query("SELECT * FROM 'history_table'")
    fun fetchAllDates() :Flow<List<HistoryEntity>>
}

```

Finally we need to create the database as shown below:


```

@Database(entities = [HistoryEntity::class], version = 1)
abstract class HistoryDataBase : RoomDatabase() {

    abstract fun historyDao () : HistoryDao

    companion object{
        @Volatile
        // Written in a way such that it is completely visible to
        other strings
        private var INSTANCE : HistoryDataBase? = null

        fun getInstance(context: Context) : HistoryDataBase{
            synchronized( lock: this){
                var instance = INSTANCE
                if(instance == null){
                    instance = Room.databaseBuilder(context
                .applicationContext,HistoryDataBase::class.java,
                name: "history_database").fallbackToDestructiveMigration().build()
                    INSTANCE = instance
                }
                return instance
            }
        }
    }
}

```

We also have to create a class that inherits from Application and instantiate the database by getting the instance from there so that it is visible to all classes within this app. **This class must also be included in the manifest**

```

import android.app.Application

class WorkOutApp : Application() {
    val db by lazy {
        HistoryDataBase.getInstance( context: this)
    }
}

<application
    android:name = ".WorkOutApp"

```

Class that instantiates the database

We will now create a method that will insert dates into the database. We will do this at the finish activity since our intention is to store finished exercises. Before inserting the data we have to create the database that will then be used to access the DAO to insert data as shown below

```
//Set up database in order to call the Dao
val historyDao = (application as WorkOutApp).db.historyDao()
```

The method that will store the date is self explanatory as shown below:

```
private fun addDateToDatabase(historyDao: HistoryDao){
    //Prepare date that will be called
    val c = Calendar.getInstance()
    val dateTime = c.time
    //MMM{Jan, Feb, Mar...} MM{01, 02, 03...}
    val sdf = SimpleDateFormat( pattern: "dd MMM yyyy HH:mm:ss", Locale.getDefault())
    val date = sdf.format(dateTime)
    lifecycleScope.launch { this: CoroutineScope
        historyDao.insert(HistoryEntity(date))
        Log.e( tag: "Date Added", msg: "OK")
    }
}
```

In terms of displaying the data to the adapter the methodology is the same. The only unique thing is the fact that the view of alternating grey and white background when displaying the dates is as shown below:

```
if(position % 2 == 0){
    //The context is needed if the color value is gotten from our resource file
    holder.llHistoryItemMain.setBackgroundColor(ContextCompat.getColor(holder.itemView.context, R.color.light_grey))
}else{
    //Parsing a color does not need a context
    holder.llHistoryItemMain.setBackgroundColor(Color.parseColor("#FFFFFF"))
}
```

suspend and flow are not used together