Lesson 1 - Random Images

Objectives

- Modify the xml layout file to specify LinearLayout, TextView and Button widgets, their id attribute and layout attributes
- Describe nested classes and anonymous classes in java
- use the instance method findViewById()
- Describe the R class in android
- Explain what is meant by inflating the layout
- Write java code in onCreate()
- Write java code to modify the text attribute of a widget
- Write java code to implement a callback

Explaining the XML Layout File

Linear Layout

Edit the XML file generated for you by replacing **`ConstraintLayout`** with **`LinearLayout`**.

In Linear Layout:

- The widgets are stacked in sequence according to the orientation.
- Two possible orientations: **`horizontal`** and **`vertical`**
- If no orientation attribute is specified, the default orientation is horizontal.

```
<LinearLayout
android:orientation = ”vertical”
 >
```

The three TextView widgets are stacked vertically.

```
<LinearLayout
android:orientation = ”horizontal”
>
```

The three TextView widgets are stacked horizontally.

TextView Widget

An XML tag for a basic TextView Widget is shown below.

```
<TextView
    android:id="@+id/myTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:gravity="end"
    android:text="second"/>
```

- The **`id` attribute** enables you to give a unique ID to each widget in the XML layout file. This allows you to access the widget through the java code.
- The **`text` attribute** specifies the text that the widget should contain.

Button Widget

A possible XML tag of a basic Button widget is shown below.

```
<Button
    android:id="@+id/myButton"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Click Me"/>
```

Here, I remove some attributes, showing you the minimum necessary to specify a widget.

How are the TextView and Button classes related?

Have a look at the documentation

https://developer.android.com/reference/android/widget/Button

Sizing A Widget

For the **`layout_width`** and **`layout_height`** attributes

- **`wrap_content`** sizes the widget to fit the content
- **`match_parent`** sizes the widget to fit the screen size

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="AA"/>
```

The widget sizes itself to fit its content.

```
<TextView
    android:layout_width=”match_parent"
    android:layout_height="wrap_content"
    android:text="AA"/>
```

The width of the widget is equal to the width of the screen.

Alignment

Note the difference between the two:

- To align a widget within a layout, use the **`layout_gravity`** attribute (child to parent)
- To align the contents of a widget within itself, use the **`gravity`** attribute (parent to child)

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content”
    android:layout_gravity=”center"
    android:text="AA"/>
```

The widget aligns itself to the centre of the layout.

```
<TextView
    android:layout_width=”match_parent"
    android:layout_height="wrap_content”
    android:gravity=”center"
    android:text="AA"/>
```

The contents of the widget aligns itself in the centre of the widget.

To understand the difference:

- In which scenario would the **`gravity`** attribute have no effect?
- In which scenario would the **`layout_gravity`** attribute have no effect?

Random Class

In many applications it is useful to generate random numbers.

In Java, you do it by getting an instance of the **`Random`** class.

In this class there are three useful methods

- **`nextInt()`** gives you an integer between -2^{32} and 2^{32} (exclusive)
- **`nextInt(n)`** gives you an integer between 0 and n (exclusive)
- **`nextDouble()`** gives you a double between 0.0 and 1.0

```
Random r = new Random();
r.nextInt();
r.nextInt(100);
r.nextDouble();
```

Random number generators usually need to be initialized with a seed.

If you need the sequence of random numbers to be the same, you use the same seed.

If not, one way to get a changing seed is to use the Date object.

```
Date d = new Date();
Random r = new Random(d.getTime());
```

Nested Classes

A class definition can contain class definitions. We call these classes **nested classes**.

```
public class OuterClass {
   // code not shown

   class InnerClass{
       //code not shown
   }

}
```

This is typically done when you have classes that logically depend on the outer class and are used together with the outer class.

Inner Class

A nested class that is not declared static is called an **Inner Class**.

- To instantiate an inner class, you need an instance of the outer class, which is usually called the **enclosing class**.
- The inner class can access all methods and variables of the enclosing outer class.
- The inner class can access the instance of the outer class by the variable **`OuterClass.this`**
- You can prefix the inner class declaration using the four access modifiers. Recall that top-level classes can only have access modifier **`public`** or be package-private.

```
public class OuterClass {

   int a;
   OuterClass(){  a = 10;  }
   void outerPrintA(){  System.out.println(a);  }

   class InnerClass{
       int c;

       InnerClass(){ c = 100; }

       void innerPrintA(){  System.out.println(a);  }

       OuterClass giveBackOuter(){  return OuterClass.this; }
   }
}
```

Activity. For **`OuterClass`**, complete the main function below to illustrate the following properties.

```
public class TestOuterClass {
   public static void main(String[] args){
       //Instantiate OuterClass
       OuterClass outerClass = new OuterClass();

       //Instantiate the InnerClass
       OuterClass.InnerClass innerClass = outerClass.new
InnerClass();

       //Show that InnerClass can access variables in OuterClass

       //Show that InnerClass stores a reference to OuterClass
   }
}
```

Static Nested Classes

By declaring a nested class as static, it is known as a **static nested class.**

- It can only access static variables and methods in the outer class.
- It can be instantiated without an instance of the outer class.

A static nested class behaves like a top-level class and is a way to organize classes that are used only by some other classes.

Activity.

- Modify **`OuterClass.java`** by declaring **`InnerClass`** as static and adjusting other parts of the class accordingly e.g. which other variables must be static? Which methods do not work anymore?
- Write code to show that you can instantiate OuterClass and InnerClass separately.

Nested Interface & Anonymous Classes

Recall that interfaces make your code reusable. We may nest interfaces as well. Recall also that Interfaces are inherently static. In the following code, any object that implements **`Foo.Bar`** interface can be passed to **thirsty()**.

```
public class Foo {

   interface Bar{
       void drink();
   }

   Foo(){
   }

   void thirsty(Bar bar){
       bar.drink();
   }

}
```

The inner class **`C`** implements **`Foo.Bar`** and an instance is passed to **`thirsty()`**. The inner class is declared **`static`** because it is invoked from **`main()`**.

```
public class TestFoo {

   public static void main(String[] argv){
       Foo f = new Foo();
       f.thirsty( new C() );
   }

   static class C implements Foo.Bar {
       @Override
       public void drink() {
           System.out.println("gulp");

       }
   }
}
```

Anonymous Class

Often, if the Inner Class is used only once, an alternative is an **Anonymous Inner Class**, to avoid declaring too many classes. You may not want to have too many inner classes that are practically used only once.

The following code shows how the **`TestFoo`** example above can be implemented using an anonymous inner class.

```
public class TestFoo1 {

   public static void main(String[] argv){

       Foo f = new Foo();
       f.thirsty( new Foo.Bar(){
           @Override
           public void drink() {
               System.out.println("Gulp");
           }
       });
   }
}
```

The **`new`** keyword is used to instantiate an object that implements **`Foo.Bar`**. Because **`Foo.Bar`** is an interface, the implementation is then specified immediately.

As you can see, we have an **anonymous class** because

- We do not name the class that implements the interface
- We do not assign a variable name to the class that implements the interface

We see nested interfaces, nested static classes and anonymous classes in Android programming frequently.

Delegation

We go back to the **`Foo`** class and notice that what happens when **`thirsty()`** is executed depends on objects implementing **`Foo.Bar`** that are passed to it.

In other words, the behaviour of **`thirsty()`** is **delegated** to objects that implement **`Foo.Bar`**, (thanks to subtype polymorphism).

This illustrates two design principles:
Program to a supertype - because the input to **`thirsty()`** is an interface, it can accept any object that implements **`Foo.Bar`**.

Favour composition over inheritance - since **`thirsty()`** can accept any object that implements **`Foo.Bar`**, the objects of the **`Foo`** class become more flexible and its behaviour can change at runtime.

```
public class Foo {

   interface Bar{
       void drink();
   }

   Foo(){
   }

   void thirsty(Bar bar){
       bar.drink();
   }

}
```

Further Reading

- **Nested Classes and Anonymous Classes at Oracle's Java Tutorial**
  - https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html
  - https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html

- ***Bloch, Effective Java,* Item 22.**

The Android Programming You need to know

onCreate is called when the Activity is first launched

Within the **`MainActivity`** class, you would see this code

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
}
```

The **`onCreate`** method is called whenever your Activity is first launched e.g. when the user clicks on your app icon.

This method is part of the methods in the **Android activity life cycle**, which will be discussed in the next lesson.

You write code in **`onCreate`** to implement what you want the user to see when the activity is launched.

The R class contains resource IDs to the resources in the res folder.

When the app is compiled, an **R class** is generated that contains IDs to the resources in the **res** folder.

Since **`activity_main.xml`** is stored in the layout folder, its R class reference is **`R.layout.activity_main`**.

In onCreate, the layout is first inflated

`R.layout.activity_main` is passed to the **`setContentView`** method to **inflate the layout**. In this process, Android reads the XML code in the layout file and instantiates objects in the memory that represent each of the widgets on the Activity.

More examples of Resource IDs

Widget ID

If your widget has the following attribute

```
android:id =”@+id/myWidget”
```

then it can be accessed by **`R.id.myWidget`**.

Images in drawables

If you have an image stored in the drawable folder named **pikachu.png**, then it can be accessed by **`R.drawable.pikachu`**.

Note that File-based resource names

- ***must contain only lowercase a-z, 0-9, or underscore, and***
- ***must start with a letter***

Clicker Question - What type of class is the R class?

The R class contains nested classes. True/False.

Seeing the R class (Optional)

Before Android Studio 3.6

The R class is generated for you but you may view it in your project as follows

- Change to Project View
- Access the folder path: app/build/generated/source/r/debug/<your.package.name>

In general, we don’t actually have to do anything to this file

Android Studio 3.6 and later

The R class is generated and directly compiled into bytecodes. We can’t see it unless we decompile R.jar

Use findViewById() method to assign a widget to a variable

If a widget has an id attribute **`myTextView`**, then the corresponding reference in the R class is **`R.id.myTextView.`**

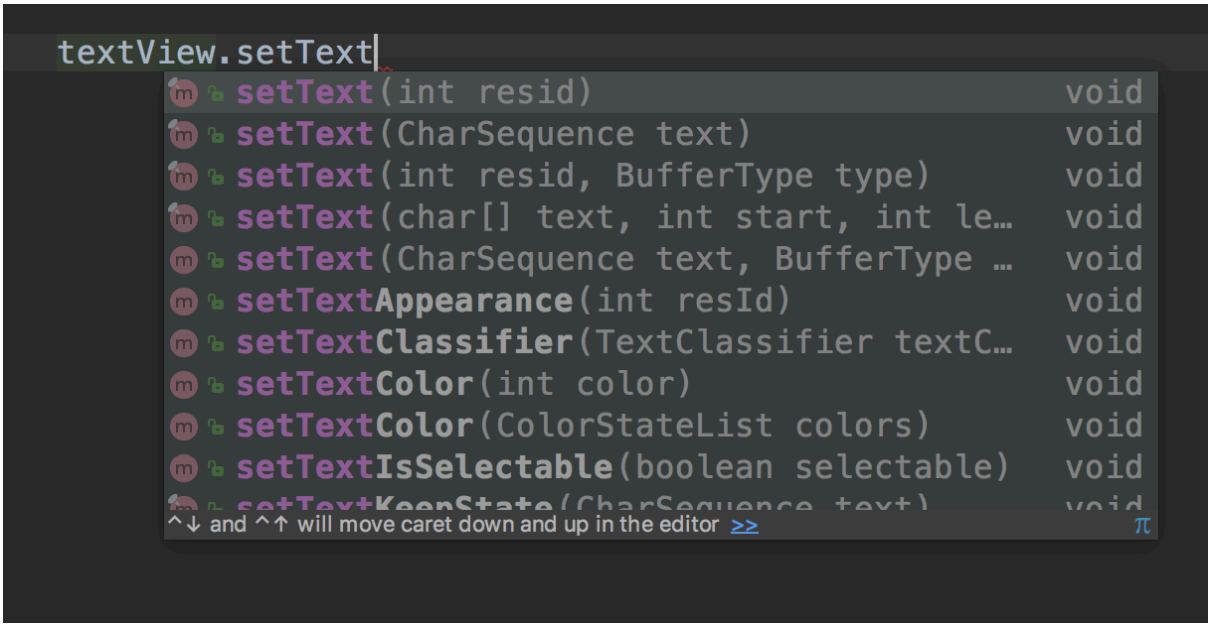
This reference is then passed to **`findViewById()`** , which returns a reference to the widget. This can then be assigned to a variable. A sample code is as follows:

```
TextView textView = findViewById(R.id.myTextView);
```

When typing the code out in Android Studio, press **Alt + Enter (Windows)** or **Option + Enter (Mac)** to import the necessary library.

Use the dot operator to see what methods are available for that widget

Once you have a variable, use the dot operator to see what methods are available. For example, you can see that the setText method is an overloaded method, it can take in another resource ID or a character array.



You can control a widget's properties in Java

Some methods, like the **`setText`** method, enable you to control a widget's properties programmatically in Java. For example:

```
textView.setText("My New String")
```

This action thus replaces what is written in the XML layout file.

When a View object is clicked, what happens next is specified by calling setOnClickListener()

Typically, we want Button to be clicked, but it is also possible to have other widgets clicked, including TextView, LinearLayout etc.

The input to **`setOnClickListener`** is an object of a class that implements the **`View.OnClickListener`** **interface**. There is one method to implement, called **`onClick`**. You may implement this in several ways, here I list three ways:

Choice 1. As an inner class in `MainActivity`.

This method shows you clearly what you are doing. But it may cause your MainActivity.java to become bloated with inner classes that you use only once.

```
public class MainActivity extends AppCompatActivity {

   Button button;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);

       button = findViewById(R.id.myButton1);
       button.setOnClickListener( new ClickMe());
   }

   //*** this is an inner class ***
   class ClickMe implements View.OnClickListener{

       @Override
       public void onClick(View v) {
               //code goes here
       }
   }
}
```

Choice 2 (Recommended). As an anonymous class that is defined in the input to setOnClickListener():

This is the recommended method because it is used very frequently, and in many other situations.

```
public class MainActivity extends AppCompatActivity {

   Button button;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);

       button = findViewById(R.id.myButton1);

       button.setOnClickListener(new View.OnClickListener() {
           @Override
           public void onClick(View v) {
                 //code goes here
           }
       });
   }
}
```

Choice 3. Define an instance method in MainActivity specifying what is to be done. Then specify it as an attribute in the widget.

Although it looks straightforward and easy, I don't recommend this choice, for the following reasons.

- Many code examples used in teaching android use anonymous classes instead
- This does not work in many other situations.

Define an instance method in MainActivity with any name you like and the following signature.

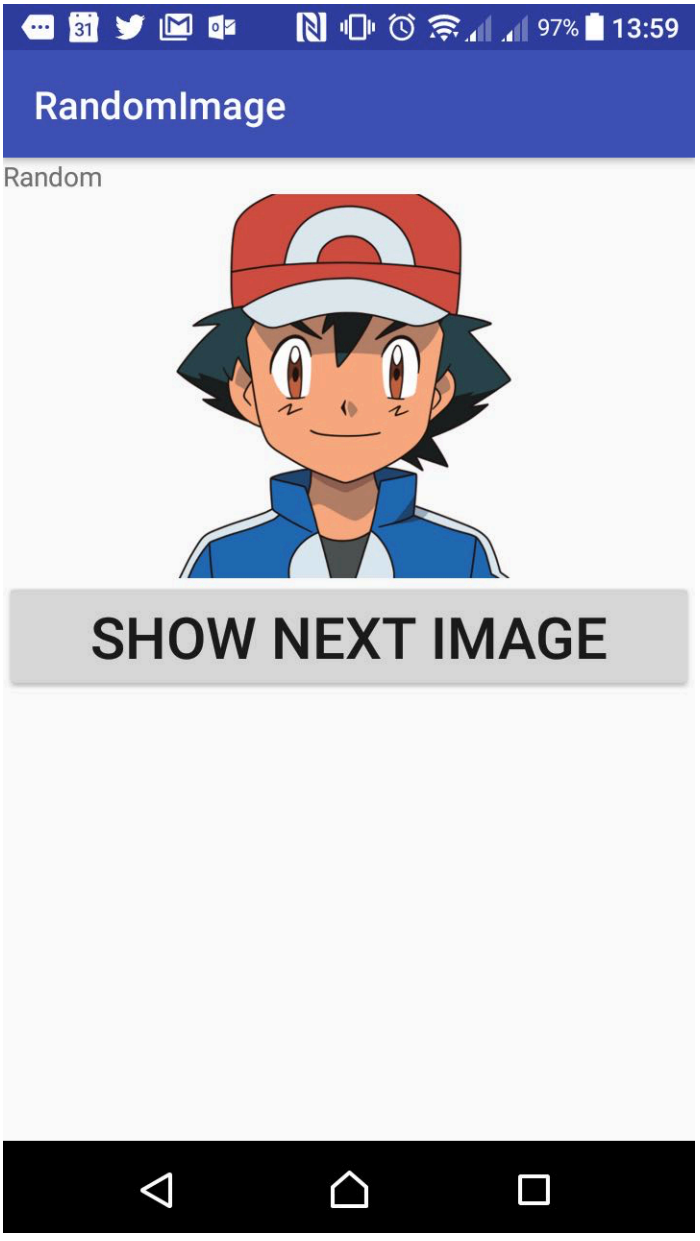
```
public class MainActivity extends AppCompatActivity {

   Button button;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);

       button = findViewById(R.id.myButton1);

   }
   //this method is what myButton1 will do
   public void whenClick(View view){
        //code here
   }
}
```

Then in the XML file, the button widget will have the **onClick** attribute.

```
<Button
   android:id="@+id/myButton1"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:onClick="whenClick"
   android:text="Click Me"/>
```

Making our App

What the app should look like



The strategy with this app:

- Store all images in the res/drawables folder. Put the Image IDs in an ArrayList
- When the Button is clicked, retrieve the image ID from the ArrayList in sequence.
- Use the image ID to retrieve the image and place it in the ImageView widget.

Code Stump For MainActivity.java

Click on the Github classroom link to obtain the starter code for Lesson 1. The starter code for Lesson 1 just provides you with MainActivity.java, plus a folder containing some images.

Copy the code below the **`package`** statement (i.e. do not copy the package statement) and paste it in the **MainActivity.java** in the project that you generated in Lesson 0 (again, below your own **`package`** statement). Import any classes as needed.

This is a screenshot of the starter code.

```
package com.example.myfirstapp;

// Create a new android studio project with Empty Activity
// Copy the code below
// Go to your own MainActivity.java and
// paste it over the existing code BELOW the package statement ***
// ***Sep 2019

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;


//TODO 1.1 Put in some images in the drawables folder
//TODO 1.2 Go to activity_main.xml and modify the layout

public class MainActivity extends AppCompatActivity {

    //TODO 1.2 Instance variables are declared for you, please import the libraries
    ArrayList<Integer> images;
    Button charaButton;
    ImageView charaImage;
    int count = 0;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        //TODO 1.3 Instantiate An ArrayList object
        //TODO 1.4 Add the image IDs to the ArrayList
        //TODO 1.5 Get references to the charaButton and charaImage widgets using findViewById
        //TODO 1.6 For charaButton, invoke the setOnClickListener method
        //TODO 1.7 Create an anonymous class which implements View.OnClickListener interface
        //TODO 1.8 Within onClick, write code to randomly select an image ID from the ArrayList and display it in the ImageView
        //TODO 1.9 [On your own] Create another button, which when clicked, will cause one image to always be displayed


    }
}
```

TODO 1.1 and 1.2 - Images and Layout

After putting the images in the drawables folder, you are ready to modify **`activity_main.xml`**

Layout

- Replace the tag **`ConstraintLayout`** with **`LinearLayout`**
- In the list of attributes, specify that its **`orientation`** is vertical

Widgets

- Put one TextView widget containing the text “Random Images”
    - Give it the id `textViewRandomImages`
- Put one ImageView widget
    - assign any image to it using the **`src`** attribute
    - Give it the id **`charaImage`**

- Put one Button widget
    - Its text shall be “Show Next Image” or anything else you like
    - Give it the id **`charaButton`**

Compile the app and view it on your phone. You should see three widgets.
If you don’t, you may have forgotten to specify that the linear layout is vertical.

TODO 1.3 and 1.4 - Instantiate the ArrayList object and add the image IDs to it

You would have learnt how to instantiate an ArrayList object in week 1.
How to write the image IDs using the R class has also been explained earlier.

```
images = new ArrayList<Integer>();
images.add(R.drawable.ashketchum);
images.add(R.drawable.bartsimpson);
```

TODO 1.5 - Get references to the charaButton and charaImage widgets

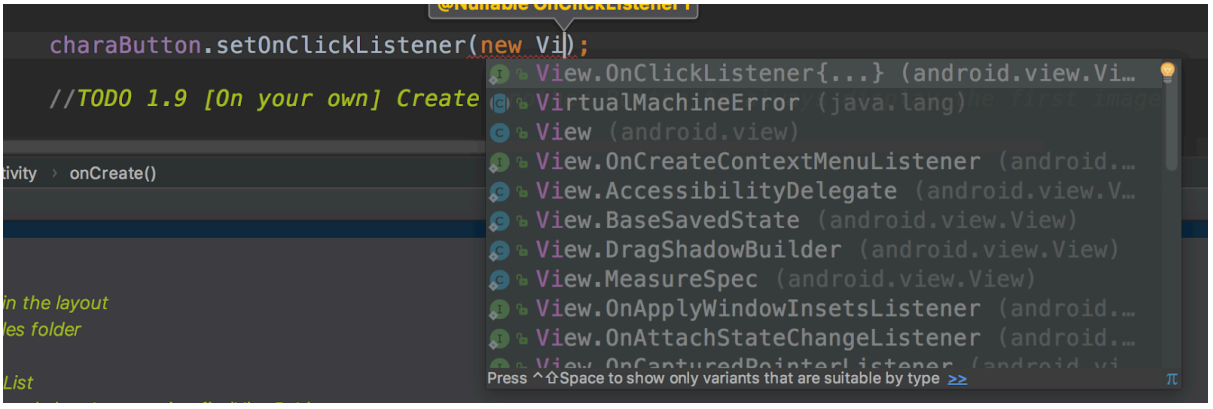
Recall that you use findViewById() and you use the R class to reference the widget IDs.

```
charaButton = findViewById(R.id.charaButton);
charaImage = findViewById(R.id.charaImage);
```

TODO 1.6 & 1.7 For charaButton, Invoke the setOnClickListener() and create an anonymous class which implements View.OnClickListener

After typing out the first few characters, select the first one from the drop-down list.

The code stump is automatically created.



We create an anonymous class instead of the other methods.

This is by far the most common way that I have seen.

TODO 1.8 Write code to randomly select an image ID from the ArrayList and display it in the ImageView

From the code stump generated, within **`onClick`**, write code to retrieve a random element from the ArrayList. The `Random` class will be helpful.
Once you have done so, you assign it to `charaImage` using the `setImageResource()` method.

```
charaImage.setImageResource(id);
```

Solution Code for TODO 1.6 - 1.8

I present the solution where the images are accessed from the array in sequence. Modify it such that it is accessed randomly using the Random class.

```
charaButton.setOnClickListener(

        new View.OnClickListener() {
            @Override
            public void onClick(View view) {

                int index = (count )% images.size();
                count = count + 1;

                int id = images.get(index);
                charaImage.setImageResource(id);
            }
        }
);
```

TODO 1.9 [Try Yourself] Create another Button to always display the first image

To test yourself to see if you have understood what to do, try this.

- Add another button to the layout.
- When this button is clicked, it will display a particular image of your choice.