LAB MANUAL

Course: CSC303 Mobile Application Development

Department of Computer Science

COMSATS University Islamabad, Abbottabad Campus

Table of Contents

Preface

This lab manual is designed for a one-semester course of Mobile Application Development. The prerequisite for students using this manual is a one-semester course in Object Oriented Programing language usually in Java. Before this course, the students also learn object oriented programming concepts in Java language. Java is well suited to learn object oriented software.

We start with the revision of Object Oriented Programming concepts in Java such as Abstraction, Encapsulation, information hiding, Inheritance and Polymorphism as well as environment setup and their understanding. After the fist lab we directly starts with a simple application and introduce the different aspect of creating apps in android.

The lab is then proceeds with GUI programming and touching some advanced UI components. Then we focus on the based Android Architectural elements and then advance the students into different aspects of the Android developments. The last labs usually focus on the emerging trends in mobile application development as well as the cross platform development in the field.

Every lab is design with specific objectives, scope, Lab Activities ranging from simple examples to advanced and exercises such that they help them try the different concepts in Android from different perspectives. The lab also encourages them to utilize the knowledge in their Final Year Projects where applicable.

We would like to thank all the faculty members of Computer Science Department, COMSATS Institute of Information Technology Abbottabad for their helpful comments and suggestions. Mr. Mukhtiar Zamin, Mr. Ibtisam Gul and Madam Bushra prepared its first draft version and then its quality was improved with valuable suggestions, comments via a systematic review process by Dr. Osman Khalid. I would like to thank all of them for their contribution and support. I am also thankful to Mr. Tariq Baloch for his valuable suggestions in compellation of the manual.

LAB 01: Environment setup & understanding

Objective

In this practical you learn how to install Android Studio, understanding the Android development environment. Creation of first project in Android studio to print a welcome message. Familiarize with different components of android studio IDE and their role in order to develop an android app.

Scope

The scope of this lab activity is the student's ability to:

- How to install Android studio development environment
- Explore the different components of Android Studio IDE.
- Creation of First Project in Android Studio
- Understanding the role of relevant components for the development of an android app.
- Explore the Android project Layout.

Useful Concepts

Step-1: Install Android Studio

Android Studio provides a complete integrated development environment (IDE) including an advanced code editor and a set of app templates. In addition, it contains tools for development, debugging, testing, and performance that make it faster and easier to develop apps. You can test your apps with a large range of preconfigured emulators or on your own mobile device, build production apps, and publish on the Google Play store.

Android Studio is available for computers running Windows or Linux, and for Macs running macOS. The newest OpenJDK (Java Development Kit) is bundled with Android Studio.

To get up and running with Android Studio, first check the system requirements (https://developer.android.com/studio/index.html#Requirements) to ensure that your system meets them. The installation is similar for all platforms. Any differences are noted below.

1. Navigate to the Android developers site (https://developer.android.com/studio)and follow the instructions to download and install Android Studio(https://developer.android.com/studio/install.html).
2. Accept the default configurations for all steps, and ensure that all components are selected for installation.

3. After finishing the install, the Setup Wizard will download and install some additional components including the Android SDK. Be patient, this might take some time depending on your Internet speed, and some of the steps may seem redundant.
4. When the download completes, Android Studio will start, and you are ready to create your first project.

Step-2: Understanding Android Studio Environment

Task 2: Create the Welcome-in-Android app

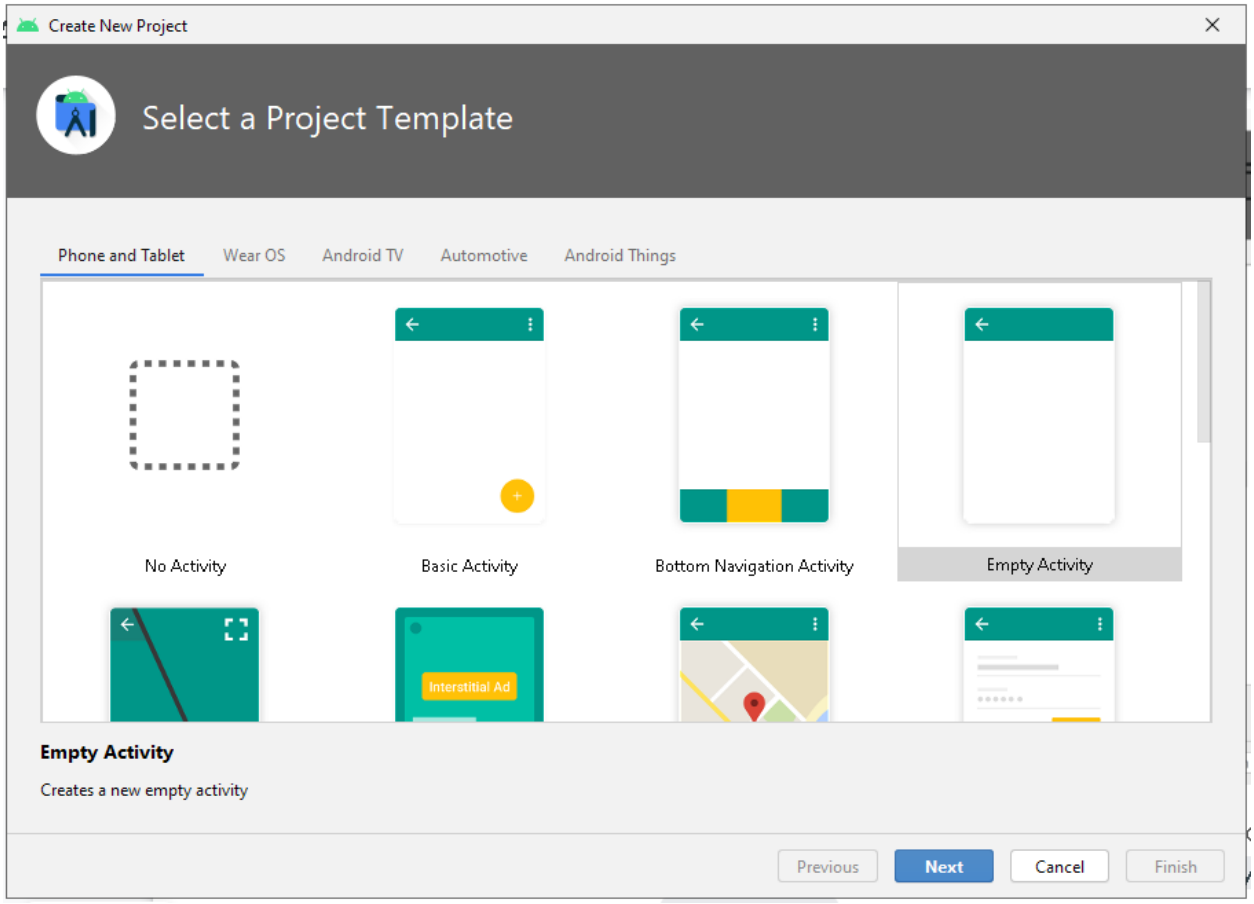
In this task, you will create an app that displays "Welcome in the world of Android Development " to verify that Android studio is correctly installed, and to learn the basics of developing with Android Studio.

2.1 Create the app project

1. Open Android Studio if it is not already opened.

2. In the main **Welcome to Android Studio** window, click **Start a new Android Studio project**.

3. Choose the project template as **Empty Activity**. An Activity is a single, focused thing that the user can do. It is a crucial component of any Android app. An Activity typically has a layout associated with it that defines how UI elements appear on a screen.

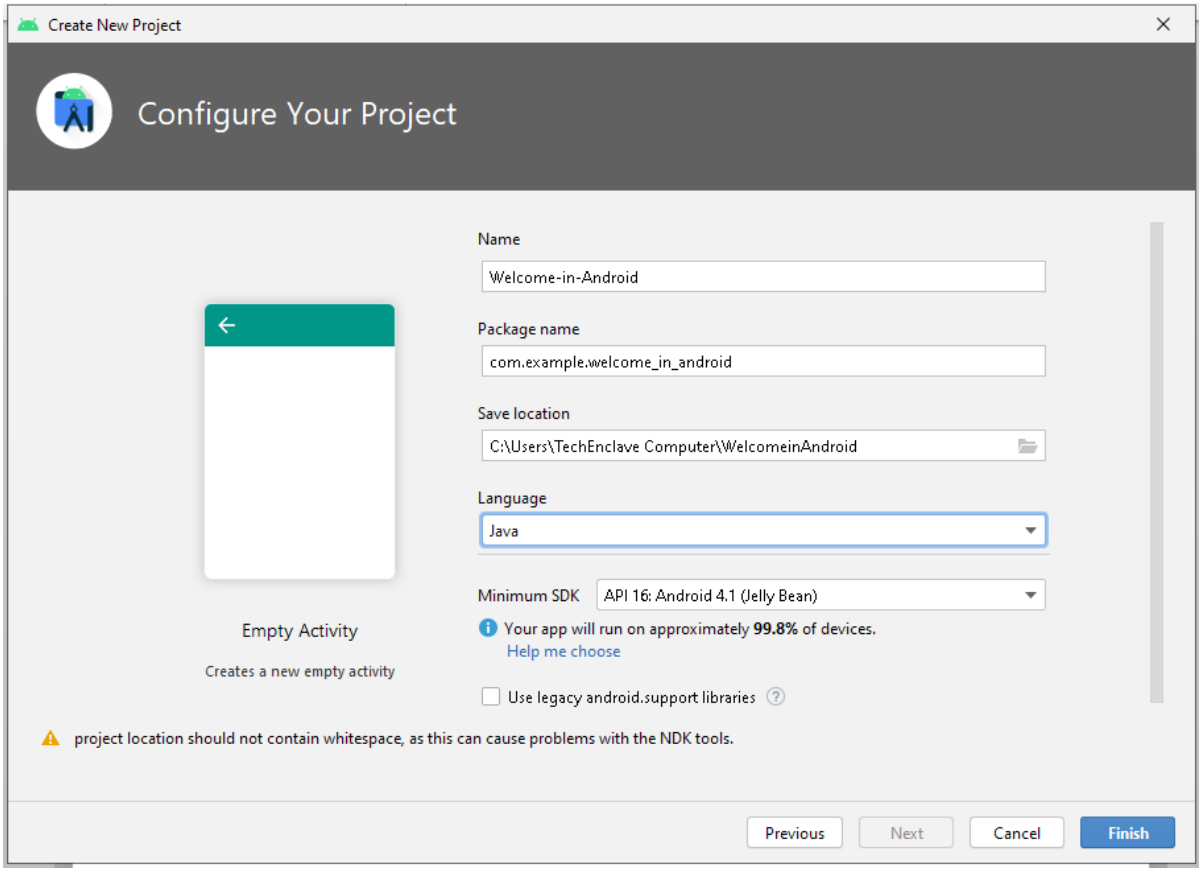


1.

Android Studio provides Activity templates to help you get started. For the Welcome-in-

Android project, choose **Empty Activity**, and click **Next**.

4. In the **Configure Your Project** window, enter **Welcome-in-Android** for the **Application name**.



5. Verify that the default **Project location** is where you want to store your Welcome-in-Android app and other Android Studio projects, or change it to your preferred directory.

6. Accept the default **com.example** for **Company Domain**, or create a unique company domain.

If you are not planning to publish your app, you can accept the default. Be aware that changing the package name of your app later is extra work.

7. Note that Minimum SDK is set to API **16: Android 4.1 (Jelly Bean)** is set as the Minimum SDK; if it is not, use the popup menu to set it. These are the settings for Android Studio version 4.1 to make our Welcome-in-Android app compatible with 99.8% of Android devices active on the Google Play Store.

8. Leave unchecked the options to IUse legacy android support libraries,

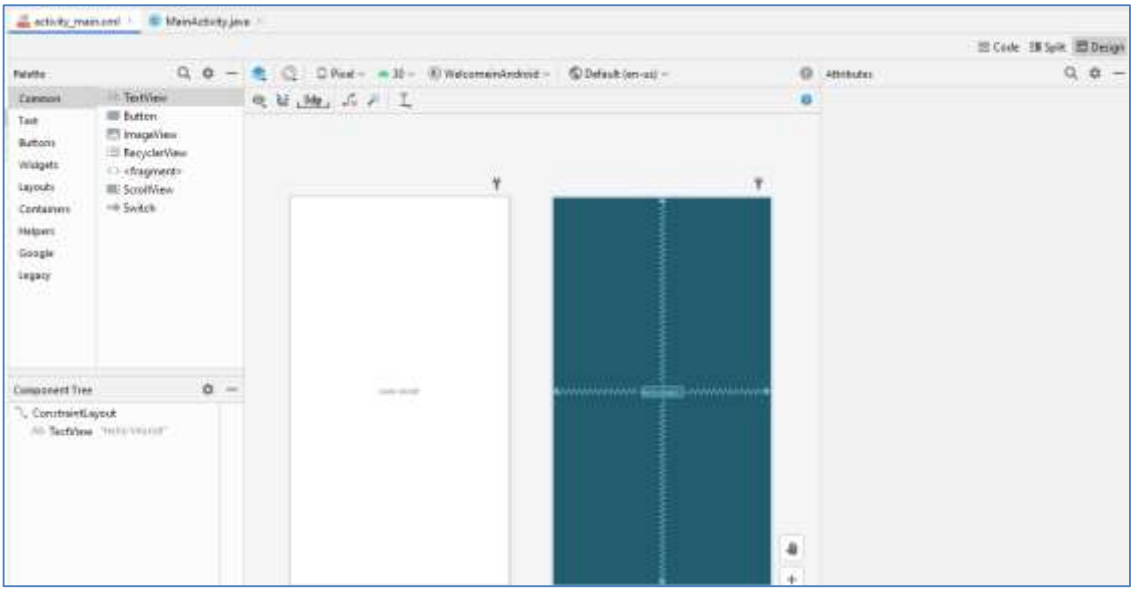
9. click Finish.

Android Studio creates a folder for your projects, and builds the project with Gradle (this may take a few moments).

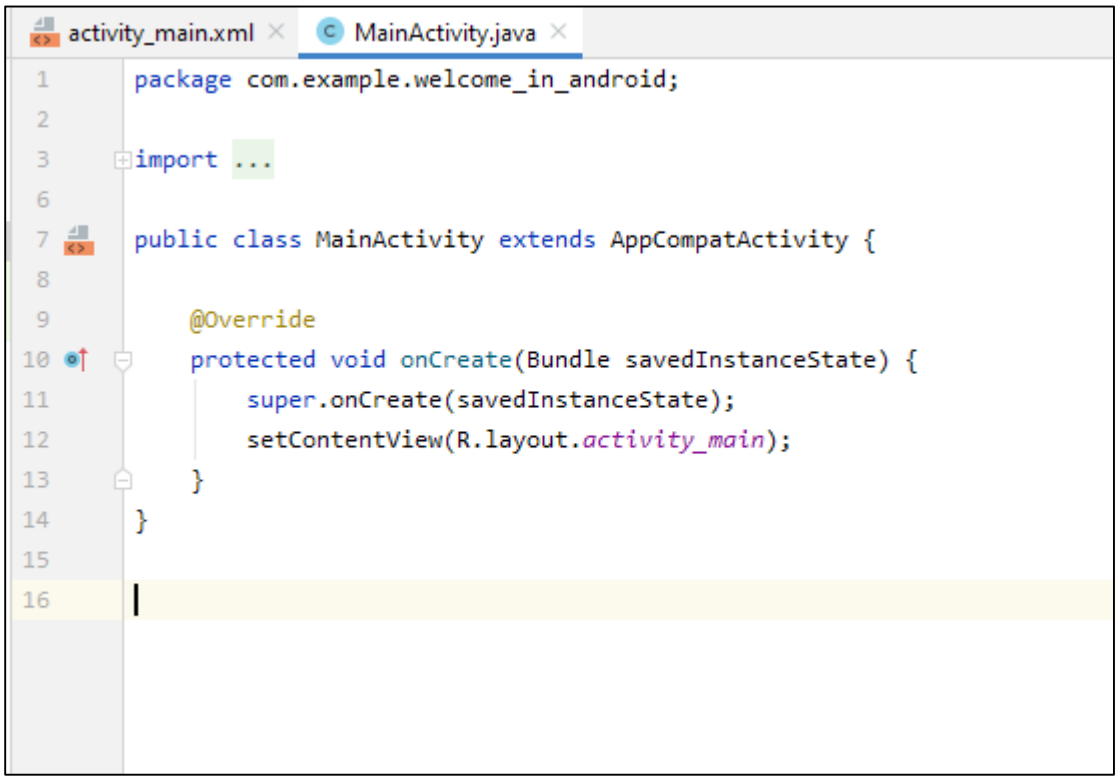
The Android Studio editor appears. Follow these steps:

1. Click the **activity_main.xml** tab to see the layout editor.

2. Click the layout editor **Design** tab, if not already selected, to show a graphical rendition of the layout as shown below.



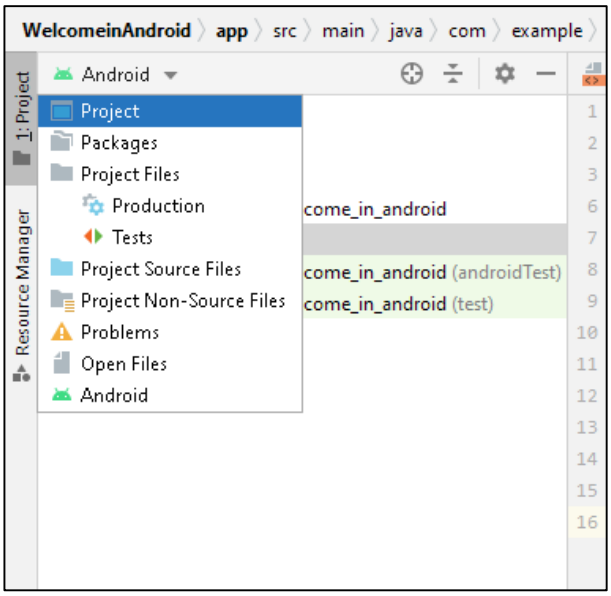
3. Click the **MainActivity.java** tab to see the code editor as shown below.



2.2 Explore the Project > Android pane

In this practical, you will explore how the project is organized in Android Studio.

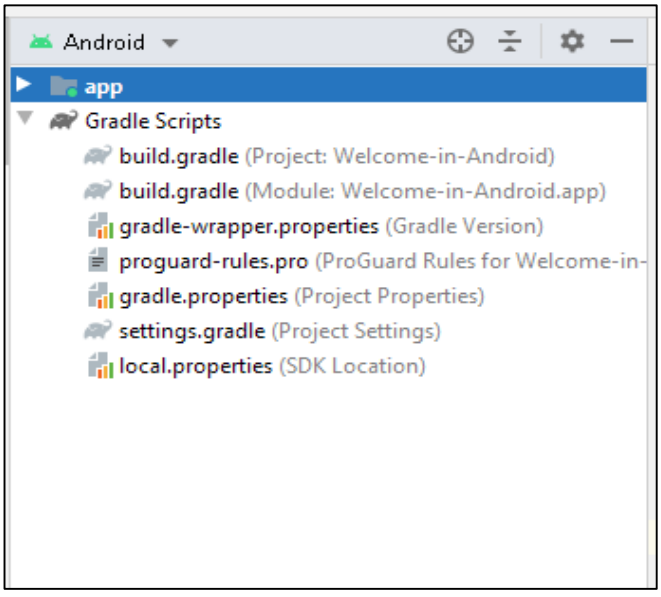
1. If not already selected, click the **Project** tab in the vertical tab column on the left side of the Android Studio window. The Project pane appears.
2. To view the project in the standard Android project hierarchy, choose **Android** from the popup menu at the top of the Project pane, as shown below.



2.3 Explore the Gradle Scripts folder

The Gradle build system in Android Studio makes it easy to include external binaries or other library modules to your build as dependencies.

When you first create an app project, the **Project > Android** pane appears with the **Gradle Scripts** folder expanded as shown below.



Follow these steps to explore the Gradle system:

1. If the **Gradle Scripts** folder is not expanded, click the triangle to expand it.

   This folder contains all the files needed by the build system.

2. Look for the **build.gradle(Project: Welcome-in-Android)** file.

   This is where you'll find the configuration options that are common to all of the modules that make up your project. Every Android Studio project contains a single, top-level Gradle build file. Most of the time, you won't need to make any changes to this file, but it's still useful to understand its contents.

   By default, the top-level build file uses the buildscript block to define the Gradle repositories and dependencies that are common to all modules in the project. When your dependency is something other than a local library or file tree, Gradle looks for the files in whichever online repositories are specified in the repositories block of this file. By default, new Android Studio projects declare JCenter and Google (which includes the Google Maven repository) as the repository locations:

```
allprojects {
    repositories {
        google()
        jcenter()
    }
}
```

3. Look for the **build.gradle(Module:app)** file.

In addition to the project-level build.gradle file, each module has a build.gradle file of its own, which allows you to configure build settings for each specific module (the Welcome-in-Android app has only one module). Configuring these build settings allows you to provide custom packaging options, such as additional build types and product flavors. You can also override settings in the AndroidManifest.xml file or the top-level build.gradle file.

This file is most often the file to edit when changing app-level configurations, such as declaring dependencies in the dependencies section. You can declare a library dependency using one of several different dependency configurations. Each dependency configuration provides Gradle different instructions about how to use the library. For example, the statement implementation fileTree(dir: 'libs', include: ['*.jar']) adds a dependency of all ".jar" files inside the libs directory.

The following is the **build.gradle(Module:app)** file for the Welcome-in-Android app:

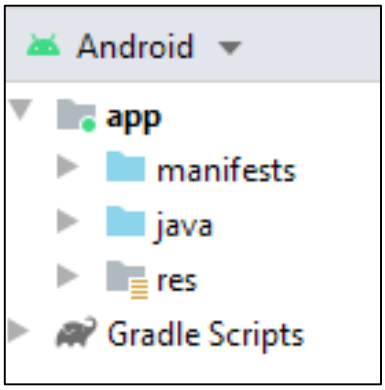
```
plugins {
    id 'com.android.application'
}

android {
    compileSdkVersion 30
    buildToolsVersion "30.0.2"
```

```
    defaultConfig {
        applicationId "com.example.welcome_in_android"
        minSdkVersion 16
        targetSdkVersion 30
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
'proguard-rules.pro'
        }
    }
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
}

dependencies {

    implementation 'androidx.appcompat:appcompat:1.2.0'
    implementation 'com.google.android.material:material:1.2.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.0.4'
    testImplementation 'junit:junit:4.+'
    androidTestImplementation 'androidx.test.ext:junit:1.1.2'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'
}
```

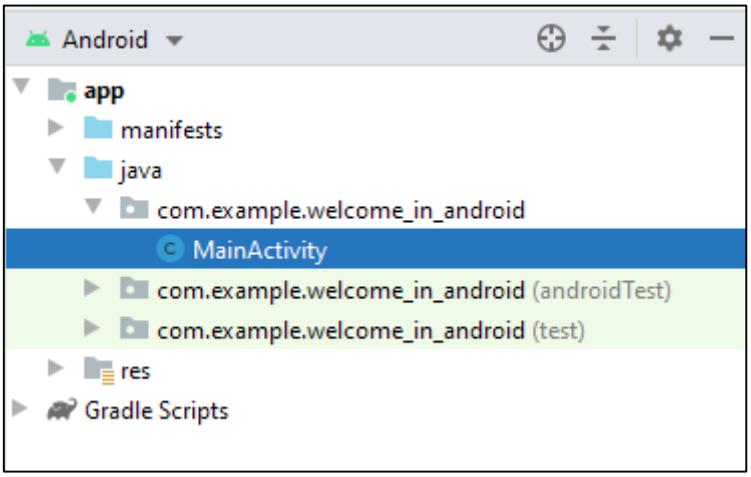
4. Click the triangle to close **Gradle Scripts**.

2.4 Explore the app and res folders

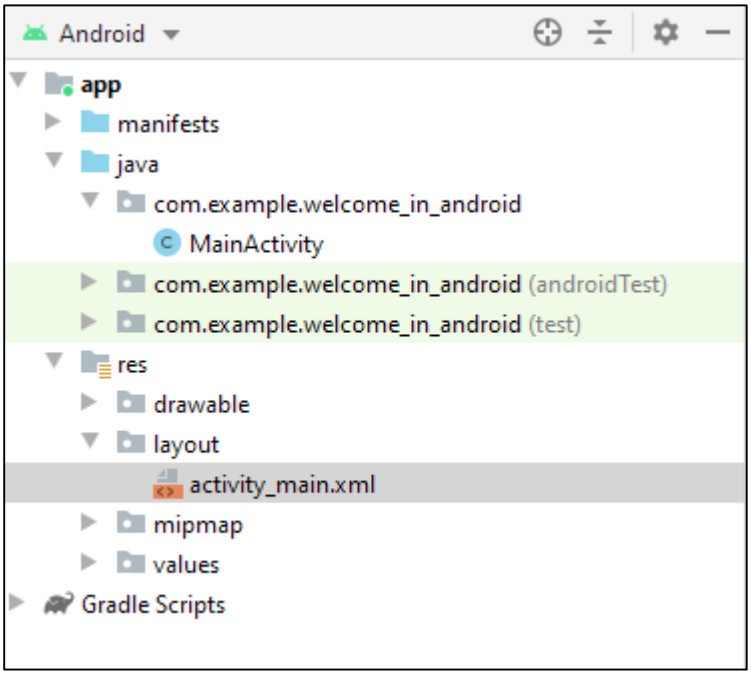
All code and resources for the app are located within the app and res folders.



1. Expand the **app** folder, the **java** folder, and the **com.example.welcome_in_android** folder to see the **MainActivity** java file. Double-clicking the file opens it in the code editor.



The **java** folder includes Java class files in three subfolders, as shown in the figure above. The **com.example.welcome_in_android** (or the domain name you have specified) folder contains all the files for an app package. The other two folders are used for testing and described in another lesson. For the Welcome-in-Android app, there is only one package and it contains MainActivity.java. The name of the first Activity (screen) the user sees, which also initializes app-wide resources, is customarily called **MainActivity** (the file extension is omitted in the **Project > Android** pane).
Expand the **res** folder and the **layout** folder and double-click the **activity_main.xml** file to open it in the layout editor.



The **res** folder holds resources, such as layouts, strings, and images. An Activity is usually associated with a layout of UI views defined as an XML file. This file is usually named after its Activity.

2.5 Explore the manifests folder

The manifests folder contains files that provide essential information about your app to the Android system, which the system must have before it can run any of the app's code.

1. Expand the **manifests** folder.
2. Open the **AndroidManifest.xml** file.

The AndroidManifest.xml file describes all of the components of your Android app. All components for an app, such as each Activity, must be declared in this XML file. In other course lessons you will modify this file to add features and feature permissions.

Task 3: Use a virtual device (emulator)

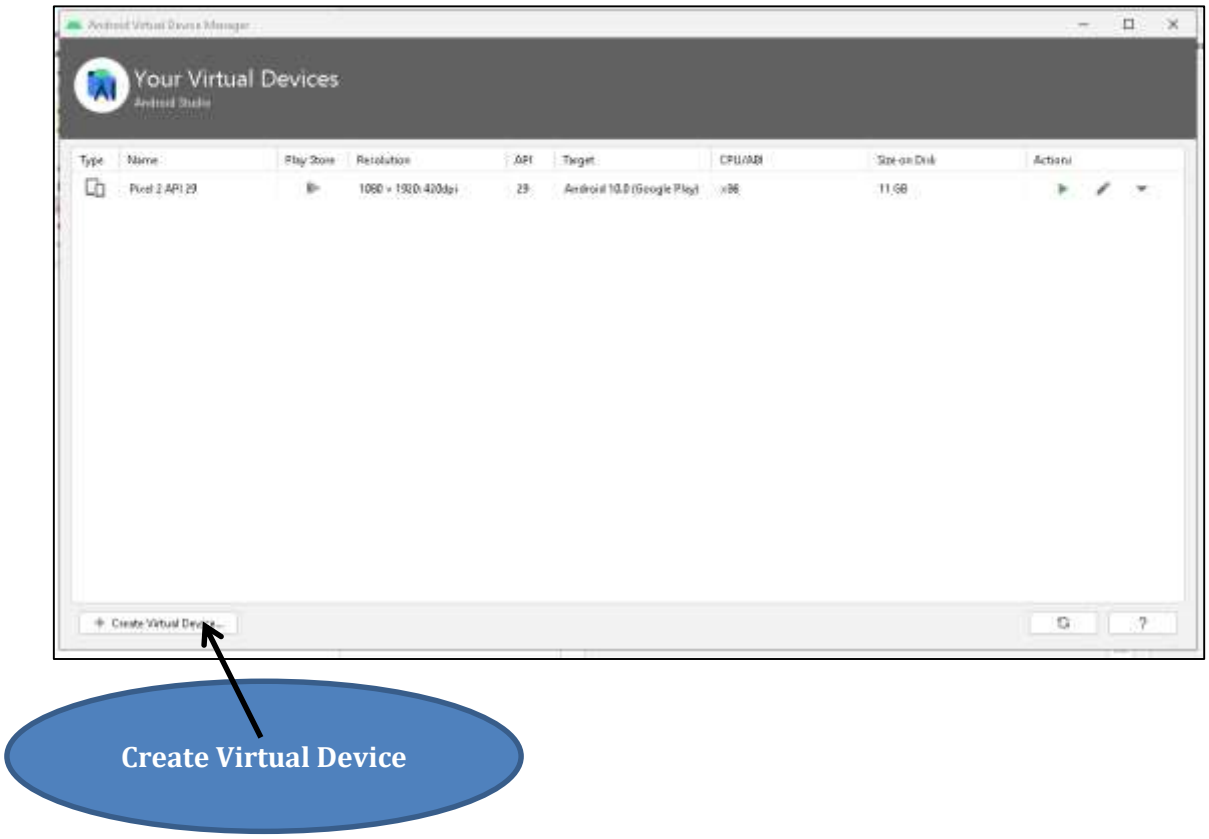
In this task, you will use the Android Virtual Device (AVD) manager to create a virtual device (also known as an emulator) that simulates the configuration for a particular type of Android device, and use that virtual device to run the app. Note that the Android Emulator has additional requirements beyond the basic system requirements for Android Studio.

Using the AVD Manager, you define the hardware characteristics of a device, its API level, storage, skin and other properties and save it as a virtual device. With virtual devices, you can test apps on different device configurations (such as tablets and phones) with different API levels, without having to use physical devices.

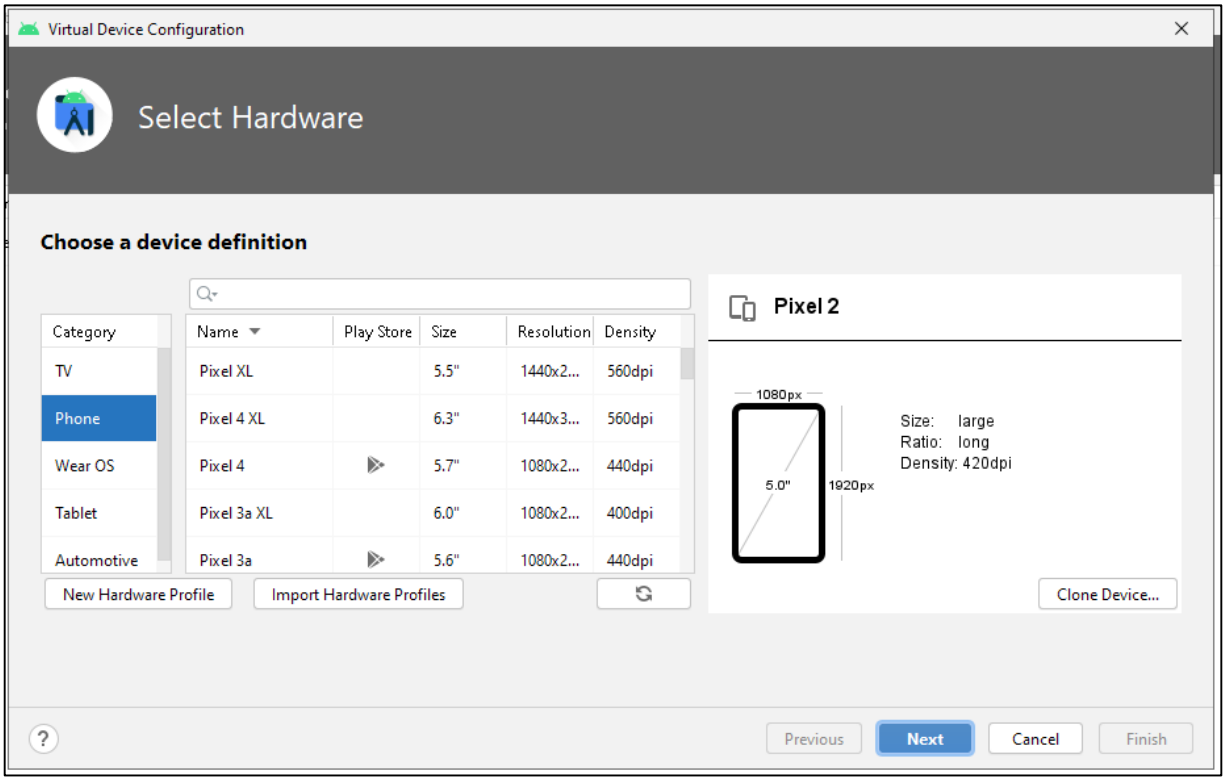
3.1 Create an Android virtual device (AVD)

In order to run an emulator on your computer, you have to create a configuration that describes the virtual device.

1. In Android Studio, select **Tools** > **AVD Manager**, or click the AVD Manager icon  in the toolbar. The **Your Virtual Devices** screen appears. If you've already created virtual devices, the screen shows them (as shown in the figure below); otherwise you see a blank list.

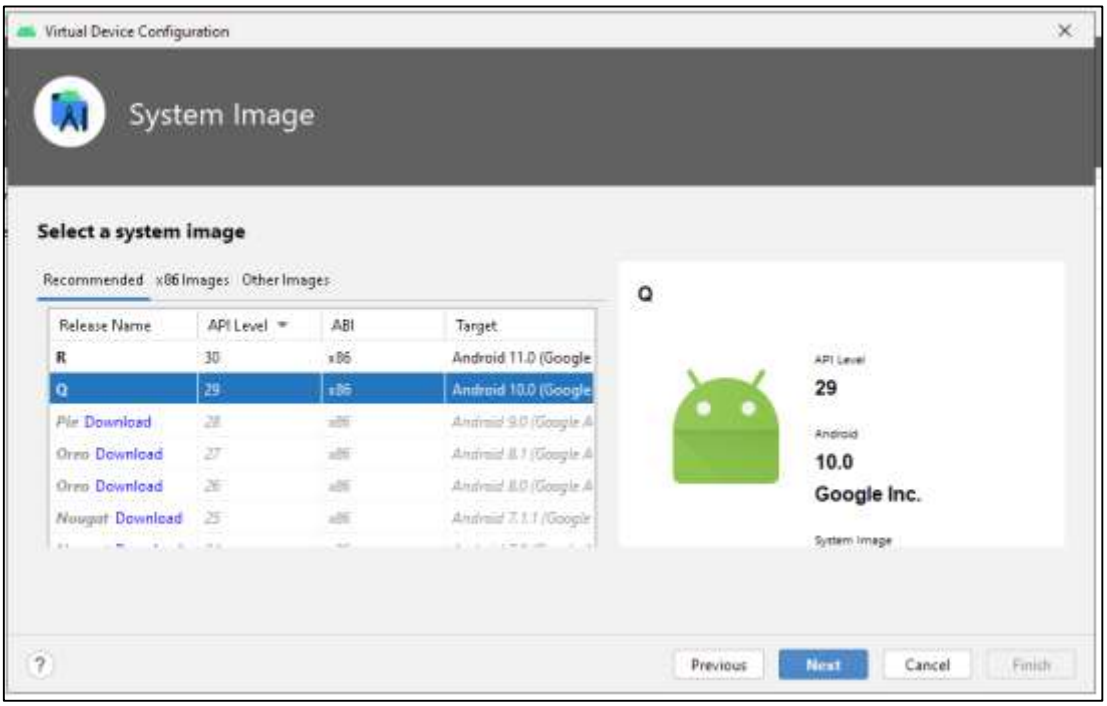


2. Click the +Create Virtual Device. The Select Hardware window appears showing a list of pre configured hardware devices. For each device, the table provides a column for its diagonal display size (Size), screen resolution in pixels (Resolution), and pixel density (Density).



3. Choose a device such as Nexus 5x or Pixel XL, and click Next. The System Image screen appears.

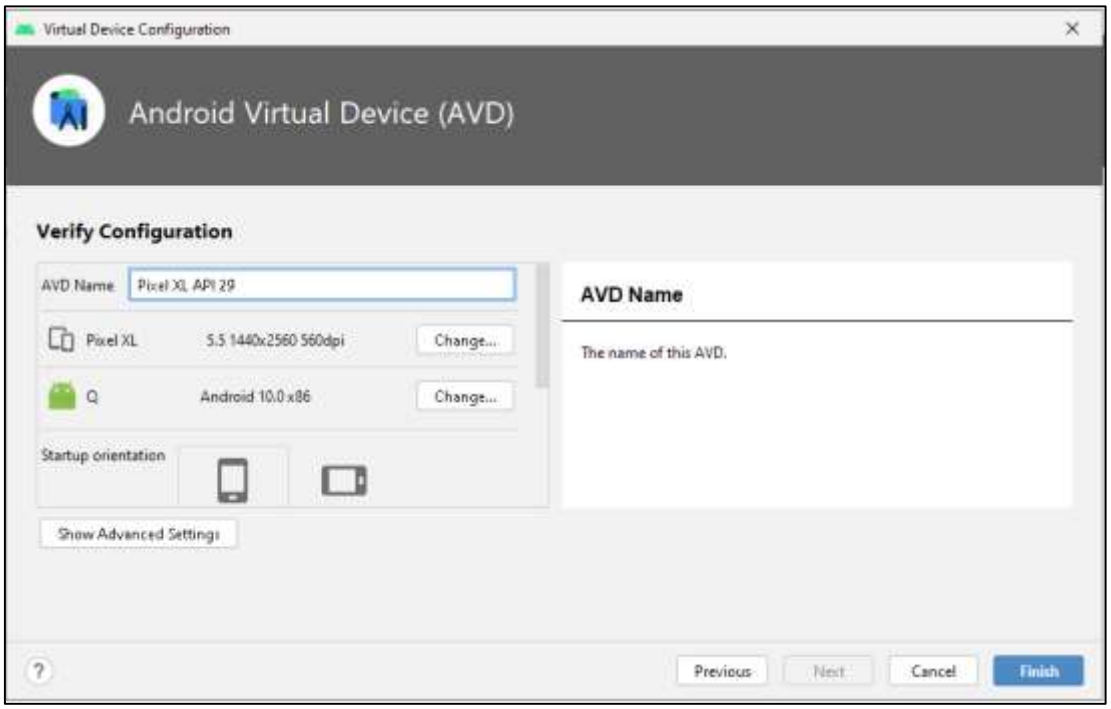
4. Click the Recommended tab if it is not already selected, and choose which version of the Android system to run on the virtual device (such as Oreo).



5. There are many more versions available than shown in the Recommended tab. Look at the x86 Images and Other Images tabs to see them.

6. If a Download link is visible next to a system image you want to use, it is not installed yet. Click the link to start the download, and click Finish when it's done.

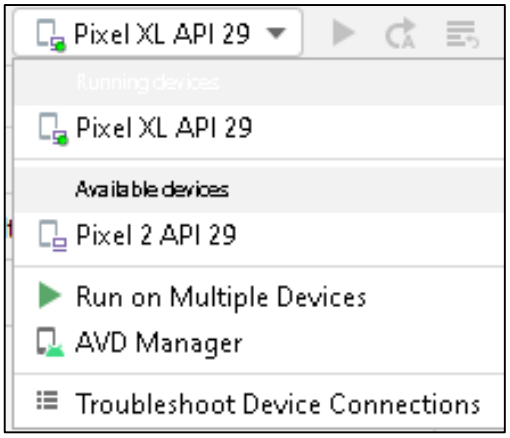
7. After choosing a system image, click Next. The Android Virtual Device (AVD) window appears. You can also change the name of the AVD. Check your configuration and click Finish.



3.2 Run the app on the virtual device

In this task, you will finally run your Welcome-in-Android app.

1. In Android Studio, choose **Run > Run app** or click the **Run** icon  in the toolbar.
2. The **Select Deployment Target** window, under **Available Virtual Devices**, select the virtual device, which you just created, and click **OK**



The emulator starts and boots just like a physical device. Depending on the speed of your computer, this may take a while. Your app builds, and once the emulator is ready, Android Studio will upload the app to the emulator and run it.

You should see the Welcome-in-Android app as shown in the following figure.

Tip: When testing on a virtual device, it is a good practice to start it up once, at the very beginning of your session. You should not close it until you are done testing your app, so that your app doesn't have to go through the device startup process again. To close the virtual device, click the **X** button at the top of the emulator, choose **Quit** from the menu, or press **Control-Q** in Windows or **Command-Q** in macOS.

Task 4: (Optional) Use a physical device

In this final task, you will run your app on a physical mobile device such as a phone or tablet. You should always test your apps on both virtual and physical devices.

What you need:

- An Android device such as a phone or tablet.
- A data cable to connect your Android device to your computer via the USB port.
- If you are using a Linux or Windows system, you may need to perform additional steps to run on a hardware device. Check the Using Hardware Devices http://developer.android.com/tools/device.html documentation. You may also need to install the appropriate USB driver for your device. For Windows-based USB drivers, see OEM USB Drivers https://developer.android.com/studio/run/oem-usb .

4.1 Turn on USB debugging

To let Android Studio communicate with your device, you must turn on USB Debugging on your Android device. This is enabled in the **Developer options** settings of your device. On Android 4.2 and higher, the **Developer options** screen is hidden by default. To show developer options and enable USB Debugging:

1. On your device, open **Settings**, search for **About phone**, click on **About phone**, and tap **Build number** seven times.
2. Return to the previous screen (**Settings / System**). **Developer options** appears in the list. Tap **Developer options**.
3. Choose **USB Debugging**

4.2 Run your app on a device

Now you can connect your device and run the app from Android Studio.

1. Connect your device to your development machine with a USB cable.
2. Click the **Run** button  in the toolbar. The **Select Deployment Target** window opens with the list of available emulators and connected devices.

3.Select your device, and click **OK**.Android Studio installs and runs the app on your device.

Troubleshooting

If your Android Studio does not recognize your device, try the following:

1. Unplug and replug your device.
2. Restart Android Studio.

If your computer still does not find the device or declares it "unauthorized", follow these steps:

1. Unplug the device.
2. On the device, open **Developer Options in Settings app**.
3. Tap Revoke **USB Debugging** authorizations.
4. Reconnect the device to your computer.
5. When prompted, grant authorizations.

You may need to install the appropriate USB driver for your device. See the Using Hardware Devices documentation

https://developer.android.com/studio/run/device .

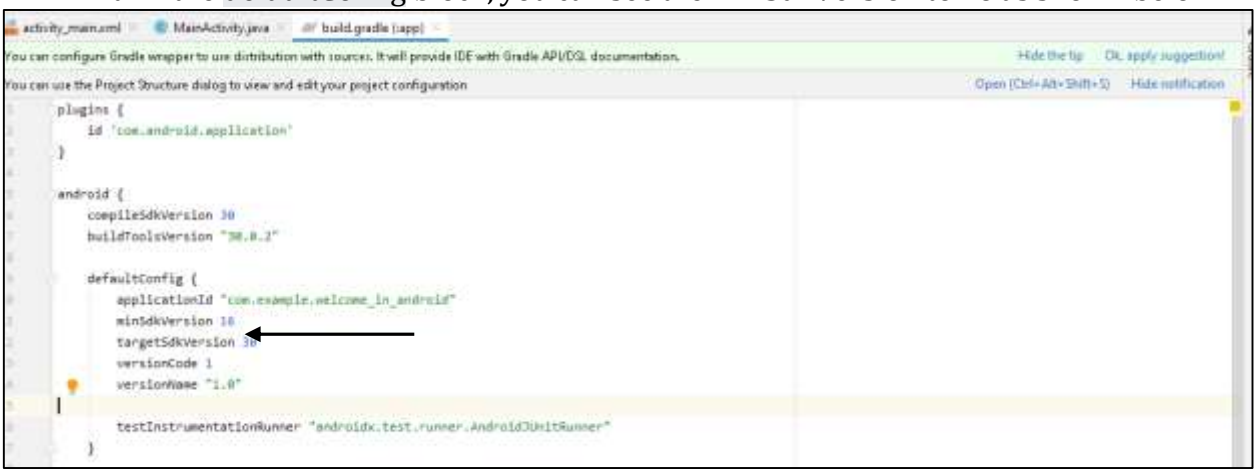
Task 5: Change the app Gradle configuration

In this task you will change something about the app configuration in the `build.gradle(Module:app)` file in order to learn how to make changes and synchronize them to your Android Studio project.

5.1 Change the minimum SDK version for the app

Follow these steps:

1. Expand the **Gradle Scripts** folder if it is not already open, and double-click the **build.gradle(Module:app)** file.
The content of the file appears in the code editor.
2. Within the `defaultConfig` block, you can see the minSdkVersion to 16 as shown below:



3. change the value of `minSdkVersion` to `17` as shown below (it was originally set to `16`).

The code editor shows a notification bar at the top with the Sync Now link

5.2 Sync the new Gradle configuration

When you make changes to the build configuration files in a project, Android Studio requires that you *sync* the project files so that it can import the build configuration changes and run some checks to make sure the configuration won't create build errors.

To sync the project files, click **Sync Now** in the notification bar that appears when making a change (as shown in the previous figure), or click the **Sync Project with Gradle Files** icon  in the toolbar.

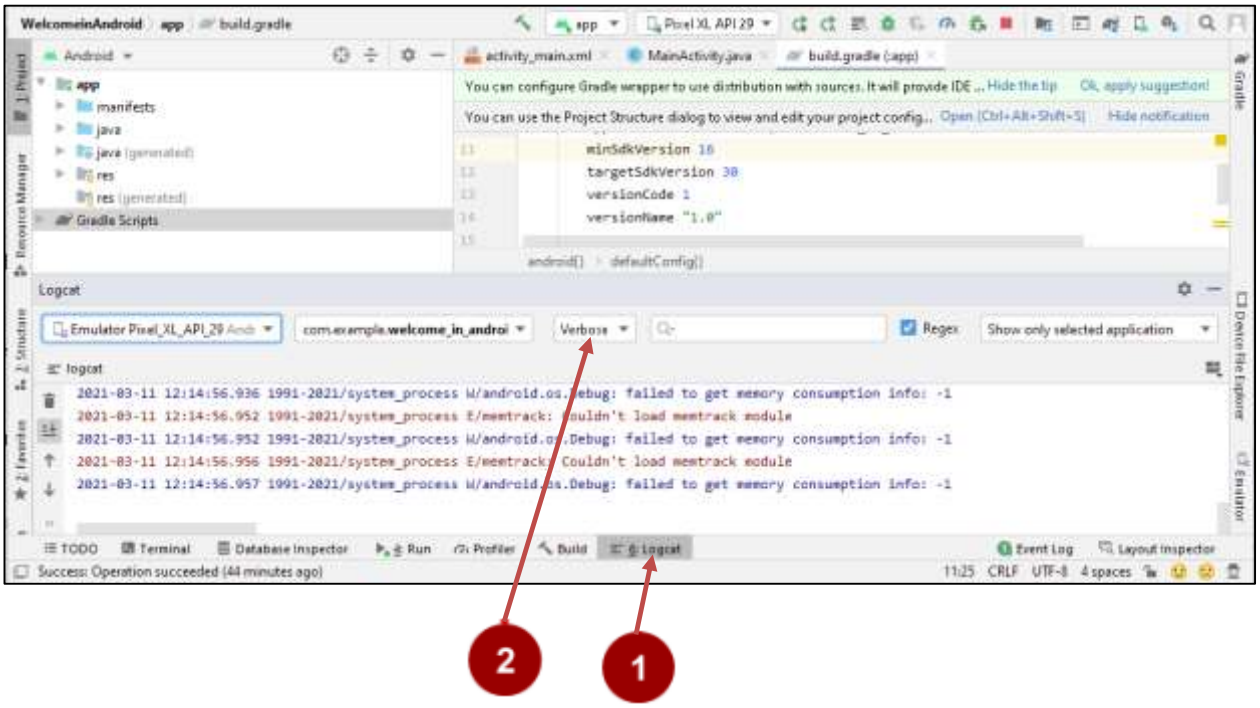
When the Gradle synchronization is finished, the message Gradle build finished appears in the bottom left corner of the Android Studio window.

Task 6: Add log statements to your app

In this task, you will add Log statements to your app, which display messages in the **Logcat** pane. Log messages are a powerful debugging tool that you can use to check on values, execution paths, and report exceptions.

6.1 View the Logcat pane

To see the **Logcat** pane, click the **Logcat** tab at the bottom of the Android Studio window as shown in the figure below.



In the figure above:

1. The **Logcat** tab for opening and closing the **Logcat** pane, which displays information about your app as it is running. If you add Log statements to your app, Log messages appear here.

2. The Log level menu set to **Verbose** (the default), which shows all Log messages. Other settings include **Debug**, **Error**, **Info**, and **Warn**.

6.2 Add log statements to your app

Log statements in your app code display messages in the Logcat pane. For example:

```
Log.d("MainActivity", "Welcome-in-Android World");
```

The parts of the message are:

- Log: The Log class for sending log messages to the Logcat pane.
- d: The **Debug** Log level setting to filter log message display in the Logcat pane. Other log levels are e for **Error**, w for **Warn**, and i for **Info**.
- "MainActivity": The first argument is a tag which can be used to filter messages in the Logcat pane. This is commonly the name of the Activity from which the message originates. However, you can make this anything that is useful to you for debugging.

By convention, log tags are defined as constants for the Activity:

```
private static final String LOG_TAG = MainActivity.class.getSimpleName();
```

- " Welcome-in-Android World ": The second argument is the actual message.

Follow these steps:

1. Open your Welcome-in-Android app in Android studio, and open MainActivity.
2. To add unambiguous imports automatically to your project (such as android.util.Log required for using Log), choose **File > Settings** in Windows, or **Android Studio > Preferences** in macOS.
3. Choose **Editor > General >Auto Import**. Select all checkboxes and set **Insert imports on paste** to **All**.
4. Click **Apply** and then click **OK**.
5. In the onCreate() method of MainActivity, add the following statement:

```
Log.d("MainActivity", "Welcome in Android World");
```

The onCreate() method should now look like the following code:

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        //my code
        Log.d( tag: "MainActivity",  msg: "Welcome in Android World");
    }
}
```

6. If the Logcat pane is not already open, click the **Logcat** tab at the bottom of Android Studio to open it.
7. Check that the name of the target and package name of the app are correct.

8. Change the Log level in the **Logcat** pane to **Debug** (or leave as **Verbose** since there are so few log messages).

9. Run your app.

The following message should appear in the Logcat pane:

2021-09-23 12:45:35.696 10688-10688/com.example.lab_21sep2021 D/MainActivity: Welcome in Android World

Lab Tasks

Activity 1

1. Create a new project in Android Studio.

2. Change the "Hello World" greeting to "Happy Birthday to " and the name of someone with a recent birthday.

Note: In case of physical labs this activity is optional but recommended.

Activity 2

A common use of the Log class is to log Java exceptions when they occur in your program. There are some useful methods, such as Log.e(), that you can use for this purpose. Explore methods you can use to include an exception with a Log message. Then, write code in your app to trigger and log an exception.

Exercises

- Create a new Android project from the Empty Template.
- Add logging statements for various log levels in onCreate() in the main activity.
- Create an emulator for a device, targeting any version of Android you like, and run the app.
- Use filtering in **Logcat** to find your log statements and adjust the levels to only display debug or error logging statements.

LAB 02: Basic Application Development

Objective

In this practical you learn how to create your first interactive app—an app that enables user interaction. You create an app using the Empty Activity template. You also learn how to use the layout editor to design a layout, and how to edit the layout in XML

- Create an app with interactive behavior.
- Use the layout editor to design a layout.
- Edit the layout in XML.

App overview:

The Lab2_Toast_Test app consists of two Button elements and one TextView. When the user taps the first Button, it displays a short message (a Toast) on the screen. Tapping the second Button increases a "click" counter displayed in the TextView, which starts at zero.

Here's what the finished app looks like:

Scope

Dear Students, In this practical you will learn the following concepts.

- Create an app and add two **Button** elements and a **TextView** to the layout.
- Manipulate each element in the **ConstraintLayout** to constrain them to the margins and other elements.
- Change UI element attributes.
- Edit the app's layout in XML.
- Extract hardcoded strings into string resources.
- Implement click-handler methods to display messages on the screen when the user taps each Button.

Useful Concepts

The user interface (UI) that appears on a screen of an Android device consists of a hierarchy of objects called *views* — every element of the screen is a View. The View class represents the basic building block for all UI components, and the base class for classes that provide interactive UI components such as buttons, checkboxes, and text entry fields. Commonly used View subclasses described over several lessons include:

- TextView for displaying text.
- EditText to enable the user to enter and edit text.
- Button and other clickable elements (such as RadioButton, CheckBox, and Spinner) to provide interactive behavior.
- ScrollView and RecyclerView to display scrollable items.
- ImageView for displaying images.
- ConstraintLayout and LinearLayout for containing other View elements and positioning them.

The Java code that displays and drives the UI is contained in a class that extends Activity. An Activity is usually associated with a layout of UI views defined as an XML (eXtended Markup Language) file. This XML file is usually named after its Activity and defines the layout of View elements on the screen.

For example, the MainActivity code in the Lab01 app displays a layout defined in the activity_main.xml layout file, which includes a TextView with the text "Hello World".

In more complex apps, an Activity might implement actions to respond to user taps, draw graphical content, or request data from a database or the internet. You learn more about the Activity class in another lab.

Lab Tasks

In this practical you will do the following sequence of activities for the creation of first interactive app as described in app overview:

Activity 1 : Create and explore a new project

In this practical, you design and implement a project for the HelloToast app.

Create the Android Studio project

1. Start Android Studio and create a new project with the following parameters:

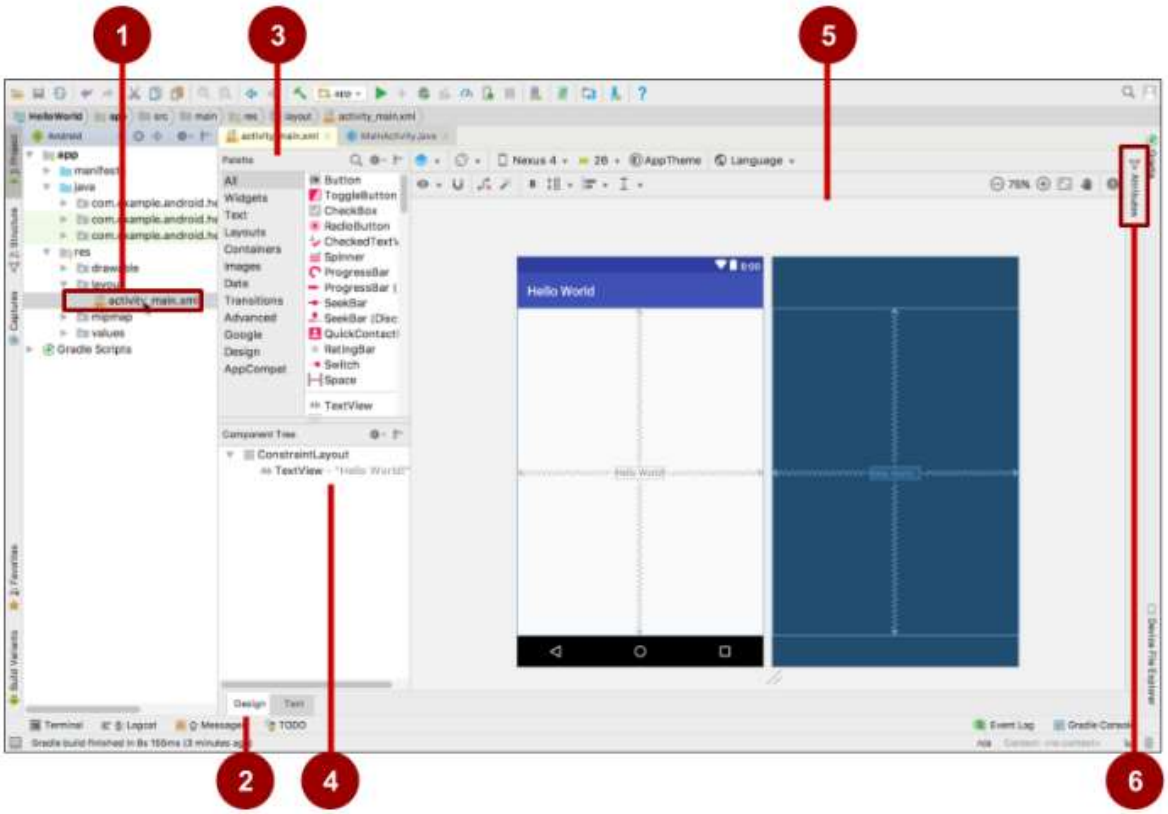
Attribute	Value
Project Name	**Lab2_Toast_Test**
Package Name	**com.example.lab2_toast_test** (or your own domain)
Phone and Tablet Minimum SDK	**API16: Android 4.1 Jelly Bean**
Template	**Empty Activity**

2. Select Run > Run app or click the Run icon  in the toolbar to build and execute the app on the emulator or your device.

Explore the layout editor

Android Studio provides the layout editor for quickly building an app's layout of user interface (UI) elements. It lets you drag elements to a visual design and blueprint view, position them in the layout, add constraints, and set attributes. *Constraints* determine the position of a UI element within the layout. A constraint represents a connection or alignment to another view, the parent layout, or an invisible guideline.

Explore the layout editor, and refer to the figure below as you follow the numbered steps:



1. In the **app > res > layout** folder in the **Project > Android** pane, double-click the **activity_main.xml** file to open it, if it is not already open.

2. Click the **Design** tab if it is not already selected. You use the **Design** tab to manipulate elements and the layout, and the **Text** tab to edit the XML code for the layout.

3. The **Palettes** pane shows UI elements that you can use in your app's layout.

4. The **Component tree** pane shows the view hierarchy of UI elements. View elements are organized into a tree hierarchy of parents and children, in which a child inherits the attributes of its parent. In the figure above, the TextView is a child of the ConstraintLayout. You will learn about these elements later.

5. The design and blueprint panes of the layout editor showing the UI elements in the layout. In the figure above, the layout shows only one element: a TextView that displays "Hello World".

6. The Attributes tab displays the Attributes pane for setting properties for a UI element.

Activity 2: Add View elements in the layout editor

In this task you create the UI layout for the Lab2_Toast_Test app in the layout editor using the ConstraintLayout features. You can create the constraints manually, as shown later, or automatically using the **Autoconnect** tool.

2.1 Examine the element constraints

Follow the following steps:

1. Open activity_main.xml from the **Project > Android** pane if it is not already open. If the **Design** tab is not already selected, click it.

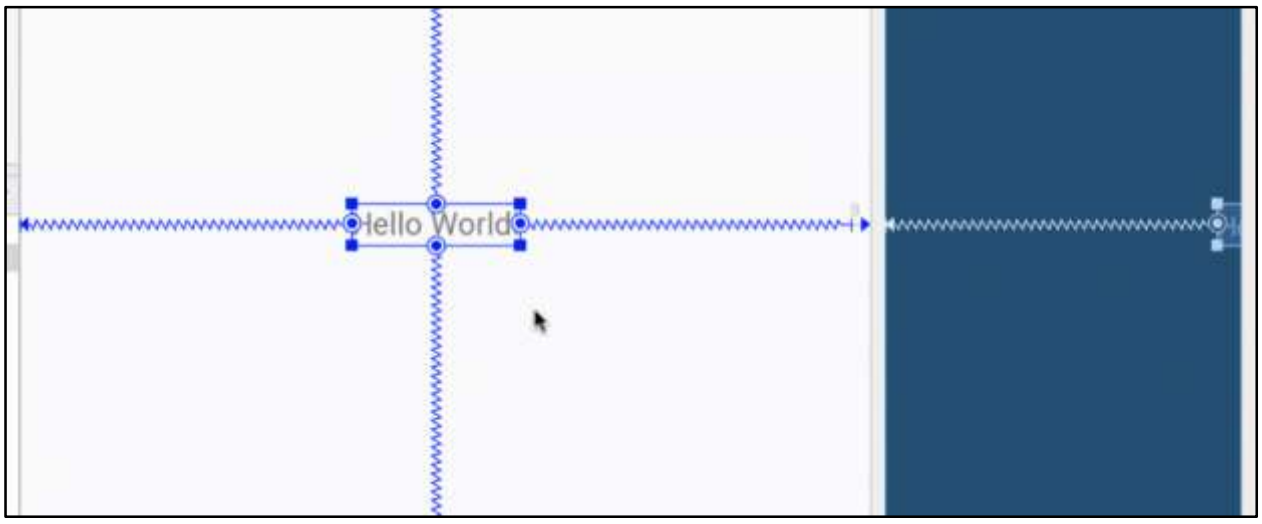
If there is no blueprint, click the **Select Design Surface** button  in the toolbar and choose **Design + Blueprint**.

2. The **Autoconnect** tool  is also located in the toolbar. It is enabled by default. For this step, ensure that the tool is not disabled.

3. Click the zoom in button to zoom into the design and blueprint panes for a close-up look.

4. Select **TextView** in the Component Tree pane. The "Hello World" TextView is highlighted in the design and blueprint panes and the constraints for the element are visible.

5. Click the circular handle on the right side of the TextView to delete the horizontal constraint that binds the view to the right side of the layout. The TextView jumps to the left side because it is no longer constrained to the right side. To add back the horizontal constraint, click the same handle



In the blueprint or design panes, the following handles appear on the `TextView` element:

- **Constraint handle**: To create a constraint as shown in the figure above, click a constraint handle, shown as a circle on the side of an element. Then drag the handle to another constraint handle, or to a parent boundary. A zigzag line represents the constraint.

- **Resizing handle**: To resize the element, drag the square resizing handles. The handle changes to an angled corner while you are dragging it.

2.2 Add a Button to the layout

When enabled, the **Autoconnect** tool automatically creates two or more constraints for a UI element to the parent layout. After you drag the element to the layout, it creates constraints based on the element's position.

Follow these steps to add a Button:

1. Start with a clean slate. The TextView element is not needed, so while it is still selected, press the **Delete** key or choose **Edit > Delete**. You now have a completely blank layout.

2. Drag a **Button** from the **Palette** pane to any position in the layout. If you drop the Button in the top middle area of the layout, constraints may automatically appear. If not, you can drag constraints to the top, left side, and right side of the layout.

2.3 Add a second Button to the layout

1. Drag another **Button** from the **Palette** pane to the middle of the layout as shown in the animated figure below. Autoconnect may provide the horizontal constraints for you (if not, you can drag them yourself).

2. Drag a vertical constraint to the bottom of the layout (refer to the figure below).

You can remove constraints from an element by selecting the element and hovering your pointer over it to show the Clear Constraints  button. Click this button to remove *all* constraints on the selected element. To clear a single constraint, click the specific handle that sets the constraint.

To clear all constraints in the entire layout, click the **Clear All Constraints** tool in the toolbar. This tool is useful if you want to redo all the constraints in your layout.

Activity 3: Change UI element attributes

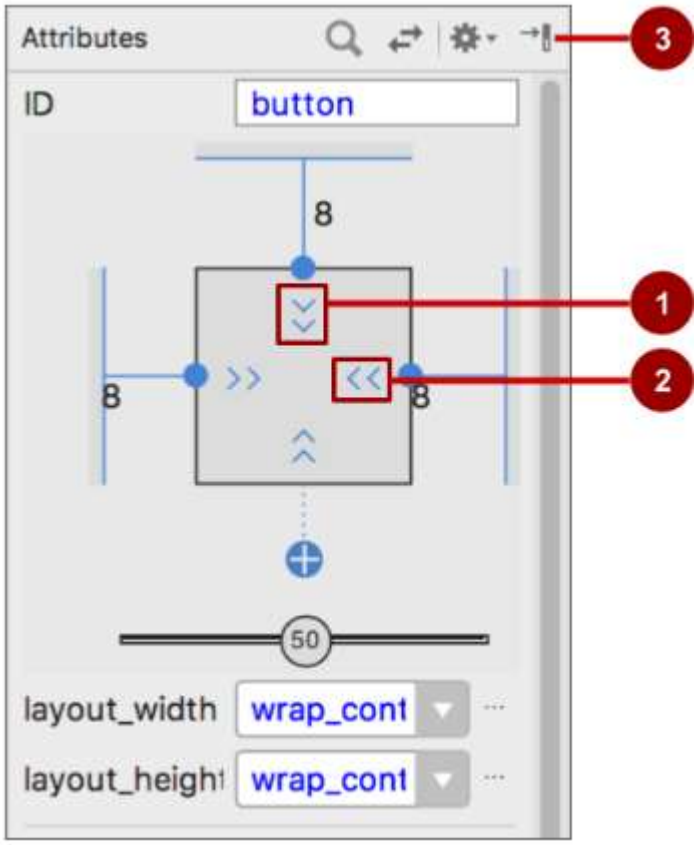
The **Attributes** pane offers access to all of the XML attributes you can assign to a UI element. You can find the attributes (known as *properties*) common to all views in the View class documentation http://developer.android.com/reference/android/view/View.html.

In this task you enter new values and change values for important Button attributes, which are applicable to most View types

3.1 Change the Button size

The layout editor offers resizing handles on all four corners of a View so you can resize the View quickly. You can drag the handles on each corner of the View to resize it, but doing so hardcodes the width and height dimensions. Avoid hardcoding sizes for most View elements, because hardcoded dimensions can't adapt to different content and screen sizes.

Instead, use the **Attributes** pane on the right side of the layout editor to select a sizing mode that doesn't use hardcoded dimensions. The **Attributes** pane includes a square sizing panel called the *view inspector* at the top. The symbols inside the square represent the height and width settings as follows:



In the above figure:

1. **Height control**. This control specifies the layout_height attribute and appears in two segments on the top and bottom sides of the square. The angles indicate that this control is set to wrap_content, which means the View will expand vertically as needed to fit its contents. The "8" indicates a standard margin set to 8dp.

2. **Width control**. This control specifies the layout_width and appears in two segments on the left and right sides of the square. The angles indicate that this control is set to wrap_content, which means the View will expand horizontally as needed to fit its contents, up to a margin of 8dp.

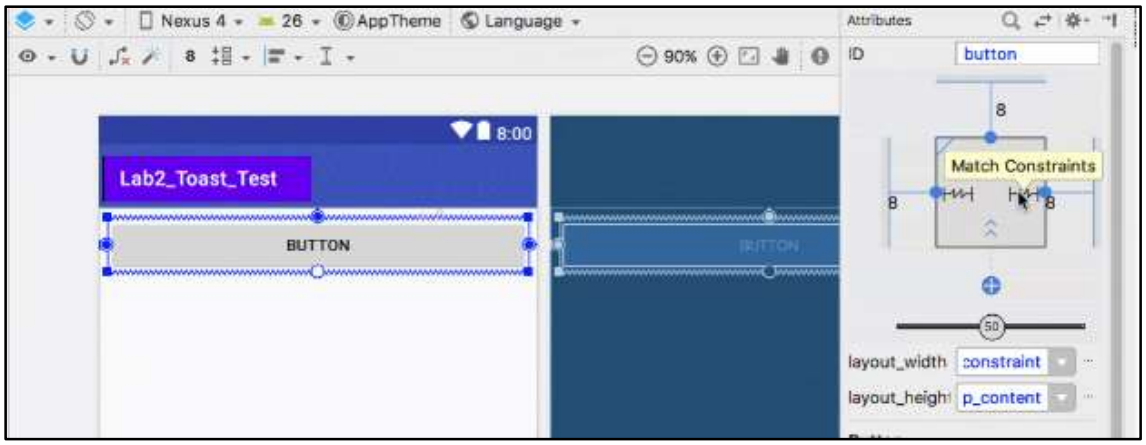
3. **Attributes** pane close button. Click to close the pane.

Follow these steps:

1. Select the top Button in the **Component Tree** pane.

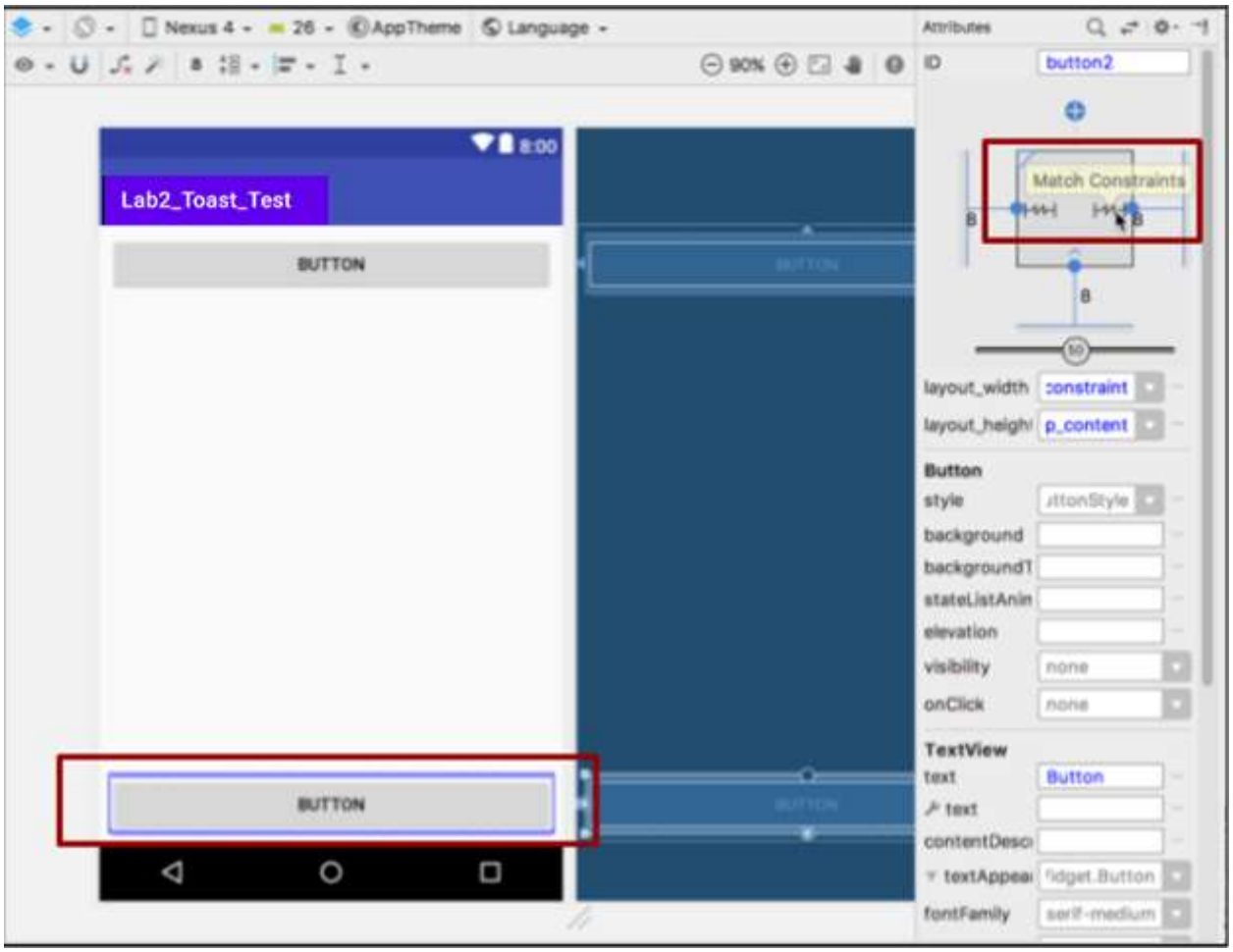
2. Click the **Attributes** tab on the right side of the layout editor window.

3. Click the width control twice—the first click changes it to Fixed with straight lines, and the second click changes it to Match Constraints with spring coils, as shown in the animated figure below.



As a result of changing the width control, the layout_width attribute in the **Attributes** pane shows the value match_constraint and the Button element stretches horizontally to fill the space between the left and right sides of the layout.

4. Select the second Button, and make the same changes to the layout_width as in the previous step, as shown in the figure below.



As shown in the previous steps, the layout_width and layout_height attributes in the **Attributes** pane change as you change the height and width controls in the inspector. These attributes can take one of three values for the layout, which is a ConstraintLayout:

- The match_constraint setting expands the View element to fill its parent by width or height—up to a margin, if one is set. The parent in this case is the ConstraintLayout.
- The wrap_content setting shrinks the View element's dimensions so it is just big enough to enclose its content. If there is no content, the View element becomes invisible.
- To specify a fixed size that adjusts for the screen size of the device, use a fixed number of density-independent pixels (dp units). For example, 16dp means 16 density-independent pixels.

Tip: If you change the layout_width attribute using its popup menu, the layout_width attribute is set to zero because there is no set dimension. This setting is the same as match_constraint— the view can expand as much as possible to meet constraints and margin settings.

3.2 Change the Button attributes

To identify each View uniquely within an Activity layout, each View or View subclass (such as Button) needs a unique ID. And to be of any use, the Button elements need text. View elements can also have backgrounds that can be colors or images.

The **Attributes** pane offers access to all of the attributes you can assign to a View element. You can enter values for each attribute, such as the android:id, background, textColor, and text attributes.

1. After selecting the first Button, edit the ID field at the top of the **Attributes** pane to **button_toast** for the android:id attribute, which is used to identify the element in the layout.
2. Set the background attribute to **@color/purple_500**. (As you enter **@c**, choices appear for easy selection.). You can use any other color too.
3. Set the textColor attribute to **@android:color/white**.
4. Edit the text attribute to **Toast**.
5. Perform the same attribute changes for the second Button, using button_count as the ID, Count for the text attribute, and the same colors for the background and text as the previous steps.

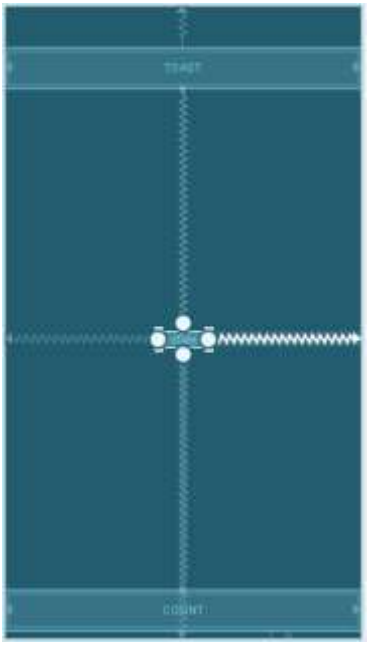
The purple_500 is the primary color of the theme, one of the predefined theme base colors defined in the colors.xml resource file. It is used for the app bar. Using the base colors for other UI elements creates a uniform UI.

Activity 4: Add a TextEdit and set its attributes

One of the benefits of ConstraintLayout is the ability to align or otherwise constrain elements relative to other elements. In this task you will add a TextView in the middle of the layout, and constrain it horizontally to the margins and vertically to the two Button elements. You will then change the attributes for the TextView in the **Attributes** pane.

4.1 Add a TextView and constraints

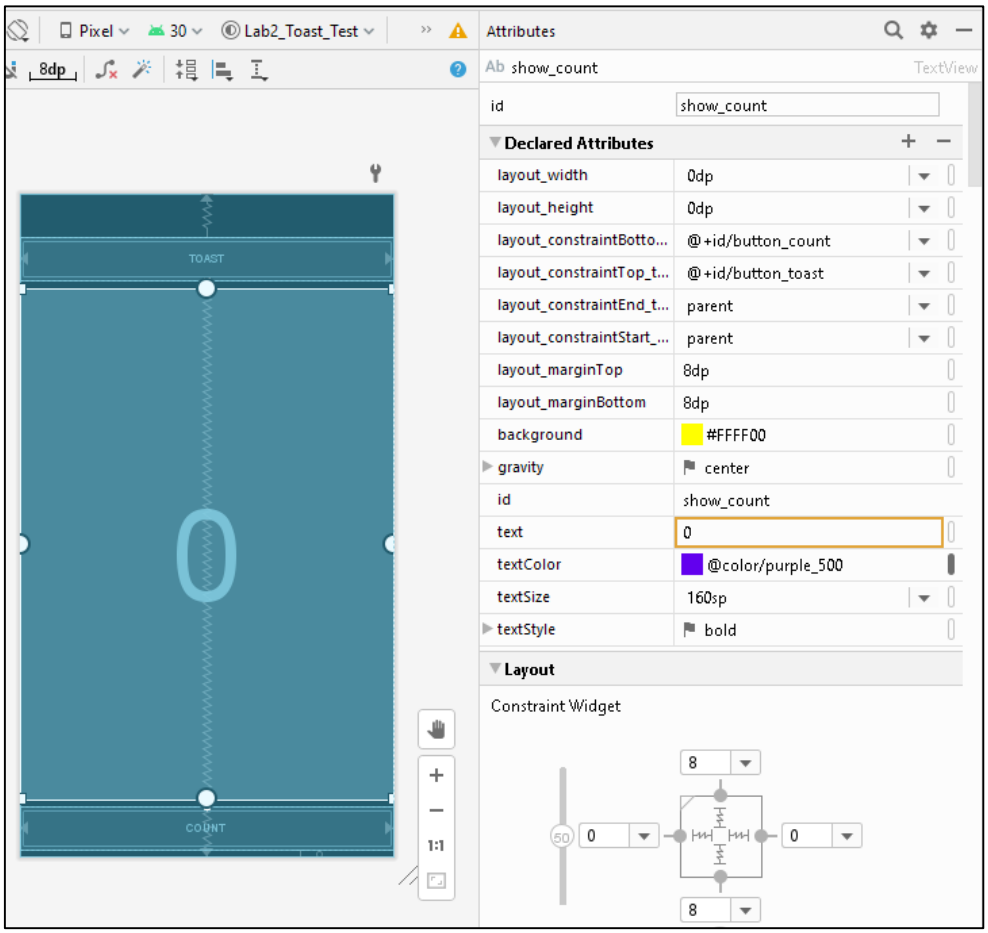
1. Drag a TextView from the **Palette** pane to the upper part of the layout, and drag a constraint from the top of the TextView to the handle on the bottom of the **Toast** Button. This constrains the TextView to be underneath the Button.
2. Drag a constraint from the bottom of the TextView to the handle on the top of the **Count** Button, and from the sides of the TextView to the sides of the layout. This constrains the TextView to be in the middle of the layout between the two Button elements.



4.2 Set the TextView attributes

With the TextView selected, open the **Attributes** pane, if it is not already open. Set attributes for the TextView as shown in the figure below. The attributes you haven't encountered yet are explained after the figure:

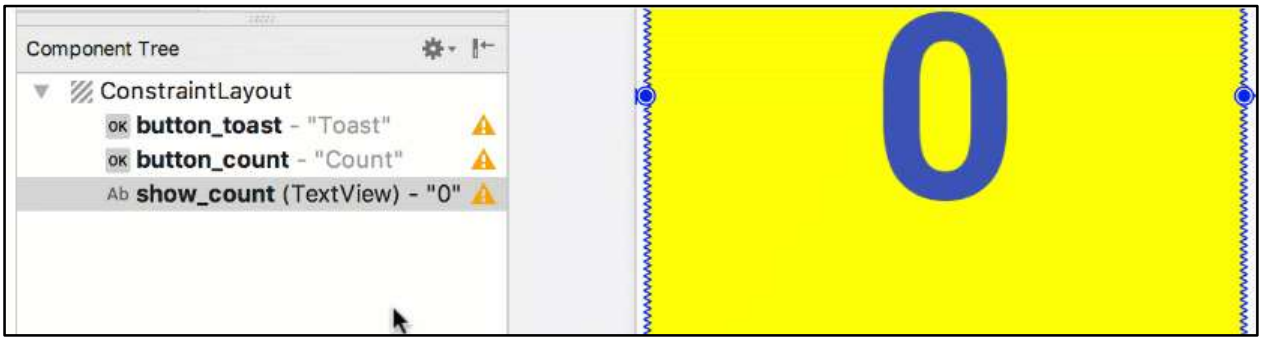
1. Set the ID to **show_count**.
2. Set the text to **0**.
3. Set the textSize to **160sp**.
4. Set the textStyle to **B** (bold) and the textAlignment to ALIGNCENTER (center the paragraph).
5. Change the horizontal and vertical view size controls (layout_width and layout_height) to **match_constraint**.(0dp)
6. Set the textColor to **@color/colorPrimary**.
7. Scroll down the pane and click **View all attributes**, scroll down the second page of attributes to background, and then enter **#FFFF00** for a shade of yellow.
8. Scroll down to gravity, expand gravity, and select **center_ver** (for center-vertical).



- textSize: The text size of the TextView. For this lesson, the size is set to 160sp. The sp stands for *scale-independent pixel*, and like dp, is a unit that scales with the screen density and user's font size preference. Use dp units when you specify font sizes so that the sizes are adjusted for both the screen density and the user's preference.
- textStyle and textAlignment: The text style, set to **B** (bold) in this lesson, and the text alignment, set to ALIGNCENTER (center the paragraph).
- gravity: The gravity attribute specifies how a View is aligned within its *parent* View or ViewGroup. In this step, you center the TextView to be centered vertically within the parent ConstraintLayout.

Activity 5: Edit the layout in XML

The Lab2_Toast_Test app layout is nearly finished! However, an exclamation point appears next to each UI element in the Component Tree. Hover your pointer over these exclamation points to see warning messages, as shown below. The same warning appears for all three elements: hardcoded strings should use resources.

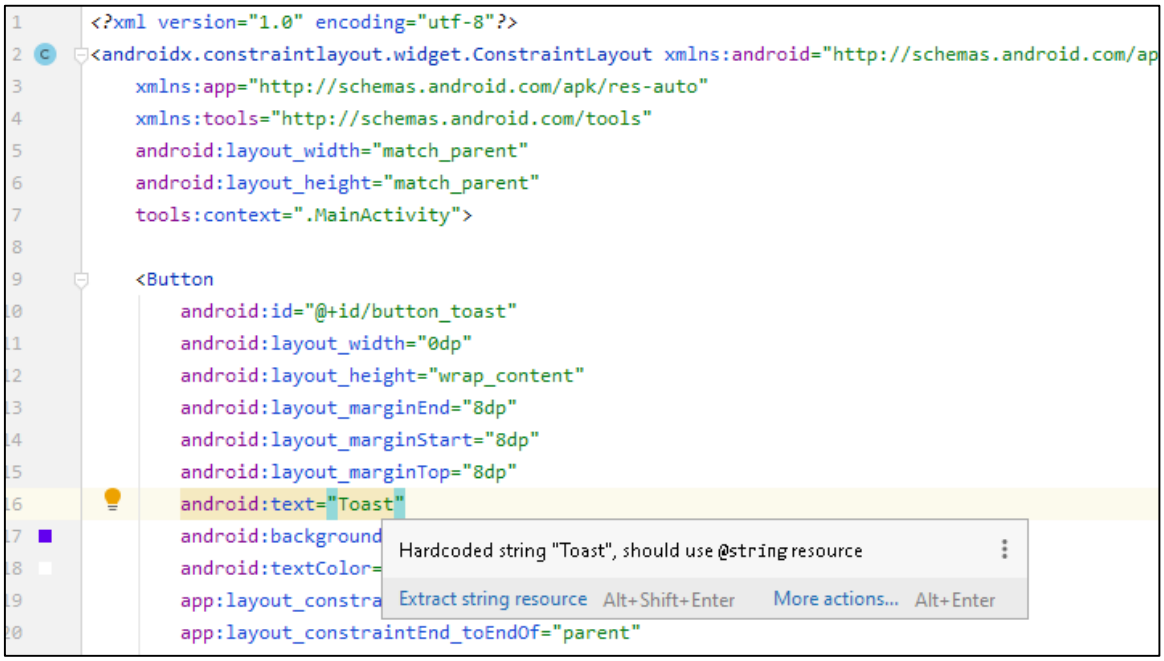


The easiest way to fix layout problems is to edit the layout in XML. While the layout editor is a powerful tool, some changes are easier to make directly in the XML source code.

5.1 Open the XML code for the layout

For this task, open the `activity_main.xml` file if it is not already open, and click the **Text/Code** tab at the top/bottom of the layout editor.

The XML editor appears, replacing the design and blueprint panes. As you can see in the figure below, which shows part of the XML code for the layout, the warnings are highlighted—the hardcoded strings `"Toast"` and `"Count"`. (The hardcoded `"0"` is also highlighted but not shown in the figure.) Hover your pointer over the hardcoded string `"Toast"` to see the warning message.



5.2 Extract string resources

Instead of hard-coding strings, it is a best practice to use string resources, which represent the strings. Having the strings in a separate file makes it easier to manage them, especially if you use these strings more than once. Also, string resources are mandatory for translating and localizing your app, because you need to create a string resource file for each language.

1. Click once on the word "Toast"(the first highlighted warning).

2. Press **Alt-Enter** in Windows or **Option-Enter** in macOS and choose **Extract string resource** from the popup menu.

3. Enter **button_label_toast** for the **Resource name**.

4. Click **OK**. A string resource is created in the values/res/string.xml file, and the string in your code is replaced with a reference to the resource:

@string/button_label_toast

5. Extract the remaining strings: button_label_count for "Count", and count_initial_value for "0".

6. In the **Project > Android** pane, expand **values** within **res**, and then double-click **strings.xml** to see your string resources in the strings.xml file:

```
<resources>
    <string name="app_name"> Lab2_Toast_Test </string>
    <string name="button_label_toast">Toast</string>
    <string name="button_label_count">Count</string>
    <string name="count_initial_value">0</string>
</resources>
```

7. You need another string to use in a subsequent task that displays a message. Add to the strings.xml file another string resource named toast_message for the phrase "Hello Toast!":

```
<resources>
    <string name="app_name"> Lab2_Toast_Test </string>
    <string name="button_label_toast">Toast</string>
    <string name="button_label_count">Count</string>
    <string name="count_initial_value">0</string>
    <string name="toast_message">Hello Toast!</string>
</resources>
```

Tip: The string resources include the app name, which appears in the app bar at the top of the screen if you start your app project using the Empty Template. You can change the app name by editing the **app_name** resource.

Activity 6: Add onClick handlers for the buttons

In this task, you add a Java method for each Button in MainActivity that executes when the user taps the Button.

6.1 Add the onClick attribute and handler to each Button

A ***click handler*** is a method that is invoked when the user clicks or taps on a clickable UI element. In Android Studio you can specify the name of the method in the **onClick** field in the Design tab's Attributes pane. You can also specify the name of the handler method in the XML editor by adding the **android:onClick** property to the **Button**. You will use the latter method because you haven't yet created the handler methods, and the XML editor provides an automatic way to create those methods.

1. With the XML editor open (the Text tab), find the Button with the android:id set to button_toast

2. Add the android:onClick attribute to the end of the button_toast element after the last attribute and before the /> end indicator:

```
android:onClick="showToast" />
```

3. Click the red bulb icon that appears next to attribute. Select **Create click handler**, choose **MainActivity**, and click **OK**.

If the red bulb icon doesn't appear, click the method name ("showToast"). Press **Alt-Enter** (**Option-Enter** on the Mac), select **Create 'showToast(view)' in MainActivity**, and click **OK**.

This action creates a placeholder method stub for the showToast() method in MainActivity, as shown at the end of these steps.

4. Repeat the last two steps with the button_count Button: Add the android:onClick attribute to the end, and add the click handler:

```
android:onClick="countUp" />
```

The XML code for the UI elements within the ConstraintLayout now looks like this:

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

    <Button
        android:id="@+id/button_toast"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:text="@string/button_label_toast"
        android:background="@color/purple_500"
        android:textColor="@color/white"
        android:onClick="showToast"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        />
```

```
<Button
        android:id="@+id/button_count"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginBottom="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:text="@string/button_label_count"
        android:background="@color/purple_500"
        android:textColor="@color/white"
        android:onClick="countUp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent" />

    <TextView
        android:id="@+id/show_count"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:layout_marginTop="8dp"
        android:layout_marginBottom="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginEnd="8dp"
        android:gravity="center"
        android:text="@string/count_initial_value"
        android:textAlignment="center"
        android:textSize="160sp"
        android:textStyle="bold"
        android:background="#FFFF00"
        android:textColor="@color/purple_500"
        app:layout_constraintBottom_toTopOf="@+id/button_count"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/button_toast" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

5. If MainActivity.java is not already open, expand **java** in the Project > Android view, expand **com.example.lab2_toast_test**, and then double-click **MainActivity**. The code editor appears with the code in MainActivity:

```
package com.example.lab2_toast_test;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;

public class MainActivity extends AppCompatActivity {

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

```
    public void showToast(View view) {
    }

    public void countUp(View view) {
    }
}
```

6.2 Edit the Toast Button handler

You will now edit the showToast() method—the **Toast** Button click handler in MainActivity—so that it shows a message. A Toast provides a way to show a simple message in a small popup window. It fills only the amount of space required for the message. The current activity remains visible and interactive. A Toast can be useful for testing interactivity in your app—add a Toast message to show the result of tapping a Button or performing an action.

Follow these steps to edit the **Toast** Button click handler:

1. Locate the newly created showToast() method.

```
public void showToast(View view) {
}
```

2. To create an instance of a Toast, call the makeText() factory method on the Toast class.

```
public void showToast(View view) {
    Toast toast = Toast.makeText(
}
```

This statement is incomplete until you finish all of the steps.

3. Supply the context of the app Activity. Because a Toast displays on top of the Activity UI, the system needs information about the current Activity. When you are already within the context of the Activity whose context you need, use this as a shortcut.

```
Toast toast = Toast.makeText(this,
```

4. Supply the message to display, such as a string resource (the toast_message you created in a previous step). The string resource toast_message is identified by R.string.

```
Toast toast = Toast.makeText(this, R.string.toast_message,
```

5. Supply a duration for the display. For example, Toast.LENGTH_SHORT displays the toast for a relatively short time.

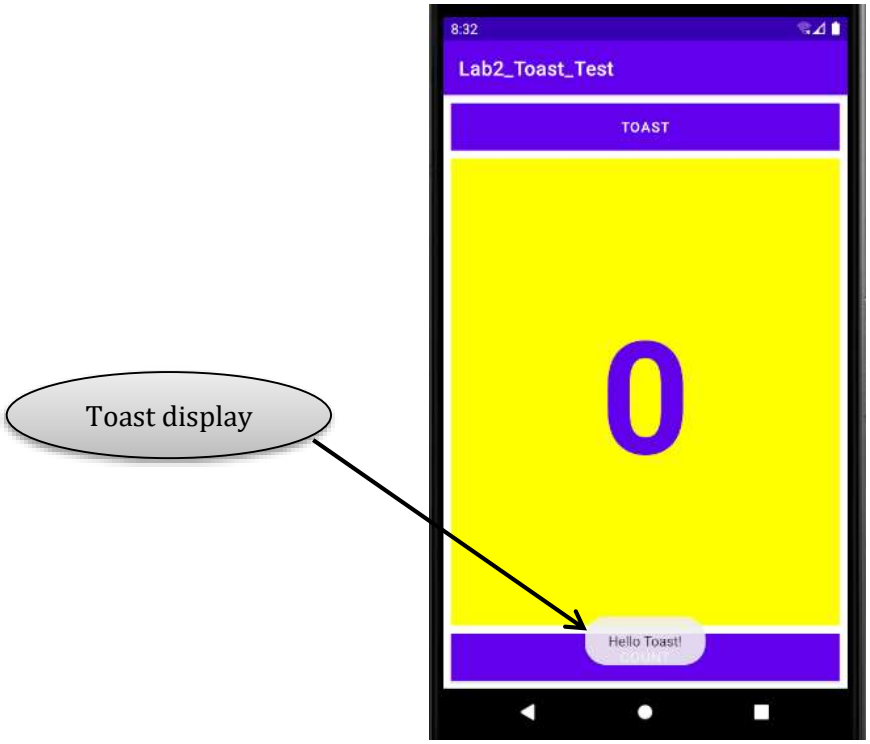
```
Toast toast = Toast.makeText(this, R.string.toast_message,
                                        Toast.LENGTH_SHORT);
```

The duration of a Toast display can be either Toast.LENGTH_LONG or Toast.LENGTH_SHORT. The actual lengths are about 3.5 seconds for the long Toast and 2 seconds for the short Toast.

6. Show the Toast by calling show(). The following is the entire showToast() method:

```
public void showToast(View view) {
Toast.makeText(this,R.string.toast_message,
Toast.LENGTH_SHORT).show();

}
```

Run the app and verify that the Toast message appears when the **Toast** button is tapped.



6.3 Edit the Count Button handler

You will now edit the countUp() method—the **Count** Button click handler in MainActivity—so that it displays the current count after **Count** is tapped. Each tap increases the count by one.

The code for the handler must:

- Keep track of the count as it changes.
- Send the updated count to the TextView to display it.

Follow these steps to edit the **Count** Button click handler:

1. Locate the newly created countUp() method.

```
public void countUp(View view) {
}
```

2. To keep track of the count, you need a private member variable. Each tap of the **Count** button increases the value of this variable. Enter the following, which will be highlighted in red and show a red bulb icon:

```
public void countUp(View view) {
   mCount++;
}
```

If the red bulb icon doesn't appear, select the mCount++ expression. The red bulb eventually appears.

3. Click the red bulb icon and choose **Create field 'mCount'** from the popup menu. This creates a private member variable at the top of MainActivity, and Android Studio assumes that you want it to be an integer (int):

```
public class MainActivity extends AppCompatActivity {
   private int mCount;
```

4. Change the private member variable statement to initialize the variable to zero:

```
public class MainActivity extends AppCompatActivity {
   private int mCount = 0;
```

5. Along with the variable above, you also need a private member variable for the reference of the show_count TextView, which you will add to the click handler. Call this variable mShowCount:

```
public class MainActivity extends AppCompatActivity {
   private int mCount = 0;
   private TextView mShowCount;
```

6. Now that you have mShowCount, you can get a reference to the TextView using the ID you set in the layout file. In order to get this reference only once, specify it in the onCreate() method. As you will learn that the onCreate() method is used to *inflate the layout*, which means to set the content view of the screen to the XML layout. You can also use it to get references to other UI elements in the layout, such as the TextView. Locate the onCreate() method in MainActivity:

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

7. Add the findViewById statement to the end of the method:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  mShowCount = (TextView) findViewById(R.id.show_count);
}
```

A View, like a string, is a resource that can have an id. The findViewById call takes the ID of a view as its parameter and returns the View. Because the method returns a View, you have to cast the result to the view type you expect, in this case (TextView).

8. Now that you have assigned to mShowCount the TextView, you can use the variable to set the text in the TextView to the value of the mCount variable. Add the following to the countUp() method:

```
if (mShowCount != null)
            mShowCount.setText(Integer.toString(mCount));
```

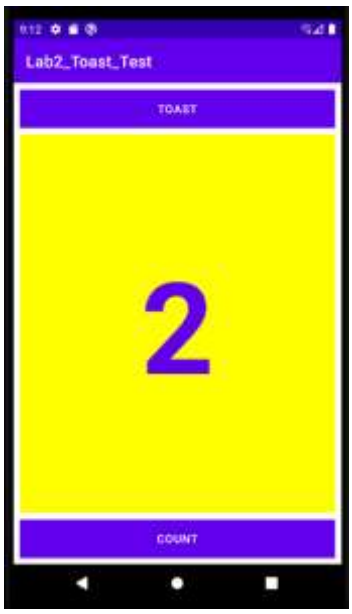
The entire countUp() method now looks like this:

```
public void countUp(View view) {
    ++mCount;
    if (mShowCount != null)
        mShowCount.setText(Integer.toString(mCount));
}
Finally the complete MainActivity.java is as follows:
```

```
package com.example.lab2_toast_test;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;
import android.widget.TextView;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {
    private int mCount = 0;
    private TextView mShowCount;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        //my code
        mShowCount = (TextView) findViewById(R.id.show_count);
    }
```

```
    public void showToast(View view) {
        Toast toast = Toast.makeText(this,
R.string.toast_message,Toast.LENGTH_LONG);
        toast.show();
    }

    public void countUp(View view) {
        mCount++;
        if(mShowCount!=null)
        {mShowCount.setText(Integer.toString(mCount));}
    }
}
```

9. Run the app to verify that the count increases when you tap the **Count** button two times:



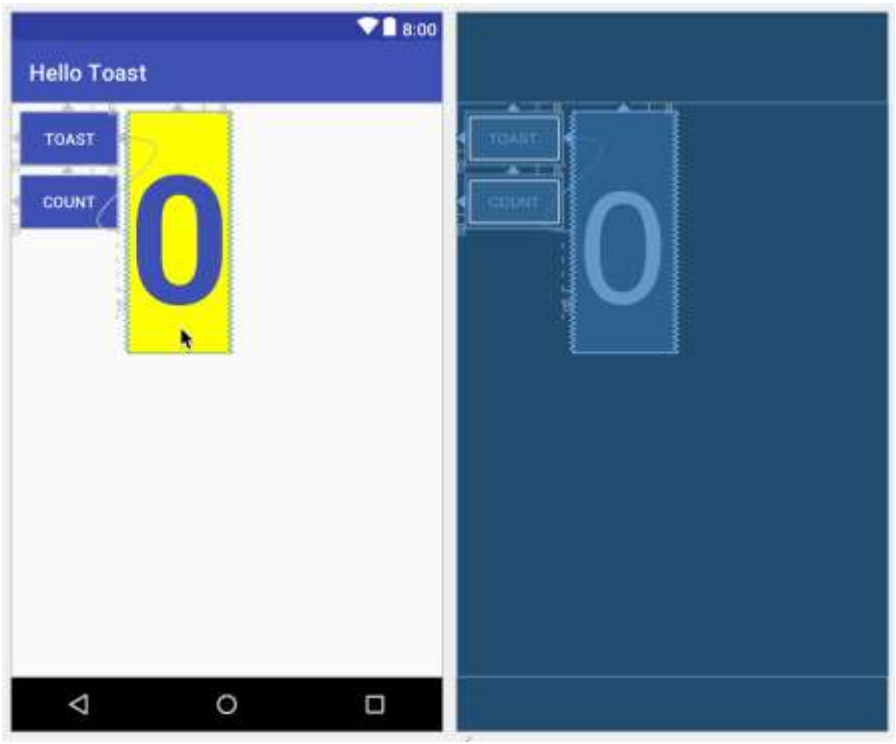
Use different colors of your own choice for above app text in text view, background color of textView and text on buttons, background color of button.

Exercises

The Lab2_Toast_Test app looks fine when the device or emulator is vertically oriented. However, if you switch the device or emulator to horizontal orientation, the **Count** Button may overlap the TextView along the bottom as shown in the figure below.

Challenge: Change the layout so that it looks good in both horizontal and vertical orientations:

1. On your computer, make a copy of the **HelloToast** project folder and rename it to **HelloToastChallenge**.
2. Open **HelloToastChallenge** in Android Studio and refactor it. (See Appendix: Utilities for instructions on copying and refactoring a project.)
3. Change the layout so that the **Toast** Button and **Count** Button appear on the left side, as shown in the figure below. The TextView appears next to them, but only wide enough to show its contents. (Hint: Use wrap_content.)
4. Run the app in both horizontal and vertical orientations.




3:13
Hello Toast
TOAST
COUNT
0

LAB 03: Basic UI components and widget

Objective

- Use of basic UI input controls and widgets to capture user input data at runtime and manipulate the data
- How to change the input methods to enable suggestions, auto-capitalization, and password obfuscation.
- How to change the generic on-screen keyboard to a phone keypad or other specialized keyboards.
- How to add radio buttons for the user to select one item from a set of items.
- How to add a spinner to show a drop-down menu with values, from which the user can select one.

Scope

- Show a keyboard for entering an email address.
- Show a numeric keypad for entering phone numbers.
- Allow multiple-line text entry with automatic sentence capitalization.
- Add radio buttons for selecting an option.
- Set an onClick handler for the radio buttons.
- Add a spinner for the phone number field for selecting one value from a set of values

Useful Concepts

To enable the user to enter text or numbers, you use an EditText element. Some input controls are EditText attributes that define the type of keyboard that appears, to make entering data easier for users. For example, you might choose phone for the android:inputType attribute to show a numeric keypad instead of an alphanumeric keyboard.

Other input controls make it easy for users to make choices. For example, RadioButton elements enable a user to select one (and only one) item from a set of items.

In this practical, you use attributes to control the on-screen keyboard appearance, and to set the type of data entry for an EditText. You also add radio buttons to the RegistrationForm app so the user can select one item from a set of items.

Lab Tasks

The following Lab-example will use TextView, EditText with different input types, Radio Buttons in a RadioGriup, MultiLine EditText and Spinners

Activity 1: Use of TextView and EditText(with text entry attributes)

Touching an EditText editable text field places the cursor in the text field and automatically displays the on-screen keyboard so that the user can enter text.

An editable text field expects a certain type of text input, such as plain text, an email address, a phone number, or a password. It's important to specify the input type for each text field in your app so that the system displays the appropriate soft input method, such as an on-screen keyboard for plain text, or a numeric keypad for entering a phone number.

In this lab we are creating a general registration form hence give name of app will be "RegistrationForm". We put the app name as "Registration Form" in the string.xml file as the title of activities.

1.1 Add an EditText for entering a name

In this step you add a TextView and an EditText to the MainActivity layout in the RegistrationForm app so that the user can enter a person's name.

1.1.1. Open the **activity_main.xml** layout file, which uses a ConstraintLayout.

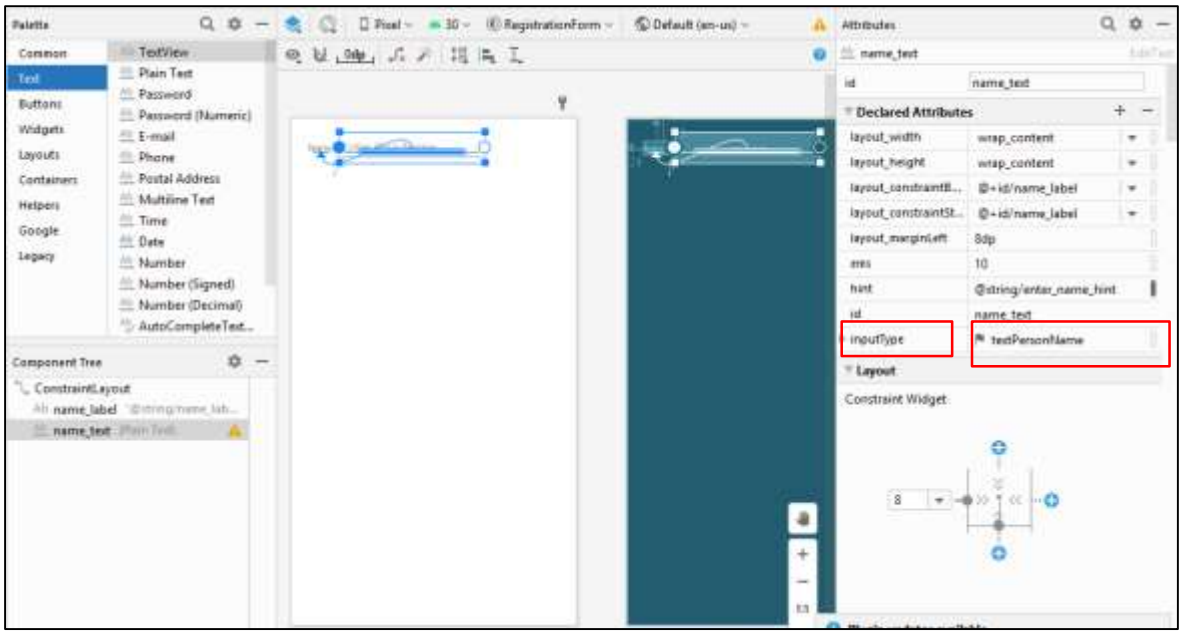
1.1.2. Add a TextView to the ConstraintLayout in activity_main.xml. Use the following attributes for the new TextView:

TextView attribute	Value
android:id	"@+id/name_label"
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:layout_marginLeft	"24dp"
android:layout_marginTop	"32dp"
android:text	"Name"
app:layout_constraintStart_toStartOf	"parent"

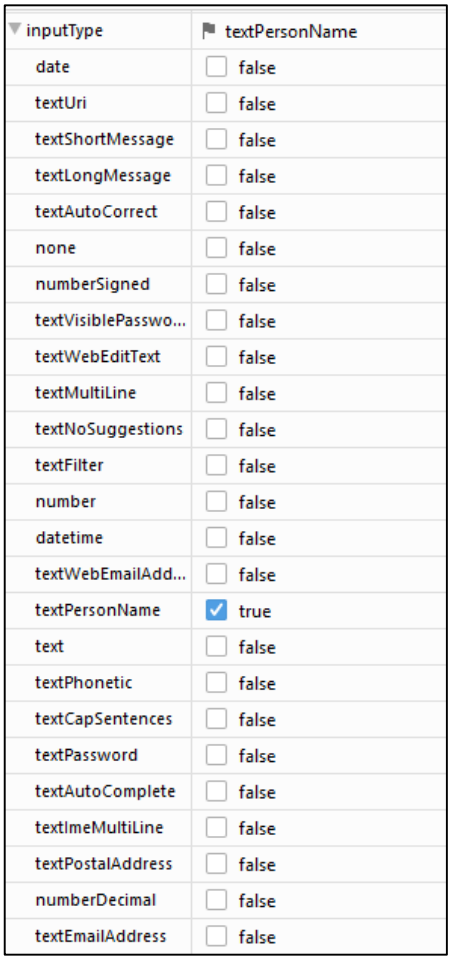
```
app:layout_constraintTop_toTopOf                    "parent"
```

1.1.3. Extract the string resource for the android:text attribute value to create and entry for it called name_label_text in strings.xml.

1.1.4. Add an EditText element. To use the visual layout editor, drag a **Plain Text** element from the **Palette** pane to a position next to the name_label TextView. Then enter **name_text** for the **ID** field, and constrain the left side and baseline of the element to the name_label element right side and baseline as shown in the figure below:



1.1.5. The figure above highlights the **inputType** field in the **Attributes** pane to show that Android Studio automatically assigned the textPersonName type. Click the **inputType** field to see the menu of input types:



In the figure above, **textPersonName** is selected as the input type.

1.1.6. Add a hint for text entry, such as **Enter your name**, in the **hint** field in the **Attributes** pane, and delete the **Name** entry in the **text** field. As a hint to the user, the text "Enter your name" should be dimmed inside the EditText.

1.1.7. Check the XML code for the layout by clicking the **Text** tab. Extract the string resource for the android:hint attribute value to enter_name_hint. The following attributes should be set for the new EditText (add the layout_marginLeft attribute for compatibility with older versions of Android):

EditText attribute	Value
android:id	"@+id/name_text"
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:layout_marginLeft	35dp

android:ems	"10"
android:hint	"@string/enter_name_hint"
android:inputType	"textPersonName"
app:layout_constraintBaseline_toBaselineOf	"@+id/name_label"
app:layout_constraintStart_toEndOf	"@+id/name_label"

As you can see in the XML code, the android:inputType attribute is set to textPersonName.

1.1.8. Run the app.Tap inside the text entry field to show the keyboard and enter text, as shown in the figure below

Note that suggestions automatically appear for words that you enter. Tap a suggestion to use it. This is one of the properties of the textPersonName value for the android:input Type attribute. The inputType attribute controls a variety of features, including keyboard layout, capitalization, and multiple lines of text.

1.1.9. To close the keyboard, tap the checkmark icon in a green circle ,

which appears in the lower right corner of the keyboard. This is known as the **Done** key.

1.2 Add a multiple-line EditText

In this step you add another EditText to the MainActivity layout in the RegistrationForm app so that the user can enter an address using multiple lines.

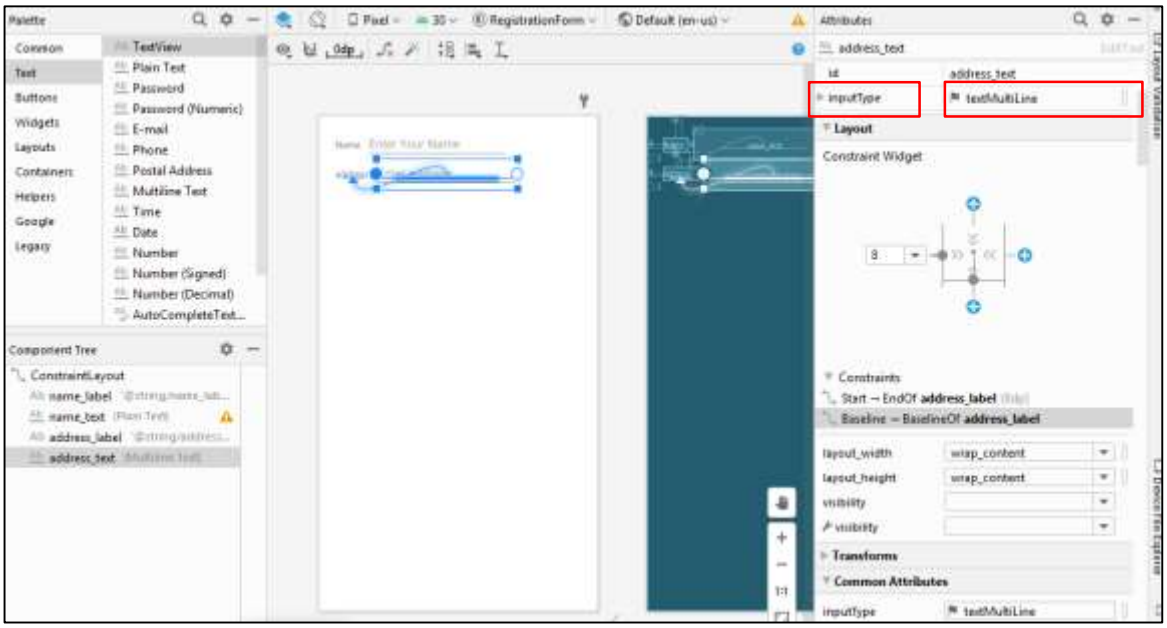
1.2.1 Open the activity_main.xml layout file if it is not already open.

1.2.2 Add a TextView under the name_label element already in the layout. Use the following attributes for the new TextView

TextView attribute	Value
android:id	"@+id/address_label"
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:layout_marginTop	"28dp"
android:layout_marginLeft	"24dp"
android:layout_marginTop	"24dp"
android:text	"Address"
app:layout_constraintStart_toStartOf	"parent"
app:layout_constraintTop_toBottomOf	"@+id/name_label"

1.2.3 Extract the string resource for the android:text attribute value to create and entry for it called address_label_text in strings.xml.

1.2.4 Add an EditText element. To use the visual layout editor, drag a **Multiline Text** element from the **Palette** pane to a position next to the address_label TextView. Then enter **address_text** for the **ID** field, and constrain the left side and baseline of the element to the address_label element right side and baseline as shown in the figure below:

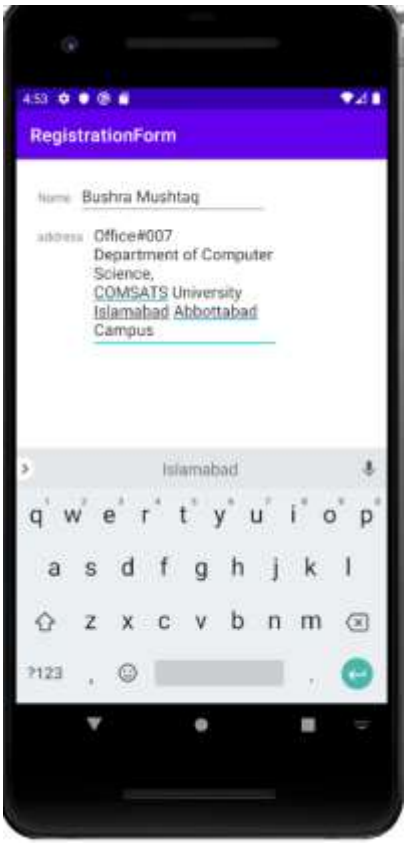


1.2.5 Add a hint for text entry, such as **Enter address**, in the **hint** field in the **Attributes** pane. As a hint to the user, the text "Enter address" should be dimmed inside the EditText.

1.2.6 Check the XML code for the layout by clicking the **Text** tab. Extract the string resource for the android:hint attribute value to enter_address_hint. The following attributes should be set for the new EditText (add the layout_marginLeft attribute for compatibility with older versions of Android):

EditText attribute	Value
android:id	"@+id/address_text"
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:layout_marginTop	28dp
android:layout_marginLeft	32dp
android:ems	"10"
android:hint	"@string/enter_address_hint"
android:inputType	"textMultiLine"
app:layout_constraintBaseline_toBaselineOf	"@+id/address_label"
app:layout_constraintStart_toEndOf	"@+id/address_label"

1.2.7 Run the app. Inside the "Address" text entry field to show the keyboard and enter text, as shown in the figure below, using the Return key in the lower right corner of the keyboard (also known as the Enter or New Line key) to start a new line of text. The Return key appears if you set the `textMultiLine` value for the `android:inputType` attribute.



1.2.8 To close the keyboard, tap the down-arrow button that appears instead of the Back button in the bottom row of buttons.

1.3 Use a Keypad for phone numbers

In this step you add another `EditText` to the `MainActivity` layout in the RegistrationForm app so that the user can enter a phone number on a numeric keypad.

1.3.1 Open the activity_main.xml layout file if it is not already open

1.3.2 Add TextView under the address_label element already in the layout. Use the following attributes for the new TextView.

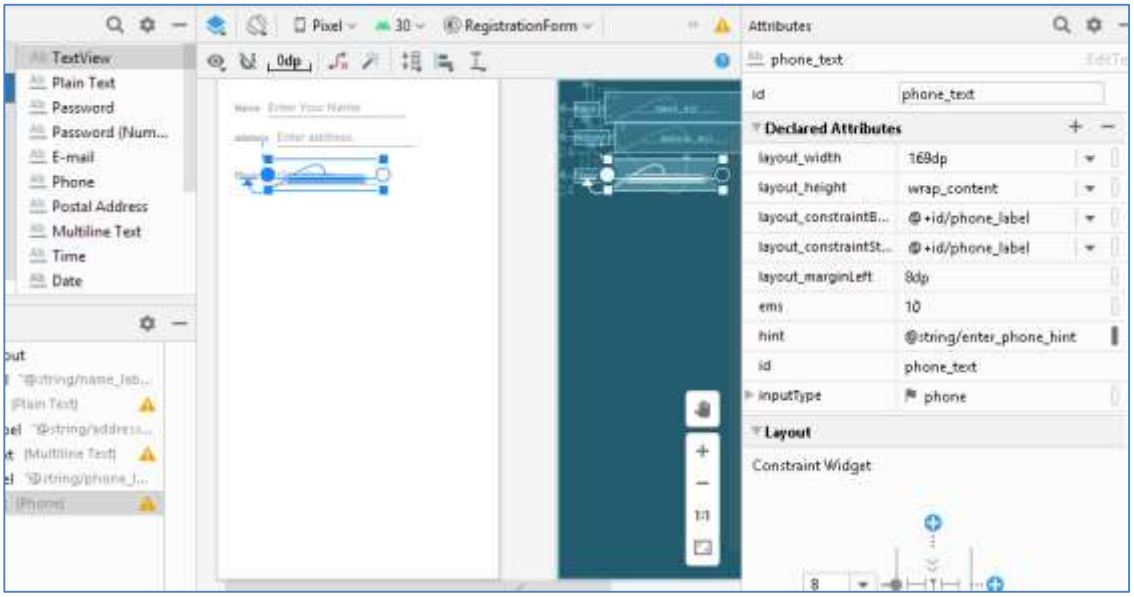
TextView attribute	Value
android:id	"@+id/phone_label"
android:layout_width	"wrap_content"

android:layout_height	"wrap_content"
android:layout_marginStart	"24dp"
android:layout_marginLeft	"24dp"
android:layout_marginTop	"24dp"
android:text	"Phone"
app:layout_constraintStart_toStartOf	"parent"
app:layout_constraintTop_toBottomOf	"@+id/address_text"

Note that this TextView is constrained to the bottom of the multiple-line EditText (address_text). This is because address_text can grow to multiple lines, and this TextView should appear beneath it.

1.3.3 Extract the string resource for the android:text attribute value to create and entry for it called phone_label_text in strings.xml.

1.3.4 Add an EditText element. To use the visual layout editor, drag a **Phone** element from the **Palette** pane to a position next to the phone_label TextView. Then enter **phone_text** for the **ID** field, and constrain the left side and baseline of the element to the phone_label element right side and baseline as shown in the figure below:

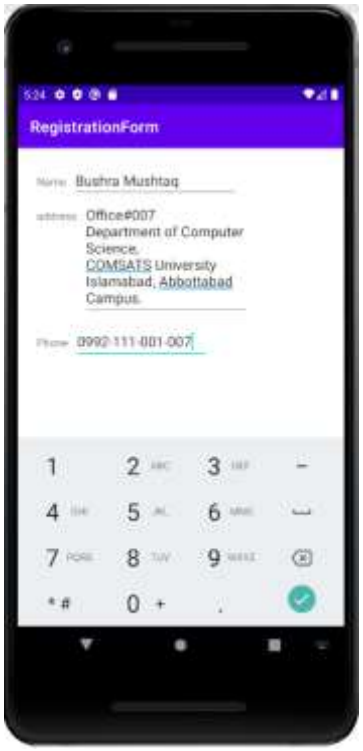


1.3.5 Add a hint for text entry, such as **Enter phone**, in the **hint** field in the **Attributes** pane. As a hint to the user, the text "Enter phone" should be dimmed inside the EditText.

1.3.6 Check the XML code for the layout by clicking the **Text** tab. Extract the string resource for the android:hin attribute value to enter_phone_hint. The following attributes should be set for the new EditText (add the layout_marginLeft attribute for compatibility with older versions of Android):

EditText attribute	Value
android:id	"@+id/phone_text"
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:layout_marginTop	24dp
android:layout_marginLeft	32dp
android:ems	"10"
android:hint	"@string/enter_phone_hint"
android:inputType	"phone"
app:layout_constraintBaseline_toBaselineOf	"@+id/phone_label"
app:layout_constraintStart_toEndOf	"@+id/phone_label"

1.3.7 Run app, Tap Inside the phone field to show the numeric keypad. You can then enter a phone number as shown in the following figure.



1.3.8 To close the keypad, tap the Done key

1.4 Combine Input Types in one EditText

You can combine inputType attribute values that don't conflict with each other. For example, you can combine the textMultiLine and textCapSentences attribute values for multiple lines of text in which each sentence starts with a capital letter.

1.4.1 Open the activity_main.xml layout file if it is not already open

1.4.2 Add a TextView under the phone_label element already in the layout. Use the following attributes for the new TextView:

TextView attribute	Value
android:id	"@+id/note_label"
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:layout_marginStart	"24dp"
android:layout_marginLeft	"24dp"
android:layout_marginTop	"24dp"
android:text	"Note"
app:layout_constraintStart_toStartOf	"parent"
app:layout_constraintTop_toBottomOf	"@+id/phone_label"

1.4.3 Extract the string resource for the android:text attribute value to create and entry for it called note_label_text in strings.xml

1.4.4 Add an EditText element. To use the visual layout editor, drag a **Multiline Text** element from the **Palette** pane to a position next to the note_label TextView. Then enter **note_text** for the **ID** field, and constrain the left side and baseline of the element to the note_label element right side and baseline as you did previously with the other EditText elements.

1.4.5 Add a **hint** for text entry, such as **Enter note** in the hint field in the Attributes pane.

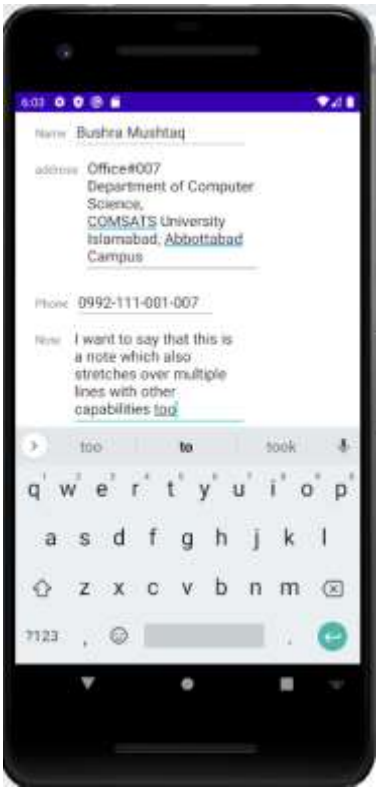
1.4.6 Click inside the **inputType** field in the **Attributes** pane. The **textMultiLine** value is already selected. In addition, select **textCapSentences** to combine these attributes.

1.4.7 Check the XML code for the layout by clicking the **Text** tab. Extract the string resource for the android:hint attribute value to enter_note_hint. The following attributes should be set for the new EditText (add the layout_marginLeft attribute for compatibility with older versions of Android):

EditText attribute	Value
android:id	"@+id/note_text"
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:layout_marginTop	24dp
android:layout_marginLeft	32dp
android:ems	"10"
android:hint	"@string/enter_note_hint"
android:inputType	"textCapSentences
app:layout_constraintBaseline_toBaselineOf	"@+id/note_label"
app:layout_constraintStart_toEndOf	"@+id/note_label"

To combine values for the android:inputType attribute, concatenate them using the pipe (|) character.

1.4.8 Run the app. Tap inside the "Note" filed enter complete sentences as shown in the figure below. Use the return key to create new line, or simple type to wrap sentences over multiple lines.



Activity 2: Use Radio Buttons

Input controls are the interactive elements in your app's UI that accept data input. Radio buttons are input controls that are useful for selecting only one option from a set of options.

In this task you add a group of radio buttons to the RegistrationForm app selecting the gender type.

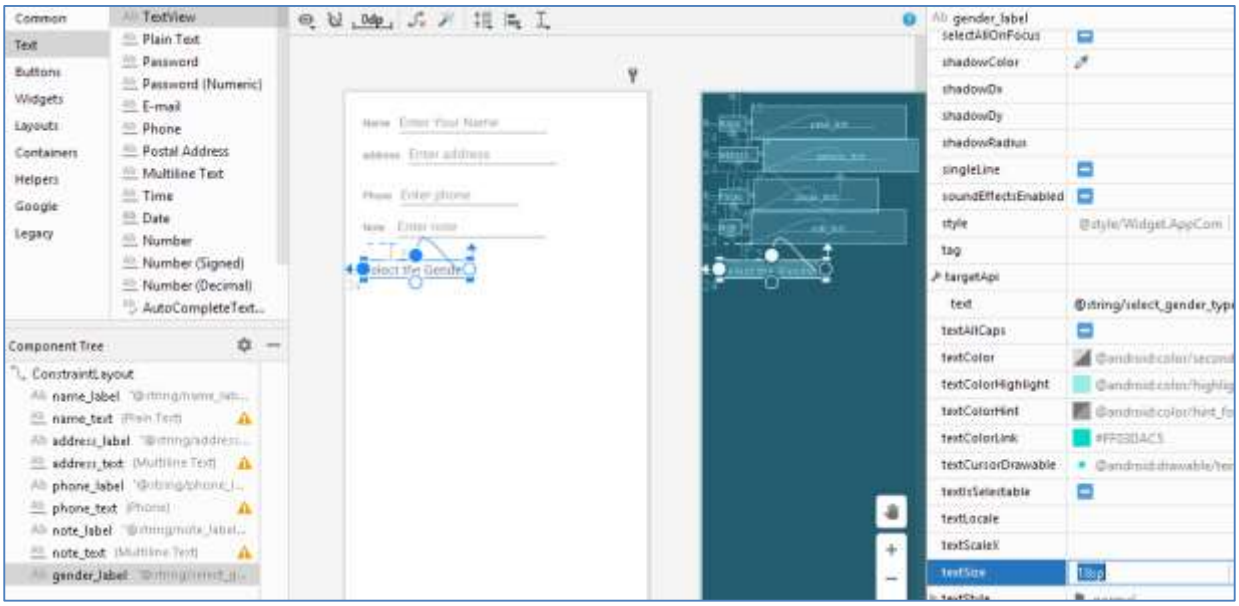
Add a RadioGroup and Radio button

To add radio buttons to MainActivity in the RegistrationForm app, you create RadioButton elements in the activity_main.xml layout file. After editing the layout file, the layout for the radio buttons in MainActivity will look something like the figure below.

Because radio button selections are mutually exclusive, you group them together inside a RadioGroup. By grouping them together, the Android system ensures that only one radio button can be selected at a time.

Note: The order in which you list the RadioButton elements determines the order that they appear on the screen

Open **activity_main.xml** and add a TextView element constrained to the bottom of the note_text element already in the layout. And to the left margin, as shown in the following figure:



Switch to the editing XML, and make sure that you have the following attributes set for the new TextView

TextView attribute	Value
android:id	"@+id/gender_label"
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:layout_marginStart	"24dp"
android:layout_marginLeft	"24dp"
android:layout_marginTop	"24dp"
android:text	"Select the Gender: "
android:textSize	"18sp"
app:layout_constraintStart_toStartOf	"parent"
app:layout_constraintTop_toBottomOf	"@+id/note_text"

Extract the string resource for "Select the Gender:" to be select_gender_type.

To add radio buttons, enclose them within a RadioGroup. Add the RadioGroup to the layout underneath the TextView you just added, enclosing three RadioButton elements as shown in the XML code below::

```
<RadioGroup
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="24dp"
    android:layout_marginLeft="24dp"
    android:orientation="vertical"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/gender_label">

    <RadioButton
        android:id="@+id/male"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="onRadioButtonClicked"
        android:text="Male " />

    <RadioButton
        android:id="@+id/female"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="onRadioButtonClicked"
        android:text="Female " />

    <RadioButton
        android:id="@+id/other"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="onRadioButtonClicked"
        android:text="Other " />

</RadioGroup>
```

The "onRadioButtonClicked" entry for the android:onClick attribute for each RadioButton will be underlined in red until you add that method in the next step of this task.

Extract the three string resources for the android:text attributes to the following names so that the strings can be translated easily: male_gender, female_gender, other_gender.

Add the radio button click handler

The android:onClick attribute for each radio button element specifies the onRadioButtonClicked() method to handle the click event. Therefore, you need to add a new onRadioButtonClicked() method in the MainActivity class.

Open **activity_main.xml** (if it is not already open) and find one of the onRadioButtonClicked values for the android:onClick attribute that is underlined in red.

Click the onRadioButtonClicked value, and then click the red bulb warning icon in the left margin

Choose **Create onRadioButtonClicked(View) in MainActivity** in the red bulb's menu. Android Studio creates the onRadioButtonClicked(View view) method in OrderActivity:

```
public void onRadioButtonClicked(View view) {
}
```

In addition, the onRadioButtonClicked values for the other android:onClick attributes in activity_main.xml are resolved and no longer underlined.

To display which radio button is clicked (that is, the type of delivery the user chooses), use a Toast message. Open **MainActivity** and add the following displayToast method:

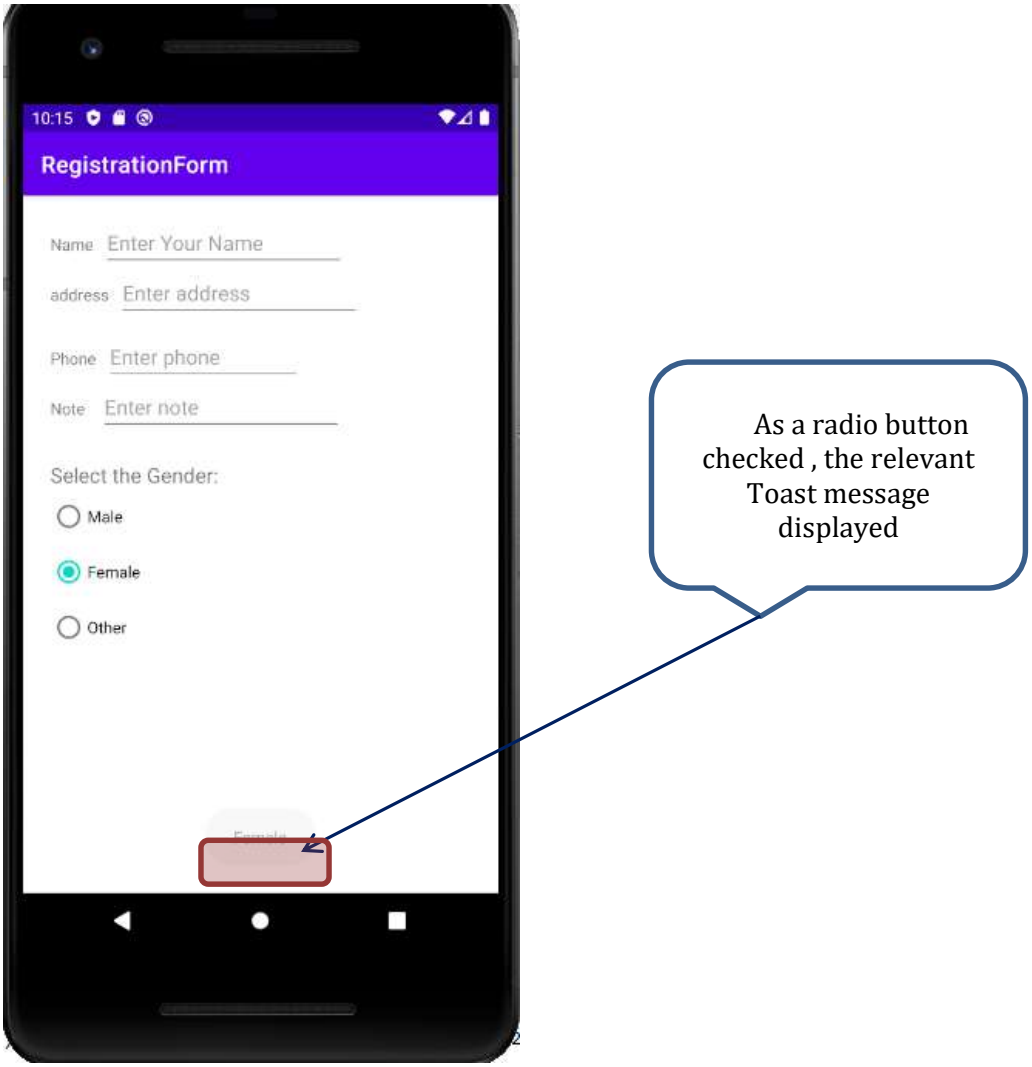
```
public void displayToast(String message) {
   Toast.makeText(getApplicationContext(), message,
                          Toast.LENGTH_SHORT).show();
}
```

In the new onRadioButtonClicked() method, add a switch case block to check which radio button has been selected and to call displayToast() with the appropriate message. The code uses the isChecked() method of the Checkable interface, which returns true if the button is selected. It also uses the View getId() method to get the identifier for the selected radio button view:

```
public void onRadioButtonClicked(View view) {
        // Is the button now checked?
        boolean checked = ((RadioButton) view).isChecked();
        // Check which radio button was clicked.
        switch (view.getId()) {
            case R.id.male:
                if (checked)
                    // male selected
                    displayToast(getString(R.string.male_gender));
                break;
            case R.id.female:
                if (checked)
                    // female selected
                    displayToast(getString(R.string.female_gender));
                break;
            case R.id.other:
                if (checked)
                    // other selected
                    displayToast(getString(R.string.other_gender));
```

```
                break;
            default:
                // Do nothing.
                break;
        }
}
```

Run the app .Tap on any gender choice and you see a Toast message at the bottom of the screen with the choice, as shown in the figure below.



Activity 3: Use a Spinner for user choices

A Spinner provides a quick way to select one value from a set. Touching the Spinner displays a drop-down list with all available values, from which the user can select one. If you are

providing only two or three choices, you might want to use radio buttons for the choices if you have room in your layout for them; however, with more than three choices, a Spinner works very well, scrolls as needed to display items, and takes up little room in your layout.

To provide a way to select a label for a phone number (such as **Home**, **Work**, **Mobile**, or **Other**), you can add a spinner to the MainActivity layout in the RegistrationForm app to appear right next to the phone number field.

Add a Spinner to the layout

To add a spinner to the MainActivity layout in the RegistrationForm app, follow these steps, which are numbered in the figure below:

Open **activity_main.xml** and drag **Spinner** from the **Palette** pane to the layout .

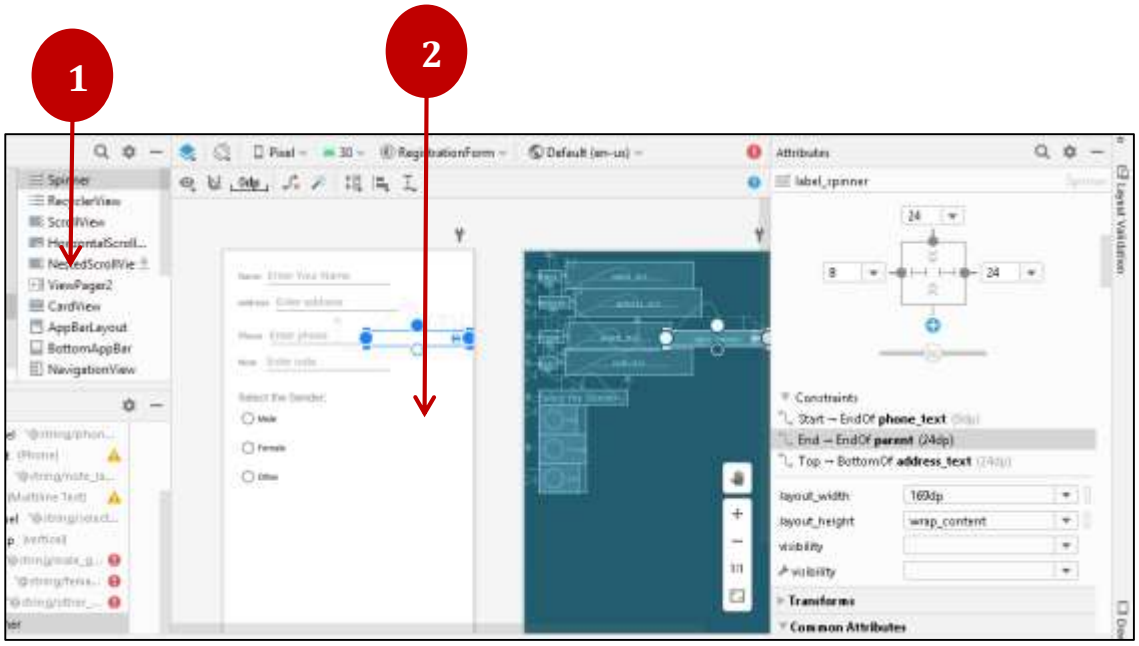
Constrain the top of the Spinner element to the bottom of address_text, the right side to the right side of the layout, and the left side to phone_text.

To align the Spinner and phone_text elements horizontally, use the pack button  in the toolbar, which provides options for packing or expanding selected UI elements.

Select both the Spinner and phone_text elements in the **Component Tree**, click the pack button, and choose **Expand Horizontally**. As a result, both the Spinner and phone_text elements are set to fixed widths.

In the Attributes pane, set the Spinner **ID** to **label_spinner**, and set the top margin 20dp ,right margins to 8, and the left margin to **260dp**. Choose **match_constraint** for the **layout_width** drop-down menu, and **wrap_content** for the **layout_height** drop-down menu.

The layout should look like the figure below. The phone_text element's **layout_width** drop-down menu in the Attributes pane is set to 134dp. You can optionally experiment with other width settings.



To look at the XML code for activity_main.xml, click the **Text** tab.

The Spinner should have the following attributes:

```
<Spinner
        android:id="@+id/label_spinner"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginRight="8dp"
        android:layout_marginLeft="260dp"
        android:layout_marginTop="24dp"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintLeft_toRightOf="@+id/phone_text"
        app:layout_constraintTop_toBottomOf="@+id/address_text" />
```

Be sure to add the android:layout_marginRight and android:layout_marginLeft attributes shown in the code snippet above to maintain compatibility with older versions of Android.

The phone_text element should now have the following attributes (after using the pack tool):

```
<EditText
    android:id="@+id/phone_text"
    android:layout_width="134dp"
    android:layout_height="wrap_content"
    android:layout_marginLeft="8dp"
    android:ems="10"
    android:hint="@string/enter_phone_hint"
    android:inputType="phone"
    app:layout_constraintBaseline_toBaselineOf="@+id/phone_label"
    app:layout_constraintStart_toEndOf="@+id/phone_label" />
```

Add code to activate the spinner and its listener

The choices for the Spinner are well-defined static strings such as "Home" and "Work," so you can use a text array defined in strings.xml to hold the values for it.

To activate the Spinner and its listener, implement the AdapterView.OnItemSelectedListener interface, which requires also adding the onItemSelected() and onNothingSelected() callback methods.

Open **strings.xml** and define the selectable values (**Home**, **Work**, **Mobile**, and **Other**) for the Spinner as the string array labels_array:

```
<string-array name="labels_array">
  <item>Home</item>
  <item>Work</item>
  <item>Mobile</item>
  <item>Others</item>
</string-array>
```

To define the selection callback for the Spinner, change your MainActivity class to implement the AdapterView.OnItemSelectedListener interface as shown:

```
public class MainActivity extends AppCompatActivity implements
            AdapterView.OnItemSelectedListener {
```

As you type **AdapterView.** in the statement above, Android Studio automatically imports the AdapterView widget. The reason why you need the AdapterView is because you need an adapter—specifically an ArrayAdapter—to assign the array to the Spinner.
An *adapter* connects your data—in this case, the array of spinner items—to the Spinner. You learn more about this pattern of using an adapter to connect data in another practical.
This line should appear in your block of import statements:

```
import android.widget.AdapterView;
```

After typing **OnItemSelectedListener** in the statement above, wait a few seconds for a red light bulb to appear in the left margin.

Click the light bulb and select **Implement methods**.

The onItemSelected() and onNothingSelected() methods, which are required for OnItemSelectedListener, should be highlighted, and the "Insert @Override" option should be selected. Click **OK**

This step automatically adds empty onItemSelected() and onNothingSelected() callback

methods to the bottom of the MainActivity class. Both methods use the parameter AdapterView<?>. The <?> is a Java type wildcard, enabling the method to be flexible enough to accept any type of AdapterView as an argument.

3.2.4 Instantiate a Spinner in the onCreate() method using the label_spinner element in the layout, and set its listener (spinner.setOnItemSelectedListener) in the onCreate() method, as shown in the following code snippet:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   // ... Rest of onCreate code ...
   // Create the spinner.
   Spinner spinner = findViewById(R.id.label_spinner);
   if (spinner != null) {
            spinner.setOnItemSelectedListener(this);
   }
   // Create ArrayAdapter using the string array and default spinner layout
```

Continuing to edit the onCreate() method, add a statement that creates the ArrayAdapter with the string array (labels_array) using the Android- supplied Spinner layout for each item (layout.simple_spinner_item):

```
// Create ArrayAdapter using the string array and default spinner layout.
ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(this,
                R.array.labels_array, android.R.layout.simple_spinner_item);
// Specify the layout to use when the list of choices appears.
```

The simple_spinner_item layout used in this step, and the simple_spinner_dropdown_item layout used in the next step, are the default predefined layouts provided by Android in the R.layout class. You should use these layouts unless you want to define your own layouts for the items in the Spinner and its appearance.

Specify the layout for the Spinner choices to be simple_spinner_dropdown_item, and then apply the adapter to the spinner.

```
// Specify the layout to use when the list of choices appears.
adapter.setDropDownViewResource
                        (android.R.layout.simple_spinner_dropdown_item)
;
// Apply the adapter to the spinner.
if (spinner != null) {
     spinner.setAdapter(adapter);
```

```
}
// ... End of onCreate code ...
```

Add code to response to spinner selection

When the user selects an item in the Spinner, the Spinner receives an on-item-selected event. To handle this event, you already implemented the AdapterView.OnItemSelectedListener interface in the previous step, adding empty onItemSelected() and onNothingSelected() callback methods.

In this step you fill in the code for the onItemSelected() method to retrieve the selected item in the Spinner, using getItemAtPosition(), and assign the item to the spinnerLabel variable of String data type:

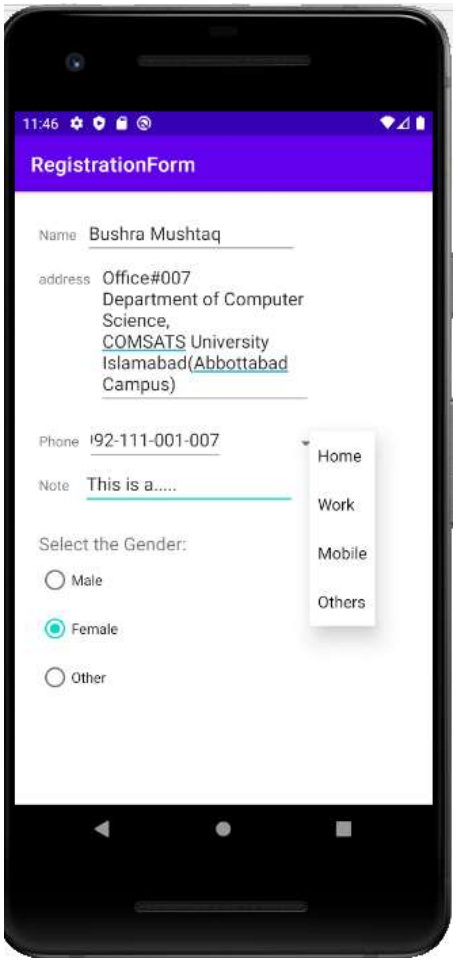
Add code to the empty onItemSelected() callback method, as shown below, to retrieve the user's selected item using getItemAtPosition(), and assign it to spinnerLabel. You can also add a call to the displayToast() method you already added to MainActivity

```
public void onItemSelected(AdapterView<?> adapterView, View view, int
        i, long l) {
  String spinnerLabel = adapterView.getItemAtPosition(i).toString();
  displayToast(spinnerLabel);
}
```

There is no need to add code to the empty onNothingSelected() callback method for this example

Run the app.

The Spinner appears next to the phone entry field and shows the first choice (Home). Tapping the Spinner reveals all the choices, as shown on the left side of the figure below. Tapping a choice in the Spinner shows a Toast message with the choice, as shown on the right side of the figure



Exercises

Q1: Write code to perform an action directly from the keyboard by tapping a **Send** key, such as for dialing a phone number:

Q2. Add the checkboxes in the registration to ask about computer skills and attach the event listener with each option selected to show a toast message

LAB 04: Activity, Intent, and Intent filters

Objective

- Learning and Practice with Activities in Android
- Learning and utilizing different Intents in different situations
- Understanding of the Intent Filters with some examples

Scope

The Activity class is a crucial component of an Android app, and the way activities are launched and put together is a fundamental part of the platform's application model. Unlike programming paradigms in which apps are launched with a main() method, the Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle.

This lab introduces the concept of activities, and then provides some lightweight guidance about how to work with them.

Useful Concepts

Activity in Android:

- Serves as the entry point for an app's interaction with the user. Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle.
- Provides the window in which the app draws its UI.
- Generally, one activity implements one screen in an app. Most apps contain multiple screens, which means they comprise multiple activities.
- One activity is specified as the *main activity* as the first screen to appear when launch the app.
- Activity can start another activity in order to perform different actions.
- To use activities in your app, you must register information about them in the app's manifest, and you must manage activity lifecycles appropriately.

Activity stack

An android application usually contains lots of activities. We can navigate over these activities. When a user starts an activity, android OS pushes that into a stack. If user starts another activity then first activity goes down and newly started activity is added to the top of the stack. When user pushes back button, then top activity is destroyed and the activity which is below the top activity is resumed.

Configuring the manifest

To use activities, you must declare the activities, and certain of their attributes, in the manifest. For example:

```
<manifest ... >
  <application ... >
      <activity android:name=".ExampleActivity" />
      ...
  </application ... >
  ...
</manifest >
```

- The only required attribute for this element is android:name
- **Note:** After you publish your app, you should not change activity names.

Intent Filters

- Provides the ability to launch an activity based not only on an explicit request, but also an implicit one.
- Implicit request tells the system to "Start a Send Email screen in any activity that can do the job." When the system UI asks a user which app to use in performing a task, that's an intent filter at work.
- The definition of this element includes an <action> element and, optionally, a <category> element and/or a <data> element.
- Activities that you don't want to make available to other applications should have no intent filters, and you can start them yourself using explicit intents.

For example, the following code snippet shows how to configure an activity that sends text data, and receives requests from other activities to do so:

```
<activity android:name=".ExampleActivity" android:icon="@drawable/app_icon">
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="text/plain" />
    </intent-filter>
</activity>
```

The following code snippet shows how to call the activity described above:

```
// Create the text message with a string
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.setType("text/plain");
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
// Start the activity
startActivity(sendIntent);
```

Declare permissions

Use the manifest's <activity> tag to control which apps can start a particular activity. A parent activity cannot launch a child activity unless both activities have the same permissions in their manifest.

For example, if your app wants to use a hypothetical app named SocialApp to share a post on social media, SocialApp itself must define the permission that an app calling it must have:

```
<manifest>
<activity android:name="...."
   android:permission=”com.google.socialapp.permission.SHARE_POST”

/>
```

Then, to be allowed to call SocialApp, your app must match the permission set in SocialApp's manifest:

```
<manifest>
   <uses-permission android:name="com.google.socialapp.permission.SHARE_POST" />
</manifest>
```

Types of Intents

Intents have been classified into two types. They are

- Explicit Intents

- Implicit Intents

Explicit Intents:

Explicit intent is called for internal communication of an application. It is being invoked by mentioning the target component name. The component or activity can be specified which should be active on receiving the intent. For this reason mostly it is used for intra application communication. The following code describes the way of creating an explicit intent. While creating explicit intent, the target activity is specified so that the android system will invoke the activity.

Now the addition of two numbers program has been modified in order to use intent. In the first activity, inside the onClick() method intent object is created. While creating intent object the target component is specified that should be get activated. Here the MainActivity2 is the target activity that will be invoked by this intent. Since intents are partof the core message system of android, data is passed to the second activity

_MainActivity2‘. i.e. input numbers are read in the first activity, when the button is clicked intent object is created and the result is passed through the intent object. It will activate the target activity. We will get the result in the second activity. The code for second activity is given below.

The second activity is activated by the intent object. Also the second activity receives data or message from the first activity through intent object. The Bundle class is used to save the activity state. So data from the intent object is read with the help of Bundle. The getIntent() method will retrieve data from the intent object. Intent can have the following information.

a) **Component name** - The name of the component to start. This is optional, but it's the criticalpiece of information that makes an intent explicit, meaning that the intent should be delivered

only to the app component defined by the component name. Without a component name,the intent is implicit and the system decides which component should receive the intent based on the other intent information (such as the action, data, and category).

b) **Action** - A string that specifies the generic action to perform (such as view or pick). Somecommon actions for starting an activityare:

• ACTION_VIEW - This action is used in an intent with startActivity() when it has

some information that an activity can show to the user, such as a photo to view in a gallery app, or an address to view in a map app.

• ACTION_SEND - Also known as the "share" intent, this intent is used with startActivity() when some data that the user can share through another app, such as an email app or socialsharing app.

c) **Data** - The URI (a Uri object) that references the data to be acted on and/or the MIME type ofthat data. The type of data supplied is generally dictated by the intent's action. For example, if the action is ACTION_EDIT, the data should contain the URI of the document toedit.

d) **Categories** - A string containing additional information about the kind of component thatshould handle the intent.

• **Extras** - Key-value pairs that carry additional information required to accomplish therequested action.

• **Flags** - Flags defined in the Intent class that function as metadata for the intent. Theflagsmay instruct the Android system how to launch an activity.

Implicit Intents:

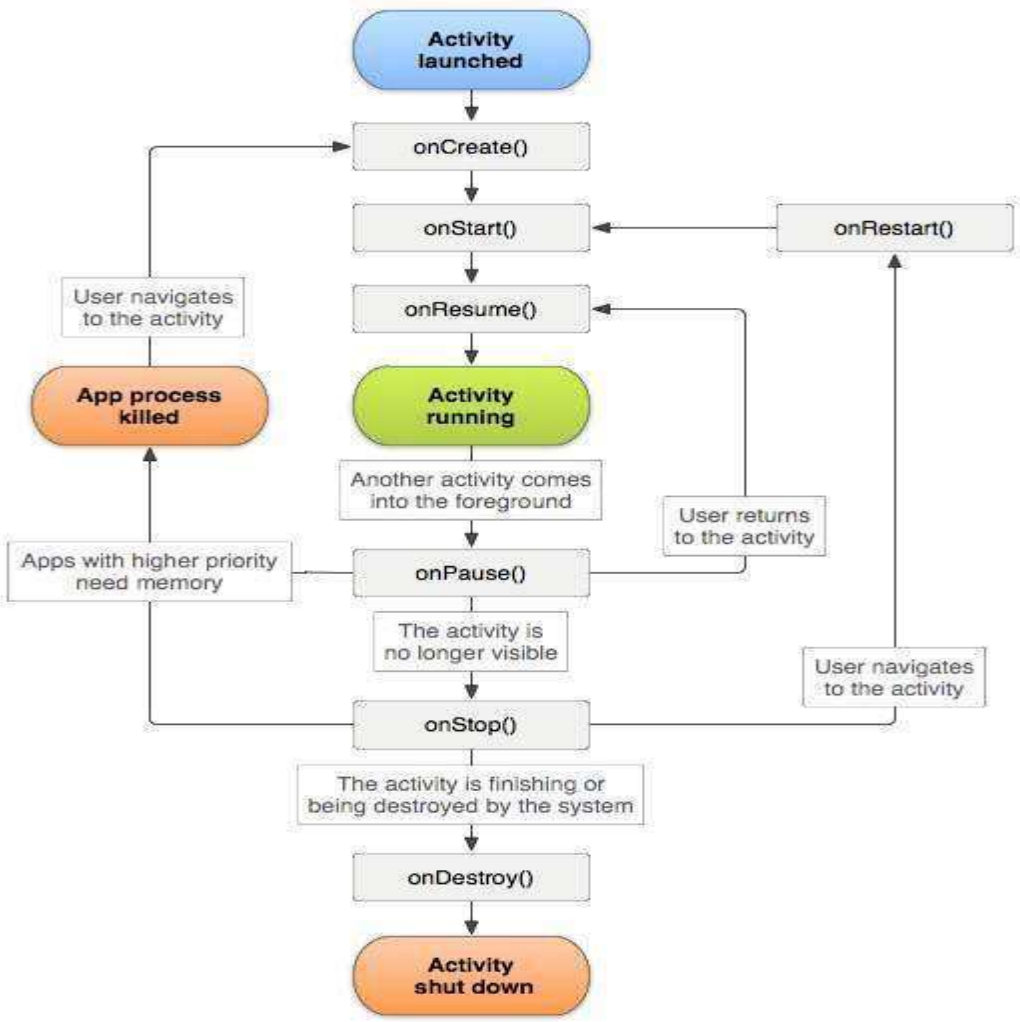
When implicit intents are used, a message is sent to the android system to find an appropriate activity to respond to the intent. For example, to share a video, we can use intent. The video can be shared through any type of external application. To do this we canuse intent. When intent is received, android system will invoke an activity which is capable of sending video. If there is more than one activity is capable of receiving the intent, the android system will present a chooser so that the user can select which activity or application should handle it. The following sample code shows the use of implicit intents.

Managing the activity lifecycle

Activity goes through a number of states during its lifetime. You use a series of callbacks to handle transitions between states.

Methods for Activity Life Cycle

When an activity transitions from one state to another, set of callback methods are invoked for each state. How a set of call back methods for each state of an activity is invoked in its life cycle is given in the below picture.



1. **onCreate() : Mandatory** and fires when the system creates your activity.
    a. Perform initialization work here.
    b. You must call setContentView() to define the layout for the activity's user interface.
2. **onStart() :** Activity becomes visible to the user.
    a. Contains final preparations for coming to the foreground and becoming interactive.
3. **onResume():** Invokes just before the activity starts interacting with the user.

a. At this point, the activity is at the top of the activity stack,

b. Captures user input.

c. Most of an app's core functionality is implemented in the onResume() method.

4. **onPause():** Activity loses focus and enters a Paused state.

   a. Occurs when, the user taps the Back or Recent button.

   b. It technically means your activity is still partially visible, but most often is an indication that the user is leaving the activity, and the activity will soon enter the Stopped or Resumed state.

   c. You should not use onPause() to save application or user data, make network calls, or execute database transactions.

5. **onStop():** Activity is no longer visible to the user. This may happen because:

   a. Activity is being destroyed,

   b. A new activity is starting, or an existing activity is entering a Resumed state and

   c. Covering the stopped activity.

6. **onRestart():** Restores the state of the activity from the time that it was stopped.

7. **onDestroy():** Invokes this callback before an activity is destroyed.

   a. Usually implemented to ensure that all of an activity's resources are released when the activity, or the process containing it, is destroyed.

Lab Tasks

Activity 1

In this activity you are going to learn the configuration of activity in the manifest file. Create a default application with a basic activity and

1. Run it to view the output on your emulator or attached device.
2. Remove the activity declaration from the manifest file and try to run it again. You will notice that declaration is must.
3. Explore all the mandatory and non-mandatory attribute of the activity tag in the mangiest file.

Activity 2

In this activity you are going to learn of launching an activity into another application protected by permissions.

1. Create two application app1 and app2 with two different activities app1_activity1 and app2_activity2.

2. Ensure to try running the example with:
3. App 1 tries to launch activity 2 in App 2 without any permission in Manifest of both for this.
4. App2 defines “APP2_ACTIVITY2_USAGE”” permission with “android:permission” attribute, but App 1 has no usage of it in its Manifest and then try to run it.
5. App2 defines “APP2_ACTIVITY2_USAGE”” permission with “android:permission” attribute And App 1 has usage permission declaration of it in its Manifest and then try to run it.

Activity 3

Run few examples to debug the different stats of the activity lifecycle from the following GitHub repository.

https://github.com/Ibtisam/Android-Lifecycle-Example

Exercises

For your semester project try to finalize these:

1. What are the functionalities that you will perform in different lifecycle callback for different screens in your project? Mention at least two examples of each lifecycle callback.
2. Implement the identified behavior in 1 using your semester project main code.
3. Create a demonstration video for your work.

Assignment Deliverables

Share a video demonstration of your home exercises from cloud storage with your class teacher.

LAB 05: UI Layouts and Advanced UI Components

Objective

In this lab we will be learning how to use and extend the Android user interface library. In a number of ways it is very similar to the Java Swing library, and in perhaps just as many ways it is different.

Scope

At the end of this lab students will be expected to know:

- What Views, View Groups, Layouts, and Widgets are and how they relate to each other.
- How to declare layouts dynamically at runtime.
- How to reference resources in code and from other resource layout files.
- How to use Events and Event Listeners

Useful Concepts

ANDROID APPLICATIONS BASICS

A single screen of UI that appears in your app –the fundamental units of GUI in an Android app are:

- view: items that appear onscreen in an activity
- –widget: GUI control such as a button or text field
- –layout: invisible container that manages positions/sizes of widgets
- –event: action that occurs when user interacts with widgets –e.g. clicks, typing, scrolling.

Designing a user interface/ Layout

- Open XML file for your layout (e.g. activity_main.xml)
- Drag widgets from left Palette to the preview image
- Set their properties in lower-right Properties panel

Android-widgets/views

Views are also referred to as widgets. Typical examples include standard items such as the Button, CheckBox, ProgressBarand TextViewclasses. Views have an integer id associated with them. These

ids are assigned in the layout XML files. Define a Button in the layout file and assign it a unique ID.

Layout

Layouts are described in XML and mirrored in Java code. Following are the different layouts available in android:

Android Layout Types

There are number of Layouts provided by Android which you will use in almost all the Android applications to provide different view, look and feel.

Sr.No	Layout & Description
1	Linear Layout LinearLayout is a view group that aligns all children in a single direction, vertically or horizontally.
2	Relative Layout RelativeLayout is a view group that displays child views in relative positions.
3	Table Layout TableLayout is a view that groups views into rows and columns.
4	Absolute Layout AbsoluteLayout enables you to specify the exact location of its children.
5	Frame Layout The FrameLayout is a placeholder on screen that you can use to display a single view.
6	List View ListView is a view group that displays a list of scrollable items.
7	Grid View GridView is a ViewGroup that displays items in a two-dimensional, scrollable grid.

Layout Attributes

Each layout has a set of attributes which define the visual properties of that layout. There are few common attributes among all the layouts and their are other attributes which are specific to that layout. Following are common attributes and will be applied to all the layouts:

Sr.No	Attribute & Description
1	**android:id** This is the ID which uniquely identifies the view.
2	**android:layout_width** This is the width of the layout.
3	**android:layout_height** This is the height of the layout
4	**android:layout_marginTop** This is the extra space on the top side of the layout.
5	**android:layout_marginBottom** This is the extra space on the bottom side of the layout.
6	**android:layout_marginLeft** This is the extra space on the left side of the layout.
7	**android:layout_marginRight** This is the extra space on the right side of the layout.
8	**android:layout_gravity** This specifies how child Views are positioned.
9	**android:layout_weight** This specifies how much of the extra space in the layout should be allocated to the View.
10	**android:layout_x** This specifies the x-coordinate of the layout.
11	**android:layout_y**

	This specifies the y-coordinate of the layout.
12	**android:layout_width** This is the width of the layout.
13	**android:paddingLeft** This is the left padding filled for the layout.
14	**android:paddingRight** This is the right padding filled for the layout.
15	**android:paddingTop** This is the top padding filled for the layout.
16	**android:paddingBottom** This is the bottom padding filled for the layout.

Here width and height are the dimension of the layout/view which can be specified in terms of dp (Density-independent Pixels), sp (Scale-independent Pixels), pt (Points which is 1/72 of an inch), px(Pixels), mm (Millimeters) and finally in (inches).

You can specify width and height with exact measurements but more often, you will use one of these constants to set the width or height −

- android:layout_width=wrap_content tells your view to size itself to the dimensions required by its content.
- android:layout_width=fill_parent tells your view to become as big as its parent view.

Gravity attribute plays important role in positioning the view object and it can take one or more (separated by '|') of the following constant values.

Constant	Value	Description
top	0x30	Push object to the top of its container, not changing its size.
bottom	0x50	Push object to the bottom of its container, not changing its size.

left	0x03	Push object to the left of its container, not changing its size.
right	0x05	Push object to the right of its container, not changing its size.
center_vertical	0x10	Place object in the vertical center of its container, not changing its size.
fill_vertical	0x70	Grow the vertical size of the object if needed so it completely fills its container.
center_horizontal	0x01	Place object in the horizontal center of its container, not changing its size.
fill_horizontal	0x07	Grow the horizontal size of the object if needed so it completely fills its container.
center	0x11	Place the object in the center of its container in both the vertical and horizontal axis, not changing its size.
fill	0x77	Grow the horizontal and vertical size of the object if needed so it completely fills its container.
clip_vertical	0x80	Additional option that can be set to have the top and/or bottom edges of the child clipped to its container's bounds. The clip will be based on the vertical gravity: a top gravity will clip the bottom edge, a bottom gravity will clip the top edge, and neither will clip both edges.
clip_horizontal	0x08	Additional option that can be set to have the left and/or right edges of the child clipped to its container's bounds. The clip will be based on the horizontal gravity: a left gravity will clip the right edge, a right gravity will clip the left edge, and neither will clip both edges.
start	0x00800003	Push object to the beginning of its container, not changing its size.

end	0x00800005	Push object to the end of its container, not changing its size.

Lab Tasks

Activity 1

Run the following code examples:

- https://github.com/Ibtisam/Using-WebView
- https://github.com/Ibtisam/Using-RecyclerView

Activity 2

You need to update both examples of the first activity and add your unique updates such that it will not be a duplicate to your class fellows.

Exercises

Find at least two other advanced components and tell their usages in your semester project.

Assignment Deliverables

Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teacher.

LAB 06: Sessional 1 Exam

Purpose

The purpose of this lab is to conduct the first sessional exam based on the activitiesconducted so far.

Tasks

The tasks will be decided by the respective course instructor/lab tutor.

LAB 07: Fragments

Objective

- Fragment manager
- Fragment transactions
- Animate transitions between fragments
- Fragment lifecycle
- Saving state with fragments
- Communicate between fragments and activities
- Debug your fragments
- Test your fragments

Scope

A Fragment represents a reusable portion of your app's UI. A fragment defines and manages its own layout, has its own lifecycle, and can handle its own input events. Fragments cannot live on their own--they must be *hosted* by an activity or another fragment. The fragment's view hierarchy becomes part of, or *attaches to*, the host's view hierarchy.

Fragments introduce modularity and reusability into your activity's UI by allowing you to divide the UI into discrete chunks. Activities are an ideal place to put global elements around your app's user interface, such as a navigation drawer. Conversely, fragments are better suited to define and manage the UI of a single screen or portion of a screen.

We are going to learn about the basics and some advanced skills about Fragments in this lab.

Useful Concepts

Create a fragment

To create a fragment, extend the AndroidX Fragment class, and override its methods to insert your app logic, similar to the way you would create an Activity class. To create a minimal fragment that defines its own layout, provide your fragment's layout resource to the base constructor, as shown in the following example:

```
class ExampleFragment extends Fragment {
    public ExampleFragment() {
        super(R.layout.example_fragment);
    }
}
```

Add a fragment to an activity

Generally, your fragment must be embedded within an AndroidX FragmentActivity to contribute a portion of UI to that activity's layout. FragmentActivity is the base class for AppCompatActivity, so if you're already subclassing AppCompatActivity to provide backward compatibility in your app, then you do not need to change your activity base class.

You can add your fragment to the activity's view hierarchy either by defining the fragment in your activity's layout file or by defining a fragment container in your activity's layout file and then programmatically adding the fragment from within your activity.

Add a fragment via XML

To declaratively add a fragment to your activity layout's XML, use a FragmentContainerView element.

Here's an example activity layout containing a single FragmentContainerView:

```
<!-- res/layout/example_activity.xml -->
<androidx.fragment.app.FragmentContainerView
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:id="@+id/fragment_container_view"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:name="com.example.ExampleFragment" />
```

Add a fragment programmatically

To programmatically add a fragment to your activity's layout, the layout should include a FragmentContainerView to serve as a fragment container, as shown in the following example:

```
<!-- res/layout/example_activity.xml -->
<androidx.fragment.app.FragmentContainerView
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:id="@+id/fragment_container_view"
   android:layout_width="match_parent"
   android:layout_height="match_parent" />
```

Unlike the XML approach, the android:name attribute isn't used on the FragmentContainerView here, so no specific fragment is automatically instantiated. Instead, a FragmentTransaction is used to instantiate a fragment and add it to the activity's layout.

```
public class ExampleActivity extends AppCompatActivity {
    public ExampleActivity() {
        super(R.layout.example_activity);
    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (savedInstanceState == null) {
            getSupportFragmentManager().beginTransaction()
                .setReorderingAllowed(true)
```

```
                    .add(R.id.fragment_container_view, ExampleFragment.class, null)
                    .commit();
            }
        }
    }
```

Navigate between fragments using animations

The Fragment API provides two ways to use motion effects and transformations to visually connect fragments during navigation. One of these is the Animation Framework, which uses both Animation and Animator. The other is the Transition Framework, which includes shared element transitions.

For details on Fragment animation follow Navigate between fragments using animations | Android Developers

Lab Tasks

Activity 1

Run the following Fragment examples:

- https://github.com/Ibtisam/SimpleVideoPlayerUsingFragments
- https://github.com/Ibtisam/SimpleFragmentCommUsingViewModel
- https://github.com/Ibtisam/ProgrammaticFragmentExample
- https://github.com/Ibtisam/ProgrammaticFragmentExample

Activity 2

Suggest improvements in the examples of activity 1 to let other understand them with more ease.

Exercises

In your semester project, Identify the screen where you feel Fragments need to be utilized and which type of Fragment you will use. Give examples of at least two such situation with code.

Assignment Deliverables

Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teacher.

LAB 08: Application Security and Permissions

Objective

- Work with data more securely Part of Android Jetpack.
- Understanding the best practices for android app security
- Running examples to understand them
- Exploring the build in support for security

The Security library provides an implementation of the security best practices related to reading and writing data at rest, as well as key creation and verification.

The library uses the builder pattern to provide safe default settings for the following security levels:

- **Strong security that balances great encryption and good performance.** This level of security is appropriate for consumer apps, such as banking and chat apps, as well as enterprise apps that perform certificate revocation checking.
- **Maximum security.** This level of security is appropriate for apps that require a hardware-backed keystore and user presence for providing key access.

This guide shows how to work with the Security library's recommended security configurations, as well as how to read and write encrypted data that's stored in files and shared preferences easily and safely.

Scope

When you safeguard the data that you exchange between your app and other apps, or between your app and a website, you improve your app's stability and protect the data that you send and receive. How to achieve this will be covered in the lab via useful concepts and then running different examples to support our objectives for both application and data security in android application.

Useful Concepts

Security tips

> Android has built-in security features that significantly reduce the frequency and impact of application security issues. The system is designed so that you can typically build your apps with the default system and file permissions and avoid difficult decisions about security.

The following core security features help you build secure apps:

- The Android Application Sandbox, which isolates your app data and code execution from other apps.
- An application framework with robust implementations of common security functionality such as cryptography, permissions, and secure IPC.
- Technologies like ASLR, NX, ProPolice, safe_iop, OpenBSD dlmalloc, OpenBSD calloc, and Linux mmap_min_addr to mitigate risks associated with common memory management errors.
- An encrypted file system that can be enabled to protect data on lost or stolen devices.
- User-granted permissions to restrict access to system features and user data.
- Application-defined permissions to control application data on a per-app basis.

It is important that you be familiar with the Android security best practices in this document. Following these practices as general coding habits reduces the likelihood of inadvertently introducing security issues that adversely affect your users.

Use implicit intents and non-exported content providers

Show an app chooser

If an implicit intent can launch at least two possible apps on a user's device, explicitly show an app chooser. This interaction strategy allows users to transfer sensitive information to an app that they trust.

```
Intent intent = new Intent(Intent.ACTION_SEND);
List<ResolveInfo> possibleActivitiesList = getPackageManager()
        .queryIntentActivities(intent, PackageManager.MATCH_ALL);

// Verify that an activity in at least two apps on the user's device
// can handle the intent. Otherwise, start the intent only if an app
// on the user's device can handle the intent.
if (possibleActivitiesList.size() > 1) {

    // Create intent to show chooser.
    // Title is something similar to "Share this photo with".

    String title = getResources().getString(R.string.chooser_title);
    Intent chooser = Intent.createChooser(intent, title);
    startActivity(chooser);
} else if (intent.resolveActivity(getPackageManager()) != null) {
    startActivity(intent);
}
```

Apply signature-based permissions

When sharing data between two apps that you control or own, use signature-based permissions. These permissions don't require user confirmation and instead check

that the apps accessing the data are signed using the same signing key. Therefore, these permissions offer a more streamlined, secure user experience.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.myapp">
   <permission android:name="my_custom_permission_name"
         android:protectionLevel="signature" />
```

Disallow access to your app's content providers

Unless you intend to send data from your app to a different app that you don't own, you should explicitly disallow other developers' apps from accessing the ContentProvider objects that your app contains. This setting is particularly important if your app can be installed on devices running Android 4.1.1 (API level 16) or lower, as the android:exported attribute of the <provider> element is true by default on those versions of Android.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.myapp">
   <application ... >
      <provider
         android:name="android.support.v4.content.FileProvider"
         android:authorities="com.example.myapp.fileprovider"
         ...
         android:exported="false">
         <!-- Place child elements of <provider> here. -->
      </provider>
      ...
   </application>
</manifest>
```

Ask for credentials before showing sensitive information

When requesting credentials from users so that they can access sensitive information or premium content in your app, ask for either a PIN/password/pattern or a biometric credential, such as using face recognition or fingerprint recognition.

To learn more about how to request biometric credentials, see the guide about biometric authentication.

Apply network security measures

The following sections describe how you can improve your app's network security.

Use SSL traffic

If your app communicates with a web server that has a certificate issued by a well-known, trusted CA, the HTTPS request is very simple:

```
URL url = new URL("https://www.google.com");
HttpsURLConnection urlConnection = (HttpsURLConnection)
url.openConnection();
urlConnection.connect();
InputStream in = urlConnection.getInputStream();
```

Lab Tasks

Activity 1

Watch and self-assess yourself on subexpert against the following lecture related to permissions in android.

- https://www.subexpert.com/CourseLectures/OfSubject/Android-Labs

Activity 2

Review the remaining best practices for security on
App security best practices | Android Developers

Sort all practices in order of their severity.

Exercises

Which security measures will you implement in your semester project. Give code examples of at least two measures.

Assignment Deliverables

Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teacher.

LAB 9: Data Storage & Content Providers

Objective

The objective of this lab is to understand how you can use multiple storage options provided by the Android to manage your data.

File based Storage:

Android uses a file system that's like disk-based file systems on other platforms. The system provides several options for you to save your app data:

App-specific storage: Store files that are meant for your app's use only, either in dedicated directories within an internal storage volume or different dedicated directories within external storage. Use the directories within internal storage to save sensitive information that other apps shouldn't access.

Shared storage: Store files that your app intends to share with other apps, including media, documents, and other files.

Preferences: Store private, primitive data in key-value pairs.

Databases: Store structured data in a private database using the Room persistence library.

Cloud based Storage:

Other than the file storage options android provides APIs to use cloud based storage using Firebase. Firebase offers two cloud-based, client-accessible database solutions that support real time data syncing:

Realtime: Database is Firebase's original database. It's an efficient, low-latency solution for mobile apps that require synced states across clients in realtime.

Cloud Firestore: is Firebase's newest database for mobile app development. It builds on the successes of the Realtime Database with a new, more intuitive data model. Cloud Firestore also features richer, faster queries and scales further than the Realtime Database.

Content Providers:

A content provider manages access to a central repository of data. A provider is part of an Android application, which often provides its own UI for working with the data.

Typically, you work with content providers in one of two scenarios; you may want to implement code to access an existing content provider in another application, or you may want to create a new content provider in your application to share data with other applications.

Scope

The scope of this lab activity is the student's ability to:

- Understand what storage options are suitable in which scenarios.
- How to use file-based storage options.
- How to use Firebase Dashboard to manage real-time databases.
- How to use content providers.

Useful Concepts

Permission and access to external storage:

Android defines the following storage-related permissions: READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE, and MANAGE_EXTERNAL_STORAGE.

App-specific storage usage guidelines:

The system provides the following locations for storing such app-specific files:

Internal storage directories: These directories include both a dedicated location for storing persistent files, and another location for storing cache data.

External storage directories: These directories include both a dedicated location for storing persistent files, and another location for storing cache data.

Access from Internal Storage:

For each app, the system provides directories within internal storage where an app can organize its files. One directory is designed for your app's persistent files, and another contains your app's cached files. Your app doesn't require any system permissions to read and write to files in these directories.

Access persistent files:

Your app's ordinary, persistent files reside in a directory that you can access using the filesDir property of a context object. The framework provides several methods to help you access and store files in this directory.

Access and store files:

You can use the File API to access and store files.

To help maintain your app's performance, don't open and close the same file multiple times.

The following code snippet demonstrates how to use the File API:

```
File file = new File(context.getFilesDir(), filename);
```

Store a file using a stream:

As an alternative to using the File API, you can call openFileOutput() to get a FileOutputStream that writes to a file within the filesDir directory.

The following code snippet shows how to write some text to a file:

```
String filename = "myfile";
String fileContents = "Hello world!";
try (FileOutputStream fos = context.openFileOutput(filename, Context.MODE_PRIVATE)) {
    fos.write(fileContents.toByteArray());
}
```

To allow other apps to access files stored in this directory within internal storage, use a FileProvider with the FLAG_GRANT_READ_URI_PERMISSION attribute.

Access a file using stream:

To read a file as a stream, use openFileInput():

```
FileInputStream fis = context.openFileInput(filename);
InputStreamReader inputStreamReader =
        new InputStreamReader(fis, StandardCharsets.UTF_8);
StringBuilder stringBuilder = new StringBuilder();
try (BufferedReader reader = new BufferedReader(inputStreamReader)) {
    String line = reader.readLine();
    while (line != null) {
        stringBuilder.append(line).append('\n');
        line = reader.readLine();
    }
} catch (IOException e) {
    // Error occurred when opening raw file for reading.
} finally {
    String contents = stringBuilder.toString();
}
```

View list of files:

You can get an array containing the names of all files within the filesDir directory by calling fileList(), as shown in the following code snippet:

```
Array<String> files = context.fileList();
```

Create nested directories:

You can also create nested directories, or open an inner directory, by calling getDir() in Kotlin-based code or by passing the root directory and a new directory name into a File constructor in Java-based code:

```
File directory = context.getFilesDir();
File file = new File(directory, filename);
```

Create cache files:

To create a cached file, call File.createTempFile():

```
File.createTempFile(filename, null, context.getCacheDir());
```

Your app accesses a file in this directory using the cacheDir property of a context object and the File API:

```
File cacheFile = new File(context.getCacheDir(), filename);
```

Remove cache files:

To remove a file from the cache directory within internal storage, use one of the following methods:

The delete() method on a File object that represents the file:

```
cacheFile.delete();
```

The deleteFile() method of the app's context, passing in the name of the file:

```
context.deleteFile(cacheFileName);
```

Access from external storage:

If internal storage doesn't provide enough space to store app-specific files, consider using external storage instead. The system provides directories within external storage where an app can organize files that provide value to the user only within your app. One directory is designed for your app's persistent files, and another contains your app's cached files.

Verify that storage is available:

You can query the volume's state by calling Environment.getExternalStorageState(). If the returned state is MEDIA_MOUNTED, then you can read and write app-specific files within external storage. If it's MEDIA_MOUNTED_READ_ONLY, you can only read these files.

For example, the following methods are useful to determine the storage availability:

```
// Checks if a volume containing external storage is available
// for read and write.
private boolean isExternalStorageWritable() {
    return Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED);
}

// Checks if a volume containing external storage is available to at least read.
private boolean isExternalStorageReadable() {
     return Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED) ||
            Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED_READ_ONLY);
}
```

Select a physical storage location:

To access the different locations, call ContextCompat.getExternalFilesDirs().

```
File[] externalStorageVolumes =
        ContextCompat.getExternalFilesDirs(getApplicationContext(), null);
File primaryExternalStorage = externalStorageVolumes[0];
```

Access persistent files:

The following code snippet demonstrates how to call getExternalFilesDir():

```
File appSpecificExternalDir = new File(context.getExternalFilesDir(null), filename);
```

Create cache files:

To add an app-specific file to the cache within external storage, get a reference to the externalCacheDir:

```
File externalCacheFile = new File(context.getExternalCacheDir(), filename);
```

Remove cache files:

To remove a file from the external cache directory, use the delete() method on a File object that represents the file:

```
externalCacheFile.delete();
```

Media Content:

If your app works with media files that provide value to the user only within your app, it's best to store them in app-specific directories within external storage, as demonstrated in the following code snippet:

```
@Nullable
File getAppSpecificAlbumStorageDir(Context context, String albumName) {
    // Get the pictures directory that's inside the app-specific directory on
    // external storage.
    File file = new File(context.getExternalFilesDir(
            Environment.DIRECTORY_PICTURES), albumName);
    if (file == null || !file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

Example Codes:

Ibtisam/InternalStorageExample (github.com)

Ibtisam/ExternalStorageExample (github.com)

Preferences:

Interface for accessing and modifying preference data returned by Context.getSharedPreferences(String, int). For any particular set of preferences, there is a single instance of this class that all clients share.

If you have a relatively small collection of key-values that you'd like to save, you should use the SharedPreferences APIs. A SharedPreferences object points to a file containing key-value pairs and provides simple methods to read and write them. Each SharedPreferences file is managed by the framework and can be private or shared.

Get a handle to shared preferences:

You can create a new shared preference file or access an existing one by calling one of these methods: getSharedPreferences() OR getPreferences()

```
Context context = getActivity();
SharedPreferences sharedPref = context.getSharedPreferences(
        getString(R.string.preference_file_key), Context.MODE_PRIVATE);
```

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
```

Write to shared preferences:

To write to a shared preferences file, create a SharedPreferences.Editor by calling edit() on your SharedPreferences.

Pass the keys and values you want to write with methods such as putInt() and putString(). Then call apply() or commit() to save the changes. For example:

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.saved_high_score_key), newHighScore);
editor.apply();
```

Read from shared preferences:

To retrieve values from a shared preferences file, call methods such as getInt() and getString(), providing the key for the value you want, and optionally a default value to return if the key isn't present. For example:

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
int defaultValue = getResources().getInteger(R.integer.saved_high_score_default_key);
int highScore = sharedPref.getInt(getString(R.string.saved_high_score_key), defaultValue);
```

Example Code:

Ibtisam/SharedPreferencesExample (github.com)

Databases:

Apps that handle non-trivial amounts of structured data can benefit greatly from persisting that data locally. The most common use case is to cache relevant pieces of data so that when the device cannot access the network, the user can still browse that content while they are offline.

Save data in local database using Room API:

The Room persistence library provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite.

Setup:

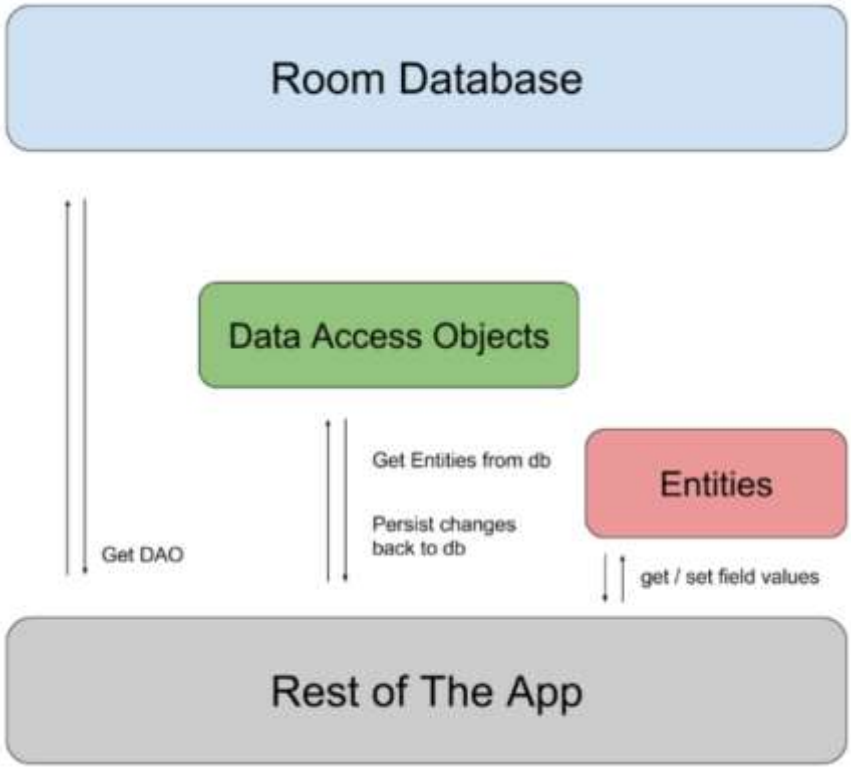
To use Room in your app, add the following dependencies to your app's build.gradle file:

```
dependencies {
    def room_version = "2.3.0"

    implementation "androidx.room:room-runtime:$room_version"
    annotationProcessor "androidx.room:room-compiler:$room_version"

    // optional - RxJava2 support for Room
    implementation "androidx.room:room-rxjava2:$room_version"

    // optional - RxJava3 support for Room
    implementation "androidx.room:room-rxjava3:$room_version"

    // optional - Guava support for Room, including Optional and ListenableFuture
    implementation "androidx.room:room-guava:$room_version"

    // optional - Test helpers
    testImplementation "androidx.room:room-testing:$room_version"

    // optional - Paging 3 Integration
    implementation "androidx.room:room-paging:2.4.0-beta01"
}
```

Primary Components:

There are three major components in Room:

- The database class that holds the database and serves as the main access point for the underlying connection to your app's persisted data.
- Data entities that represent tables in your app's database.
- Data access objects (DAOs) that provide methods that your app can use to query, update, insert, and delete data in the database.



Sample Implementation:

This section presents an example implementation of a Room database with a single data entity and a single DAO.

Data Entry:

The following code defines a User data entity. Each instance of User represents a row in a user table in the app's database.

```
@Entity
public class User {
    @PrimaryKey
    public int uid;

    @ColumnInfo(name = "first_name")
    public String firstName;

    @ColumnInfo(name = "last_name")
    public String lastName;
}
```

Data Access Object (DAO):

The following code defines a DAO called UserDao. UserDao provides the methods that the rest of the app uses to interact with data in the user table.

```
@Dao
public interface UserDao {
    @Query("SELECT * FROM user")
    List<User> getAll();

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    List<User> loadAllByIds(int[] userIds);

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +
           "last_name LIKE :last LIMIT 1")
    User findByName(String first, String last);

    @Insert
    void insertAll(User... users);

    @Delete
    void delete(User user);
}
```

Database:

The following code defines an AppDatabase class to hold the database. AppDatabase defines the database configuration and serves as the app's main access point to the persisted data. The database class must satisfy the following conditions:

- The class must be annotated with a @Database annotation that includes an entities array that lists all of the data entities associated with the database.
- The class must be an abstract class that extends RoomDatabase.
- For each DAO class that is associated with the database, the database class must define an abstract method that has zero arguments and returns an instance of the DAO class.

```
@Database(entities = {User.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract UserDao userDao();
}
```

Usage:

After you have defined the data entity, the DAO, and the database object, you can use the following code to create an instance of the database:

```
AppDatabase db = Room.databaseBuilder(getApplicationContext(),
        AppDatabase.class, "database-name").build();
```

You can then use the abstract methods from the AppDatabase to get an instance of the DAO. In turn,

you can use the methods from the DAO instance to interact with the database:

```
UserDao userDao = db.userDao();
List<User> users = userDao.getAll();
```

Code Example:

Ibtisam/RoomAPIExample (github.com)

Cloud based Storage – Firebase:

Firebase provides several services for cloud-based storage like Realtime database and Firestore.

Add Firebase to your Android Project:

Adding Firebase to your app involves tasks both in the Firebase console and in your open Android project (for example, you download Firebase config files from the console, then move them into your Android project).

Step 1: Create a Firebase Project:

1. In the Firebase console, click Add project.
    a. To add Firebase resources to an existing Google Cloud project, enter its project name or select it from the dropdown menu.
    b. To create a new project, enter the desired project name. You can also optionally edit the project ID displayed below the project name.
2. If prompted, review and accept the Firebase terms.
3. Click Continue.
4. Click Create project (or Add Firebase, if you're using an existing Google Cloud project).

Firebase automatically provisions resources for your Firebase project. When the process completes, you'll be taken to the overview page for your Firebase project in the Firebase console.

Step 2: Register your app with Firebase:

To use Firebase in your Android app, you need to register your app with your Firebase project. Registering your app is often called "adding" your app to your project.

1. Go to the Firebase console.
2. In the center of the project overview page, click the Android icon ( ) or Add app to launch the setup workflow.
3. Enter your app's package name in the Android package name field.
4. Click Register

Step 3: Add a Firebase configuration file:

1. Add the Firebase Android configuration file to your app:
    a. Click **Download google-services.json** to obtain your Firebase Android config file (google-services.json).
    b. Move your config file into the module (app-level) directory of your app.

    Notes:

    - The Firebase config file contains unique, but non-secret identifiers for your project. To learn more about this config file, visit Understand Firebase Projects.
    - You can download your Firebase config file again at any time.
    - Make sure the config file name is not appended with additional characters, like (2).
2. To enable Firebase products in your app, add the google-services plugin to your Gradle files.
    a. In your root-level (project-level) Gradle file (build.gradle), add rules to include the Google Services Gradle plugin. Check that you have Google's Maven repository, as well.

```
buildscript {

  repositories {
    // Check that you have the following line (if not, add it):
    google()  // Google's Maven repository
  }

  dependencies {
    // ...

    // Add the following line:
    classpath 'com.google.gms:google-services:4.3.10'  // Google Services plugin
  }
}

allprojects {
  // ...

  repositories {
    // Check that you have the following line (if not, add it):
    google()  // Google's Maven repository
    // ...
  }
}
```

    b. In your module (app-level) Gradle file (usually app/build.gradle), apply the Google Services Gradle plugin:

```
apply plugin: 'com.android.application'
// Add the following line:
apply plugin: 'com.google.gms.google-services'  // Google Services plugin

android {
  // ...
}
```

Step 4: Add Firebase SDKs to your app:

1. Using the Firebase Android BoM, declare the dependencies for the Firebase products that you want to use in your app. Declare them in your module (app-level) Gradle file (usually app/build.gradle).

```
dependencies {
  // ...

  // Import the Firebase BoM
  implementation platform('com.google.firebase:firebase-bom:28.4.2')

  // When using the BoM, you don't specify versions in Firebase library dependencies

  // Declare the dependency for the Firebase SDK for Google Analytics
  implementation 'com.google.firebase:firebase-analytics'

  // Declare the dependencies for any other desired Firebase products
  // For example, declare the dependencies for Firebase Authentication and Cloud Firestore
  implementation 'com.google.firebase:firebase-auth'
  implementation 'com.google.firebase:firebase-firestore'
}
```

2. Sync your app to ensure that all dependencies have the necessary versions.

Create a Realtime Database:

1. Navigate to the Realtime Database section of the Firebase console. You'll be prompted to select an existing Firebase project. Follow the database creation workflow.
2. Select a starting mode for your Firebase Security Rules:
    a. Test Mode: Good for getting started with the mobile and web client libraries, but allows anyone to read and overwrite your data. After testing, make sure to review the Understand Firebase Realtime Database Rules section.
    b. Locked Mode: Denies all reads and writes from mobile and web clients. Your authenticated application servers can still access your database.
3. Choose a region for the database.
4. Click Done.

When you enable Realtime Database, it also enables the API in the Cloud API Manager.

Add Realtime database SDK to your app:

Using the Firebase Android BoM, declare the dependency for the Realtime Database Android library in your module (app-level) Gradle file (usually app/build.gradle).

```
dependencies {
    // Import the BoM for the Firebase platform
    implementation platform('com.google.firebase:firebase-bom:28.4.2')

    // Declare the dependency for the Realtime Database library
    // When using the BoM, you don't specify versions in Firebase library dependencies
    implementation 'com.google.firebase:firebase-database'
}
```

Configure Realtime database rules:

The Realtime Database provides a declarative rules language that allows you to define how your data should be structured, how it should be indexed, and when your data can be read from and written to.

Write to your Database:

Retrieve an instance of your database using getInstance() and reference the location you want to write to.

```
// Write a message to the database
FirebaseDatabase database = FirebaseDatabase.getInstance();
DatabaseReference myRef = database.getReference("message");

myRef.setValue("Hello, World!");
```

You can save a range of data types to the database this way, including Java objects. When you save an object the responses from any getters will be saved as children of this location.

Read from your Database:

```
// Read from the database
myRef.addValueEventListener(new ValueEventListener() {
    @Override
    public void onDataChange(DataSnapshot dataSnapshot) {
        // This method is called once with the initial value and again
        // whenever data at this location is updated.
        String value = dataSnapshot.getValue(String.class);
        Log.d(TAG, "Value is: " + value);
    }

    @Override
    public void onCancelled(DatabaseError error) {
        // Failed to read value
        Log.w(TAG, "Failed to read value.", error.toException());
    }
});
```

Important Links:

Installation & Setup on Android | Firebase Documentation (google.com)

Create a Cloud Firestore Database:

1. If you haven't already, create a Firebase project: In the Firebase console, click Add project, then follow the on-screen instructions to create a Firebase project or to add Firebase services to an existing GCP project.
2. Navigate to the Cloud Firestore section of the Firebase console. You'll be prompted to select an existing Firebase project. Follow the database creation workflow.
3. Select a starting mode for your Cloud Firestore Security Rules:
    a. Test mode: Good for getting started with the mobile and web client libraries, but allows anyone to read and overwrite your data. After testing, make sure to review the Secure your data section.
    b. Locked mode: Denies all reads and writes from mobile and web clients. Your authenticated application servers (C#, Go, Java, Node.js, PHP, Python, or Ruby) can still access your database.
4. Select a location for your database.
5. Click Done.

Setup your development environment:

Add the required dependencies and client libraries to your app.

```
dependencies {
    // Import the BoM for the Firebase platform
    implementation platform('com.google.firebase:firebase-bom:28.4.2')

    // Declare the dependency for the Cloud Firestore library
    // When using the BoM, you don't specify versions in Firebase library dependencies
    implementation 'com.google.firebase:firebase-firestore'
}
```

Initialize Cloud Firestore:

Initialize an instance of Cloud Firestore:

```
// Access a Cloud Firestore instance from your Activity


FirebaseFirestore db = FirebaseFirestore.getInstance();
```

Add Data:

Cloud Firestore stores data in Documents, which are stored in Collections. Cloud Firestore creates collections and documents implicitly the first time you add data to the document. You do not need to explicitly create collections or documents.

Create a new collection and a document using the following example code.

```
// Create a new user with a first and last name
Map<String, Object> user = new HashMap<>();
user.put("first", "Ada");
user.put("last", "Lovelace");
user.put("born", 1815);

// Add a new document with a generated ID
db.collection("users")
        .add(user)
        .addOnSuccessListener(new OnSuccessListener<DocumentReference>() {
            @Override
            public void onSuccess(DocumentReference documentReference) {
                Log.d(TAG, "DocumentSnapshot added with ID: " + documentReference.getId());
            }
        })
        .addOnFailureListener(new OnFailureListener() {
            @Override
            public void onFailure(@NonNull Exception e) {
                Log.w(TAG, "Error adding document", e);
            }
        });
```

Now add another document to the users collection. Notice that this document includes a key-value pair (middle name) that does not appear in the first document. Documents in a collection can contain different sets of information.

```
// Create a new user with a first, middle, and last name
Map<String, Object> user = new HashMap<>();
user.put("first", "Alan");
user.put("middle", "Mathison");
user.put("last", "Turing");
user.put("born", 1912);

// Add a new document with a generated ID
db.collection("users")
        .add(user)
        .addOnSuccessListener(new OnSuccessListener<DocumentReference>() {
            @Override
            public void onSuccess(DocumentReference documentReference) {
                Log.d(TAG, "DocumentSnapshot added with ID: " + documentReference.getId());
            }
        })
        .addOnFailureListener(new OnFailureListener() {
            @Override
            public void onFailure(@NonNull Exception e) {
                Log.w(TAG, "Error adding document", e);
            }
        });
```

Read Data:

To quickly verify that you've added data to Cloud Firestore, use the data viewer in the Firebase console.

You can also use the "get" method to retrieve the entire collection.

```
db.collection("users")
        .get()
        .addOnCompleteListener(new OnCompleteListener<QuerySnapshot>() {
            @Override
            public void onComplete(@NonNull Task<QuerySnapshot> task) {
                if (task.isSuccessful()) {
                    for (QueryDocumentSnapshot document : task.getResult()) {
                        Log.d(TAG, document.getId() + " => " + document.getData());
                    }
                } else {
                    Log.w(TAG, "Error getting documents.", task.getException());
                }
            }
        });
```

Secure your data:

If you're using the Web, Android, or iOS SDK, use Firebase Authentication and Cloud Firestore Security Rules to secure your data in Cloud Firestore.

Here are some basic rule sets you can use to get started. You can modify your security rules in the Rules tab of the console.

```
// Allow read/write access on all documents to any user signed in to the application
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if request.auth != null;
    }
  }
}
```

Important Links:

Getting Started With Cloud Firestore on Android - Firecasts - YouTube

Get started with Cloud Firestore | Firebase Documentation (google.com)

Add Firebase Authentication to your app:

Using the Firebase Android BoM, declare the dependency for the Firebase Authentication Android library in your module (app-level) Gradle file (usually app/build.gradle).

```
dependencies {
    // Import the BoM for the Firebase platform
    implementation platform('com.google.firebase:firebase-bom:28.4.2')

    // Declare the dependency for the Firebase Authentication library
    // When using the BoM, you don't specify versions in Firebase library dependencies
    implementation 'com.google.firebase:firebase-auth'
}
```

To use an authentication provider, you need to enable it in the Firebase console. Go to the Sign-in Method page in the Firebase Authentication section to enable Email/Password sign-in and any other identity providers you want for your app.

Check Current Authentication state:

1. Declare an instance of FirebaseAuth.

```
private FirebaseAuth mAuth;
```

2. In the onCreate() method, initialize the FirebaseAuth instance.

```
// Initialize Firebase Auth
mAuth = FirebaseAuth.getInstance();
```

3. When initializing your Activity, check to see if the user is currently signed in.

```
@Override
public void onStart() {
    super.onStart();
    // Check if user is signed in (non-null) and update UI accordingly.
    FirebaseUser currentUser = mAuth.getCurrentUser();
    if(currentUser != null){
        reload();
    }
}
```

Signup new user:

Create a new createAccount method that takes in an email address and password, validates them, and then creates a new user with the createUserWithEmailAndPassword method.

```
mAuth.createUserWithEmailAndPassword(email, password)
        .addOnCompleteListener(this, new OnCompleteListener<AuthResult>() {
            @Override
            public void onComplete(@NonNull Task<AuthResult> task) {
                if (task.isSuccessful()) {
                    // Sign in success, update UI with the signed-in user's information
                    Log.d(TAG, "createUserWithEmail:success");
                    FirebaseUser user = mAuth.getCurrentUser();
                    updateUI(user);
                } else {
                    // If sign in fails, display a message to the user.
                    Log.w(TAG, "createUserWithEmail:failure", task.getException());
                    Toast.makeText(EmailPasswordActivity.this, "Authentication failed.",
                            Toast.LENGTH_SHORT).show();
                    updateUI(null);
                }
            }
        });
```

Add a form to register new users with their email and password and call this new method when it is submitted.

Sign in existing users:

Create a new signIn method which takes in an email address and password, validates them, and then signs a user in with the signInWithEmailAndPassword method.

```
mAuth.signInWithEmailAndPassword(email, password)
        .addOnCompleteListener(this, new OnCompleteListener<AuthResult>() {
            @Override
            public void onComplete(@NonNull Task<AuthResult> task) {
                if (task.isSuccessful()) {
                    // Sign in success, update UI with the signed-in user's information
                    Log.d(TAG, "signInWithEmail:success");
                    FirebaseUser user = mAuth.getCurrentUser();
                    updateUI(user);
                } else {
                    // If sign in fails, display a message to the user.
                    Log.w(TAG, "signInWithEmail:failure", task.getException());
                    Toast.makeText(EmailPasswordActivity.this, "Authentication failed.",
                            Toast.LENGTH_SHORT).show();
                    updateUI(null);
                }
            }
        });
```

Access User Authentication:

If a user has signed in successfully you can get their account data at any point with the getCurrentUser method.

```
FirebaseUser user = FirebaseAuth.getInstance().getCurrentUser();
if (user != null) {
    // Name, email address, and profile photo Url
    String name = user.getDisplayName();
    String email = user.getEmail();
    Uri photoUrl = user.getPhotoUrl();

    // Check if user's email is verified
    boolean emailVerified = user.isEmailVerified();

    // The user's ID, unique to the Firebase project. Do NOT use this value to
    // authenticate with your backend server, if you have one. Use
    // FirebaseUser.getIdToken() instead.
    String uid = user.getUid();
}
```

Code Example:

quickstart-android/auth/app/src/main at master · firebase/quickstart-android (github.com)

Important Links:

Authenticate Using Google Sign-In on Android | Firebase Documentation

Lab Tasks

Activity 1

Create an app that manages student records in Sqlite database using ROOM API. Your app must be able to add a student record, search and modify the record.

Activity 2

Modify the app you have created in **Activity 1** to use Firebase Firestore as storage.

Exercises

1. Create a simple app that stores user details as preferences. (Details can be anything).
2. Create a simple app that read data from a file in cache folder.

Assignment Deliverables

Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teacher.

LAB 10: Multithreading

Objective

When an application component starts and the application does not have any other components running, the Android system starts a new Linux process for the application with a single thread of execution. By default, all components of the same application run in the same process and thread (called the "main" thread). You are going to learn these in this lab.

Scope

Understanding and implementing Process, Threads and their respective types.

Useful Concepts

Process

By default, all components of the same application run in the same process and most applications should not change this. However, if you find that you need to control which process a certain component belongs to, you can do so in the manifest file.

The manifest entry for each type of component element—<activity>, <service>, <receiver>, and <provider>—supports an android:process attribute that can specify a process in which that component should run.

Thread

When an application is launched, the system creates a thread of execution for the application, called "main." This thread is very important because it is in charge of dispatching events to the appropriate user interface widgets, including drawing events.

The system does not create a separate thread for each instance of a component. All components that run in the same process are instantiated in the UI thread, and system calls to each component are dispatched from that thread.

When your app performs intensive work in response to user interaction, this single thread model can yield poor performance unless you implement your application properly. Specifically, if everything is happening in the UI thread, performing long operations such as network access or database queries will block the whole UI. When the thread is blocked, no events can be dispatched, including drawing events. From the user's perspective, the application appears to hang. Even worse, if the UI thread is blocked for more than a few seconds (about 5 seconds currently) the user is presented with the infamous "application not responding" (ANR) dialog. The user might then decide to quit your application and uninstall it if they are unhappy.

Additionally, the Android UI toolkit is not thread-safe. So, you must not manipulate your UI from a worker thread—you must do all manipulation to your user interface from the UI thread. Thus, there are simply two rules to Android's single thread model:

- Do not block the UI thread
- Do not access the Android UI toolkit from outside the UI thread

Worker Threads

Because of the single threaded model described above, it's vital to the responsiveness of your application's UI that you do not block the UI thread. If you have operations to perform that are not instantaneous, you should make sure to do them in separate threads ("background" or "worker" threads).

However, note that you cannot update the UI from any thread other than the UI thread or the "main" thread.

To fix this problem, Android offers several ways to access the UI thread from other threads. Here is a list of methods that can help:

Activity.runOnUiThread(Runnable)

View.post(Runnable)

View.postDelayed(Runnable, long)

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            // a potentially time consuming task
            final Bitmap bitmap =
                    processBitMap("image.png");
            imageView.post(new Runnable() {
                public void run() {
                    imageView.setImageBitmap(bitmap);
                }
            });
        }
    }).start();
}
```

This implementation is thread-safe: the background operation is done from a separate thread while the ImageView is always manipulated from the UI thread.

However, as the complexity of the operation grows, this kind of code can get complicated and difficult to maintain. To handle more complex interactions with a worker thread, you might consider using a Handler in your worker thread, to process messages delivered from the UI thread. See Threading on Android for a full explanation of how to schedule work on background threads and communicate back to the UI thread.

Thread-safe Methods

In some situations, the methods you implement might be called from more than one thread, and therefore must be written to be thread-safe.

This is primarily true for methods that can be called remotely—such as methods in a bound service.

Similarly, a content provider can receive data requests that originate in other processes.

Android offers a mechanism for interprocess communication (IPC) using remote procedure calls (RPCs), in which a method is called by an activity or other application component, but executed remotely (in another process), with any result returned back to the caller.

To perform IPC, your application must bind to a service, using bindService(). For more information, see the Services Lab.

Code Example

Ibtisam/Producer-Consumer (github.com)

Ibtisam/Downloader-using-Worker-Thread (github.com)

Important Links

Processes and threads overview | Android Developers

Better Performance through Threading

Making adept use of threads on Android can help you boost your app's performance.

Main Thread Internals

The main thread has a very simple design: Its only job is to take and execute blocks of work from a thread-safe work queue until its app is terminated. The framework generates some of these blocks of work from a variety of places. These places include callbacks associated with lifecycle information, user events such as input, or events coming from other apps and processes. In addition, app can explicitly enqueue blocks on their own, without using the framework.

Nearly any block of code your app executes is tied to an event callback, such as input, layout inflation, or draw. When something triggers an event, the thread where the event happened pushes the event out of itself, and into the main thread's message queue. The main thread can then service the event.

Moving numerous or long tasks from the main thread, so that they don't interfere with smooth rendering and fast responsiveness to user input, is the biggest reason for you to adopt threading in your app.

Thread and UI object reference

By design, Android View objects are not thread-safe. An app is expected to create, use, and destroy UI objects, all on the main thread. If you try to modify or even reference a UI object in a thread other than the main thread, the result can be exceptions, silent failures, crashes, and other undefined misbehavior.

Issues with references fall into two distinct categories: explicit references and implicit references.

Explicit References

Many tasks on non-main threads have the end goal of updating UI objects. However, if one of these threads accesses an object in the view hierarchy, application instability can result: If a worker thread changes the properties of that object at the same time that any other thread is referencing the object, the results are undefined.

In all cases, your app should only update UI objects on the main thread. This means that you should craft a negotiation policy that allows multiple threads to communicate work back to the main thread, which tasks the topmost activity or fragment with the work of updating the actual UI object.

Implicit References

A common code-design flaw with threaded objects can be seen in the snippet of code below:

```
public class MainActivity extends Activity {
  // ...
  public class MyAsyncTask extends AsyncTask<Void, Void, String>    {
    @Override protected String doInBackground(Void... params) {...}
    @Override protected void onPostExecute(String result) {...}
  }
}
```

The flaw in this snippet is that the code declares the threading object MyAsyncTask as a non-static inner class of some activity (or an inner class in Kotlin). This declaration creates an implicit reference to the enclosing Activity instance. As a result, the object contains a reference to the activity until the threaded work completes, causing a delay in the destruction of the referenced activity. This delay, in turn, puts more pressure on memory.

A direct solution to this problem would be to define your overloaded class instances either as static classes, or in their own files, thus removing the implicit reference.

Threads and app activity lifecycles

The app lifecycle can affect how threading works in your application. You may need to decide that a thread should, or should not, persist after an activity is destroyed. You should also be aware of the relationship between thread prioritization and whether an activity is running in the foreground or background.

Persisting Threads

Threads persist past the lifetime of the activities that spawn them. Threads continue to execute, uninterrupted, regardless of the creation or destruction of activities, although they will be terminated together with the application process once there are no more active application components. In some cases, this persistence is desirable.

Consider a case in which an activity spawns a set of threaded work blocks, and is then destroyed before a worker thread can execute the blocks. What should the app do with the blocks that are in flight?

If the blocks were going to update a UI that no longer exists, there's no reason for the work to continue. For example, if the work is to load user information from a database, and then update views, the thread is no longer necessary.

Managing lifecycle responses manually for all threading objects can become extremely complex. If you don't manage them correctly, your app can suffer from memory contention and performance

issues.

Thread Priority

Every time you create a thread, you should call setThreadPriority(). The system's thread scheduler gives preference to threads with high priorities, balancing those priorities with the need to eventually get all the work done. Generally, threads in the foreground group get about 95% of the total execution time from the device, while the background group gets roughly 5%.

The Process class helps reduce complexity in assigning priority values by providing a set of constants that your app can use to set thread priorities. For example, THREAD_PRIORITY_DEFAULT represents the default value for a thread. Your app should set the thread's priority to THREAD_PRIORITY_BACKGROUND for threads that are executing less-urgent work.

Helper Classes for Threading

The framework also provides the same Java classes and primitives to facilitate threading, such as the Thread, Runnable , and Executors classes, as well as additional ones such as HandlerThread. For further information, please refer to Threading on Android.

The HandlerThread Class

A handler thread is effectively a long-running thread that grabs work from a queue and operates on it.

When your app creates a thread using HandlerThread, don't forget to set the thread's priority based on the type of work it's doing. Remember, CPUs can only handle a small number of threads in parallel. Setting the priority helps the system know the right ways to schedule this work when all other threads are fighting for attention.

The ThreadPoolExecutor Class

ThreadPoolExecutor is a helper class to make this process easier. This class manages the creation of a group of threads, sets their priorities, and manages how work is distributed among those threads. As workload increases or decreases, the class spins up or destroys more threads to adjust to the workload.

This class also helps your app spawn an optimum number of threads. When it constructs a ThreadPoolExecutor object, the app sets a minimum and maximum number of threads. As the workload given to the ThreadPoolExecutor increases, the class will take the initialized minimum and maximum thread counts into account, and consider the amount of pending work there is to do. Based on these factors, ThreadPoolExecutor decides on how many threads should be alive at any given time.

Important Links

Better performance through threading | Android Developers

Background Threads

All Android apps use a main thread to handle UI operations. Calling long-running operations from this main thread can lead to freezes and unresponsiveness. For example, if your app makes a network request from the main thread, your app's UI is frozen until it receives the network response. You can create additional background threads to handle long-running operations while the main thread continues to handle UI updates.

Creating Multiple Threads

A thread pool is a managed collection of threads that runs tasks in parallel from a queue. New tasks are executed on existing threads as those threads become idle. To send a task to a thread pool, use the ExecutorService interface. Note that ExecutorService has nothing to do with Services, the Android application component.

Creating threads is expensive, so you should create a thread pool only once as your app initializes. Be sure to save the instance of the ExecutorService either in your Application class or in a dependency injection container. The following example creates a thread pool of four threads that we can use to run background tasks.

```
public class MyApplication extends Application {
    ExecutorService executorService = Executors.newFixedThreadPool(4);
}
```

Executing in background Thread

Making a network request on the main thread causes the thread to wait, or block, until it receives a response. Since the thread is blocked, the OS can't call onDraw(), and your app freezes, potentially leading to an Application Not Responding (ANR) dialog. Instead, let's run this operation on a background thread.

Executor

An object that executes submitted Runnable tasks. This interface provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc. An Executor is normally used instead of explicitly creating threads. For example, rather than invoking new Thread(new RunnableTask()).start() for each of a set of tasks, you might use:

```
Executor executor = anExecutor();
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
...
```

However, the Executor interface does not strictly require that execution be asynchronous. In the simplest case, an executor can run the submitted task immediately in the caller's thread:

```
class DirectExecutor implements Executor {
  public void execute(Runnable r) {
    r.run();
  }
}
```

More typically, tasks are executed in some thread other than the caller's thread. The executor below spawns a new thread for each task.

```
class ThreadPerTaskExecutor implements Executor {
  public void execute(Runnable r) {
    new Thread(r).start();
  }
}
```

Many Executor implementations impose some sort of limitation on how and when tasks are scheduled. The executor below serializes the submission of tasks to a second executor, illustrating a composite executor.

```
class SerialExecutor implements Executor {
  final Queue<Runnable> tasks = new ArrayDeque<>();
  final Executor executor;
  Runnable active;

  SerialExecutor(Executor executor) {
    this.executor = executor;
  }

  public synchronized void execute(final Runnable r) {
    tasks.add(new Runnable() {
      public void run() {
        try {
          r.run();
        } finally {
          scheduleNext();
        }
      }
    });
    if (active == null) {
      scheduleNext();
    }
  }

  protected synchronized void scheduleNext() {
    if ((active = tasks.poll()) != null) {
      executor.execute(active);
    }
  }
}
```

The Executor implementations provided in this package implement ExecutorService, which is a more extensive interface. The ThreadPoolExecutor class provides an extensible thread pool implementation. The Executors class provides convenient factory methods for these Executors.

Memory consistency effects: Actions in a thread prior to submitting a Runnable object to an Executor happen-before its execution begins, perhaps in another thread.

Handler

A Handler allows you to send and process Message and Runnable objects associated with a thread's MessageQueue. Each Handler instance is associated with a single thread and that thread's message queue. When you create a new Handler it is bound to a Looper. It will deliver messages and runnables to that Looper's message queue and execute them on that Looper's thread.

There are two main uses for a Handler: (1) to schedule messages and runnables to be executed at some point in the future; and (2) to enqueue an action to be performed on a different thread than your own.

Scheduling messages is accomplished with the post(Runnable), postAtTime(java.lang.Runnable, long), postDelayed(Runnable, Object, long), sendEmptyMessage(int), sendMessage(Message), sendMessageAtTime(Message, long), and sendMessageDelayed(Message, long) methods. The post versions allow you to enqueue Runnable objects to be called by the message queue when they are received; the sendMessage versions allow you to enqueue a Message object containing a bundle of data that will be processed by the Handler's handleMessage(Message) method (requiring that you implement a subclass of Handler).

When posting or sending to a Handler, you can either allow the item to be processed as soon as the message queue is ready to do so, or specify a delay before it gets processed or absolute time for it to be processed. The latter two allow you to implement timeouts, ticks, and other timing-based behavior.

When a process is created for your application, its main thread is dedicated to running a message queue that takes care of managing the top-level application objects (activities, broadcast receivers, etc) and any windows they create. You can create your own threads, and communicate back with the main application thread through a Handler. This is done by calling the same post or sendMessage methods as before, but from your new thread. The given Runnable or Message will then be scheduled in the Handler's message queue and processed when appropriate.

Code Examples

Important Links

Lab Tasks

Activity 1

Run the following coding examples:

- Ibtisam/Downloader-using-Handler-Runnable (github.com)
- Ibtisam/Downloader-using-Handler-Message (github.com)

Activity 2

Go through the following articles and implement them where you felt it necessary for your understanding:

- Running Android tasks in background threads | Android Developers
- Executor | Android Developers
- Handler | Android Developers

Exercises

If any activity is left incomplete then accomplish that at home.

Assignment Deliverables

- Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teacher.

LAB 11: Broadcast Receivers

Objective

In an android device, broadcast message is generated, to inform the currently running app that certain event has occurred. We are going to learn them via different examples in this lab and then you will perform some exercises to enrich your skills in receivers.

Scope

After completing this lab we will be able to understand the following topics.

- Introduction to Broadcast Receivers
- Types of Broadcast Receivers
- Creating and Registering Broadcast Receivers

Useful Concepts

Broadcasts can be generated in two ways:

- **The system-level event broadcast:**

  The broadcast message generated by the system is known as system-level event broadcast,such as the screen is being turned off, the battery is low, or an SMS has arrived.

- **An app-level event broadcast:**

  The broadcast message generated by an app is known as app-level event broadcast, for example, an app that is downloading some data may want to inform other apps that the data has been downloaded and ready to use. By using app-level event broadcast, we can inform other apps that some event has occurred. This can be achieved using a broadcast receiver, which enables the apps to register themselves to receive a notification whenever anew feed is available.

Normal Broadcasts

The broadcasts messenger sends message in undefined order to all the registered receivers at the same time. For example, the broadcasts, such as battery low (ACTION_BATTERYLOW)or timezone changed (ACTION_TIMEZONE_CHANGED) will be received by all the registered receivers at the same time. This type of broadcasts are sent using the sendBroadcast() method of the android.content.Context class.

Ordered Broadcasts

The broadcast messenger sends message to all the registered receivers in an ordered manner, which means that a broadcast is delivered to one receiver at a time. When a receiver receives a broadcast, it can either propagate the broadcast to the next receiver

or itcan completely abort the broadcast. If a broadcast message is aborted by a receiver, it will not be passed to other receivers.

Creating and Registering a Broadcast Receiver

1. Creating the Broadcast Receiver

Each broadcast is delivered in the form of an intent object. Broadcast receivers enable appsto receive intents object that are either broadcasted by the system or by other apps. A broadcast receiver is implemented as a subclass of Broadcast Receiver class and overriding the onReceive() method where each message is received as a Intent object parameter.

The onReceive() method is called when a broadcast receiver receives a broadcast intent object. This method must include the code that needs to be executed when a broadcast is received. This method is usually called within the main thread of its process. Therefore, we should not perform long-running operations in this method. The system allows a timeout of 10 seconds before considering the receiver to be blocked. The system can kill a receiver thathas been blocked. Also, we cannot launch a pop-up dialog in our onReceive ( ) method implementation. The signature of the onReceive () method is:

public abstract void onReceive (Context context, Intent intent)

Registering a Broadcast Receiver

To register the broadcast receiver, app‘s broadcast intent has to be registered in AndroidManifest.xml file.

To declare a broadcast receiver in the manifest file, you need to add a <receiver> element asa child of the <application> element by specifying the class name of the broadcast receiver
to register. The <receiver> element also needs to include an <intent-filter> element thatspecifies the action string being listened for.

Broadcaster receivers can be registered by either one the two methods.

- Statically
- Dynamically

Static Registering

By using the method registerReceiver() a broadcast receiver is registered dynamically. If themanifest file is used for registering the broadcast receiver, it is known as static method.

Dynamic Registering

We can register a broadcast receiver using the registerReceiver() method. The registerReceiver() method is called with a broadcast intent that matches the filter in

themain app thread. The signature of the registerReceiver() method is :

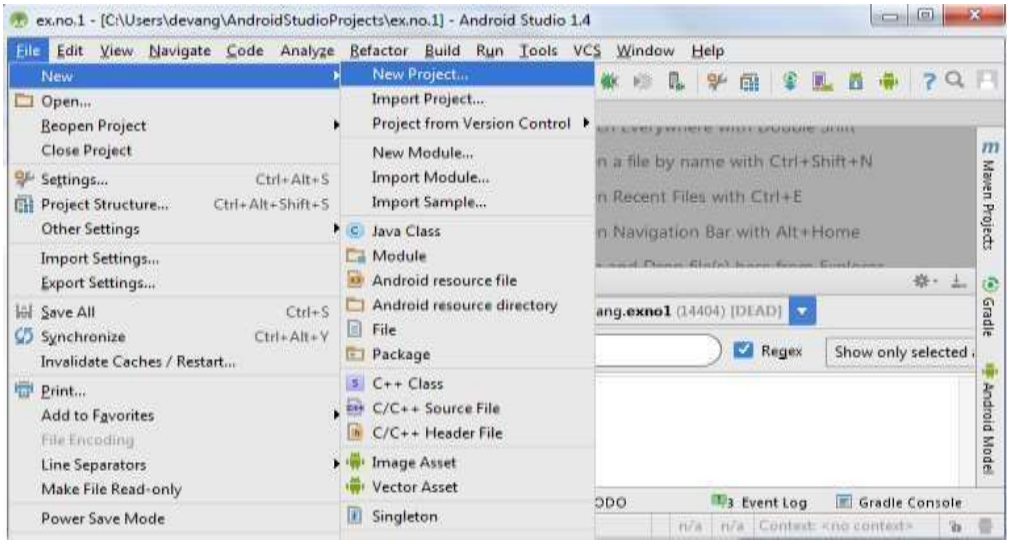
public abstract Intent registerReceiver(BroadcastReceiver receiver, IntentFilterfilter)

Lab Tasks

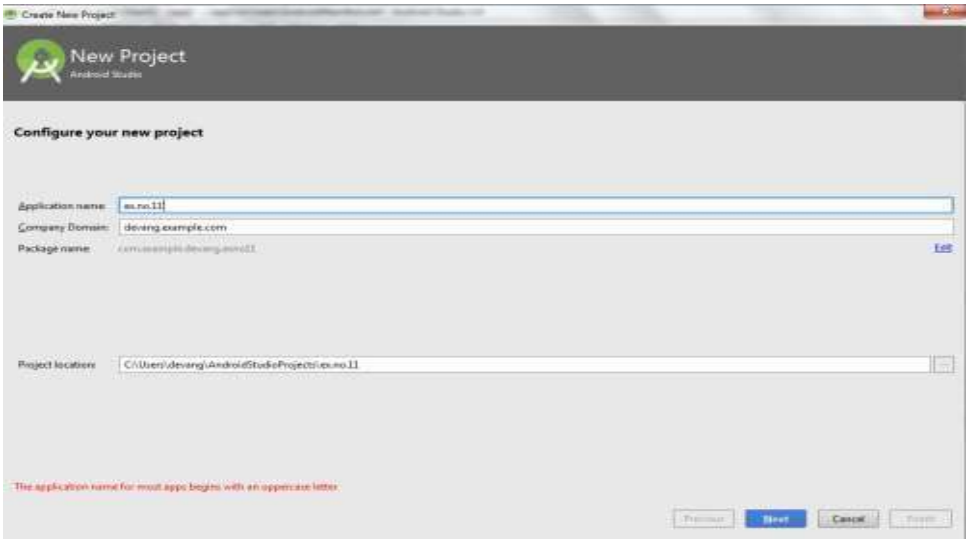
Activity 1

Creating a New project:

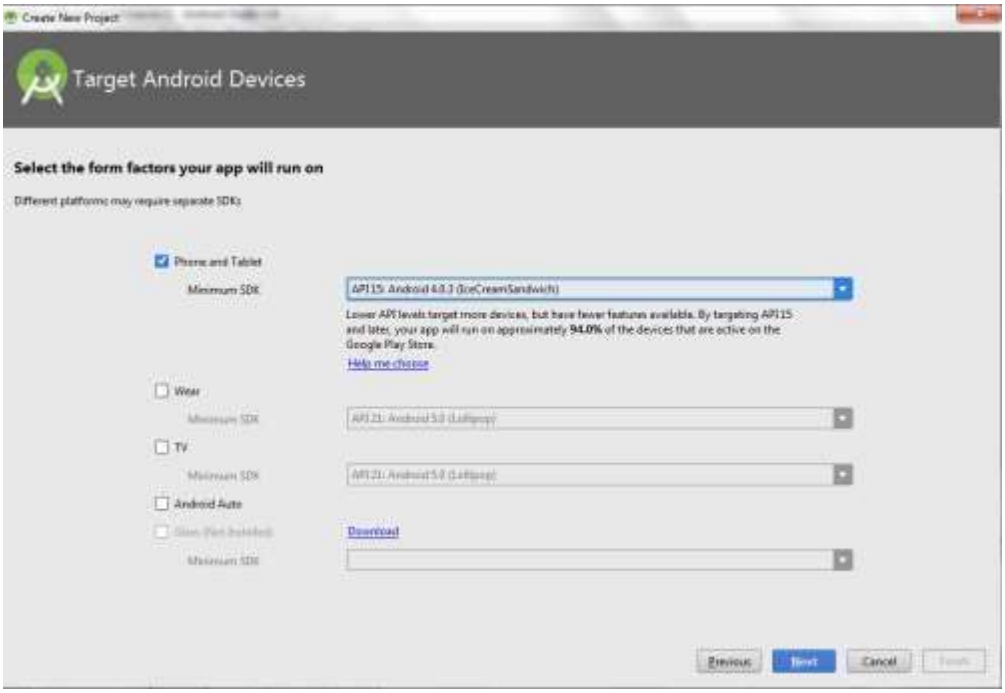
- Open Android Studio and then click on **File -> New -> New project.**



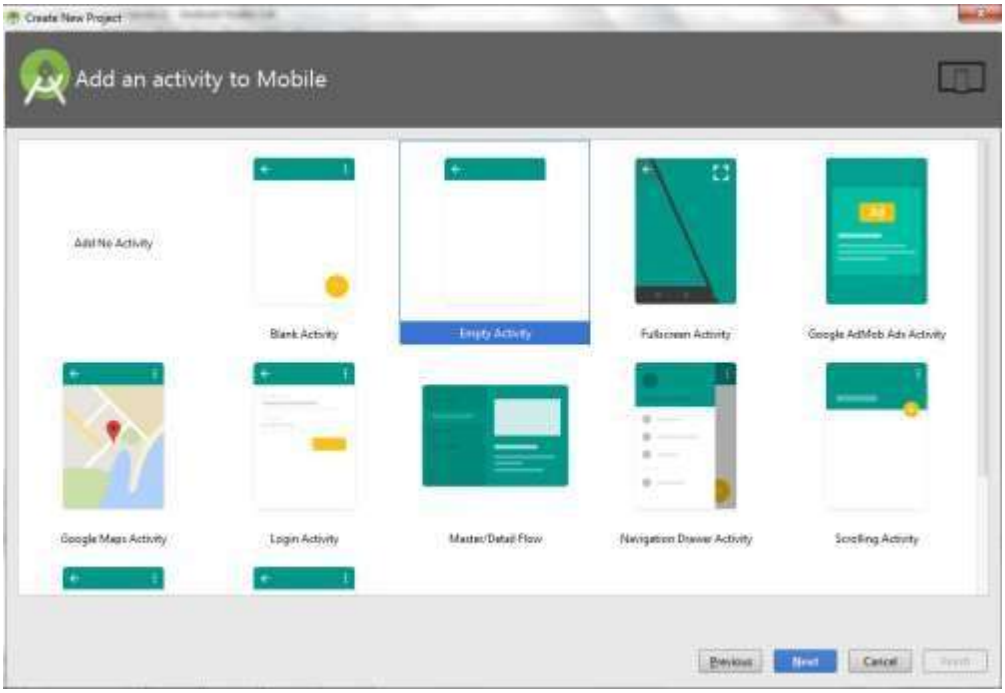
- Then type the Application name as "**ex.no.11"** and click **Next.**



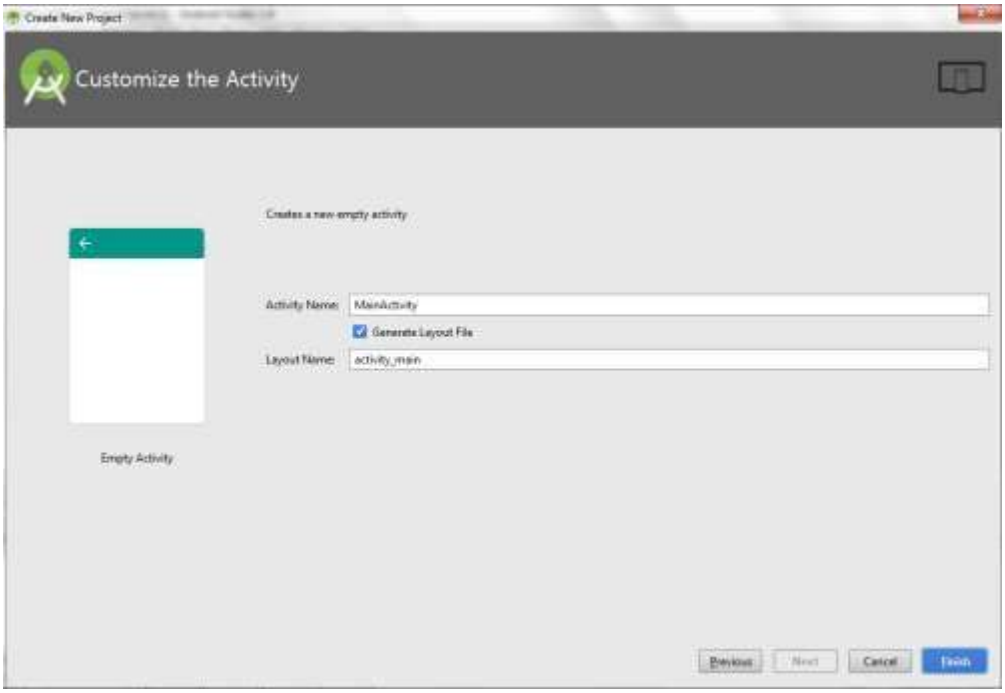
- Then select the **Minimum SDK** as shown below and click **Next**.



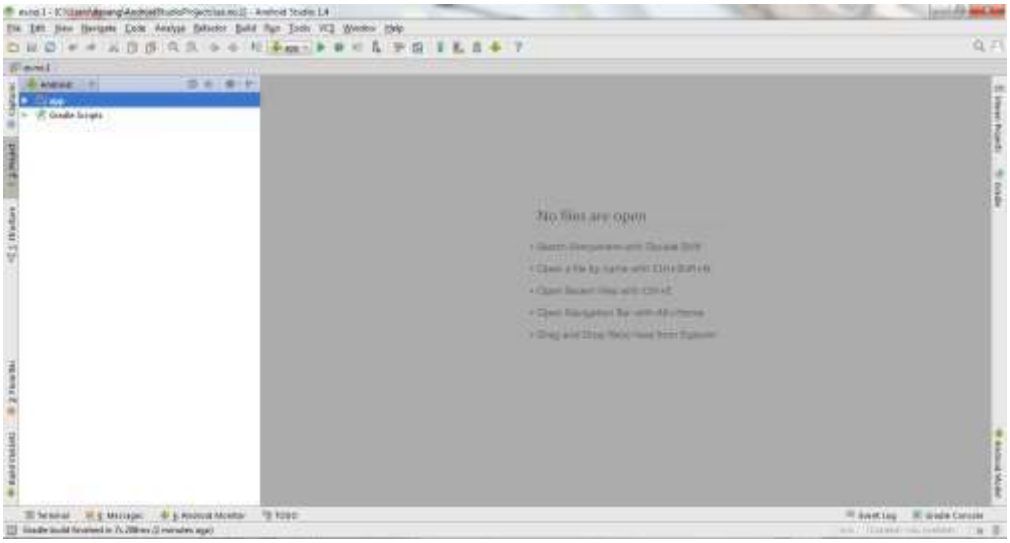
- Then select the **Empty Activity** and click **Next.**



- Finally click **Finish**.

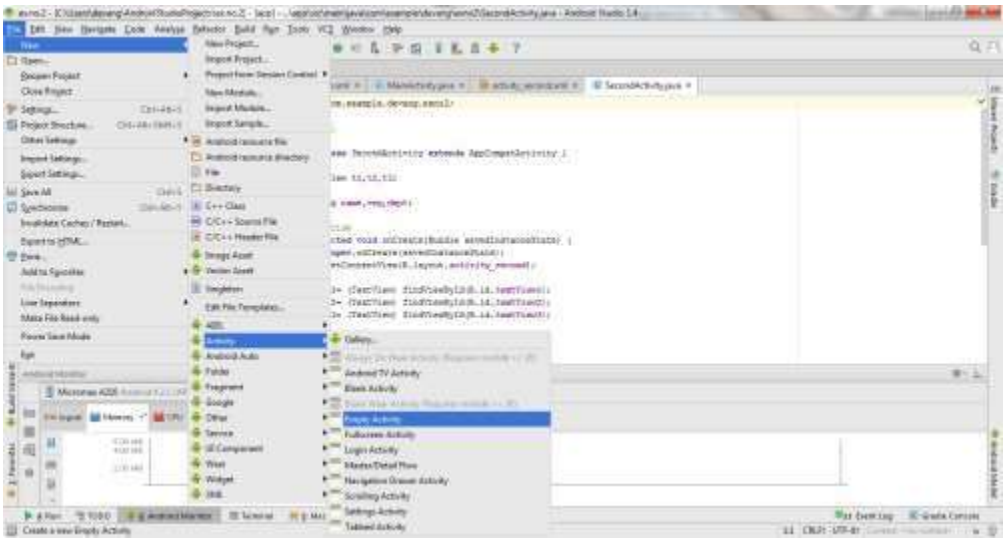


- It will take some time to build and load the project.
- After completion it will look as given below.

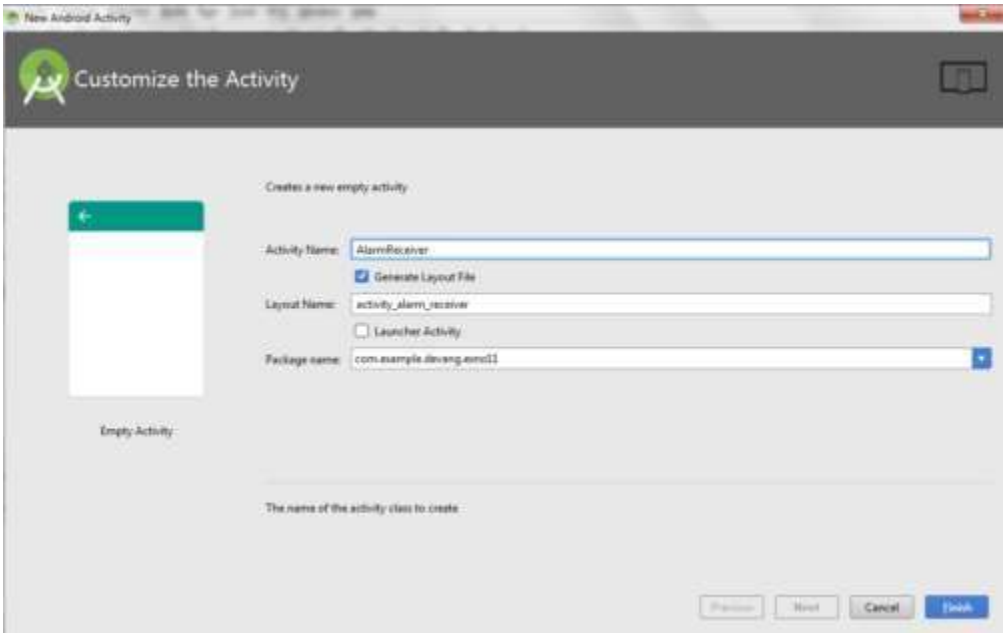


Creating Second Activity for the Android Application:

- Click on **File -> New -> Activity -> Empty Activity.**



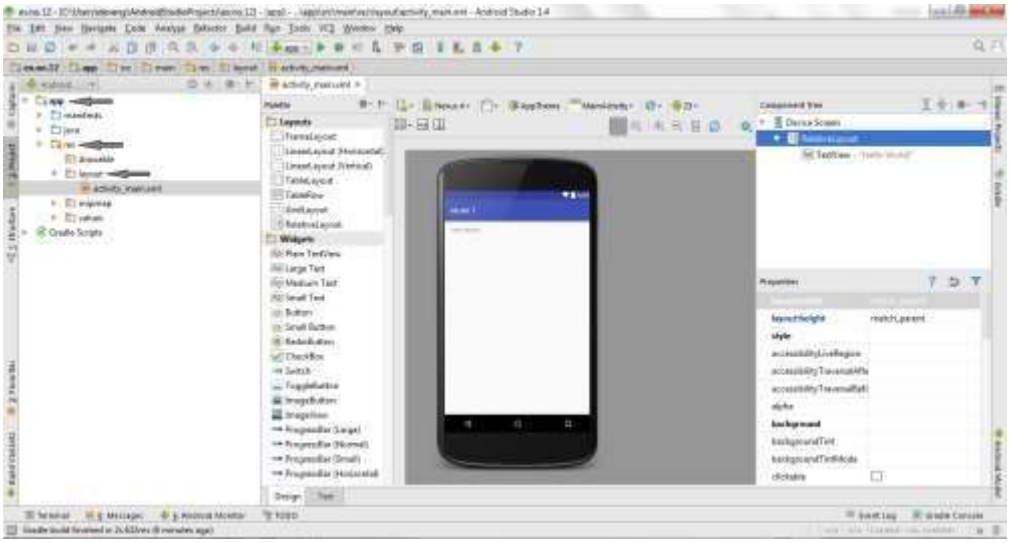
- Type the Activity Name as **AlarmReceiver** and click **Finish** button.



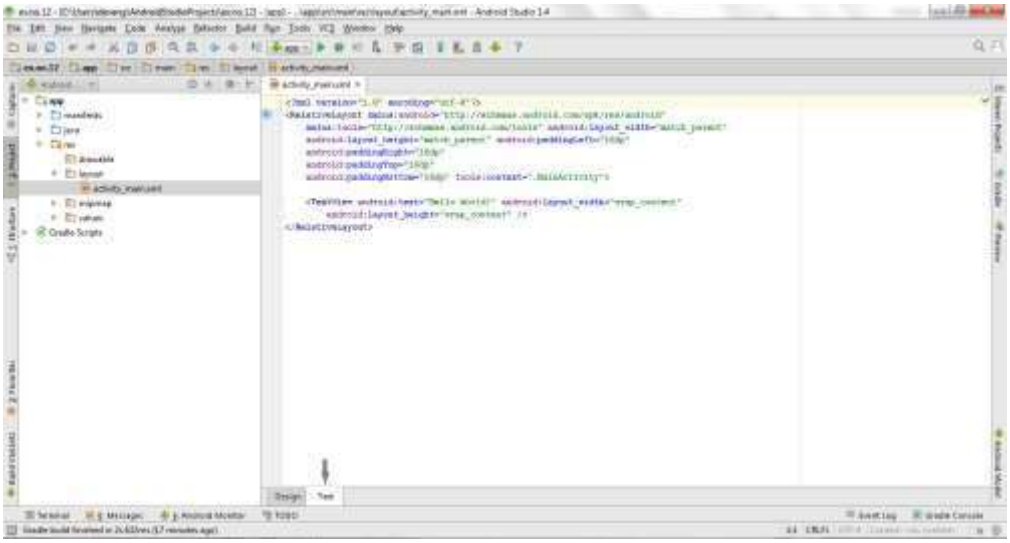
- Thus Second Activity For the application is created.

  Designing layout for the Android Application:

- Click on **app -> res -> layout -> activity_main.xml.**



- Now click on **Text** as shown below.

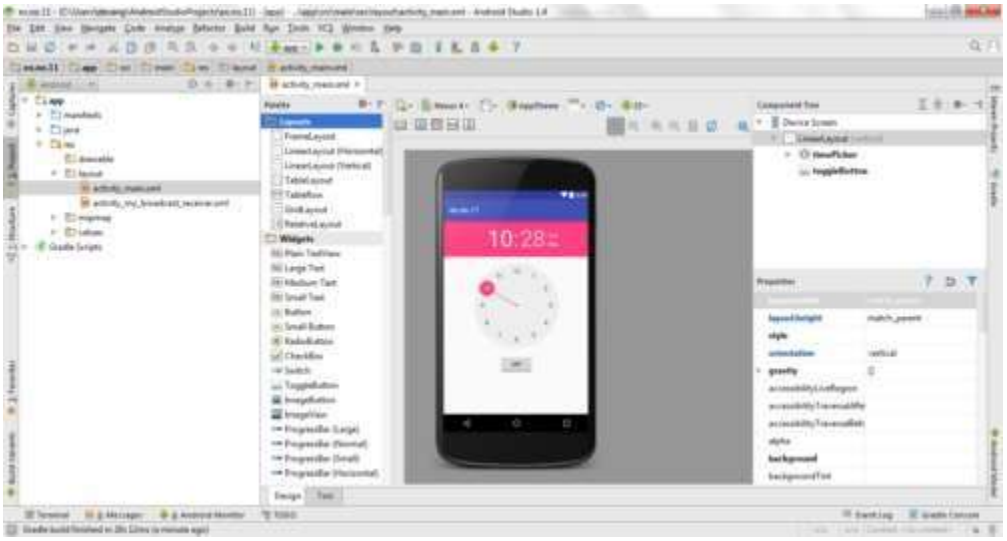


- Then delete the code which is there and type the code as given below.

Code for Activity_main.xml:

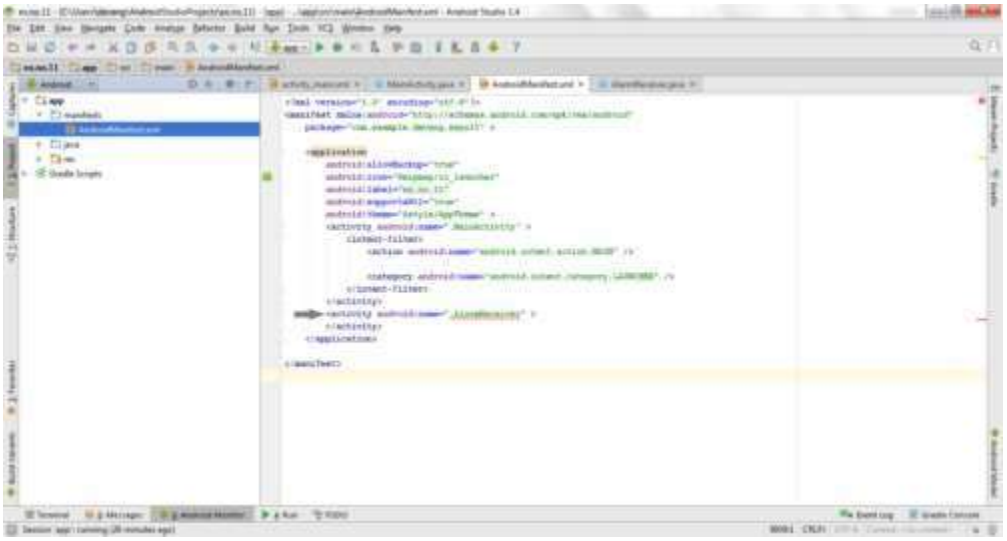
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TimePicker
        android:id="@+id/timePicker"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center" />

    <ToggleButton
        android:id="@+id/toggleButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="20dp"
        android:checked="false"
        android:onClick="OnToggleClicked" />

</LinearLayout>
```

- Now click on **Design** and your application will look as given below.

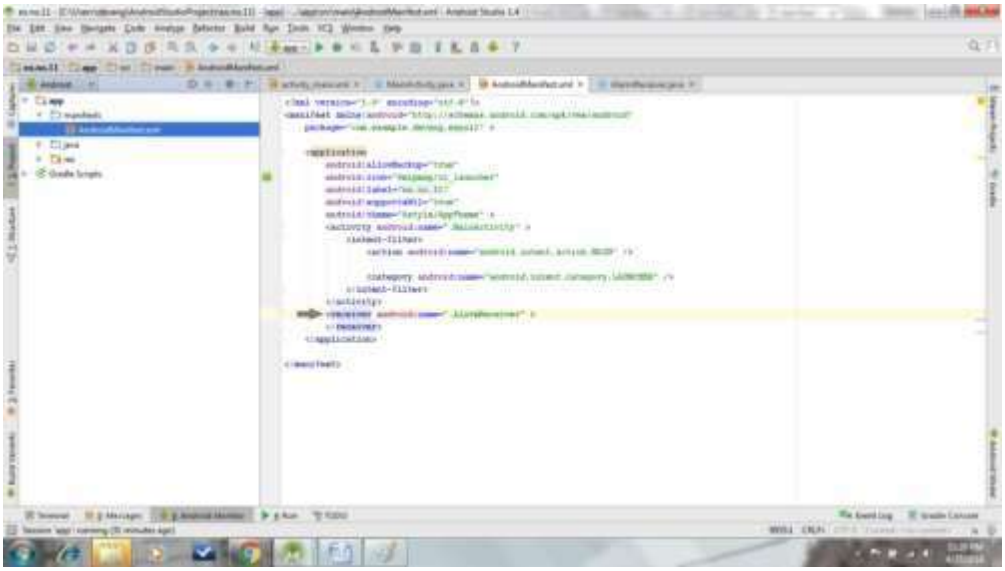


- So now the designing part is completed.

Changes in Manifest for the Android Application: Click on app -> manifests -> AndroidManifest.xml



- Now change the **activity tag** to **receiver tag** in the AndroidManifest.xml file as shown below



Code for AndroidManifest.xml:

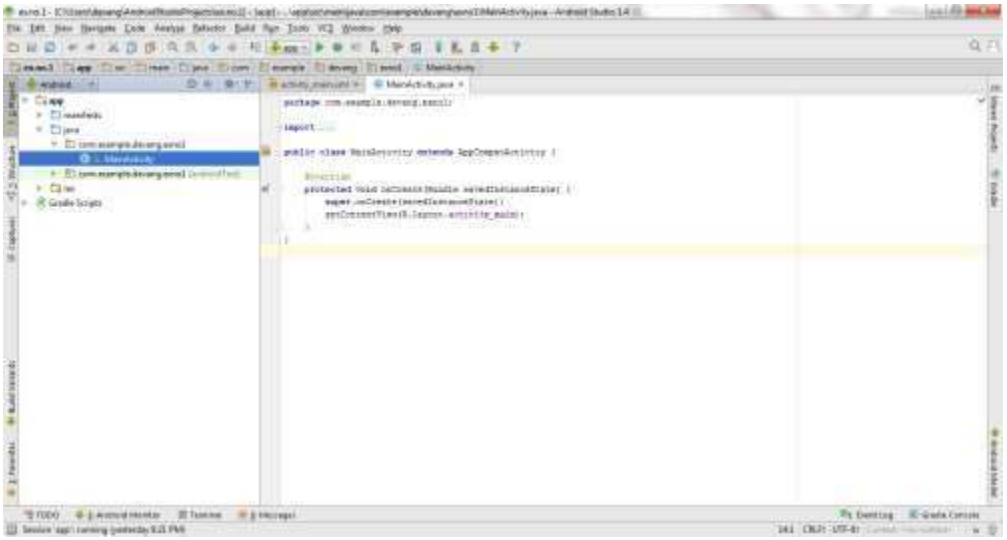
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.exno11" >

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >
        <activity android:name=".MainActivity" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver android:name=".AlarmReceiver" >
        </receiver>
    </application>

</manifest>
```

- So now the changes are done in the Manifest.

Java Coding for the Android Application:

Java Coding for Main Activity:

- Click on **app -> java -> com.example.exno11 -> MainActivity.**



- Then delete the code which is there and type the code as given below.

Code for MainActivity.java:

```
package com.example.exno11;

import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.TimePicker;
import android.widget.Toast;
import android.widget.ToggleButton;

import java.util.Calendar;

public class MainActivity  extends AppCompatActivity
{
    TimePicker alarmTimePicker;
    PendingIntent pendingIntent;
    AlarmManager alarmManager;

    @Override
    protected void onCreate(Bundle  savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        alarmTimePicker = (TimePicker) findViewById(R.id.timePicker);
        alarmManager = (AlarmManager) getSystemService(ALARM_SERVICE);
    }
    public void OnToggleClicked(View  view)
    {
        long time;
        if (((ToggleButton)  view).isChecked())
        {
            Toast.makeText(MainActivity.this, "ALARM ON",
Toast.LENGTH_SHORT).show();
            Calendar calendar = Calendar.getInstance();
```

```
            calendar.set(Calendar.HOUR_OF_DAY,
alarmTimePicker.getCurrentHour());
```

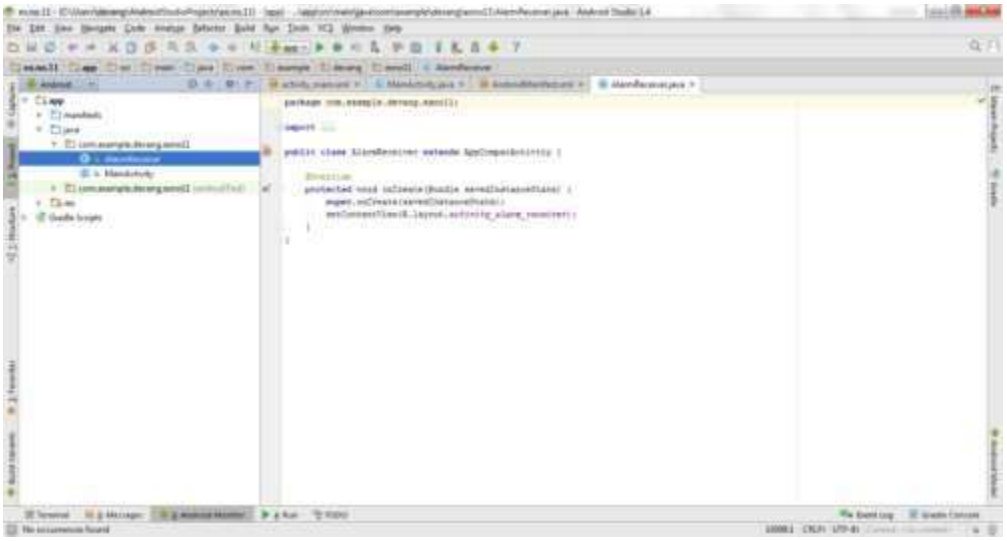
```
            calendar.set(Calendar.MINUTE,
alarmTimePicker.getCurrentMinute());
            Intent intent = new Intent(this, AlarmReceiver.class);
            pendingIntent = PendingIntent.getBroadcast(this, 0, intent, 0);

            time=(calendar.getTimeInMillis()-
(calendar.getTimeInMillis()%60000));
            if(System.currentTimeMillis()>time)
            {
                if (calendar.AM_PM == 0)
                    time = time + (1000*60*60*12);
                else

            }       time = time + (1000*60*60*24);
```

```
            alarmManager.setRepeating(AlarmManager.RTC_WAKEUP, time, 10000,
pendingIntent);
        }
        else
        {
            alarmManager.cancel(pendingIntent);
            Toast.makeText(MainActivity.this, "ALARM OFF",
Toast.LENGTH_SHORT).show();
        }
    }
}
```

- So now the Coding part of Main Activity is completed.

Java Coding for Alarm Receiver:

- Click on **app -> java -> com.example.exno11 -> AlarmReceiver.**



- Then delete the code which is there and type the code as given below.

Code for AlarmReceiver.java:

```
package com.example.exno11;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import  android.media.Ringtone;
import android.media.RingtoneManager;
import android.net.Uri;
import android.widget.Toast;
public class AlarmReceiver  extends BroadcastReceiver
{
    @Override
    public void onReceive(Context  context,  Intent  intent)
    {
        Toast.makeText(context, "Alarm! Wake up! Wake up!",
Toast.LENGTH_LONG).show();
        Uri alarmUri =
RingtoneManager.getDefaultUri(RingtoneManager.TYPE_ALARM);
        if (alarmUri  ==  null)
        {
```

```
            alarmUri =
RingtoneManager.getDefaultUri(RingtoneManager.TYPE_NOTIFICATION);
        }
        Ringtone ringtone = RingtoneManager.getRingtone(context, alarmUri);
        ringtone.play();
    }
}
```

- So now the Coding part of Alarm Receiver is also completed.
- Now run the application to see the output.

Output:

Result:

Thus Android Application that creates Alarm Clock is developed and executed successfully.

Activity 2

Run the following examples of broadcast receivers:

- https://github.com/JimSeker/BroadCastReceiver

Exercises

Think on what enhancements can be done in the examples of Activity 2 and them implement them and share with your class fellows.

Assignment Deliverables

- Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teacher.

LAB 12: Sessional 2 Exam

Objectives

The purpose of this lab is to conduct the second sessional exam based on the activities conducted so far.

Tasks

The tasks will be decided by the respective course instructor/lab tutor.

LAB 13: Services

Objective

A Service is an application component that can perform long-running operations in the background. A service doesn't provide a user interface. Once started, a service might continue running for some time, even after the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform inter-process communication (IPC). For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background

A service runs in the main thread of its hosting process. The service does not create its own thread unless specified, you should run any blocking operations on a separate thread within the service to avoid Application Not Responding (ANR) errors.

The objective of this lab is to understand and implement several concepts related to Service.

Scope

The scope of this lab is the student's ability to:

- Understand the basic concepts of Service.
- Implement Services using Service and IntentService classes.
- Using Notifications to notify the user from background service.
- Implement bound Service to achieve IPC.

Useful Concepts

Service

A Service is an application component that can perform long-running operations in the background. It does not provide a user interface.

Basics

To create a service, you must create a subclass of Service or use one of its existing subclasses. In your implementation, you must override some callback methods that handle key aspects of the service lifecycle and provide a mechanism that allows the components to bind to the service, if appropriate. These are the most important callback methods that you should override:

onStartCommand()

The system invokes this method by calling startService() when another component (such as an activity) requests that the service be started. When this method executes, the service is started and can run in the background indefinitely. If you implement this, it is your responsibility to stop the service when its work is complete by calling stopSelf() or stopService(). If you only want to provide

binding, you don't need to implement this method.

onBind()

The system invokes this method by calling bindService() when another component wants to bind with the service (such as to perform RPC). In your implementation of this method, you must provide an interface that clients use to communicate with the service by returning an IBinder. You must always implement this method; however, if you don't want to allow binding, you should return null.

onCreate()

The system invokes this method to perform one-time setup procedures when the service is initially created (before it calls either onStartCommand() or onBind()). If the service is already running, this method is not called.

onDestroy()

The system invokes this method when the service is no longer used and is being destroyed. Your service should implement this to clean up any resources such as threads, registered listeners, or receivers. This is the last call that the service receives.

Types of Service

These are the three different types of services:

Foreground Service

Foreground services perform operations that are noticeable to the user. Foreground services show a status bar notification, so that users are actively aware that your app is performing a task in the foreground and is consuming system resources. The notification cannot be dismissed unless the service is either stopped or removed from the foreground.

You should only use a foreground service when your app needs to perform a task that is noticeable by the user even when they're not directly interacting with the app. If the action is of low enough importance that you want to use a minimum-priority notification, create a background task instead.

Background Service

A background service performs an operation that isn't directly noticed by the user. For example, if an app used a service to compact its storage, that would usually be a background service.

Bound Service

A service is bound when an application component binds to it by calling bindService(). A bound service offers a client-server interface that allows components to interact with the service, send requests, receive results, and even do so across processes with interprocess communication (IPC). A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

If a component starts the service by calling startService() (which results in a call to onStartCommand()), the service continues to run until it stops itself with stopSelf() or another component stops it by calling stopService().

If a component calls bindService() to create the service and onStartCommand() is not called, the service runs only as long as the component is bound to it. After the service is unbound from all of its clients, the system destroys it.

Implementing Service

Declaring a Service in the manifest

You must declare all services in your application's manifest file, just as you do for activities and other components. To declare your service, add a <service> element as a child of the <application> element. Here is an example:

```
<manifest ... >
  ...
  <application ... >
      <service android:name=".ExampleService" />
      ...
  </application>
</manifest>
```

Creating a Started Service

A started service is one that another component starts by calling startService(), which results in a call to the service's onStartCommand() method.

When a service is started, it has a lifecycle that's independent of the component that started it. The service can run in the background indefinitely, even if the component that started it is destroyed. As such, the service should stop itself when its job is complete by calling stopSelf(), or another component can stop it by calling stopService().

An application component such as an activity can start the service by calling startService() and passing an Intent that specifies the service and includes any data for the service to use. The service receives this Intent in the onStartCommand() method.

The Service class is the base class for all services. When you extend this class, it's important to create a new thread in which the service can complete all of its work; the service uses your application's main thread by default, which can slow the performance of any activity that your application is running.

The Android framework also provides the JobIntentService subclass of Service that uses a worker thread to handle all of the start requests, one at a time.

You can extend the Service class to handle each incoming intent. Here's how a basic implementation might look:

```
public class HelloService extends Service {
  private Looper serviceLooper;
  private ServiceHandler serviceHandler;

  // Handler that receives messages from the thread
  private final class ServiceHandler extends Handler {
      public ServiceHandler(Looper looper) {
          super(looper);
      }
      @Override
      public void handleMessage(Message msg) {
          // Normally we would do some work here, like download a file.
          // For our sample, we just sleep for 5 seconds.
          try {
              Thread.sleep(5000);
          } catch (InterruptedException e) {
              // Restore interrupt status.
              Thread.currentThread().interrupt();
          }
          // Stop the service using the startId, so that we don't stop
          // the service in the middle of handling another job
          stopSelf(msg.arg1);
      }
  }
```

```
@Override
public void onCreate() {
  // Start up the thread running the service. Note that we create a
  // separate thread because the service normally runs in the process's
  // main thread, which we don't want to block. We also make it
  // background priority so CPU-intensive work doesn't disrupt our UI.
  HandlerThread thread = new HandlerThread("ServiceStartArguments",
          Process.THREAD_PRIORITY_BACKGROUND);
  thread.start();

  // Get the HandlerThread's Looper and use it for our Handler
  serviceLooper = thread.getLooper();
  serviceHandler = new ServiceHandler(serviceLooper);
}

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show();

    // For each start request, send a message to start a job and deliver the
    // start ID so we know which request we're stopping when we finish the job
    Message msg = serviceHandler.obtainMessage();
    msg.arg1 = startId;
    serviceHandler.sendMessage(msg);

    // If we get killed, after returning from here, restart
    return START_STICKY;
}
```

```
  @Override
  public IBinder onBind(Intent intent) {
      // We don't provide binding, so return null
      return null;
  }

  @Override
  public void onDestroy() {
    Toast.makeText(this, "service done", Toast.LENGTH_SHORT).show();
  }
}
```

The example code handles all incoming calls in onStartCommand() and posts the work to a Handler running on a background thread. It works just like an IntentService and processes all requests serially, one after another. You could change the code to run the work on a thread pool, for example, if you'd like to run multiple requests simultaneously.

Notice that the onStartCommand() method must return an integer. The integer is a value that describes how the system should continue the service in the event that the system kills it. The return value from onStartCommand() must be one of the following constants:

START_NOT_STICKY

If the system kills the service after onStartCommand() returns, do not recreate the service unless there are pending intents to deliver. This is the safest option to avoid running your service when not necessary and when your application can simply restart any unfinished jobs.

START_STICKY

If the system kills the service after onStartCommand() returns, recreate the service and call onStartCommand(), but do not redeliver the last intent. Instead, the system calls onStartCommand() with a null intent unless there are pending intents to start the service. In that case, those intents are delivered. This is suitable for media players (or similar services) that are not executing commands but are running indefinitely and waiting for a job.

START_REDELIVER_INTENT

If the system kills the service after onStartCommand() returns, recreate the service and call onStartCommand() with the last intent that was delivered to the service. Any pending intents are delivered in turn. This is suitable for services that are actively performing a job that should be immediately resumed, such as downloading a file.

Starting Service

You can start a service from an activity or other application component by passing an Intent to startService() or startForegroundService(). The Android system calls the service's onStartCommand() method and passes it the Intent, which specifies which service to start.

For example, an activity can start the example service in the previous section (HelloService) using an explicit intent with startService(), as shown here:

```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

The startService() method returns immediately, and the Android system calls the service's onStartCommand() method. If the service isn't already running, the system first calls onCreate(), and then it calls onStartCommand().

If the service doesn't also provide binding, the intent that is delivered with startService() is the only mode of communication between the application component and the service. However, if you want the service to send a result back, the client that starts the service can create a PendingIntent for a broadcast (with getBroadcast()) and deliver it to the service in the Intent that starts the service. The service can then use the broadcast to deliver a result.

Stopping a Service

A started service must manage its own lifecycle. That is, the system doesn't stop or destroy the service unless it must recover system memory and the service continues to run after onStartCommand() returns. The service must stop itself by calling stopSelf(), or another component can stop it by calling stopService().

Once requested to stop with stopSelf() or stopService(), the system destroys the service as soon as possible.

Foreground Service

Apps that target Android 9 (API level 28) or higher and use foreground services must request the FOREGROUND_SERVICE permission, as shown in the following code snippet. This is a normal permission, so the system automatically grants it to the requesting app.

Start Foreground Service

Before you request the system to run a service as a foreground service, start the service itself:

```
Context context = getApplicationContext();
Intent intent = new Intent(...); // Build the intent for the service
context.startForegroundService(intent);
```

Inside the service, usually in onStartCommand(), you can request that your service run in the foreground. To do so, call startForeground(). This method takes two parameters: a positive integer that uniquely identifies the notification in the status bar and the Notification object itself.

```
Intent notificationIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingIntent =
        PendingIntent.getActivity(this, 0, notificationIntent, 0);

Notification notification =
          new Notification.Builder(this, CHANNEL_DEFAULT_IMPORTANCE)
    .setContentTitle(getText(R.string.notification_title))
    .setContentText(getText(R.string.notification_message))
    .setSmallIcon(R.drawable.icon)
    .setContentIntent(pendingIntent)
    .setTicker(getText(R.string.ticker_text))
    .build();

// Notification ID cannot be 0.
startForeground(ONGOING_NOTIFICATION_ID, notification);
```

Remove a Service from the foreground

To remove the service from the foreground, call stopForeground(). This method takes a boolean, which indicates whether to remove the status bar notification as well. Note that the service continues to run. If you stop the service while it's running in the foreground, its notification is removed.

Creating a Bound Service

A bound service is the server in a client-server interface. It allows components (such as activities) to bind to the service, send requests, receive responses, and perform interprocess communication (IPC). A bound service typically lives only while it serves another application component and does not run in the background indefinitely.

When creating a service that provides binding, you must provide an IBinder that provides the programming interface that clients can use to interact with the service. There are three ways you can define the interface:

Extending the Binder class

If your service is private to your own application and runs in the same process as the client (which is common), you should create your interface by extending the Binder class and returning an instance of it from onBind(). The client receives the Binder and can use it to directly access public methods available in either the Binder implementation or the Service.

This is the preferred technique when your service is merely a background worker for your own application. The only reason you would not create your interface this way is because your service is used by other applications or across separate processes.

Using a Messenger

If you need your interface to work across different processes, you can create an interface for the service with a Messenger. In this manner, the service defines a Handler that responds to different types of Message objects. This Handler is the basis for a Messenger that can then share an IBinder with the client, allowing the client to send commands to the service using Message objects. Additionally, the client can define a Messenger of its own, so the service can send messages back.

This is the simplest way to perform interprocess communication (IPC), because the Messenger queues all requests into a single thread so that you don't have to design your service to be thread-safe.

Using AIDL

Android Interface Definition Language (AIDL) decomposes objects into primitives that the operating system can understand and marshals them across processes to perform IPC. The previous technique, using a Messenger, is actually based on AIDL as its underlying structure. As mentioned above, the Messenger creates a queue of all the client requests in a single thread, so the service receives requests one at a time. If, however, you want your service to handle multiple requests simultaneously, then you can use AIDL directly. In this case, your service must be thread-safe and capable of multi-threading.

To use AIDL directly, you must create an .aidl file that defines the programming interface. The Android SDK tools use this file to generate an abstract class that implements the interface and handles IPC, which you can then extend within your service.

Extending the Binder Class

If your service is used only by the local application and does not need to work across processes, then you can implement your own Binder class that provides your client direct access to public methods in the service.

Here's how to set it up:

- In your service, create an instance of Binder that does one of the following:
    - Contains public methods that the client can call.
    - Returns the current Service instance, which has public methods the client can call.
    - Returns an instance of another class hosted by the service with public methods the client can call.
- Return this instance of Binder from the onBind() callback method.
- In the client, receive the Binder from the onServiceConnected() callback method and make calls to the bound service using the methods provided.

For example, here's a service that provides clients with access to methods in the service through a Binder implementation:

```
public class LocalService extends Service {
    // Binder given to clients
    private final IBinder binder = new LocalBinder();
    // Random number generator
    private final Random mGenerator = new Random();

    /**
     * Class used for the client Binder.  Because we know this service always
     * runs in the same process as its clients, we don't need to deal with IPC.
     */
    public class LocalBinder extends Binder {
        LocalService getService() {
            // Return this instance of LocalService so clients can call public methods
            return LocalService.this;
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return binder;
    }

    /** method for clients */
    public int getRandomNumber() {
      return mGenerator.nextInt(100);
    }
}
```

The LocalBinder provides the getService() method for clients to retrieve the current instance of LocalService. This allows clients to call public methods in the service. For example, clients can call getRandomNumber() from the service.

Here's an activity that binds to LocalService and calls getRandomNumber() when a button is clicked:

```
public class BindingActivity extends Activity {
    LocalService mService;
    boolean mBound = false;

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

    @Override
    protected void onStart() {
        super.onStart();
        // Bind to LocalService
        Intent intent = new Intent(this, LocalService.class);
        bindService(intent, connection, Context.BIND_AUTO_CREATE);
    }

    @Override
    protected void onStop() {
        super.onStop();
        unbindService(connection);
        mBound = false;
    }
```

```
    /** Called when a button is clicked (the button in the layout file attaches to
      * this method with the android:onClick attribute) */
    public void onButtonClick(View v) {
        if (mBound) {
            // Call a method from the LocalService.
            // However, if this call were something that might hang, then this request should
            // occur in a separate thread to avoid slowing down the activity performance.
            int num = mService.getRandomNumber();
            Toast.makeText(this, "number: " + num, Toast.LENGTH_SHORT).show();
        }
    }

    /** Defines callbacks for service binding, passed to bindService() */
    private ServiceConnection connection = new ServiceConnection() {

        @Override
        public void onServiceConnected(ComponentName className,
                IBinder service) {
            // We've bound to LocalService, cast the IBinder and get LocalService instance
            LocalBinder binder = (LocalBinder) service;
            mService = binder.getService();
            mBound = true;
        }

        @Override
        public void onServiceDisconnected(ComponentName arg0) {
            mBound = false;
        }
    };
}
```

Using a Messenger

If you need your service to communicate with remote processes, then you can use a Messenger to provide the interface for your service. This technique allows you to perform interprocess communication (IPC) without the need to use AIDL.

Using a Messenger for your interface is simpler than using AIDL because Messenger queues all calls to the service. A pure AIDL interface sends simultaneous requests to the service, which must then handle multi-threading.

For most applications, the service doesn't need to perform multi-threading, so using a Messenger allows the service to handle one call at a time. If it's important that your service be multi-threaded, use AIDL to define your interface.

Here's a summary of how to use a Messenger:

- The service implements a Handler that receives a callback for each call from a client.
- The service uses the Handler to create a Messenger object (which is a reference to the Handler).
- The Messenger creates an IBinder that the service returns to clients from onBind().
- Clients use the IBinder to instantiate the Messenger (that references the service's Handler),

which the client uses to send Message objects to the service.

- The service receives each Message in its Handler—specifically, in the handleMessage() method.

In this way, there are no methods for the client to call on the service. Instead, the client delivers messages (Message objects) that the service receives in its Handler.

Here's a simple example service that uses a Messenger interface:

```
public class MessengerService extends Service {
    /**
     * Command to the service to display a message
     */
    static final int MSG_SAY_HELLO = 1;

    /**
     * Handler of incoming messages from clients.
     */
    static class IncomingHandler extends Handler {
        private Context applicationContext;

        IncomingHandler(Context context) {
            applicationContext = context.getApplicationContext();
        }

        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MSG_SAY_HELLO:
                    Toast.makeText(applicationContext, "hello!", Toast.LENGTH_SHORT).show();
                    break;
                default:
                    super.handleMessage(msg);
            }
        }
    }
```

```
    /**
     * Target we publish for clients to send messages to IncomingHandler.
     */
    Messenger mMessenger;

    /**
     * When binding to the service, we return an interface to our messenger
     * for sending messages to the service.
     */
    @Override
    public IBinder onBind(Intent intent) {
        Toast.makeText(getApplicationContext(), "binding", Toast.LENGTH_SHORT).show();
        mMessenger = new Messenger(new IncomingHandler(this));
        return mMessenger.getBinder();
    }
}
```

Notice that the handleMessage() method in the Handler is where the service receives the incoming Message and decides what to do, based on the what member.

All that a client needs to do is create a Messenger based on the IBinder returned by the service and send a message using send(). For example, here's a simple activity that binds to the service and delivers the MSG_SAY_HELLO message to the service:

```
public class ActivityMessenger extends Activity {
    /** Messenger for communicating with the service. */
    Messenger mService = null;

    /** Flag indicating whether we have called bind on the service. */
    boolean bound;

    /**
     * Class for interacting with the main interface of the service.
     */
    private ServiceConnection mConnection = new ServiceConnection() {
        public void onServiceConnected(ComponentName className, IBinder service) {
            // This is called when the connection with the service has been
            // established, giving us the object we can use to
            // interact with the service.  We are communicating with the
            // service using a Messenger, so here we get a client-side
            // representation of that from the raw IBinder object.
            mService = new Messenger(service);
            bound = true;
        }

        public void onServiceDisconnected(ComponentName className) {
            // This is called when the connection with the service has been
            // unexpectedly disconnected -- that is, its process crashed.
            mService = null;
            bound = false;
        }
    };
```

```
    public void sayHello(View v) {
        if (!bound) return;
        // Create and send a message to the service, using a supported 'what' value
        Message msg = Message.obtain(null, MessengerService.MSG_SAY_HELLO, 0, 0);
        try {
            mService.send(msg);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

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

    @Override
    protected void onStart() {
        super.onStart();
        // Bind to the service
        bindService(new Intent(this, MessengerService.class), mConnection,
            Context.BIND_AUTO_CREATE);
    }
```

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

    @Override
    protected void onStart() {
        super.onStart();
        // Bind to the service
        bindService(new Intent(this, MessengerService.class), mConnection,
            Context.BIND_AUTO_CREATE);
    }

    @Override
    protected void onStop() {
        super.onStop();
        // Unbind from the service
        if (bound) {
            unbindService(mConnection);
            bound = false;
        }
    }
}
```

Notice that this example does not show how the service can respond to the client. If you want the service to respond, you need to also create a Messenger in the client. When the client receives the onServiceConnected() callback, it sends a Message to the service that includes the client's Messenger in the replyTo parameter of the send() method.

Binding to a Service

Application components (clients) can bind to a service by calling bindService(). The Android system then calls the service's onBind() method, which returns an IBinder for interacting with the service.

The binding is asynchronous, and bindService() returns immediately without returning the IBinder to the client. To receive the IBinder, the client must create an instance of ServiceConnection and pass it to bindService(). The ServiceConnection includes a callback method that the system calls to deliver the IBinder.

To bind to a service from your client, follow these steps:

- Implement ServiceConnection.
- Your implementation must override two callback methods:
- onServiceConnected()
    - The system calls this to deliver the IBinder returned by the service's onBind() method.
- onServiceDisconnected()
    - The Android system calls this when the connection to the service is unexpectedly lost, such as when the service has crashed or has been killed. This is not called when the client unbinds.
- Call bindService(), passing the ServiceConnection implementation.
- When the system calls your onServiceConnected() callback method, you can begin making calls to the service, using the methods defined by the interface.
- To disconnect from the service, call unbindService().

  If your client is still bound to a service when your app destroys the client, destruction causes the client to unbind. It is better practice to unbind the client as soon as it is done interacting with the service. Doing so allows the idle service to shut down. For more information about appropriate times to bind and unbind, see Additional notes.

The following example connects the client to the service created above by extending the Binder class, so all it must do is cast the returned IBinder to the LocalBinder class and request the LocalService instance:

```
LocalService mService;
private ServiceConnection mConnection = new ServiceConnection() {
    // Called when the connection with the service is established
    public void onServiceConnected(ComponentName className, IBinder service) {
        // Because we have bound to an explicit
        // service that is running in our own process, we can
        // cast its IBinder to a concrete class and directly access it.
        LocalBinder binder = (LocalBinder) service;
        mService = binder.getService();
        mBound = true;
    }

    // Called when the connection with the service disconnects unexpectedly
    public void onServiceDisconnected(ComponentName className) {
        Log.e(TAG, "onServiceDisconnected");
        mBound = false;
    }
};
```

With this ServiceConnection, the client can bind to a service by passing it to bindService(), as shown in the following example:

```
Intent intent = new Intent(this, LocalService.class);
bindService(intent, connection, Context.BIND_AUTO_CREATE);
```

The first parameter of bindService() is an Intent that explicitly names the service to bind.

The second parameter is the ServiceConnection object.

The third parameter is a flag indicating options for the binding. It should usually be BIND_AUTO_CREATE in order to create the service if it's not already alive. Other possible values are BIND_DEBUG_UNBIND and BIND_NOT_FOREGROUND, or 0 for none.

Important Links

Services overview | Android Developers

JobIntentService | Android Developers

Foreground services | Android Developers

Bound services overview | Android Developers

Lab Tasks

Activity 1

Run the following coding examples:

- Ibtisam/Simple-Service (github.com)
- Ibtisam/Downloader-using-Service (github.com)

Activity 2

Run the following coding example:

Exercises

1. Create one example of bound service that print the time to a file after 1 minutes and runs for infinite time. The service will print when in starts mode and no printing when in stop mode. Start and Stop are controlled from an activity.
2. Create one example of un-bound service that print the time to a file after 1 minutes and runs for infinite time. Create another application to view the results stored in a file or database where the messages were logged.

Assignment Deliverables

Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teacher.

LAB 14 Sensors and Third-party APIs

Objective

Most Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy and are useful if you want to monitor three-dimensional device movement or positioning, or you want to monitor changes in the ambient environment near a device.

For example, a game might track readings from a device's gravity sensor to infer complex user gestures and motions, such as tilt, shake, rotation, or swing. Likewise, a weather application might use a device's temperature sensor and humidity sensor to calculate and report the dewpoint, or a travel application might use the geomagnetic field sensor and accelerometer to report a compass bearing.

In this lab you will learn the implementation of several sensors provided by the Android.

Scope

The scope of this lab is the student's ability to:

- Understand the basics about Android Sensors Framework.
- Using motion, environment and position sensors to get raw data.

Useful Concepts

The Android platform supports three broad categories of sensors:

Motion sensors

These sensors measure acceleration forces and rotational forces along three axes. This category includes accelerometers, gravity sensors, gyroscopes, and rotational vector sensors.

Environmental sensors

These sensors measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity. This category includes barometers, photometers, and thermometers.

Position sensors

These sensors measure the physical position of a device. This category includes orientation sensors and magnetometers.

You can access sensors available on the device and acquire raw sensor data by using the Android sensor framework. The sensor framework provides several classes and interfaces that help you perform a wide variety of sensor-related tasks. For example, you can use the sensor framework to

do the following:

- Determine which sensors are available on a device.
- Determine an individual sensor's capabilities, such as its maximum range, manufacturer, power requirements, and resolution.
- Acquire raw sensor data and define the minimum rate at which you acquire sensor data.
- Register and unregister sensor event listeners that monitor sensor changes.

Android Sensors Framework

The Android sensor framework lets you access many types of sensors. Some of these sensors are hardware-based and some are software-based. Hardware-based sensors are physical components built into a handset or tablet device. They derive their data by directly measuring specific environmental properties, such as acceleration, geomagnetic field strength, or angular change.

Few Android-powered devices have every type of sensor. For example, most handset devices and tablets have an accelerometer and a magnetometer, but fewer devices have barometers or thermometers. Also, a device can have more than one sensor of a given type. For example, a device can have two gravity sensors, each one having a different range.

You can access these sensors and acquire raw sensor data by using the Android sensor framework. The sensor framework is part of the android.hardware package and includes the following classes and interfaces:

SensorManager

You can use this class to create an instance of the sensor service. This class provides various methods for accessing and listing sensors, registering and unregistering sensor event listeners, and acquiring orientation information. This class also provides several sensor constants that are used to report sensor accuracy, set data acquisition rates, and calibrate sensors.

Sensor

You can use this class to create an instance of a specific sensor. This class provides various methods that let you determine a sensor's capabilities.

SensorEvent

The system uses this class to create a sensor event object, which provides information about a sensor event. A sensor event object includes the following information: the raw sensor data, the type of sensor that generated the event, the accuracy of the data, and the timestamp for the event.

SensorEventListener

You can use this interface to create two callback methods that receive notifications (sensor events) when sensor values change or when sensor accuracy changes.

In a typical application you use these sensor-related APIs to perform two basic tasks:

Identifying sensors and sensor capabilities

Identifying sensors and sensor capabilities at runtime is useful if your application has features that

rely on specific sensor types or capabilities. For example, you may want to identify all of the sensors that are present on a device and disable any application features that rely on sensors that are not present. Likewise, you may want to identify all of the sensors of a given type so you can choose the sensor implementation that has the optimum performance for your application.

Monitor sensor events

Monitoring sensor events is how you acquire raw sensor data. A sensor event occurs every time a sensor detects a change in the parameters it is measuring. A sensor event provides you with four pieces of information: the name of the sensor that triggered the event, the timestamp for the event, the accuracy of the event, and the raw sensor data that triggered the event.

Table 1: Sensor types supported by the Android platform.

Sensor	Type	Description	Common Uses
TYPE_ACCELEROMETER	Hardware	Measures the acceleration force in m/s^2 that is applied to a device on all three physical axes (x, y, and z), including the force of gravity.	Motion detection (shake, tilt, etc.).
TYPE_AMBIENT_TEMPERATURE	Hardware	Measures the ambient room temperature in degrees Celsius (°C). See note below.	Monitoring air temperatures.
TYPE_GRAVITY	Software or Hardware	Measures the force of gravity in m/s^2 that is applied to a device on all three physical axes (x, y, z).	Motion detection (shake, tilt, etc.).
TYPE_GYROSCOPE	Hardware	Measures a device's rate of rotation in rad/s around each of the three physical axes (x, y, and z).	Rotation detection (spin, turn, etc.).
TYPE_LIGHT	Hardware	Measures the ambient light level (illumination) in lx.	Controlling screen brightness.
TYPE_LINEAR_ACCELERATION	Software or Hardware	Measures the acceleration force in m/s^2 that is applied to a device on all three physical axes (x, y, and z), excluding the force of gravity.	Monitoring acceleration along a single axis.
TYPE_MAGNETIC_FIELD	Hardware	Measures the ambient geomagnetic field for all three physical axes (x, y, z) in μT.	Creating a compass.
TYPE_ORIENTATION	Software	Measures degrees of rotation that a device makes around all three physical axes (x, y, z). As of API level 3 you can obtain the inclination matrix and rotation matrix for a device by using the gravity sensor and the geomagnetic field sensor in conjunction with the getRotationMatrix() method.	Determining device position.
TYPE_PRESSURE	Hardware	Measures the ambient air pressure in hPa or mbar.	Monitoring air pressure changes.
TYPE_PROXIMITY	Hardware	Measures the proximity of an object in cm relative to the view screen of a device. This sensor is typically used to determine whether a handset is being held up to a person's ear.	Phone position during a call.
TYPE_RELATIVE_HUMIDITY	Hardware	Measures the relative ambient humidity in percent (%).	Monitoring dewpoint, absolute, and relative humidity.
TYPE_ROTATION_VECTOR	Software or	Measures the orientation of a device by	Motion detection and

	Hardware	providing the three elements of the device's rotation vector.	rotation detection.
TYPE_TEMPERATURE	Hardware	Measures the temperature of the device in degrees Celsius (°C). This sensor implementation varies across devices and this sensor was replaced with the TYPE_AMBIENT_TEMPERATURE sen sor in API Level 14	Monitoring temperatures.

Identifying Sensors and Sensor capabilities

o identify the sensors that are on a device you first need to get a reference to the sensor service. To do this, you create an instance of the SensorManager class by calling the getSystemService() method and passing in the SENSOR_SERVICE argument. For example:

```
private SensorManager sensorManager;
...
sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
```

Next, you can get a listing of every sensor on a device by calling the getSensorList() method and using the TYPE_ALL constant. For example:

```
List<Sensor> deviceSensors = sensorManager.getSensorList(Sensor.TYPE_ALL);
```

If you want to list all of the sensors of a given type, you could use another constant instead of TYPE_ALL such as TYPE_GYROSCOPE, TYPE_LINEAR_ACCELERATION, or TYPE_GRAVITY.

You can also determine whether a specific type of sensor exists on a device by using the getDefaultSensor() method and passing in the type constant for a specific sensor. If a device has more than one sensor of a given type, one of the sensors must be designated as the default sensor. If a default sensor does not exist for a given type of sensor, the method call returns null, which means the device does not have that type of sensor. For example, the following code checks whether there's a magnetometer on a device:

```
private SensorManager sensorManager;
...
sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
if (sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD) != null){
    // Success! There's a magnetometer.
} else {
    // Failure! No magnetometer.
}
```

In addition to listing the sensors that are on a device, you can use the public methods of the Sensor class to determine the capabilities and attributes of individual sensors. This is useful if you want your application to behave differently based on which sensors or sensor capabilities are available on a device. For example, you can use the getResolution() and getMaximumRange() methods to obtain a sensor's resolution and maximum range of measurement. You can also use the getPower() method to obtain a sensor's power requirements.

Two of the public methods are particularly useful if you want to optimize your application for different manufacturer's sensors or different versions of a sensor. For example, if your application needs to monitor user gestures such as tilt and shake, you could create one set of data filtering rules and optimizations for newer devices that have a specific vendor's gravity sensor, and another

set of data filtering rules and optimizations for devices that do not have a gravity sensor and have only an accelerometer. The following code sample shows you how you can use the getVendor() and getVersion() methods to do this. In this sample, we're looking for a gravity sensor that lists Google LLC as the vendor and has a version number of 3. If that particular sensor is not present on the device, we try to use the accelerometer.

```
private SensorManager sensorManager;
private Sensor mSensor;

...

sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mSensor = null;

if (sensorManager.getDefaultSensor(Sensor.TYPE_GRAVITY) != null){
    List<Sensor> gravSensors = sensorManager.getSensorList(Sensor.TYPE_GRAVITY);
    for(int i=0; i<gravSensors.size(); i++) {
        if ((gravSensors.get(i).getVendor().contains("Google LLC")) &&
           (gravSensors.get(i).getVersion() == 3)){
            // Use the version 3 gravity sensor.
            mSensor = gravSensors.get(i);
        }
    }
}
if (mSensor == null){
    // Use the accelerometer.
    if (sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER) != null){
        mSensor = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
    } else{
        // Sorry, there are no accelerometers on your device.
        // You can't play this game.
    }
}
```

Another useful method is the getMinDelay() method, which returns the minimum time interval (in microseconds) a sensor can use to sense data. Any sensor that returns a non-zero value for the getMinDelay() method is a streaming sensor. Streaming sensors sense data at regular intervals and were introduced in Android 2.3 (API Level 9). If a sensor returns zero when you call the getMinDelay() method, it means the sensor is not a streaming sensor because it reports data only when there is a change in the parameters it is sensing.

The getMinDelay() method is useful because it lets you determine the maximum rate at which a sensor can acquire data. If certain features in your application require high data acquisition rates or a streaming sensor, you can use this method to determine whether a sensor meets those requirements and then enable or disable the relevant features in your application accordingly.

Monitoring Sensor Events

To monitor raw sensor data you need to implement two callback methods that are exposed through the SensorEventListener interface: onAccuracyChanged() and onSensorChanged(). The Android system calls these methods whenever the following occurs:

- **A sensor's accuracy changes.**

  In this case the system invokes the onAccuracyChanged() method, providing you with a reference to the Sensor object that changed and the new accuracy of the sensor. Accuracy is represented by one of four status constants: SENSOR_STATUS_ACCURACY_LOW, SENSOR_STATUS_ACCURACY_MEDIUM, SENSOR_STATUS_ACCURACY_HIGH, or SENSOR_STATUS_UNRELIABLE.

- **A sensor reports a new value.**

  In this case the system invokes the onSensorChanged() method, providing you with a SensorEvent object. A SensorEvent object contains information about the new sensor data, including: the accuracy of the data, the sensor that generated the data, the timestamp at which the data was generated, and the new data that the sensor recorded.

The following code shows how to use the onSensorChanged() method to monitor data from the light sensor. This example displays the raw sensor data in a TextView that is defined in the main.xml file as sensor_data.

```
public class SensorActivity extends Activity implements SensorEventListener {
    private SensorManager sensorManager;
    private Sensor mLight;

    @Override
    public final void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        mLight = sensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
    }

    @Override
    public final void onAccuracyChanged(Sensor sensor, int accuracy) {
        // Do something here if sensor accuracy changes.
    }

    @Override
    public final void onSensorChanged(SensorEvent event) {
        // The light sensor returns a single value.
        // Many sensors return 3 values, one for each axis.
        float lux = event.values[0];
        // Do something with this sensor value.
    }
```

```
    @Override
    protected void onResume() {
        super.onResume();
        sensorManager.registerListener(this, mLight, SensorManager.SENSOR_DELAY_NORMAL);
    }

    @Override
    protected void onPause() {
        super.onPause();
        sensorManager.unregisterListener(this);
    }
}
```

In this example, the default data delay (SENSOR_DELAY_NORMAL) is specified when the registerListener() method is invoked. The data delay (or sampling rate) controls the interval at which sensor events are sent to your application via the onSensorChanged() callback method. The default data delay is suitable for monitoring typical screen orientation changes and uses a delay of 200,000 microseconds. You can specify other data delays, such as SENSOR_DELAY_GAME (20,000 microsecond delay), SENSOR_DELAY_UI (60,000 microsecond delay), or SENSOR_DELAY_FASTEST (0 microsecond delay). As of Android 3.0 (API Level 11) you can also specify the delay as an absolute value (in microseconds).

It's also important to note that this example uses the onResume() and onPause() callback methods to register and unregister the sensor event listener. As a best practice you should always disable sensors you don't need, especially when your activity is paused. Failing to do so can drain the battery in just a few hours because some sensors have substantial power requirements and can use up battery power quickly. The system will not disable sensors automatically when the screen turns off.

Detecting Sensors at Runtime

If your application uses a specific type of sensor, but doesn't rely on it, you can use the sensor framework to detect the sensor at runtime and then disable or enable application features as appropriate. For example, a navigation application might use the temperature sensor, pressure sensor, GPS sensor, and geomagnetic field sensor to display the temperature, barometric pressure, location, and compass bearing. If a device doesn't have a pressure sensor, you can use the sensor framework to detect the absence of the pressure sensor at runtime and then disable the portion of your application's UI that displays pressure. For example, the following code checks whether there's a pressure sensor on a device:

```
private SensorManager sensorManager;
...
sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
if (sensorManager.getDefaultSensor(Sensor.TYPE_PRESSURE) != null){
    // Success! There's a pressure sensor.
} else {
    // Failure! No pressure sensor.
}
```

Using Google Play filters to target specific sensor configurations

If you are publishing your application on Google Play you can use the <uses-feature> element in your manifest file to filter your application from devices that do not have the appropriate sensor configuration for your application. The <uses-feature> element has several hardware descriptors that let you filter applications based on the presence of specific sensors. The sensors you can list include: accelerometer, barometer, compass (geomagnetic field), gyroscope, light, and proximity. The following is an example manifest entry that filters apps that do not have an accelerometer:

```
<uses-feature android:name="android.hardware.sensor.accelerometer"
              android:required="true" />
```

Best Practices for Accessing and Using Sensors

As you design your sensor implementation, be sure to follow the guidelines that are discussed in this section. These guidelines are recommended best practices for anyone who is using the sensor framework to access sensors and acquire sensor data.

Only gather sensor data in the foreground

On devices running Android 9 (API level 28) or higher, apps running in the background have the following restrictions:

- Sensors that use the continuous reporting mode, such as accelerometers and gyroscopes, don't receive events.
- Sensors that use the on-change or one-shot reporting modes don't receive events.

Given these restrictions, it's best to detect sensor events either when your app is in the foreground or as part of a foreground service.

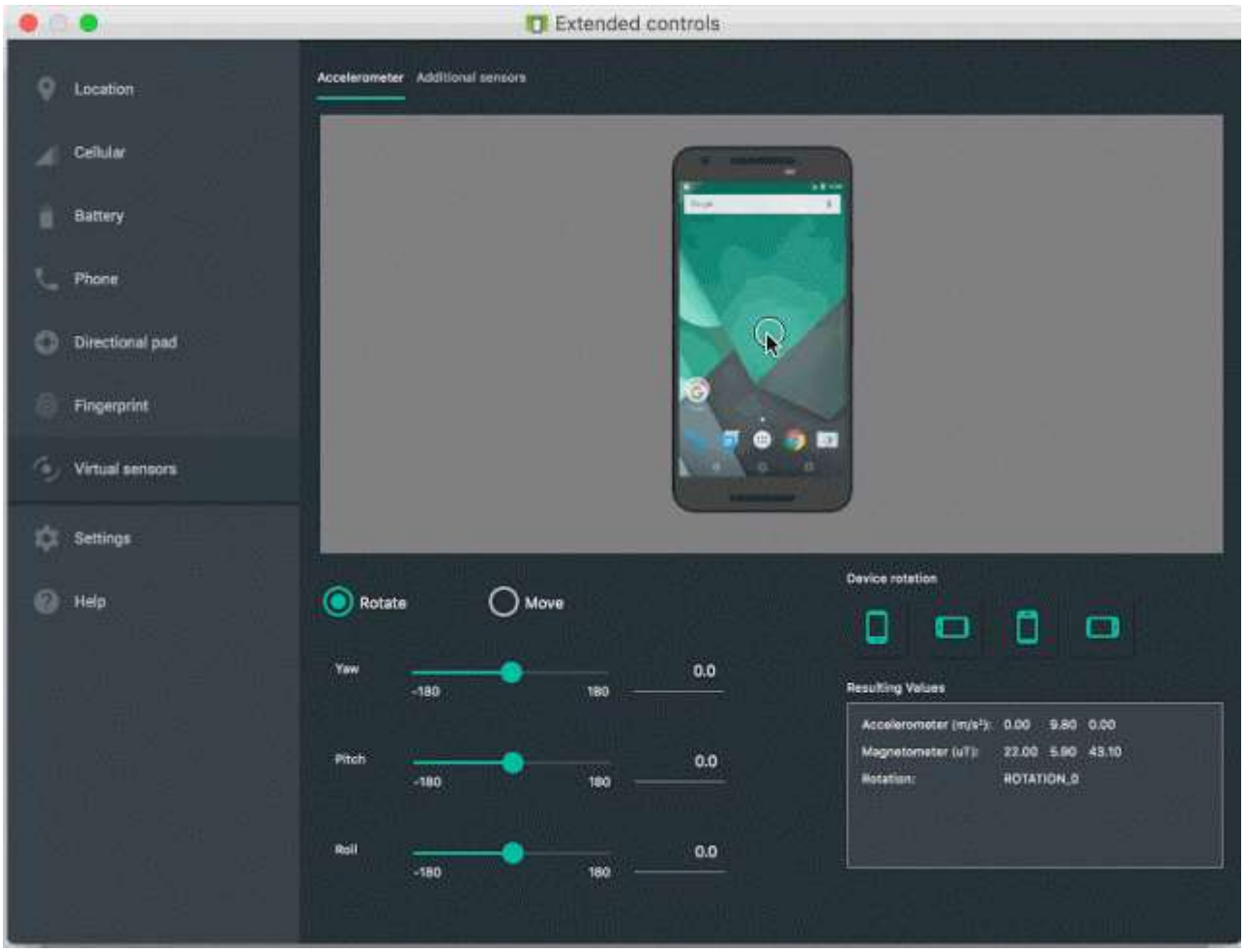
Unregister Sensor Listeners

Be sure to unregister a sensor's listener when you are done using the sensor or when the sensor activity pauses. If a sensor listener is registered and its activity is paused, the sensor will continue to acquire data and use battery resources unless you unregister the sensor. The following code shows how to use the onPause() method to unregister a listener:

```
private SensorManager sensorManager;
...
@Override
protected void onPause() {
    super.onPause();
    sensorManager.unregisterListener(this);
}
```

Test with Android Emulator

The Android Emulator includes a set of virtual sensor controls that allow you to test sensors such as accelerometer, ambient temperature, magnetometer, proximity, light, and more.



Don't block the onSensorChanged() method

Sensor data can change at a high rate, which means the system may call the onSensorChanged(SensorEvent) method quite often. As a best practice, you should do as little as possible within the onSensorChanged(SensorEvent) method so you don't block it. If your application requires you to do any data filtering or reduction of sensor data, you should perform that work outside of the onSensorChanged(SensorEvent) method.

Avoid using deprecated methods or sensor types

Several methods and constants have been deprecated. In particular, the TYPE_ORIENTATION sensor type has been deprecated. To get orientation data you should use the getOrientation() method instead. Likewise, the TYPE_TEMPERATURE sensor type has been deprecated. You should use the TYPE_AMBIENT_TEMPERATURE sensor type instead on devices that are running Android 4.0.

Verify sensors before you use them

Always verify that a sensor exists on a device before you attempt to acquire data from it. Don't assume that a sensor exists simply because it's a frequently-used sensor. Device manufacturers are not required to provide any particular sensors in their devices.

Choose sensor delays carefully

When you register a sensor with the registerListener() method, be sure you choose a delivery rate

that is suitable for your application or use-case. Sensors can provide data at very high rates. Allowing the system to send extra data that you don't need wastes system resources and uses battery power.

Important Links

Sensors Overview | Android Developers

Motion sensors | Android Developers

Position sensors | Android Developers

Environment sensors | Android Developers

Lab Tasks

Activity 1

Run the following code example:

- samples/ApiDemos/src/com/example/android/apis/os/RotationVectorDemo.java - platform/development - Git at Google (googlesource.com)

Activity 2

Follow the examples in the important links above and implement them.

Exercises

- Create a Simple App that gets raw data from Motion and Position sensors upon pressing a button and store it in Firebase Firestore.

Assignment Deliverables

Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teacher.

LAB 15 Cross-Platform Development

Objective

The developer's pipe dream: to write code once that runs on all the platforms out there, adapting and adjusting to the platform's capabilities on-the-fly without intervention. The good news is that it's usually easier to think in terms of mobile vs. non-mobile platforms when selecting from the plethora of solutions out there. The main objectives of the lab are:

- Discussing the approaches to cross-platform development
- Covering the pros and cons of the cross-platform approach
- Covering points to consider when deciding if cross-platform development is right for your app
- Common cross-platform development languages and protocols

Scope

Useful Concepts

What is cross-platform mobile development?

In a nutshell, cross-platform development is the creation of an app that's compatible across all mobile devices (i.e. Android and iOS) using a single codebase.

Cross-platform development is possible for a couple of reasons:

- Backend servers handle the execution of application code, not the operating systems or devices themselves.
- Developers can use subroutines like application programming interfaces (APIs) to speak to multiple programming languages and OSs.

The two types of cross-platform mobile development, native and hybrid HTML5, take advantage of these reasons when developing apps.

Native Cross-Platform

Each mobile OS runs on its own software development kit (DSK) and tech stack. Android apps, for example, are programmed using Java. iOS apps, on the other hand, are programmed using Objective-C and Swift. Developers, however, can work around these differences by creating an API that acts as a middleman between the code base and the OS.

Native cross-platform development is accomplished by using a third-party vendor which provides an integrated development environment (IDE) specific to this type of programming.

Hybrid HTML5

Since application code execution is handled on backend servers, developers can get around OS differences by developing aspects of an app's graphic user interface (GUI) in languages like CSS, HTML5, and JavaScript. Developers use HTML5, in particular, because today's mobile OSs have advanced browser capabilities (HTML5 is the markup language used on the internet).

By coding in HTML5, developers create a "hybrid" app that consists of the OS's native frame and the code that's executed in a WebView.

Advantages of cross-platform development over native?

Cross-platform mobile development offers several advantages to developers and companies alike. These all start with the fact that, by choosing the right cross-platform tools and having a strong development plan, developers can use up to 80 percent of the same codebase. This cut in time alone can:

- Reduce overall project costs: Development costs for even the most basic app start at $10 thousand. If you're developing native Android and iOS apps, you'll need to double that, then add another 30% to make up for Android's higher development costs. Any reductions in costs can save companies thousands.
- Speed up the time to release: Competition is fierce in today's mobile market, which is especially important in highly competitive industries or niches. Companies that are able to release their app, address bugs and push updates faster will have an edge.
- Optimize maintenance costs: Bug fixes and release updates can be handled faster as code changes would only need to be addressed once.
- Optimize unit testing: The time needed to write unit tests for each OS could be used instead to write more comprehensive tests. This, in turn, could reduce bugs.
- Optimize your development resources: Cross-platform development means your developers only need to know one language. This eliminates the need to either bring on another developer for a specific iOS or spend money to train your developers on a particular language.
- Increase your audience: By releasing your app on both iOS and Android, you expose your app to a wide audience faster.

Languages in cross-platform development

We've already discussed one advantage of cross-platform development: the ability to hire developers that only need an expertise in one language. Even better, cross-platform development can be accomplished with several programming languages, including:

Java

The gold standard of programming languages for cross-platform development, not to mention it's the core language used for Android.

Further reading: An introduction to Java

C++

Though not as sophisticated as Java, C++ will still get the job done for any cross-platform development project. The only drawback is that C++ may add some strain to mobile device resources.

Further reading: What is C++

JavaScript and HTML5

JavaScript is the programming language used for HTML5, the markup language designed to make mobile apps compatible with desktops. While each isn't particularly useful on their own, together, these two make for an excellent choice for cross-platform development.

Further reading: What is HTML5?

C#

Created by Microsoft, C# originally started off as an equivalent to the Objective-C language for Mac. However, C# has become a popular choice for cross-platform developers.

Further Reading: Why C# is among the most popular programming languages in the world.

Ruby

A mobile-specific programming language that's straightforward and meant to use as few resources as possible on mobile devices.

Further Reading: Beginner's Guide to Ruby

Fortunately, developers today have a variety of tools to choose from. Each tool offers something different, such as the use of certain programming languages. When researching tools for your development team, consider the following:

Apache Cordova/PhoneGap

One of the most popular cross-platform development frameworks is the open source Apache Cordova. The framework uses JavaScript, CSS, and HTML5 for app creation and gives developers several advantages, including direct access to a smartphone's assets (e.g. contact data, file storage, notifications). Cordova also includes a straightforward API and the ability to employ most JS frameworks.

Developers may also know the popular Apache Cordova cross-platform development product PhoneGap, which was purchased from creator Nitobi in 2011 by Adobe. PhoneGap is Cordova's cloud-based development tool, which eliminates the need for compilers, hardware, and SDKs altogether.

Xamarin

Another popular cross-platform tool is Xamarin, founded by the creators of the cross-platform implementations MonoTouch and MonoDroid. Using C#, developers can easily write and reuse their code across various platforms. Xamarin also simplifies cross-platform processes, such as the creation of dynamic layouts for iOS.

Xamarin integrates with Microsoft Visual Studio, which includes add-ins that allow for Android, iOS, and Windows development. Xamarin passed the 1 million developer milestone back in 2015.

Unity

Unity is a cross-platform game engine created by Unity Technologies. Unity is used to create both 2 and 3-dimensional games for consoles (Xbox, PlayStation, etc.), desktops (Windows, Linux, macOS), and mobile (Android and iOS). The engine primarily uses C# for coding.

Among the advantages of Unity are the availability of free plugins and the availability of detailed documentation to help developers with almost every aspect of the engine. However, Unity does have a steep learning curve and requires licensing fees for superior graphics and deployment, both of which can drive up development costs.

NativeScript

NativeScript is an open-source development platform that allows developers to build Android and iOS apps using their native UIs and development libraries. Programming is done mainly through JavaScript, though NativeScript also supports Angular and TypeScript. Furthermore, NativeScript allows developers to use most JavaScript libraries that don't rely on the internet.

For teams that need to create feature-rich apps, NativeScript is an excellent choice. However, keep in mind that NativeScript doesn't have as much documentation available as other platforms. This may affect development time if developers need to troubleshoot unfamiliar issues.

Sencha

Sencha is a web development framework specifically designed for mobile application development. Among Sencha's popular tools is Sencha Ext JS, which allows users to create HTML5 apps. Ext JS also includes over one hundred UI components and native-looking themes for Android, iOS, Windows, and

even Blackberry. Finally, Sencha integrates with other cross-platform platforms like Apache Cordova.

Despite these benefits, Sencha is far from free. The platform is currently priced at over three thousand per year for up to five developers. Pricing jumps up to over twelve thousand for twenty developers.

Appcelerator

Appcelerator is a development platform for building enterprise apps. The platform uses the common JavaScript language for creating native and cloud-connected mobile apps for Android, iOS, HTML5, and more.

Appcelerator comes with a few advantages for companies looking to build an internal enterprise app. The use of JavaScript makes it easier for companies to find skilled developers. Appcelerator also comes with an optional virtual private cloud option for companies that handle sensitive and private data.

Kony AppPlatform

Another enterprise-focused platform is Kony AppPlatform. This low-code development framework is great for companies that want to push their app out to workers with minimal development hassle. Developers use the drag-and-drop interface to build their apps using component from the Kony Marketplace or their own libraries.

RhoMobile Suite

RhoMobile's development platform is great for developers looking to build data-centric enterprise apps. RhoStudio includes the Eclipse plug-in so developers can test applications without the need for emulators or hardware. From a security standpoint, RhoMobile Suite also provides automatic data encryption. Applications can be built for mobile and non-mobile OSs.

Lab Tasks

Activity 1

IN the languages list there are better programming languages emerged now, Your tasks is to search on latest trends and find which language is mostly used in the industry now.

Justify your finalized language by giving at least five top apps developed in the language.

What is the future of the selected language?

Activity 2

From the language of your choice for cross-platform development develop a "Hello World" simple application and show the output to your teacher.

Activity 3

Search on the available tools available for creating the cross-platform apps. Also suggest the better tool to follow for development along with its advantages over others.

Exercises

For home exercise you need to develop a simple mobile app by extending your application of activity 2 and provide a user friendly use cases implementation in your app.

Assignment Deliverables

Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teacher.

LAB 16: Final Exam

Objectives

The purpose of this lab is to conduct the final exam based on the activities conducted throughout the semester.

Lab Tasks

The tasks will be decided by the respective course instructor/lab tutor.