ANDHRA
Engineering College

Mobile Application Development Lab (20A05706)

L A B O R A T O R Y   O B S E R V A T I O N

Department of

COMPUTER SCIENCE &ENGINEERING

Name	
Roll No.	
Class	IV B.TECH. I SEM
Branch	CSE
Regulation	R20
Name of the Lab	**Mobile Application Development Lab-20A05706**
Academic year	2025-26
Prepared By	Mr.O Devanand

VISION (AECN)

- To emerge as a leading Engineering institution imparting quality education

MISSION (AECN)

- IM_1 Implement Effective teaching-learning strategies for quality education
- IM_2 Build Congenial academic ambience for progressive learning
- IM_3 Facilitate Skill development through Industry-Institute initiatives
- IM_4 Groom environmentally conscious and socially responsible technocrats

DEPARTMENT OF COMPUTER SCIENCE &ENGINEERING

VISION

- To develop as a lead learning resource centre producing skilled professionals

MISSION

- DM_1 Provide dynamic and application oriented education through advanced teaching learning methodologies
- DM_2 Create sufficient physical infrastructural facilities to enhance learning
- DM_3 Strengthen the professional skills through effective Industry- Institute Interaction
- DM_4 Organize personality development activities to inculcate life skills and ethical values

ANDHRA ENGINEERING COLLEGE

(Approved by AICTE, New Delhi and Affiliated to JNTU, Anantapur)
3-76-2 ATMAKUR, SPSR NELLORE, AP 524322

Mobile Application Development Lab (20A05706)

L A B O R A T O R Y   O B S E R V A T I O N

Department of

COMPUTER SCIENCE &ENGINEERING

Name	
Roll No.	
Class	IV B.TECH. I SEM
Branch	CSE
Regulation	R20
Name of the Lab	**Mobile Application Development Lab-20A05706**
Academic year	2025-26

S.no	Name of the Experiment	Date	Signature

Android operating system's initial release was in the year 2008. Even at its start, the team behind the operating system built it on top of the shoulders of giants. Beyond the user interface that the Android OS presents at the surface level, it is made up of multiple layers. These layers include custom code and open-source technologies that have been under continuous development for decades.

Android has been developed through massive collaborative efforts and investments by many companies. The main company behind android development is Google. Other companies include device manufacturers such as Samsung, LG; processor manufacturers such as Intel and ARM, but to name a few.

When we talk about Android architecture, we mean how the Android system has been designed, segmented into layers, and built up to work as a system. Building such a complex system requires careful structuring to ensure all the components work together cohesively. Its architecture ensures that the many components function as a whole without crashing.

Layers

The following are the layers that compose the Android architecture as labeled on the diagram:

1. Application
2. Application Framework
3. Android Runtime and Core Libraries
4. Linux Kernel

Developing an operating system for mobile devices comes with a set of challenges. Using this layered architecture ensures that different problems are broken down and solved at different levels.

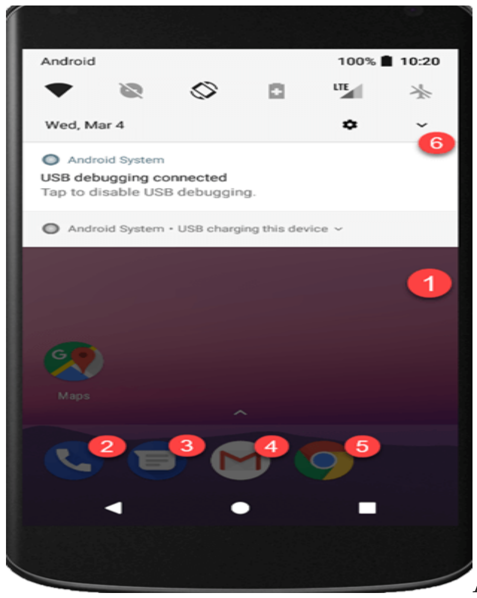
A layered architecture helps separate concerns and ensure android software developers don't have to deal with low-level problems at every turn. They can instead focus on delivering business value concerned with the layer they're working on.

Developers are working on making apps don't have to worry about the application framework implementation. That work is left for the system developers working on the Application framework.

The Application Framework developers work on developer experience and don't have to worry about the low-level drivers. Low-level system engineers can focus completely on low-level components such as Bluetooth or Audio drivers and the like.

Android's layered structure makes it possible to apply updates with bug fixes or improvements to each layer on its own. This ensures that changes across layers don't interfere with each other.

This makes it possible for people working at a different level of the OS to work with obstructing each other as new updates and releases are done.**Android Application**



Android Application

This is the layer that end-users interact with. It is on this layer where application developers publish their applications to run.

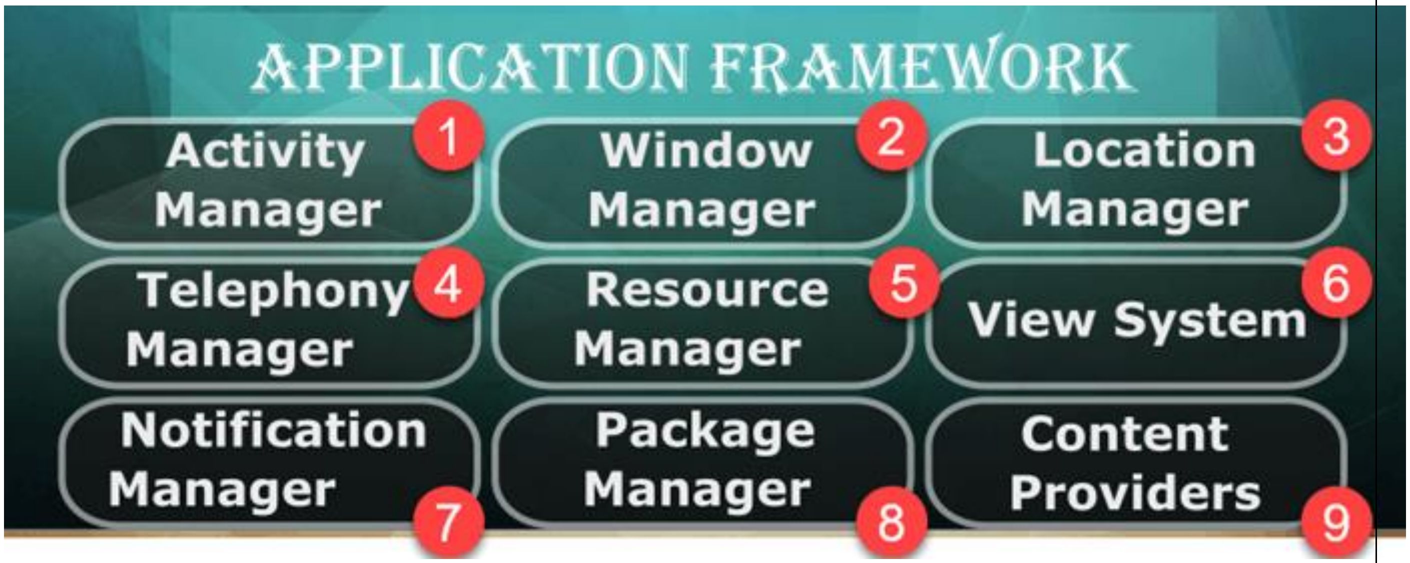
Android, by default, comes with a set of applications that make android devices usable from the offset.

1. **Home:** The Homepage on Android consists of launcher icons for commonly used applications that the end-user may want quick access to. You can start the apps by clicking on the launchers of these apps. At the very top of the screen, you have widgets that show network, battery level, date, and time.
2. **Contacts:** Android, by default, provides a means to store and retrieve contacts. Contact information is shared across other apps to enhance functionality.
3. **Messages:** Android provides the capability to send and receive SMS messages.
4. **Email:** Android comes with native support for email services. Setting up an Android device requires a Gmail account. Setting up Gmail activates other email-dependent components on Android devices. Some email dependent features include security and recovery mechanisms. Another email dependent feature is access to the Play Store, a marketplace for Android applications.
5. **Browser:** Android comes with a default browser.
6. **Notification Drawer:** Swiping down on the screen exposes the notification drawer. It provides application events that the user should be aware of. Above the notification are a set of shortcuts to some commonly used device settings that the users can toggle. These settings include on and off toggles for various hardware components such as Bluetooth and Wifi. Long pressing these events enables us to navigate to their configurations page.

This layer is also referred to as user-level in contrast to the layers below that are mostly tuned for application development. Application developers create and customize the experiences for their apps on this layer. The layers below the application layer are not customized by application developers. They are considered part of the system layer. These layers are customized by device manufacturers, Google android teams, or third parties who want to use the Android source code for their product or research.

Application Framework

The Android OS exposes the underlying libraries and features of the Android device that are using a Java API. This is what is known as the Android framework. The framework exposes a safe and uniform means to utilize Android device resources.



Application framework

1) Activity Manager

Applications use the Android activity component for presenting an entry point to the app. Android Activities are the components that house the user interface that app users interact with. As end-users interact with the Android device, they start, stop, and jump back and forth across many applications. Each navigation event triggers activation and deactivation of many activities in respective applications.

The Android ActivityManager is responsible for predictable and consistent behavior during application transitions. The ActivityManager provides a slot for app creators to have their apps react when the Android OS performs global actions. Applications can listen to events such as device rotation, app destruction due to memory shortage, an app being shifted out of focus, and so on.

Some examples of the way applications can react to these transitions include pausing activity in a game, stopping music playing during a phone call.

2) Window Manager

Android can determine screen information to determine the requirements needed to create windows for applications. Windows are the slots where we can view our app user interface. Android uses the Window manager to provide this information to the apps and the system as they run so that they can adapt to the mode the device is running on.

The Window Manager helps in delivering a customized app experience. Apps can fill the complete screen for an immersive experience or share the screen with other apps. Android enables this by allowing multi-windows for each app.

3) Location Manager

Most Android devices are equipped with GPS devices that can get user location using satellite information to which can go all the way to meters precision. Programmers can prompt for location permission from the users, deliver location, and aware experiences.

Android is also able to utilize wireless technologies to further enrich location details and increase coverage when devices are enclosed spaces. Android provides these features under the umbrella of the Location-Manager.

4) Telephony Manager

Most Android devices serve a primary role in telephony. Android uses TelephoneManager to combine hardware and software components to deliver telephony features. The hardware components include external parts such as the sim card, and device parts such as the microphone, camera, and speakers. The software components include native components such as dial pad, phone book, ringtone profiles. Using the TelephoneManager, a developer can extend or fine-tune the default calling functionality.

5) Resource Manager

Android app usually come with more than just code. They also have other resources such as icons, audio and video files, animations, text files, and the like. Android helps in making sure that there is efficient, responsive access to these resources. It also ensures that the right resources are delivered to the end-users. For example, the proper language text files are used when populating fields in the apps.

6) View System

Android also provides a means to easily create common visual components needed for app interaction. These components include widgets like buttons, image holders such as ImageView, components to display a list of items such as ListView, and many more. The components are premade but are also customizable to fit app developer needs and branding.

7) Notification Manager

The Notification Manager is responsible for informing Android users of application events. It does this by giving users visual, audio or vibration signals or a combination of them when an event occurs. These events have external and internal triggers. Some examples of internal triggers are low-battery status events that trigger a notification to show low battery. Another example is user-specified events like an alarm. Some examples of external triggers include new messages or new wifi networks detected.

Android provides a means for programmers and end-users to fine-tune the notifications system. This can help to guarantee they can send and receive notification events in a means that best suits them and their current environments.

8) Package Manager

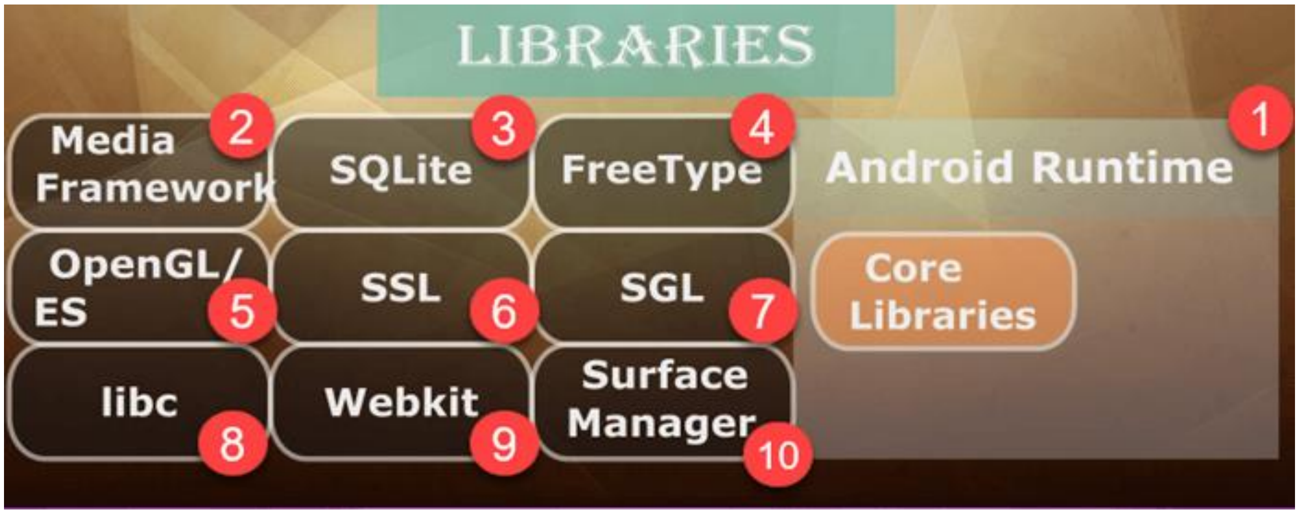
Android also provides access to information about installed applications. Android keeps track of application information such as installation and uninstallation events, permissions the app requests, and resource utilization such as memory consumption.

This information can enable developers to make their applications to activate or deactivate functionality depending on new features presented by companion apps.

9) Content Provider

Android has a standardized way to share data between applications on the device using the content provider. Developers can use the content provider to expose data to other applications. For example, they can make the app data searchable from external search applications. Android itself exposes data such as calendar data, contact data, and the like using the same system.

Android Runtime and Core/Native Libraries



Libraries

1) Android Runtime

Android currently uses Android Runtime (ART) to execute application code. ART is preceded by the Dalvik Runtime that compiled developer code to Dalvik Executable files (Dex files). These execution environments are optimized for the android platform taking into consideration the processor and memory constraints on mobile devices.

The runtime translates code written by programmers into machine code that does computations and utilizes android framework components to deliver functionality. Android hosts multiple applications and system components that each run in their processes.

Core Libraries

In this segment, we will discuss some of the core libraries that are present in the Android operating system.

2) MediaFramework

Android also natively supports popular media codecs, making it easy for apps created on the Android platform to use/play multimedia components out of the box.

3) SQLite

Android also has an SQLite database that enables applications to have very fast native database functionality without the need for third party libraries.

4) Freetype

Android comes with a preinstalled fast and flexible font engine. This makes it possible for application developers to style components of their application and deliver a rich experience that communicates the developer's intent.

5) OpenGL

Android also comes with the OpenGL graphics system. It's a C library that helps Android use hardware components in the real-time rendering of 2D and 3D graphics.

6) SSL

Android also comes with an inbuilt security layer to enable secure communication between applications on Android and other devices such as servers, other mobile devices, routers 6.

7) SGL

Android comes with a graphics library implemented in low-level code that efficiently renders graphics for the android platform. It works with the higher-level components of the Android framework Android graphics pipeline.

8) Libc

The core of Android contains libraries written in C and C++, which are low-level languages meant for embedded use that help in maximizing performance. Libc provides a means to expose low-level system functionalities such as Threads, Sockets, IO, and the like to these libraries.

9) Webkit

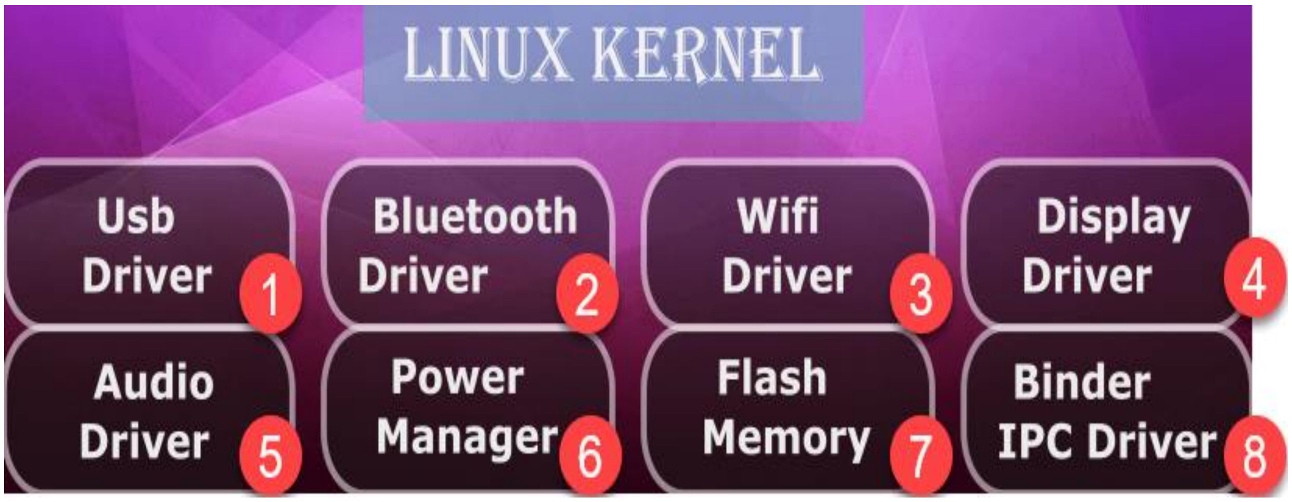
This is an open-source Browser engine used as a basis to build browsers. The default Android browser before version 4.4 KitKat uses it for rendering web pages. It enables application developers to render web components in the view-system by using WebView. This enables apps to integrate web components into their functionality.

10) Surface Manager

The surface manager is responsible for ensuring the smooth rendering of application screens. It does this by composing 2D and 3D graphics for rendering. It further enables this by doing off-screen buffering.

Linux Kernel

The root component of the Android System is the Linux Kernel. It is the foundational piece that enables all of Android's functionality.



The Linux Kernel is a battle-tested piece of software that has been used in developing operating systems for devices of wide range, from supercomputers to small gadgets. It has limited processing abilities like small networked gadgets for the Internet of Things (IoT).

The Linux Kernel can be tweaked to meet the device specifications to make it possible for manufacturers to make Android devices with different capabilities to match user experience.

With regards to Android, the Kernel is responsible for many foundational functionalities including but not limited to these:

1. Device drivers
2. Memory Management
3. Process Management

Device Drivers

The Linux Kernel houses the drivers needed to make it possible for the operating system to work with different hardware components. These drivers provide a standard interface with which hardware components sourced from different manufacturers can work with.

This makes it possible for device manufacturers to source different components, such as Bluetooth components, Wifi components, camera components. As long as the manufacturers match the Android standard specifications, integration is seamless.

1) USB Driver

Linux also provides Android with a means to interface with USB devices. Modern devices come with different USB ports, including USB 2.0 and new versions of USB, including USB-C. These drivers make it possible to use the USB port to charge, transfer live data such as logs from the Android devices, and interact with the android file system.

2) Bluetooth Driver

Linux Kernel provides support for interfacing with Bluetooth hardware components. It provides a way to read and write data received from supported bluetooth radio frequencies. It also provides a set of facilities for Android to configure Bluetooth.

3) Wifi Driver

The Linux kernel provides drivers to integrate the WiFi networking hardware components. WiFi components embedded in mobile devices enable Android devices to connect to wifi networks. The driver enables the wifi components to broadcast wifi networks and create hotspots.

4) Display Driver

Android makes it possible to interface with display components. For most devices, the interface component is an LCD touch-screen. It allows support for configuring and drawing pixels.

5) Audio Driver

Android devices commonly come with hardware components for audio input and output. Audio drivers in the kernel enable the Android system to use audio received from these components and also produce audio output.

6) Power Manager

Most Android devices are used while disconnected from power outlets. They thus depend on batteries to power them for a large chunk of their usage. Linux Kernel comes with a power management system that’s configurable to meet the needs of the devices using it.

Android OS uses the power manager to make other components on the device power-aware. It does this by broadcasting various power-related states. These states are Standby, Sleep, and Low-Battery. On Android, the power manager is tweaked to default to sleep mode to ensure maximum battery life.

The Power Manager exposes means for applications to react to different power modes. Applications can also change their behavior to match the current power state of the device.

An application can also request to change the default power policies. Applications can achieve the desired functionality, such as keeping the hardware components active. An example is keeping the screen active when reading a book to ensure a user isn’t interrupted. Another example is keeping the audio components turned on when listening to music in the background.

7) Flash Memory

Most Android devices use flash memory as a means of storage. Flash memory is fast and takes less space making it perfect for small devices. Linux kernel provides a means for Android devices to read and write into flash memory. It provides a means to partition the memory in such a way that the OS and other applications can easily and efficiently share the memory resource.

8) Binder

Android hosts many applications and system components that each run in their processes. In most cases, these processes should be isolated from each other to prevent interference and data corruption. Yet, there are instances that we wish to pass data from one process to another.

The Linux kernel enables data sharing functionality by providing binder drivers. Binder drivers enable inter-process communication, IPC. Using IPC processes can discover other processes and share information.

Memory Management

Another responsibility of the Linux Kernel is memory management. As different applications run, the Kernel ensures the memory space they use doesn’t conflict and overwrite each other.

It also helps ensure that all running apps get adequate memory to function, making sure no single app takes too much space.

Process Management

Every app in Android runs in a process. The Kernel is also responsible for managing processes. This means it’s responsible for creating, pausing, stopping, shutting, or killing down processes.

The Kernel enables various functionalities such as running multiple processes at the same time, communicating between processes, running processes in the background, and so on.

As each process requires its own memory space to function correctly, the Kernel ensures that memory spaces allotted for each process are protected from other processes. It also ensures that resources like RAM allotted to processes are freed up when the processes are shut down.

The Linux Kernel is also responsible for distributing work to the processors present in the device. This makes it possible to maximize the performance of devices with multiple cores as different apps will have processes being run on a different core.

The Linux Kernel does more task under the hood including enforcing security.

Summary:

- Android architecture is organized in layers.
- Each layer solves a unique set of problems.
- End-users interact with apps on the Application layer
- Application developers develop apps to be used on the Application layer. They do so using tools and abstractions provided by the Application Framework.
- Android Framework layer simplifies access to low-level components by creating an API over native libraries.
- Android Runtime and Core-Libraries use low-level languages together with optimizations for mobile devices. This ensures code written by application developers runs smoothly despite Android device constraints.
- At the bottom of the Android software stack is the Linux kernel. It interfaces with the hardware components common in Android devices.

Module – I

Installation of Android studio

Android Studio is the official **IDE (Integrated Development Environment)** for Android app development and it is based on **JetBrains' IntelliJ IDEA** software. Android Studio provides many excellent features that enhance productivity when building Android apps, such as:

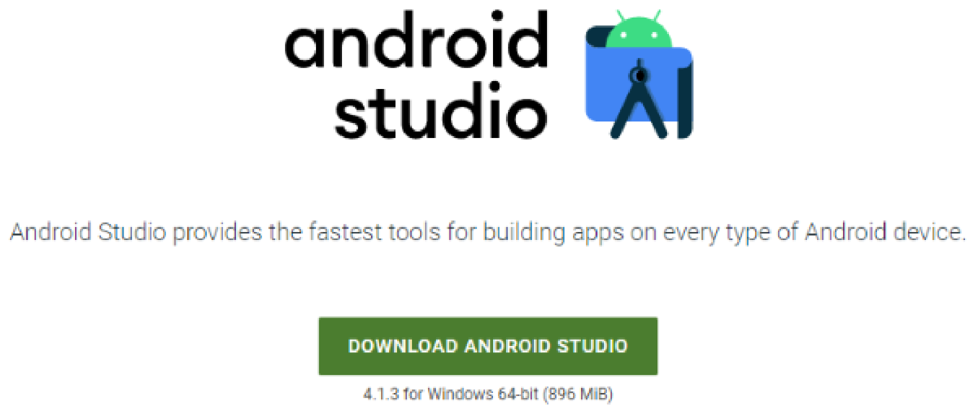
- A blended environment where one can develop for all Android devices
- Apply Changes to push code and resource changes to the running app without restarting the app
- A flexible Gradle-based build system
- A fast and feature-rich emulator
- GitHub and Code template integration to assist you to develop common app features and import sample code
- Extensive testing tools and frameworks
- C++ and NDK support
- Built-in support for Google Cloud Platform, making it easy to integrate Google Cloud Messaging and App Engine, and many more.
- Provides GUI tools that simplify the less interesting parts of app development.
- Easy integration with real time database 'firebase'.

System Requirements

- Microsoft Windows 7/8/10 (32-bit or 64-bit)
- 4 GB RAM minimum, 8 GB RAM recommended (plus 1 GB for the Android Emulator)
- 2 GB of available disk space minimum, 4 GB recommended (500 MB for IDE plus 1.5 GB for Android SDK and emulator system image)
- 1280 x 800 minimum screen resolution

Installation Guide

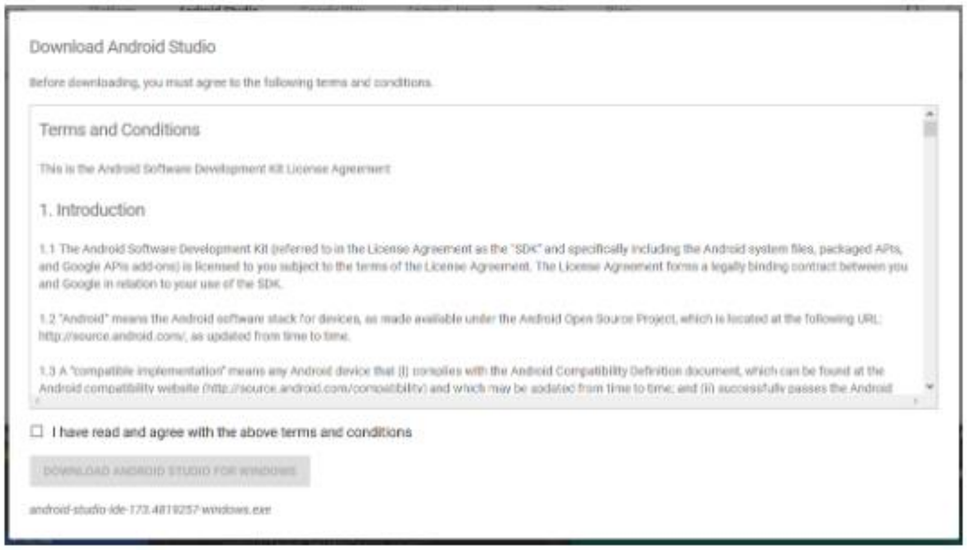
Step 1: Head over to **this link** to get the Android Studio executable or zip file.
Step 2: Click on the **Download Android Studio** Button.



Android Studio provides the fastest tools for building apps on every type of Android device.

4.1.3 for Windows 64-bit (896 MiB)

Click on the “I have read and agree with the above terms and conditions” checkbox followed by the download button.



Download Android Studio

Before downloading, you must agree to the following terms and conditions.

Terms and Conditions

This is the Android Software Development Kit License Agreement

1. Introduction

1.1 The Android Software Development Kit (referred to in the License Agreement as the "SDK" and specifically including the Android system files, packaged APIs, and Google APIs add-ons) is licensed to you subject to the terms of the License Agreement. The License Agreement forms a legally binding contract between you and Google in relation to your use of the SDK.

1.2 "Android" means the Android software stack for devices, as made available under the Android Open Source Project, which is located at the following URL: http://source.android.com/, as updated from time to time.

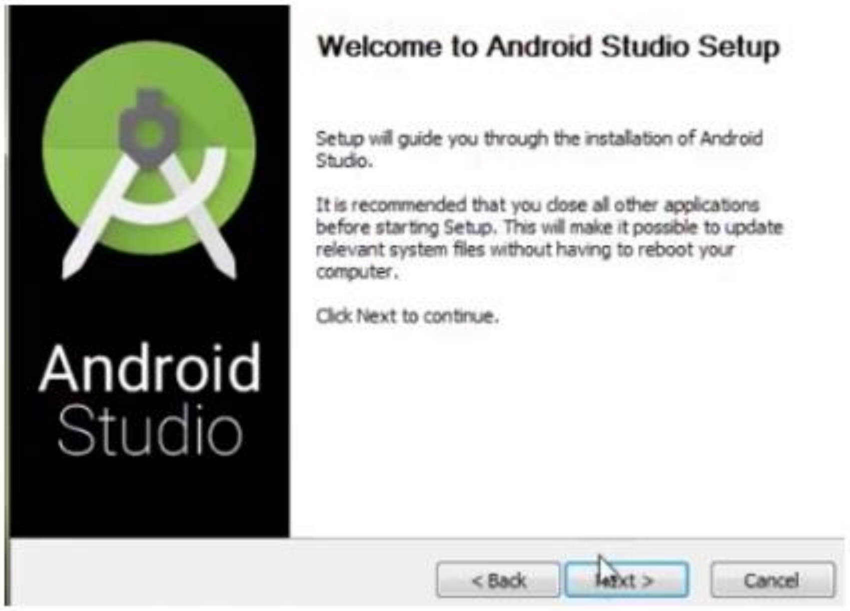
1.3 A "compatible implementation" means any Android device that (i) complies with the Android Compatibility Definition document, which can be found at the Android compatibility website (http://source.android.com/compatibility) and which may be updated from time to time; and (ii) successfully passes the Android

☐ I have read and agree with the above terms and conditions

android-studio-ide-173.4819257-windows.exe

Click on the Save file button in the appeared prompt box and the file will start downloading.

Step 3: After the downloading has finished, open the file from downloads and run it. It will prompt the following dialog box.



Welcome to Android Studio Setup

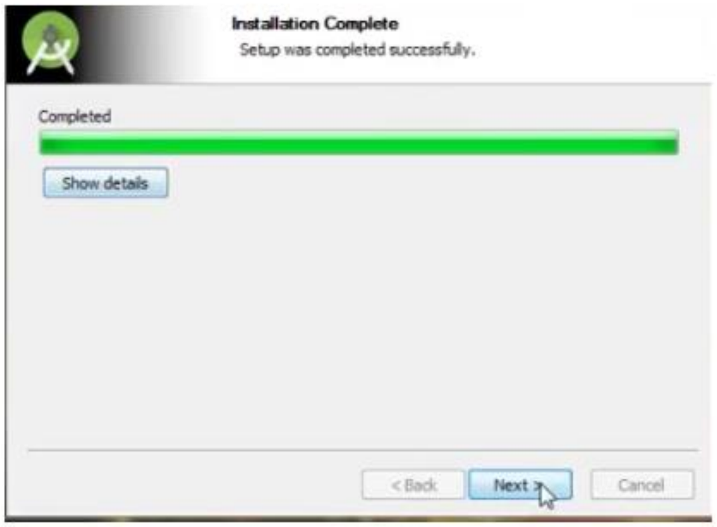
Setup will guide you through the installation of Android Studio.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

Click on next. In the next prompt, it'll ask for a path for installation. Choose a path and hit next.

Step 4: It will start the installation, and once it is completed, it will be like the image shown below.

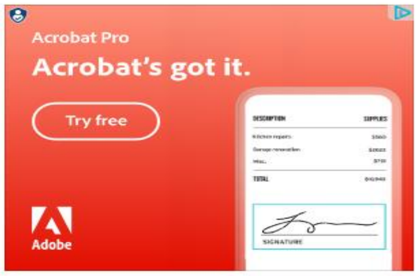


Installation Complete

Setup was completed successfully.

Completed

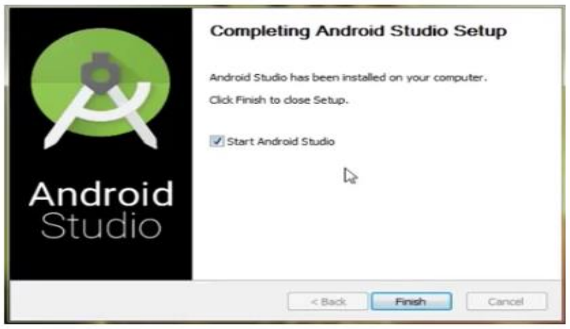
Click on next.


Acrobat Pro

Acrobat's got it.

Try free

Adobe

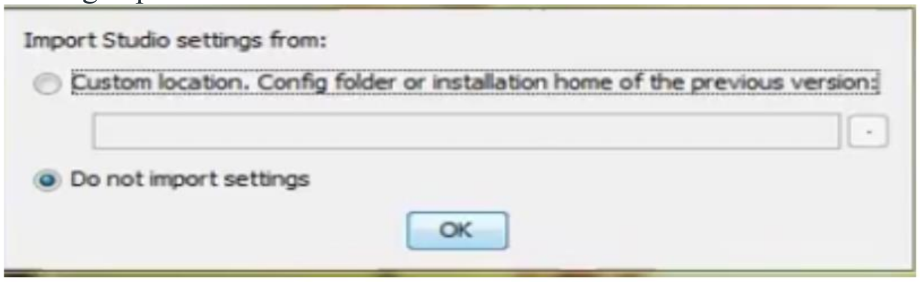

Completing Android Studio Setup

Android Studio has been installed on your computer.

Click Finish to close Setup.

[✓] Start Android Studio

Step 5: Once "**Finish**" is clicked, it will ask whether the previous settings need to be imported [if the android studio had been installed earlier], or not. It is better to choose the 'Don't import Settings option'.

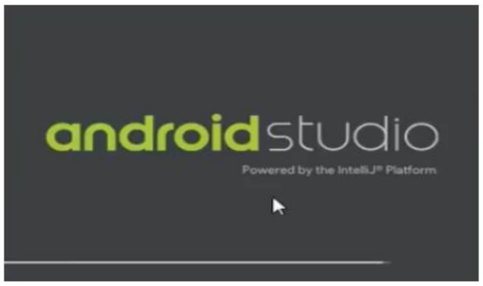

Import Studio settings from:

○ Custom location. Config folder or installation home of the previous version:

[✓ Do not import settings]

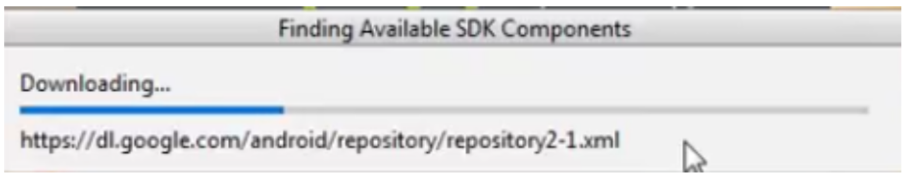
Click the **OK** button.

Step 6: This will start the Android Studio.



Meanwhile, it will be finding the available SDK components.

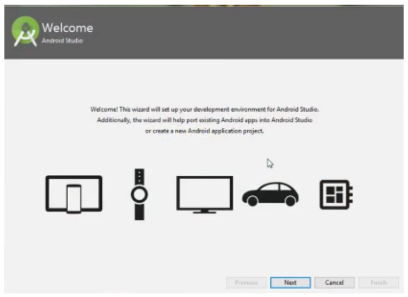
Meanwhile, it will be finding the available SDK components.


Finding Available SDK Components

Downloading...

https://dl.google.com/android/repository/repository2-1.xml

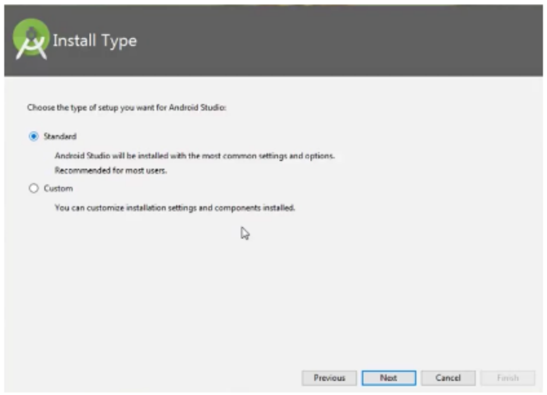
Step 7: After it has found the SDK components, it will redirect to the Welcome dialog box.


Welcome

Android Studio

Welcome! This wizard will set up your development environment for Android Studio.
Additionally, the wizard will help port existing Android apps into Android Studio
or create a new Android application project.

Click on **Next**.


Install Type

Choose the type of setup you want for Android Studio:

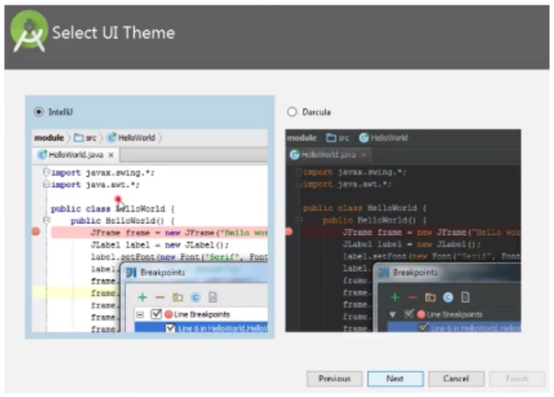
[✓ Standard]

Android Studio will be installed with the most common settings and options.
Recommended for most users.

Custom

You can customize installation settings and components installed.

Choose Standard and click on Next. Now choose the theme, whether the **Light** theme or the **Dark** one. The light one is called the **IntelliJ** theme whereas the dark theme is called **Dracula**. Choose as required.

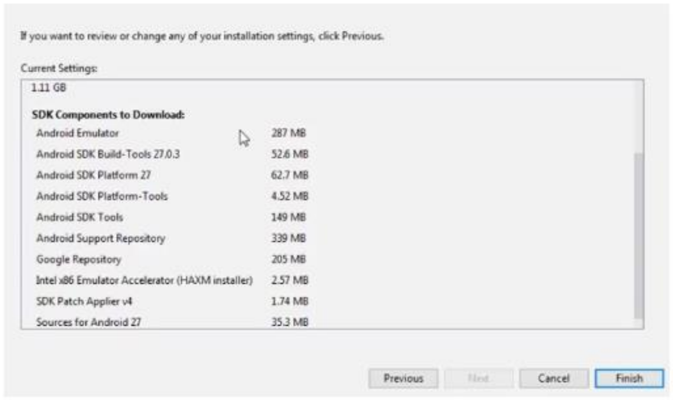

Select UI Theme

[✓ IntelliJ] Darcula

```
import javax.swing.*;
import java.awt.*;

public class HelloWorld {
    public HelloWorld() {
        JFrame frame = new JFrame("Hello wor
        JLabel label = new JLabel();
        label.setFont(new Font("Serif", Font
```

Click on the **Next** button.

Step 8: Now it is time to download the SDK components.



If you want to review or change any of your installation settings, click Previous.

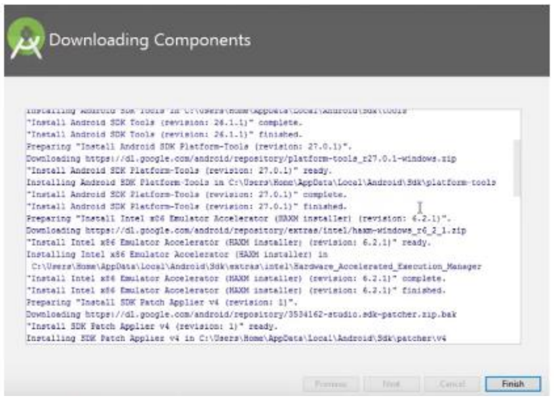
Current Settings:

1.11 GB

SDK Components to Download:

Android Emulator	287 MB
Android SDK Build-Tools 27.0.3	52.6 MB
Android SDK Platform 27	62.7 MB
Android SDK Platform-Tools	4.52 MB
Android SDK Tools	149 MB
Android Support Repository	339 MB
Google Repository	205 MB
Intel x86 Emulator Accelerator (HAXM installer)	2.57 MB
SDK Patch Applier v4	1.74 MB
Sources for Android 27	35.3 MB

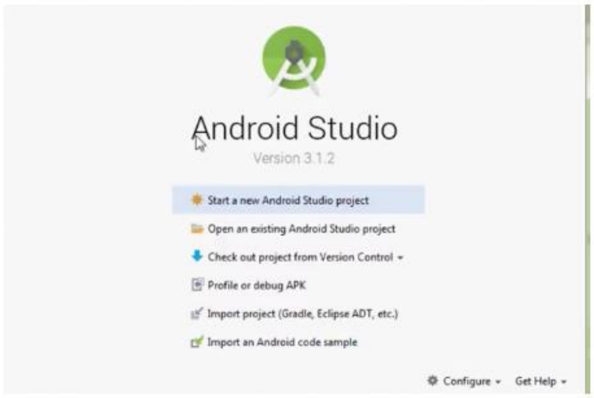
Click on Finish. Components begin to download let it complete.



```
Installing Android SDK Tools in C:\Users\Home\AppData\Local\Android\Sdk\tools
"Install Android SDK Tools (revision: 26.1.1)" complete.
"Install Android SDK Tools (revision: 26.1.1)" finished.
Preparing "Install Android SDK Platform-Tools (revision: 27.0.1)".
Downloading https://dl.google.com/android/repository/platform-tools_r27.0.1-windows.zip
"Install Android SDK Platform-Tools (revision: 27.0.1)" ready.
Installing Android SDK Platform-Tools in C:\Users\Home\AppData\Local\Android\Sdk\platform-tools
"Install Android SDK Platform-Tools (revision: 27.0.1)" complete.
"Install Android SDK Platform-Tools (revision: 27.0.1)" finished.
Preparing "Install Intel x86 Emulator Accelerator (HAXM installer) (revision: 6.2.1)".
Downloading https://dl.google.com/android/repository/extras/intel/haxm-windows_r6_2_1.zip
"Install Intel x86 Emulator Accelerator (HAXM installer) (revision: 6.2.1)" ready.
Installing Intel x86 Emulator Accelerator (HAXM installer) in
 C:\Users\Home\AppData\Local\Android\Sdk\extras\intel\Hardware_Accelerated_Execution_Manager
"Install Intel x86 Emulator Accelerator (HAXM installer) (revision: 6.2.1)" complete.
"Install Intel x86 Emulator Accelerator (HAXM installer) (revision: 6.2.1)" finished.
Preparing "Install SDK Patch Applier v4 (revision: 1)".
Downloading https://dl.google.com/android/repository/3534162-studio.sdk-patcher.zip.bak
"Install SDK Patch Applier v4 (revision: 1)" ready.
Installing SDK Patch Applier v4 in C:\Users\Home\AppData\Local\Android\Sdk\patcher\v4
```

The Android Studio has been successfully configured. Now it's time to launch and build apps. Click on the Finish button to launch it.

Step 9: Click on **Start a new Android Studio project** to build a new app.



Android Studio
Version 3.1.2
Start a new Android Studio project
Open an existing Android Studio project
Check out project from Version Control
Profile or debug APK
Import project (Gradle, Eclipse ADT, etc.)
Import an Android code sample
Configure
Get Help

To run your first android app in Android Studio you may refer to Running your first Android app.

Last Updated : 16 Nov, 2022

37

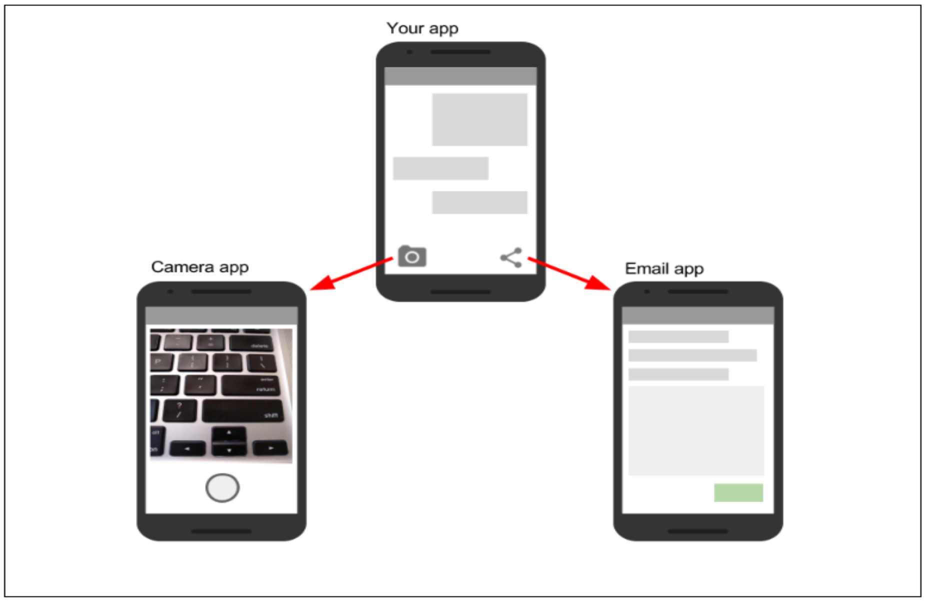
Module – 3

Building your First Application: Understanding Activities and Intents, Activity Lifecycle and Managing State, Activities and Implicit Intents

About activities

An activity represents a single screen in your app with an interface the user can interact with. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading individual messages. Your app is a collection of activities that you either create yourself, or that you reuse from other apps.

Although the activities in your app work together to form a cohesive user experience in your app, each one is independent of the others. This enables your app to start activities in other apps, and other apps can start your activities (if your app allows it). For example, a messaging app you write could start an activity in a camera app to take a picture, and then start the activity in an email app to let the user share that picture in email.



Typically, one activity in an app is specified as the "main" activity, which is presented to the user when launching the application for the first time. Each activity can then start other activities in order to perform different actions.

Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When the user is done with the current activity and presses the Back button, it is popped from the stack (and destroyed) and the previous activity resumes.

When an activity is stopped because a new activity starts, the first activity is notified of that change with the activity's lifecycle callback methods. The Activity lifecycle is the set of states an activity can be in, from when it is first created, to each time it is stopped or resumed, to when the system destroys it. You'll learn more about the activity lifecycle in the next chapter.

Creating activities

To implement an activity in your app, do the following:

- Create an activity Java class.
- Implement a user interface for that activity.
- Declare that new activity in the app manifest.

When you create a new project for your app, or add a new activity to your app, in Android Studio (with File > New > Activity), template code for each of these tasks is provided for you.

Create the activity class

Activities are subclasses of the Activity class, or one of its subclasses. When you create a new project in Android Studio, your activities are, by default, subclasses of the AppCompatActivity class. The AppCompatActivity class is a subclass of Activity that lets you to use up-to-date Android app features such as the action bar and material design, while still enabling your app to be compatible with devices running older versions of Android.

Here is a skeleton subclass of AppCompatActivity:

```
public class MainActivity extends AppCompatActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
  }
}
```

The first task for you in your activity subclass is to implement the standard activity lifecycle callback methods (such as OnCreate()) to handle the state changes for your activity. These state changes include things such as when the activity is created, stopped, resumed, or destroyed. You'll learn more about the activity lifecycle and lifecycle callbacks in the next chapter.

The one required callback your app must implement is the onCreate() method. The system calls this method when it creates your activity, and all the essential components of your activity should be initialized here. Most importantly, the OnCreate() method calls setContentView() to create the primary layout for the activity.

You typically define the user interface for your activity in one or more XML layout files. When the setContentView() method is called with the path to a layout file, the system creates all the initial views from the specified layout and adds them to your activity. This is often referred to as *inflating* the layout.

You may often also want to implement the onPause() method in your activity class. The system calls this method as the first indication that the user is leaving your activity (though it does not always mean the activity is being destroyed). This is usually where you should commit any changes that should be persisted beyond the current user session (because the user might not come back). You'll learn more about onPause() and all the other lifecycle callbacks in the next chapter.

In addition to lifecycle callbacks, you may also implement methods in your activity to handle other behavior such as user input or button clicks.

Implement a user interface

the user interface for an activity is provided by a hierarchy of views, which controls a particular space within the activity's window and can respond to user interaction.

The most common way to define a user interface using views is with an XML layout file stored as part of your app's resources. Defining your layout in XML enables you to maintain the design of your user interface separately from the source code that defines the activity's behavior.

You can also create new views directly in your activity code by inserting new view objects into a ViewGroup, and then passing the root ViewGroup to setContentView(). After your layout has been inflated -- regardless of its source -- you can add more views in Java anywhere in the view hierarchy.

Declare the activity in the manifest

Each activity in your app must be declared in the Android app manifest with the <activity> element, inside <application>. When you create a new project or add a new activity to your project in Android Studio, your manifest is created or updated to include skeleton activity declarations for each activity. Here's the declaration for the main activity.

```
<activity android:name=".MainActivity" >
   <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
   </intent-filter>
</activity>
```

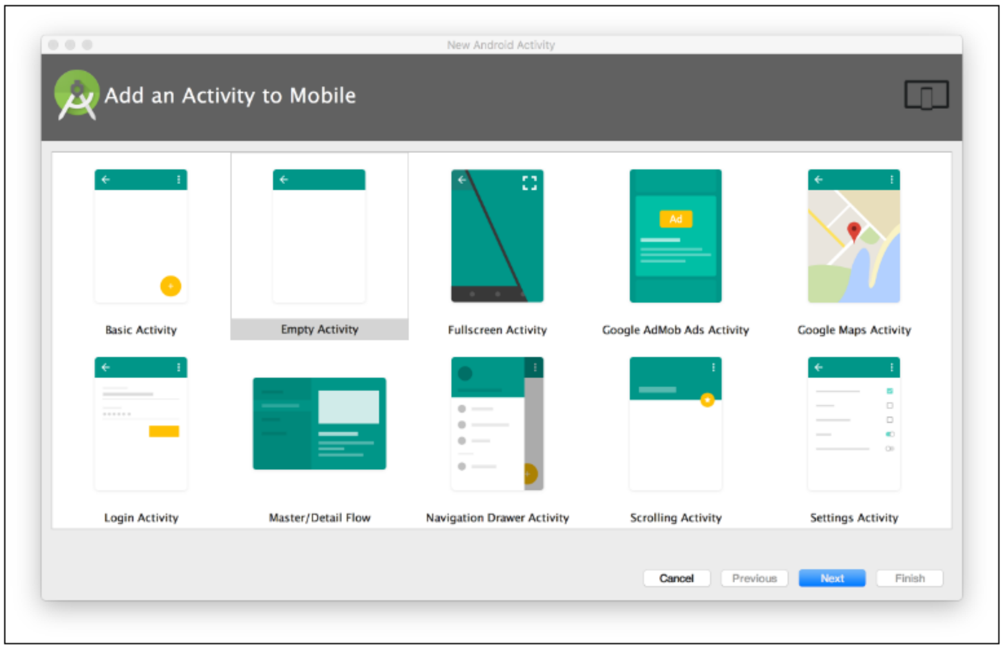
The <activity> element includes a number of attributes to define properties of the activity such as its label, icon, or theme. The only required attribute is android:name, which specifies the class name for the activity (such as "MainActivity"). See the <activity> element reference for more information on activity declarations.
The <activity> element can also include declarations for intent filters. The intent filters specify the kind of intents your activity will accept.

```
 <intent-filter>
   <action android:name="android.intent.action.MAIN" />
   <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

Intent filters must include at least one <action> element, and can also include a <category> and optional <data>. The main activity for your app needs an intent filter that defines the "main" action and the "launcher" category so that the system can launch your app. Android Studio creates this intent filter for the main activity in your project:
The <action> element specifies that this is the "main" entry point to the application. The <category> element specifies that this activity should be listed in the system's application launcher (to allow users to launch this activity).
Other activities in your app can also declare intent filters, but only your main activity should include the "main" action.. You'll learn more about implicit intents and intent filters in a later section.

Add more activities to your project

The main activity for your app and its associated layout file comes with your project when you create it. You can add new activities to your project in Android Studio with the **File > New > Activity** menu. Choose the activity template you want to use, or open the Gallery to see all the available templates.



When you choose an activity template, you'll see the same set of screens for creating the new activity that you did when you initially created the project. Android Studio provides these three things for each new activity in your app:

- A Java file for the new activity with a skeleton class definition and onCreate() method. The new activity, like the main activity, is a subclass of AppCompatActivity.
- An XML file containing the layout for the new activity. Note that the setContentView() method in the activity class inflates this new layout.
- An additional <activity> element in the Android manifest that specifies the new activity. The second activity definition does not include any intent filters. If you intend to use this activity

only within your app (and not enable that activity to be started by any other app), you do not need to add filters.

About intents

All Android activities are started or activated with an *intent*. Intents are message objects that make a request to the Android runtime to start an activity or other app component in your app or in some other app. You don't start those activities yourself;

When your app is first started from the device home screen, the Android runtime sends an intent to your app to start your app's main activity (the one defined with the MAIN action and the LAUNCHER category in the Android Manifest). To start other activities in your app, or request that actions be performed by some other activity available on the device, you build your own intents with the Intent class and call the startActivity() method to send that intent.

In addition to starting activities, intents are also used to pass data between activities. When you create an intent to start a new activity, you can include information about the data you want that new activity to operate on. So, for example, an email activity that displays a list of messages can send an intent to the activity that displays that message. The display activity needs data about the message to display, and you can include that data in the intent.

In this chapter you'll learn about using intents with activities, but intents are also used to start services and broadcast receivers. You'll learn about both those app components later on in the book.

Intent types

There are two types of intents in Android:

- *Explicit intents* specify the receiving activity (or other component) by that activity's fully-qualified class name. Use an explicit intent to start a component in your own app (for example, to move between screens in the user interface), because you already know the package and class name of that component.
- *Implicit intents* do not specify a specific activity or other component to receive the intent. Instead you declare a general action to perform in the intent. The Android system matches your request to an activity or other component that can handle your requested action. You'll learn more about implicit intents in a later chapter.

Intent objects and fields

An Intent object is an instance of the Intent class. For explicit intents, the key fields of an intent include the following:

- The activity *class* (for explicit intents). This is the class name of the activity or other component that should receive the intent, for example, com.example.SampleActivity.class. Use the intent constructor or the intent's setComponent(), setComponentName() or setClassName() methods to specify the class.

- The intent *data*. The intent data field contains a reference to the data you want the receiving activity to operate on, as a Uri object.
- Intent *extras*. These are key-value pairs that carry information the receiving activity requires to accomplish the requested action.
- Intent *flags*. These are additional bits of metadata, defined by the Intent class. The flags may instruct the Android system how to launch an activity or how to treat it after it's launched.

For implicit intents, you may need to also define the intent action and category. You'll learn more about intent actions and categories in section 2.3.

Starting an activity with an explicit intent

To start a specific activity from another activity, use an explicit intent and the startActivity() method. Explicit intents include the fully-qualified class name for the activity or other component in the Intent object. All the other intent fields are optional, and null by default.

For example, if you wanted to start the ShowMessageActivity to show a specific message in an email app, use code like this.

```
Intent messageIntent = new Intent(this, ShowMessageActivity.class);
startActivity(messageIntent);
```

The Intent constructor takes two arguments for an explicit intent.

- An application context. In this example, the activity class provides the content (here, this).
- The specific component to start (ShowMessageActivity.class).

Use the startActivity() method with the new intent object as the only argument. The startActivity() method sends the intent to the Android system, which launches the ShowMessageActivity class on behalf of your app. The new activity appears on the screen, and the originating activity is paused.

The started activity remains on the screen until the user taps the back button on the device, at which time that activity closes and is reclaimed by the system, and the originating activity is resumed. You can also manually close the started activity in response to a user action (such as a button click) with the finish() method:

```
public void closeActivity (View view) {
   finish();
}
```

Passing data between activities with intents

In addition to simply starting one activity from another, you also use intents to pass information between activities. The intent object you use to start an activity can include intent *data* (the URI of an object to act on), or intent *extras*, which are bits of additional data the activity might need.

In the first (sending) activity, you:

1. Create the Intent object.
2. Put data or extras into that intent.
3. Start the new activity with startActivity().

In the second (receiving) activity, you:

1. Get the intent object the activity was started with.
2. Retrieve the data or extras from the Intent object.

When to use intent data or intent extras

You can use either intent data and intent extras to pass data between the activities. There are several key differences between data and extras that determine which you should use.

The intent *data* can hold only one piece of information. A URI representing the location of the data you want to operate on. That URI could be a web page URL (http://), a telephone number (tel://), a goegraphic location (geo://) or any other custom URI you define.

Use the intent data field:

- When you only have one piece of information you need to send to the started activity.
- When that information is a data location that can be represented by a URI.

Intent *extras* are for any other arbitrary data you want to pass to the started activity. Intent extras are stored in a Bundle object as key and value pairs. Bundles are a map, optimized for Android, where the keys are strings, and the values can be any primitive or object type (objects must implement the Parcelable interface). To put data into the intent extras you can use any of the Intent class's putExtra() methods, or create your own bundle and put it into the intent with putExtras().

Use the intent extras:

- If you want to pass more than one piece of information to the started activity.
- If any of the information you want to pass is not expressible by a URI.

Intent data and extras are not exclusive; you can use data for a URI and extras for any additional information the started activity needs to process the data in that URI.

Add data to the intent

To add data to an explicit intent from the originating activity, create the intent object as you did before:

```
Intent messageIntent = new Intent(this, ShowMessageActivity.class);
```

Use the setData() method with a Uri object to add that URI to the intent. Some examples of using setData() with URIs:

```
// A web page URL
```

```
messageIntent.setData(Uri.parse("http://www.google.com"));
// a Sample file URI
messageIntent.setData(Uri.fromFile(new File("/sdcard/sample.jpg")));
// A sample content: URI for your app's data model
messageIntent.setData(Uri.parse("content://mysample.provider/data"));
// Custom URI
messageIntent.setData(Uri.parse("custom:" + dataID + buttonId));
```

Keep in mind that the data field can only contain a single URI; if you call setData() multiple times only the last value is used. Use intent extras to include additional information (including URIs.)

After you've added the data, you can start the activity with the intent as usual.

```
startActivity(messageIntent);
```

Add extras to the intent

To add intent extras to an explicit intent from the originating activity:

1. Determine the keys to use for the information you want to put into the extras, or define your own. Each piece of information needs its own unique key.
2. Use the putExtra() methods to add your key/value pairs to the intent extras. Optionally you can create a Bundle object, add your data to the bundle, and then add the bundle to the intent.

The Intent class includes several intent extra keys you can use, defined as constants that begin with the word EXTRA_. For example, you could use Intent.EXTRA_EMAIL to indicate an array of email addresses (as strings), or Intent.EXTRA_REFERRER to specify information about the originating activity that sent the intent.

You can also define your own intent extra keys. Conventionally you define intent extra keys as static variables with names that begin with EXTRA_. To guarantee that the key is unique, the string value for the key itself should be prefixed with your app's fully qualified class name. For example:

```
public final static String EXTRA_MESSAGE = "com.example.mysampleapp.MESSAGE";
public final static String EXTRA_POSITION_X = "com.example.mysampleapp.X";
public final static String EXTRA_POSITION_Y = "com.example.mysampleapp.Y";
```

Create an intent object (if one does not already exist):

```
Intent messageIntent = new Intent(this, ShowMessageActivity.class);
```

Use a putExtra() method with a key to put data into the intent extras. The Intent class defines many putExtra() methods for different kinds of data:

```
messageIntent.putExtra(EXTRA_MESSAGE, "this is my message");
messageIntent.putExtra(EXTRA_POSITION_X, 100);
messageIntent.putExtra(EXTRA_POSITION_Y, 500);
```

Alternately, you can create a new bundle and populate that bundle with your intent extras. Bundle defines many "put" methods for different kinds of primitive data as well as objects that implement Android's Parcelable interface or Java's Serializable.

```
Bundle extras = new Bundle();
extras.putString(EXTRA_MESSAGE, "this is my message");
extras.putInt(EXTRA_POSITION_X, 100);
extras.putInt(EXTRA_POSITION_Y, 500);
```

After you've populated the bundle, add it to the intent with the putExtras() method (note the "s" in Extras):

```
messageIntent.putExtras(extras);
```

Start the activity with the intent as usual:

```
startActivity(messageIntent);
```

Retrieve the data from the intent in the started activity

When you start an activity with an intent, the started activity has access to the intent and the data it contains.

To retrieve the intent the activity (or other component) was started with, use the getIntent() method:

```
Intent intent = getIntent();
```

Use getData() to get the URI from that intent:

```
Uri locationUri = getData();
```

To get the extras out of the intent, you'll need to know the keys for the key/value pairs. You can use the standard Intent extras if you used those, or you can use the keys you defined in the originating activity (if they were defined as public.)

Use one of the getExtra() methods to extract extra data out of the intent object:

```
String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
int positionX = intent.getIntExtra(MainActivity.EXTRA_POSITION_X);
int positionY = intent.getIntExtra(MainActivity.EXTRA_POSITION_Y);
```

Or you can get the entire extras bundle from the intent and extract the values with the various Bundle methods:

```
Bundle extras = intent.getExtras();
String message = extras.getString(MainActivity.EXTRA_MESSAGE);
```

Getting data back from an activity

When you start an activity with an intent, the originating activity is paused, and the new activity remains on the screen until the user clicks the back button, or you call the finish() method in a click handler or other function that ends the user's involvement with this activity.

Sometimes when you send data to an activity with an intent, you would like to also get data back from that intent. For example, you might start a photo gallery activity that lets the user pick a photo. In this case your original activity needs to receive information about the photo the user chose back from the launched activity.

To launch a new activity and get a result back, do the following steps in your originating activity:

1. Instead of launching the activity with startActivity(), call startActivityForResult() with the intent and a request code.
2. Create a new intent in the launched activity and add the return data to that intent.
3. Implement onActivityResult() in the originating activity to process the returned data.

You'll learn about each of these steps in the following sections.

Use startActivityForResult() to launch the activity

To get data back from a launched activity, start that activity with the startActivityForResult() method instead of startActivity().

```
startActivityForResult(messageIntent, TEXT_REQUEST);
```

The startActivityForResult() method, like startActivity(), takes an intent argument that contains information about the activity to be launched and any data to send to that activity. The startActivityForResult() method, however, also needs a request code.

The request code is an integer that identifies the request and can be used to differentiate between results when you process the return data. For example, if you launch one activity to take a photo and another to pick a photo from a gallery, you'll need different request codes to identify which request the returned data belongs to.

Conventionally you define request codes as static integer variables with names that include REQUEST. Use a different integer for each code. For example:

```
public static final int PHOTO_REQUEST = 1;
public static final int PHOTO_PICK_REQUEST = 2;
public static final int TEXT_REQUEST = 3;
```

Return a response from the launched activity

The response data from the launched activity back to the originating activity is sent in an intent, either in the data or the extras. You construct this return intent and put the data into it in much the same way you do for the sending intent. Typically your launched activity will have an onClick or other user input callback method in which you process the user's action and close the activity. This is also where you construct the response.

To return data from the launched activity, create a new empty intent object.

```
Intent returnIntent = new Intent();
```

Note: To avoid confusing sent data with returned data, use a new intent object rather than reusing the original sending intent object.

Return result intents do not need a class or component name to end up in the right place. The Android system directs the response back to the originating activity for you.

Add data or extras to the intent the same way you did with the original intent. You may need to define keys for the return intent extras at the start of your class.

```
public final static String EXTRA_RETURN_MESSAGE =
    "com.example.mysampleapp.RETURN_MESSAGE";
```

Then put your return data into the intent as usual. Here the return message is an intent extra with the key EXTRA_RETURN_MESSAGE.

```
messageIntent.putExtra(EXTRA_RETURN_MESSAGE, mMessage);
```

Use the setResult() method with a response code and the intent with the response data:

```
setResult(RESULT_OK,replyIntent);
```

The response codes are defined by the Activity class, and can be

- RESULT_OK. the request was successful.
- RESULT_CANCELED: the user cancelled the operation.
- RESULT_FIRST_USER. for defining your own result codes.

You'll use the result code in the originating activity.

Finally, call finish() to close the activity and resume the originating activity:

```
finish();
```

Read response data in onActivityResult()

Now that the launched activity has sent data back to the originating activity with an intent, that first activity must handle that data. To handle returned data in the originating activity, implement the onActivityResult() callback method. Here is a simple example.

```
public void onActivityResult(int requestCode, int resultCode,  Intent data) {
    super.onActivityResult(requestCode, resultCode, data);

    if (requestCode == TEXT_REQUEST) {
        if (resultCode == RESULT_OK) {
            String reply =
                data.getStringExtra(SecondActivity.EXTRA_RETURN_MESSAGE);
                // process data
        }
    }
}
```

The three arguments to the onActivityResult() contain all the information you need to handle the return data.

- **Request code.** The request code you set when you launched the activity with startActivityForResult(). If you launch different activities to accomplish different operations, use this code to identify the specific data you're getting back.
- **Result code:** the result code set in the launched activity, usually one of RESULT_OK or RESULT_CANCELED.
- **Intent data**. the intent that contains the data returned from the launch activity.

The example method shown above shows the typical logic for handling the request and response codes. The first test is for the TEXT_REQUEST request, and that the result was successful. Inside the body of those tests you extract the return information out of the intent. Use getData() to get the intent data, or getExtra() to retrieve values out of the intent extras with a specific key.

Activity navigation

Any app of any complexity that you build will include multiple activities, both designed and implemented by you, and potentially in other apps as well. As your users move around your app and between activities, consistent navigation becomes more important to the app's user experience. Few things frustrate users more than basic navigation that behaves in inconsistent and unexpected ways. Thoughtfully designing your app's navigation will make using your app predictable and reliable for your users.

Android system supports two different forms of navigation strategies for your app.

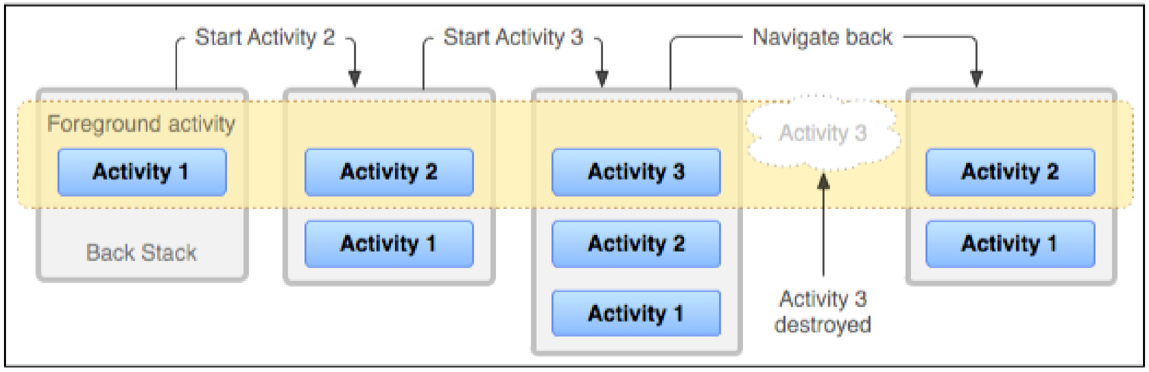
- Temporal or Back navigation, provided by the device back button, and the back stack.
- Ancestral, or Up navigation, provided by you as an option in the app's action bar.

Back navigation, tasks, and the back stack

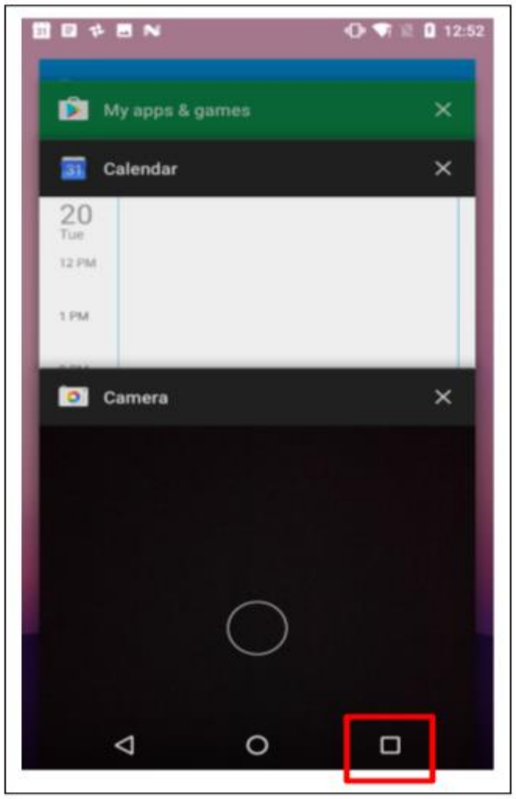
Back navigation allows your users to return to the previous activity by tapping the device back button ◁. Back navigation is also called *temporal* navigation because the back button navigates the history of recently viewed screens, in reverse chronological order.

The *back stack* is the set of activities that the user has visited and that can be returned to by the user with the back button. Each time a new activity starts, it is pushed onto the back stack and takes user focus. The previous activity is stopped but is still available in the back stack. The back stack operates on a "last in, first out" mechanism, so when the user is done with the current activity and presses the Back button, that activity is popped from the stack (and destroyed) and the previous activity resumes.

Because an app can start activities both inside and outside a single app, the back stack contains all the activities that have been launched by the user in reverse order. Each time the user presses the Back button, each activity in the stack is popped off to reveal the previous one, until the user returns to the Home screen.



Android provides a back stack for each *task*. A task is an organizing concept for all the activities the user interacts with when performing an operation, whether they are inside your app or across multiple apps. Most tasks start from the Android home screen, and tapping an app icon starts a task (and a new back stack) for that app. If the user uses an app for a while, taps home, and starts a new app, that new app launches in its own task and has its own back stack. If the user returns to the first app, that first task's back stack returns. Navigating with the back button only returns to the activities in the current task, not for all tasks running on the device. Android enables the user to navigate between tasks with the overview or recent tasks screen, accessible with the square button on lower right corner of the device

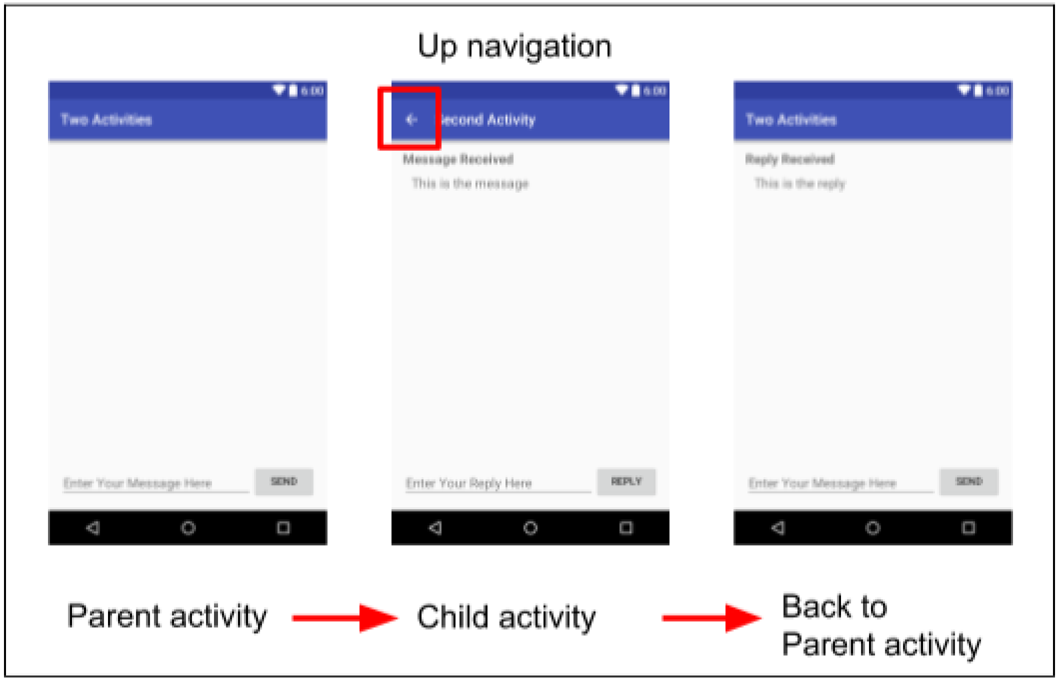


In most cases you don't have to worry about managing either tasks or the back stack for your app—the system keeps track of these things for you, and the back button is always available on the device.

There may, however, be times where you may want to override the default behavior for tasks or for the back stack. For example, if your screen contains an embedded web browser where users can navigate between web pages, you may wish to use the browser's default back behavior when users press the device's *Back* button, rather than returning to the previous activity. You may also need to change the default behavior for your app in other special cases such as with notifications or widgets, where activities deep within your app may be launched as their own tasks, with no back stack at all. You'll learn more about managing tasks and the back stack in a later section.

Up navigation

Up navigation, sometimes referred to as ancestral or logical navigation, is used to navigate within an app based on the explicit hierarchical relationships between screens. With Up navigation, your activities are arranged in a hierarchy, and "child" activities show a left-facing arrow in the action bar that returns the user to the "parent" activity. The topmost activity in the hierarchy is usually your main activity, and the user cannot go up from there.



For instance, if the main activity in an email app is a list of all messages, selecting a message launches a second activity to display that single email. In this case the message activity would provide an Up button that returns to the list of messages.

The behavior of the Up button is defined by you in each activity based on how you design your app's navigation. In many cases, Up and Back navigation may provide the same behavior: to just return to the previous activity. For example, a Settings activity may be available from any activity in your app, so "up" is the same as back -- just return the user to their previous place in the hierarchy.

Providing Up behavior for your app is optional, but a good design practice, to provide consistent navigation for the activities in your app.

Implement up navigation with parent activities

With the standard template projects in Android Studio, it's straightforward to implement Up navigation. If one activity is a child of another activity in your app's activity hierarchy, specify that activity's parent in the Android Manifest.

Beginning in Android 4.1 (API level 16), declare the logical parent of each activity by specifying the android:parentActivityName attribute in the `<activity>` element. To support older versions of Android, include `<meta-data>` information to define the parent activity explicitly. Use both methods to be backwards-compatible with all versions of Android.
Here are the skeleton definitions for both a main (parent) activity and a second (child) activity:

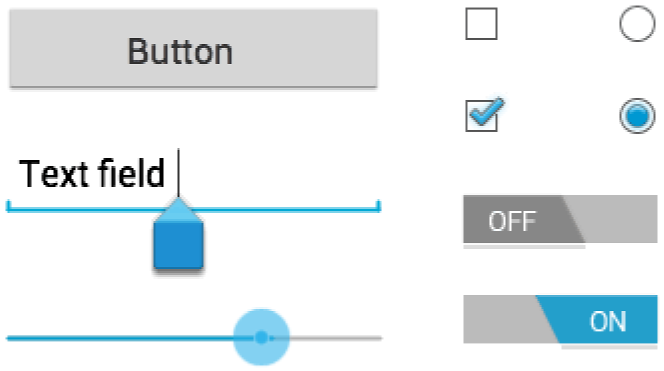
```
<application ... >
    <!-- The main/home activity (it has no parent activity) -->
    <activity
        android:name=".MainActivity" ...>
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>

    </activity>
    <!-- A child of the main activity -->
    <activity android:name=".SecondActivity"
        android:label="@string/activity2_name"
        android:parentActivityName=".MainActivity">
        <meta-data
            android:name="android.support.PARENT_ACTIVITY"
            android:value="com.example.android.twoactivities.MainActivity" />
    </activity>
</application>
```

Module – 4

Android UI components: Text Controls, Buttons, Widgets, Layouts, Containers

In android **UI** or **input** controls are the interactive or View components that are used to design the user interface of an application. In android we have a wide variety of UI or input controls available, those are TextView, EditText, Buttons, Checkbox, Progressbar, Spinners, etc.

Following is the pictorial representation of user interface (UI) or input controls in android application.



Generally, in android the user interface of an app is made with a collection of **View** and **ViewGroup** objects.

The **View** is a base class for all UI components in android and it is used to create interactive UI components such as TextView, EditText, Checkbox, Radio Button, etc. and it is responsible for event handling and drawing.

The **ViewGroup** is a subclass of **View** and it will act as a base class for layouts and layout parameters. The ViewGroup will provide invisible containers to hold other Views or ViewGroups and to define the layout properties.

To know more about View and ViewGroup in android applications, check this Android View and ViewGroup.

In android, we can define a UI or input controls in two ways, those are

- Declare UI elements in XML
- Create UI elements at runtime

The android framework will allow us to use either or both of these methods to define our application’s UI.

Declare UI Elements in XML

In android, we can create layouts same as web pages in HTML by using default **Views** and **ViewGroups** in the XML file. The layout file must contain only one root element, which must be a **View** or **ViewGroup** object. Once we define the root element, then we can add additional layout objects or widgets as a child elements to build View hierarchy that defines our layout.

Following is the example of defining UI controls (TextView, EditText, Button) in the XML file (**activity_main.xml**) using LinearLayout.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
   <TextView
      android:id="@+id/fstTxt"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Enter Name" />
   <EditText
      android:id="@+id/name"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:ems="10"/>
   <Button
      android:id="@+id/getName"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Get Name" />
</LinearLayout>
```

In android, each input control is having a specific set of events and these events will be raised when the user performs particular action like, entering the text or touches the button.

Note: we need to create a user interface (**UI**) layout files in **/res/layout** project directory, then only the layout files will compile properly.

Load XML Layout File from an Activity

Once we are done with the creation of layout with UI controls, we need to load the XML layout resource from our activity **onCreate()** callback method like as shown below.

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

If you observe above code we are calling our layout using **setContentView** method in the form of **R.layout.layout_file_name**. Here our xml file name is **activity_main.xml** so we used file name **activity_main**.

Generally, during the launch of our activity, **onCreate()** callback method will be called by android framework to get the required layout for an activity.

Create UI Element at Runtime

If we want to create UI elements at runtime, we need to create our own custom **View** and **ViewGroup** objects programmatically with required layouts.

Following is the example of creating UI elements (TextView, EditText, Button) in LinearLayout using custom **View** and **ViewGroup** objects in an activity programmatically.

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView textView1 = new TextView(this);
        textView1.setText("Name:");
        EditText editText1 = new EditText(this);
        editText1.setText("Enter Name");
        Button button1 = new Button(this);
        button1.setText("Add Name");
        LinearLayout linearLayout = new LinearLayout(this);
        linearLayout.addView(textView1);
        linearLayout.addView(editText1);
        linearLayout.addView(button1);
        setContentView(linearLayout);
    }
}
```

By creating a custom **View** and **ViewGroups** programmatically, we can define UI controls in layouts based on our requirements in android applications.

Width and Height

When we define a UI controls in a layout using an XML file, we need to set width and height for every **View** and **ViewGroup** elements using **layout_width** and **layout_height** attributes.

Following is the example of setting width and height for **View** and **ViewGroup** elements in the XML layout file.

```
<?xml                                                    version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
                                                      android:layout_width="match_parent"
    android:layout_height="match_parent">
                                                                              <TextView
        android:id="@+id/fstTxt"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Enter                                                      Name" />
</LinearLayout>
```

If you observe above example, we used different values to set layout_width and layout_height, those are

- match_parent
- wrap_content

If we set value **match_parent**, then the **View** or **ViewGroup** will try to match with parent width or height.

If we set value **wrap_content**, then the **View** or **ViewGroup** will try to adjust its width or height based on the content.

Android Different Types of UI Controls

We have a different type of UI controls available in android to implement the user interface for our android applications.

Following are the commonly used UI or input controls in android applications.

Android TextView
In android, **TextView** is a user interface control that is used to display the text to the user.

To know more about TextView control check this, Android TextView with Examples.

Android EditText
In android, **EditText** is a user interface control which is used to allow the user to enter or modify the text.

To know more about EditText, check this Android EditText with Examples.

Android AutoCompleteTextView
In android, **AutoCompleteTextView** is an editable text view which is used to show the list of suggestions based on the user typing text. The list of suggestions will be shown as a dropdown menu from which the user can choose an item to replace the content of the textbox.

To know more about AutoCompleteTextView, check this Android AutoCompleteTextView with Examples.

Android Button
In android, **Button** is a user interface control that is used to perform an action when the user clicks or tap on it.

To know more about Button in android check this, Android Buttons with Examples.

Android Image Button
In android, **Image Button** is a user interface control that is used to display a button with an image to perform an action when the user clicks or tap on it.
Generally, the Image button in android looks similar as regular Button and perform the actions same as regular button but only difference is for image button we will add an image instead of text.

To know more about Image Button in android check this, Android Image Button with Examples.

Android Toggle Button
In android, **Toggle Button** is a user interface control that is used to display ON (Checked) or OFF (Unchecked) states as a button with a light indicator.

To know more about Toggle Button in android check this, Android Toggle Button with Examples.

Android CheckBox

In android, **Checkbox** is a two-states button that can be either checked or unchecked.

To know more about CheckBox in android check this, Android CheckBox with Examples.

Android Radio Button

In android, **Radio Button** is a two-states button that can be either checked or unchecked and it cannot be unchecked once it is checked.

To know more about Radio Button in android check this, Android Radio Button with Examples.

Android Radio Group

In android, **Radio Group** is used to group one or more radio buttons into separate groups based on our requirements.

In case if we group radio buttons using the radio group, at a time only one item can be selected from the group of radio buttons.

To know more about Radio Group in android check this, Android Radio Group with Examples.

Android ProgressBar

In android, **ProgressBar** is a user interface control which is used to indicate the progress of an operation.

To know more about ProgressBar, check this Android ProgressBar with Examples.

Android Spinner

In android, **Spinner** is a drop-down list which allows a user to select one value from the list.

To know more about Spinner, check this Android Spinner with Examples.

Android TimePicker

In android, **TimePicker** is a widget for selecting the time of day, either in 24-hour or AM/PM mode.

To know more about TimePicker, check this, Android TimePicker with Examples.

Android DatePicker

In android, DatePicker is a widget for selecting a date.
To know more about DatePicker, check this Android DatePicker with Examples.

We learn all android user interface (UI) controls in next chapters in detailed manner with examples.
n android, **TextView** is a user interface control that is used to set and display the text to the user based on our requirements. The TextView control will act as like label control and it won't allow users to edit the text.

In android, we can create a TextView control in two ways either in XML layout file or create it in Activity file programmatically.

Create a TextView in Layout File

Following is the sample way to define **TextView** control in XML layout file in android application.

```
<?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">
    <TextView
        android:id="@+id/textView1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="10dp"
        android:text="Welcome to Tutlane"
        android:textColor="#86AD33"
        android:textSize="20dp"
        android:textStyle="bold" />
</LinearLayout>
```

If you observe above code snippet, here we defined a TextView control in xml layout file to display the text in android application.

Create a TextView in Activity File

In android, we can create a **TextView** control programmatically in an activity file based on our requirements.

Following is the example of creating a TextView control dynamically in an activity file.

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        LinearLayout linearLayout = (LinearLayout) findViewById(R.id.linearlayout);
        TextView textView = new TextView(this);
        textView.setText("Welcome to Tutlane");
        linearLayout.addView(textView);
    }
}
```

Set the Text of Android TextView

In android, we can set the text of **TextView** control either while declaring it in **Layout** file or by using **setText()** method in Activity file.

Following is the example to set the text of TextView control while declaring it in the XML Layout file.

```
<TextView
        android:id="@+id/textView1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Welcome to Tutlane" />
```

If you observe above example we used **android:text** property to the set required text for TextView control in XML Layout file.

Following is another way to set the text of textview control programmatically in activity file using **setText()** method.

```
TextView tv = (TextView)findViewById(R.id.textView1);
tv.setText("Welcome to Tutlane");
```

If you observe above code snippet, we are getting the **TextView** control which we defined in XML layout file using **id** property and setting the text using **setText()** method.

Android TextView Attributes

The following are some of the commonly used attributes related to TextView control in android applications.

Attribute	Description
android: id	It is used to uniquely identify the control
android:autoLink	It will automatically found and convert URLs and email addresses as clickable links.
android: ems	It is used to make the textview be exactly this many ems wide.
android:hint	It is used to display the hint text when text is empty
android:width	It makes the TextView be exactly this many pixels wide.
android:height	It makes the TextView be exactly this many pixels tall.
android:text	It is used to display the text.
android:textColor	It is used to change the color of the text.
android:gravity	It is used to specify how to align the text by the view's x and y-axis.

Attribute	**Description**
android:maxWidth	It is used to make the TextView be at most this many pixels wide.
android:minWidth	It is used to make the TextView be at least this many pixels wide.
android:textSize	It is used to specify the size of the text.
android:textStyle	It is used to change the style (bold, italic, bolditalic) of text.
android:textAllCaps	It is used to present the text in all CAPS
android:typeface	It is used to specify the Typeface (normal, sans, serif, monospace) for the text.
android:textColor	It is used to change the color of the text.
android:textColorHighlight	It is used to change the color of text selection highlight.
android:textColorLink	It is used to change the text color of links.

Attribute	Description
android:inputType	It is used to specify the type of text being placed in text fields.
android:fontFamily	It is used to specify the fontFamily for the text.
android:editable	If we set, it specifies that this TextView has an input method.

Android TextView Example

Following is the example of using TextView control in the android application.

Create a new android application using android studio and give names as **TextViewExample**. In case if you are not aware of creating an app in android studio check this article Android Hello World App.

Now open an **activity_main.xml** file from **\res\layout** path and write the code like as shown below

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:layout_marginTop="10dp"
   android:orientation="vertical"
   android:padding="10dp">
                                                                                <TextView
       android:id="@+id/textView1"
       android:layout_width="match_parent"
```

```
    android:layout_height="wrap_content"
                                                    android:layout_marginBottom="10dp"
    android:text="Welcome                            to                            Tutlane"
    android:textColor="#86AD33"
    android:textSize="20dp"
    android:textStyle="bold" />
                                                                          <TextView
    android:id="@+id/textView2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="15dp"
    android:textAllCaps="true" />
                                                                          <TextView
    android:id="@+id/textView3"
    android:layout_width="wrap_content"
                                                    android:layout_height="wrap_content"
    android:text="Welcome                            to                            Tutlane"
    android:textStyle="bold"
    android:textColor="#fff"
    android:background="#7F3AB5"
    android:layout_marginBottom="15dp"/>
                                                                          <TextView
    android:id="@+id/textView4"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:autoLink="email|web"
    android:text="For  more  details  visit  http://tutlane.com  and  send  mail  to
support@tutlane.com" />
</LinearLayout>
```

Once we are done with creation of layout with required controls, we need to load the XML layout resource from our activity **onCreate()** callback method, for that open main activity file **MainActivity.java** from **\java\com.tutlane.textviewexample** path and write the code like as shown below.

MainActivity.java

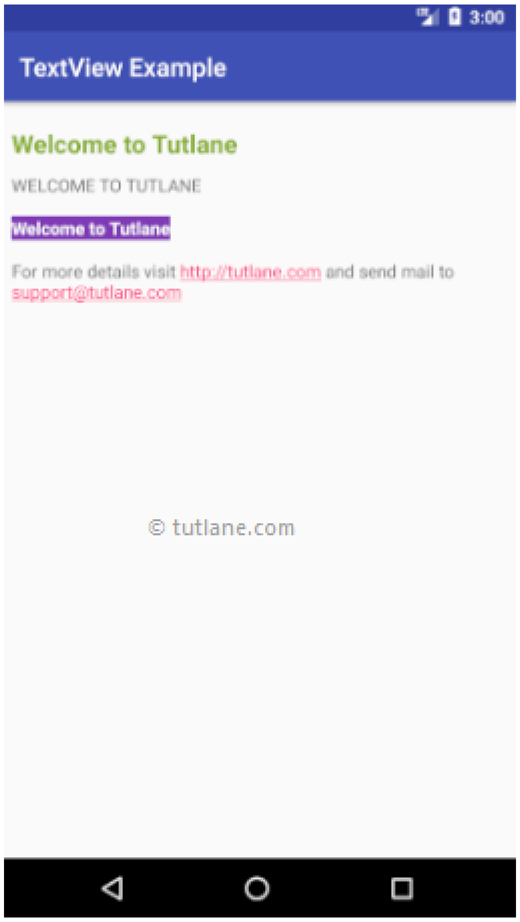
```
package com.tutlane.textviewexample;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.TextView;

public                    class MainActivity extends AppCompatActivity                {
  @Override
  protected               void onCreate(Bundle              savedInstanceState)          {
        super.onCreate(savedInstanceState);
                                                                  setContentView(R.layout.activity_main);
                    TextView           tv           =           (TextView)findViewById(R.id.textView2);
                                  tv.setText("Welcome                          to                         Tutlane");
                                                                                                                       }
}
```

If you observe above code we are calling our layout using **setContentView** method in the form of **R.layout.layout_file_name**. Here our xml file name is **activity_main.xml** so we used file name **activity_main** and we are setting text to one of our **TextView** control (**textView2**) in our activity file.

Generally, during the launch of our activity, the **onCreate()** callback method will be called by the android framework to get the required layout for an activity.

Output of Android TextView Example

When we run the above example using the android virtual device (AVD) we will get a result like as shown below.



TextView Example

Welcome to Tutlane

WELCOME TO TUTLANE

Welcome to Tutlane

For more details visit http://tutlane.com and send mail to support@tutlane.com

Executed Output

Result:

Module – 5 :

Material Design for Android: Material theme and widgets, Elevation shadows, Cards, Animations, Drawables

Material Design is a comprehensive guide for visual, motion, and interaction design across platforms and devices. To use Material Design in your Android apps, follow the guidelines defined in the Material Design specification. If your app uses Jetpack Compose, you can use the Compose Material 3 library. If your app uses views, you can use the Android Material Components library.

Android provides the following features to help you build Material Design apps:

- A Material Design app theme to style all your UI widgets
- Widgets for complex views, such as lists and cards
- APIs for custom shadows and animations

Material theme and widgets

To take advantage of the Material features, such as styling for standard UI widgets, and to streamline your app's style definition, apply a Material-based theme to your app.

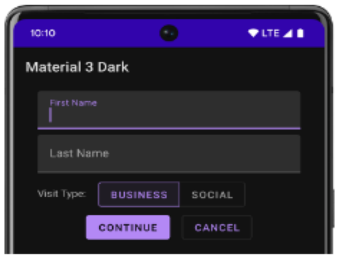


Figure 1. Dark Material theme.

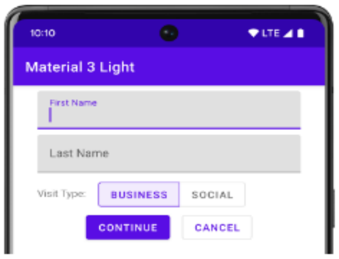


Figure 2. Light Material theme.

If you use Android Studio to create your Android project, it applies a Material theme by default. To learn how to update your project's theme, see Styles and themes.

To provide your users a familiar experience, use Material's most common UX patterns:

- Promote your UI's main action with a floating action button (FAB).
- Show your brand, navigation, search, and other actions using the app bar.
- Show and hide your app's navigation with the navigation drawer.

- Choose from the many other Material Components for your app layout and navigation, such as collapsing toolbars, tabs, a bottom nav bar, and more. To see them all, see the Material Components for Android catalog.

Whenever possible, use predefined Material Icons. For example, for the navigation "menu" button for your navigation drawer, use the standard "hamburger" icon. See Material Design Icons for a list of available icons. You can also import SVG icons from the Material Icon library with Android Studio's Vector Asset Studio.

Elevation shadows and cards

In addition to the *X* and *Y* properties, views in Android have a *Z* property. This property represents the elevation of a view, which determines the following:

- The size of its shadow: views with higher *Z* values cast bigger shadows.
- The drawing order: views with higher *Z* values appear on top of other views.

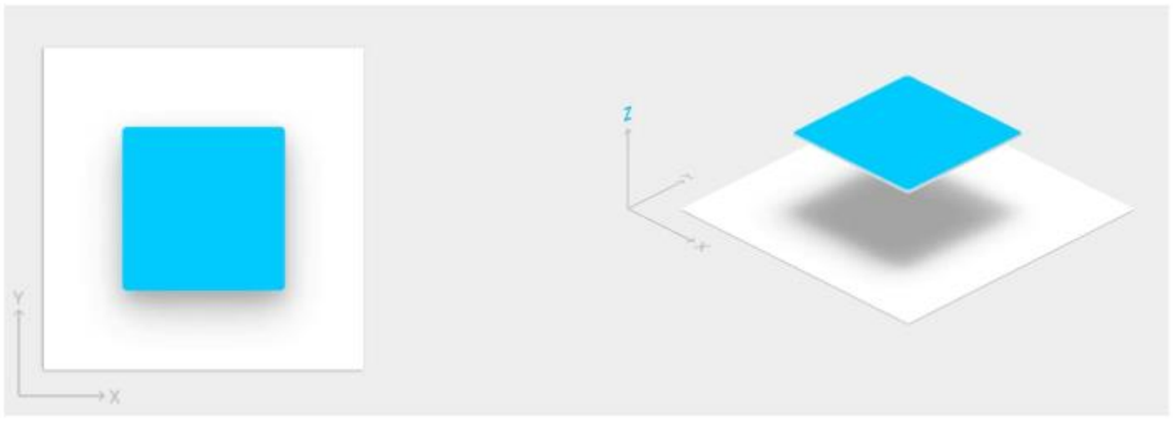


Figure 3. The *Z* value representing elevation.

You can apply elevation to a card-based layout, which helps you display important pieces of information inside cards that provide a Material look. You can use the CardView widget to create cards with a default elevation. For more information, see Create a card-based layout.

For information about adding elevation to other views, see Create shadows and clip views.

Animations

Animation APIs let you create custom animations for touch feedback in UI controls, changes in view state, and activity transitions.

These APIs let you:

- Respond to touch events in your views with **touch feedback** animations.
- Hide and show views with **circular reveal** animations.
- Switch between activities with custom **activity transition** animations.
- Create more natural animations with **curved motion**.
- Animate changes in one or more view properties with **view state change** animations.
- Show animations in **state list drawables** between view state changes.

Touch feedback animations are built into several standard views, such as buttons. The animation APIs let you customize these animations and add them to your custom views.

For more information, see Introduction to animations.

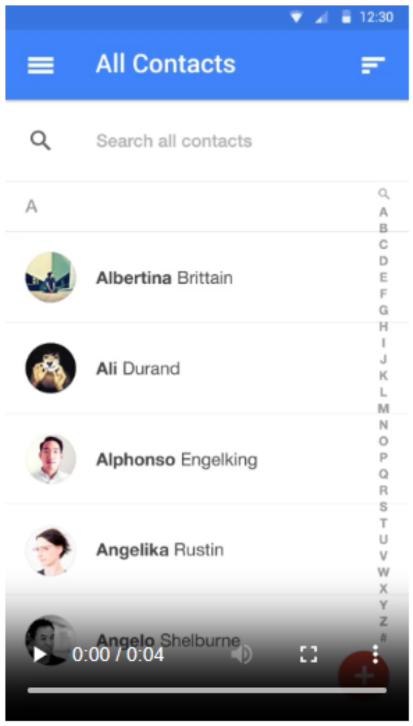
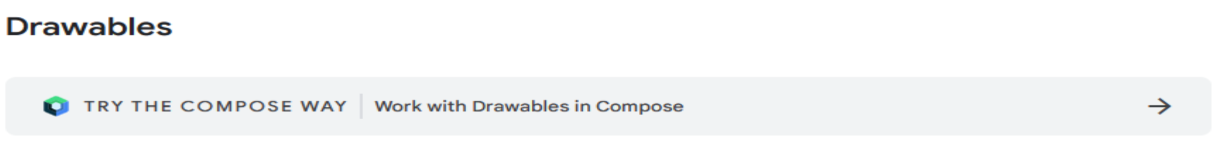


Figure 4. A touch feedback animation.

Drawables



Drawables

TRY THE COMPOSE WAY | Work with Drawables in Compose →

TRY THE COMPOSE WAY

Work with Drawables in Compose

arrow_forward

These capabilities for drawables help you implement Material Design apps:

- **Vector drawables** are scalable without losing definition and are perfect for single-color in-app icons. Learn more about vector drawables.
- **Drawable tinting** lets you define bitmaps as an alpha mask and tint them with a color at runtime. See how to add tint to drawables.
- **Color extraction** lets you automatically extract prominent colors from a bitmap image. See how to select colors with the Palette API.

ImageBitmap

In Compose, a raster image (often referred to as a Bitmap) can be loaded up into an ImageBitmap instance, and a BitmapPainter is what is responsible for drawing the bitmap to screen.

For simple use cases, the painterResource() can be used which takes care of creating an ImageBitmap and returns a Painter object (in this case - a BitmapPainter):

```
Image(
    painter = painterResource(id = R.drawable.dog),
    contentDescription = stringResource(id = R.string.dog_content_description)
)
```

VectorVsBitmapSnippets.kt

If you require further customization (for instance a custom painter implementation) and need access to the ImageBitmap itself, you can load it in the following way:

```
val imageBitmap = ImageBitmap.imageResource(R.drawable.dog)
```

VectorVsBitmapSnippets.kt

ImageVector

A VectorPainter is responsible for drawing an ImageVector to screen. ImageVector supports a subset of SVG commands. Not all images can be represented as vectors (for example, the photos you take with your camera cannot be transformed into a vector).

You can create a custom ImageVector either by importing an existing vector drawable XML file (imported into Android Studio using the import tool) or implementing the class and issuing path commands manually.

For simple use cases, the same way in which painterResource() works for the ImageBitmap class, it also works for ImageVectors, returning a VectorPainter as the result. painterResource() handles the loading of VectorDrawables and BitmapDrawables into VectorPainter and BitmapPainter respectively.
To load a VectorDrawable into an image, use:

```
Image(
        painter = painterResource(id = R.drawable.baseline_shopping_cart_24),
        contentDescription = stringResource(id = R.string.shopping_cart_content_desc)
)
```

VectorVsBitmapSnippets.kt

If you'd require further customization and need to access to the ImageVector itself, you can load it in the following way:

```
val imageVector = ImageVector.vectorResource(id = R.drawable.baseline_shopping_cart_24)
```

Executed Output:

Result

Module -6

Navigation: Back-button navigation, Hierarchical navigation patterns, Ancestral navigation (Up button), Descendant navigation, Lateral navigation with tabs and swipes

Providing users with a path through your app

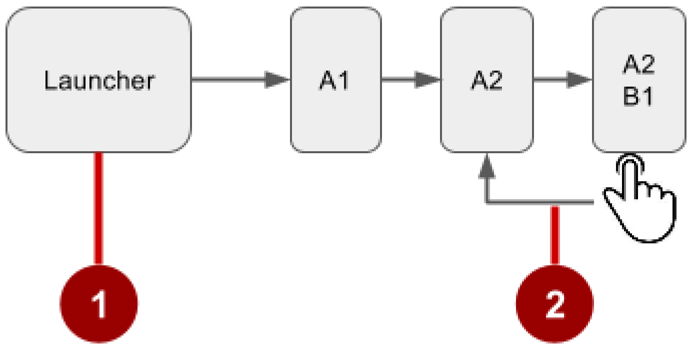
In the early stages of developing an app, you should determine the path you want users to take through your app to do each task. (The tasks are things like placing an order or browsing content.) Each path enables users to navigate across, into, and out of the tasks and pieces of content within the app.

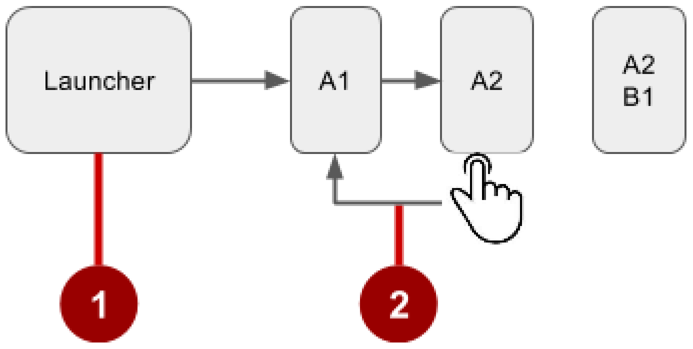
Often you need several paths through your app that offer the following types of navigation:

- *Back* navigation, where users navigate to the previous screen using the Back button.
- *Hierarchical* navigation, where users navigate through a hierarchy of screens. The hierarchy is organized with a *parent* screen for every set of *child* screens.

Back-button navigation

Back-button navigation—navigation back through the history of screens—is deeply rooted in the Android system. Android users expect the Back button in the bottom left corner of every screen to take them to the previous screen. The set of historical screens always starts with the user's Launcher (the device's Home screen), as shown in the figure below. Pressing Back enough times should return the user back to the Launcher.





In the figure above:

1. Starting from Launcher.
2. Clicking the Back button to navigate to the previous screen.

You don't have to manage the Back button in your app. The system handles tasks and the *back stack*—the list of previous screens—automatically. The Back button by default simply traverses this list of screens, removing the current screen from the list as the user presses it.

There are, however, cases where you may want to override the behavior for the Back button. For example, if your screen contains an embedded web browser in which users can interact with page elements to navigate between web pages, you may wish to trigger the embedded browser's default back behavior when users press the device's Back button.

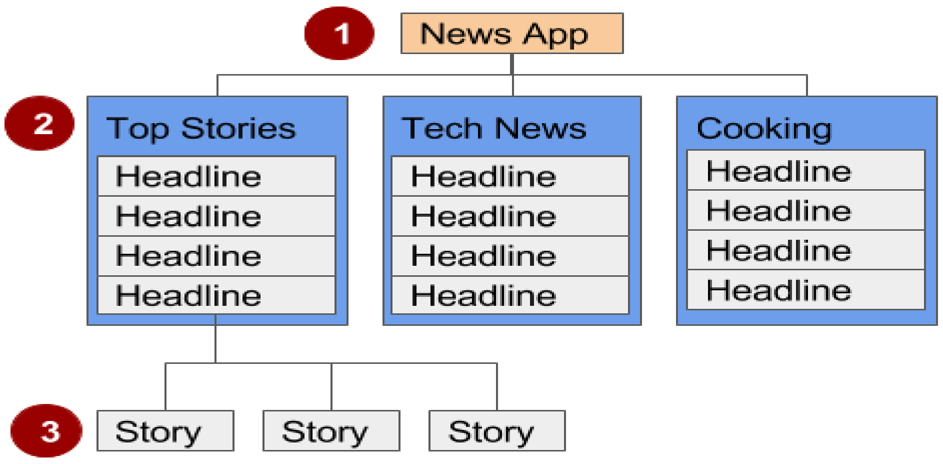
The onBackPressed() method of the Activity class is called whenever the Activity detects the user's press of the Back key. The default implementation simply finishes the current Activity, but you can override this to do something else:

```
@Override
public void onBackPressed() {
   // Add the Back key handler here.
```

```
    return;
}
```
If your code triggers an embedded browser with its own behavior for the Back key, you should return the Back key behavior to the system's default behavior if the user uses the Back key to go beyond the beginning of the browser's internal history.

Hierarchical navigation patterns

To give the user a path through the full range of an app's screens, the best practice is to use some form of hierarchical navigation. An app's screens are typically organized in a parent-child hierarchy, as shown in the figure below:



In the figure above:

1. Parent screen
2. First-level child screen siblings
3. Second-level child screen siblings

Parent screen

A parent screen (such as a news app's home screen) enables navigation down to *child* screens.

- The main Activity of an app is usually the parent screen.
- Implement a parent screen as an activity with *descendant* navigation to one or more child screens.

First-level child screen siblings

Siblings are screens in the same position in the hierarchy that share the same parent screen (like brothers and sisters).

- In the first level of siblings, the child screens may be *collection* screens that collect the headlines of stories, as shown above.
- Implement each child screen as an Activity or Fragment.
- Implement *lateral* navigation to navigate from one sibling to another on the same level.
- If there is a second level of screens, the first level child screen is the *parent* to the second level child screen siblings. Implement *descendant* navigation to the second-level child screens.

Second-level child screen siblings

In news apps and others that offer multiple levels of information, the second level of child screen siblings might offer content, such as stories.

- Implement each second-level child screen sibling as another Activity or Fragment.
- Stories at this level may include embedded story elements such as videos, maps, and comments, which might be implemented as fragments.

You can enable the user to navigate up to and down from a parent, and sideways among siblings:

- *Descendant* navigation: Navigating down from a parent screen to a child screen.
- *Ancestral* navigation: Navigating up from a child screen to a parent screen.
- *Lateral* navigation: Navigating from one sibling to another sibling (at the same level).

You can use the main Activity of the app as a parent screen, and then add an Activity or Fragment for each child screen.

Main Activity with an activity for each child

If the first-level child screen siblings have another level of child screens under them, you should implement each first-level screen as an activity, so that the lifecycle of each screen is managed properly before calling any second-level child screens.

For example, in the figure above, the parent screen is most likely the main activity. An app's main activity (usually MainActivity.java) is typically the parent screen for all other screens in your app. You implement a navigation pattern in the main activity to enable the user to go to another activity or fragment. For example, you can implement navigation using an Intent that starts an Activity.
Tip: Using an Intent in the current activity to start another activity adds the current activity to the call stack, so that the Back button in the other activity (described in the previous section) returns the user to the current activity.
As you've learned, the Android system initiates code in an Activity with callback methods that manage the Actactivityivity lifecycle for you. (A previous lesson covers the activity lifecycle; for more information, see Activities in the Android developer documentation.)
The declaration of each child activity is defined in the AndroidManifest.xml file with its parent activity. For example, the following defines OrderActivity as a child of the parent MainActivity:

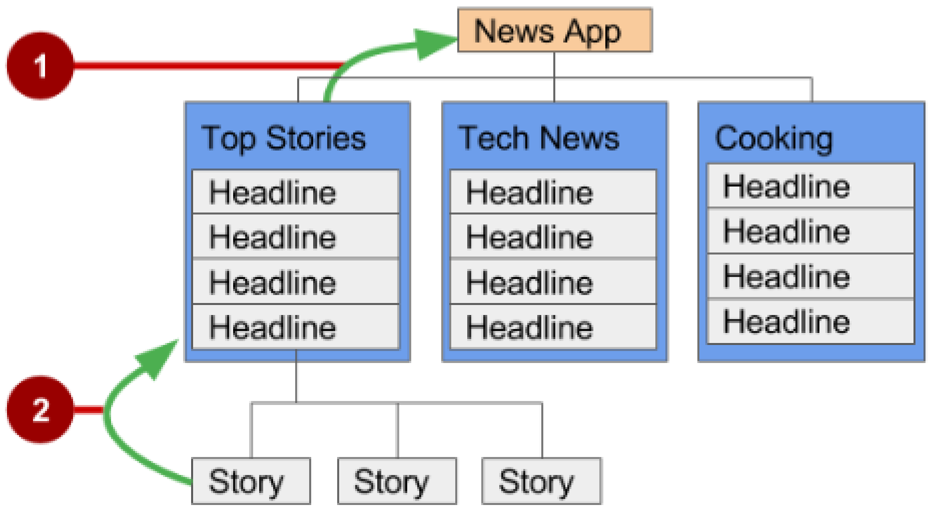
```
<activity android:name=".OrderActivity"
  android:label="@string/title_activity_order"
  android:parentActivityName=
                  "com.example.android.droidcafe.MainActivity">
  <meta-data
    android:name="android.support.PARENT_ACTIVITY"
    android:value=".MainActivity"/>
</activity>
```

Main Activity with a Fragment for each child

If the child screen siblings do *not* have another level of child screens under them, you can define each one as a Fragment, which represents a behavior or portion of a UI within in an activity. Think of a fragment as a modular section of an activity which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running. You can add more than one fragment in a single activity. For example, in a section sibling screen showing a news story and implemented as an Activity, you might have a child screen for a video clip implemented as a Fragment. You would implement a way for the user to navigate to the video clip Fragment, and then back to the Activity that shows the story.

Ancestral navigation (the Up button)

With ancestral navigation in a multitier hierarchy, you enable the user to go *up* from a section sibling to the collection sibling, and then *up* to the parent screen.



In the figure above:

1. **Up** button for ancestral navigation from the first-level siblings to the parent.
2. **Up** button for ancestral navigation from second-level siblings to the first-level child screen acting as a parent screen.

The **Up** button is used to navigate within an app based on the hierarchical relationships between screens. For example (referring to the figure above):

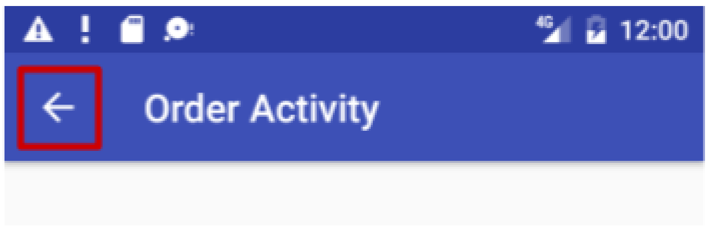
- If a first-level child screen offers headlines to navigate to second-level child screens, the second-level child screen siblings should offer **Up** buttons that return to the first-level child screen, which is their shared *parent*.
- If the parent screen offers navigation to first-level child siblings, then the first-level child siblings should offer an **Up** button that returns to the parent screen.
- If the parent screen is the topmost screen in an app (that is, the app's home screen), it should not offer an **Up** button.

Tip: The Back button below the screen differs from the **Up** button. The Back button provides navigation to whatever screen you viewed previously. If you have several children screens that the user can navigate through using a lateral navigation pattern (as described later in this chapter), the Back button would send the user back to the previous child screen, not to the parent screen. Use an **Up** button if you want to provide ancestral navigation from a child screen back to the parent screen. For more information about Up navigation, see Providing Up Navigation. See the concept chapter on menus and pickers for details on how to implement the app bar.

To provide the **Up** button for a child screen Activity, declare the parent of the Activity to be the main Activity in the AndroidManifest.xml file:

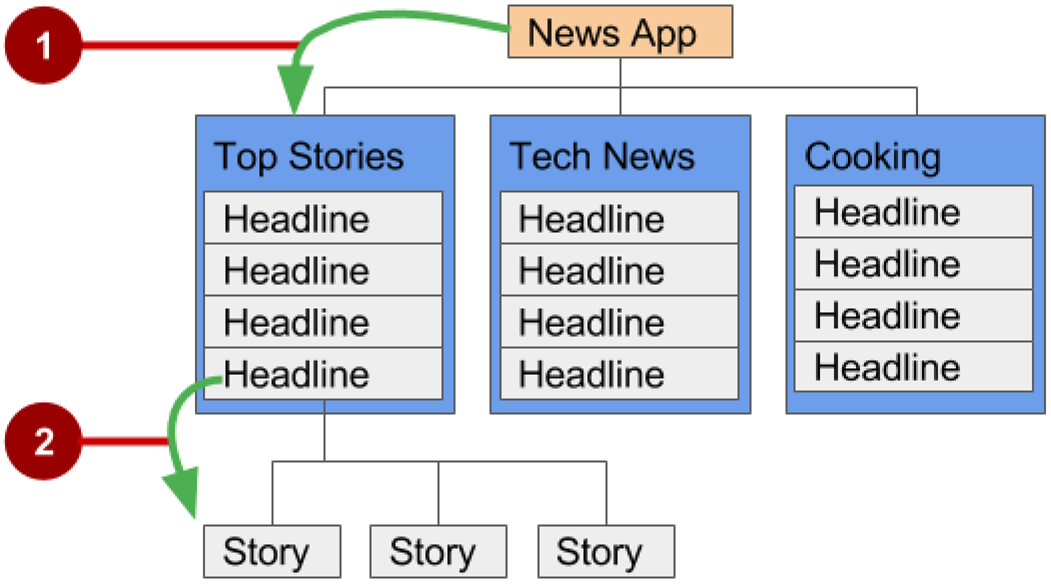
```
<activity android:name="com.example.android.droidcafeinput.OrderActivity"
    android:label="Order Activity"
    android:parentActivityName=".MainActivity">
    <meta-data android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity"/>
</activity>
```

The snippet above in AndroidManifest.xml declares the parent for the child screen OrderActivity to be MainActivity. It also sets the android:label to a title for the Activity screen to be "Order Activity". The child screen now includes the **Up** button in the app bar (highlighted in the figure below), which the user can tap to navigate back to the parent screen.



Descendant navigation

With descendant navigation, you enable the user to go from the parent screen to a first-level child screen, and from a first-level child screen down to a second-level child screen.



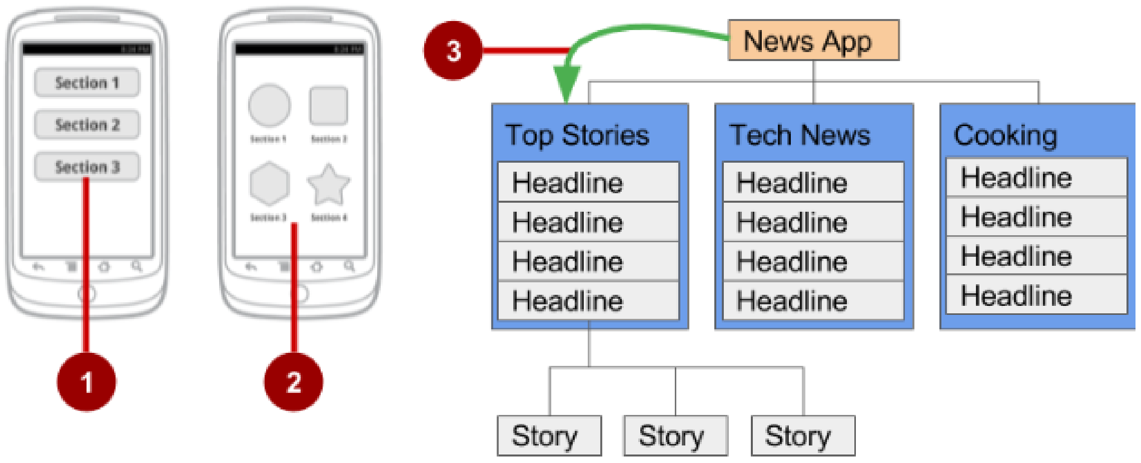
In the figure above:

1. Descendant navigation from parent to first-level child screen
2. Descendant navigation from headline in a first-level child screen to a second-level child screen

Buttons or targets

The best practice for descendant navigation from the parent screen to collection siblings is to use buttons or simple *targets* such as an arrangement of images or iconic buttons (also known as a *dashboard*). When the user touches a button, the collection sibling screen opens, replacing the current context (screen) entirely.

Tip: Buttons and simple targets are rarely used for navigating to section siblings *within* a collection. See lists, carousels, and cards in the next section.



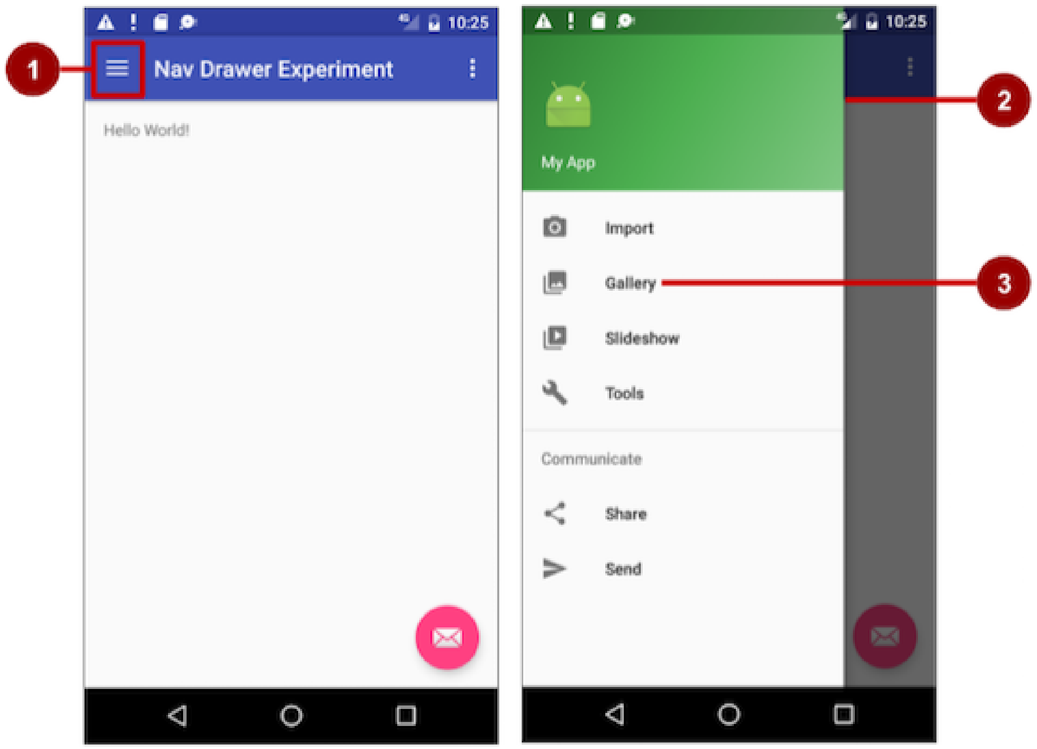
In the figure above:

1. Buttons on a parent screen
2. Targets (Image buttons or icons) on a parent screen
3. Descendant navigation pattern from parent screen to first-level child siblings

A dashboard usually has either two or three rows and columns, with large touch targets to make it easy to use. Dashboards are best when each collection sibling is equally important. You can use a LinearLayout, RelativeLayout, or GridLayout. See Layouts for an overview of how layouts work.

Navigation drawer

A *navigation drawer* is a panel that usually displays navigation options on the left edge of the screen, as shown on the right side of the figure below. It is hidden most of the time, but is revealed when the user swipes a finger from the left edge of the screen or touches the navigation icon in the app bar, as shown on the left side of the figure below.



In the figure above:

1. Navigation icon in the app bar
2. Navigation drawer
3. Navigation drawer menu item

A good example of a navigation drawer is in the Gmail app, which provides access to the inbox, labeled email folders, and settings. The best practice for employing a navigation drawer is to provide descendant navigation from the parent Activity to all of the other child screens in an app. It can display many navigation targets at once—for example, it can contain buttons (like a dashboard), tabs, or a list of items (like the Gmail drawer).
To make a navigation drawer in your app, you need to create the following layouts:

- A navigation drawer as the Activity layout root ViewGroup
- A navigation View for the drawer itself
- An app bar layout that includes room for a navigation icon button
- A content layout for the Activity that displays the navigation drawer
- A layout for the navigation drawer header

Follow these general steps:

1. Populate the navigation drawer menu with item titles and icons.
2. Set up the navigation drawer and item listeners in the Activity code.
3. Handle the navigation menu item selections.

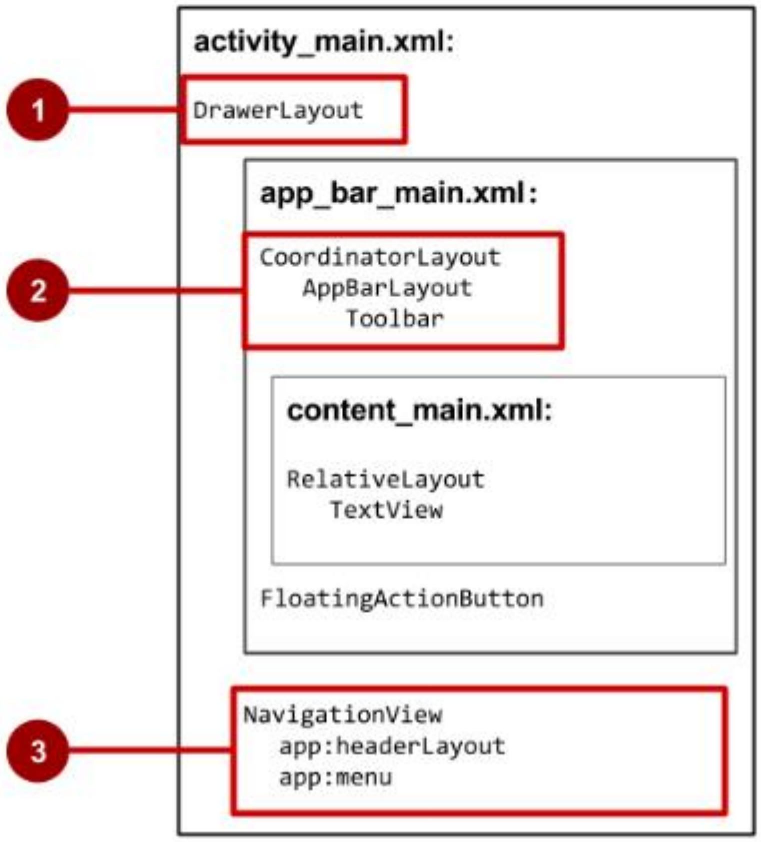
Creating the navigation drawer layout

To create a navigation drawer layout, use the DrawerLayout APIs available in the Support Library. For design specifications, follow the design principles for navigation drawers in the Navigation Drawer design guide.
To add a navigation drawer, use a DrawerLayout as the root ViewGroup of your Activity layout. Inside the DrawerLayout, add one View that contains the main content for the screen (your primary layout when the drawer is hidden) and another View, typically a NavigationView, that contains the contents of the navigation drawer.
Tip: To make your layouts simpler to understand, use the include tag to include an XML layout within another XML layout.
For example, the following layout uses:

- A DrawerLayout as the root of the Activity layout in activity_main.xml.
- The main content of screen defined in the app_bar_main.xml layout file.
- A NavigationView that represents a standard navigation menu that can be populated by a menu resource XML file.

Refer to the figure below that corresponds to this layout:

```
<android.support.v4.widget.DrawerLayout
    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:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:openDrawer="start">

    <include
        layout="@layout/app_bar_main"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <android.support.design.widget.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="true"
        app:headerLayout="@layout/nav_header_main"
        app:menu="@menu/activity_main_drawer" />

</android.support.v4.widget.DrawerLayout>
```



In the figure above:

1. DrawerLayout is the root ViewGroup of the Activity layout.

2. The included app_bar_main.xml uses a CoordinatorLayout as its root, and defines the app bar layout with a Toolbar which will include the navigation icon to open the drawer.
3. The NavigationView defines the navigation drawer layout and its header, and adds menu items to it.

Note the following in the activity_main.xml layout:

- The android:id for the DrawerLayout is drawer_layout. You will use this id to instantiate a drawer object in your code.
- The android:id for the NavigationView is nav_view. You will use this id to instantiate a navigationView object in your code.
- The NavigationView must specify its horizontal gravity with the android:layout_gravity attribute. Use the "start" value for this attribute (rather than "left"), so that if the app is used with right-to-left (RTF) languages, the drawer appears on the right rather than the left side.

```
android:layout_gravity="start"
```

- Use the android:fitsSystemWindows="true" attribute to set the padding of the DrawerLayout and the NavigationView to ensure the contents don't overlay the system windows. DrawerLayout uses fitsSystemWindows as a sign that it needs to inset its children (such as the main content ViewGroup), but still draw the top status bar background in that space. As a result, the navigation drawer appears to be overlapping, but not obscuring, the translucent top status bar. The insets you get from fitsSystemWindows will be correct on all platform versions to ensure that your content does not overlap with system-provided UI components.

The navigation drawer header

The NavigationView specifies the layout for the *header* of the navigation drawer with the attribute app:headerLayout="@layout/nav_header_main". The nav_header_main.xml file defines the layout of this header to include an ImageView and a TextView, which is typical for a navigation drawer, but you could also include other View elements.
Tip: The header's height should be 160dp, which you should extract into a dimension resource (nav_header_height).
The following is the code for the nav_header_main.xml file:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="@dimen/nav_header_height"
    android:background="@drawable/side_nav_bar"
    android:gravity="bottom"
    android:orientation="vertical"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:theme="@style/ThemeOverlay.AppCompat.Dark">

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/nav_header_vertical_spacing"
        android:src="@android:drawable/sym_def_app_icon" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/nav_header_vertical_spacing"
        android:text="@string/my_app_title"
        android:textAppearance="@style/TextAppearance.AppCompat.Body1" />

</LinearLayout>
```

The app bar layout

The include tag in the activity_main.xml layout file includes the app_bar_main.xml layout file, which uses a CoordinatorLayout as its root. The app_bar_main.xml file defines the app bar layout with the Toolbar class as shown previously in the chapter about menus and pickers. It also defines a floating action button, and uses an include tag to include the content_main.xml layout.
The following is the code for the app_bar_main.xml file:

```
<android.support.design.widget.CoordinatorLayout
    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"
    android:fitsSystemWindows="true"
    tools:context="com.example.android.navigationexperiments.MainActivity">

    <android.support.design.widget.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />

    </android.support.design.widget.AppBarLayout>

    <include layout="@layout/content_main" />

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="@dimen/fab_margin"
        android:src="@android:drawable/ic_dialog_email" />

</android.support.design.widget.CoordinatorLayout>
```

Note the following:

- The app_bar_main.xml layout uses a CoordinatorLayout as its root, and includes the content_main.xml layout.
- The app_bar_main.xml layout uses the android:fitsSystemWindows="true" attribute to set the padding of the app bar to ensure that it doesn't overlay the system windows such as the status bar.

The content layout for the main activity screen

The layout above uses an include tag to include the content_main.xml layout, which defines the layout of the main Activity screen. In the example layout below, the main Activity screen shows a TextView that displays the string "Hello World!". The following is the code for the content_main.xml file:

```
<RelativeLayout
    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"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:context="com.example.android.navigationexperiments.MainActivity"
    tools:showIn="@layout/app_bar_main">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />
</RelativeLayout>
```

Note the following:

- The content_main.xml layout must be the first child in the DrawerLayout because the drawer must be on top of the content. In our layout above, the content_main.xm layout is included in the app_bar_main.xml layout, which is the first child.
- The content_main.xml layout uses a RelativeLayout ViewGroup set to match the parent view's width and height, because it represents the entire UI when the navigation drawer is hidden.
- The layout *behavior* for the RelativeLayout is set to the string resource @string/appbar_scrolling_view_behavior, which controls the scrolling behavior of the screen in relation to the app bar at the top. The AppBarLayout.ScrollingViewBehavior class defines this behavior. View elements that scroll vertically should use this behavior, because it supports nested scrolling to automatically scroll any AppBarLayout siblings.

Populating the navigation drawer menu

The NavigationView in the activity_main.xml layout specifies the menu items for the navigation drawer using the following statement:

```
app:menu="@menu/activity_main_drawer"
```

The menu items are defined in the activity_main_drawer.xml file, which is located under **app > res > menu** in the **Project > Android** pane. The <group></group> tag defines a *menu group*—a collection of items that share traits, such as whether they are visible, enabled, or checkable. A group must contain one or more <item></> elements and be a child of a <menu> element, as shown below. In addition to defining each menu item's title with the android:title attribute, the file also defines each menu item's icon with the android:icon attribute.

The group is defined with the android:checkableBehavior attribute. This attribute lets you put interactive elements within the navigation drawer, such as toggle switches that can be turned on or off, and checkboxes and radio buttons that can be selected. The choices for this attribute are:

- single: Only one item from the group can be selected. Use for radio buttons.
- all: All items can be selected. Use for checkboxes.
- none: No items can be selected.

The following XML code snippet shows how to define a menu group:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">

   <group android:checkableBehavior="none">
      <item
         android:id="@+id/nav_camera"
         android:icon="@drawable/ic_menu_camera"
         android:title="@string/import_camera" />
```

```
        <item
            android:id="@+id/nav_gallery"
            android:icon="@drawable/ic_menu_gallery"
            android:title="@string/gallery" />
        <item
            android:id="@+id/nav_slideshow"
            android:icon="@drawable/ic_menu_slideshow"
            android:title="@string/slideshow" />
        <item
            android:id="@+id/nav_manage"
            android:icon="@drawable/ic_menu_manage"
            android:title="@string/tools" />
    </group>

    <item android:title="@string/communicate">
        <menu>
            <item
                android:id="@+id/nav_share"
                android:icon="@drawable/ic_menu_share"
                android:title="@string/share" />
            <item
                android:id="@+id/nav_send"
                android:icon="@drawable/ic_menu_send"
                android:title="@string/send" />
        </menu>
    </item>

</menu>
```

Setting up the navigation drawer and item listeners

To use a listener for the navigation drawer's menu items, the Activity hosting the navigation drawer must implement the OnNavigationItemSelectedListener interface:

1. Implement NavigationView.OnNavigationItemSelectedListener in the class definition:
2. public class MainActivity extends AppCompatActivity implements
3.   NavigationView.OnNavigationItemSelectedListener {

This interface offers the onNavigationItemSelected() method, which is called when an item in the navigation drawer menu item is tapped. As you enter OnNavigationItemSelectedListener, the red light bulb appears on the left margin.

4. Click the light bulb, choose **Implement methods**, and choose the **onNavigationItemSelected(item:MenuItem):boolean** method.

Android Studio adds a stub for the method:

```
@Override
public boolean onNavigationItemSelected(MenuItem item) {
    return false;
}
```

You learn how to use this stub in the next section.

5. Before setting up the navigation item listener, add code to the onCreate() method for the Activity to instantiate the DrawerLayout and NavigationView objects (drawer and navigationView in the code below):

```
 @Override
 protected void onCreate(Bundle savedInstanceState) {
   // ... Rest of onCreate code.
   DrawerLayout drawer = (DrawerLayout)
                   findViewById(R.id.drawer_layout);
   ActionBarDrawerToggle toggle =
           new ActionBarDrawerToggle(this, drawer, toolbar,
           R.string.navigation_drawer_open,
           R.string.navigation_drawer_close);
   if (drawer != null) {
     drawer.addDrawerListener(toggle);
   }
   toggle.syncState();

   NavigationView navigationView = (NavigationView)
                   findViewById(R.id.nav_view);
   if (navigationView != null) {
     navigationView.setNavigationItemSelectedListener(this);
   }
 }
```

The code above instantiates an ActionBarDrawerToggle, which substitutes a special drawable for the **Up** button in the app bar, and links the Activity to the DrawerLayout. The special drawable appears as a "hamburger" navigation icon when the drawer is closed, and animates into an arrow as the drawer opens.

Note: Be sure to use the ActionBarDrawerToggle in support-library-v7.appcompact, *not* the version in support-library-v4.

Tip: You can customize the animated toggle by defining the drawerArrowStyle in your ActionBar theme. For more detailed information about the ActionBar theme, see Adding the App Bar in the Android Developer documentation.

The code above implements addDrawerListener() to listen for drawer open and close events, so that when the user taps custom drawable button, the navigation drawer slides out.

You must also use the syncState() method of ActionBarDrawerToggle to synchronize the state of the drawer indicator. The synchronization must occur after the DrawerLayout instance state has been restored, and any other time when the state may have diverged in such a way that the ActionBarDrawerToggle was not notified.

The code above ends by setting a listener, setNavigationItemSelectedListener(), to the navigation drawer to listen for item clicks.

The ActionBarDrawerToggle also lets you specify the strings to use to describe the open/close drawer actions for accessibility services. Define the strings in your strings.xml file:

```
<string name="navigation_drawer_open">Open navigation drawer</string>
<string name="navigation_drawer_close">Close navigation drawer</string>
```

Handling navigation menu item selections

Add code to the onNavigationItemSelected() method stub to handle menu item selections. This method is called when an item in the navigation drawer menu is tapped. You can use switch case statements to take the appropriate action based on the menu item's id, which you can retrieve using the getItemId() method:

```
@Override
public boolean onNavigationItemSelected(MenuItem item) {
   DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
   // Handle navigation view item clicks here.
   switch (item.getItemId()) {
      case R.id.nav_camera:
         // Handle the camera import action (for now display a toast).
         drawer.closeDrawer(GravityCompat.START);
         displayToast(getString(R.string.chose_camera));
         return true;
      case R.id.nav_gallery:
         // Handle the gallery action (for now display a toast).
         drawer.closeDrawer(GravityCompat.START);
         displayToast(getString(R.string.chose_gallery));
         return true;
      case R.id.nav_slideshow:
         // Handle the slideshow action (for now display a toast).
         drawer.closeDrawer(GravityCompat.START);
         displayToast(getString(R.string.chose_slideshow));
         return true;
      case R.id.nav_manage:
         // Handle the tools action (for now display a toast).
         drawer.closeDrawer(GravityCompat.START);
         displayToast(getString(R.string.chose_tools));
         return true;
      case R.id.nav_share:
         // Handle the share action (for now display a toast).
         drawer.closeDrawer(GravityCompat.START);
         displayToast(getString(R.string.chose_share));
         return true;
      case R.id.nav_send:
         // Handle the send action (for now display a toast).
         drawer.closeDrawer(GravityCompat.START);
         displayToast(getString(R.string.chose_send));
         return true;
      default:
         return false;
```

```
  }
}
```
After the user taps a navigation drawer selection or taps outside the drawer, the DrawerLayout closeDrawer() method closes the drawer.

Lists and carousels

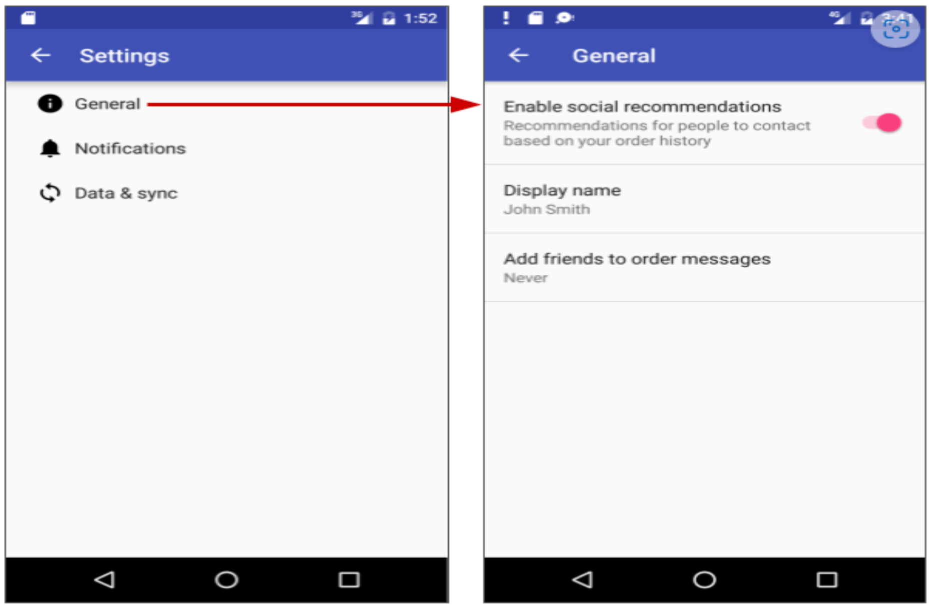
Use a scrolling list, such as a RecyclerView, to provide navigation targets for descendant navigation. Vertically scrolling lists are often used for a screen that lists stories, with each list item acting as a button to each story. For more visual or media-rich content items such as photos or videos, you may want to use a horizontally scrolling list (also known as a *carousel*). These UI elements are good for presenting items in a collection (for example, a list of news stories).
You learn about RecyclerView in another chapter.

Master/detail navigation flow

In a master/detail navigation flow, a master screen contains a list of items, and a detail screen shows detailed information about one item. You usually implement descendant navigation using one of the following techniques:

- Use an intent to starts an activity that represents the detail screen. For more information about intents, see Intents and Intent Filters in the Android developer documentation.
- When adding a Settings Activity, extend PreferenceActivity to create a two-pane master/detail layout to support large screens. Replace the activity content with a Settings Fragment. This is a useful pattern if you have multiple groups of settings and need to support tablet-sized screens as well as smartphones. You learn about the Settings activity and PreferenceActivity in another chapter. For more information about using fragments, see Fragments in the Android developer documentation.

Smartphones are best suited for displaying one screen at a time—for example a master screen (on the left side of the figure below) and a detail screen (on the right side of the figure below).



1:52

Settings

General

Notifications

Data & sync

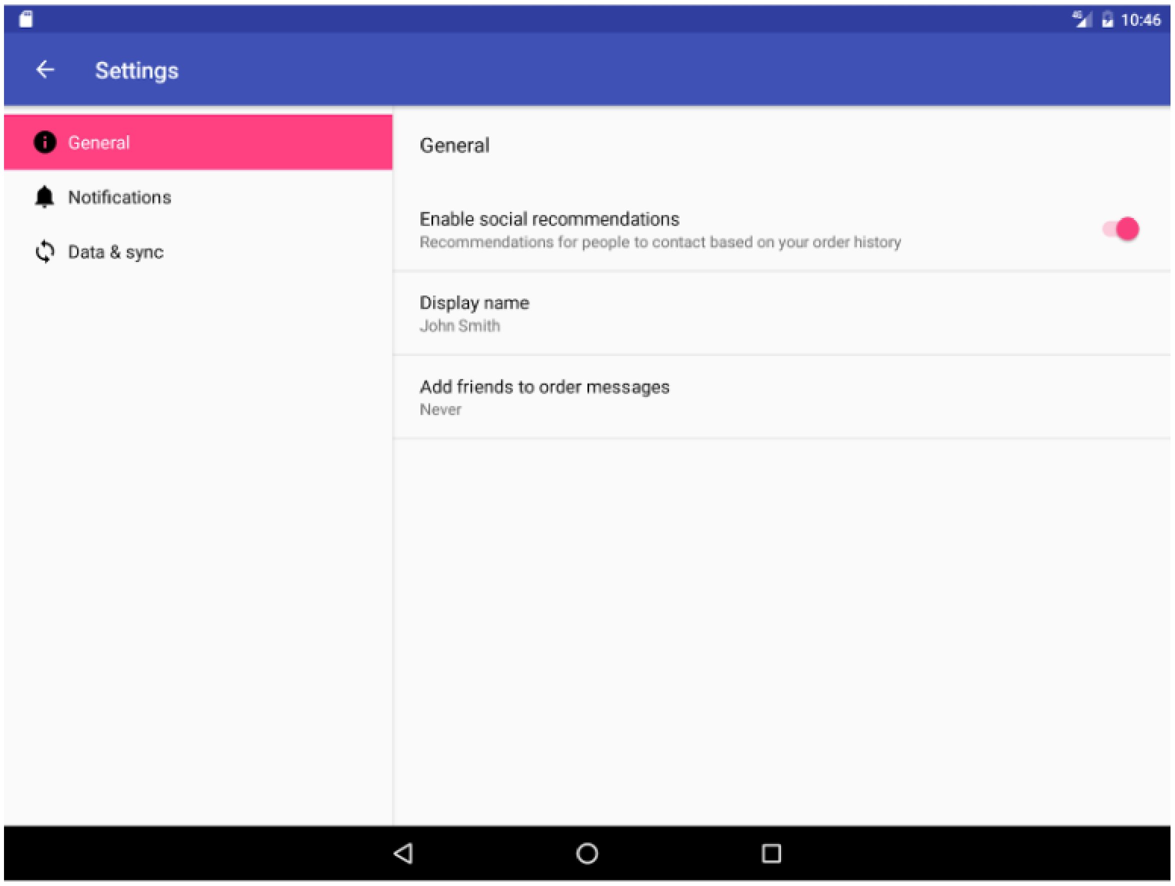
General

Enable social recommendations
Recommendations for people to contact based on your order history

Display name
John Smith

Add friends to order messages
Never

On the other hand, tablet displays, especially when viewed in the landscape orientation, are best suited for showing multiple content panes at a time: the master on the left, and the detail to the right, as shown below.



General

Enable social recommendations
Recommendations for people to contact based on your order history

Display name
John Smith

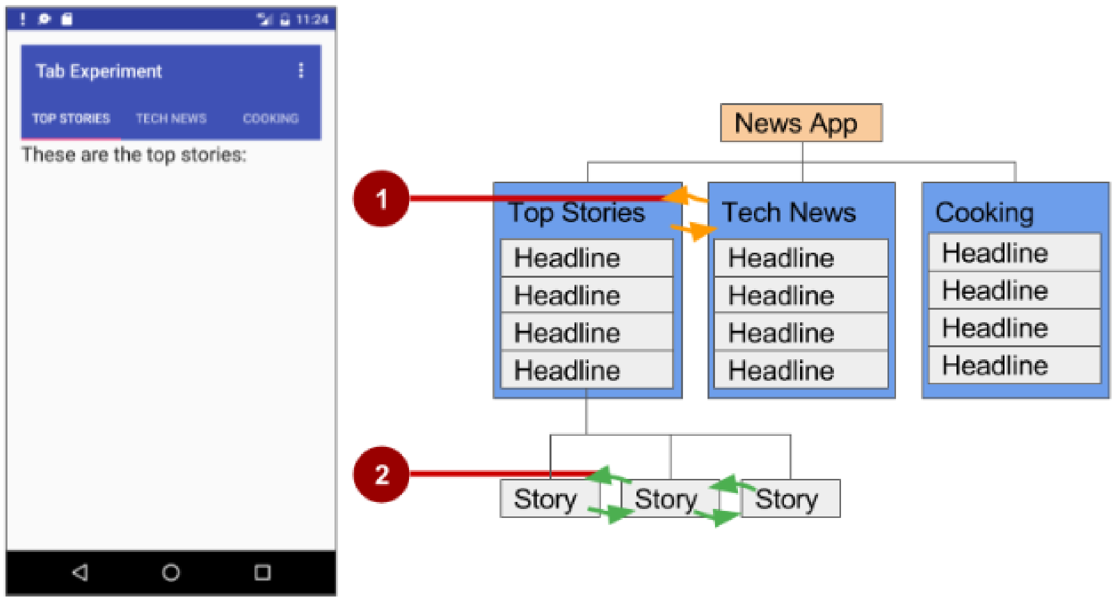
Add friends to order messages
Never

Options menu in the app bar

The app bar typically contains the options menu, which is most often used for navigation patterns for descendant navigation. It may also contain an **Up** button for ancestral navigation, a nav icon for opening a navigation drawer, and a filter icon to filter page views. You learn how to set up the options menu and the app bar in another chapter.

Lateral navigation with tabs and swipes

With lateral navigation, you enable the user to go from one sibling to another (at the same level in a multitier hierarchy). For example, if your app provides several categories of stories (such as Top Stories, Tech News, and Cooking, as shown in the figure below), you would want to provide your users the ability to navigate from one category to the next, or from one top story to the next, without having to navigate back up to the parent screen.



In the figure above:

1. Lateral navigation from one category screen to another
2. Lateral navigation from one story screen to another

Another example of lateral navigation is the ability to swipe left or right in a Gmail conversation to view a newer or older email in the same inbox.

You can implement lateral navigation with *tabs* that represent each screen. Tabs appear across the top of a screen, as shown on the left side of the figure above, providing navigation to other screens. Tab navigation is a common solution for lateral navigation from one child screen to another child screen that is a *sibling*—in the same position in the hierarchy and sharing the same parent screen.

Tabs are most appropriate for small sets (four or fewer) of sibling screens. You can combine tabs with swipe views, so that the user can swipe across from one screen to another as well as tap a tab.

Tabs offer two benefits:

- Because there is a single, initially selected tab, users already have access to that tab's content from the parent screen without any further navigation.
- Users can navigate quickly between related screens, without needing to first revisit the parent.

Keep in mind the following best practices when using tabs:

- Tabs are usually laid out horizontally.
- Tabs should always run along the top of the screen, and should not be aligned to the bottom of the screen.

- Tabs should be persistent across related screens. Only the designated content region should change when tapping a tab, and tab indicators should remain available at all times.
- Switching to another tab should not be treated as history. For example, if a user switches from tab A to tab B, pressing the **Up** button in the app bar should not reselect tab A but should instead return the user to the parent screen.

The key steps for implementing tabs are as follows:

1. Define the tab layout. The main class used for displaying tabs is TabLayout. It provides a horizontal layout to display tabs. You can show the tabs below the app bar.
2. Implement a Fragment for each tab content screen. A Fragment is a behavior or a portion of a UI within an Activity. It's like a mini-Activity within the main Activity, with its own lifecycle. One benefit of using a Fragment for each tabbed content is that you can isolate the code for managing the tabbed content inside the Fragment. To learn about Fragment, see Fragments in the API Guide.
3. Add a pager adapter. Use the PagerAdapter class to populate "pages" (screens) inside of a ViewPager, which is a layout manager that lets the user flip left and right through screens of data. You supply an implementation of a PagerAdapter to generate the screens that the View shows. ViewPager is most often used in conjunction with Fragment, which is a convenient way to supply and manage the lifecycle of each screen.
4. Create an instance of the tab layout, and set the text for each tab.
5. Use PagerAdapter to manage screens ("pages"). Each screen is represented by its own Fragment.
6. Set a listener to determine which tab is tapped.

There are standard adapters for using a Fragment with the ViewPager:

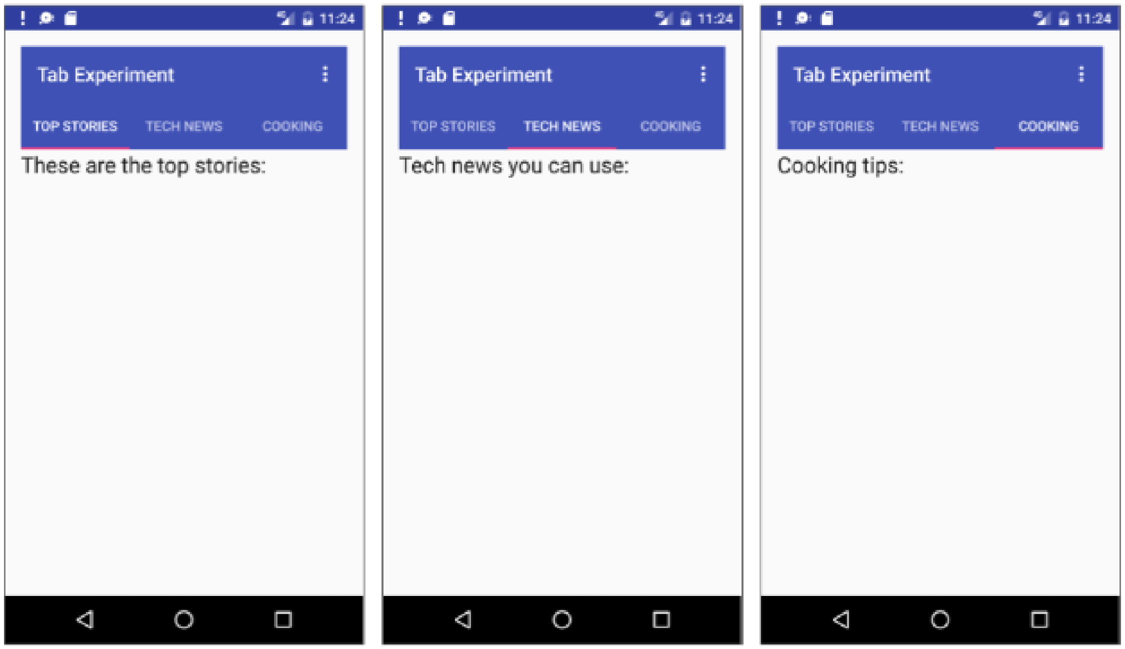
- FragmentPagerAdapter: Designed for navigating between sibling screens (pages) representing a fixed, small number of screens.
- FragmentStatePagerAdapter: Designed for paging across a collection of screens (pages) for which the number of screens is undetermined. It destroys each Fragment as the user navigates to another screen, minimizing memory usage.

Defining tab layout

To use a TabLayout, you can design the main Activity layout to use a Toolbar for the app bar, a TabLayout for the tabs below the app bar, and a ViewPager within the root layout to switch child elements. The layout should look similar to the following, assuming each child element fills the screen:

```
<android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:background="?attr/colorPrimary"
        android:minHeight="?attr/actionBarSize"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
        app:popupTheme="@style/ThemeOverlay.AppCompat.Light"/>

<android.support.design.widget.TabLayout
        android:id="@+id/tab_layout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/toolbar"
        android:background="?attr/colorPrimary"
        android:minHeight="?attr/actionBarSize"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"/>

<android.support.v4.view.ViewPager
        android:id="@+id/pager"
        android:layout_width="match_parent"
        android:layout_height="fill_parent"
        android:layout_below="@id/tab_layout"/>
```

For each child view, create a layout for each Fragment such as tab_fragment1.xml, tab_fragment2.xml, and so on.



Adding a pager adapter

Add a PagerAdapter that extends FragmentStatePagerAdapter. The code should do the following:

1. Define the number of tabs.
2. Use the getItem() method of the Adapter class to determine which tab is clicked.
3. Use a switch case block to return the screen (page) to show based on which tab is clicked.

The following is an example:

```
public class PagerAdapter extends FragmentStatePagerAdapter {
    int mNumOfTabs;

    public PagerAdapter(FragmentManager fm, int NumOfTabs) {
        super(fm);
        this.mNumOfTabs = NumOfTabs;
    }

    @Override
    public Fragment getItem(int position) {
        switch (position) {
            case 0: return new TabFragment1();
            case 1: return new TabFragment2();
            case 2: return new TabFragment3();
            default: return null;
        }
    }

    @Override
    public int getCount() {
        return mNumOfTabs;
    }
}
```

Creating an instance of the tab layout

In the onCreate() method of the main Activity, create an instance of the tab layout from the tab_layout element in the layout, and set the text for each tab using addTab():

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   // ... Rest of onCreate code
   // Create an instance of the tab layout from the view.
   TabLayout tabLayout = (TabLayout) findViewById(R.id.tab_layout);
   // Set the text for each tab.
   tabLayout.addTab(tabLayout.newTab().setText("Top Stories"));
   tabLayout.addTab(tabLayout.newTab().setText("Tech News"));
   tabLayout.addTab(tabLayout.newTab().setText("Cooking"));
   // Set the tabs to fill the entire layout.
   tabLayout.setTabGravity(TabLayout.GRAVITY_FILL);
   // Use PagerAdapter to manage page views in fragments.
}
```

Managing screen views in fragments with a listener

Use PagerAdapter in the onCreate() method of the main Activity to manage screen ("page") views in each Fragment. Each screen is represented by its own Fragment. You also need to set a listener to determine which tab is tapped. The following code should appear after the code from the previous section in the onCreate() method:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  // ... Rest of onCreate code
  // Use PagerAdapter to manage page views in fragments.
  final ViewPager viewPager = (ViewPager) findViewById(R.id.pager);
  final PagerAdapter adapter = new PagerAdapter
           (getSupportFragmentManager(), tabLayout.getTabCount());
  viewPager.setAdapter(adapter);
  // Setting a listener for clicks.
  viewPager.addOnPageChangeListener(new
           TabLayout.TabLayoutOnPageChangeListener(tabLayout));
  tabLayout.addOnTabSelectedListener(new
                    TabLayout.OnTabSelectedListener() {
    @Override
    public void onTabSelected(TabLayout.Tab tab) {
      viewPager.setCurrentItem(tab.getPosition());
    }

    @Override
      public void onTabUnselected(TabLayout.Tab tab) {

    }

    @Override
      public void onTabReselected(TabLayout.Tab tab) {
    }
  });
}
```

Using ViewPager for swipe views (horizontal paging)

ViewPager is a layout manager that lets the user flip left and right through "pages" (screens) of content. ViewPager is most often used in conjunction with Fragment, which is a convenient way to supply and manage the lifecycle of each "page". ViewPager also provides the ability to swipe "pages" horizontally.

In the previous example, you used a ViewPager within the root layout to switch child screens. This provides the ability for the user to swipe from one child screen to another. Users are able to navigate to sibling screens by touching and dragging the screen horizontally in the direction of the desired adjacent screen.

Swipe views are most appropriate where there is some similarity in content type among sibling pages, and when the number of siblings is relatively small. In these cases, this pattern can be used along with tabs above the content region to indicate the current page and available pages, to aid discoverability and provide more context to the user.

Module – 7

Connect to the Internet: Security best practices for network operations, Including permissions in the manifest, Performing network operations on a worker thread, Making an HTTP connection, Parsing the results, Managing the network state

Most Android apps engage the user with useful data. That data might be news articles, weather information, contacts, game statistics, and more. Often, data is provided over the network by a web API.

In this chapter you learn about network security and how to make network calls, which involves these general steps:

- Include permissions in your AndroidManifest.xml file.
- On a worker thread, make an HTTP client connection that connects to the network and downloads or uploads data.
- Parse the results, which are usually in JSON format.
- Check the state of the network and respond accordingly.

Network security

Network transactions are inherently risky, because they involve transmitting data that could be private to the user. People are increasingly aware of these risks, especially when their devices perform network transactions, so it's very important that your app implement best practices for keeping user data secure at all times.

Security best practices for network operations include:

- Use appropriate protocols for sensitive data. For example for secure web traffic, use the HttpsURLConnection subclass of HttpURLConnection.
- Use HTTPS instead of HTTP anywhere that HTTPS is supported on the server, because mobile devices frequently connect on insecure networks such as public Wi-Fi hotspots. Consider using SSLSocketClass to implement authenticated, encrypted socket-level communication.
- Don't use localhost network ports to handle sensitive interprocess communication (IPC), because other apps on the device can access these local ports. Instead, use a mechanism that lets you use authentication, for example, a Service.
- Don't trust data downloaded from HTTP or other insecure protocols. Validate input that's entered into a WebView and responses to intents that you issue against HTTP.

For more best practices and security tips, take a look at the Security Tips article.

Including permissions in the manifest

Before your app can make network calls, you need to include a permission in your AndroidManifest.xml file. Add the following tag inside the <manifest> tag:

```
<uses-permission android:name="android.permission.INTERNET" />
```

When using the network, it's a best practice to monitor the network state of the device so that you don't attempt to make network calls when the network is unavailable. To access the network state of the device, your app needs an additional permission:

```
<uses-permission
   android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Performing network operations on a worker thread

Always perform network operations on a worker thread, separate from the UI thread. For example, in your Java code you could create an AsyncTask (or AsyncTaskLoader) implementation that opens a network connection and queries an API. Your main code checks whether a network connection is active. If so, it runs the AsyncTask in a separate thread, then displays the results in the UI.

Note: If you run network operations on the main thread instead of on a worker thread, your code will throw a NetworkOnMainThreadException and your app will close.

Making an HTTP connection

Most network-connected Android apps use HTTP and HTTPS to send and receive data over the network. For a refresher on HTTP, visit this Learn HTTP tutorial.

Note: If a web server offers HTTPS, you should use it instead of HTTP for improved security.

The HttpURLConnection Android client supports HTTPS, streaming uploads and downloads, configurable timeouts, IPv6, and connection pooling. To use the HttpURLConnection client, build a URI (the request's destination). Then obtain a connection, send the request and any request headers, download and read the response and any response headers, and disconnect.

Building your URI

To open an HTTP connection, you need to build a request URI as a Uri object. A URI object is usually made up of a base URL and a collection of query parameters that specify the resource in question. For example to search for the first five book results for "Pride and Prejudice" in the Google Books API, use the following URI:

```
https://www.googleapis.com/books/v1/volumes?q=pride+prejudice&maxResults=5&printTy
pe=books
```

To construct a request URI programmatically, use the URI.parse() method with the buildUpon() and appendQueryParameter() methods. The following code builds the complete URI shown above:

```
// Base URL for the Books API.
```

```
final String BOOK_BASE_URL =
   "https://www.googleapis.com/books/v1/volumes?";

// Parameter for the search string
final String QUERY_PARAM = "q";
// Parameter to limit search results.
final String MAX_RESULTS = "maxResults";
// Parameter to filter by print type
final String PRINT_TYPE = "printType";

// Build up the query URI, limiting results to 5 printed books.
Uri builtURI = Uri.parse(BOOK_BASE_URL).buildUpon()
       .appendQueryParameter(QUERY_PARAM, "pride+prejudice")
       .appendQueryParameter(MAX_RESULTS, "5")
       .appendQueryParameter(PRINT_TYPE, "books")
       .build();
```

To convert the Uri object to a string, use the toString() method:

```
String myurl = builtURI.toString();
```

Connecting and downloading data

In the worker thread that performs your network transactions, for example within your override of the doInBackground() method in an AsyncTask, use the HttpURLConnection class to perform an HTTP GET request and download the data your app needs. Here's how:

1. To obtain a new HttpURLConnection, call URL.openConnection() using the URI that you've built. Cast the result to HttpURLConnection.
   The URI is the primary property of the request, but request headers can also include metadata such as credentials, preferred content types, and session cookies.

2. Set optional parameters. For a slow connection, you might want a long connection timeout (the time to make the initial connection to the resource) or read timeout (the time to actually read the data).

   To change the request method to something other than GET, use the setRequestMethod() method. If you won't use the network for input, call the setDoInput() method with an argument of false. (The default is true.)
   For more methods you can set, see the HttpURLConnection and URLConnection reference documentation.
3. Open an input stream using the getInputStream() method, then read the response and convert it into a string. Response headers typically include metadata such as the response body content type and length, modification dates, and session cookies. If the response has no body, getInputStream() returns an empty stream.
4. Call the disconnect() method to close the connection. Disconnecting releases the resources held by a connection so they can be closed or reused.

These steps are shown in the Request example, below.

Uploading data

If you're uploading (posting) data to a web server, you need to upload a *request body*, which holds the data to be posted. To do this:

1. Configure the connection so that output is possible by calling setDoOutput(true). By default, HttpURLConnection uses HTTP GET requests. When setDoOutput is true, HttpURLConnection uses HTTP POST requests instead.
2. Open an output stream by calling the getOutputStream() method.

For more about posting data to the network, see "Posting Content" in the HttpURLConnection documentation.
Note: All network calls must be performed in a worker thread and not on the UI thread.

Request example

The following example sends a request to the URL built in the Building your URI section, above. The request obtains a new HttpURLConnection, opens an InputStream, reads the response, converts the response into a string, and closes the connection.

```
private String downloadUrl(String myurl) throws IOException {
    InputStream inputStream = null;
    // Only display the first 500 characters of the retrieved
    // web page content.
    int len = 500;

    try {
        URL url = new URL(myurl);
        HttpURLConnection conn =
          (HttpURLConnection) url.openConnection();
        conn.setReadTimeout(10000 /* milliseconds */);
        conn.setConnectTimeout(15000 /* milliseconds */);
        // Start the query
        conn.connect();
        int response = conn.getResponseCode();
        Log.d(DEBUG_TAG, "The response is: " + response);
        inputStream = conn.getInputStream();

        // Convert the InputStream into a string
        String contentAsString =
           convertInputToString(inputStream, len);
        return contentAsString;

    // Close the InputStream and connection
    } finally {
          conn.disconnect();
        if (inputStream != null) {
            inputStream.close();
        }
    }
}
```

Converting the InputStream to a string

An InputStream is a readable source of bytes. Once you get an InputStream, it's common to decode or convert it into the data type you need. In the example above, the InputStream represents plain text from the web page located at https://www.googleapis.com/books/v1/volumes?q=pride+prejudice&maxResults=5&printType=books

The convertInputToString method defined below converts the InputStream to a string so that the activity can display it in the UI. The method uses an InputStreamReader instance to read bytes and decode them into characters:

```
// Reads an InputStream and converts it to a String.
public String convertInputToString(InputStream stream, int len)
        throws IOException, UnsupportedEncodingException {
    Reader reader = null;
    reader = new InputStreamReader(stream, "UTF-8");
    char[] buffer = new char[len];
    reader.read(buffer);
    return new String(buffer);
}
```

Note: If you expect a long response, wrap your InputStreamReader inside a BufferedReader for more efficient reading of characters, arrays, and lines. For example: ``` reader = new BufferedReader(new InputStreamReader(stream, "UTF-8")); ```

Parsing the results

When you make web API queries, the results are often in JSON format. Below is an example of a JSON response from an HTTP request. It shows the names of three menu items in a popup menu and the methods that are triggered when the menu items are clicked:

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}
```

To find the value of an item in the response, use methods from the JSONObject and JSONArray classes. For example, here's how to find the "onclick" value of the third item in the "menuitem" array:

```
JSONObject data = new JSONObject(responseString);
JSONArray menuItemArray = data.getJSONArray("menuitem");
JSONObject thirdItem = menuItemArray.getJSONObject(2);
String onClick = thirdItem.getString("onclick");
```

Managing the network state

Making network calls can be expensive and slow, especially if the device has little connectivity. Being aware of the network connection state can prevent your app from attempting to make network calls when the network isn't available.

Sometimes it's also important for your app to know what kind of connectivity the device has: Wi-Fi networks are typically faster than data networks, and data networks are often metered and expensive. To control when certain tasks are performed, monitor the network state and respond appropriately. For example, you may want to wait until the device is connected to Wifi to perform a large file download.

To check the network connection, use the following classes:

- ConnectivityManager answers queries about the state of network connectivity. It also notifies apps when network connectivity changes.
- NetworkInfo describes the status of a network interface of a given type (currently either mobile or Wi-Fi).

The following code snippet tests whether Wi-Fi and mobile are connected. In the code:

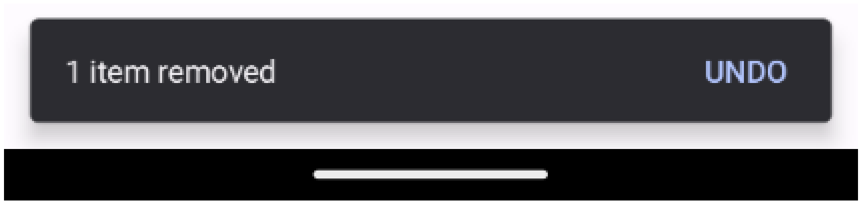
- The getSystemService() method gets an instance of ConnectivityManager from the context.
- The getNetworkInfo() method gets the status of the device's Wi-Fi connection, then its mobile connection. The getNetworkInfo() method returns a NetworkInfo object, which contains information about the given network's connection status (whether that connection is idle, connecting, and so on).
- The networkInfo.isConnected() method returns true if the given network is connected. If the network is connected, it can be used to establish sockets and pass data.

```
private static final String DEBUG_TAG = "NetworkStatusExample";
ConnectivityManager connMgr = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo networkInfo =
        connMgr.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
boolean isWifiConn = networkInfo.isConnected();
networkInfo =
        connMgr.getNetworkInfo(ConnectivityManager.TYPE_MOBILE);
boolean isMobileConn = networkInfo.isConnected();
Log.d(DEBUG_TAG, "Wifi connected: " + isWifiConn);
Log.d(DEBUG_TAG, "Mobile connected: " + isMobileConn);
```

Module – 8

Messages and Storage: Creating a Snackbar object, Showing the message to the user, instantiate a Toast object, Show the toast, Add Notification to your App, Customize Notifications, App-specific storage, Preferences, Room persistence library

You can add an action to a Snackbar to let the user respond to your message. When you do this, the Snackbar puts a button next to the message text, and the user can trigger your action by tapping the button. For example, an email app might put an *undo* button on its "email archived" message. If the user taps the *undo* button, the app takes the email back out of the archive.



igure 1. A Snackbar with an undo action button that restores a removed item.

To add an action to a Snackbar message, define a listener object that implements the View.OnClickListener interface. The system calls your listener's onClick() method if the user taps the message action. For example, this snippet shows a listener for an undo action:

KotlinJava

```
class MyUndoListener : View.OnClickListener {


  fun onClick(v: View) {
    // Code to undo the user's last action.
  }
}
```

Use one of the setAction() methods to attach the listener to your Snackbar. Attach the listener before you call show(), as shown in this code sample:

KotlinJava

```
val mySnackbar = Snackbar.make(findViewById(R.id.myCoordinatorLayout),
                         R.string.email_archived, Snackbar.LENGTH_SHORT)
```

mySnackbar.setAction(R.string.undo_string, MyUndoListener())

mySnackbar.show()

If you are using Jetpack Compose, you can show a SnackbarHost, as shown in the following example:

Kotlin

```
  override fun onCreate(savedInstanceState: Bundle?) {

    super.onCreate(savedInstanceState)

    setContent {
      DACPlaygroundTheme {
        val snackbarHostState = remember { SnackbarHostState() }
        val scope = rememberCoroutineScope()
        Scaffold(
          snackbarHost = { SnackbarHost(snackbarHostState) },
          content = { padding ->
            Button(
              modifier = Modifier.padding(padding),
              onClick = {
                scope.launch {
                  snackbarHostState.showSnackbar(
                    message = "1 item removed",
                    actionLabel = "UNDO",
                    duration = SnackbarDuration.Short
                  ).run {
                    when (this) {
                      Dismissed -> Log.d("SNACKBAR", "Dismissed")
                      ActionPerformed -> Log.d("SNACKBAR", "UNDO CLICKED")
                    }
                  }
                }
              }
            ) { Text("Show snackbar") }
          }
        )
      }
    }
  }
```

Result

Module -9

GeoLocation: Set up the project and get an API Key, Add Markers on the map, map Styles, Enable location tracking

Use API Keys
bookmark_border

Google Maps Platform products are secured from unauthorized use by restricting API calls to those that provide proper authentication credentials. These credentials are in the form of an API key - a unique alphanumeric string that associates your Google billing account with your project, and with the specific API or SDK.

This guide shows how to create, restrict, and use your API Key for Google Maps Platform.

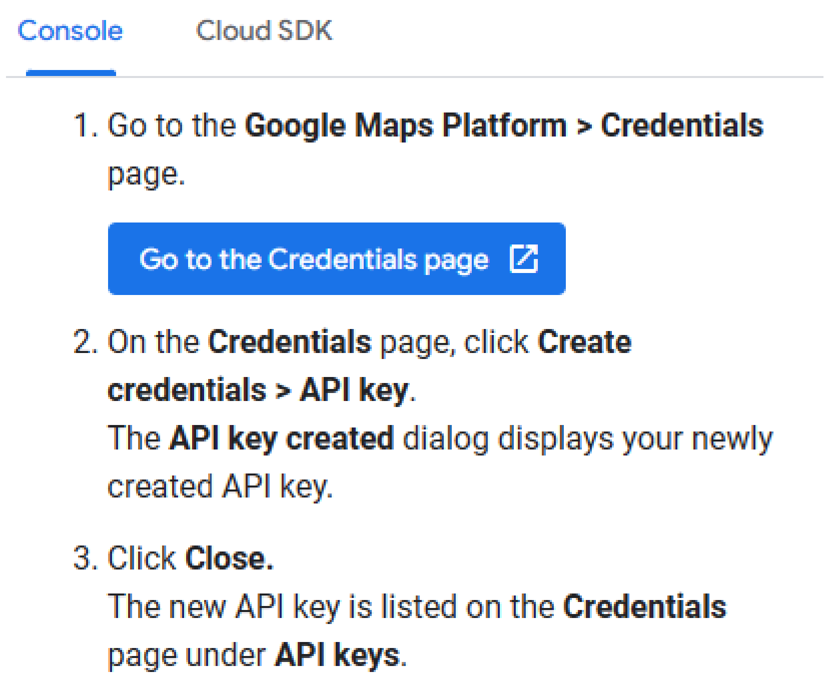
Before you begin

Before you start using the Maps JavaScript API, you need a project with a billing account and the Maps JavaScript API enabled. To learn more, see Set up in Cloud Console.

Create API keys

The API key is a unique identifier that authenticates requests associated with your project for usage and billing purposes. You must have at least one API key associated with your project.

To create an API key:



Console Cloud SDK

1. Go to the **Google Maps Platform > Credentials** page.

   Go to the Credentials page

2. On the **Credentials** page, click **Create credentials > API key**.
   The **API key created** dialog displays your newly created API key.

3. Click **Close.**
   The new API key is listed on the **Credentials** page under **API keys**.

Restrict API keys

Google strongly recommends that you restrict your API keys by limiting their usage to those only APIs needed for your application. Restricting API keys adds security to your application by protecting it from unwarranted requests. For more information, see API security best practices.

When restricting an API key in the Cloud Console, **Application restrictions** override any APIs enabled under **API restrictions**. Follow best practices by creating a separate API key for each app, and for each platform on which that app is available.

To restrict an API key:

ConsoleCloud SDK

1. Go to the **Google Maps Platform > Credentials** page.

   Go to the Credentials page

2. Select the API key that you want to set a restriction on. The API key property page appears.

3. Under **Key restrictions**, set the following restrictions:

- Application restrictions:

a. To accept requests from the list of websites that you supply, select **HTTP referers (web sites)** from the list of **Application restrictions**.

b. Specify one or more referrer websites. You can use wildcard characters to authorize all subdomains (for example, https://*.google.com accepts all sites ending in .google.com when accessed over HTTPS). Note that if you specify www.domain.com, it acts as a wildcard www.domain.com/*, and authorizes any subpath on that hostname. Specify https:// and http:// referrer schemes as-is. For other URL protocols, you must use a special representation. For example, format file:///path/to/ as __file_url__//path/to/*. After enabling websites, be sure to monitor your usage, to make sure it matches your expectations. The following protocols are supported: about://, app://, applewebdata://, asset://, chrome://, content://, file://, ftp://, ionic://, local://, ms-appx://, ms-appx-web://, ms-local-stream://, prism://, qrc://, res://, saphtmlp://.

- API restrictions:

a. Click **Restrict key**.

b. Select **Maps JavaScript API** from **Select APIs** dropdown. If the Maps JavaScript API is not listed, you need to enable it.

c. If your project uses Places Library, also select **Places API**. Similarly, if your project uses other services in the JavaScript API (Directions Service, Distance Matrix Service, Elevation Service, and/or Geocoding Service), you must also enable and select the corresponding API in this list.

4. To finalize your changes, click **Save**.

Add the API key to your request

You must include an API key with every Maps JavaScript API request. In the following example, replace YOUR_API_KEY with your API key.

```
<script>
  (g=>{var h,a,k,p="The Google Maps JavaScript
API",c="google",l="importLibrary",q="__ib__",m=document,b=window;b=b[c]||(b[c]={});var
d=b.maps||(b.maps={}),r=new Set,e=new URLSearchParams,u=()=>h||(h=new
Promise(async(f,n)=>{await (a=m.createElement("script"));e.set("libraries",[...r]+"");for(k in
g)e.set(k.replace(/[A-
Z]/g,t=>"_"+t[0].toLowerCase()),g[k]);e.set("callback",c+".maps."+q);a.src=`https://maps.${c}a
pis.com/maps/api/js?`+e;d[q]=f;a.onerror=()=>h=n(Error(p+" could not
load."));a.nonce=m.querySelector("script[nonce]")?.nonce||"";m.head.append(a)}));d[l]?console.
warn(p+" only loads once. Ignoring:",g):d[l]=(f,...n)=>r.add(f)&&u().then(()=>d[l](f,...n))})({
    key: "YOUR_API_KEY",
    v: "weekly",
    // Use the 'v' parameter to indicate the version to use (weekly, beta, alpha, etc.).
    // Add other bootstrap parameters as needed, using camel case.
  });
</script>
```

Result

Module – 10

Authentication: Add Firebase to the project, Email Authentication, Phone Authentication, Gmail Authentication

You can use Firebase Authentication to sign in a user by sending them an email containing a link, which they can click to sign in. In the process, the user's email address is also verified.

There are numerous benefits to signing in by email:

- Low friction sign-up and sign-in.
- Lower risk of password reuse across applications, which can undermine security of even well-selected passwords.
- The ability to authenticate a user while also verifying that the user is the legitimate owner of an email address.
- A user only needs an accessible email account to sign in. No ownership of a phone number or social media account is required.
- A user can sign in securely without the need to provide (or remember) a password, which can be cumbersome on a mobile device.
- An existing user who previously signed in with an email identifier (password or federated) can be upgraded to sign in with just the email. For example, a user who has forgotten their password can still sign in without needing to reset their password.

Before you begin

Set up your Android project

1. If you haven't already, add Firebase to your Android project.
2. In your **module (app-level) Gradle file** (usually <project>/<app-module>/build.gradle.kts or <project>/<app-module>/build.gradle), add the dependency for the Firebase Authentication Android library. We recommend using the Firebase Android BoM to control library versioning.

Also, as part of setting up Firebase Authentication, you need to add the Google Play services SDK to your app.

Kotlin+KTXJava

```
dependencies {
    // Import the BoM for the Firebase platform
    implementation(platform("com.google.firebase:firebase-bom:32.3.1"))

    // Add 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-ktx")

    // Also add the dependency for the Google Play services library and specify its version
    implementation("com.google.android.gms:play-services-auth:20.7.0")
}
```

By using the Firebase Android BoM, your app will always use compatible versions of Firebase Android libraries.

*(Alternative)*Add Firebase library dependencies*without*using the BoM

Enable Email Link sign-in for your Firebase project

To sign in users by email link, you must first enable the Email provider and Email link sign-in method for your Firebase project:

1. In the Firebase console, open the **Auth** section.
2. On the **Sign in method** tab, enable the **Email/Password** provider. Note that email/password sign-in must be enabled to use email link sign-in.
3. In the same section, enable **Email link (passwordless sign-in)** sign-in method.
4. Click **Save**.

Send an authentication link to the user's email address

To initiate the authentication flow, present the user with an interface that prompts the user to provide their email address and then call sendSignInLinkToEmail to request that Firebase send the authentication link to the user's email.

1. Construct the ActionCodeSettings object, which provides Firebase with instructions on how to construct the email link. Set the following fields:

- url: The deep link to embed and any additional state to be passed along. The link's domain has to be whitelisted in the Firebase Console list of authorized domains, which can be found by going to the Sign-in method tab (Authentication -> Sign-in method). The link will redirect the user to this URL if the app is not installed on their device and the app was not able to be installed.

- androidPackageName and IOSBundleId: The apps to use when the sign-in link is opened on an Android or Apple device. Learn more on how to configure Firebase Dynamic Links to open email action links via mobile apps.

- handleCodeInApp: Set to true. The sign-in operation has to always be completed in the app unlike other out of band email actions (password reset and email verifications). This is because, at the end of the flow, the user is expected to be signed in and their Auth state persisted within the app.

- dynamicLinkDomain: When multiple custom dynamic link domains are defined for a project, specify which one to use when the link is to be opened via a specified mobile app (for example, example.page.link). Otherwise the first domain is automatically selected.

  Kotlin+KTXJava

```
val actionCodeSettings = actionCodeSettings {
    // URL you want to redirect back to. The domain (www.example.com) for this
    // URL must be whitelisted in the Firebase Console.
    url = "https://www.example.com/finishSignUp?cartId=1234"
    // This must be true
    handleCodeInApp = true
    setIOSBundleId("com.example.ios")
    setAndroidPackageName(
        "com.example.android",
        true, // installIfNotAvailable
        "12", // minimumVersion
    )
}
```

  MainActivity.kt

  To learn more on ActionCodeSettings, refer to the Passing State in Email Actions section.

2. Ask the user for their email.

3. Send the authentication link to the user's email, and save the user's email in case the user completes the email sign-in on the same device.

   Kotlin+KTXJava

```
Firebase.auth.sendSignInLinkToEmail(email, actionCodeSettings)
    .addOnCompleteListener { task ->
        if (task.isSuccessful) {
            Log.d(TAG, "Email sent.")
        }
    }
```

   MainActivity.kt

   Complete sign in with the email link

   Security concerns

   To prevent a sign-in link from being used to sign in as an unintended user or on an unintended device, Firebase Auth requires the user's email address to be provided when completing the sign-in flow. For sign-in to succeed, this email address must match the address to which the sign-in link was originally sent.

   You can streamline this flow for users who open the sign-in link on the same device they request the link, by storing their email address locally - for instance using SharedPreferences - when you send the sign-in email. Then, use this address to complete the flow. Do not pass the user's email in the redirect URL parameters and re-use it as this may enable session injections.

   After sign-in completion, any previous unverified mechanism of sign-in will be removed from the user and any existing sessions will be invalidated. For example, if someone previously created an unverified account with the same email and password, the user's password will be removed to prevent the impersonator who claimed ownership and created that unverified account from signing in again with the unverified email and password.

   Also make sure you use an HTTPS URL in production to avoid your link being potentially intercepted by intermediary servers.

   Completing sign-in in an Android App

   Firebase Authentication uses Firebase Dynamic Links to send the email link to a mobile device. For sign-in completion via mobile application, the application has to be configured to detect the incoming application link, parse the underlying deep link and then complete the sign-in.

Configuring Firebase Dynamic Links

Firebase Auth uses Firebase Dynamic Links when sending a link that is meant to be opened in a mobile application. In order to use this feature, Dynamic Links **must** be configured in the Firebase Console.

1. Enable Firebase Dynamic Links:

a. In the Firebase console, open the **Dynamic Links** section.

b. If you have not yet accepted the Dynamic Links terms and created a Dynamic Links domain, do so now.

If you already created a Dynamic Links domain, take note of it. A Dynamic Links domain typically looks like the following example:

example.page.link

You will need this value when you configure your Apple or Android app to intercept the incoming link.

2. Configuring Android applications:

a. In order to handle these links from your Android application, the Android package name needs to be specified in the Firebase Console project settings. In addition, the SHA-1 and SHA-256 of the application certificate need to be provided.

b. Now that you have added a dynamic link domain and ensured that your Android app is configured correctly, the dynamic link will redirect to your application, starting from the launcher activity.

c. If you want the dynamic link to redirect to a specific activity, you will need to configure an intent filter in your **AndroidManifest.xml** file. This can be done by either specifying your dynamic link domain or the email action handler in the intent filter. By default, the email action handler is hosted on a domain like the following example:

PROJECT_ID.firebaseapp.com/

d. Caveats:

i. Do not specify the URL you set on the actionCodeSettings in your intent filter.

ii. When creating your dynamic link domain you may have also created a short URL link. This short URL will not be passed; **do not** configure your intent filter to catch it with an android:pathPrefix attribute. This means that you will not be able to catch different dynamic links in different parts of your application. However, you *can* check the mode query parameter in the link to see what operation is attempting to be performed, or use SDK methods such as isSignInWithEmailLink to see if a link that your app has received does what you want.

e. For more on receiving dynamic links, refer to Receiving Android Dynamic Links instructions.

Verify link and sign in

After you receive the link as described above, verify that it is meant for email link authentication and complete the sign in.

Kotlin+KTXJava

```
val auth = Firebase.auth
val intent = intent
val emailLink = intent.data.toString()

// Confirm the link is a sign-in with email link.
if (auth.isSignInWithEmailLink(emailLink)) {
    // Retrieve this from wherever you stored it
    val email = "someemail@domain.com"

    // The client SDK will parse the code from the link for you.
    auth.signInWithEmailLink(email, emailLink)
        .addOnCompleteListener { task ->
            if (task.isSuccessful) {
                Log.d(TAG, "Successfully signed in with email link!")
                val result = task.result
                // You can access the new user via result.getUser()
                // Additional user info profile *not* available via:
                // result.getAdditionalUserInfo().getProfile() == null
                // You can check if the user is new or existing:
                // result.getAdditionalUserInfo().isNewUser()
            } else {
                Log.e(TAG, "Error signing in with email link", task.exception)
            }
        }
}
```

MainActivity.kt

To learn more on how to handle sign-in with email link in an Apple application, refer to the Apple platforms guide.

To learn about how to handle sign-in with email link in a web application, refer to the Web guide.

Linking/re-authentication with email link

You can also link this method of authentication to an existing user. For example a user previously authenticated with another provider, such as a phone number, can add this method of sign-in to their existing account.

The difference would be in the second half of the operation:

Kotlin+KTXJava

```
// Construct the email link credential from the current URL.
val credential = EmailAuthProvider.getCredentialWithLink(email, emailLink)


// Link the credential to the current user.
Firebase.auth.currentUser!!.linkWithCredential(credential)
    .addOnCompleteListener { task ->
        if (task.isSuccessful) {
            Log.d(TAG, "Successfully linked emailLink credential!")
            val result = task.result
            // You can access the new user via result.getUser()
            // Additional user info profile *not* available via:
            // result.getAdditionalUserInfo().getProfile() == null
            // You can check if the user is new or existing:
```

```
            // result.getAdditionalUserInfo().isNewUser()
        } else {
            Log.e(TAG, "Error linking emailLink credential", task.exception)
        }
    }
```

MainActivity.kt

This can also be used to re-authenticate an email link user before running a sensitive operation.

Kotlin+KTXJava

```
// Construct the email link credential from the current URL.
val credential = EmailAuthProvider.getCredentialWithLink(email, emailLink)


// Re-authenticate the user with this credential.
Firebase.auth.currentUser!!.reauthenticateAndRetrieveData(credential)
    .addOnCompleteListener { task ->
        if (task.isSuccessful) {
            // User is now successfully reauthenticated
        } else {
            Log.e(TAG, "Error reauthenticating", task.exception)
        }
    }
```

MainActivity.kt

However, as the flow could end up on a different device where the original user was not logged in, this flow might not be completed. In that case, an error can be shown to the user to force them to open the link on the same device. Some state can be passed in the link to provide information on the type of operation and the user uid.

Differentiating email/password from email link

In case you support both password and link-based sign in with email, to differentiate the method of sign in for a password/link user, use fetchSignInMethodsForEmail. This is useful for identifier-first flows where the user is first asked to provide their email and then presented with the method of sign-in:

Kotlin+KTXJava

```
Firebase.auth.fetchSignInMethodsForEmail(email)
    .addOnSuccessListener { result ->
        val signInMethods = result.signInMethods!!
        if
(signInMethods.contains(EmailAuthProvider.EMAIL_PASSWORD_SIGN_IN_METHOD)) {
            // User can sign in with email/password
        } else if
(signInMethods.contains(EmailAuthProvider.EMAIL_LINK_SIGN_IN_METHOD)) {
            // User can sign in with email/link
        }
    }
    .addOnFailureListener { exception ->
        Log.e(TAG, "Error getting sign in methods for user", exception)
    }
```

MainActivity.kt

As described above email/password and email/link are considered the same EmailAuthProvider (same PROVIDER_ID) with different methods of sign-in.

Next steps

After a user signs in for the first time, a new user account is created and linked to the credentials—that is, the user name and password, phone number, or auth provider information—the user signed in with. This new account is stored as part of your Firebase project, and can be used to identify a user across every app in your project, regardless of how the user signs in.

- In your apps, you can get the user's basic profile information from the FirebaseUser object. See Manage Users.
- In your Firebase Realtime Database and Cloud Storage Security Rules, you can get the signed-in user's unique user ID from the auth variable, and use it to control what data a user can access.

You can allow users to sign in to your app using multiple authentication providers by linking auth provider credentials to an existing user account.

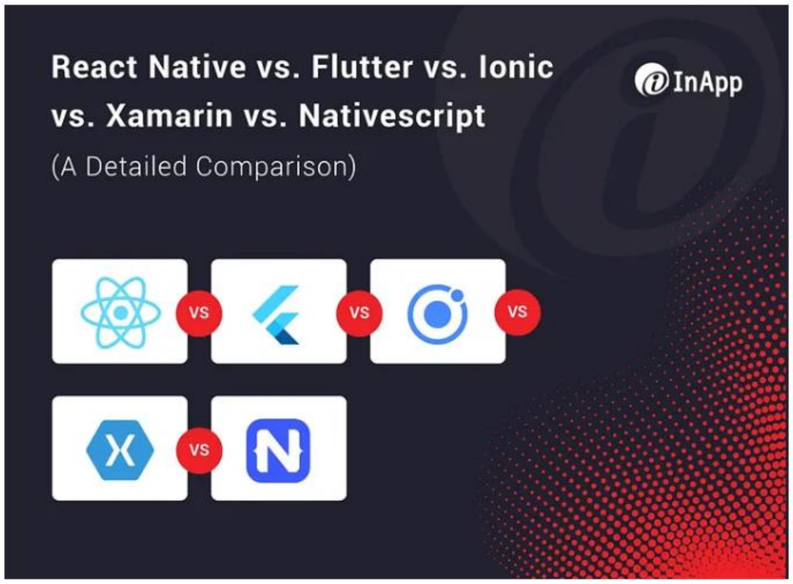
To sign out a user, call signOut:

Executed Output

Result

Module – 11

Hybrid App Development: Hybrid App vs Native App, React-Native, Flutter, Ionic, Xamarin


React Native vs. Flutter vs. Ionic vs. Xamarin vs. Nativescript
(A Detailed Comparison)
InApp
VS
VS
VS
VS

Cross-platform frameworks allow developers to create a mobile solution that is simultaneously compatible with different operating systems and platforms. That can help businesses expand the reach of a mobile app while also controlling the development costs. As a result, most companies prefer cross-platform mobile app development over native mobile app development.

Let's compare and contrast the top five mobile application frameworks: React Native, Flutter, Ionic, Xamarin, and Nativescript.

React Native

React Native is an open-source user interface (UI) software framework created by Meta Platforms, Inc., formerly Facebook. It’s used with native platform capabilities to develop applications for Android, Android TV, iOS, macOS, tvOS, web, Windows, and UWP.

Pros of React Native

- Has a single codebase for multiple applications
- Incorporates JavaScript, used by many programmers
- More choices for solutions, depending on the project’s requirements and the developer’s preferences
- Extensive libraries and UI frameworks
- Established a community of developers
- Less debugging time
- Easy to learn

Cons of React Native

- UI experience and performance are close to native applications, but not the same
- Some libraries are of low quality, and others have been abandoned
- Requires a timely decision on which navigation package to use
- Upgrades for app components are needed for every OS UI update

Flutter

Flutter is an open-source UI software development kit created by Google. It is used to develop cross-platform applications for Android, iOS, Linux, macOS, Windows, Google Fuchsia, and the web from a single codebase.

Pros of Flutter

- Allows developers to make changes in their codebase without restarting the application to reflect the updates
- Can create apps for both iOS and Android platforms with one codebase
- Lower debugging and testing time
- Uses Skia Graphics Library for fast and smooth performance
- Developers can experiment without spending money on licensing fees
- Apps look the same on older iOS or Android systems
- Easy to create a user-friendly UI

Cons of Flutter

- Apps are bigger in size than native ones

- The community of experienced Flutter developers is not yet as wide and established as others
- Dart programming language for Flutter is not as widely used
- We may need to build customized functionalities
- Reliant on continuous support from Google over the long term

Ionic

Ionic is an open-source UI toolkit for creating hybrid cross-platform mobile applications. It uses Webview for mobile instead of using native device elements. The framework leverages familiar JavaScript codebases and has multiple component presets that provide native functionality.

Pros of Ionic

- Reduces the time, effort, and resources employed to build a cross-platform application while giving it a native look and feel
- Saves build time by providing a simplistic interface for accessing native SDK and native API on each platform

- Creates a single codebase by using familiar Javascript frameworks and libraries, reducing code rewrites
- Scales efficiently as the number of active users doesn’t affect its performance

Cons of Ionic

- Developers may need to create highly specific features
- Refreshes the whole app whenever the developer makes some changes, affecting the development speed
- Older versions won’t provide code uglification

Xamarin

Owned by Microsoft, Xarmain uses a C#-shared codebase that developers can use to write native Android, iOS, and Windows apps with native user interfaces and share code across multiple platforms, including Windows, macOS, and Linux.

Pros of Xamarin

- Uses C# with .Net framework to create apps for any mobile platform
- Cross-platform development tools as a built-in part of the IDE at no additional cost

- Apps can still be classified as native
- Good app performance
- Can build platform-specific UI elements
- Many libraries are available and support linking with native libraries
- Eliminates hardware compatibility issues
- Vibrant community
- Hot reload
- Supports TVs, wearables, and IoT

Cons of Xamarin

- Slightly delayed support for platform updates
- Limited access to open-source libraries
- Costs for professional and enterprise use
- Limited developer pool and community
- Still need basic knowledge of native languages
- Larger app size
- Compatibility issues with third-party libraries and tools

Nativescript

NativeScript is an open-source framework that builds cross-platform applications for iOS and Android using JavaScript. NativeScript also has a rendering engine that provides native performance and user experience.

Pros of NativeScript

- Uses Angular, TypeScript, or JavaScript for convenient data binding and more component reusability
- Accesses native device API via native components developed with native performance
- Uses a markup language like HTML to develop applications with customized features
- Complete and direct access to all kinds of iOS and Android APIs
- NativeScript CLI allows multiple functions like adding a platform or deploying apps to a specific platform or device
- Faster installation of plugins and app debugging

Cons of NativeScript

- Requires learning different UI components

- Fewer verified plugins
- Developers must know the native functionality and APIs of iOS and Android to access the hardware of a device and any other platform-specific elements
- The app must be tested on an emulator or an actual device, which slows the testing process

Module – 12

Publish App to Play Store: Add a launcher icon and Application ID, Specify API Level targets and version number, Disable logging and debugging, Generate signed APK for release, Create a Google Developer Account, Run pre-launch reports, Review criteria for publishing, Submit your app for publishing.

Adding a launcher icon

When a new Flutter app is created, it has a default launcher icon. To customize this icon, you might want to check out the flutter_launcher_icons package.

Alternatively, you can do it manually using the following steps:

1. Review the Material Design product icons guidelines for icon design.
2. In the [project]/android/app/src/main/res/ directory, place your icon files in folders named using configuration qualifiers. The default mipmap- folders demonstrate the correct naming convention.
3. In AndroidManifest.xml, update the application tag's android:icon attribute to reference icons from the previous step (for example, <application android:icon="@mipmap/ic_launcher" ...).
4. To verify that the icon has been replaced, run your app and inspect the app icon in the Launcher.

Enabling Material Components

If your app uses Platform Views, you might want to enable Material Components by following the steps described in the Getting Started guide for Android.

For example:

1. Add the dependency on Android's Material in <my-app>/android/app/build.gradle:

content_copy

```
dependencies {
   // ...
   implementation 'com.google.android.material:material:<version>'
   // ...
}
```

To find out the latest version, visit Google Maven.

1. Set the light theme in <my-app>/android/app/src/main/res/values/styles.xml:

content_copy

```
-<style name="NormalTheme" parent="@android:style/Theme.Light.NoTitleBar">
+<style name="NormalTheme" parent="Theme.MaterialComponents.Light.NoActionBar">
```

1. Set the dark theme in <my-app>/android/app/src/main/res/values-night/styles.xml

content_copy

```
-<style name="NormalTheme" parent="@android:style/Theme.Black.NoTitleBar">
+<style name="NormalTheme" parent="Theme.MaterialComponents.DayNight.NoActionBar">
```

Signing the app

To publish on the Play Store, you need to give your app a digital signature. Use the following instructions to sign your app.

On Android, there are two signing keys: deployment and upload. The end-users download the .apk signed with the 'deployment key'. An 'upload key' is used to authenticate the .aab / .apk uploaded by developers onto the Play Store and is re-signed with the deployment key once in the Play Store.

- It's highly recommended to use the automatic cloud managed signing for the deployment key. For more information, check out the official Play Store documentation.

Create an upload keystore
If you have an existing keystore, skip to the next step. If not, create one by either:

- Following the Android Studio key generation steps
- Running the following at the command line:

  On macOS or Linux, use the following command:

  content_copy

  ```
  keytool -genkey -v -keystore ~/upload-keystore.jks -keyalg RSA \
          -keysize 2048 -validity 10000 -alias upload
  ```
  On Windows, use the following command in PowerShell:

  content_copy

  ```
  keytool -genkey -v -keystore %userprofile%\upload-keystore.jks ^
          -storetype JKS -keyalg RSA -keysize 2048 -validity 10000 ^
          -alias upload
  ```
  This command stores the upload-keystore.jks file in your home directory. If you want to store it elsewhere, change the argument you pass to the -keystore parameter. **However, keep the keystore file private; don't check it into public source control!**

info **Note:**

- The keytool command might not be in your path—it’s part of Java, which is installed as part of Android Studio. For the concrete path, run flutter doctor -v and locate the path printed after ‘Java binary at:’. Then use that fully qualified path replacing java (at the end) with keytool. If your path includes space-separated names, such as Program Files, use platform-appropriate notation for the names. For example, on Mac/Linux use Program\ Files, and on Windows use "Program Files".
- The -storetype JKS tag is only required for Java 9 or newer. As of the Java 9 release, the keystore type defaults to PKS12.

Reference the keystore from the app

Create a file named [project]/android/key.properties that contains a reference to your keystore. Don’t include the angle brackets (< >). They indicate that the text serves as a placeholder for your values.

content_copy

```
storePassword=<password-from-previous-step>
keyPassword=<password-from-previous-step>
keyAlias=upload
storeFile=<keystore-file-location>
```

The storeFile might be located at /Users/<user name>/upload-keystore.jks on macOS or C:\\Users\\<user name>\\upload-keystore.jks on Windows.

report_problem **Warning:** Keep the key.properties file private; don’t check it into public source control.

Configure signing in gradle

Configure gradle to use your upload key when building your app in release mode by editing the [project]/android/app/build.gradle file.

1. Add the keystore information from your properties file before the android block:

   content_copy

   ```
   def keystoreProperties = new Properties()
   def keystorePropertiesFile = rootProject.file('key.properties')
   if (keystorePropertiesFile.exists()) {
       keystoreProperties.load(new FileInputStream(keystorePropertiesFile))
   }

   android {
         ...
   ```

```
}
```

Load the key.properties file into the keystoreProperties object.

2. Find the buildTypes block:

content_copy

```
buildTypes {
    release {
        // TODO: Add your own signing config for the release build.
        // Signing with the debug keys for now,
        // so `flutter run --release` works.
        signingConfig signingConfigs.debug
    }
}
```

And replace it with the following signing configuration info:

content_copy

```
signingConfigs {
    release {
        keyAlias keystoreProperties['keyAlias']
        keyPassword keystoreProperties['keyPassword']
        storeFile keystoreProperties['storeFile'] ? file(keystoreProperties['storeFile']) : null
        storePassword keystoreProperties['storePassword']
    }
}
buildTypes {
    release {
        signingConfig signingConfigs.release
    }
}
```

Release builds of your app will now be signed automatically.

info **Note:** You might need to run flutter clean after changing the gradle file. This prevents cached builds from affecting the signing process.

For more information on signing your app, check out Sign your app on developer.android.com.

Shrinking your code with R8

R8 is the new code shrinker from Google, and it’s enabled by default when you build a release APK or AAB. To disable R8, pass the --no-shrink flag to flutter build apk or flutter build appbundle.

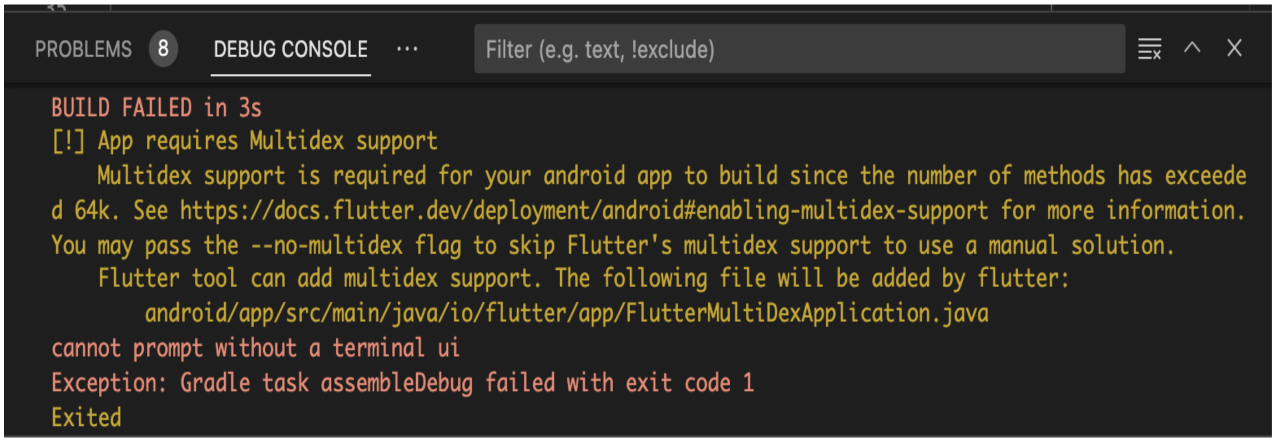
info **Note:** Obfuscation and minification can considerably extend compile time of the Android application.

Enabling multidex support

When writing large apps or making use of large plugins, you might encounter Android's dex limit of 64k methods when targeting a minimum API of 20 or below. This might also be encountered when running debug versions of your app using `flutter run` that does not have shrinking enabled.

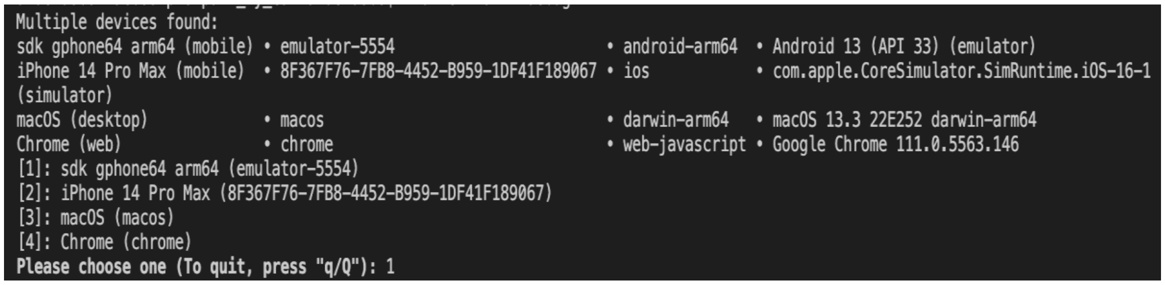
Flutter tool supports easily enabling multidex. The simplest way is to opt into multidex support when prompted. The tool detects multidex build errors and asks before making changes to your Android project. Opting in allows Flutter to automatically depend on `androidx.multidex:multidex` and use a generated `FlutterMultiDexApplication` as the project's application.

When you try to build and run your app with the **Run** and **Debug** options in your IDE, your build might fail with the following message:



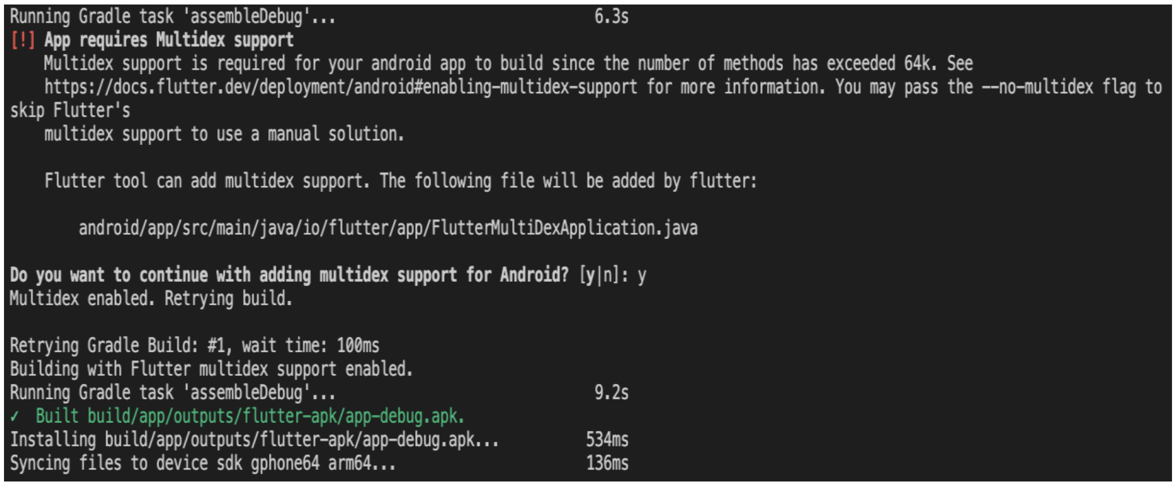
```
BUILD FAILED in 3s
[!] App requires Multidex support
    Multidex support is required for your android app to build since the number of methods has exceede
d 64k. See https://docs.flutter.dev/deployment/android#enabling-multidex-support for more information.
You may pass the --no-multidex flag to skip Flutter's multidex support to use a manual solution.
    Flutter tool can add multidex support. The following file will be added by flutter:
        android/app/src/main/java/io/flutter/app/FlutterMultiDexApplication.java
cannot prompt without a terminal ui
Exception: Gradle task assembleDebug failed with exit code 1
Exited
```

To enable multidex from the command line, run `flutter run --debug` and select an Android device:



```
Multiple devices found:
sdk gphone64 arm64 (mobile) • emulator-5554                        • android-arm64  • Android 13 (API 33) (emulator)
iPhone 14 Pro Max (mobile)  • 8F367F76-7FB8-4452-B959-1DF41F189067 • ios            • com.apple.CoreSimulator.SimRuntime.iOS-16-1
(simulator)
macOS (desktop)             • macos                                • darwin-arm64   • macOS 13.3 22E252 darwin-arm64
Chrome (web)                • chrome                               • web-javascript • Google Chrome 111.0.5563.146
[1]: sdk gphone64 arm64 (emulator-5554)
[2]: iPhone 14 Pro Max (8F367F76-7FB8-4452-B959-1DF41F189067)
[3]: macOS (macos)
[4]: Chrome (chrome)
Please choose one (To quit, press "q/Q"): 1
```

When prompted, enter `y`. The Flutter tool enables multidex support and retries the build:



```
Running Gradle task 'assembleDebug'...                          6.3s
[!] App requires Multidex support
    Multidex support is required for your android app to build since the number of methods has exceeded 64k. See
    https://docs.flutter.dev/deployment/android#enabling-multidex-support for more information. You may pass the --no-multidex flag to
skip Flutter's
    multidex support to use a manual solution.

    Flutter tool can add multidex support. The following file will be added by flutter:

        android/app/src/main/java/io/flutter/app/FlutterMultiDexApplication.java

Do you want to continue with adding multidex support for Android? [y|n]: y
Multidex enabled. Retrying build.

Retrying Gradle Build: #1, wait time: 100ms
Building with Flutter multidex support enabled.
Running Gradle task 'assembleDebug'...                          9.2s
✓  Built build/app/outputs/flutter-apk/app-debug.apk.
Installing build/app/outputs/flutter-apk/app-debug.apk...      534ms
Syncing files to device sdk gphone64 arm64...                  136ms
```

info **Note:** Multidex support is natively included when targeting Android SDK 21 or later. However, we don't recommend targeting API 21+ purely to resolve the multidex issue as this might inadvertently exclude users running older devices.

You might also choose to manually support multidex by following Android's guides and modifying your project's Android directory configuration. A multidex keep file must be specified to include:

content_copy

```
io/flutter/embedding/engine/loader/FlutterLoader.class
io/flutter/util/PathUtils.class
```

Also, include any other classes used in app startup. For more detailed guidance on adding multidex support manually, check out the official Android documentation.

Reviewing the app manifest

Review the default App Manifest file, AndroidManifest.xml. This file is located in [project]/android/app/src/main. Verify the following values:

application

Edit the android:label in the application tag to reflect the final name of the app.

uses-permission

Add the android.permission.INTERNET permission if your application code needs Internet access. The standard template doesn't include this tag but allows Internet access during development to enable communication between Flutter tools and a running app.

Reviewing the Gradle build configuration

Review the default Gradle build file (build.gradle, located in [project]/android/app), to verify that the values are correct.

Under the defaultConfig block
applicationId

Specify the final, unique application ID.

minSdkVersion

Specify the minimum API level on which you designed the app to run. Defaults to flutter.minSdkVersion.

targetSdkVersion

Specify the target API level on which on which you designed the app to run. Defaults to flutter.targetSdkVersion.

versionCode

A positive integer used as an internal version number. This number is used only to determine whether one version is more recent than another, with higher numbers indicating more recent versions. This version isn't shown to users.

versionName

A string used as the version number shown to users. This setting can be specified as a raw string or as a reference to a string resource.

buildToolsVersion

The Gradle plugin specifies the default version of the build tools that your project uses. You can use this option to specify a different version of the build tools.

Under the android block
compileSdkVersion

Specify the API level Gradle should use to compile your app. Defaults to flutter.compileSdkVersion.

For more information, check out the module-level build section in the Gradle build file.

Building the app for release

You have two possible release formats when publishing to the Play Store.

- App bundle (preferred)

- APK

info **Note:** The Google Play Store prefers the app bundle format. For more information, check out About Android App Bundles.

Build an app bundle

This section describes how to build a release app bundle. If you completed the signing steps, the app bundle will be signed. At this point, you might consider obfuscating your Dart code to make it more difficult to reverse engineer. Obfuscating your code involves adding a couple flags to your build command, and maintaining additional files to de-obfuscate stack traces.

From the command line:

1. Enter `cd [project]`
2. Run `flutter build appbundle`
   (Running `flutter build` defaults to a release build.)

The release bundle for your app is created at `[project]/build/app/outputs/bundle/release/app.aab`.

By default, the app bundle contains your Dart code and the Flutter runtime compiled for armeabi-v7a (ARM 32-bit), arm64-v8a (ARM 64-bit), and x86-64 (x86 64-bit).

Test the app bundle

An app bundle can be tested in multiple ways. This section describes two.

Offline using the bundle tool

1. If you haven't done so already, download `bundletool` from the GitHub repository.
2. Generate a set of APKs from your app bundle.
3. Deploy the APKs to connected devices.

Online using Google Play

1. Upload your bundle to Google Play to test it. You can use the internal test track, or the alpha or beta channels to test the bundle before releasing it in production.
2. Follow these steps to upload your bundle to the Play Store.

Build an APK

Although app bundles are preferred over APKs, there are stores that don't yet support app bundles. In this case, build a release APK for each target ABI (Application Binary Interface).

If you completed the signing steps, the APK will be signed. At this point, you might consider obfuscating your Dart code to make it more difficult to reverse engineer. Obfuscating your code involves adding a couple flags to your build command.

From the command line:

1. Enter cd [project].
2. Run flutter build apk --split-per-abi. (The flutter build command defaults to --release.)

This command results in three APK files:

- [project]/build/app/outputs/apk/release/app-armeabi-v7a-release.apk
- [project]/build/app/outputs/apk/release/app-arm64-v8a-release.apk
- [project]/build/app/outputs/apk/release/app-x86_64-release.apk

Removing the --split-per-abi flag results in a fat APK that contains your code compiled for *all* the target ABIs. Such APKs are larger in size than their split counterparts, causing the user to download native binaries that are not applicable to their device's architecture.

Install an APK on a device

Follow these steps to install the APK on a connected Android device.

From the command line:

1. Connect your Android device to your computer with a USB cable.
2. Enter cd [project].
3. Run flutter install.

Publishing to the Google Play Store

For detailed instructions on publishing your app to the Google Play Store, check out the Google Play launch documentation.

Updating the app's version number

The default version number of the app is 1.0.0. To update it, navigate to the pubspec.yaml file and update the following line:

```
version: 1.0.0+1
```

The version number is three numbers separated by dots, such as 1.0.0 in the example above, followed by an optional build number such as 1 in the example above, separated by a +.

Both the version and the build number can be overridden in Flutter's build by specifying --build-name and --build-number, respectively.

In Android, build-name is used as versionName while build-number used as versionCode. For more information, check out Version your app in the Android documentation.

When you rebuild the app for Android, any updates in the version number from the pubspec file will update the versionName and versionCode in the local.properties file.

Android release FAQ

Here are some commonly asked questions about deployment for Android apps.

When should I build app bundles versus APKs?

The Google Play Store recommends that you deploy app bundles over APKs because they allow a more efficient delivery of the application to your users. However, if you're distributing your application by means other than the Play Store, an APK might be your only option.

What is a fat APK?

A fat APK is a single APK that contains binaries for multiple ABIs embedded within it. This has the benefit that the single APK runs on multiple architectures and thus has wider compatibility, but it has the drawback that its file size is much larger, causing users to download and store more bytes when installing your application. When building APKs instead of app bundles, it is strongly recommended to build split APKs, as described in build an APK using the --split-per-abi flag.

What are the supported target architectures?

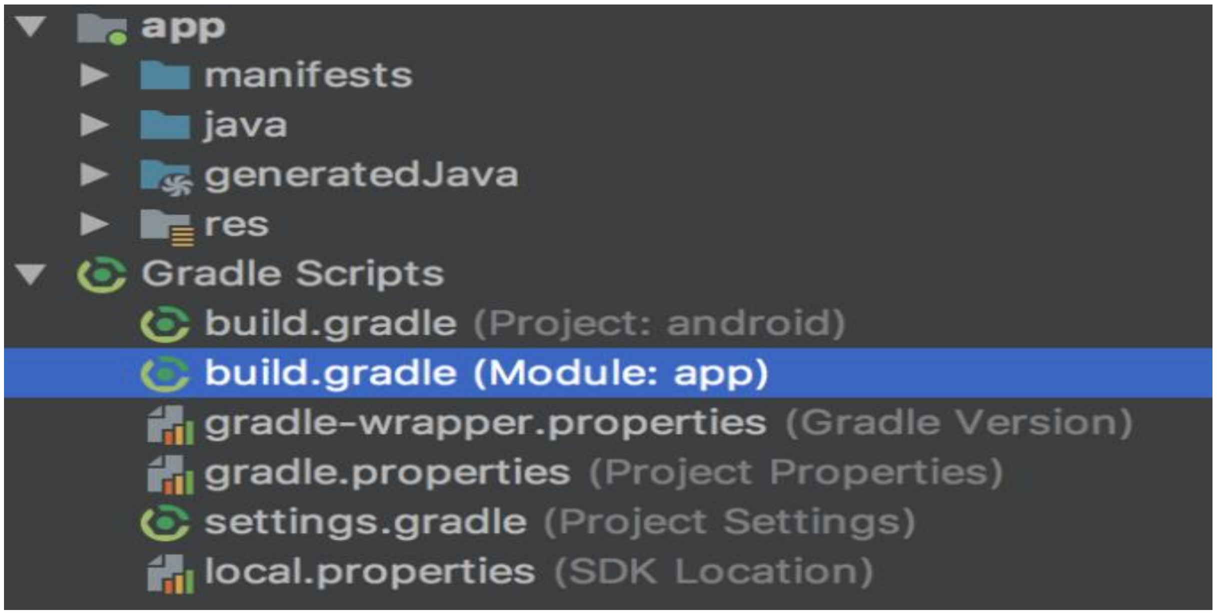
When building your application in release mode, Flutter apps can be compiled for armeabi-v7a (ARM 32-bit), arm64-v8a (ARM 64-bit), and x86-64 (x86 64-bit). Flutter supports building for x86 Android through ARM emulation.

How do I sign the app bundle created by flutter build appbundle?

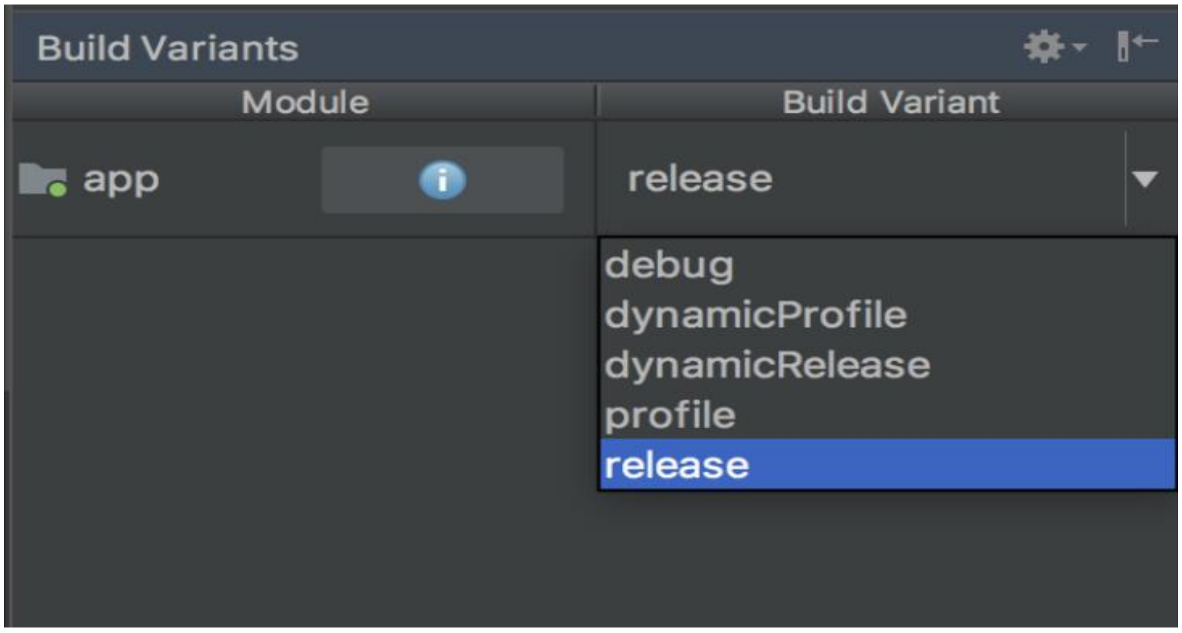
See Signing the app.

How do I build a release from within Android Studio?

In Android Studio, open the existing android/ folder under your app's folder. Then, select **build.gradle (Module: app)** in the project panel:


app
manifests
java
generatedJava
res
Gradle Scripts
build.gradle (Project: android)
build.gradle (Module: app)
gradle-wrapper.properties (Gradle Version)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)

Next, select the build variant. Click **Build > Select Build Variant** in the main menu. Select any of the variants in the **Build Variants** panel (debug is the default):

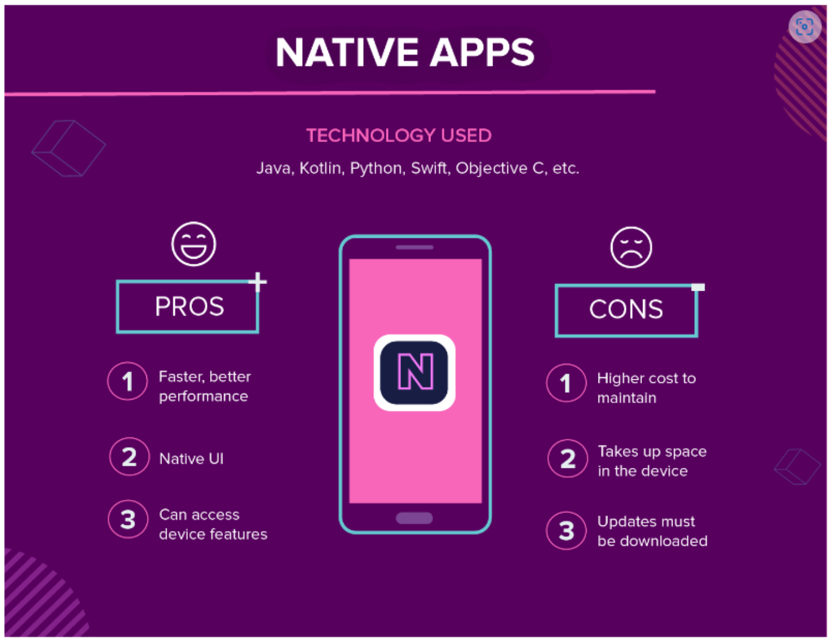


The resulting app bundle or APK files are located in build/app/outputs within your app’s folder.

Lab Programs

Module – 1

Select any two Mobile Apps used in your mobile phone and note the various functionalities and their corresponding layers



NATIVE APPS

TECHNOLOGY USED

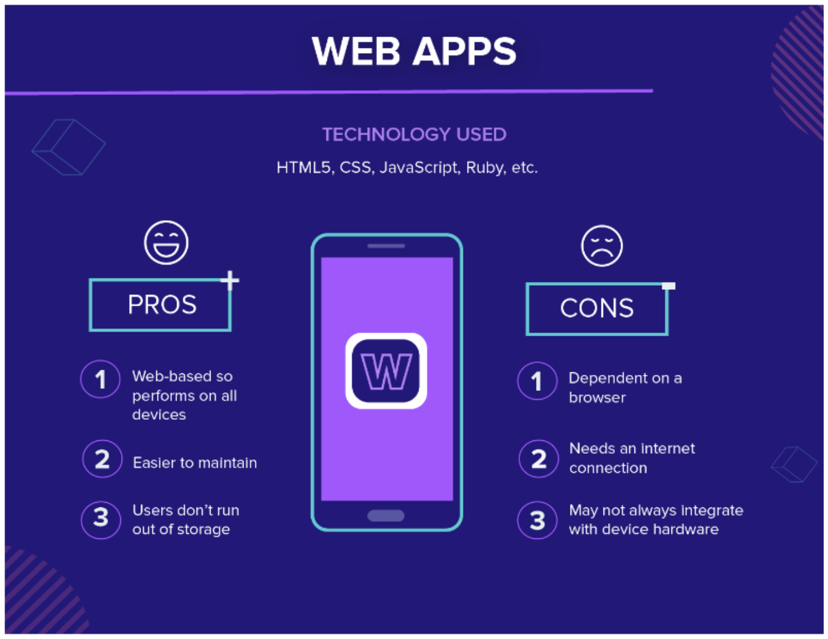
Java, Kotlin, Python, Swift, Objective C, etc.

PROS

1. Faster, better performance
2. Native UI
3. Can access device features

CONS

1. Higher cost to maintain
2. Takes up space in the device
3. Updates must be downloaded



WEB APPS

TECHNOLOGY USED

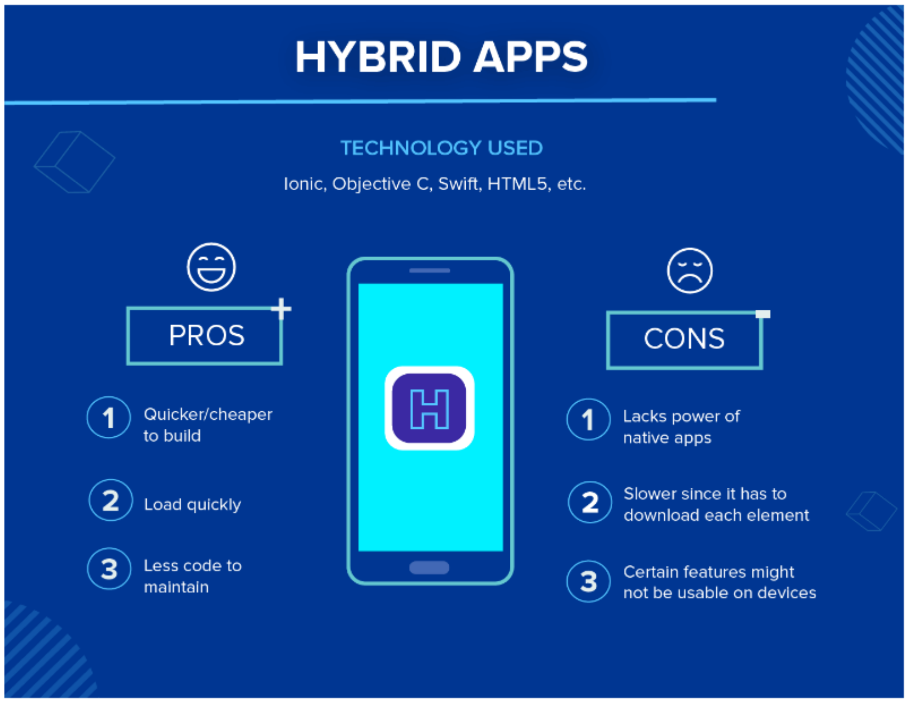
HTML5, CSS, JavaScript, Ruby, etc.

PROS

1. Web-based so performs on all devices
2. Easier to maintain
3. Users don't run out of storage

CONS

1. Dependent on a browser
2. Needs an internet connection
3. May not always integrate with device hardware



HYBRID APPS

TECHNOLOGY USED

Ionic, Objective C, Swift, HTML5, etc.

PROS

1. Quicker/cheaper to build
2. Load quickly
3. Less code to maintain

CONS

1. Lacks power of native apps
2. Slower since it has to download each element
3. Certain features might not be usable on devices

01. Native Apps

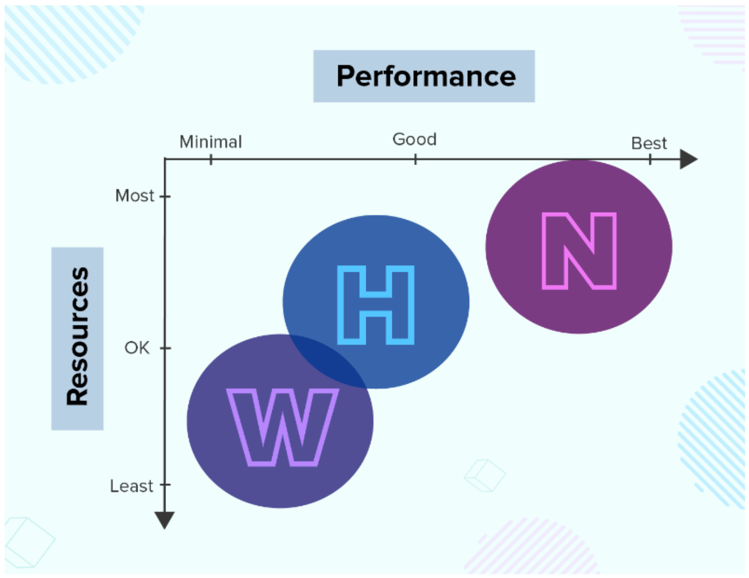
Native apps are built specifically for a mobile device’s operating system (OS). Thus, you can have native Android mobile apps or native iOS apps, not to mention all the other platforms and devices. Because they’re built for just one platform, you cannot mix and match – say, use a Blackberry app on an Android phone or use an iOS app on a Windows phone.

02. Web Apps

Web apps behave similarly to native apps but are accessed via a web browser on your mobile device. They’re not standalone apps in the sense of having to download and install code into your device. They’re actually responsive websites that adapt its user interface to the device the user is on. In fact, when you come across the option to “install” a web app, it often simply bookmarks the website URL on your device.
One kind of web app is the progressive web app (PWA), which is basically a native app running inside a browser.

03. Hybrid Apps

And then there are the hybrid apps. These are web apps that look and feel like native apps. They might have a home screen app icon, responsive design, fast performance, even be able to function offline, but they're really web apps made to look native.



Task:2 Install Android Studio and Configure Latest Android SDKs and Android Virtual Devices

Refer the Module – II

Task-3

Build and Run Hello World Application on the virtual Device and also test the app on your mobile phone

Let us start actual programming with Android Framework. Before you start writing your first example using Android SDK, you have to make sure that you have set-up your Android development environment properly as explained in Android - Environment Set-up tutorial. I also assume that you have a little bit working knowledge with Android studio.

So let us proceed to write a simple Android Application which will print "Hello World!".

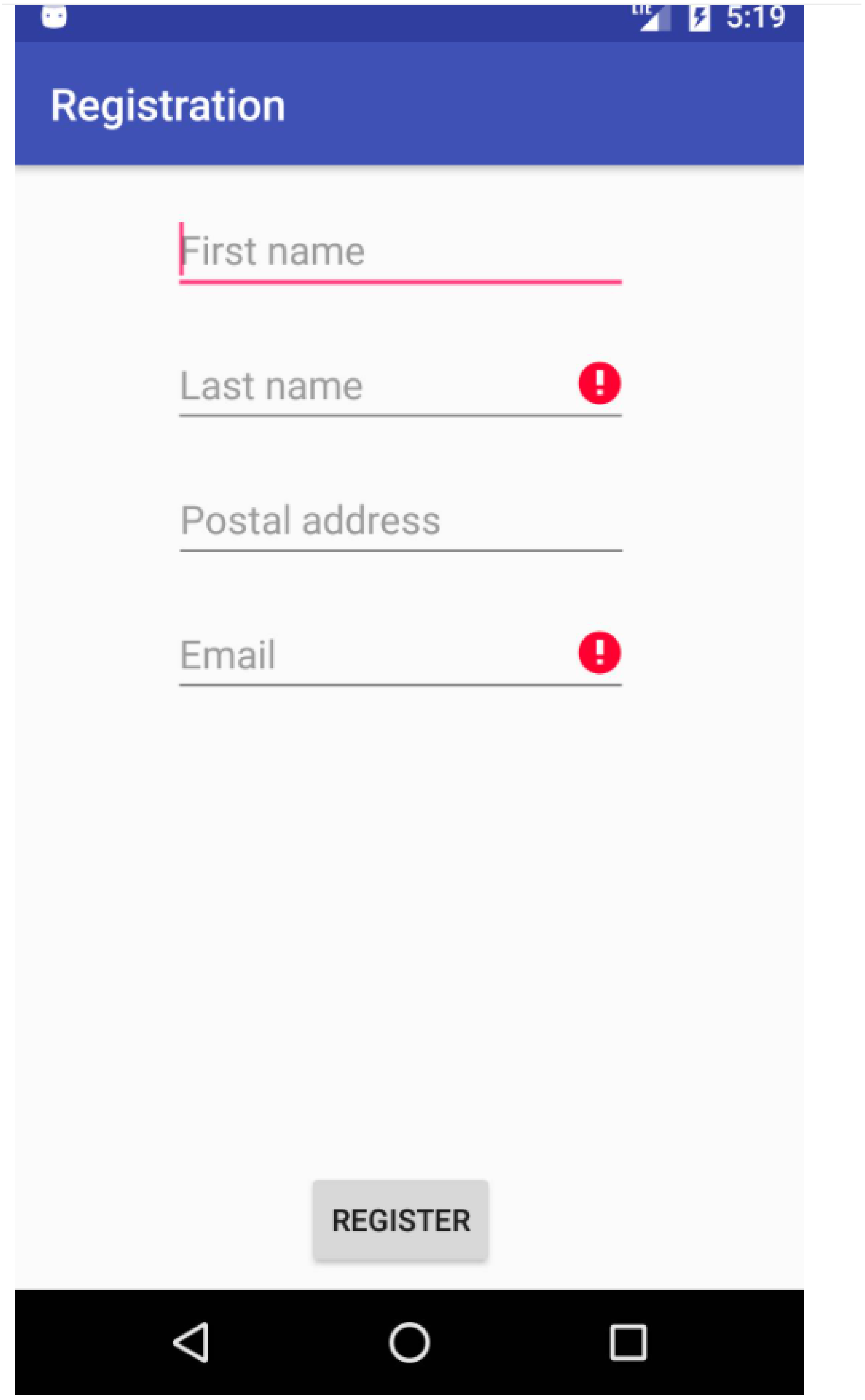
```xml
<resources>
  <string name="app_name">HelloWorld</string>
  <string name="hello_world">Hello world!</string>
  <string name="menu_settings">Settings</string>
  <string name="title_activity_main">MainActivity</string>
</resources>
```

Task- 4

Explore all the UI Controls and design a Student Registration Activity

```java
package com.codebrainer.registration.registration;


import android.support.v7.app.AppCompatActivity;

import android.os.Bundle;

import android.text.TextUtils;

import android.util.Patterns;

import android.view.View;

import android.widget.Button;

import android.widget.EditText;

import android.widget.Toast;


public class MainActivity extends AppCompatActivity {


        EditText firstName;
        EditText lastName;
        EditText address;
        EditText email;
        Button register;

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

                firstName = findViewById(R.id.firstName);
                lastName = findViewById(R.id.lastName);
                address = findViewById(R.id.address);
                email = findViewById(R.id.email);
                register = findViewById(R.id.register);

                register.setOnClickListener(new View.OnClickListener() {
                        @Override
                        public void onClick(View view) {
                                checkDataEntered();
                        }
                });
        }

        boolean isEmail(EditText text) {
        		CharSequence email = text.getText().toString();
        		return (!TextUtils.isEmpty(email) && Patterns.EMAIL_ADDRESS.matcher(email).matches());
        	}


        	boolean isEmpty(EditText text) {
        		CharSequence str = text.getText().toString();
        		return TextUtils.isEmpty(str);
        	}


        	void checkDataEntered() {
        		if (isEmpty(firstName)) {
        			Toast t = Toast.makeText(this, "You must enter first name to register!", Toast.LENGTH_SHORT);
        			t.show();
        		}


        		if (isEmpty(lastName)) {
        			lastName.setError("Last name is required!");
        		}


        		if (isEmail(email) == false) {
        			email.setError("Enter valid email!");
        		}


        	}
```

```
}
```


5:19
Registration
First name
Last name
Postal address
Email
REGISTER

Module – 5

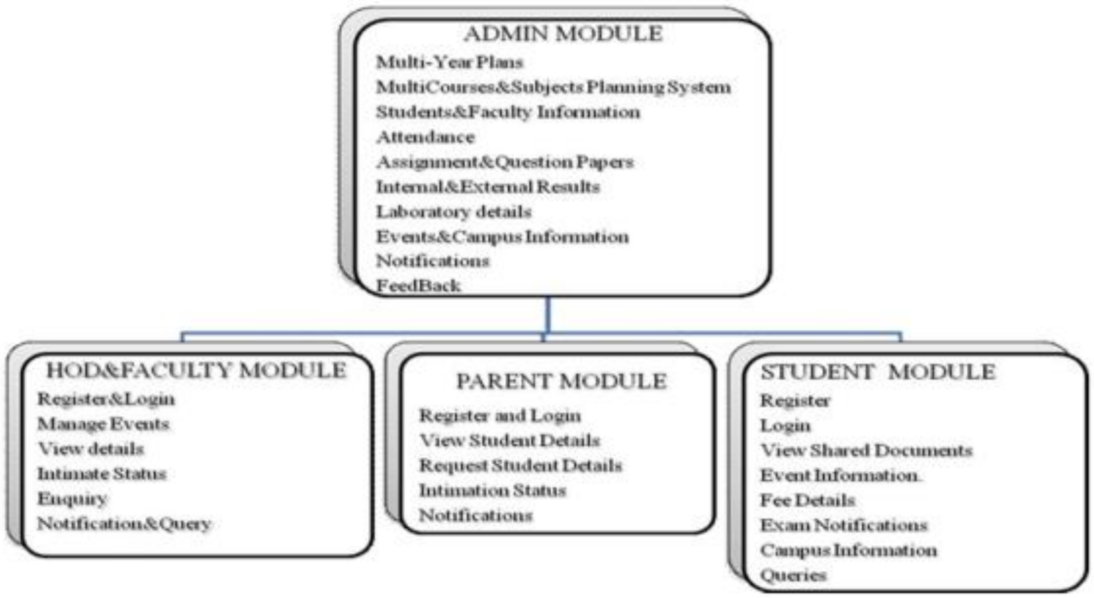
Design the Student Registration Activity using Material Design for Android Components

```
<style name="Theme.MyApp" parent="Theme.AppCompat">

  <!-- Original AppCompat attributes. -->

  <item name="colorPrimary">@color/my_app_primary_color</item>

  <item name="colorSecondary">@color/my_app_secondary_color</item>

  <item name="android:colorBackground">@color/my_app_background_color</item>

  <item name="colorError">@color/my_app_error_color</item>


  <!-- New MaterialComponents attributes. -->

  <item name="colorPrimaryVariant">@color/my_app_primary_variant_color</item>

  <item name="colorSecondaryVariant">@color/my_app_secondary_variant_color</item>

  <item name="colorSurface">@color/my_app_surface_color</item>

  <item name="colorOnPrimary">@color/my_app_color_on_primary</item>

  <item name="colorOnSecondary">@color/my_app_color_on_secondary</item>

  <item name="colorOnBackground">@color/my_app_color_on_background</item>

  <item name="colorOnError">@color/my_app_color_on_error</item>

  <item name="colorOnSurface">@color/my_app_color_on_surface</item>

  <item name="scrimBackground">@color/mtrl_scrim_color</item>

  <item
name="textAppearanceHeadline1">@style/TextAppearance.MaterialComponents.Headline1</it
em>

  <item
name="textAppearanceHeadline2">@style/TextAppearance.MaterialComponents.Headline2</it
em>
```

```
  <item
name="textAppearanceHeadline3">@style/TextAppearance.MaterialComponents.Headline3</it
em>

  <item
name="textAppearanceHeadline4">@style/TextAppearance.MaterialComponents.Headline4</it
em>

<item
name="textAppearanceHeadline5">@style/TextAppearance.MaterialComponents.Headline5</it
em>

  <item
name="textAppearanceHeadline6">@style/TextAppearance.MaterialComponents.Headline6</it
em>

  <item
name="textAppearanceSubtitle1">@style/TextAppearance.MaterialComponents.Subtitle1</item
>

  <item
name="textAppearanceSubtitle2">@style/TextAppearance.MaterialComponents.Subtitle2</item
>

  <item
name="textAppearanceBody1">@style/TextAppearance.MaterialComponents.Body1</item>

  <item
name="textAppearanceBody2">@style/TextAppearance.MaterialComponents.Body2</item>

  <item
name="textAppearanceCaption">@style/TextAppearance.MaterialComponents.Caption</item>

  <item
name="textAppearanceButton">@style/TextAppearance.MaterialComponents.Button</item>

  <item
name="textAppearanceOverline">@style/TextAppearance.MaterialComponents.Overline</item
>
</style>
<com.google.android.material.textfield.TextInputLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/textfield_label">
  <com.google.android.material.textfield.TextInputEditText
      android:layout_width="match_parent"
      android:layout_height="wrap_content"/>
</com.google.android.material.textfield.TextInputLayout>
<com.google.android.material.textfield.TextInputLayout
    style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/textfield_label">
  <com.google.android.material.textfield.TextInputEditText
      android:layout_width="match_parent"
      android:layout_height="wrap_content"/>

</com.google.android.material.textfield.TextInputLayout>
```

Module – 6

Design a complete Student Management Application using Android and provide effective navigation between various Activities



IV.EXPERIMENT AND RESULT

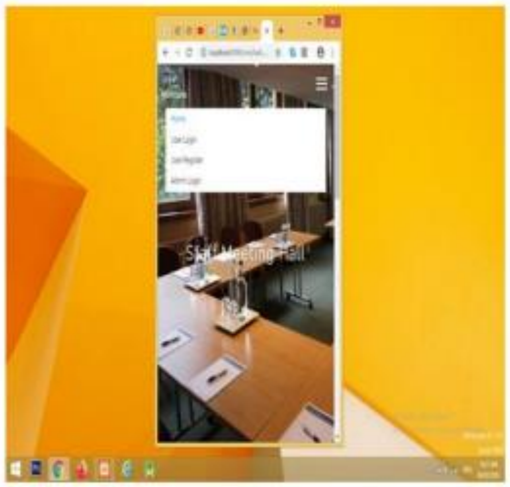


Figure 1: Home page

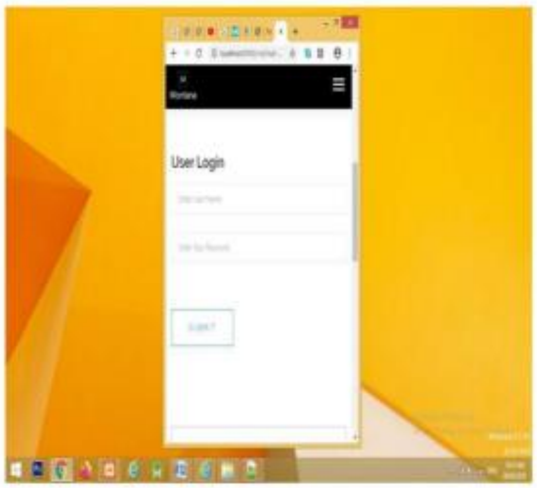


Figure 2: User login

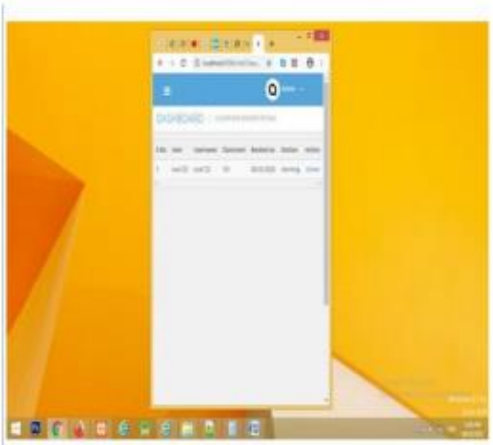
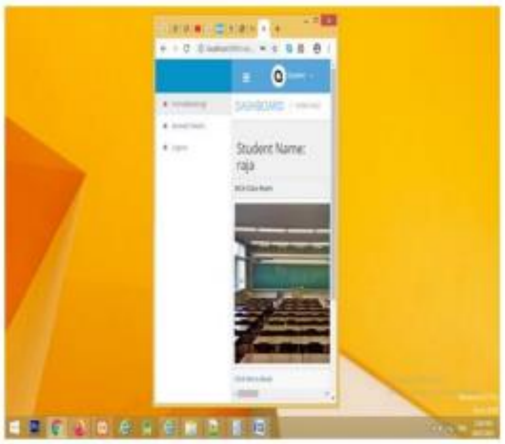
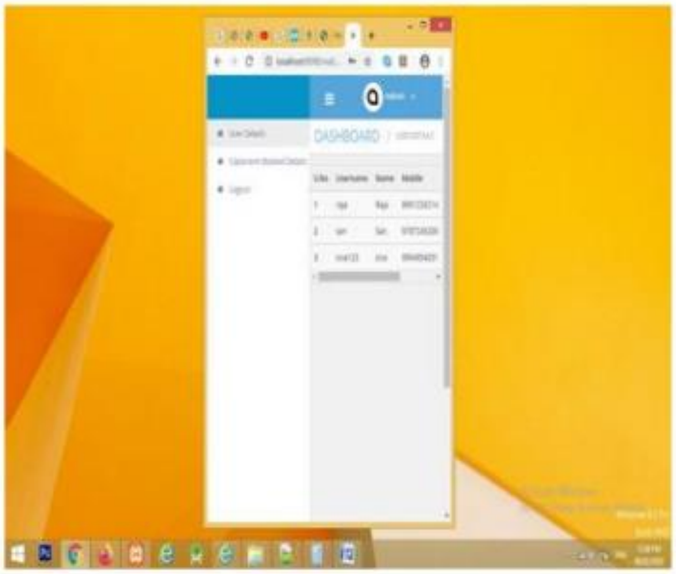


Figure 3: Student registration and booking classroom



Student Name:
raja

Figure 4: Class room booked details



S.No	Username	Name	Mobile
1	raja	Raja	[illegible]
2	sari	Sari	[illegible]
3	[illegible]	[illegible]	[illegible]

Figure 5: Attendance details

Module – 7

Develop an Android Application that stores Student Details into the hosting server and retrieve student details from the server

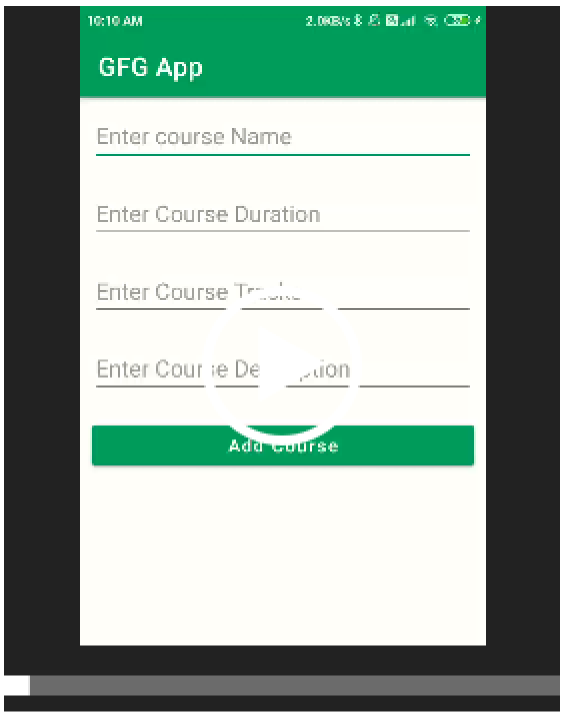
To create an **Android app** that reads student details and displays the information in table format, you can follow these steps:

- Design the layout: Create two XML layout files. One for entering student details with input fields for name, surname, class, gender, hobbies, and marks. Another layout file to display the details in a table format.
- Create a Java class: Create a Java class to handle the logic of the app. This class will handle the button click event and retrieve the values entered by the user.
- Implement onClickListener: Implement the onClickListener interface in your Java class to capture the submit button click event.
- Retrieve input values: Retrieve the values entered by the user from the input fields.
- Pass data to the next activity: Use an intent to pass the student details to the next activity.
- Display details in table format: In the second activity, retrieve the student details from the intent and display them in a table using a ListView or RecyclerView with a custom adapter.
- Run and test the app: Build and run the app on an Android device or emulator to test its functionality.

By following these steps, you can create an Android app that reads student details, passes them to another activity, and displays them in a table format when the submit button is clicked.

```
import android.content.ContentValues;
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

public class DBHandler extends SQLiteOpenHelper {

        // creating a constant variables for our database.
        // below variable is for our database name.
        private static final String DB_NAME = "coursedb";

        // below int is our database version
        private static final int DB_VERSION = 1;

        // below variable is for our table name.
        private static final String TABLE_NAME = "mycourses";

        // below variable is for our id column.
        private static final String ID_COL = "id";

        // below variable is for our course name column
        private static final String NAME_COL = "name";

        // below variable id for our course duration column.
        private static final String DURATION_COL = "duration";

        // below variable for our course description column.
        private static final String DESCRIPTION_COL = "description";
        // below variable is for our course tracks column.
        private static final String TRACKS_COL = "tracks";

        // creating a constructor for our database handler.
        public DBHandler(Context context) {
                super(context, DB_NAME, null, DB_VERSION);
        }

        // below method is for creating a database by running a sqlite query
        @Override
        public void onCreate(SQLiteDatabase db) {
                // on below line we are creating
                // an sqlite query and we are
                // setting our column names
                // along with their data types.
                String query = "CREATE TABLE " + TABLE_NAME + " ("
                                + ID_COL + " INTEGER PRIMARY KEY AUTOINCREMENT,
    "
                                + NAME_COL + " TEXT,"
                                + DURATION_COL + " TEXT,"
                                + DESCRIPTION_COL + " TEXT,"
                                + TRACKS_COL + " TEXT)";

                // at last we are calling a exec sql
        // method to execute above sql query
        db.execSQL(query);
    }
    // this method is use to add new course to our sqlite database.
    public void addNewCourse(String courseName, String courseDuration, String
courseDescription, String courseTracks) {

        // on below line we are creating a variable for
        // our sqlite database and calling writable method
        // as we are writing data in our database.
        SQLiteDatabase db = this.getWritableDatabase();

        // on below line we are creating a
        // variable for content values.
        ContentValues values = new ContentValues();

        // on below line we are passing all values
        // along with its key and value pair.
        values.put(NAME_COL, courseName);
        values.put(DURATION_COL, courseDuration);
        values.put(DESCRIPTION_COL, courseDescription);
        values.put(TRACKS_COL, courseTracks);

        // after adding all values we are passing
        // content values to our table.
```

```
                db.insert(TABLE_NAME, null, values);

                // at last we are closing our
                // database after adding database.
                db.close();
        }

        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
                // this method is called to check if the table exists already.
                db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
                onCreate(db);
        }
}
```


10:10 AM
GFG App
Enter course Name
Enter Course Duration
Enter Course Tracks
Enter Course Description
Add Course