BIT

# Native Application Development with Android

IT6306 - Mobile Application Development

Level III - Semester 6

UCSC

eLC

## Overview

This section of the course will introduce native application development with Java, covering everything from setting up your development environment to building your first application. You will learn how to use Java to create engaging user interfaces, manage data, and integrate with web services.

By the end of this section, you will have a solid foundation in native application development with Java and be able to create your own applications for Android. Whether you are an experienced programmer or new to mobile development, this section will provide you with the skills and knowledge you need to succeed in the exciting world of native application development.

## Intended Learning Outcomes

After completing this section, you'll be able to;

- Demonstrate an understanding of the Java language and its syntax, including variables, data types, operators, control structures, and functions.
- Develop and deploy native mobile applications for Android or iOS platforms using Java, including designing user interfaces, managing data, and integrating with web services.
- Apply object-oriented programming principles to develop modular, maintainable, and extensible applications using Java.
- Analyse and debug code using debugging tools and techniques and optimize application performance by identifying and addressing common issues and bottlenecks.

## List of sub topics

## 4.1. Getting started with Android development

## 4.1.1. Introduction to Android development

### What is Android?

- Mobile operating system based on Linux kernel
- User Interface for touch screens
- Used on over 80% of all smartphones
- Powers devices such as watches, TVs, and cars
- Over 2 Million Android apps in Google Play store
- Highly customizable for devices / by vendors
- Open source

## Android user interaction

- Touch gestures: swiping, tapping, pinching
- Virtual keyboard for characters, numbers, and emoji
- Support for Bluetooth, USB controllers and peripherals

## Android and Sensors

- Sensors can discover user action and respond
- Device contents rotate as needed
- Walking adjusts position on map
- Tilting steers a virtual car or controls a physical toy
- Moving too fast disables game interactions

## Android Software Developer Kit (SDK)

- Development tools (debugger, monitors, editors)
- Libraries (maps, wearables)
- Virtual devices (emulators)
- Documentation
- Sample code

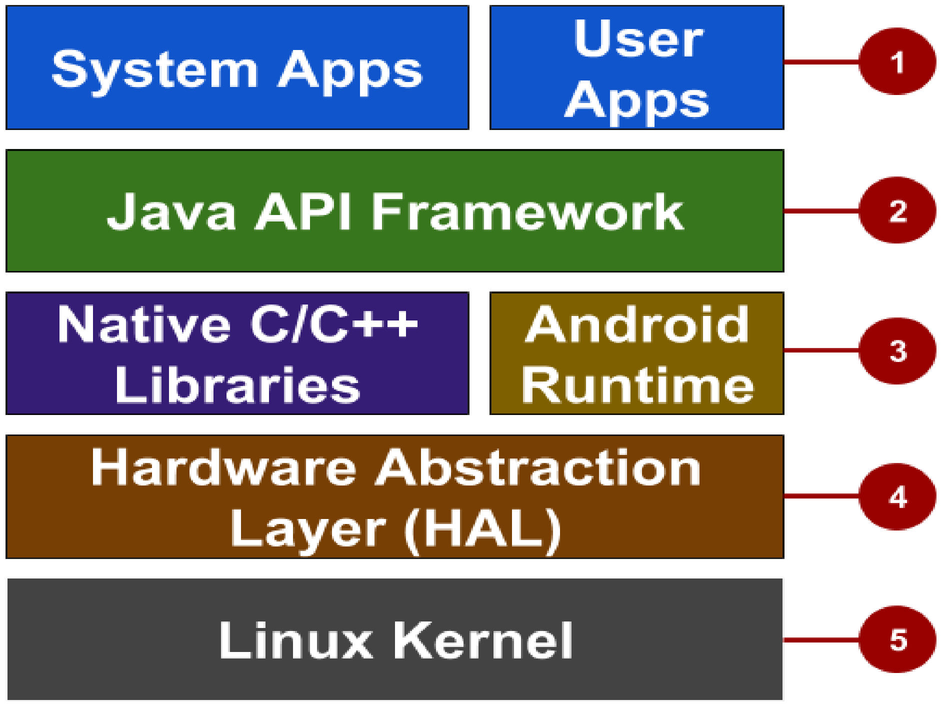
## Publishing

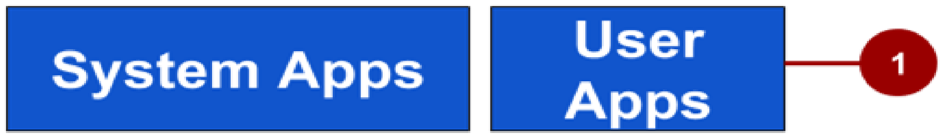
Publish apps through Google Play store:

- Official app store for Android
- Digital distribution service operated by Google

# Android Platform Architecture

1. System and user apps
2. Android OS API in Java framework
3. Expose native APIs; run apps
4. Expose device hardware capabilities
5. Linux Kernel





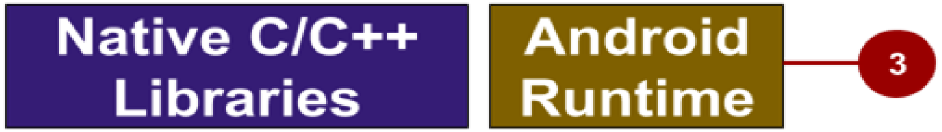
- System apps have no special status
- System apps provide key capabilities to app developers

Example:

Your app can use a system app to deliver a SMS message.

The entire feature-set of the Android OS is available to you through APIs written in the Java language.

- View class hierarchy to create UI screens
- Notification manager
- Activity manager for life cycles and navigation



- Core C/C++ Libraries give access to core native Android system components and services
- Each app runs in its own process with its own instance of the Android Runtime.

Hardware Abstraction Layer (HAL)
4

- Standard interfaces that expose device hardware capabilities as libraries

  Examples: Camera, bluetooth module

## Linux Kernel

5

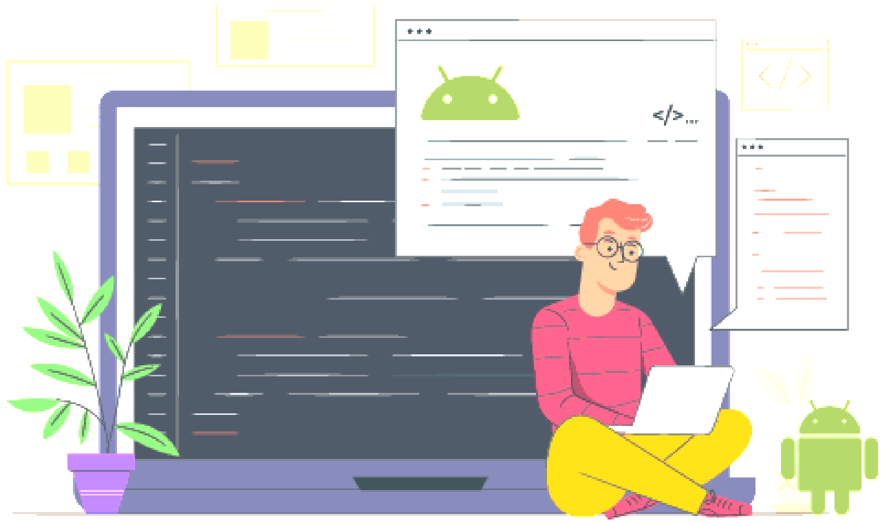
- Threading and low-level memory management
- Security features
- Drivers

## What is an Android app?

- One or more interactive screens
- Written using Java Programming Language and XML
- Uses the Android Software Development Kit (SDK)
- Uses Android libraries and Android Application Framework
- Executed by Android Runtime Virtual machine (ART)

## Challenges of Android development

- **Multiple screen sizes** and **resolutions**
- **Performance**: make your apps responsive and smooth
- **Security**: keep source code and user data safe
- **Compatibility**: run well on older platform versions
- **Marketing**: understand the market and your users
  (Hint: It doesn't have to be expensive, but it can be.)



## 4.1.1. Setting up the tools and environment

## What is Android Studio?

Android Studio is the official Integrated Development Environment (IDE) for Android app development. Android Studio offers even more features that enhance your productivity when building Android apps, such as:

- A flexible Gradle-based build system
- A fast and feature-rich emulator
- A unified environment where you can develop for all Android devices
- Apply Changes to push code and resource changes to your running app without restarting your app
- Code templates and GitHub integration to help you build common app features and import sample code
- Extensive testing tools and frameworks
- Lint tools to catch performance, usability, version compatibility, and other problems
- C++ and NDK support

## Installation Overview

- Can install on Mac, Windows, or Linux
- Download and install Android Studio from

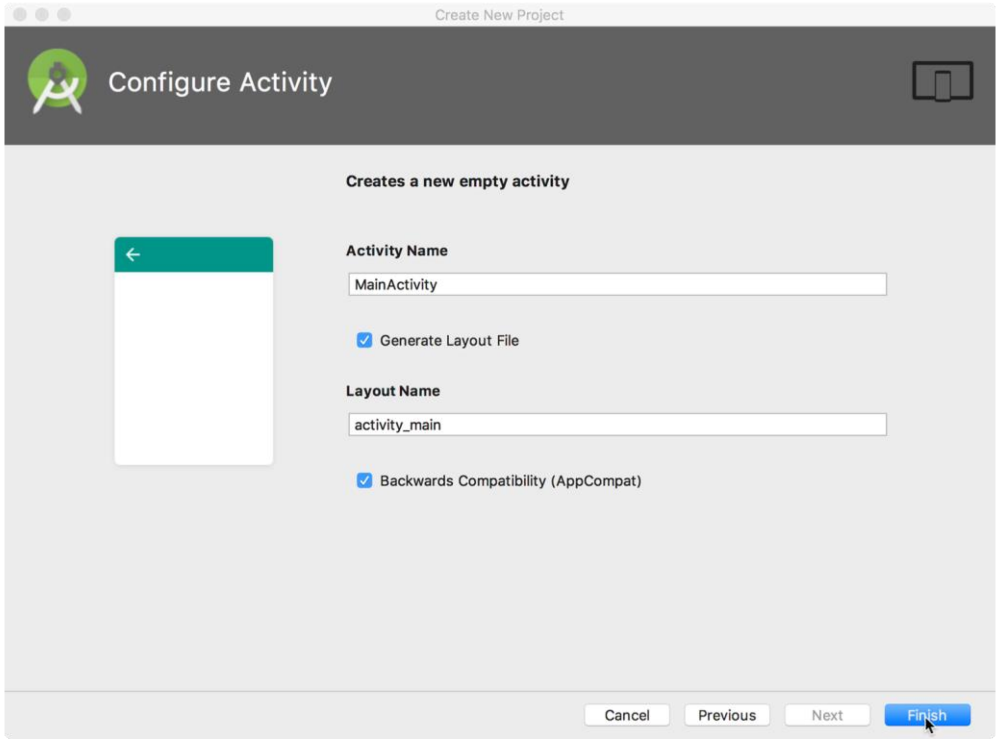
  https://developer.android.com/studio/

ACTIVITY

See Android Studio installation and Hello World Application

## Name an activity

- Good practice:
  - Name main activity `MainActivity`
  - Name layout `activity_main`
- Use AppCompat
- Generating layout file is convenient



Creates a new empty activity

Activity Name: MainActivity

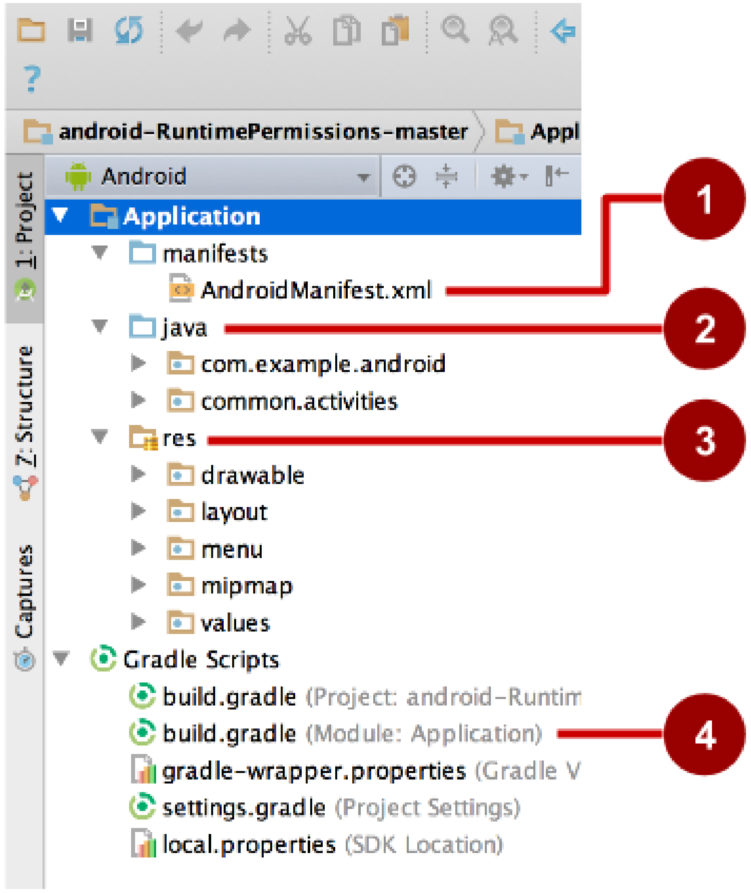
☑ Generate Layout File

Layout Name: activity_main

☑ Backwards Compatibility (AppCompat)

## Project folders

1. **manifests -** Android Manifest file - description of app read by the Android runtime
2. **java -** Java source code packages
3. **res -** Resources (XML) - layout, strings, images, dimensions, colors...
4. **build.gradle -** Gradle build files


android-RuntimePermissions-master
Appl
Android
1: Project
7: Structure
Captures
Application
manifests
AndroidManifest.xml
java
com.example.android
common.activities
res
drawable
layout
menu
mipmap
values
Gradle Scripts
build.gradle (Project: android-Runtim
build.gradle (Module: Application)
gradle-wrapper.properties (Gradle V
settings.gradle (Project Settings)
local.properties (SDK Location)
1
2
3
4

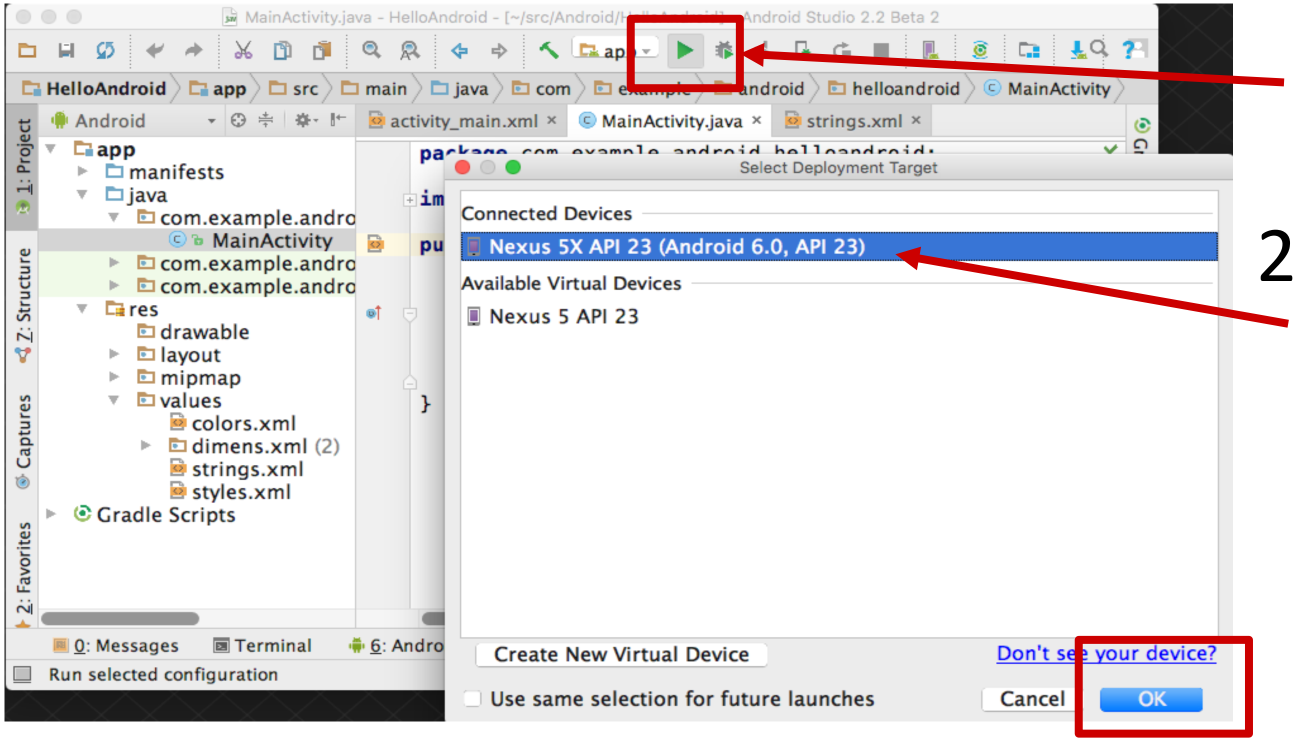
## Gradle build system

- Modern build subsystem in Android Studio
- Three build.gradle:
    - project
    - module
    - settings
- Typically not necessary to know low-level Gradle details
- Learn more about ***gradle*** at https://gradle.org/

ACTIVITY

Create Virtual device to run the program

# Run your app on a device


MainActivity.java - HelloAndroid - [~/src/Android/HelloAndroid] - Android Studio 2.2 Beta 2
HelloAndroid app src main java com example android helloandroid MainActivity
activity_main.xml MainActivity.java strings.xml
app
manifests
java
com.example.andro
MainActivity
res
drawable
layout
mipmap
values
colors.xml
dimens.xml (2)
strings.xml
styles.xml
Gradle Scripts
0: Messages
Terminal
6: Andro
Run selected configuration
Select Deployment Target
Connected Devices
Nexus 5X API 23 (Android 6.0, API 23)
Available Virtual Devices
Nexus 5 API 23
Create New Virtual Device
Don't see your device?
Use same selection for future launches
Cancel
OK

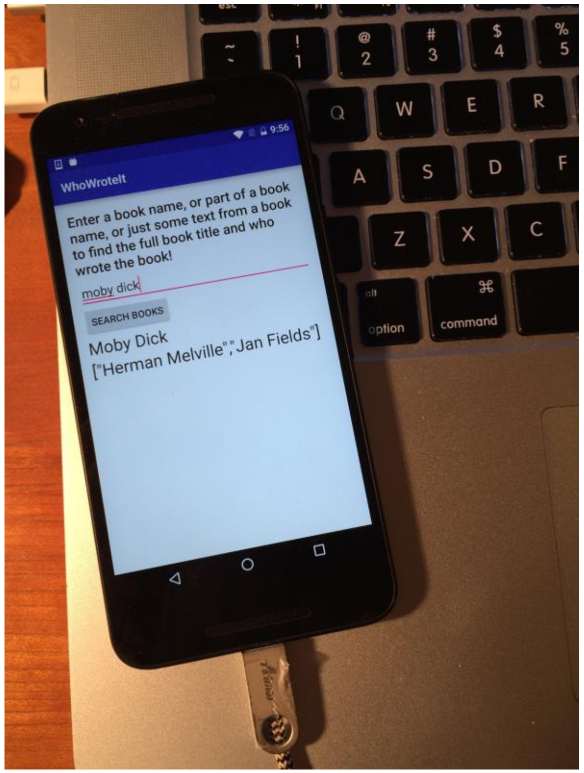
1. Run
2. Select virtual or physical device
3. OK

## Run on a physical device

1. Turn on Developer Options:
   a. **Settings > About phone**
   b. Tap **Build number** seven times
2. Turn on USB Debugging
   a. **Settings > Developer Options > USB Debugging**
3. Connect phone to computer with cable

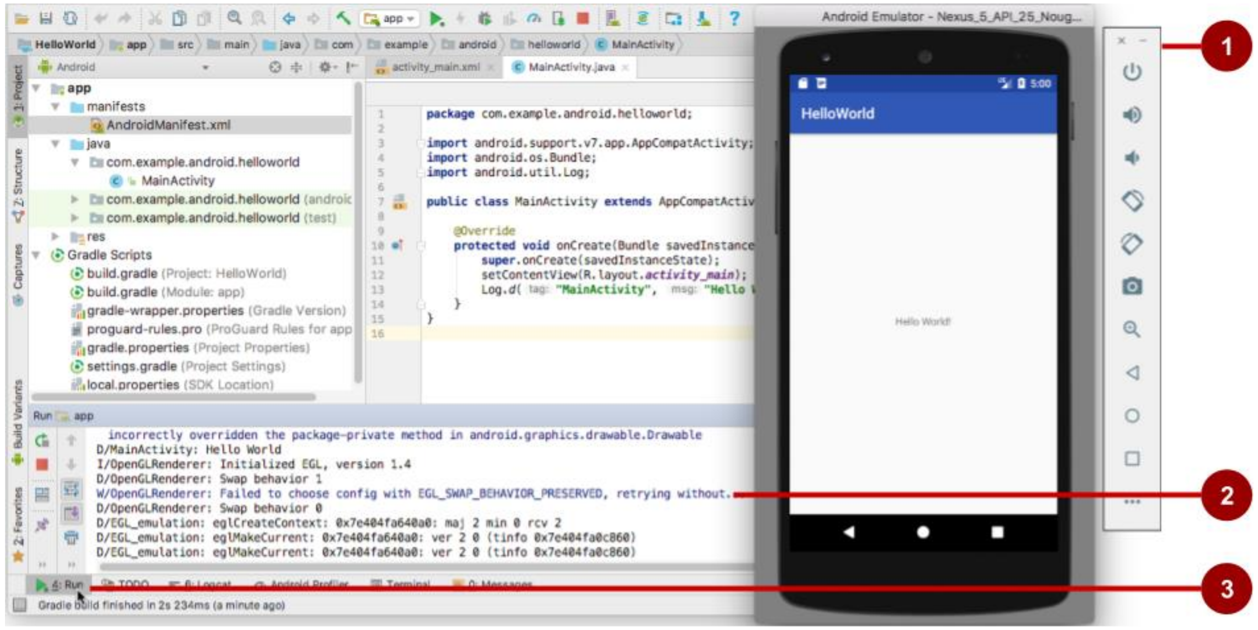
Windows/Linux additional setup:
[Using Hardware Devices]

Windows drivers:
[OEM USB Drivers]


WhoWroteIt
Enter a book name, or part of a book name, or just some text from a book to find the full book title and who wrote the book!
moby dick
SEARCH BOOKS
Moby Dick
["Herman Melville","Jan Fields"]

## Get feedback as your app runs

1. Emulator running the app
2. Run pane
3. **Run** tab to open or close the Run pane



```java
package com.example.android.helloworld;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;

public class MainActivity extends AppCompatActiv

    @Override
    protected void onCreate(Bundle savedInstance
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Log.d( tag: "MainActivity",  msg: "Hello 
    }
}
```

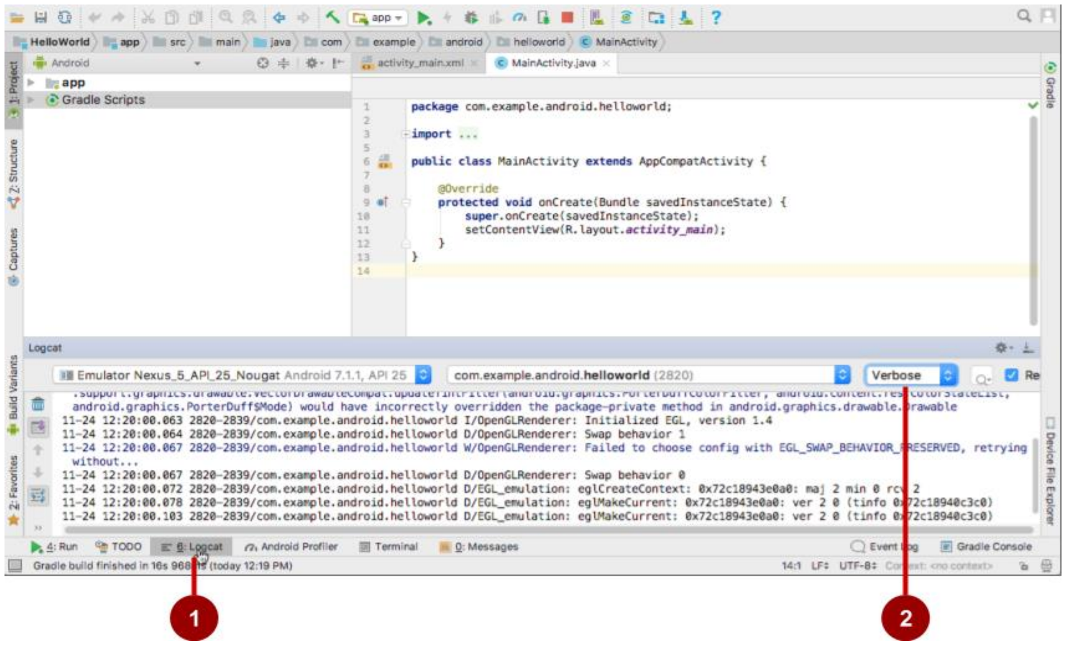
```
  incorrectly overridden the package-private method in android.graphics.drawable.Drawable
D/MainActivity: Hello World
I/OpenGLRenderer: Initialized EGL, version 1.4
D/OpenGLRenderer: Swap behavior 1
W/OpenGLRenderer: Failed to choose config with EGL_SWAP_BEHAVIOR_PRESERVED, retrying without.
D/OpenGLRenderer: Swap behavior 0
D/EGL_emulation: eglCreateContext: 0x7e404fa640a0: maj 2 min 0 rcv 2
D/EGL_emulation: eglMakeCurrent: 0x7e404fa640a0: ver 2 0 (tinfo 0x7e404fa0c860)
D/EGL_emulation: eglMakeCurrent: 0x7e404fa640a0: ver 2 0 (tinfo 0x7e404fa0c860)
```

Gradle build finished in 2s 234ms (a minute ago)

## Adding logging to your app

- As the app runs, the **Logcat** pane shows information
- Add logging statements to your app that will show up in the Logcat pane
- Set filters in **Logcat** pane to see what's important to you
- Search using tags

1. Logcat tab to show Logcat pane
2. Log level menu



```
package com.example.android.helloworld;

import ...

public class MainActivity extends AppCompatActivity {

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

```
  android.graphics.PorterDuff$Mode) would have incorrectly overridden the package-private method in android.graphics.drawable.Drawable
11-24 12:20:00.063 2820-2839/com.example.android.helloworld I/OpenGLRenderer: Initialized EGL, version 1.4
11-24 12:20:00.064 2820-2839/com.example.android.helloworld D/OpenGLRenderer: Swap behavior 1
11-24 12:20:00.067 2820-2839/com.example.android.helloworld W/OpenGLRenderer: Failed to choose config with EGL_SWAP_BEHAVIOR_PRESERVED, retrying
  without...
11-24 12:20:00.067 2820-2839/com.example.android.helloworld D/OpenGLRenderer: Swap behavior 0
11-24 12:20:00.072 2820-2839/com.example.android.helloworld D/EGL_emulation: eglCreateContext: 0x72c18943e0a0: maj 2 min 0 rcv 2
11-24 12:20:00.078 2820-2839/com.example.android.helloworld D/EGL_emulation: eglMakeCurrent: 0x72c18943e0a0: ver 2 0 (tinfo 0x72c18940c3c0)
11-24 12:20:00.103 2820-2839/com.example.android.helloworld D/EGL_emulation: eglMakeCurrent: 0x72c18943e0a0: ver 2 0 (tinfo 0x72c18940c3c0)
```

## 4.2. Working with User Interfaces

### 4.2.1. App manifest and resources

Any app project must have an **AndroidManifest.xml** file at the root of the project source set. The manifest file describes essential information about your app to the Android build tools, the Android operating system, and Google Play.

The manifest file is required to declare the following:

**The components of the app**, which include all activities, services, broadcast receivers, and content providers. Each component must define basic properties such as the name of its Kotlin or Java class. It can also declare capabilities such as which device configurations it can handle, and intent filters that describe how the component can be started.

**The permissions that the app needs** in order to access protected parts of the system or other apps. It also declares any permissions that other apps must have if they want to access content from this app.

**The hardware and software features** the app requires, which affects which devices can install the app from Google Play.

## File features

### App components

For each app component that you create in your app, you must declare a corresponding XML element in the manifest file:

- <activity> for each subclass of Activity.
- <service> for each subclass of Service.
- <receiver> for each subclass of BroadcastReceiver.
- <provider> for each subclass of ContentProvider.

If you subclass any of these components without declaring it in the manifest file, the system cannot start it.

The name of your subclass must be specified with the name attribute, using the full package designation. For example, an Activity subclass can be declared as follows:

```
<manifest ... >
    <application ... >
        <activity android:name="lk.ucsc.bitapp.MainActivity" ... >
        </activity>
    </application>
</manifest>
```

However, if the first character in the name value is a period, the app's namespace (from the module-level **build.gradle** file's **namespace** property) is prefixed to the name. For example, if the namespace is "lk.ucsc.bitapp" the following activity name is resolved to "lk.ucsc.bitapp.MainActivity".

```
<activity android:name=".MainActivity" ... >
```

## Intent filters

App activities, services, and broadcast receivers are activated by intents. An intent is a message defined by an Intent object that describes an action to perform, including the data to be acted upon, the category of component that should perform the action, and other instructions.

When an app issues an intent to the system, the system locates an app component that can handle the intent based on intent filter declarations in each app's manifest file. The system launches an instance of the matching component and passes the Intent object to that component. If more than one app can handle the intent, then the user can select which app to use.

An app component can have any number of intent filters (defined with the <intent-filter> element), each one describing a different capability of that component.

## Icons and labels

Some manifest elements have icon and label attributes for displaying a small icon and a text label, respectively, to users for the corresponding app component.

In every case, the icon and label that are set in a parent element become the default icon and label value for all child elements. For example, the icon and label that are set in the <application> element are the default icon and label for each of the app's components (such as all activities).

The icon and label that are set in a component's <intent-filter> are shown to the user whenever that component is presented as an option to fulfill an intent. By default, this icon is inherited from whichever icon is declared for the parent component, but you might want to change the icon for an intent filter if it provides a unique action that you'd like to better indicate in the chooser dialog.

## Permissions

Android apps must request permission to access sensitive user data (such as contacts and SMS) or certain system features (such as the camera and internet access). Each permission is identified by a unique label. For example, an app that needs to send SMS messages must have the following line in the manifest:

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

Your app can also protect its own components with permissions. It can use any of the permissions that are defined by Android, as listed in android.Manifest.permission, or a permission that's declared in another app. Your app can also define its own permissions. A new permission is declared with the <permission> element.

## Device compatibility

The manifest file is also where you can declare what types of hardware or software features your app requires, and thus, which types of devices your app is compatible with. Google Play Store does not allow your app to be installed on devices that don't provide the features or system version that your app requires.

There are several manifest tags that define which devices your app is compatible with. The following are just a couple of the most common tags.

**<uses-feature>**

The <uses-feature> element allows you to declare hardware and software features your app needs. For example, if your app cannot achieve basic functionality on a device without a compass sensor, you can declare the compass sensor as required with the following manifest tag:

```
<manifest ... >
    <uses-feature android:name="android.hardware.sensor.compass"
                  android:required="true" />
    ...
</manifest>
```

## <uses-sdk>

Each successive platform version often adds new APIs not available in the previous version. To indicate the minimum version with which your app is compatible, your manifest must include the <uses-sdk> tag and its minSdkVersion attribute.

```
android {
    defaultConfig {
        applicationId 'lk.uscs.bitapp'

        // Defines the minimum API level required to run the app.
        minSdkVersion 21

        // Specifies the API level used to test the app.
        targetSdkVersion 33

        ...
    }
}
```

## File conventions

### Elements

Only the <manifest> and <application> elements are required. They each must occur only once. Most of the other elements can occur zero or more times. However, some of them must be present to make the manifest file useful.

All of the values are set through attributes, not as character data within an element.

Elements at the same level are generally not ordered. For example, the <activity>, <provider>, and <service> elements can be placed in any order. There are two key exceptions to this rule:

- An <activity-alias> element must follow the <activity> for which it is an alias.
- The <application> element must be the last element inside the <manifest> element.

## Attributes

Technically, all attributes are optional. However, many attributes must be specified so that an element can accomplish its purpose. For truly optional attributes, the reference documentation indicates the default values.

Except for some attributes of the root <manifest> element, all attribute names begin with an android: prefix. For example, android:alwaysRetainTaskState. Because the prefix is universal, the documentation generally omits it when referring to attributes by name.

## Resource values

Some attributes have values that are displayed to users, such as the title for an activity or your app icon. The value for these attributes might differ based on the user's language or other device configurations (such as to provide a different icon size based on the device's pixel density), so the values should be set from a resource or theme, instead of hard-coded into the manifest file. The actual value can then change based on alternative resources that you provide for different device configurations.

Resources are expressed as values with the following format:

```
"@[package:]type/name"
```

You can omit the package name if the resource is provided by your app (including if it is provided by a library dependency, because library resources are merged into yours). The only other valid package name is **android**, when you want to use a resource from the Android framework.

The type is a type of resource, such as string or drawable, and the name is the name that identifies the specific resource. Here is an example:

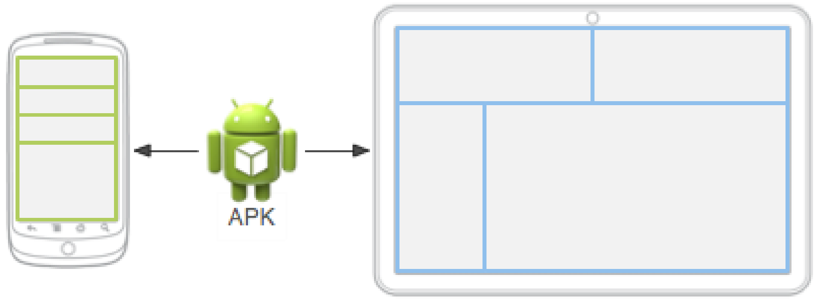
```
<activity android:icon="@drawable/smallPic" ... >
```

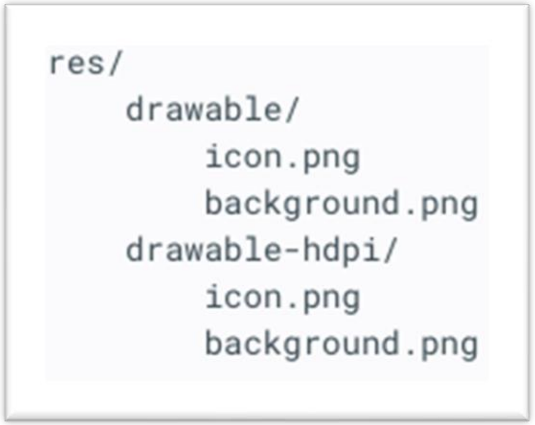
ACTIVITY

All valid elements are in the AndroidManifest.xml file. Reference

## Providing alternative resources

Almost every app should provide alternative resources to support specific device configurations





```
res/
    drawable/
        icon.png
        background.png
    drawable-hdpi/
        icon.png
        background.png
```

To specify configuration-specific alternatives for a set of resources:

- Create a new directory in res/ named in the form <resources_name>-<qualifier>.
    - <resources_name> is the directory name of the corresponding default resources.
    - <qualifier> is a name that specifies an individual configuration for which these resources are to be used.

You can append more than one <qualifier>. Separate each one with a dash.

## Grouping resource types

should place each type of resource in a specific subdirectory of your project's **res/** directory.

```
MyProject/
    src/
        MyActivity.java
    res/
        drawable/
            graphic.png
        layout/
            main.xml
            info.xml
        mipmap/
            icon.png
        values/
            strings.xml
```

ACTIVITY

How to add resources to your project, read [Providing Resources](Providing Resources).

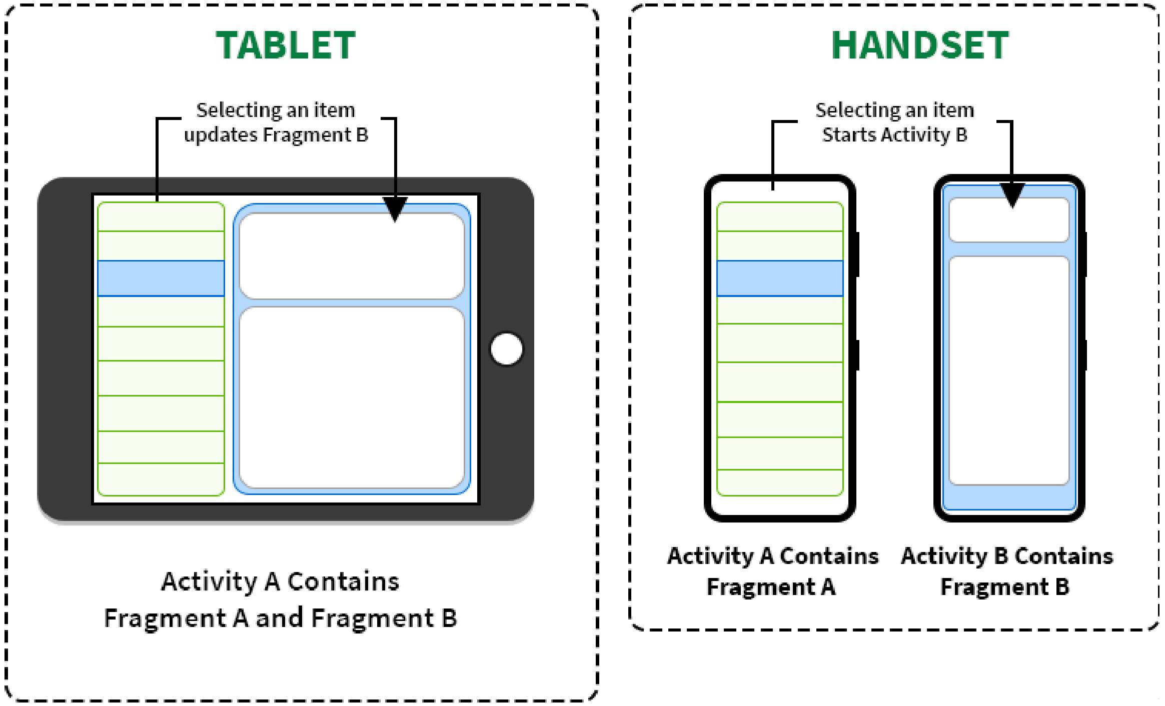
## 4.2.2. Activities and Fragments

An **activity** is a single, focused thing that the user can do. Almost all activities interact with the user, so the Activity class takes care of creating a window for you in which you can place your UI with setContentView(View). While activities are often presented to the user as full-screen windows, they can also be used in other ways: as floating windows (via a theme with R.attr.windowIsFloating set), Multi-Window mode or embedded into other windows. There are two methods almost all subclasses of Activity will implement:

- **onCreate(Bundle)** is where you initialize your activity. Most importantly, here you will usually call **setContentView(int)** with a layout resource defining your UI, and using **findViewById(int)** to retrieve the widgets in that UI that you need to interact with programmatically.
- **onPause()** is where you deal with the user pausing active interaction with the activity. Any changes made by the user should at this point be committed (usually to the ContentProvider holding the data). In this state the activity is still visible on screen.

## Fragments

The FragmentActivity subclass can make use of the Fragment class to better modularize their code, build more sophisticated user interfaces for larger screens, and help scale their application between small and large screens.


TABLET
Selecting an item updates Fragment B
Activity A Contains Fragment A and Fragment B
HANDSET
Selecting an item Starts Activity B
Activity A Contains Fragment A
Activity B Contains Fragment B

## Implement new activities

1. Define layout in XML
2. Define `Activity` Java class
    - extends `AppCompatActivity`
3. Connect `Activity` with Layout
    - Set content view in `onCreate()`
4. Declare `Activity` in the Android manifest

## 1. Define layout in XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
   <TextView
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="Let's Shop for Food!" />
</RelativeLayout>
```

## 2. Define Activity Java class

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

## 3. Connect activity with layout

```
public class MainActivity extends AppCompatActivity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);

setContentView(R.layout.activity_main);
   }
}
```

Resource is layout in this XML file

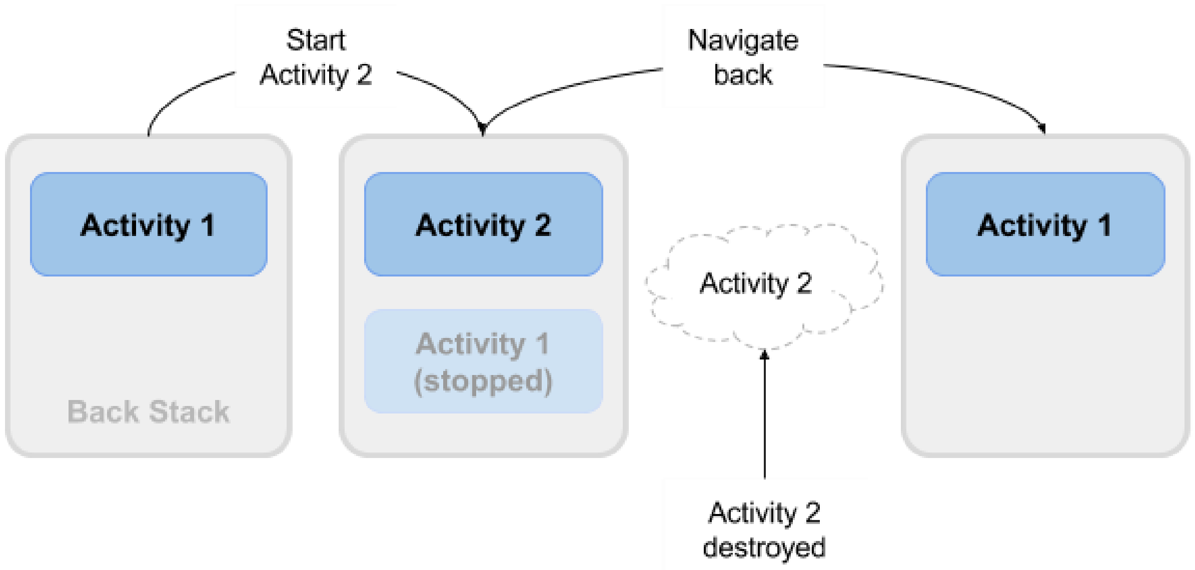
## 4. Declare activity in Android manifest

**`MainActivity`** needs to include `intent-filter` to start from launcher

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

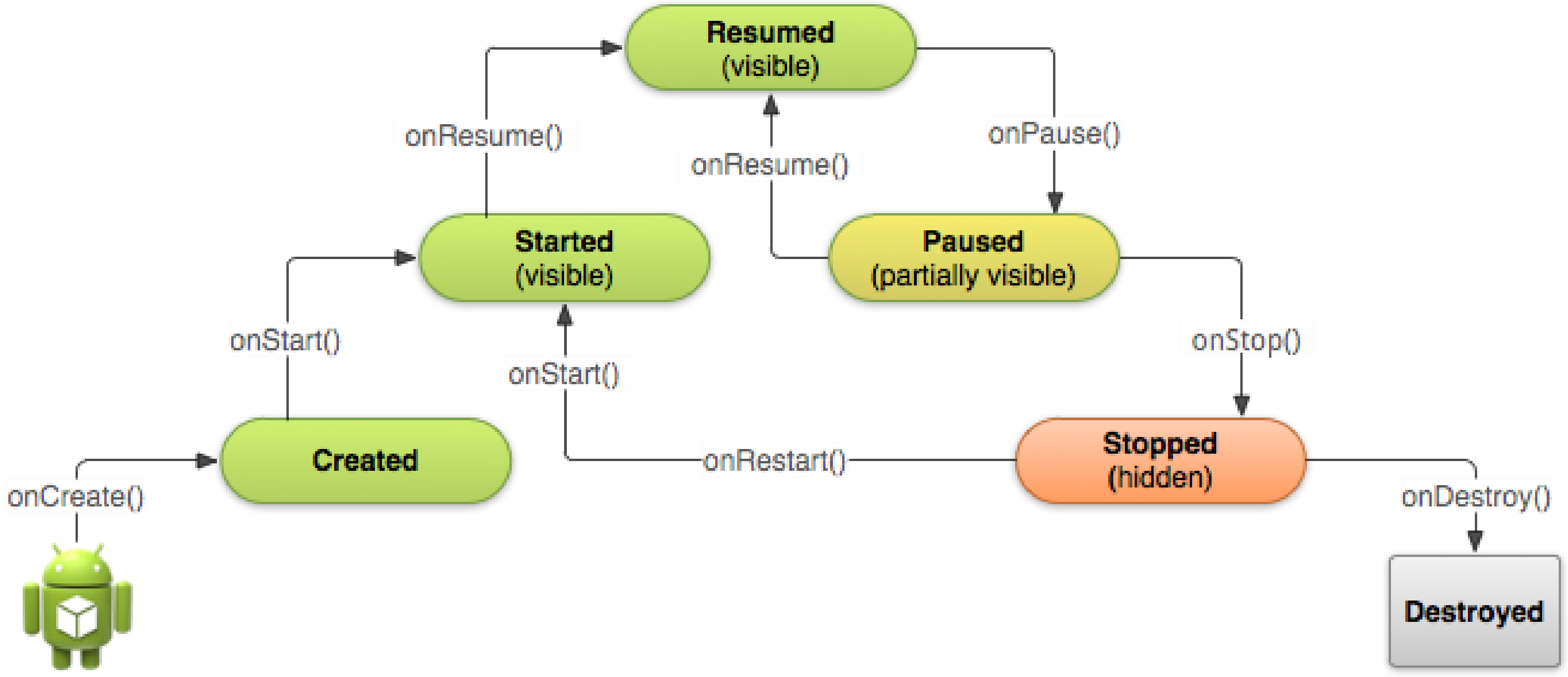
## Activity Lifecycle

- Activities in the system are managed as activity stacks.
- When a new activity is started, it is usually placed on the top of the current stack and becomes the running activity -- the previous activity always remains below it in the stack, and will not come to the foreground again until the new activity exits.
- There can be one or multiple activity stacks visible on screen.



An activity has essentially four states:

- If an activity is in the **foreground** of the screen (at the highest position of the topmost stack), it is active or running. This is usually the activity that the user is currently interacting with.
- If an activity has **lost focus** but is still presented to the user, it is visible. It is possible if a new non-full-sized or transparent activity has focus on top of your activity, another activity has higher position in multi-window mode, or the activity itself is not focusable in current windowing mode. Such activity is completely alive (it maintains all state and member information and remains attached to the window manager).
- If an activity is **completely obscured by another activity**, it is stopped or hidden. It still retains all state and member information, however, it is no longer visible to the user so its window is hidden and it will often be killed by the system when memory is needed elsewhere.
- The system can **drop** the activity from memory by either asking it to finish, or simply killing its process, making it destroyed. When it is displayed again to the user, it must be completely restarted and restored to its previous state.


Resumed (visible)
onResume()
onResume()
onPause()
Started (visible)
Paused (partially visible)
onStart()
onStart()
onStop()
Created
onRestart()
Stopped (hidden)
onCreate()
onDestroy()
Destroyed

## Three key loops

- The **entire lifetime** of an activity happens between the first call to onCreate(Bundle) through to a single final call to onDestroy(). An activity will do all setup of "global" state in onCreate(), and release all remaining resources in onDestroy().

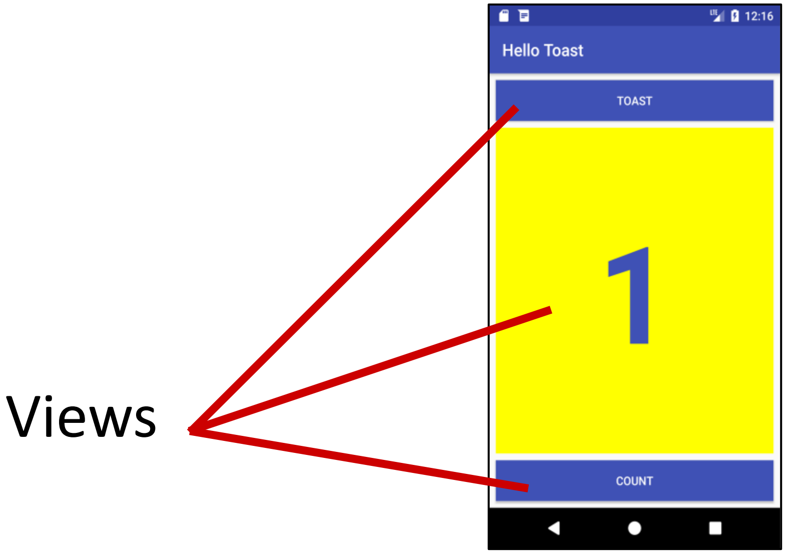
- The **visible lifetime** of an activity happens between a call to onStart() until a corresponding call to onStop(). During this time the user can see the activity on-screen, though it may not be in the foreground and interacting with the user. Between these two methods you can maintain resources that are needed to show the activity to the user.

- The **foreground lifetime** of an activity happens between a call to onResume() until a corresponding call to onPause(). During this time the activity is visible, active and interacting with the user. An activity can frequently go between the resumed and paused states.

## 4.2.3. Layouts, Adapters, Action bar, Dialogs and Notifications

**Views**

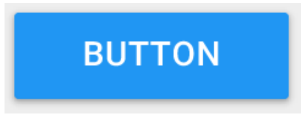
If you look at your mobile device, every user interface element that you see is a **`View`**.

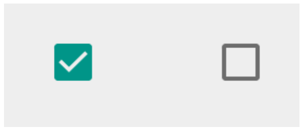


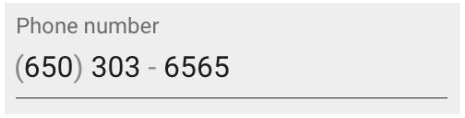
`View` subclasses are basic user interface building blocks

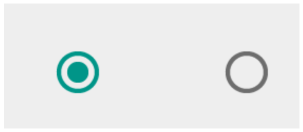
- Display text (`TextView` class), edit text (`EditText` class)
- Buttons (`Button` class), menus, other controls
- Scrollable (`ScrollView`, `RecyclerView`)
- Show images (`ImageView`)
- Group views (`ConstraintLayout` and `LinearLayout`)

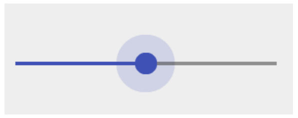
## Examples of view subclasses

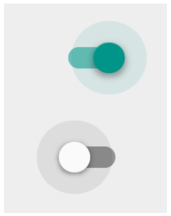
Button 

CheckBox 

EditText 

RadioButton 

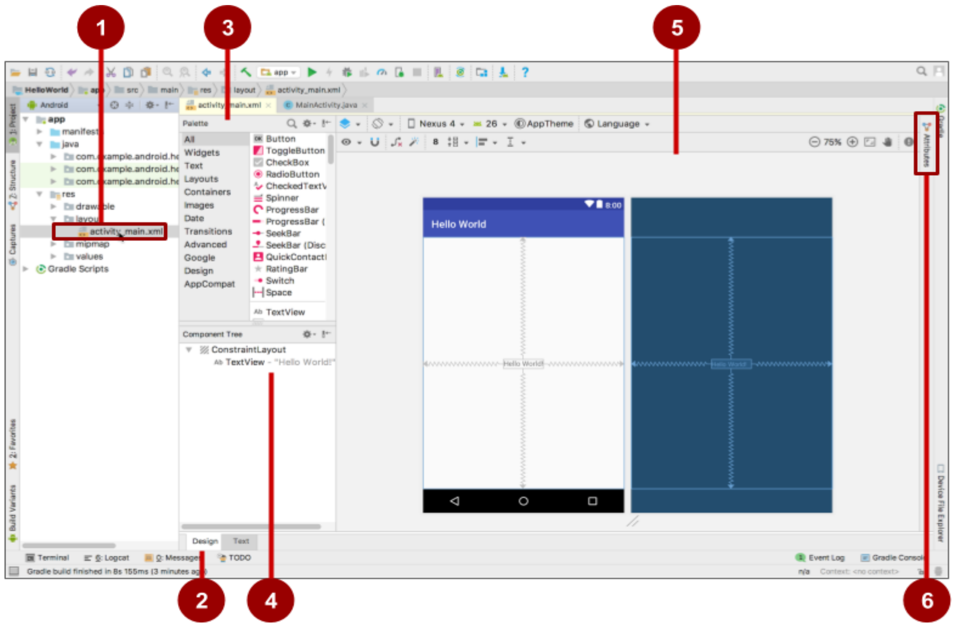
Slider 

Switch 

## View attributes

- Color, dimensions, positioning
- May have focus (e.g., selected to receive user input)
- May be interactive (respond to user clicks)
- May be visible or not
- Relationships to other views

## Android Studio layout editor



1. XML layout file
2. **Design** and **Text** tabs
3. **Palette** pane
4. **Component Tree**
5. Design and blueprint panes
6. **Attributes** tab

## Create views and layouts

- Android Studio layout editor: visual representation of XML
- XML editor
- Java code

## View defined in XML

```
<TextView
       android:id="@+id/show_count"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:background="@color/myBackgroundColor"
       android:text="@string/count_initial_value"
       android:textColor="@color/colorPrimary"
       android:textSize="@dimen/count_text_size"
       android:textStyle="bold"
/>
```

## View attributes in XML

**`android:<property_name>="<property_value>"`**
**Example:** `android:layout_width="match_parent"`

**`android:<property_name>="@<resource_type>/resource_id"`**
**Example:** `android:text="@string/button_label_next"`

**`android:<property_name>="@+id/view_id"`**
**Example:** `android:id="@+id/show_count"`

## Create View in Java code

In an `Activity`:

context

```
TextView myText = new TextView(this);
myText.setText("Display this text!");
```

## What is the context ?

- `Context` is an interface to global information about an application environment
- Get the context:
  ```
  Context context = getApplicationContext();
  ```
- An `Activity` is its own context:
  ```
  TextView myText = new TextView(this);
  ```

## ViewGroup and View hierarchy

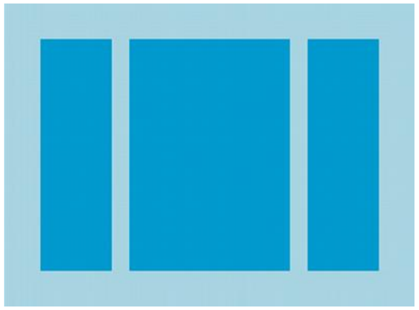
### ViewGroup contains "child" views

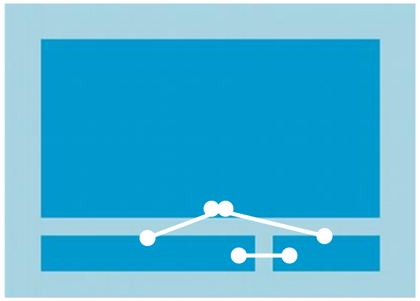
- `ConstraintLayout`: Positions UI elements using constraint connections to other elements and to the layout edges
- `ScrollView`: Contains one element and enables scrolling
- `RecyclerView`: Contains a list of elements and enables scrolling by adding and removing elements dynamically

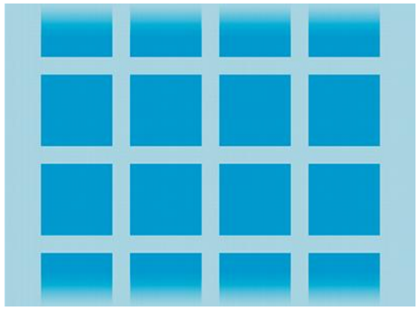
### Layouts

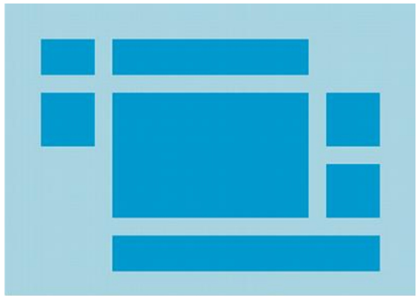
- are specific types of `ViewGroups` (subclasses of `ViewGroup`)
- contain child views
- can be in a row, column, grid, table, absolute

## Common Layout Classes


`LinearLayout`


`ConstraintLayout`


`GridLayout`

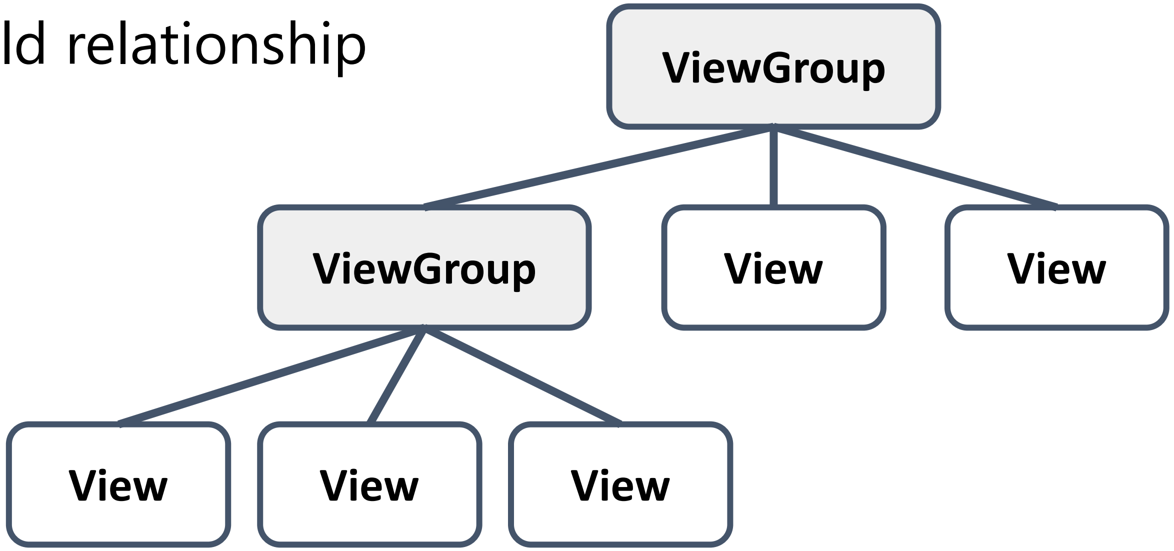

`TableLayout`

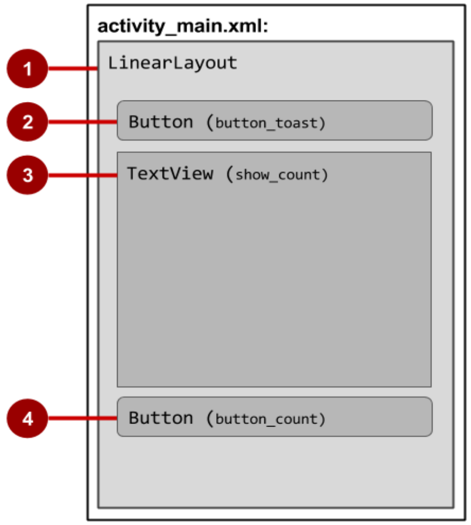
- **`ConstraintLayout`**: Connect views with constraints
- **`LinearLayout`**: Horizontal or vertical row
- **`RelativeLayout`**: Child views relative to each other
- **`TableLayout`**: Rows and columns
- **`FrameLayout`**: Shows one child of a stack of children

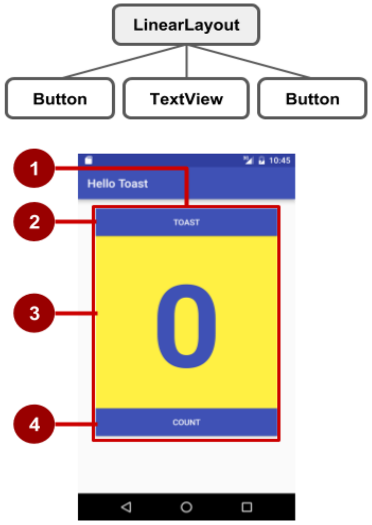
## Class hierarchy vs. Layout hierarchy

- `View` class-hierarchy is standard object-oriented class inheritance
  - For example, `Button` is-a **`TextView`** is-a `View` is-an Object
  - Superclass-subclass relationship

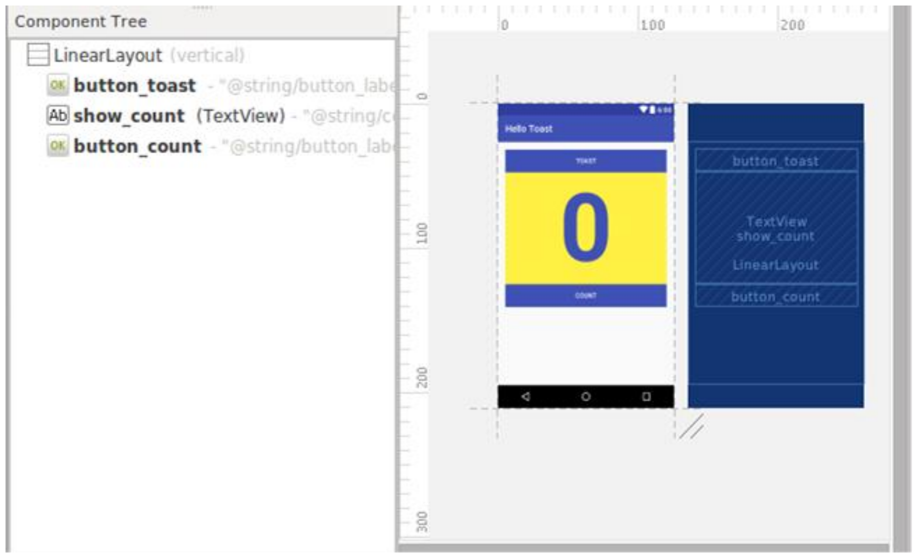
- Layout hierarchy is how views are visually arranged
  - For example, **`LinearLayout`** can contain `Buttons` arranged in a row
  - Parent-child relationship







View hierarchy and screen layout



View hierarchy in the layout editor

## Adapter

**What is an Adapter**

- Helps incompatible interfaces work together
  - Example: Takes data from database `Cursor` and prepares strings to put into a `View`
- Intermediary between data and `View`
- Manages creating, updating, adding, deleting `View` items as underlying data changes

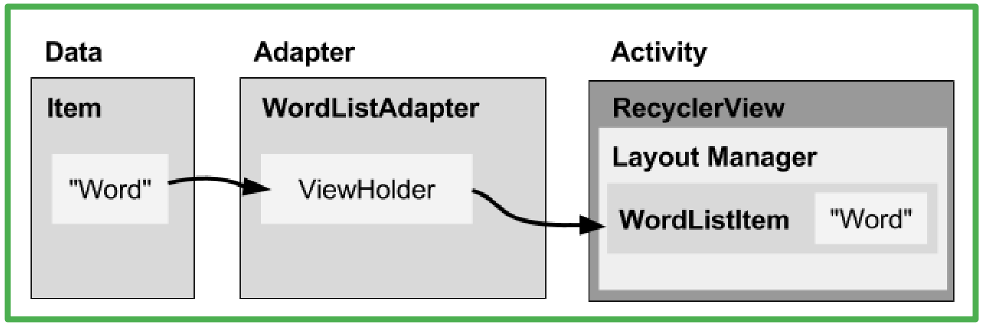
ACTIVITY

There are adapters for each view

Eg- For the recycler view

- `RecyclerView.Adapter`

Refer this material – Link



## Action Bar

A primary toolbar within the activity that may display the activity title, application-level navigation affordances, and other interactive items.

From your activity, you can retrieve an instance of ActionBar by calling getActionBar().

In some cases, the action bar may be overlayed by another bar that enables contextual actions, using an ActionMode. For example, when the user selects one or more items in your activity, you can enable an action mode that offers actions specific to the selected items, with a UI that temporarily replaces the action bar. Although the UI may occupy the same space, the ActionMode APIs are distinct and independent from those for ActionBar.

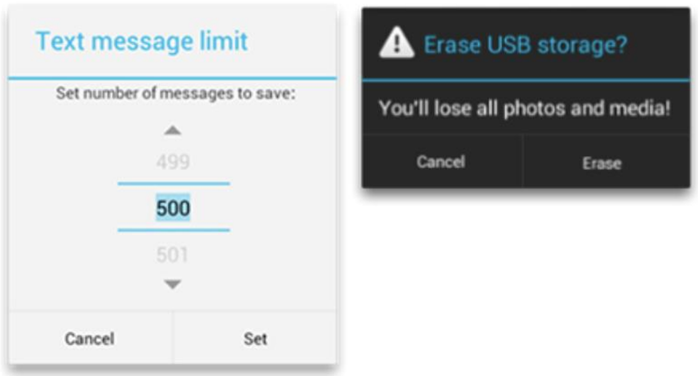
ACTIVITY

Refer. – ActionBar ,

Difference Between AppBar, ActionBar, and Toolbar in Android

## Dialogs

A dialog is a small window that prompts the user to make a decision or enter additional information. A dialog does not fill the screen and is normally used for modal events that require users to take an action before they can proceed.



The Dialog class is the base class for dialogs, but you should avoid instantiating Dialog directly. Instead, use one of the following subclasses:

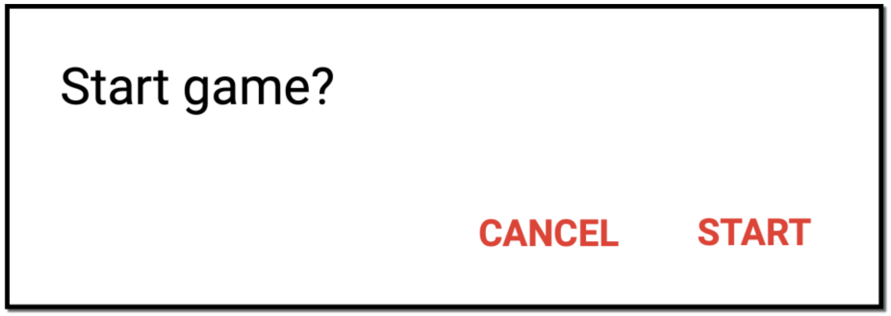
- **AlertDialog**
  A dialog that can show a title, up to three buttons, a list of selectable items, or a custom layout.
- **DatePickerDialog** or **TimePickerDialog**
  A dialog with a pre-defined UI that allows the user to select a date or time.

## Creating a Dialog Fragment

You can accomplish a wide variety of dialog designs—including custom layouts and those described in the Dialogs design guide—by extending DialogFragment and creating an AlertDialog in the onCreateDialog() callback method.

For example, REFER the code for a basic AlertDialog that's managed within a DialogFragment.

when you create an instance of this class and call show() on that object, the dialog appears as shown.


Start game?

## Building an Alert Dialog

The AlertDialog class allows you to build a variety of dialog designs and is often the only dialog class you'll need. As shown in, there are three regions of an alert dialog:

1. Title

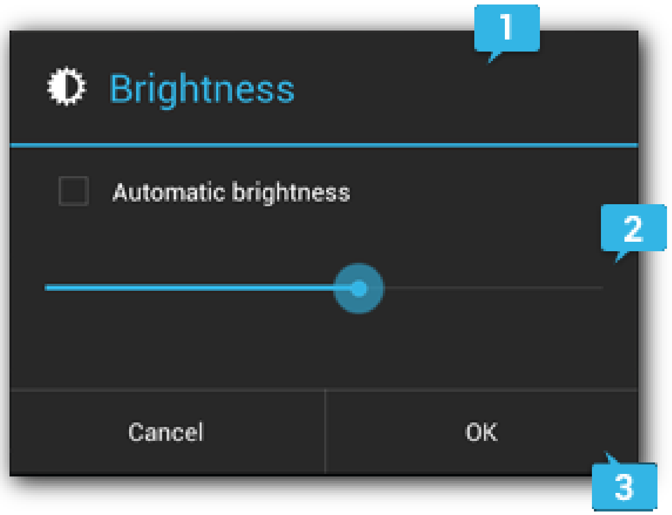
This is optional and should be used only when the content area is occupied by a detailed message, a list, or a custom layout. You don't need a title if you need to state a simple message or question (such as the dialog in figure 1).

2. Content area

This can display a message, a list, or another custom layout.

3. Action buttons

There should be no more than three action buttons in a dialog

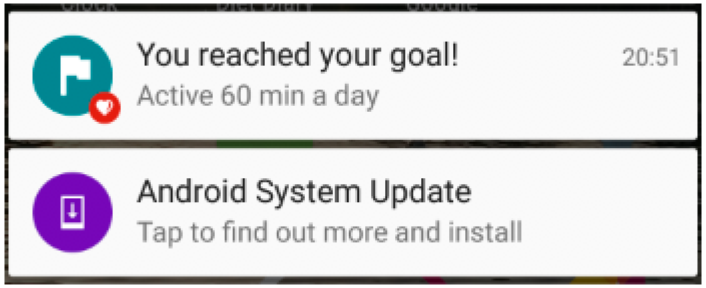


ACTIVITY Read More

## Notifications

Message displayed to user outside regular app UI

- Small icon
- Title
- Detail text



You reached your goal! 20:51
Active 60 min a day

Android System Update
Tap to find out more and install

## How are notifications used?

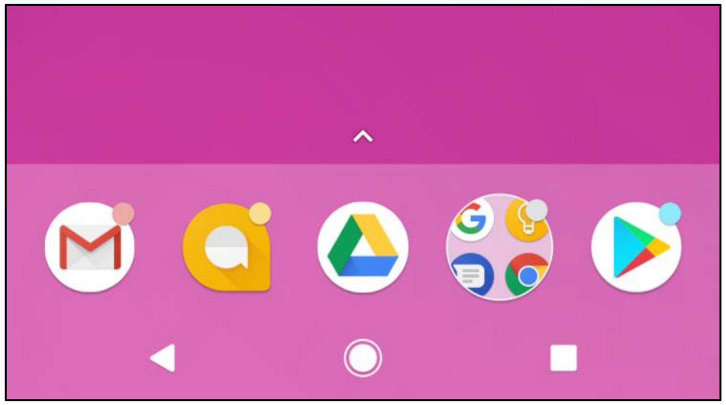
- Android issues a notification that appears as icon on the status bar.
- To see details, user opens the notification drawer.
- User can view notifications any time in the notification drawer.

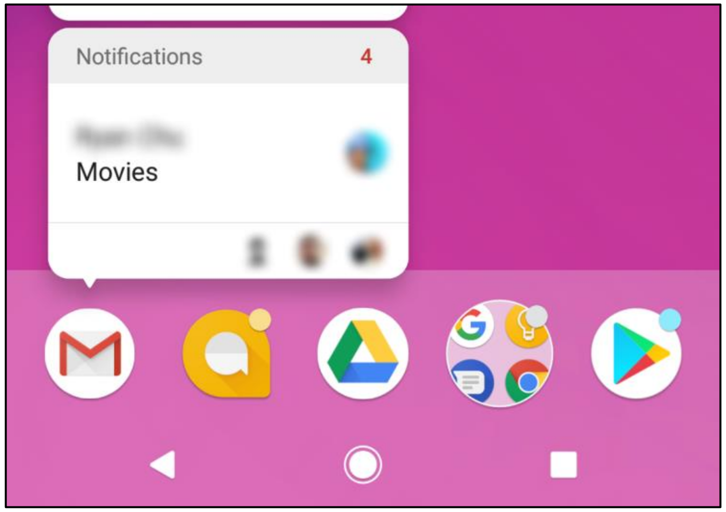
## App icon badge

Available only on the devices running Android 8.0 (API level 26) and higher.

- New notifications are displayed as a colored "badge" (also known as a "notification dot") on the app icon.

- Users can long-press on an app icon to see the notifications for that app. Similar to the notification drawer.





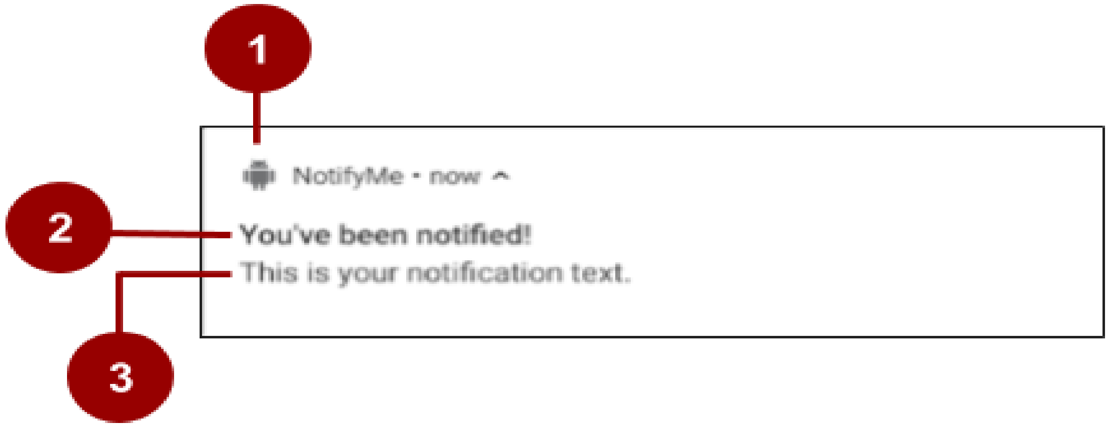
## Creating Notifications

- Notification is created using `NotificationCompat.Builder` class.
- Pass the application context and notification channel ID to the constructor.
- The `NotificationCompat.Builder` constructor takes the notification channel ID, this is only used by Android 8.0 (API level 26) and higher, but this parameter is ignored by the older versions.

```
NotificationCompat.Builder mBuilder =
  new NotificationCompat.Builder(this, CHANNEL_ID)
      .setSmallIcon(R.drawable.android_icon)
      .setContentTitle("You've been notified!")
      .setContentText(”Notification text.");
```

## Setting notification contents

1. A small icon, set by `setSmallIcon()`.
   This is the only content that's required.
1. A title, set by `setContentTitle()`.
2. The body text, set by `setContentText()`. This is the notification message.

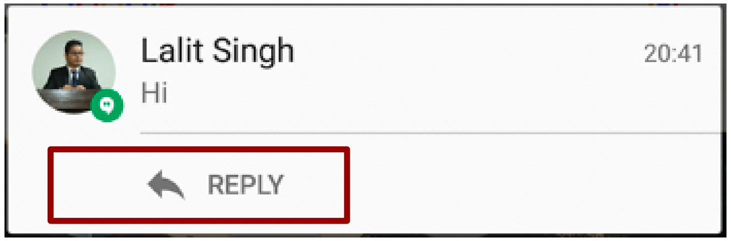


You've been notified!
This is your notification text.

## Tap action and Action buttons

- Every notification must respond when it is tapped, usually launching an Activity in your app.
- Set an content intent using `setContentIntent()` method.
- Pass the `Intent` wrapped in a `PendingIntent` object.
- Action buttons can perform a variety of actions on behalf of your app, such as starting a background task, placing a phone call and so on.
- Starting from Android 7.0 (API level 24) reply to messages directly from notifications.
- To add an action button, pass a `PendingIntent` to the `addAction()` method.

```
.addAction(R.drawable.ic_color_lens_black_24dp,
           "R.string.label",
           notificationPendingIntent);
```


Lalit Singh 20:41
Hi

## Pending intents

- A `PendingIntent` is a description of an intent and target action to perform with it.
- Give a `PendingIntent` to another application to grant it the right to perform the operation you have specified as if the other app was yourself.

To instantiate a PendingIntent, use one of the following methods:

- `PendingIntent.getActivity()`
- `PendingIntent.getBroadcast()`
- `PendingIntent.getService()`

ACTIVITY

Refer this - Link

## Create a Notification channel

Notification channel instance is created using  NotificationChannel constructor.

You must specify:

- An ID  that's unique within your package.
- User visible name of the channel.
- The importance level for the channel.

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    NotificationChannel notificationChannel =
         new NotificationChannel(CHANNEL_ID, "Mascot Notifi.",
         NotificationManager.IMPORTANCE_DEFAULT);
```

## Importance level

- Available in Android 8.0 (API level 26) and higher.
- Sets the intrusion level, like the sound and visibility for all notifications posted in the channel.
- Range from `IMPORTANCE_NONE(0)` to `IMPORTANCE_HIGH(4)`.
- To support earlier versions of Android (Lower than API level 26), set the priority.

## Notification priority

- Determines how the system displays the notification with respect to other notifications, in Android version Lower than API level 26.
- Set using the `setPriority()` method for each notification.
- Range from `PRIORITY_MIN` to `PRIORITY_MAX`.

```
setPriority(NotificationCompat.PRIORITY_HIGH)
```

ACTIVITY

Reference - LINK

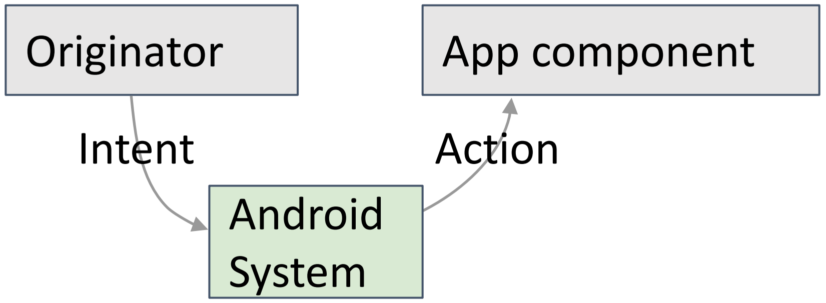
## 4.3. Data and App interaction

### 4.3.1. Intents and Broadcast Receivers

**What is an intent?**

An `Intent` is a description of an operation to be performed.

An `Intent` is an object used to request an action from another app component via the Android system.



## What can intents do?

- Start an `Activity`
  - A button click starts a new `Activity` for text entry
  - Clicking Share opens an app that allows you to post a photo
- Start an `Service`
  - Initiate downloading a file in the background
- Deliver `Broadcast`
  - The system informs everybody that the phone is now charging

## Explicit and implicit intents

### Explicit Intent

- Starts a specific `Activity`
  - Request tea with milk delivered by Nikita
  - Main activity starts the `ViewShoppingCart Activity`

### Implicit Intent

- Asks system to find an `Activity` that can handle this request
  - Find an open store that sells green tea
  - Clicking Share opens a chooser with a list of apps

ACTIVITY

Reference – [Building an intent]

## Three fundemental use cases

### 1. Starting an activity

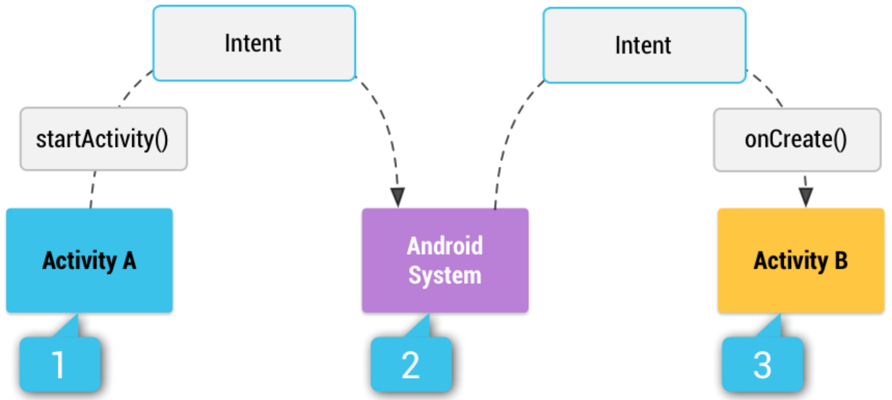
- An Activity represents a single screen in an app. You can start a new instance of an Activity by passing an Intent to startActivity(). The Intent describes the activity to start and carries any necessary data.
- If you want to receive a result from the activity when it finishes, call startActivityForResult(). Your activity receives the result as a separate Intent object in your activity's onActivityResult() callback. For more information, see the Activities guide.

## 2. Starting a service

- A Service is a component that performs operations in the background without a user interface. With Android 5.0 (API level 21) and later, you can start a service with JobScheduler. For more information about JobScheduler, see its API-reference documentation.
- For versions earlier than Android 5.0 (API level 21), you can start a service by using methods of the Service class. You can start a service to perform a one-time operation (such as downloading a file) by passing an Intent to startService(). The Intent describes the service to start and carries any necessary data.
- If the service is designed with a client-server interface, you can bind to the service from another component by passing an Intent to bindService(). For more information, see the Services guide.

## 3. Delivering a broadcast

•A broadcast is a message that any app can receive. The system delivers various broadcasts for system events, such as when the system boots up or the device starts charging. You can deliver a broadcast to other apps by passing an Intent to sendBroadcast() or sendOrderedBroadcast().



*How an implicit intent is delivered through the system to start another activity: **[1]** Activity A creates an Intent with an action description and passes it to startActivity(). **[2]** The Android System searches all apps for an intent filter that matches the intent. When a match is found, **[3]** the system starts the matching activity (Activity B) by invoking its onCreate() method and passing it the Intent.*

## Broadcast Receiver

What is a broadcast receiver?

- Broadcast receivers are app components.
- They register for various system broadcast and or custom broadcast.
- They are notified (via an `Intent`):
  - By the system, when an system event occurs that your app is registered for.
  - By another app, including your own if your app is registered for that custom event.

## Register your broadcast receiver

Broadcast receivers can be registered in two ways:

- Static receivers
  - Registered in your `AndroidManifest.xml`, also called as Manifest-declared receivers.
- Dynamic receivers
  - Registered using app or activities' `context` in your Java files, also called as Context-registered receivers.

## Receiving a system broadcast

- Starting from Android 8.0 (API level 26), static receivers can't receive most of the system broadcasts.
- Use a dynamic receiver to register for these broadcasts.
- If you register for the system broadcasts in the manifest, the Android system won't deliver them to your app.
- A few broadcasts, are excepted from this restriction. See the complete list of implicit broadcast exceptions.

ACTIVITY

Reference – Implementation

## 4.3.2. Preferences and Saving

### What is Shared Preferences?

- Read and write small amounts of primitive data as key/value pairs to a file on the device storage
- SharedPreference class provides APIs for reading, writing, and managing this data
- Save data in onPause()
  restore in onCreate()

## Shared Preferences vs. Saved Instance State

- Persist data across user sessions, even if app is killed and restarted, or device is rebooted
- Data that should be remembered across sessions, such as a user's preferred settings or their game score
- Common use is to store user preferences

- Preserves state data across activity instances in same user session
- Data that should not be remembered across sessions, such as the currently selected tab or current state of activity.
- Common use is to recreate state after the device has been rotated

- Small number of key/value pairs
- Data is private to the application

## Creating Shared Preferences

- Need only one Shared Preferences file per app
- Name it with package name of your app—unique and easy to associate with app
- MODE argument for getSharedPreferences() is for backwards compatibility—use only MODE_PRIVATE to be secure

## Saving Shared Preferences

- SharedPreferences.Editor interface
- Takes care of all file operations
- put methods overwrite if key exists
- apply() saves asynchronously and safely

## Restoring Shared Preferences

- Restore in onCreate() in Activity
- Get methods take two arguments—the key, and the default value if the key cannot be found
- Use default argument so you do not have to test whether the preference exists in the file

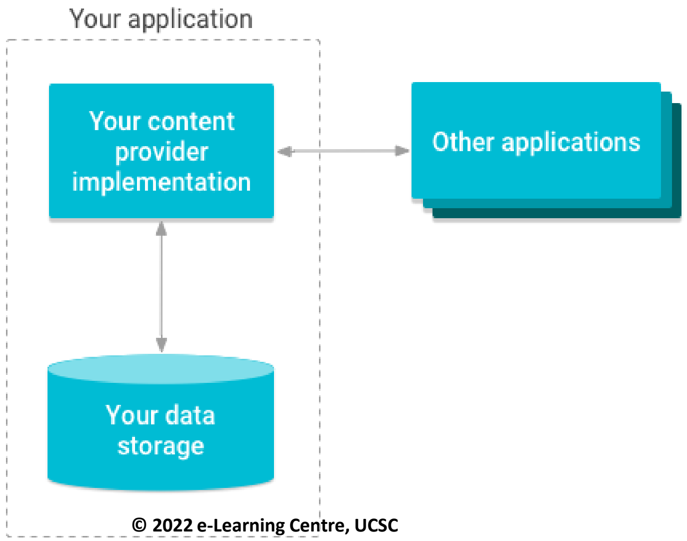
## Clearing

- Call clear() on the SharedPreferences.Editor and apply changes
- You can combine calls to put and clear. However, when you apply(), clear() is always done first, regardless of order!

- References – Preferences , Saving State

## 4.3.3. Content Providers and Services

- Content providers can help an application manage access to data stored by itself, stored by other apps, and provide a way to share data with other apps. They encapsulate the data, and provide mechanisms for defining data security.
- Content providers are the standard interface that connects data in one process with code running in another process. Implementing a content provider has many advantages. Most importantly you can configure a content provider to allow other applications to securely access and modify your app data as illustrated



## The following LINKS describe content providers in more detail:

**Content provider basics**

How to access and update data using an existing content provider.

**Creating a content provider**

How to design and implement your own content provider.

**Calendar provider**

How to access the Calendar provider that is part of the Android platform.

**Contacts provider**

How to access the Contacts provider that is part of the Android platform.

## What is a service?

A `Service` is an application component that can perform long-running operations in the background and does not provide a user interface.

**What are services good for?**

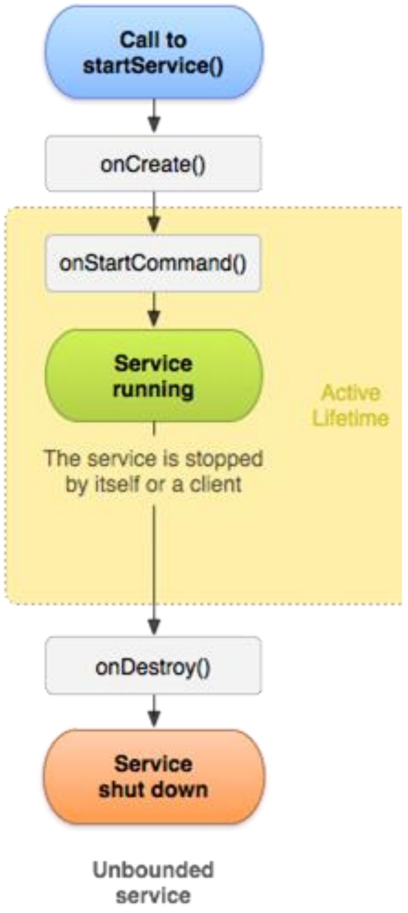
- Network transactions.
- Play music.
- Perform file I/O.
- Interact with a database.

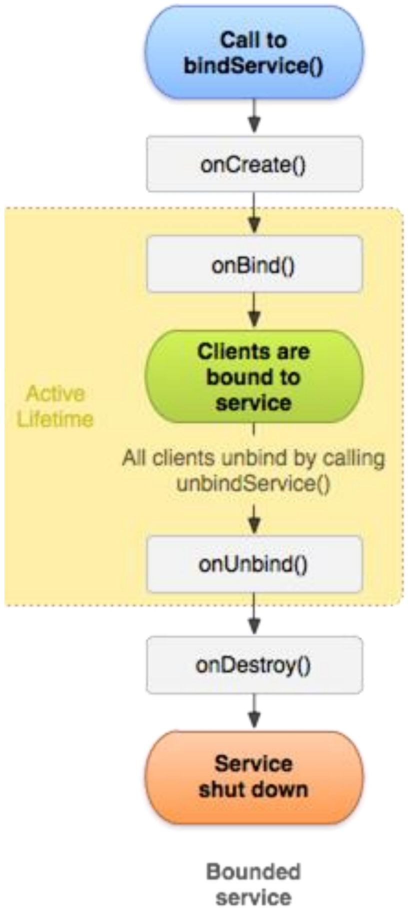
**Characteristics of services**

- Started with an `Intent`.
- Can stay running when user switches applications.
- Lifecycle—which you must manage.
- Other apps can use the service—manage permissions.
- Runs in the main thread of its hosting process.

## Forms of services: started and bound

- Started with `startService()`
- Runs indefinitely until it stops itself
- Usually does not update the UI





- Offers a client-server interface that allows components to interact with the service
- Clients send requests and get results
- Started with `bindService()`
- Ends when all clients unbind

## Foreground services

Runs in the background but requires that the user is actively aware it exists—e.g. music player using music service

- Higher priority than background services since user will notice its absence—unlikely to be killed by the system
- Must provide a notification which the user cannot dismiss while the service is running

## Background services limitations

- A foreground app, can create and run both foreground and background services.
- When an app goes into the background, the system stops the app's background services.
- The `startService()` method now throws an `IllegalStateException` if an app is targeting API 26.
- These limitations don't affect foreground services or bound services.

## Creating a service

- `<service android:name=".ExampleService" />`
- Manage permissions.
- Subclass `IntentService` or `Service` class.
- Implement lifecycle methods.
- Start service from `Activity`.
- Make sure service is stoppable.

## Stopping a service

- A **started service** must manage its own lifecycle
- If not stopped, will keep running and consuming resources
- The service must stop itself by calling stopSelf()
- Another component can stop it by calling stopService()
- **Bound service** is destroyed when all clients unbound
- **IntentService** is destroyed after onHandleIntent() returns

ACTIVITY

Reference – Guide

## 4.3.4. AsyncTask and AsyncTaskLoader

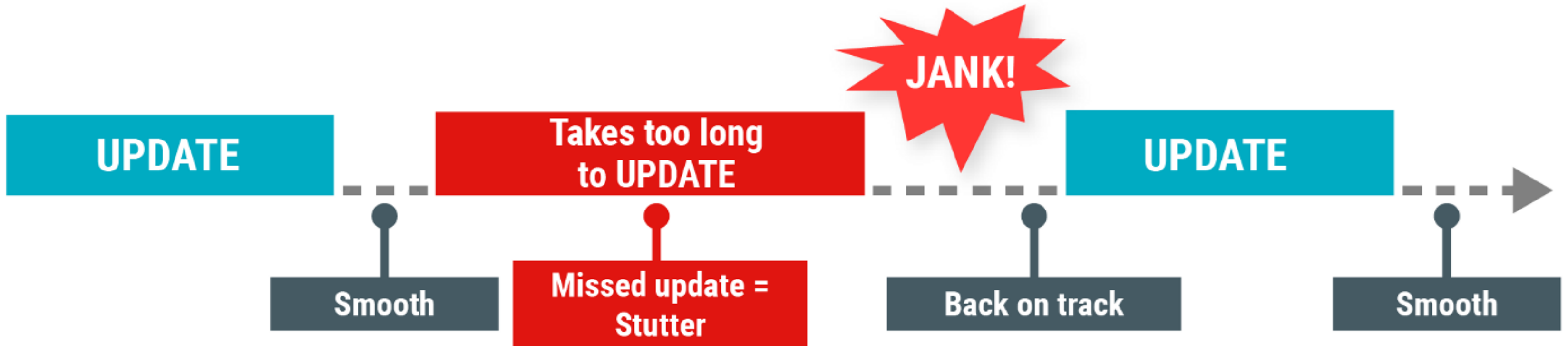
### Threads

**The main thread**

- Independent path of execution in a running program
- Code is executed line by line
- App runs on Java thread called "main" or "UI thread"
- Draws UI on the screen
- Responds to user actions by handling UI events

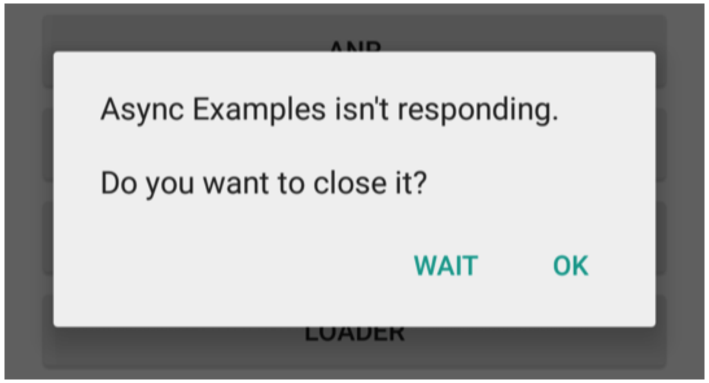
## The Main thread must be fast

- Hardware updates screen every 16 milliseconds
- UI thread has 16 ms to do all its work
- If it takes too long, app stutters or hangs



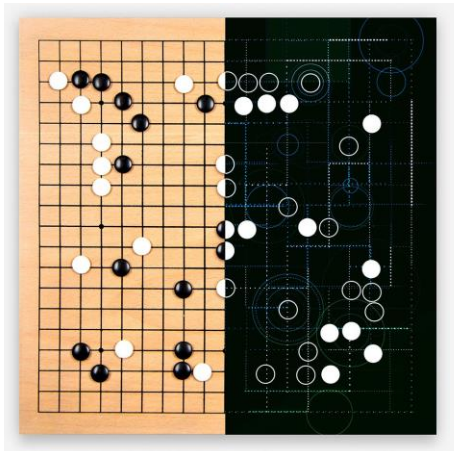
## Users uninstall unresponsive apps

- If the UI waits too long for an operation to finish, it becomes unresponsive
- The framework shows an Application Not Responding (ANR) dialog


Async Examples isn't responding.

Do you want to close it?

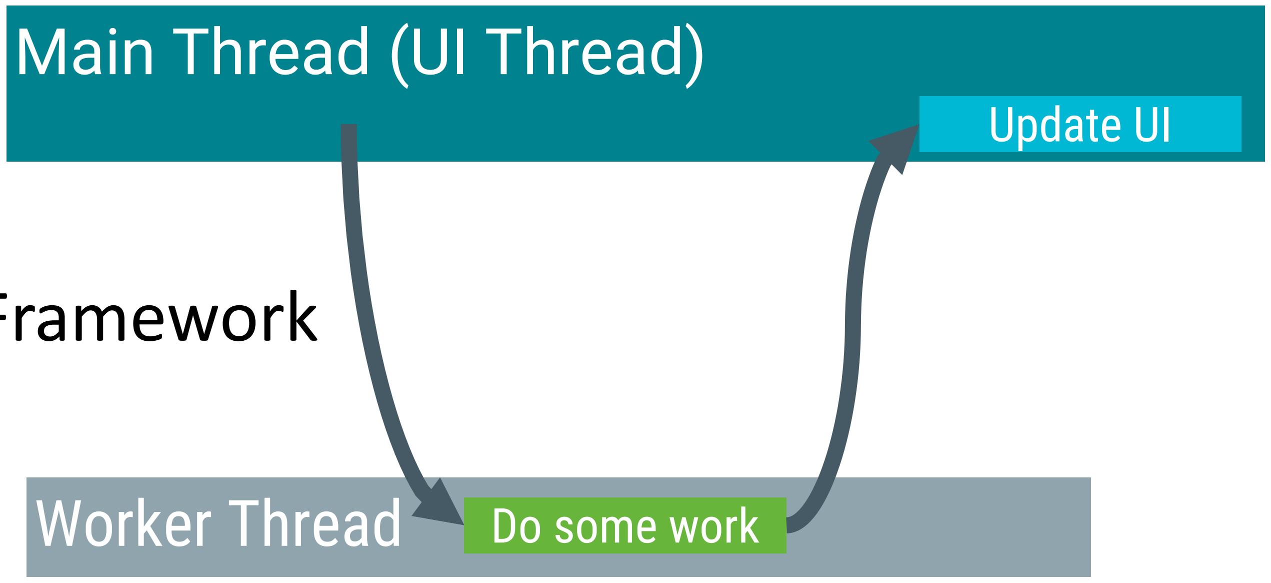
## What is a long running task?



- Network operations
- Long calculations
- Downloading/uploading files
- Processing images
- Loading data

## Background threads

Execute long running tasks on a **background thread**



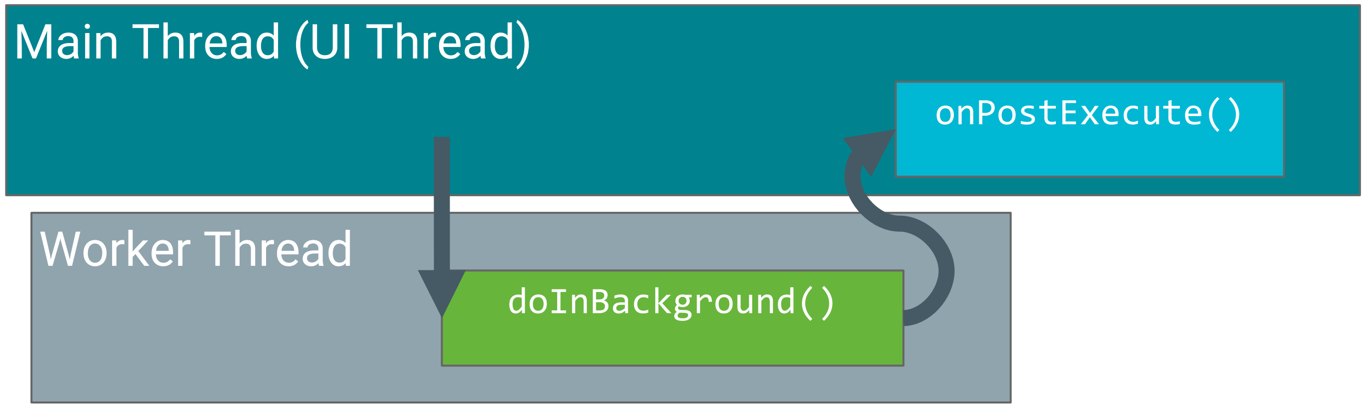
- AsyncTask
- The Loader Framework
- Services

## Two rules for Android threads

- Do not block the UI thread
  - Complete all work in less than 16 ms for each screen
  - Run slow non-UI work on a non-UI thread
- Do not access the Android UI toolkit from outside the UI thread
  - Do UI work only on the UI thread

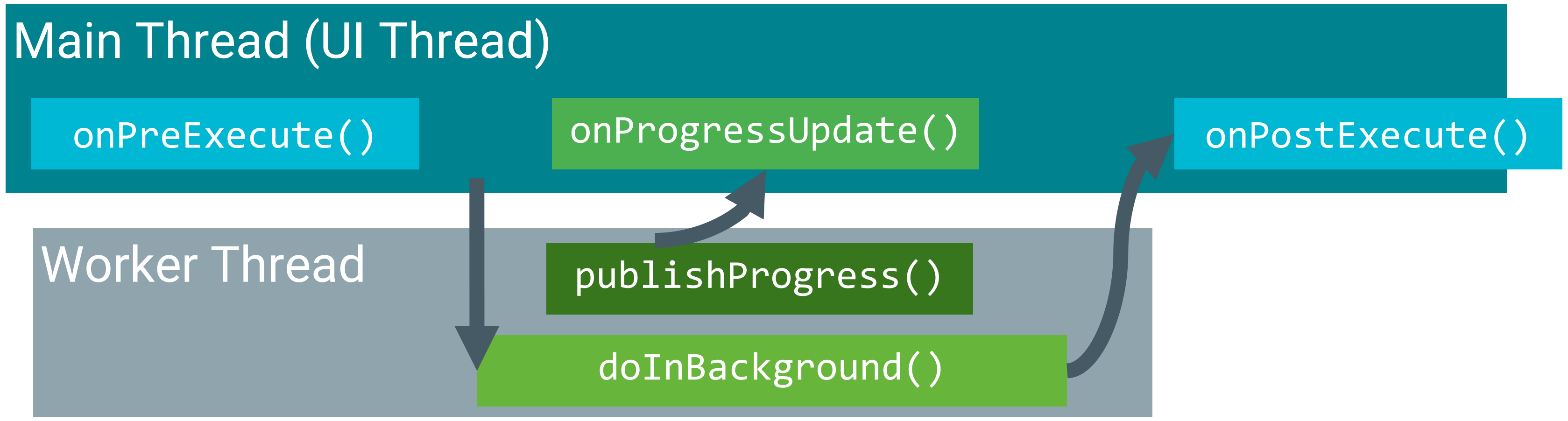
## What is AsyncTask?

Use AsyncTask to implement basic background tasks



- `doInBackground()`—runs on a background thread
  - All the work to happen in the background
- `onPostExecute()`—runs on main thread when work done
  - Process results
  - Publish results to the UI

## AsyncTask helper methods



- `onPreExecute()`
  - Runs on the main thread
  - Sets up the task

- `onProgressUpdate()`
  - Runs on the main thread
  - receives calls from `publishProgress()` from background thread

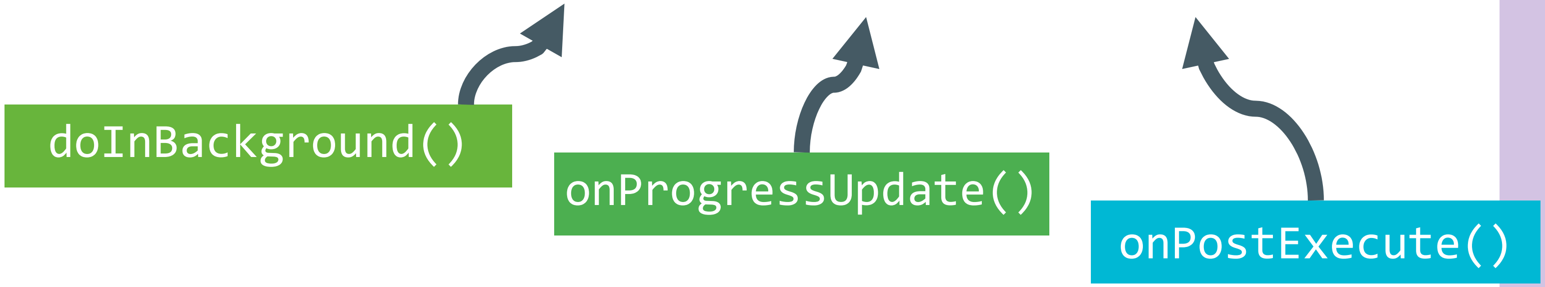
## Creating an AsyncTask

1. Subclass AsyncTask
2. Provide data type sent to `doInBackground()`
3. Provide data type of progress units for `onProgressUpdate()`
4. Provide data type of result for `onPostExecute()`

```
private class MyAsyncTask
    extends AsyncTask<URL, Integer, Bitmap> {...}
```

# MyAsyncTask class definition

```
private class MyAsyncTask
    extends AsyncTask<String, Integer, Bitmap> {...}
```



- String—could be query, URI for filename
- Integer—percentage completed, steps done
- Bitmap—an image to be displayed
- Use Void if no data passed

## When to use AsyncTask

- Short or interruptible tasks
- Tasks that do not need to report back to UI or user
- Lower priority tasks that can be left unfinished
- Use AsyncTaskLoader otherwise

## Limitations of AsyncTask

- When device configuration changes, Activity is destroyed
- AsyncTask cannot connect to Activity anymore
- New AsyncTask created for every config change
- Old AsyncTasks stay around
- App may run out of memory or crash

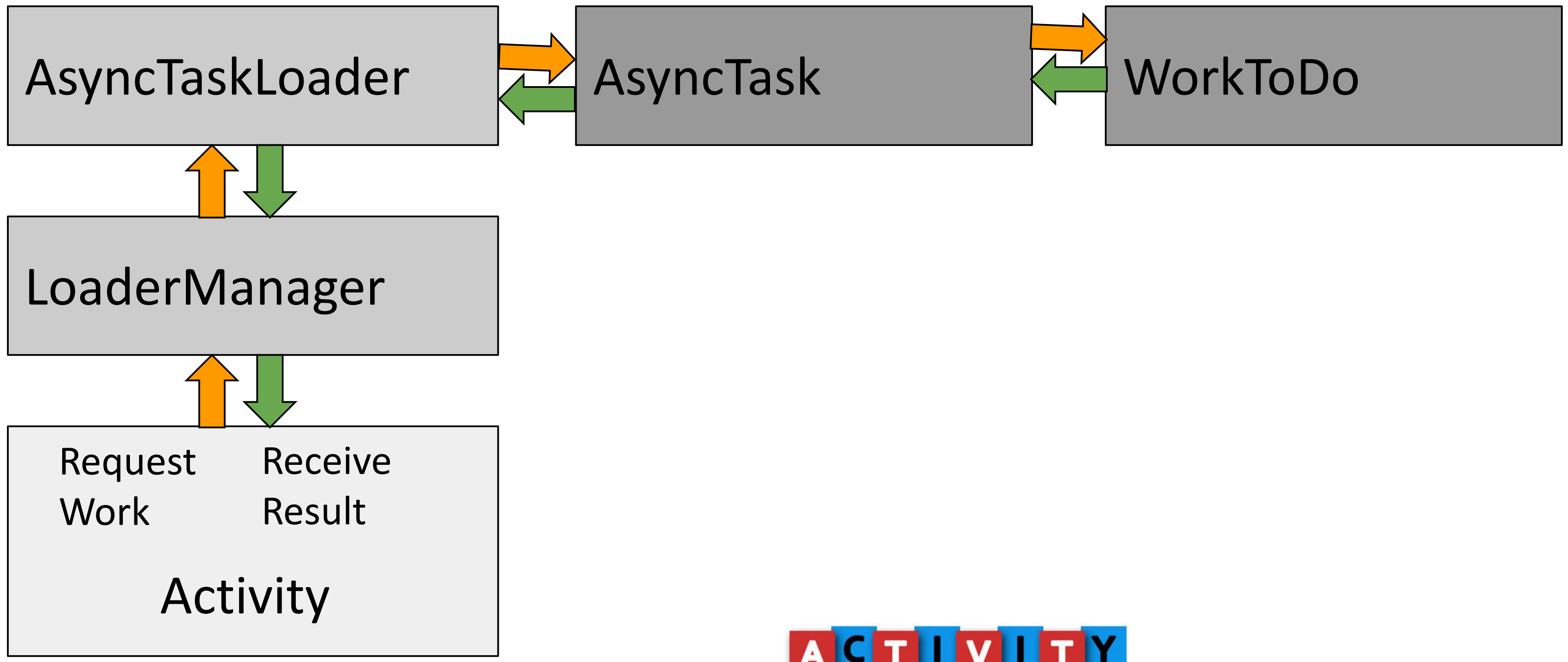
## What is a Loader?

- Provides asynchronous loading of data
- **Reconnects to Activity after configuration change**
- Can monitor changes in data source and deliver new data
- Callbacks  implemented in Activity
- Many types of loaders available
    - AsyncTaskLoader, CursorLoader

## Why use loaders?

- Execute tasks OFF the UI thread
- LoaderManager handles configuration changes for you
- Efficiently implemented by the framework
- Users don't have to wait for data to load

## AsyncTaskLoader Overview



ACTIVITY

Reference – AsyncTaskLoader

## 4.4. Sensors and Communication

### 4.4.1. Sensors (Sensor Identification and Registration)

Most Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy, and are useful if you want to monitor three-dimensional device movement or positioning, or you want to monitor changes in the ambient environment near a device.

*For example*

A game might track readings from a device's gravity sensor to infer complex user gestures and motions, such as tilt, shake, rotation, or swing.

Likewise, a weather application might use a device's temperature sensor and humidity sensor to calculate and report the dewpoint, or a travel application might use the geomagnetic field sensor and accelerometer to report a compass bearing.

## Categories of sensors

- **Motion sensors**

These sensors measure acceleration forces and rotational forces along three axes. This category includes accelerometers, gravity sensors, gyroscopes, and rotational vector sensors.

- **Environmental sensors**

These sensors measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity. This category includes barometers, photometers, and thermometers.

- **Position sensors**

These sensors measure the physical position of a device. This category includes orientation sensors and magnetometers.

## Sensors

**Motion sensors**

- Accelerometers
- Gravity sensors
- Gyroscopes
- Rotational vector sensors

**Environmental sensors**

- Barometers
- Photometers (light sensors)
- Thermometers

**Position sensors**

- Magnetometers (geomagnetic field sensors)
- Proximity sensors

## Sensor Availability

Sensor availability varies from device to device, it can also vary between

Android versions

- Most devices have accelerometer and magnetometer
- Some devices have barometers or thermometers
- Device can have more than one sensor of a given type
- Availability varies between Android version

## Android sensor framework

**Sensor Manager**

- Access and listen to sensors
- Register and unregister sensor event listeners
- Acquire orientation information
- Provides constants for accuracy, data acquisition rates, and calibration

**Framework Classes**

- Sensor : Determine specific sensor's capabilities
- SensorEvent: Info about event, including raw sensor data
- SensorEventListener: Receives notifications about sensor events
    - When sensor has new data
    - When sensor accuracy changes

Reference Links –

Sensor types supported by the Android platform.

Identifying Sensors and Sensor Capabilities.

Monitoring Sensor Events.

## 4.4.2. Orientation and Movement

The Android platform provides two sensors that let you determine the position of a device: the **geomagnetic field sensor** and the **accelerometer**.

The geomagnetic field sensor and the proximity sensor are hardware-based. Most handset and tablet manufacturers include a geomagnetic field sensor.

Likewise, handset manufacturers usually include a proximity sensor to determine when a handset is being held close to a user's face (for example, during a phone call).

For determining a device's orientation, you can use the readings from the device's accelerometer and the geomagnetic field sensor.

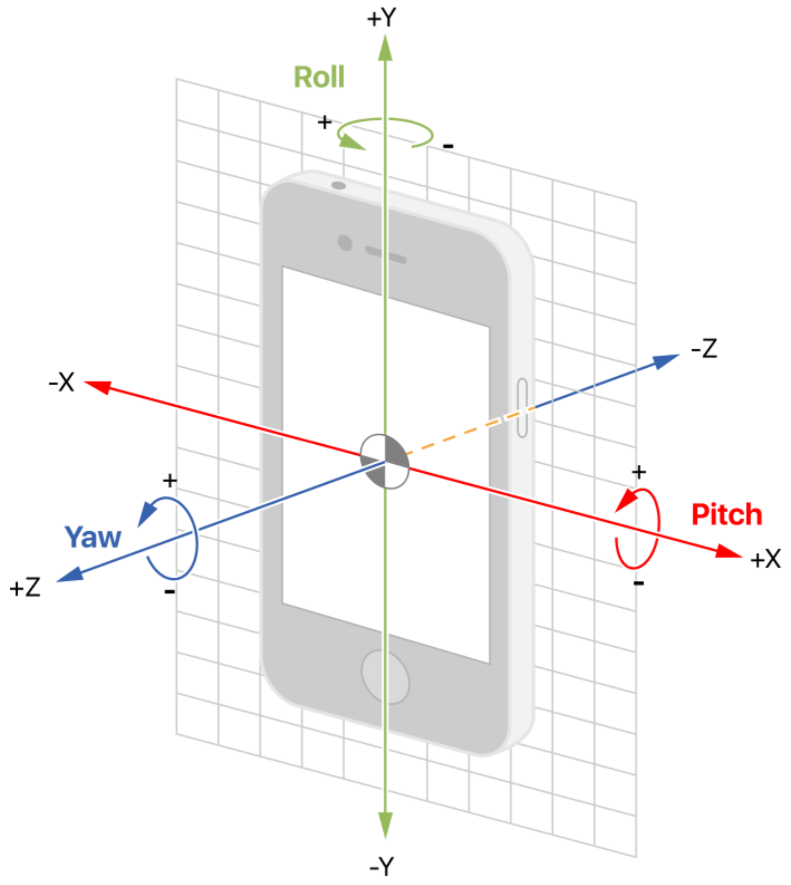
ACTIVITY

Reference – Sensor Position

## Compute the device's orientation

By computing a device's orientation, you can monitor the position of the device relative to the earth's frame of reference (specifically, the magnetic north pole).

The system computes the orientation angles by using a device's geomagnetic field sensor in combination with the device's accelerometer. Using these two hardware sensors



# Orientaion Angles

**Azimuth (degrees of rotation about the -z axis).** This is the angle between the device's current compass direction and magnetic north. If the top edge of the device faces magnetic north, the azimuth is 0 degrees; if the top edge faces south, the azimuth is 180 degrees. Similarly, if the top edge faces east, the azimuth is 90 degrees, and if the top edge faces west, the azimuth is 270 degrees.

**Pitch (degrees of rotation about the x axis).** This is the angle between a plane parallel to the device's screen and a plane parallel to the ground. If you hold the device parallel to the ground with the bottom edge closest to you and tilt the top edge of the device toward the ground, the pitch angle becomes positive. Tilting in the opposite direction— moving the top edge of the device away from the ground—causes the pitch angle to become negative. The range of values is -180 degrees to 180 degrees.

**Roll (degrees of rotation about the y axis).** This is the angle between a plane perpendicular to the device's screen and a plane perpendicular to the ground. If you hold the device parallel to the ground with the bottom edge closest to you and tilt the left edge of the device toward the ground, the roll angle becomes positive. Tilting in the opposite direction—moving the right edge of the device toward the ground— causes the roll angle to become negative. The range of values is -90 degrees to 90 degrees.

## 4.4.3. Sending and Receiving SMS

**TELEPHONY**

The Android telephony APIs let your applications access the underlying telephone hardware stack, making it possible to create your own dialer — or integrate call handling and phone state monitoring into your applications.

Because of security concerns, the current Android SDK does not allow you to create your own "in call" Activity — the screen that is displayed when an incoming call is received or an outgoing call has been placed.

The following sections focus on how to monitor and control phone, service, and cell events in your applications to augment and manage the native phone-handling functionality. If you wish, you can use the same techniques to implement a replacement dialer application.

## Telephony.Sms.Inbox

Contains all text-based SMS messages in the SMS app inbox. You can read messages and take appropriate actions like reading and OTP and authenticate a user.

ACTIVITY

References –
[Summary in Telephony]()
[Telephony.Sms.Inbox]()