1. Basics of Android

What is Android...?

- Android is an open source and Linux-based Operating System for mobile devices such as smart phones and tablet computers.
- Android was developed by the Open Handset Alliance, led by Google, and other companies.

First Release

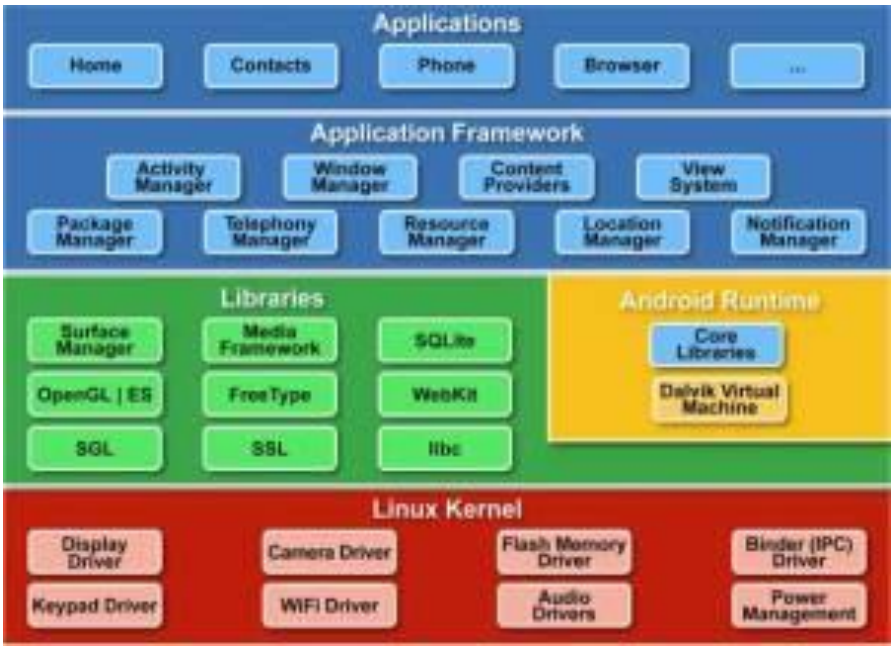
- The first beta version of the Android Software Development Kit (SDK) was released by Google in 2007.
- The first commercial version, Android 1.0, was released in September 2008.
- The first android phone was launched by HTC on October 2008(HTC Dream – T-Mobile G1 in USA and some parts of Europe).

Application

- Set of Programs with dedicated functionality is known as Application
- Two types of Applications based on provided interface
  1. System application - Application which helps to communicate between hardware and user application.
     - Ex: Operating Systems, Driver Software etc...
  2. User application - Applications which provides solution for common problems.
     - Ex: MS Office, Banking Applications, Mail & Message Applications, Video Players, etc...
- There are 2 types of Applications based on where it is installed
  1 Standalone application (unshared)
  2 Web Application (shared)
- Standalone Applications are present in our own computer and they are dedicated per device.
- For ex : Adobe Reader , Web Browser , Media Player etc.
- Web applications are not present in our own computer but they are present in some other computer where our own computer and that computer are “Network Connected”
- For ex : Gmail , Facebook ,Twitter etc.
- There are two types of Standalone Applications
- Desktop Applications
- Mobile Applications
- Desktop Applications: As the name implies ,they are present in our “Desktop/Computer”.
- Mobile Applications: As the name implies ,they are present in our “Smart Phone”.

- Web Applications can be of 3 Tiered(Layered) or 2 tiered
- 3-Tier architecture application: accessed using “Web Browser”
    - Layer1 - Client
    - Layer2 - Server
    - Layer3 - DB
- 2-Tier architecture application: accessed using “Mobile App”
    - Layer1 - Client[DB maintained in Client's device].
    - Layer2 - Server
- Now a days, users depend more on Mobile App.

Android architecture



Linux kernel

- It has Linux Version 2.6.x for core system services and thus android handles only “Kernel” portion in Linux
- A kernel is a central component of an operating system. It acts as an interface between the user applications and the hardware.
- The main tasks of the kernel are :
    - Process management
    - Device management
    - Memory management
    - Interrupt handling
    - I/O communication
    - File system etc

Android Runtime

- For devices running Android version 5.0 (API level 21) or higher, each app runs in its own process and with its own instance of the Android Runtime (ART). ART is written to run multiple virtual machines on low-memory devices by executing DEX files, a byte code format designed especially for Android that's optimized for minimal memory footprint.

Core Libraries- Uses the JAVA Programming Language

Dalvik Virtual Machine

- Android Application operates in its own process with the specific instance of the Dalvik virtual machine (DVM).
- The DalvikVM is Java based licenses free VM.
- It executes files in the Dalvik Executable (.dex) format.

Libraries

1. *Libc*: it is c standard lib.
2. *SSL*: Secure Socket Layer for security
3. *SGL*: 2D picture engine where SGL is "Scalable Graphics Library"
4. *OpenGL|ES*: 3D image engine
5. *Media Framework*: essential part of Android multi-media
6. *SQLite*: Embedded database
7. *Web Kit*: Kernel of web browser
8. *Free Type*: Bitmap and Vector
9. *Surface Manager*: Manage different windows for different applications

Application framework

- A rich and extensible View System you can use to build an app's UI, including lists, grids, text boxes, buttons, and even an embeddable web browser
- A Resource Manager, providing access to non-code resources such as localized strings, graphics, and layout files
- A Notification Manager that enables all apps to display custom alerts in the status bar
- An Activity Manager that manages the lifecycle of apps and provides a common navigation back stack
- Content Providers that enable apps to access data from other apps, such as the Contacts app, or to share their own data

Application

- Android comes with a set of core apps for email, SMS messaging, calendars, internet browsing, contacts, and more.

Android Studio project structure

Android View

manifests/

- AndroidManifest.xml is one of the most important file in the Android project structure.
- It contains information of the package, including components of the application
- It is responsible to protect the application to access any protected parts by providing the permissions
- It also declares the android api that the application is going to use

java/

- The java folder contains the Java source code files of the application organized into packages.

res/

- Res folder is where all the external resources for the application are stored.
- **Drawable:** The folders are to provide alternative image resources to specific screen configurations.
- **Layout:** It contains XML files that define the User Interface of the application
- **Mipmap:** The mipmap folder is used for placing the app icons only..
- **Values :** XML files that define simple values such as strings, arrays, integers, dimensions, colors, styles etc. are placed in this folder

Gradle Scripts

- Gradle scripts are used to automate tasks.
- For the most part, Android Studio performs application builds in the background without any intervention from the developer.
- This build process is handled using the Gradle system,

Project View

.idea/

- In this folder the project specific metadata is stored by Android Studio.

Project Module(app)

- This is the actual project folder where the application code resides. The application folder has following sub directories

- **build** : This has all the complete output of the make process i.e. classes.dex, compiled classes and resources, etc. The important part is that the R.java is found here under build/generated/source/r/…/R.java
- **libs** : This is a commonly seen folder in eclipse and android studio, which optionally can hold the libraries or .jar files
- **src** : The src folder can have both application code and android unit test script. You will find two folders named “androidTest” and “main” correspond to src folder.
- **Gradle**: This is where the gradle build systems jar wrapper is found. This jar is how Android Studio communicates with gradle installed in Windows/MAC.

External Libraries

- This is not actually a folder but a place where Referenced Libraries and information on targeted platform SDK are shown.

What is R.java?

- Android R.java is an auto-generated file by **aapt** (Android Asset Packaging Tool) that contains resource IDs for all the resources of res/ directory.
- On creating any component in the activity_main.xml file, id for the corresponding component is automatically created in this file.
- This id can be used in the activity source file to perform any action on the component.

Build Process

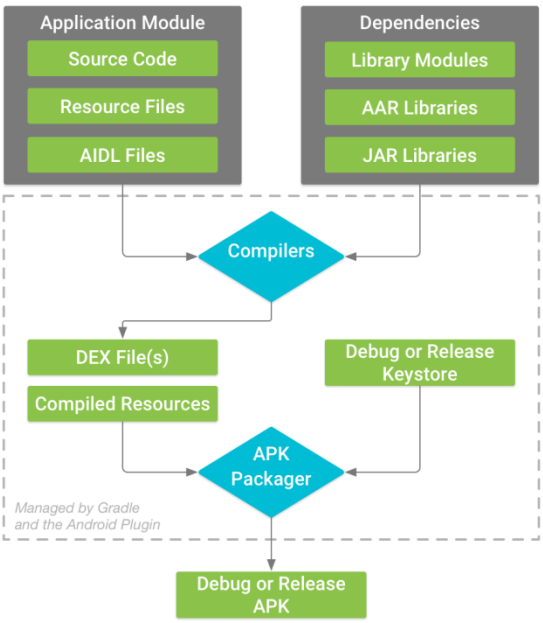
AIDL
Android Interface Defininatoin Language

AAR
Android Archive

JAR
Java Archive

DEX
Dalvik Executable

APK
Android Application Package



The build process for a typical Android app module is follows these general steps:

- The compilers convert your source code into DEX (Dalvik Executable) files, which include the bytecode that runs on Android devices, and everything else into compiled resources.
- The APK Packager combines the DEX files and compiled resources into a single APK. Before your app can be installed and deployed onto an Android device, the APK must be signed.
- The APK Packager signs your APK using either the debug or release keystore:
    - If you are building a debug version of your app, that is, an app you intend only for testing and profiling, the packager signs your app with the debug keystore.
    - Android Studio automatically configures new projects with a debug keystore.
    - If you are building a release version of your app that you intend to release externally, the packager signs your app with the release keystore.
- Before generating your final APK, the packager uses the zipalign tool to optimize your app to use less memory when running on a device.
- At the end of the build process, you have either a debug APK or release APK of your app that you can use to deploy, test, or release to external users.

Application Components

- Application components are the essential **building blocks of an Android application.**
- There are four different types of application components.
    1. Activities
    2. Services
    3. Broadcast receivers
    4. Content providers

Activities

- An activity represents a single screen with a user interface.
- For example, an email application might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails.
- Although the activities work together to form a cohesive user experience in the email application, each one is independent of the others.

Services

- A service is a component that runs in the background to perform long-running operations or to perform work for remote processes.
- A service does not provide a user interface.

- For example, a service might play music in the background while the user is in a different application,
- or it might fetch data over the network without blocking user interaction with an activity.

Broadcast receivers

- A broadcast receiver is a component that responds to system-wide broadcast announcements.
- Many broadcasts originate from the system—for example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured.
- Applications can also initiate broadcasts—for example, to let other applications know that some data has been downloaded to the device and is available for them to use.
- Although broadcast receivers don't display a user interface, they may create a status bar notification to alert the user when a broadcast event occurs.

Content providers

- A content provider manages a shared set of application data.
- You can store the data in the file system, SQLite database, on the web, or any other persistent storage location your application can access.
- Through the content provider, other applications can query or even modify the data (if the content provider allows it).
- For example, the Android system provides a content provider that manages the user's contact information.

Activity Life Cycle

- Android Activity Lifecycle is controlled by 7 methods of android.app.Activity class.
- The android Activity is the subclass of ContextThemeWrapper class.

onCreate()

- You must implement this callback, which fires when the system first creates the activity.
- On activity creation, the activity enters the Created state.
- In the onCreate() method, you perform basic application startup logic that should happen only once for the entire life of the activity.

onStart()

- When the activity enters the Started state, the system invokes this callback.

- The onStart() call makes the activity visible to the user, as the app prepares for the activity to enter the foreground and become interactive.

onResume()

- When the activity enters the Resumed state, it comes to the foreground, and then the system invokes the onResume() callback.
- This is the state in which the app interacts with the user.
- The app stays in this state until something happens to take focus away from the app.
- Such an event might be, for instance, receiving a phone call, the user's navigating to another activity, or the device screen's turning off.

onPause()

- 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).
- Use the onPause() method to pause operations such animations and music playback that should not continue while the Activity is in the Paused state, and that you expect to resume shortly.
- There are several reasons why an activity may enter this state. For example: Some event interrupts app execution, as described in the onResume() section. This is the most common case.

onStop()

- When your activity is no longer visible to the user, it has entered the Stopped state, and the system invokes the onStop() callback.
- This may occur, for example, when a newly launched activity covers the entire screen.
- The system may also call onStop() when the activity has finished running, and is about to be terminated.
- In the onStop() method, the app should release almost all resources that aren't needed while the user is not using it.

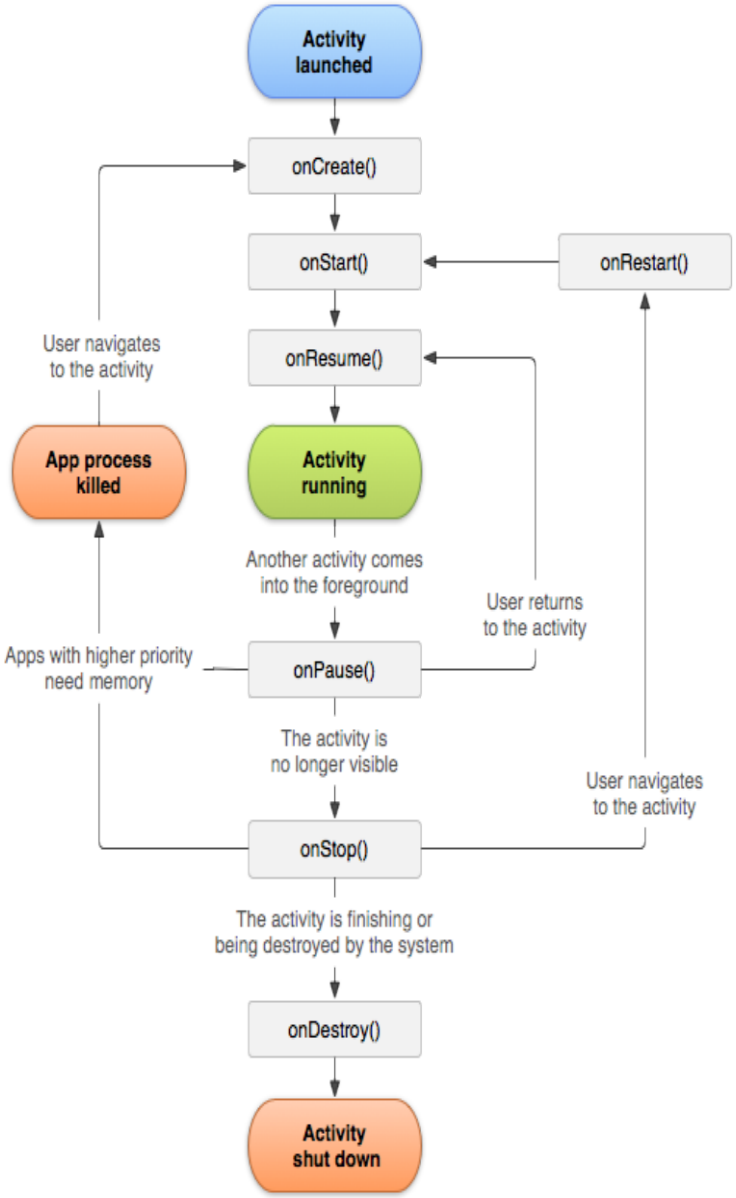
onRestart ()

- This callback method called after onStop() when the current activity is being re-displayed to the user (the user has navigated back to it).
- It will be followed by onStart() and then onResume().

onDestroy()

- Called before the activity is destroyed. This is the final call that the activity receives.

- The system either invokes this callback because the activity is finishing due to someone's calling finish(), or because the system is
- temporarily destroying the process containing the activity to save space.
- The system may also call this method when an orientation change occurs, and then immediately call onCreate() to recreate the process (and the components that it contains)
- in the new orientation.
- The onDestroy() callback releases all resources that have not yet been released by earlier callbacks such as onStop().



Layout, View, ViewGroup

- A layout defines the visual structure for a user interface, such as the UI for an activity or app widget.
- A View is an object that draws something on the screen that the user can interact with.
- A ViewGroup is an object that holds other View (and ViewGroup) objects in order to define the layout of the user interface.
- You define your layout in an XML file which offers a human-readable.
- The advantage to declaring your UI in XML is that it enables you to better separate the presentation of your application from the code that controls its behavior.
- In general, the XML vocabulary for declaring UI elements closely follows the structure and naming of the classes and methods, where element names correspond to class names and attribute names correspond to methods.

Android units of measurements px, in, mm, dp, dip and sp

- Web designers have always designed web pages in pixels.
- The problem with pixels is that with increased screen resolutions the,
- with more dots for inch, the display content such as text and images become smaller and smaller.
- And with so many devices today with varied resolutions it becomes harder for designers to design the best interface suited
- for all of them. Here comes the dip and sp to rescue for android programmers.

Android supports the following measurements:

- px (Pixels) - Actual pixels or dots on the screen.
- in (Inches) - Physical size of the screen in inches.
- mm (Millimeters) - Physical size of the screen in millimeters.
- pt (Points) - 1/72 of an inch.
- dp (Density-independent Pixels) - An abstract unit that is based on the physical density of the screen.
    - These units are relative to a 160 dpi screen, so one dp is one pixel on a 160 dpi screen.
    - The ratio of dp-to-pixel will change with the screen density, but not necessarily in direct proportion. "dip" and "dp" are same.
- sp (Scale-independent Pixels) - Similar to dp unit, but also scaled by the user's font size preference.

Note: Always use sp for font sizes and dp for everything else.

Toasts

- A toast provides simple feedback about an operation in a small popup.
- It only fills the amount of space required for the message and the current activity remains visible and interactive.
- Toasts automatically disappear after a timeout.
- Method used to create toast

public static Toast makeText(Context context, CharSequence text, int duration)
public void show()

Constants

public static final int LENGTH_LONG
public static final int LENGTH_SHORT

Usage

```
Toast.makeText(context_object,"Hello",Toast.LENGTH_SHORT).show();
```

Note : context_object – getApplicationContext()
getBaseContext()
this – if and only if the class is of type Context

Buttons

- A Button is a Push-button which can be pressed, or clicked, by the user to perform an action.

Button Handlings

- If you use android:onClick="setLogin" then you need to make a method with the same name, setLogin:

```
// Please be noted that you need to add the "View v" parameter
public void setLogin(View v) {

}
```

Note : Do not mix layout with code by using android:onClick tag in your XML. Instead, move the click method to your class with OnClickListener method like:

```
Button button = (Button) findViewById(R.id.button1);
button.setOnClickListener(new OnClickListener() {
 public void onClick(View v) {
   // TODO Auto-generated method stub
 }
});
```

Radio Buttons

- Radio Buttons are used when we need to select only one item from a list of presented items.
- A radio button consists of two states – checked and unchecked. Clicking an unchecked button changes its state to "checked" state and "unchecked" for the previously selected radio button.
- To toggle a checked state to unchecked state, we need to chose another item. Clicking a checked state button doesn't do any good.
- A RadioGroup is a set of radio buttons. Radio Buttons that have different parent ViewGroup can't be termed as a RadioGroup

Some of the attributes of android radio button and radio group are listed below:

- **android:orientation** : This property on the Radio group defines the orientation to position its child view consisting of Radio Buttons. It can be either horizontal or vertical
- **check(id)** : This sets the selection to the radio button whose identifier is passed in parameter. -1 is used as the selection identifier to clear the selection
- **clearCheck()** : It clears the selection. When the selection is cleared, no radio button in this group is selected and getCheckedRadioButtonId() returns null
- **getCheckedRadioButtonId()** : It returns the identifier of the selected radio button in this group. If its empty selection, the returned value is -1
- **setOnCheckedChangeListener()** : This registers a callback to be invoked when the checked radio button changes in this group. We must supply instanceof RadioGroup.OnCheckedChangeListener to setOnCheckedChangeListener() method

CheckBox

- Checkboxes allows the user to select one or more options from a list of options.
- You can choose one or more options from the list.
- To get user selected check box values from the UI
- we will use the isChecked() to check if the check box was selected or not.

ImageView and ImageButton

- Android ImageView is used to display an image file. Android also has an ImageButton.
- As the name suggests, the ImageButton component is a button with an image on.

Android DatePickerDialog and TimePickerDialog

- Date Picker and Time Picker can be used as independent widgets but they occupy more space on the screen. Hence using them inside a Dialog is a better choice. Fortunately android provides use with its own DatePickerDialog and TimePickerDialog classes.
- DatePickerDialog and TimePickerDialog classes have onDateSetListener() and onTimeSetListener() callback methods respectively.
- These callback methods are invoked when the user is done with filling the date and time respectively.

The DatePickerDialog class consists of a 5 argument constructor with the parameters listed below.

1. Context: It requires the application context
2. CallBack Function: onDateSet() is invoked when the user sets the date with the following parameters:
3. int year : It will be store the current selected year from the dialog
4. int monthOfYear : It will be store the current selected month from the dialog
5. int dayOfMonth : It will be store the current selected day from the dialog
6. int mYear : It shows the the current year that's visible when the dialog pops up
7. int mMonth : It shows the the current month that's visible when the dialog pops up
8. int mDay : It shows the the current day that's visible when the dialog pops up

The TimePickerDialog class consists of a 5 argument constructor with the parameters listed below.

1. Context: It requires the application context
2. CallBack Function: onTimeSet() is invoked when the user sets the time with the following parameters:
3. int hourOfDay : It will be store the current selected hour of the day from the dialog
4. int minute : It will be store the current selected minute from the dialog
5. int mHours : It shows the the current Hour that's visible when the dialog pops up
6. int mMinute : It shows the the current minute that's visible when the dialog pops up
7. boolean false : If its set to false it will show the time in 24 hour format else not

Android Spinner

- Android Spinner is just a drop down list similar to what's seen in other programming languages such as in HTML pages.
- In Android, Spinner is used to select one value from a set of values. In the default state, a spinner shows its currently selected value.
- Touching the spinner displays a drop down menu with all other available values, from which the user can select a new one.
- Android spinner is associated with AdapterView. So we need to set the adapter class with the Spinner**.**

ArrayAdapter can be created using different variants of Constructor. One of the constructor is
ArrayAdapter(Context context, int layoutResource, T[] objects)
T[] refers to generic array item type in the following example T[] = String[]

Example:

```
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
        android.R.layout.simple_spinner_dropdown_item, spinnerArrayItems);
spinner.setAdapter(adapter);
```

AutoCompleteTextView & MultiCompleteTextView

- AutoCompleteTextView completes the word based on the reserved words, so no need to write all the characters of the word.
- This view is a editable text field, it displays a list of suggestions in a drop down menu from which user can select only one suggestion or value.
- AutoCompleteTextView is the subclass of EditText class. The MultiAutoCompleteTextView is the subclass of AutoCompleteTextView class.
- MultiAutoCompleteTextView works the same way as the AutoCompleteTextView except you can add aTokenizer that parses the text and allows you to suggest where to start suggesting words.
- Android MultiAutoCompleteTextView is an editable text view, extending AutoCompleteTextView, that can show completion suggestions for the substring of the text where the user is typing instead of necessarily for the entire thing.
- You must must provide a MultiAutoCompleteTextView. Tokenizer to distinguish the various substring like comma (,).

BroadcastReceiver

- Android BroadcastReceiver is an application component of android that listens to system-wide broadcast events or intents.
- When any of these events occur it brings the application into action by either creating a status bar notification or performing a task.
- Unlike activities, android BroadcastReceiver doesn't contain any user interface. Broadcast receiver is generally implemented to delegate the tasks to services depending on the type of intent data that's received.
- Following are some of the important system wide generated intents.
    1. android.intent.action.BATTERY_LOW : Indicates low battery condition on the device.
    2. android.intent.action.BOOT_COMPLETED : This is broadcast once, after the system has finished booting
    3. android.intent.action.CALL : To perform a call to someone specified by the data
    4. android.intent.action.DATE_CHANGED : The date has changed
    5. android.intent.action.REBOOT : Have the device reboot
    6. android.net.conn.CONNECTIVITY_CHANGE : The mobile network or wifi connection is changed(or reset)

To set up a Broadcast Receiver in android application we need to do the following two things.

1. Creating a BroadcastReceiver
2. Registering a BroadcastReceiver

Creating a BroadcastReceiver

```
public class MyReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    Toast.makeText(context, "Action: " + intent.getAction(),
Toast.LENGTH_SHORT).show();
  }
}
```

- BroadcastReceiver is an abstract class with the onReceiver() method being abstract.
- The onReceiver() method is first called on the registered Broadcast Receivers when any event occurs.

Registering the BroadcastReceiver in android app

- A BroadcastReceiver can be registered in two ways.

1. By defining it in the AndroidManifest.xml file as shown below.

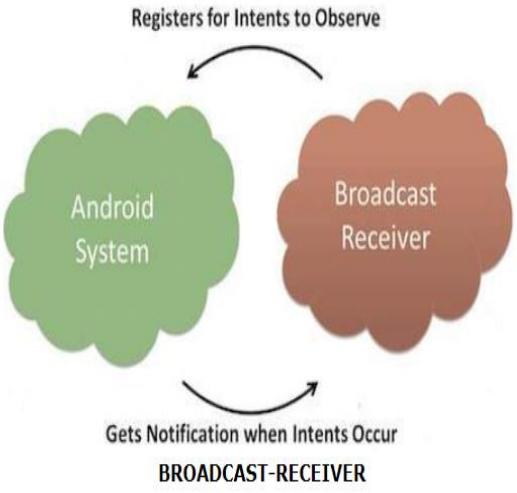
```
<receiver android:name=".ConnectionReceiver" >
        <intent-filter>
          <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
        </intent-filter>
</receiver>
```

- Using intent filters we tell the system any intent that matches our sub elements should get delivered to that specific broadcast receiver.

2. By defining it programmatically

- Following snippet shows a sample example to register broadcast receiver programmatically.

```
IntentFilter filter = new IntentFilter();
intentFilter.addAction(getPackageName() +
"android.net.conn.CONNECTIVITY_CHANGE");

MyReceiver myReceiver = new MyReceiver();
registerReceiver(myReceiver, filter);
```



Registering in process in two ways

Through manifest file:

```
<receiver android:name=".MyReceiver" >
        <intent-filter>
          <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
        </intent-filter>
</receiver>
```

Through IntentFilter class:

```
IntentFilter filter = new IntentFilter();
intentFilter.addAction(getPackageName() +"android.net.conn.CONNECTIVITY_CHANGE");

MyReceiver myReceiver = new MyReceiver();
registerReceiver(myReceiver, filter);
```

To unregister a broadcast receiver in onStop() or onPause() of the activity the following snippet can be used.

```
@Override
protected void onPause() {
  unregisterReceiver(myReceiver);
  super.onPause();
}
```

Sending Broadcast intents from the Activity

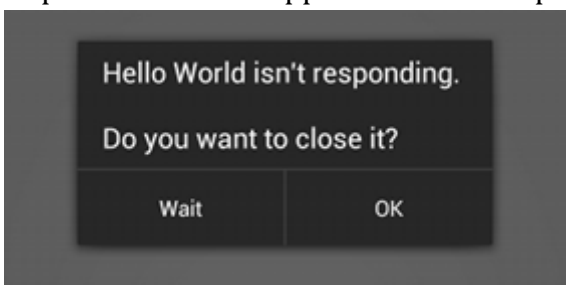
- The following snippet is used to send an intent to all the related BroadcastReceivers.

```
Intent intent = new Intent();
intent.setAction("com.journaldev.CUSTOM_INTENT");
sendBroadcast(intent);
```

Note : Don't forget to add the above action in the intent filter tag of the manifest or programmatically.

Android UI Thread and ANR

- On the Android platform, applications operate, by default, on one thread. This thread is called the *UI thread*.
- It is often called that because this single thread displays the user interface and listens for events that occur when the user interacts with the app.
- Developers quickly learn that if code running on that thread hogs that single thread and prevents user interaction (for more than 5 seconds), it causes Android to throw up the infamous "Application Not Responding" (ANR) error.



How to Avoid ANRs

- Android applications normally run entirely on a single thread by default the "UI thread" or "main thread").
- This means anything your application is doing in the UI thread that takes a long time to complete can trigger the ANR dialog because your application is not giving itself a chance to handle the input event or intent broadcasts.

- The most effective way to create a worker thread for longer operations is with the AsyncTask class. Simply extend AsyncTask and implement thedoInBackground() method to perform the work.
- To post progress changes to the user, you can call publishProgress(), which invokes theonProgressUpdate() callback method.
- From your implementation of onProgressUpdate() (which runs on the UI thread), you can notify the user.

Android AsyncTask

- Android AsyncTask is an abstract class provided by Android which gives us the liberty to perform heavy tasks in the background and keep the UI thread light thus making the application more responsive.
- Android application runs on a single thread when launched. Due to this single thread model tasks that take longer time to fetch the response can make the application non-responsive.
- To avoid this we use android AsyncTask to perform the heavy tasks in background on a dedicated thread and passing the results back to the UI thread.
- Hence use of AsyncTask in android application keeps the UI thread responsive at all times.

The basic methods used in an android AsyncTask class are defined below :

- **doInBackground() :** This method contains the code which needs to be executed in background. In this method we can send results multiple times to the UI thread by **publishProgress()** method. To notify that the background processing has been completed we just need to use the return statements
- **onPreExecute() :** This method contains the code which is executed before the background processing starts
- **onPostExecute() :** This method is called after doInBackground method completes processing. Result from doInBackground is passed to this method
- **onProgressUpdate() :** This method receives progress updates from doInBackground method, which is published via publishProgress method, and this method can use this progress update to update the UI thread

The three generic types used in an android AsyncTask class are given below :

- **Params :** The type of the parameters sent to the task upon execution
- **Progress :** The type of the progress units published during the background computation
- **Result :** The type of the result of the background computation

Android AsyncTask Example

To start an AsyncTask the following snippet must be present in the MainActivity class :

```
MyTask myTask = new MyTask();

myTask.execute();
```

- MyTask is a sample class, it extends from AsyncTask & execute method is used to start the background thread.

Note :

The AsyncTask instance must be created and invoked in the UI thread.

- The methods overridden in the AsyncTask class should never be called.
- They're called automatically **(Callback methods)**
- AsyncTask can be called only once. Executing it again will throw an exception

❖ Android Services

- In Android, a Service is an application component that can perform long-running operations in the background on the UI thread.
- By background, it means that it doesn't have a user interface.
- A Service runs on the main thread of the calling Component's process by default (and hence can degrade responsiveness and cause ANRs), hence you should create a new Thread to perform long running operations.
- A Service can also be made to run in a completely different process.
- Unlike Activity components, Services do not have any graphical interfaces. Also Broadcast Receivers are for receiving broadcast messages and perform short tasks whereas Services are meant to do lengthy processing like streaming music, network transactions, file I/O, interact with databases, etc.
- When a Service is started by an application component like an Activity it runs in the background and keeps running even if the user switches to another application or the starting component is itself destroyed.

Started Service

- Started services are those that are launched by other application components like an Activity or a Broadcast Receiver.

- They can run indefinitely in the background until stopped or destroyed by the system to free up resources.

```
public class TestService extends Service {

   private String TAG = "TestService";

   public TestService() {
   }

   @Override
   public void onCreate() {
      Log.d(TAG, "onCreate called");
   }

   @Override
   public int onStartCommand(Intent intent, int flags, int startId) {
      Log.d(TAG, "onStartCommand executed");
      return Service.START_NOT_STICKY;
   }

   @Override
   public IBinder onBind(Intent intent) {
      // TODO: Return the communication channel to the service.
      return null;
   }

}
```

You'll also need to create an entry in the manifest file inside the application tags:

```
<service  android:name="com.example.app.TestService"/>
```

- Now that we're set with our Service, it's time to see how to trigger its execution.
- Here's how a Service (from let's say an Activity or a Broadcast Receiver component) is started:

```
// Create an Explicit Intent

Intent intent = new Intent(this, TestService.class);

// Start the Service

startService(intent);
```

- If the Service is not yet running, then startService() triggers the onCreate() method of the Service first.
- Once the Service is started (or it is already running), the onStartCommand() is called with the Intent object passed to startService().
- So basically a Service is started only once no matter how often you call startService() (delivered/processed sequentially).
- Calls to startService() are non-blocking even if some processing has to be done in the Service class.

Let's examine the 3 different arguments passed to onStartCommand():

- Intent – Data to be used by the Service for any sort of asynchronous execution. For example you can pass an image URL that must be downloaded.
- Flag – Representing the history of this start request. We will discuss this in a bit.
- Start ID – A unique ID provided by the runtime for this start request. If the process is terminated and then at a later stage restarted then onStartCommand() will be called with the same start ID. It can be used with stopSelfResult() which is similar to stopService() but this way you can avoid stopping the Service for old start requests that have not been handled by onStartCommand() yet because maybe during that start request the Service terminated and restarted.

Binding to a service

- Application components (clients) can bind to a service by calling bindService().
- The Android system then calls the service's onBind() method, which returns an IBinder for interacting with the service.
- The binding is asynchronous, and bindService() returns immediately *without* returning the IBinder to the client.
- To receive the IBinder, the client must create an instance of ServiceConnection and pass it to bindService(). The ServiceConnection includes a callback method that the system calls to deliver the IBinder.

Note: Only activities, services, and content providers can bind to a service—you **can't** bind to a service from a broadcast receiver.

To bind to a service from your client, follow these steps:

1. Implement ServiceConnection.

   Your implementation must override two callback methods:

onServiceConnected()

The system calls this to deliver the IBinder returned by the service's onBind() method.

onServiceDisconnected()

The Android system calls this when the connection to the service is unexpectedly lost, such as when the service has crashed or has been killed. This is *not* called when the client unbinds.

2. Call bindService(), passing the ServiceConnection implementation.

   Note: If the method returns false, your client does not have a valid connection to the service. However, your client should still call unbindService(); otherwise, your client will keep the service from shutting down when it is idle.

3. When the system calls your onServiceConnected() callback method, you can begin making calls to the service, using the methods defined by the interface.

4. To disconnect from the service, call unbindService().

   If your client is still bound to a service when your app destroys the client, destruction causes the client to unbind. It is better practice to unbind the client as soon as it is done interacting with the service. Doing so allows the idle service to shut down. For more information about appropriate times to bind and unbind, see Additional notes.

The following example connects the client to the service created above by extending the Binder class, so all it must do is cast the returned IBinder to the LocalService class and request the LocalService instance:

```
LocalService mService;
private ServiceConnection mConnection = new ServiceConnection() {
   // Called when the connection with the service is established
   public void onServiceConnected(ComponentName className, IBinder service) {
      // Because we have bound to an explicit
      // service that is running in our own process, we can
      // cast its IBinder to a concrete class and directly access it.
      LocalBinder binder = (LocalBinder) service;
      mService = binder.getService();
      mBound = true;
   }

   // Called when the connection with the service disconnects unexpectedly
   public void onServiceDisconnected(ComponentName className) {
      Log.e(TAG, "onServiceDisconnected");
      mBound = false;
```

```
    }
};
```

- With this ServiceConnection, the client can bind to a service by passing it to bindService(), as shown in the following example:

```
Intent intent = new Intent(this, LocalService.class);
bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
```

- The first parameter of bindService() is an Intent that explicitly names the service to bind.

  Caution: If you use an intent to bind to a Service, ensure that your app is secure by using an explicit intent. Using an implicit intent to start a service is a security hazard because you can't be certain what service will respond to the intent, and the user can't see which service starts. Beginning with Android 5.0 (API level 21), the system throws an exception if you call bindService() with an implicit intent.

- The second parameter is the ServiceConnection object.
- The third parameter is a flag indicating options for the binding. It should usually be BIND_AUTO_CREATE in order to create the service if it's not already alive. Other possible values are BIND_DEBUG_UNBIND and BIND_NOT_FOREGROUND, or 0 for none.

IntentService

- Android provides us with IntentService (extends the Service class) that has all the properties of a Service's lifecycle with increased process rank and at the same time allows processing tasks on a background (worker) thread.
- So using an IntentService you can run long-running operations in the background without affecting the app's responsiveness and at the same time one of the biggest advantage of it is that, it isn't affected by most of the user interface (Activity/Fragment) lifecycle events, hence continues to run even when the Activity is destroyed for instance.

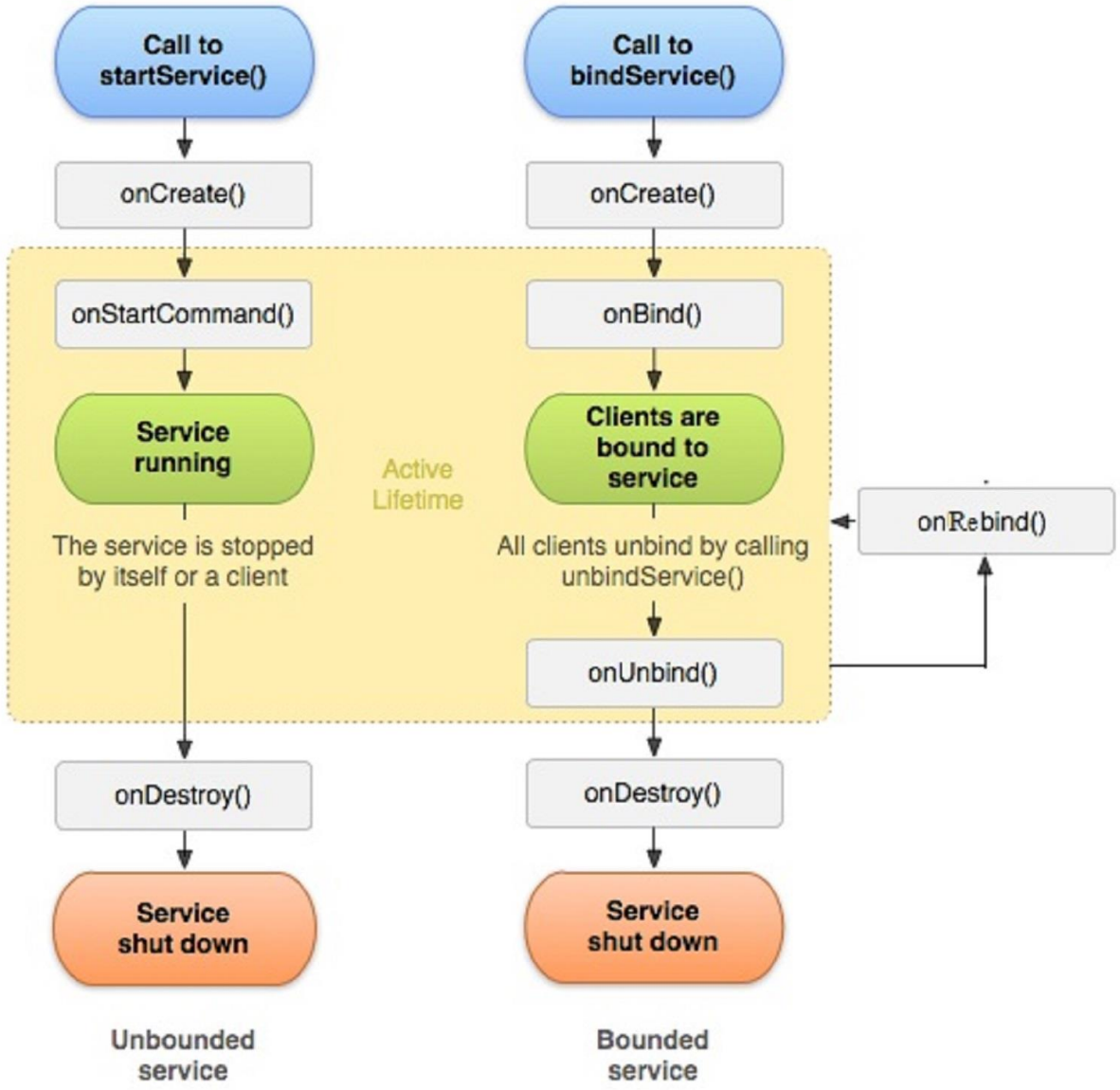
```
public class MyIntentService extends IntentService {

  public MyIntentService(String name) {
    // Used to name the worker thread
    super("MyIntentService");
  }
  @Override
  protected void onHandleIntent(Intent intent) {
    // Do some work in background without affecting the UI thread
  }
}
```

- As you can see, we only overrode onHandleIntent() which is called on the background thread and this is where all your long running operations have to be executed.

- Other callbacks of a regular Service component like onStartCommand() are automatically invoked internally by IntentService so you should avoid overriding them.
- As we're done with the creation and implementation part, now we'll see how to actually send work requests to it.
- Client components (Activity, Fragment, BroadcastReceiver, etc.) that wish to send a start request to the IntentService will have to create an Intent and pass it to Context.startService().

```
Intent intent = new Intent(this, MyIntentService.class);
startService(intent);
```

- You won't have to stop the IntentService as it calls stopSelf() internally.
- Under the hood, an IntentService makes use of a HandlerThread to execute tasks in background and not an AsyncTask.



❖ Android Fragment

- ✓ Fragment class in Android is used to build dynamic User Interfaces.
- ✓ **Fragment** should be used within the Activity.
- ✓ A greatest advantage of fragments is that it simplifies the task of creating UI for multiple screen sizes.
- ✓ A activity can contain any number of fragments.
- ✓ An Android fragment is not by itself a subclass of View which most other UI components are. Instead, a fragment has a view inside it.
- ✓ It is this view which is eventually displayed inside the activity in which the fragment lives.
- ✓ Because an android fragment is not a view, adding it to an activity looks somewhat different than adding a view (e.g. TextView).
- ✓ A fragment is added to a **ViewGroup** inside the activity.
- ✓ The fragment's view is displayed inside this ViewGroup.

❖ Fragment Lifecycle

Below are the methods of fragment lifecycle.

1. onAttach() :This method will be called first, even before onCreate(), letting us know that your fragment has been attached to an activity. You are passed the Activity that will host your fragment

2. onCreateView() : The system calls this callback when it's time for the fragment to draw its UI for the first time. To draw a UI for the fragment, a View component must be returned from this method which is the root of the fragment's layout. We can return null if the fragment does not provide a UI

3. onViewCreated() : This will be called after onCreateView(). This is particularly useful when inheriting the onCreateView() implementation but we need to configure the resulting views, such as with a ListFragment and when to set up an adapter

4. onActivityCreated() :This will be called after onCreate() and onCreateView(), to indicate that the activity's onCreate() has completed. If there is something that's needed to be initialised in the fragment that depends upon the activity's onCreate() having completed its work then onActivityCreated() can be used for that initialisation work

5. onStart() : The onStart() method is called once the fragment gets visible

6. onPause() : The system calls this method as the first indication that the user is leaving the fragment. This is usually where you should commit any changes that should be persisted beyond the current user session

7. onStop() : Fragment going to be stopped by calling onStop()

8. onDestroyView() : It's called before onDestroy(). This is the counterpart to onCreateView() where we set up the UI. If there are things that are needed to be cleaned up specific to the UI, then that logic can be put up in onDestroyView()

9. onDestroy() : onDestroy() called to do final clean up of the fragment's state but Not guaranteed to be called by the Android platform.

10. onDetach() : It's called after onDestroy(), to notify that the fragment has been disassociated from its hosting activity

Android fragment lifecycle is illustrated in below image.

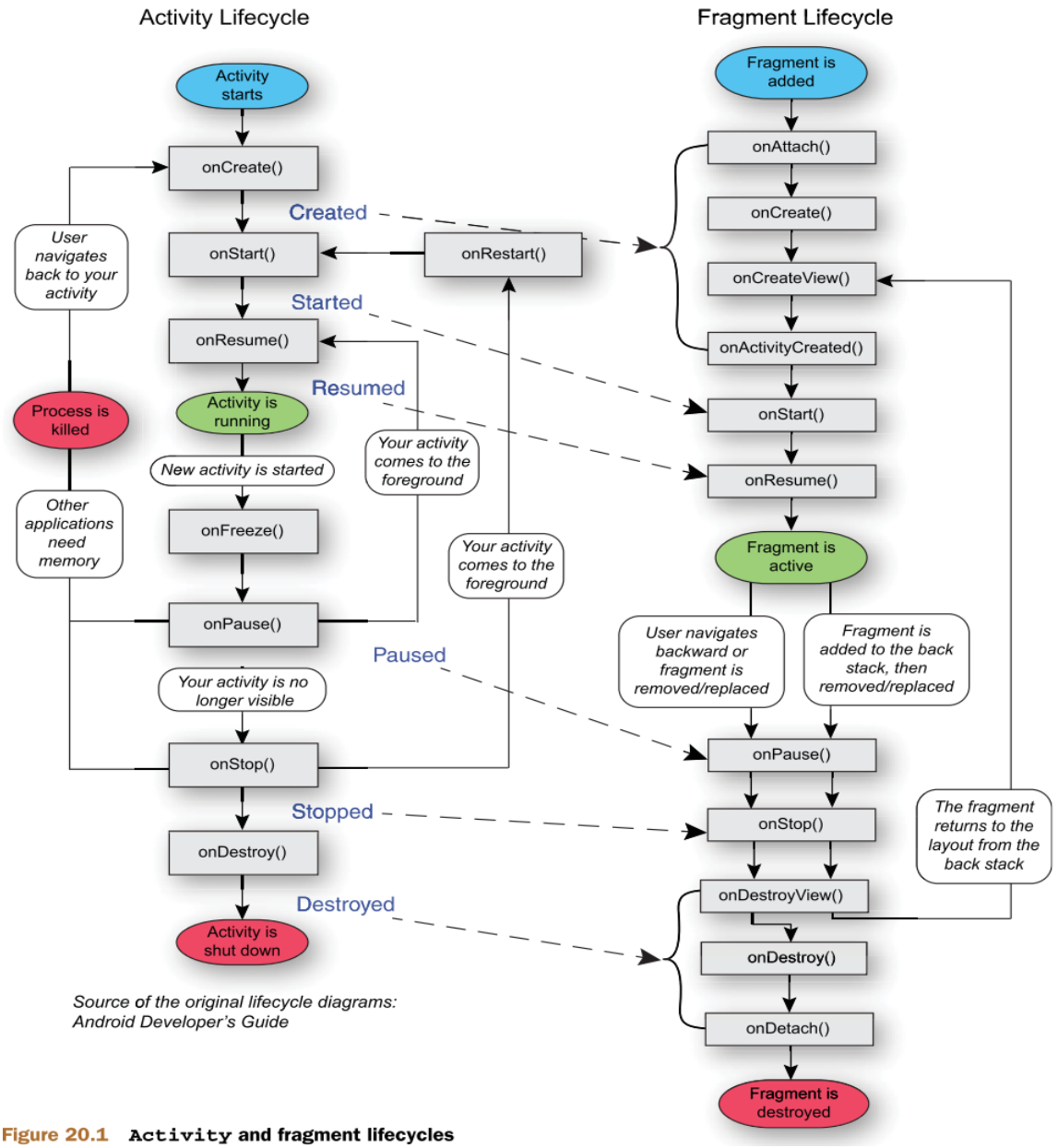


Figure 20.1 **`Activity` and fragment lifecycles**