Introduction to ANDROID

Introduction to Android

- Objectives :
  - What Is Mobile Computing
  - Limitations in Mobile Computing
  - What is Android
  - Android Architecture
  - Android Components

Challenges in Smart Phone Programming

- Screens are small.
- Keyboards, if they exist, are small.
- Pointing devices, if they exist, are annoying (large fingers and “multitouch” LCDs are not a good mix).
- CPU speed and memory are limited compared with what are available on desktops and servers.
- You can have any programming language and development framework you want, as long as it is supported by the device manufacturer.

Challenges in Smart Phone Programming

- While programming mobile device you have to be careful that you are not breaking the phone by
  - By tying up the CPU so that calls can’t be received.
  - By crashing the phone’s operating system, by leaking lots of memory.
  - By not quietly fading into the background when calls come or need to be made.

What Android tries to do is meet you halfway

- You get a commonly used programming language (Java) with some commonly used libraries (e.g., some Apache Commons APIs), with support for tools you may be used to using (Eclipse).
- You get a fairly rigid and separate framework in which your programs need to run so they can be “good citizens” on the phone and not interfere with other programs or the operation of the phone itself.

What is Android

- Android is a software stack for mobile devices that includes an operating system, middleware and key applications.
- The Android OS is a Linux based operating system.
- The Android SDK provides the tools and APIs necessary to begin developing applications on the Android platform using the Java programming language.
- The Android Platform embraces the idea of general-purpose computing for handheld devices.
- Android's libraries cover telephony, video, graphics, UI programming, and a number of other aspects of the device.

What is Android?

- The Android SDK supports most of the Java SE except for the AWT and Swing.
- Android SDK has its own extensive modern UI framework.
- Android offers its own optimized JVM to run the compiled Java class files in order to counter the handheld device limitations such as memory, processor speed, and power.
- This virtual machine is called the Dalvik VM.

Android Features

- **Application framework** enabling reuse and replacement of components
- **Dalvik virtual machine** optimized for mobile devices
- **Integrated browser** based on the open source WebKit engine
- **Optimized graphics** powered by a custom 2D graphics library; 3D graphics based on the OpenGL ES 1.0 specification (hardware acceleration optional)
- **SQLite** for structured data storage
- **Media** support for common audio, video, and still image formats (MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF)

Android Features

- **GSM Telephony** (hardware dependent)
- **Bluetooth, EDGE, 3G, and WiFi** (hardware dependent)
- **Camera, GPS, compass, and accelerometer** (hardware dependent)
- **Rich development environment** including a device emulator, tools for debugging, memory and performance profiling, and a plugin for the Eclipse IDE

Dalvik VM

- Every Android application runs in its own process, with its own instance of the Dalvik virtual machine.
- Dalvik has been written so that a device can run multiple VMs efficiently.
- The Dalvik VM relies on the Linux kernel for underlying functionality such as threading and low-level memory management.

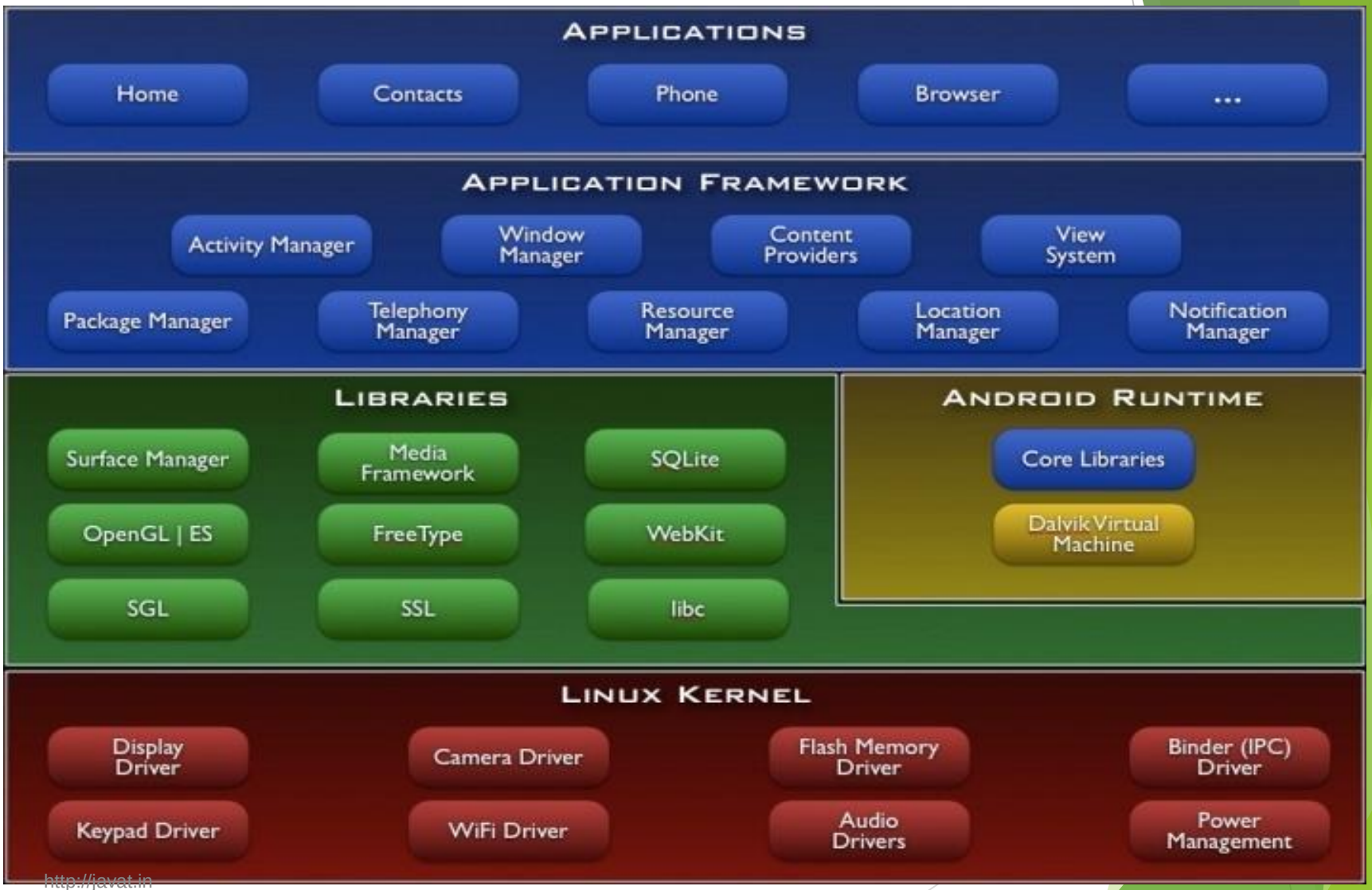
What Dalvik VM does?

- The Dalvik VM takes the generated Java class files and combines them into one or more Dalvik Executable (.dex) files. It reuses duplicate information from multiple class files, effectively reducing the space requirement by half from a traditional .jar file.
- Google has fine-tuned the garbage collection in the Dalvik VM, but it has chosen to omit a just-in-time (JIT) compiler, in early releases. It is anticipated that it will be part of future releases. Google reasoned that the impact of JIT compilation would not be significant.
- Dalvik VM uses a different kind of assembly-code generation, in which it uses registers as the primary units of data storage instead of the stack. Google is hoping to accomplish 30 percent fewer instructions as a result.

Linux Kernel

- Android relies on Linux version 2.6 for core system services such as security, memory management, process management, network stack, and driver model.
- The kernel acts as an abstraction layer between the hardware and the rest of the software stack.

Android Architecture



Android Architecture

- At the core of the Android Platform is Linux kernel version 2.6.29, responsible for device drivers, resource access, power management, and other OS duties.
- The supplied device drivers include Display, Camera, Keypad, WiFi, Flash Memory, Audio, and IPC (inter process communication).
- At the next level, on top of the kernel, are a number of C/C++ libraries such as OpenGL, WebKit, FreeType, Secure Sockets Layer (SSL), the C runtime library (libc), SQLite, and Media.

Android Libraries

- **System C library -** a BSD-derived implementation of the standard C system library (libc), tuned for embedded Linux-based devices
- **Media Libraries -** based on PacketVideo's OpenCORE; the libraries support playback and recording of many popular audio and video formats, as well as static image files, including MPEG4, H.264, MP3, AAC, AMR, JPG, and PNG
- **Surface Manager -** manages access to the display subsystem and seamlessly composites 2D and 3D graphic layers from multiple applications

Android Libraries

- **LibWebCore** - a modern web browser engine which powers both the Android browser and an embeddable web view
- **SGL** - the underlying 2D graphics engine
- **3D libraries** - an implementation based on OpenGL ES 1.0 APIs; the libraries use either hardware 3D acceleration (where available) or the included, highly optimized 3D software rasterizer
- **FreeType** - bitmap and vector font rendering
- **SQLite** - a powerful and lightweight relational database engine available to all applications

Android Application Framework

- Each process has its own Java virtual machine (VM)
- Each application is assigned a unique Linux user ID.
  It is possible to have two processes share the same user ID.
  In that case the two processes can see each other's files.
- An Android application can make use of elements of other application, by starting up executing the piece of the application in which it is interested.
- Therefore, unlike applications on most other systems, Android applications don't have a single entry point for everything in the application (no main() function, for example). Rather, they have essential *components* that the system can instantiate and run as needed.

Android Components

- Activities
- Services
- Broadcast receivers
- Content providers

Android Components: **Activities**

- The building block of the user interface is the activity.
- While it is possible for activities to not have a user interface, most likely your “headless” code will be packaged in the form of content providers or services.
- Even if the activities work together, they are independent of each other.
- **Each Activity is implemented as the derived class of Activity class.**
- An application may have one or any number of activities. One of the activities is marked as the first one that should be presented when the application starts. First Activity invoked can in turn invoke other activities.

Android Components: **Activities**

- The visual content of the window of an activity is provided by a hierarchy of views — objects derived from the base View class. Each view controls a particular rectangular space within the window.
- Parent views contain and organize the layout of their children. Leaf views (those at the bottom of the hierarchy) draw in the rectangles they control and respond to user actions directed at that space.
- Android has a number of ready-made views that you can use — including buttons, text fields, scroll bars, menu items, check boxes, and more.
- A view hierarchy is placed within an activity's window by the Activity.setContentView() method.

Android Components: **Services**

- A *service* doesn't have a visual user interface, but rather runs in the background for an indefinite period of time.
- Each service extends the Service base class. For example, a service might play background music as the user attends to other matters, or it might fetch data over the network or calculate something and provide the result to activities that need it.
- It's possible to connect to (bind to) an ongoing service (and start the service if it's not already running). While connected, you can communicate with the service through an interface that the service exposes. Like activities and the other components, services run in the main thread of the application process.

Android components: **Broadcast receivers**

- A *broadcast receiver* is a component that receives and react to broadcast announcements.
- 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.
- An application can have any number of broadcast receivers to respond to any announcements it considers important. All receivers extend the BroadcastReceiver base class.
- Broadcast receivers do not display a user interface. However, they may start an activity in response to the information they receive, or they may use the NotificationManager to alert the user.

Android components: **Content providers**

- A *content provider* makes a specific set of the application's data available to other applications.
- The data can be stored in the file system, in an SQLite database, or in any other manner that makes sense.
- The content provider extends the ContentProvider base class to implement a standard set of methods that enable other applications to retrieve and store data of the type it controls.
- However, applications do not call these methods directly. Rather they use a ContentResolver object and call its methods instead.
- A ContentResolver can talk to any content provider; it cooperates with the provider to manage any interprocess communication that's involved.

Activating components: **intents**

- Content providers are activated when they're targeted by a request from a ContentResolver. The other three components — activities, services, and broadcast receivers — are activated by asynchronous messages called *intents*.
- An intent is an Intent object that holds the content of the message. For activities and services, it names the action being requested and specifies the URI of the data to act on, among other things.
- For example, it might convey a request for an activity to present an image to the user or let the user edit some text.
- For broadcast receivers, the Intent object names the action being announced. For example, it might announce to interested parties that the camera button has been pressed.

Activating components: **intents (Activating Activities)**

- There are separate methods for activating each type of component:
- An activity is launched by passing an Intent object to Context.startActivity() or Activity.startActivityForResult().
- The responding activity can look at the initial intent that caused it to be launched by calling its getIntent() method.
- Android calls the activity's onNewIntent() method to pass it any subsequent intents.
- One activity often starts the next one. If it expects a result back from the activity it's starting, it calls startActivityForResult() instead of startActivity().

Activating components: **intents (Activating Activities)**

- For example, if it starts an activity that lets the user pick a photo, it might expect to be returned the chosen photo. The result is returned in an Intent object that's passed to the calling activity's onActivityResult() method.

Activating components: **intents (Activating Service)**

- A service is started or new instructions are given to an ongoing service by passing an Intent object to Context.startService().
- Android calls the service's onStart() method and passes it the Intent object.
- Similarly, an intent can be passed to Context.bindService() to establish an ongoing connection between the calling component and a target service.
- The service receives the Intent object in an onBind() call. (If the service is not already running, bindService() can optionally start it.)

Activating components: **intents (Activating Service)**

- For example, an activity might establish a connection with the music playback service mentioned earlier so that it can provide the user with the means (a user interface) for controlling the playback. The activity would call bindService() to set up that connection, and then call methods defined by the service to affect the playback.

Activating components: **intents (Activating Broadcast receiver)**

- An application can initiate a broadcast by passing an Intent object to methods like Context.sendBroadcast(), Context.sendOrderedBroadcast(), and Context.sendStickyBroadcast() in any of their variations.
- Android delivers the intent to all interested broadcast receivers by calling their onReceive() methods.

Shutting down components

- A content provider is active only while it's responding to a request from a ContentResolver.
  And a broadcast receiver is active only while it's responding to a broadcast message. So there's no need to explicitly shut down these components.
- Activities, on the other hand, provide the user interface. They're in a long-running conversation with the user and may remain active, even when idle, as long as the conversation continues. Similarly, services may also remain running for a long time. So Android has methods to shut down activities and services in an orderly way:
- An activity can be shut down by calling its finish() method. One activity can shut down another activity (one it started with startActivityForResult()) by calling finishActivity().

Shutting down components

- A service can be stopped by calling its stopSelf() method, or by calling Context.stopService().
- Components might also be shut down by the system when they are no longer being used or when Android must reclaim memory for more active components.

manifest file

- Applications declare their components in a manifest file that's bundled into the Android package, the .apk file that also holds the application's code, files, and resources.
- The manifest is a structured XML file and is always named AndroidManifest.xml for all applications.
- It is also used for naming any libraries the application needs to be linked against (besides the default Android library) and identifying any permissions the application expects to be granted.

manifest file

```
<?xml version="1.0" encoding="utf-8"?>
  <manifest . . . >
     <application . . . >
        <activity
  android:name="com.example.project.FreneticActivity"
                android:icon="@drawable/small_pic.png"
                android:label="@string/freneticLabel"
                . . .  >
        </activity>
        . . .
     </application>
  </manifest>
```

manifest file

- The name attribute of the <activity> element names the Activity subclass that implements the activity. The icon and label attributes point to resource files containing an icon and label that can be displayed to users to represent the activity.
- The other components are declared in a similar way — <service> elements for services, <receiver> elements for broadcast receivers, and <provider> elements for content providers.
- Activities, services, and content providers that are not declared in the manifest are not visible to the system and are consequently never run.

manifest file

- Broadcast receivers can either be declared in the manifest, or they can be created dynamically in code (as BroadcastReceiver objects) and registered with the system by calling Context.registerReceiver().

Intent filters

- An Intent object can explicitly name a target component. If it does, Android finds that component (based on the declarations in the manifest file) and activates it. But if a target is not explicitly named, Android must locate the best component to respond to the intent.
- It does so by comparing the Intent object to the *intent filters* of potential targets.
- A component's intent filters inform Android of the kinds of intents the component is able to handle. It is declared in manifest file.

Intent filters

```
<?xml version="1.0" encoding="utf-8"?>
    <manifest . . . >
        <application . . . >
            <activity android:name="com.example.project.FreneticActivity"
                    android:icon="@drawable/small_pic.png"
                    android:label="@string/freneticLabel"
                    . . .  >
                <intent-filter . . . >
                    <action android:name="android.intent.action.MAIN" />
                    <category android:name="android.intent.category.LAUNCHER" />
                </intent-filter>
                <intent-filter . . . >
                    <action android:name="com.example.project.BOUNCE" />
                    <data android:mimeType="image/jpeg" />
                    <category android:name="android.intent.category.DEFAULT" />
                </intent-filter>
            </activity>
            . . .
        </application>
    </manifest>
```

Intent filters

- The first filter in the example, the combination of the action "android.intent.action.MAIN" and the category "android.intent.category.LAUNCHER“ indicates that the activity is the entry point for the application.
- The second filter declares an action that the activity can perform on a particular type of data.
- A component can have any number of intent filters, each one declaring a different set of capabilities. If it doesn't have any filters, it can be activated only by intents that explicitly name the component as the target.
- For a broadcast receiver that's created and registered in code, the intent filter is instantiated directly as an IntentFilter object.

Activities & Tasks

- A task is what the user experiences as an "application." It's a group of related activities, arranged in a stack. The root activity in the stack is the one that began the task.
- The activity at the top of the stack is one that's currently running — the one that is the focus for user actions.
- When one activity starts another, the new activity is pushed on the stack; it becomes the running activity. The previous activity remains in the stack. When the user presses the BACK key, the current activity is popped from the stack, and the previous one resumes as the running activity.
- Activities in the stack are never rearranged, only pushed and popped.

Activities & Tasks

- A task is a stack of activities, not a class or an element in the manifest file. So there's no way to set values for a task independently of its activities. Values for the task as a whole are set in the root activity.
- All the activities in a task move together as a unit. The entire task (the entire activity stack) can be brought to the foreground or sent to the background.
- The association of activities with tasks, and the behavior of an activity within a task, is controlled by the interaction between flags set in the Intent object that started the activity and attributes set in the activity's <activity> element in the manifest.

Activities & tasks

- In this regard, the principal Intent flags are
  - FLAG_ACTIVITY_NEW_TASK
  - FLAG_ACTIVITY_CLEAR_TOP
  - FLAG_ACTIVITY_RESET_TASK_IF_NEEDED
  - FLAG_ACTIVITY_SINGLE_TOP
- The principal <activity> attributes are:
  - taskAffinity
  - launchMode
  - allowTaskReparenting
  - clearTaskOnLaunch
  - alwaysRetainTaskState
  - finishOnTaskLaunch

Affinities and new tasks

- By default, all the activities in an application have an *affinity* for each other — that is, there's a preference for them all to belong to the same task.
- However, an individual affinity can be set for each activity with the taskAffinity attribute of the <activity> element.
- Activities defined in different applications can share an affinity, or activities defined in the same application can be assigned different affinities.
- The affinity comes into play in two circumstances: When the Intent object that launches an activity contains the FLAG_ACTIVITY_NEW_TASK flag, and when an activity has its allowTaskReparenting attribute set to "true".

Affinities and new tasks

- The FLAG_ACTIVITY_NEW_TASK flag
  - If the Intent object passed to startActivity() contains the FLAG_ACTIVITY_NEW_TASK flag, the system looks for a different task to house the new activity.
  - If there's already an existing task with the same affinity as the new activity, the activity is launched into that task. If not, it begins a new task.
- The allowTaskReparenting attribute
  - If an activity has its allowTaskReparenting attribute set to "true", it can move from the task it starts in to the task it has an affinity for when that task comes to the fore.

Launch modes

- There are four different launch modes that can be assigned to an <activity> element's launchMode attribute:
  - "standard" (the default mode)
  - "singleTop"
  - "singleTask"
  - "singleInstance"

Which task will hold the activity that responds to the intent?

- For the "standard" and "singleTop" modes, it's the task that originated the intent (and called startActivity()) — unless the Intent object contains the FLAG_ACTIVITY_NEW_TASK flag.
- In contrast, the "singleTask" and "singleInstance" modes mark activities that are always at the root of a task. They define a task; they're never launched into another task.

Whether there can be multiple instances of the activity?

- A "standard" or "singleTop" activity can be instantiated many times. They can belong to multiple tasks, and a given task can have multiple instances of the same activity.
- In contrast, "singleTask" and "singleInstance" activities are limited to just one instance. Since these activities are at the root of a task, this limitation means that there is never more than a single instance of the task on the device at one time.

Whether the instance can have other activities in its task?

- A "singleInstance" activity stands alone as the only activity in its task. If it starts another activity, that activity will be launched into a different task regardless of its launch mode — as if FLAG_ACTIVITY_NEW_TASK was in the intent.
- In all other respects, the "singleInstance" mode is identical to "singleTask".
- The other three modes permit multiple activities to belong to the task.
- A "singleTask" activity will always be the root activity of the task, but it can start other activities that will be assigned to its task.
- Instances of "standard" and "singleTop" activities can appear anywhere in a stack.

Whether a new instance of the class will be launched to handle a new intent?

- For the default "standard" mode, a new instance is created to respond to every new intent. Each instance handles just one intent.
- For the "singleTop" mode, an existing instance of the class is re-used to handle a new intent if it resides at the top of the activity stack of the target task. If it does not reside at the top, it is not re-used. Instead, a new instance is created for the new intent and pushed on the stack.

Launch Modes

- Note that when a new instance of an Activity is created to handle a new intent, the user can always press the BACK key to return to the previous state (to the previous activity).
- But when an existing instance of an Activity handles a new intent, the user cannot press the BACK key to return to what that instance was doing before the new intent arrived.

Clearing the stack

- If the user leaves a task for a long time, the system clears the task of all activities except the root activity. When the user returns to the task again, it's as the user left it, except that only the initial activity is present.
- That's the default. There are some activity attributes that can be used to control this behavior and modify it:
- The alwaysRetainTaskState attribute If this attribute is set to "true" in the root activity of a task, the default behavior just described does not happen. The task retains all activities in its stack even after a long period.

Clearing the stack

- The clearTaskOnLaunch attribute If this attribute is set to "true" in the root activity of a task, the stack is cleared down to the root activity whenever the user leaves the task and returns to it. In other words, it's the polar opposite of alwaysRetainTaskState. The user always returns to the task in its initial state, even after a momentary absence.
- The finishOnTaskLaunch attribute
  This attribute is like clearTaskOnLaunch, but it operates on a single activity, not an entire task. And it can cause any activity to go away, including the root activity.
  When it's set to "true", the activity remains part of the task only for the current session. If the user leaves and then returns to the task, it no longer is present.

Clearing the Stack

- There's another way to force activities to be removed from the stack. If an Intent object includes the FLAG_ACTIVITY_CLEAR_TOP flag, and the target task already has an instance of the type of activity that should handle the intent in its stack, all activities above that instance are cleared away so that it stands at the top of the stack and can respond to the intent.
- If the launch mode of the designated activity is "standard", it too will be removed from the stack, and a new instance will be launched to handle the incoming intent. That's because a new instance is always created for a new intent when the launch mode is "standard".

Clearing the Stack

- FLAG_ACTIVITY_CLEAR_TOP is most often used in conjunction with FLAG_ACTIVITY_NEW_TASK. When used together, these flags are a way of locating an existing activity in another task and putting it in a position where it can respond to the intent.

Starting tasks

- An activity is set up as the entry point for a task by giving it an intent filter with "android.intent.action.MAIN" as the specified action and "android.intent.category.LAUNCHER" as the specified category.
- This second ability is important: Users must be able to leave a task and then come back to it later. For this reason, the two launch modes that mark activities as always initiating a task, "singleTask" and "singleInstance", should be used only when the activity has a MAIN and LAUNCHER filter.

Processes and Threads

- When the first of an application's components needs to be run, Android starts a Linux process for it with a single thread of execution.
- By default, all components of the application run in that process and thread.
- However, you can arrange for components to run in other processes, and you can spawn additional threads for any process.

Processes

- The process where a component runs is controlled by the manifest file. The component elements — <activity>, <service>, <receiver>, and <provider> — each have a process attribute that can specify a process where that component should run.
- They can also be set so that components of different applications run in the same process — provided that the applications share the same Linux user ID and are signed by the same authorities.
- The <application> element also has a process attribute, for setting a default value that applies to all components.

Processes

- All components are instantiated in the main thread of the specified process, and system calls to the component are dispatched from that thread. Separate threads are not created for each instance.
- Android may decide to shut down a process at some point, when memory is low and required by other processes that are more immediately serving the user.
- Application components running in the process are consequently destroyed.
- A process is restarted for those components when there's again work for them to do.

Threads

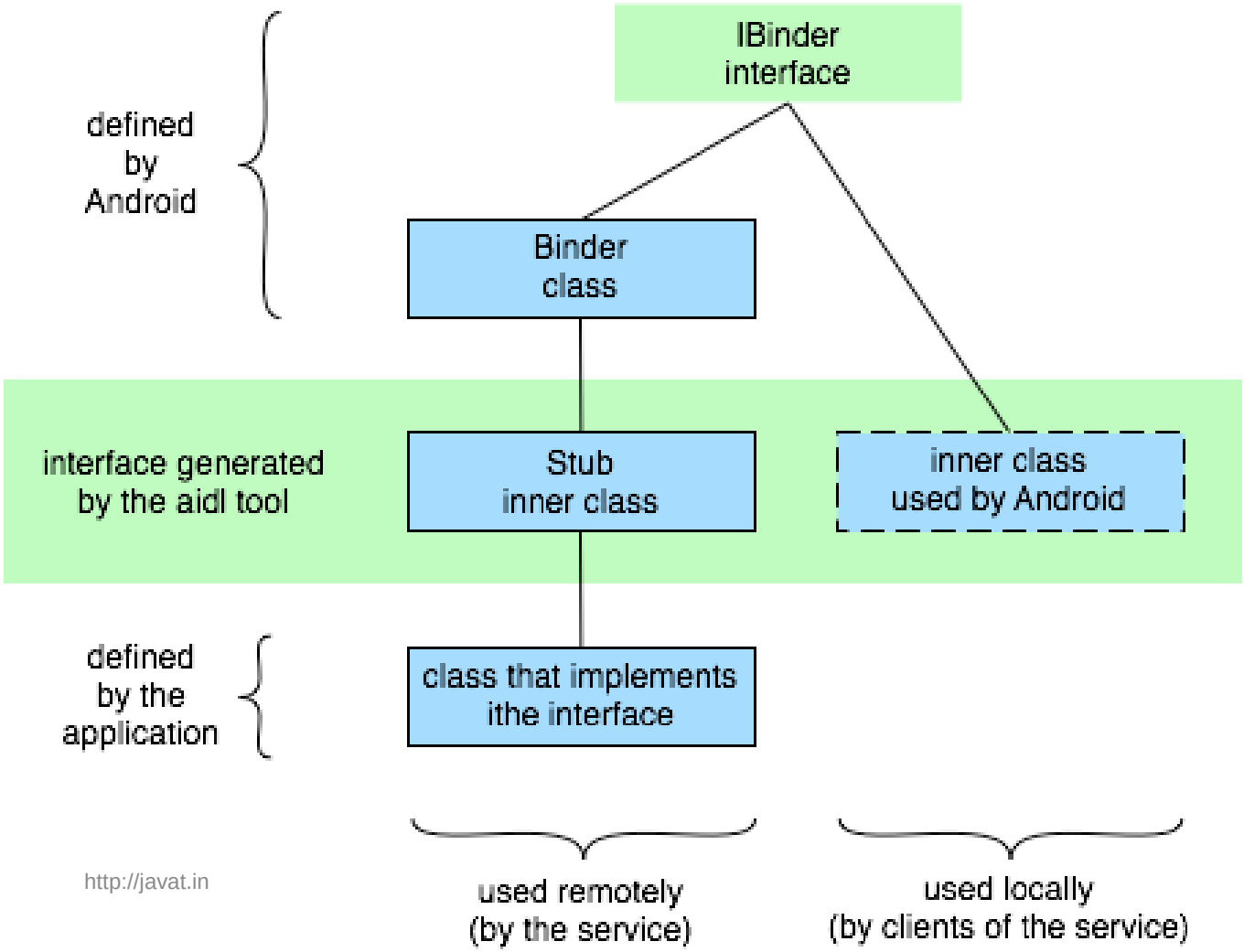
- Threads are created in code using standard Java Thread objects.
- Android provides a number of convenience classes for managing threads — Looper for running a message loop within a thread, Handler for processing messages, and HandlerThread for setting up a thread with a message loop.

Remote procedure calls

- Android has a lightweight mechanism for remote procedure calls (RPCs) — where a method is called locally, but executed remotely (in another process), with any result returned back to the caller.
- Android provides all the code to do that work, so that you can concentrate on defining and implementing the RPC interface itself.
- An RPC interface can include only methods. All methods are executed synchronously (the local method blocks until the remote method finishes), even if there is no return value.

Remote procedure calls

- The mechanism works as follows:



Component Life Cycle

- Application components have a lifecycle — a beginning when Android instantiates them to respond to intents through to an end when the instances are destroyed.
- In between, they may sometimes be active or inactive,or, in the case of activities, visible to the user or invisible.

Activity lifecycle

- An activity has essentially three states:
  - It is *active* or *running* when it is in the foreground of the screen . This is the activity that is the focus for the user's actions.
  - It is *paused* if it has lost focus but is still visible to the user. That is, another activity lies on top of it and that activity either is transparent or doesn't cover the full screen, so some of the paused activity can show through. A paused activity is completely alive (it maintains all state and member information and remains attached to the window manager), but can be killed by the system in extreme low memory situations.

Activity lifecycle

- It is stopped if it is completely obscured by another activity. 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.

- If an activity is paused or stopped, the system can drop it from memory either by asking it to finish (calling its finish() method), or simply killing its process. When it is displayed again to the user, it must be completely restarted and restored to its previous state.

Activity Life Cycle

- As an activity transitions from state to state, it is notified of the change by calls to the following protected methods:
  - void onCreate(Bundle *savedInstanceState*)
  - void onStart()
  - void onRestart()
  - void onResume()
  - void onPause()
  - void onStop()
  - void onDestroy()
- Taken together, these seven methods define the entire lifecycle of an activity.
- All activities must implement onCreate() to do the initial setup when the object is first instantiated.

Activity Life Cycle

- The **entire lifetime** of an activity happens between the first call to onCreate() through to a single final call to onDestroy(). An activity does all its initial setup of "global" state in onCreate(), and releases all remaining resources in onDestroy().
- The **visible lifetime** of an activity happens between a call to onStart() until a corresponding call to onStop().
  For example, you can register a BroadcastReceiver in onStart() to monitor for changes that impact your UI, and unregister it in onStop()
- The **foreground lifetime** of an activity happens between a call to onResume() until a corresponding call to onPause().

Activity Life Cycle

- Three methods (onPause(), onStop(), and onDestroy()) are marked “Killable."
- onPause() is the only one that's guaranteed to be called before the process is killed — onStop() and onDestroy() may not be.

Saving Activity state

- onSaveInstanceState() : To capture that state before the activity is killed
  - Android calls this method before making the activity vulnerable to being destroyed — that is, before onPause() is called.
  - It passes the method a Bundle object where you can record the dynamic state of the activity as name-value pairs.
  - When the activity is again started, the Bundle is passed both to onCreate() and to a method that's called after onStart(), onRestoreInstanceState(), so that either or both of them can recreate the captured state.
  - onSaveInstanceState() and onRestoreInstanceState() are not lifecycle methods.

Saving Activity state

- Android calls onSaveInstanceState() before the activity becomes vulnerable to being destroyed by the system, but does not bother calling it when the instance is actually being destroyed by a user action (such as pressing the BACK key).
- Because onSaveInstanceState() is not always called, you should use it only to record the transient state of the activity, not to store persistent data. Use onPause() for that purpose instead.

Coordinating activities

- When one activity starts another, they both experience lifecycle transitions. One pauses and may stop, while the other starts up.
- The order of lifecycle callbacks is well defined:
  - The current activity's onPause() method is called.
  - Next, the starting activity's onCreate(), onStart(), and onResume() methods are called in sequence.
  - Then, if the starting activity is no longer visible on screen, its onStop() method is called.

Service Life Cycle

- A service can be used in two ways:
  - It can be started and allowed to run until someone stops it or it stops itself.
    - In this mode, it's started by calling Context.startService() and stopped by calling Context.stopService().
    - It can stop itself by calling Service.stopSelf() or Service.stopSelfResult().
  - Clients establish a connection to the Service object and use that connection to call into the service.
    - The connection is established by calling Context.bindService(), and is closed by calling Context.unbindService().
    - Multiple clients can bind to the same service. If the service has not already been launched, bindService() can optionally launch it.

Service Life Cycle

- The two modes are not entirely separate. You can bind to a service that was started with startService().
- stopService() will not actually stop the service until the last binding is closed.
- Service Life Cycle methods are:
  - void onCreate()
  - void onStart(Intent *intent*)
  - void onDestroy()

Service Life Cycle

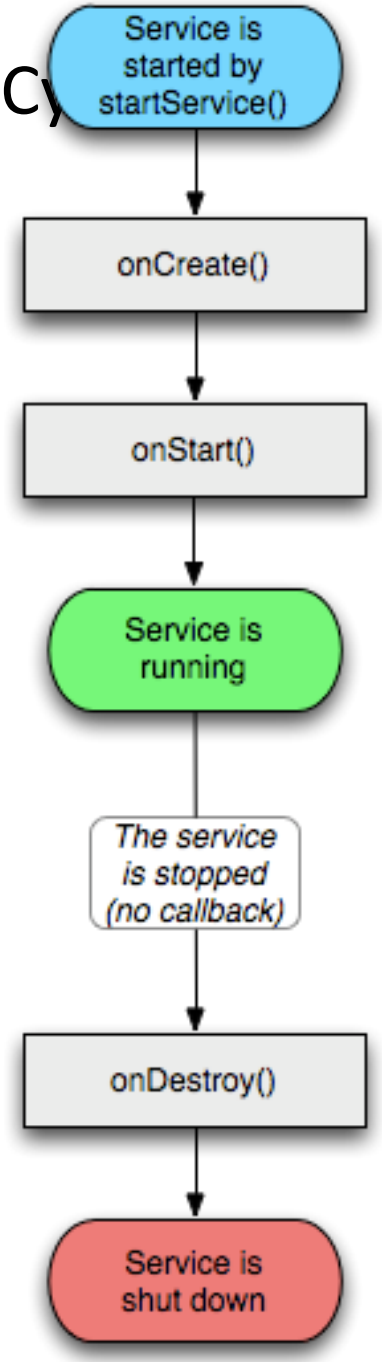
- The **entire lifetime** of a service happens between the time onCreate() is called and the time onDestroy() returns.
  - Like an activity, a service does its initial setup in onCreate(), and releases all remaining resources in onDestroy().
- The **active lifetime** of a service begins with a call to onStart().
  - This method is handed the Intent object that was passed to startService().
- There's no equivalent callback for when the service stops — no onStop() method.
- The onCreate() and onDestroy() methods are called for all services, whether they're started by Context.startService() or Context.bindService(). However, onStart() is called only for services started by startService().

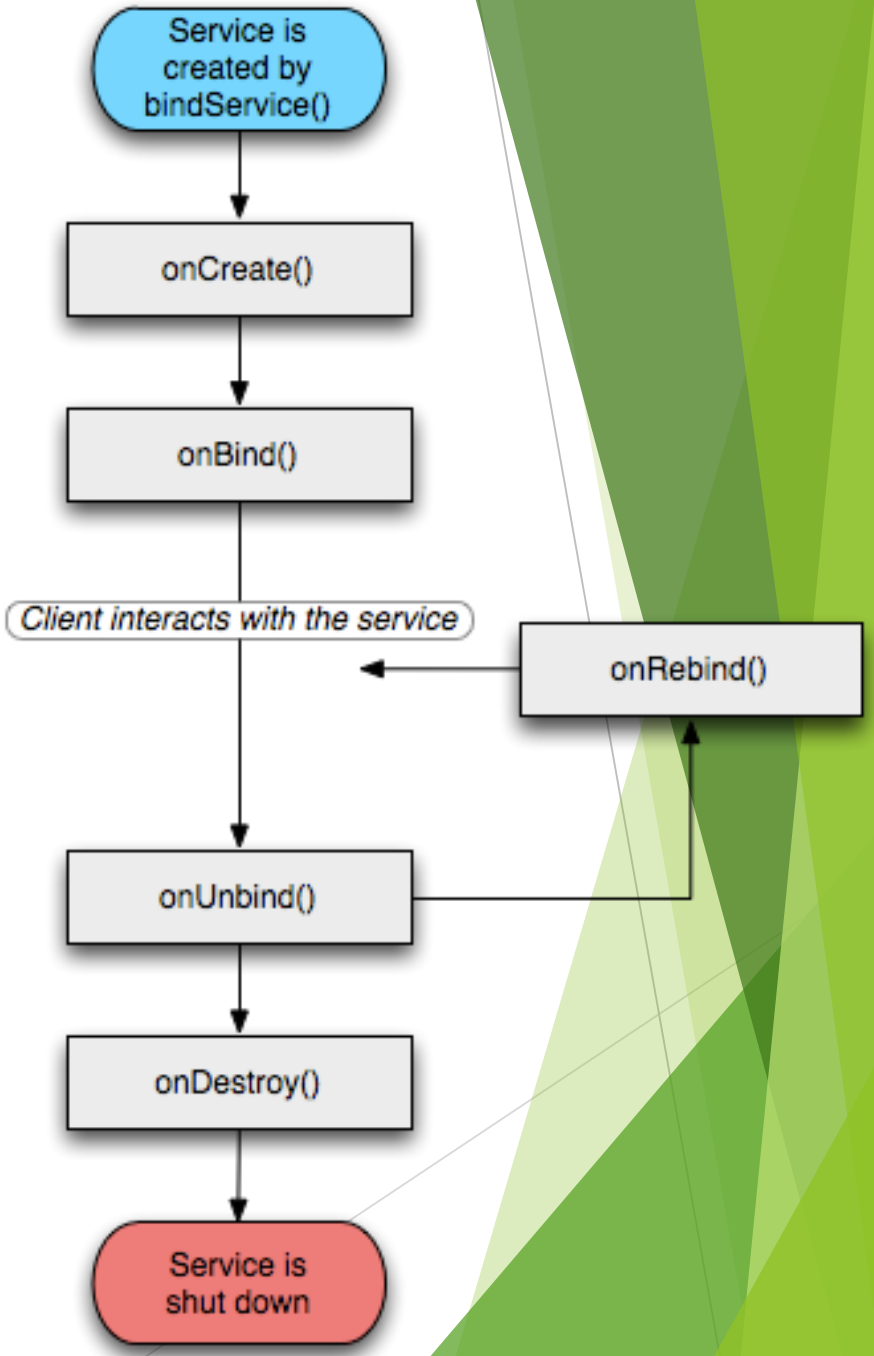
Service Life Cycle

- If a service permits others to bind to it, there are additional callback methods for it to implement:
  - IBinder onBind(Intent *intent*)
  - boolean onUnbind(Intent *intent*)
  - void onRebind(Intent *intent*)
- If the service permits the binding, onBind() returns the communications channel that clients use to interact with the service.
- The onUnbind() method can ask for onRebind() to be called if a new client connects to the service.

Service Life Cy

Any service, no matter how it's started,
can potentially allow clients to bind to it,
so any service may receive onBind()
and onUnbind()
calls.





Broadcast receiver Life Cycle

- A broadcast receiver has single callback method:
  - void onReceive(Context *curContext*, Intent *broadcastMsg*)
- When a broadcast message arrives for the receiver, Android calls its onReceive() method and passes it the Intent object containing the message.
- The broadcast receiver is considered to be active only while it is executing this method. When onReceive() returns, it is inactive.
- A process with an active broadcast receiver is protected from being killed.

Processes and Life Cycles

- To determine which processes to keep and which to kill, Android places each process into an "importance hierarchy" based on the components running in it and the state of those components.
- Processes with the lowest importance are eliminated first, then those with the next lowest, and so on.
- There are five levels in importance hierarchy.
  - Foreground process
  - Visible Process
  - Service
  - Background
  - Empty

Level 1: Foreground Process

- A **foreground process** is one that is required for what the user is currently doing.
- A process is considered to be in the foreground if any of the following conditions hold: It is running an activity that the user is interacting with (the Activity object's onResume() method has been called).
- It hosts a service that's bound to the activity that the user is interacting with.
- It has a Service object that's executing one of its lifecycle callbacks (onCreate(), onStart(), or onDestroy()).
- It has a BroadcastReceiver object that's executing its onReceive() method.
- Only a few foreground processes will exist at any given time. They are killed only as a last resort

Level 2: Visible Process

- A **visible process** is one that doesn't have any foreground components, but still can affect what the user sees on screen. A process is considered to be visible if either of the following conditions holds:
  - It hosts an activity that is not in the foreground, but is still visible to the user (its onPause() method has been called). It hosts a service that's bound to a visible activity.
  - A visible process is considered extremely important and will not be killed unless doing so is required to keep all foreground processes running.

Level 3: Service process

- A **service process** is one that is running a service that has been started with the startService() method and that does not fall into either of the two higher categories.
- Although service processes are not directly tied to anything the user sees, they are generally doing things that the user cares about (such as playing an mp3 in the background or downloading data on the network), so the system keeps them running unless there's not enough memory to retain them along with all foreground and visible processes.

Level 4: Background process

- A **background process** is one holding an activity that's not currently visible to the user (the Activity object's onStop() method has been called).
- These processes have no direct impact on the user experience, and can be killed at any time to reclaim memory for a foreground, visible, or service process.
- Usually there are many background processes running, so they are kept in an LRU (least recently used) list to ensure that the process with the activity that was most recently seen by the user is the last to be killed.
- If an activity implements its lifecycle methods correctly, and captures its current state, killing its process will not have a deleterious effect on the user experience.

Level 5: Empty process

- An **empty process** is one that doesn't hold any active application components.
- The only reason to keep such a process around is as a cache to improve startup time the next time a component needs to run in it.
- The system often kills these processes in order to balance overall system resources between process caches and the underlying kernel caches.

Processes, Life Cycles

- Android ranks a process at the highest level it can, based upon the importance of the components currently active in the process. For example, if a process hosts a service and a visible activity, the process will be ranked as a visible process, not a service process.
- In addition, a process's ranking may be increased because other processes are dependent on it. A process that is serving another process can never be ranked lower than the process it is serving.