Application Fundamentals

Android apps are written in the Java programming language. The Android SDK tools compile your code—along with any data and resource files—into an APK: an *Android package*, which is an archive file with an .apk suffix. One APK file contains all the contents of an Android app and is the file that Android-powered devices use to install the app.

Once installed on a device, each Android app lives in its own security sandbox:

- The Android operating system is a multi-user Linux system in which each app is a different user.
- By default, the system assigns each app a unique Linux user ID (the ID is used only by the system and is unknown to the app). The system sets permissions for all the files in an app so that only the user ID assigned to that app can access them.
- Each process has its own virtual machine (VM), so an app's code runs in isolation from other apps.
- By default, every app runs in its own Linux process. Android starts the process when any of the app's components need to be executed, then shuts down the process when it's no longer needed or when the system must recover memory for other apps.

In this way, the Android system implements the *principle of least privilege*. That is, each app, by default, has access only to the components that it requires to do its work and no more. This creates a very secure environment in which an app cannot access parts of the system for which it is not given permission.

However, there are ways for an app to share data with other apps and for an app to access system services:

- It's possible to arrange for two apps to share the same Linux user ID, in which case they are able to access each other's files. To conserve system resources, apps with the same user ID can also arrange to run in the same Linux process and share the same VM (the apps must also be signed with the same certificate).
- An app can request permission to access device data such as the user's contacts, SMS messages, the mountable storage (SD card), camera, Bluetooth, and more. The user has to explicitly grant these permissions. For more information, see Working with System Permissions.

That covers the basics regarding how an Android app exists within the system. The rest of this document introduces you to:

- The core framework components that define your app.
- The manifest file in which you declare components and required device features for your app.

- Resources that are separate from the app code and allow your app to gracefully optimize its behavior for a variety of device configurations.

App Components

App components are the essential building blocks of an Android app. Each component is a different point through which the system can enter your app. Not all components are actual entry points for the user and some depend on each other, but each one exists as its own entity and plays a specific role—each one is a unique building block that helps define your app's overall behavior.

There are four different types of app components. Each type serves a distinct purpose and has a distinct lifecycle that defines how the component is created and destroyed.

Here are the four types of app components:

Activities

An *activity* represents a single screen with a user interface. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email app, each one is independent of the others. As such, a different app can start any one of these activities (if the email app allows it). For example, a camera app can start the activity in the email app that composes new mail, in order for the user to share a picture.

An activity is implemented as a subclass of Activity and you can learn more about it in the Activities developer guide.

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 app, or it might fetch data over the network without blocking user interaction with an activity. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it.

A service is implemented as a subclass of Service and you can learn more about it in the Services developer guide.

Content providers

A *content provider* manages a shared set of app data. You can store the data in the file system, an SQLite database, on the web, or any other persistent storage location your app can access. Through the content provider, other apps 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. As such, any app with the proper permissions can query part of the content provider (such as ContactsContract.Data) to read and write information about a particular person.

Content providers are also useful for reading and writing data that is private to your app and not shared. For example, the Note Pad sample app uses a content provider to save notes.

A content provider is implemented as a subclass of ContentProvider and must implement a standard set of APIs that enable other apps to perform transactions. For more information, see the Content Providers developer guide.

You don't need to develop your own provider if you don't intend to share your data with other applications. However, you do need your own provider to provide custom search suggestions in your own application. You also need your own provider if you want to copy and paste complex data or files from your application to other applications.

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. Apps can also initiate broadcasts—for example, to let other apps 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. More commonly, though, a broadcast receiver is just a "gateway" to other components and is intended to do a very minimal amount of work. For instance, it might initiate a service to perform some work based on the event.

A broadcast receiver is implemented as a subclass of BroadcastReceiver and each broadcast is delivered as an Intent object. For more information, see the BroadcastReceiver class.

****

A unique aspect of the Android system design is that any app can start another app's component. For example, if you want the user to capture a photo with the device camera, there's probably another app that does that and your app can use it, instead of developing an activity to capture a photo yourself. You don't need to incorporate or even link to the code from the camera app. Instead, you can simply start the activity in the camera app that captures a photo. When complete, the photo is even returned to your app so you can use it. To the user, it seems as if the camera is actually a part of your app.

When the system starts a component, it starts the process for that app (if it's not already running) and instantiates the classes needed for the component. For example, if your app starts the activity in the camera app that captures a photo, that activity runs in the process that

belongs to the camera app, not in your app's process. Therefore, unlike apps on most other systems, Android apps don't have a single entry point (there's no main() function, for example).
Because the system runs each app in a separate process with file permissions that restrict access to other apps, your app cannot directly activate a component from another app. The Android system, however, can. So, to activate a component in another app, you must deliver a message to the system that specifies your *intent* to start a particular component. The system then activates the component for you.

Activating Components

Three of the four component types—activities, services, and broadcast receivers—are activated by an **asynchronous** message called an ***intent***. Intents bind individual components to each other at runtime (you can think of them as the messengers that request an action from other components), whether the component belongs to your app or another.

An intent is created with an Intent object, which defines a message to activate either a specific component or a specific *type* of component—an intent can be either explicit or implicit, respectively.

For activities and services, an intent defines the action to perform (for example, to "view" or "send" something) and may specify the URI of the data to act on (among other things that the component being started might need to know). For example, an intent might convey a request for an activity to show an image or to open a web page. In some cases, you can start an activity to receive a result, in which case, the activity also returns the result in an Intent (for example, you can issue an intent to let the user pick a personal contact and have it returned to you—the return intent includes a URI pointing to the chosen contact).

For broadcast receivers, the intent simply defines the announcement being broadcast (for example, a broadcast to indicate the device battery is low includes only a known action string that indicates "battery is low").

The other component type, content provider, is not activated by intents. Rather, it is activated when targeted by a request from a ContentResolver. The content resolver handles all direct transactions with the content provider so that the component that's performing transactions with the provider doesn't need to and instead calls methods on the ContentResolver object. This leaves a layer of abstraction between the content provider and the component requesting information (for security).

There are separate methods for activating each type of component:

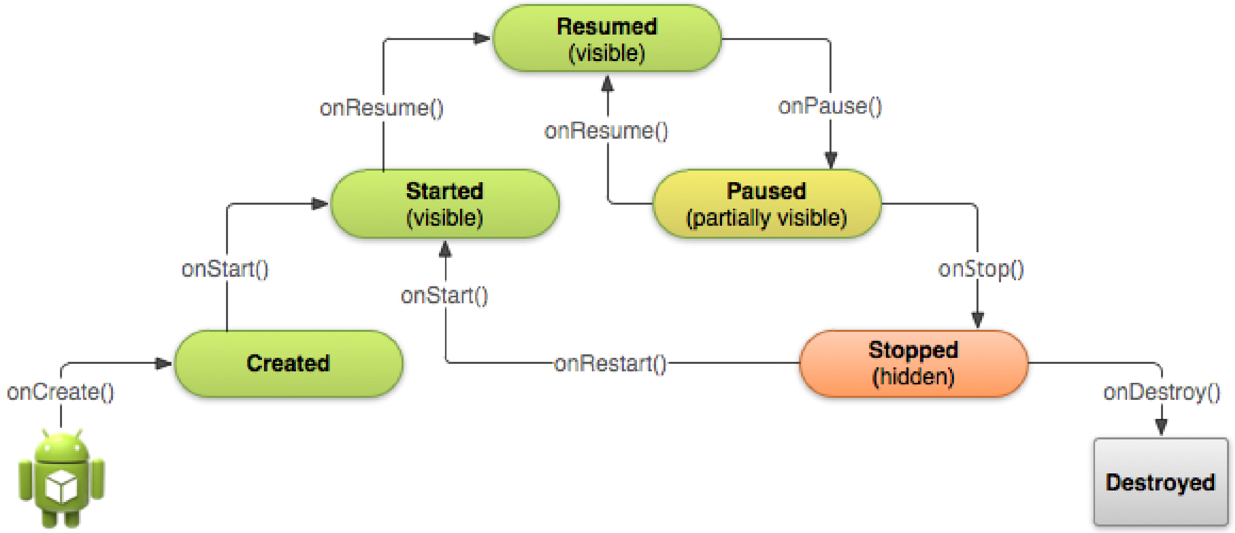
- You can start an activity (or give it something new to do) by passing an Intent to startActivity() or startActivityForResult() (when you want the activity to return a result).

- You can start a service (or give new instructions to an ongoing service) by passing an Intent to startService(). Or you can bind to the service by passing an Intent to bindService().
- You can initiate a broadcast by passing an Intent to methods like sendBroadcast(), sendOrderedBroadcast(), or sendStickyBroadcast().
- You can perform a query to a content provider by calling query() on a ContentResolver.

For more information about using intents, see the Intents and Intent Filters document. More information about activating specific components is also provided in the following documents: Activities, Services, BroadcastReceiver and Content Providers.

Android Activity Lifecycle

https://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle
https://developer.android.com/training/basics/activity-lifecycle/index.html



There are three key loops you may be interested in monitoring within your activity:

- 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(). For example, if it has a thread running in the background to download data from the network, it may create that thread in onCreate() and then stop the thread 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. For example, you can register a BroadcastReceiver in onStart() to monitor for changes that impact your UI, and unregister it in onStop() when the user no longer sees what you are displaying. The onStart() and onStop() methods can be called multiple times, as the activity becomes visible and hidden 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 in front of all other activities and interacting with the user. An activity can frequently go between the resumed and paused states -- for example when the device goes to sleep, when an activity result is delivered, when a new intent is delivered -- so the code in these methods should be fairly lightweight.

Lifecycle after Rotate

Your activity will be destroyed and recreated each time the user rotates the screen. When the screen changes orientation, the system destroys and recreates the foreground activity because the screen configuration has changed and your activity might need to load alternative resources (such as the layout).

A change in device orientation is interpreted as a configuration change which causes the current activity to be destroyed and then recreated. Android calls **onPause()**, **onStop()**, and **onDestroy()** on currently instance of activity, then a new instance of the same activity is recreated calling **onCreate()**, **onStart()**, and **onResume()**. The reason why Android have to do this, is because depending of screen orientation, portrait or landscape, we may need to load and display different resources, and only through re-creation Android can guarantee that all our resources will be re-requested.

Xiaoyaoworm: TwoWaysToHandleRotation.
Second method is very common: Rewrite **onSaveInstanceState**() and **onRestoreInstanceState**() to save and restore the data.

Recreate An Activity

When your activity is destroyed because the **user presses *Back*** or the **activity finishes itself**, the system's concept of that **Activity instance is gone foreve**r because the behavior indicates the activity is no longer needed.
However, if the system destroys the activity due to system constraints (rather than normal app behavior), then although the actual Activity instance is gone, the system remembers that it existed such that if the user navigates back to it, the system creates a new instance of the activity using **a set of saved data that describes the state of the activity**("instance state") when it was destroyed.

https://developer.android.com/training/basics/activity-lifecycle/recreating.html

Main Thread

When an application is launched, the system creates a thread of execution for the application, called "main." This thread is very important because it is in charge of dispatching events to the appropriate user interface widgets, including drawing events. It is also the thread in which your application interacts with components from the Android UI toolkit (components from the android.widget and android.view packages). As such, the main thread is also sometimes called the UI thread.

The system does *not* create a separate thread for each instance of a component. All components that run in the same process are instantiated in the UI thread, and system calls to each component are dispatched from that thread. Consequently, methods that respond to system callbacks (such as onKeyDown() to report user actions or a lifecycle callback method) always run in the UI thread of the process.

For instance, when the user touches a button on the screen, your app's UI thread dispatches the touch event to the widget, which in turn sets its pressed state and posts an invalidate request to the event queue. The UI thread dequeues the request and notifies the widget that it should redraw itself.

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

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

1. **Do not block the UI thread**
2. **Do not access the Android UI toolkit from outside the UI thread**

AsyncTask

AsyncTask enables proper and easy use of the UI thread. This class allows to perform background operations and publish results on the UI thread without having to manipulate threads and/or handlers.

https://androidresearch.wordpress.com/2012/03/17/understanding-asynctask-once-and-forever/

AsyncTasks should ideally be used for short operations (a few seconds at the most.) If you need to keep threads running for long periods of time, it is highly recommended you use the various APIs provided by the java.util.concurrent package such as Executor, ThreadPoolExecutor and FutureTask.

Rotate issue - The AsyncTask and Activity lifecycle

AsyncTasks don't follow Activity instances' life cycle. If you start an AsyncTask inside an Activity and you rotate the device, the Activity will be destroyed and a new instance will be created. But the AsyncTask will not die. It will go on living until it completes.
And when it completes, the AsyncTask won't update the UI of the new Activity. Indeed it updates the former instance of the activity that is not displayed anymore. This can lead to an Exception of the type *java.lang.**IllegalArgumentException**: View not attached to window manager* if you use, for instance, findViewById to retrieve a view inside the Activity.

Memory leak issue

It is very convenient to create AsyncTasks as inner classes of your Activities. As the AsyncTask will need to manipulate the views of the Activity when the task is complete or in progress, using an inner class of the Activity seems convenient : inner classes can access directly any field of the outer class.
Nevertheless, it means the inner class will hold an invisible reference on its outer class instance : the Activity.
On the long run, this produces a memory leak : if the AsyncTask lasts for long, it keeps the activity "alive" whereas Android would like to get rid of it as it can no longer be displayed. The activity can't be garbage collected and that's a central mechanism for Android to preserve resources on the device.
http://stackoverflow.com/questions/12797550/android-asynctask-for-long-running-operations

How to handle

Solution 1 – Think twice if you really need an AsyncTask. Consider about **IntentService**.
Solution 2 – Put the AsyncTask in a Fragment.
Solution 3 – Lock the screen orientation
Solution 4 – Prevent the Activity from being recreated.
https://androidresearch.wordpress.com/2013/05/10/dealing-with-asynctask-and-screen-orientation/

Xiaoyaoworm:
http://stackoverflow.com/questions/6373826/execute-asynctask-several-times
If you click on one button several times, which execute asynctask several times, what’s going on?

AsyncTask instances can only be used one time.

Instead, just call your task like new MyAsyncTask().execute("");
From the AsyncTask API docs:
Threading rules
There are a few threading rules that must be followed for this class to work properly:

- The task instance must be created on the UI thread.
- execute(Params...) must be invoked on the UI thread.
- Do not call onPreExecute(), onPostExecute(Result), doInBackground(Params...), onProgressUpdate(Progress...) manually.
- The task can be executed only once (an exception will be thrown if a second execution is attempted.) → This is that exception: Cannot execute task: the task has already been executed (a task can be executed only once)

Service & IntentService

http://stackoverflow.com/questions/15524280/service-vs-intentservice
http://techtej.blogspot.com.es/2011/03/android-thread-constructspart-4.html

Xiaoyaoworm: All communication are included here:
https://guides.codepath.com/android/Starting-Background-Services

Service

How to start a service?

- You can start a service (or give new instructions to an ongoing service) by passing an Intent to startService(). Or you can bind to the service by passing an Intent to bindService().

What is a service?

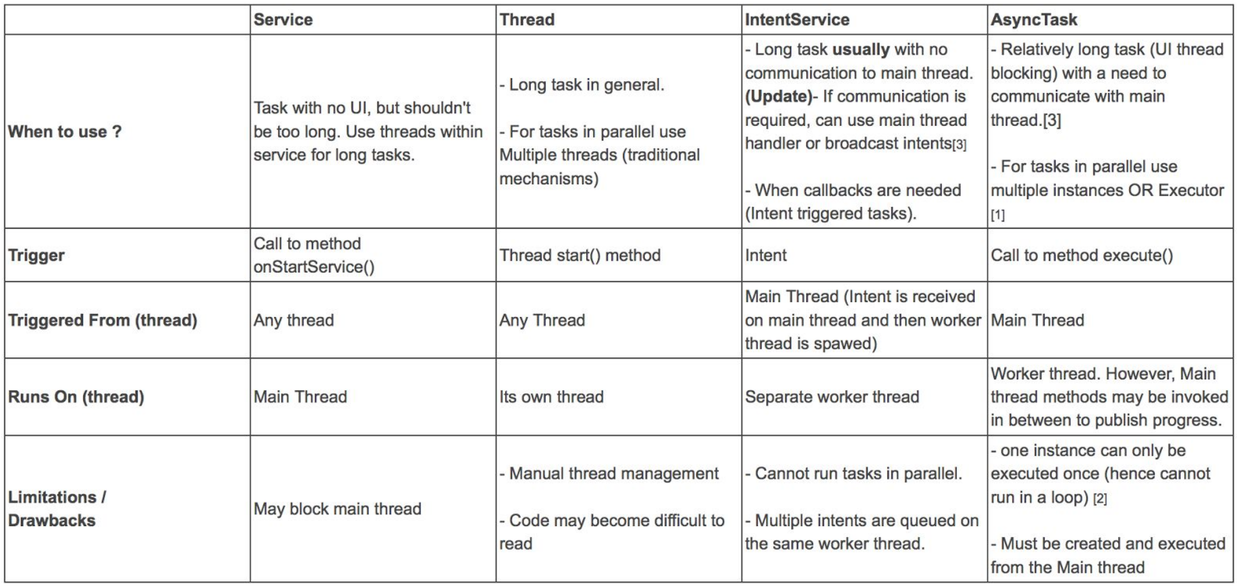
- A facility for the application to tell the system *about* something it wants to be doing in the background (even when the user is not directly interacting with the application). This corresponds to calls to Context.startService(), which ask the system to schedule work for the service, to be run until the service or someone else explicitly stop it.
- A facility for an application to expose some of its functionality to other applications. This corresponds to calls to Context.bindService(), which allows a long-standing connection to be made to the service in order to interact with it.

Note that services, like other application objects, run in the **main thread** of their hosting process. This means that, if your service is going to do any CPU intensive (such as MP3 playback) or blocking (such as networking) operations, it should spawn its own thread in which

to do that work. More information on this can be found in Processes and Threads. The IntentService class is available as a standard implementation of Service that has its own thread where it schedules its work to be done.
https://developer.android.com/reference/android/app/Service.html

Note: Sync adapters run **asynchronously**, so you should use them with the expectation that they transfer data regularly and efficiently, but **not instantaneously**. If you need to do real-time data transfer, you should do it in an AsyncTask or an IntentService.
https://developer.android.com/training/sync-adapters/index.html

Summary



	Service	**Thread**	**IntentService**	**AsyncTask**
When to use ?	Task with no UI, but shouldn't be too long. Use threads within service for long tasks.	- Long task in general. - For tasks in parallel use Multiple threads (traditional mechanisms)	- Long task **usually** with no communication to main thread. **(Update)**- If communication is required, can use main thread handler or broadcast intents[3] - When callbacks are needed (Intent triggered tasks).	- Relatively long task (UI thread blocking) with a need to communicate with main thread.[3] - For tasks in parallel use multiple instances OR Executor [1]
Trigger	Call to method onStartService()	Thread start() method	Intent	Call to method execute()
Triggered From (thread)	Any thread	Any Thread	Main Thread (Intent is received on main thread and then worker thread is spawed)	Main Thread
Runs On (thread)	Main Thread	Its own thread	Separate worker thread	Worker thread. However, Main thread methods may be invoked in between to publish progress.
Limitations / Drawbacks	May block main thread	- Manual thread management - Code may become difficult to read	- Cannot run tasks in parallel. - Multiple intents are queued on the same worker thread.	- one instance can only be executed once (hence cannot run in a loop) [2] - Must be created and executed from the Main thread

IntentService

https://androidresearch.wordpress.com/2012/03/04/working-with-services-in-android-intentservice/

ListView

http://lucasr.org/2012/04/05/performance-tips-for-androids-listview/
ListView is designed for scalability and performance. In practice, this essentially means:

1. It tries to do as few view inflations as possible.
2. It only paints and lays out children that are (or are about to become) visible on screen.

Android ListView Recycling

1. Since it’s expensive to inflate a new view, ListView recycles non-visible views (scrap view). Developers can simply update the contents of recycled views instead of inflating the layout of every single row—more on that later.
2. *ListView* uses the view recycler to keep adding recycled views below or above the current viewport and moving active views to a recyclable pool as they move off-screen while scrolling. This way *ListView* only needs to keep enough views in memory to fill its allocated space in the layout and some additional recyclable views

Use a Background Thread

Using a background thread ("worker thread") removes strain from the main thread so it can focus on drawing the UI. In many cases, using AsyncTask provides a simple way to perform your work outside the main thread. AsyncTask automatically queues up all the execute() requests and performs them serially. This behavior is global to a particular process and means you don’t need to worry about creating your own thread pool.

In the sample code below, an AsyncTask is used to load images in a background thread, then apply them to the UI once finished. It also shows a progress spinner in place of the images while they are loading.

```
// Using an AsyncTask to load the slow images in a background thread
new AsyncTask<ViewHolder, Void, Bitmap>() {
    private ViewHolder v;

    @Override
    protected Bitmap doInBackground(ViewHolder... params) {
        v = params[0];
        return mFakeImageLoader.getImage();
    }

    @Override
    protected void onPostExecute(Bitmap result) {
        super.onPostExecute(result);
        if (v.position == position) {
            // If this item hasn't been recycled already, hide the
            // progress and set and show the image
            v.progress.setVisibility(View.GONE);
            v.icon.setVisibility(View.VISIBLE);
            v.icon.setImageBitmap(result);
        }
    }
}.execute(holder);
```

Add ViewHolder For ListView

Every time *ListView* needs to show a new row on screen, it will call the *getView()* method from its adapter. ListView dynamically inflates and recycles tons of views when scrolling so it's key to make your adapter's *getView()* **as lightweight as possible**.

```
public View getView(int position, View convertView, ViewGroup parent) {
    if (convertView == null) {
        convertView = mInflater.inflate(R.layout.your_layout, null);
    }

    TextView text = (TextView) convertView.findViewById(R.id.text);
    text.setText("Position " + position);

    return convertView;
}
```

findViewById() could be expensive especially if your row layout is non-trivial. The ViewHolder pattern is about **reducing** the number of *findViewById()* calls in the adapter's *getView()*.

```
public View getView(int position, View convertView, ViewGroup parent) {
    ViewHolder holder;

    if (convertView == null) {
        convertView = mInflater.inflate(R.layout.your_layout, null);

        holder = new ViewHolder();
        holder.text = (TextView) convertView.findViewById(R.id.text);

        convertView.setTag(holder);
    } else {
        holder = convertView.getTag();
    }

    holder.text.setText("Position " + position);

    return convertView;
}

private static class ViewHolder {
    public TextView text;
}
```

Other ref link:
http://stackoverflow.com/questions/11945563/how-listviews-recycling-mechanism-works

RecycleView

https://guides.codepath.com/android/using-the-recyclerview#attaching-click-handlers-to-items

RecyclerView is an UI component designed to render a collection of data just like ListView and GridView, actually, it is intended to be a replacement of these two. What interests us from a testing point of view, is that a RecyclerView is no longer an AdapterView. This means that you can not use onData() to interact with list items.

Features of RecyclerView

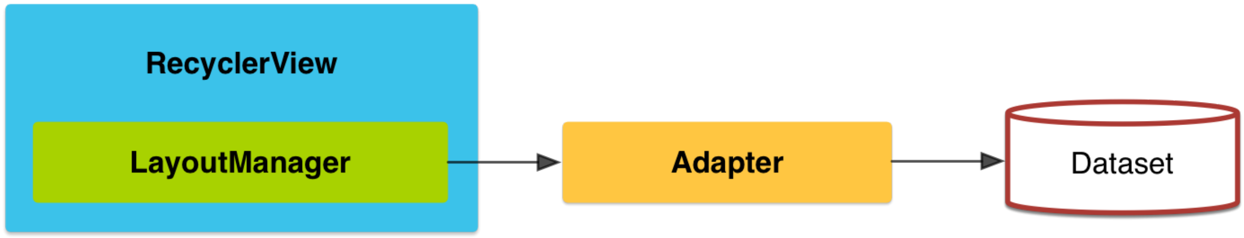
http://stackoverflow.com/questions/26728651/recyclerview-vs-listview

1. **Reuses cells while scrolling up/down** - this is possible with implementing View Holder in the listView adapter, but it was an optional thing, while in the RecycleView it's the default way of writing adapter.
2. **Decouples list from its container** - so you can put list items easily at run time in the different containers (linearLayout, gridLayout) with setting LayoutManager. Example:

```
mRecyclerView = (RecyclerView) findViewById(R.id.my_recycler_view);
mRecyclerView.setLayoutManager(new LinearLayoutManager(this));
//or
mRecyclerView.setLayoutManager(new GridLayoutManager(this, 2));
```

3. **Animates common list actions** - Animations are decoupled and delegated to **ItemAnimator**.

Components



RecyclerView.LayoutManager

A LayoutManager is responsible for measuring and positioning item views within a RecyclerView as well as determining the policy for when to recycle item views that are no longer visible to the user. By changing the LayoutManager a RecyclerView can be used to implement a standard vertically scrolling list, a uniform grid, staggered grids, horizontally scrolling collections and more. Several stock layout managers are provided for general use.

- *LinearLayoutManager* shows items in a vertical or horizontal scrolling list.
- *GridLayoutManager* shows items in a grid.
- *StaggeredGridLayoutManager* shows items in a staggered grid.

RecyclerView.Adapter

In ArrayAdapter(for ListView), it creates and binds views.
In a RecyclerView adapter, instead of creating and binding Views, your RecyclerView.Adapter creates and binds **RecyclerView.ViewHolder**, which is required.

You will have to override two main methods: one to inflate the view and its viewholder, and another one to bind data to the view.

- `bindViewHolder(VH holder, int position)`
- `createViewHolder(ViewGroup parent, int viewType)`

RecyclerView.ViewHolder

Required.
Implement `View.OnClickListener` to handle simple click.

Notifying Adapter When Data Changed

Unlike ListView, there is no way to add or remove items directly through the RecyclerView adapter. You need to make changes to the data source directly and notify the adapter of any changes.

There are two different classes of data change events:

1. **Item changes** are when a single item has its data updated but no positional changes have occurred.
2. **Structural changes** are when items are inserted, removed or moved within the data set.

   `void notifyDataSetChanged ()` - Does not specify what about the data set has changed, forcing any observers to assume that all existing items and structure may no longer be valid. LayoutManagers will be forced to fully rebind and relayout all visible views.

   If you are writing an adapter it will always be more efficient to use the more specific change events if you can. Rely on notifyDataSetChanged() as a last resort.

   Better ways:
   `notifyItemChanged(int)`
   `notifyItemInserted(int)`
   `notifyItemRemoved(int)`
   `notifyItemRangeChanged(int, int)`
   `notifyItemRangeInserted(int, int)`
   `notifyItemRangeRemoved(int, int)`

RecyclerView.ItemAnimator

Animates `ViewGroup` modifications such as add/delete/select that are notified to adapter. `DefaultItemAnimator` can be used for basic default animations and works quite well.

Compared to ListView

RecyclerView differs from its predecessor ListView primarily because of the following features:

- **Required ViewHolder in Adapters** - ListView adapters do not require the use of the ViewHolder pattern to improve performance. In contrast, implementing an adapter for RecyclerView requires the use of the ViewHolder pattern.
- **Customizable Item Layouts** - ListView can only layout items in a vertical linear arrangement and this cannot be customized. In contrast, the RecyclerView has a RecyclerView.LayoutManager that allows any item layouts including horizontal lists or staggered grids.
- **Easy Item Animations** - ListView contains no special provisions through which one can animate the addition or deletion of items. In contrast, the RecyclerView has the RecyclerView.ItemAnimator class for handling item animations.
- **Manual Data Source** - ListView had adapters for different sources such as ArrayAdapter and CursorAdapter for arrays and database results respectively. In contrast, the RecyclerView.Adapter requires a custom implementation to supply the data to the adapter.
- **Manual Item Decoration** - ListView has the android:divider property for easy dividers between items in the list. In contrast, RecyclerView requires the use of a RecyclerView.ItemDecoration object to setup much more manual divider decorations.
- **Manual Click Detection** - ListView has a AdapterView.OnItemClickListener interface for binding to the click events for individual items in the list. In contrast, RecyclerView only has support forRecyclerView.OnItemTouchListener which manages individual touch events but has no built-in click handling.

Intent & PendingIntent

Intent

An Android Intent is an object carrying an intent ie. message from one component to another component within the application or outside the application. The intents can communicate messages among any of the three core components of an application - activities, services, and broadcast receivers.

The intent itself, an Intent object, is a passive data structure holding an abstract description of an operation to be performed.

For example, let's assume that you have an Activity that needs to launch an email client and sends an email using your Android device. For this purpose, your Activity would send an ACTION_SEND along with appropriate chooser, to the Android Intent Resolver. The specified chooser gives the proper interface for the user to pick how to send your email data.

Explicit Intents

- Explicit intents specify the component to start by name (the fully-qualified class name). You'll typically use an explicit intent to start a component in your own app, because you know the class name of the activity or service you want to start. For example, start a new activity in response to a user action or start a service to download a file in the background.

```
// Explicit Intent by specifying its class name
Intent i = new Intent(this, TargetActivity.class);
i.putExtra("Key1", "ABC");
i.putExtra("Key2", "123");
// Starts TargetActivity
startActivity(i);
```

Implicit Intents

- Implicit intents do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it. For example, if you want to show the user a location on a map, you can use an implicit intent to request that another capable app show a specified location on a map.

```
// Implicit Intent by specifying a URI
Intent i = new Intent(Intent.ACTION_VIEW, Uri.parse("http://www.example.com"));
// Starts Implicit Activity
startActivity(i);
```

Intent Filters

When you create an **implicit intent**, the Android system finds the appropriate component to start by comparing the contents of the intent to the *intent filters* declared in the manifest file of other apps on the device. If the intent matches an intent filter, the system starts that component and delivers it the Intent object. If multiple intent filters are compatible, the system displays a dialog so the user can pick which app to use.

Caution: To ensure your app is secure, **always use an explicit intent when starting a Service and do not declare intent filters for your services.** Using an implicit intent to start a service is a security hazard because you cannot be certain what service will respond to the intent, and the user cannot 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.

PendingIntent

https://developer.android.com/reference/android/app/PendingIntent.html

A PendingIntent is a token that you give to a foreign application (e.g. NotificationManager, AlarmManager, Home Screen AppWidgetManager, or other 3rd party applications), which allows the foreign application to use your application's permissions to execute a predefined piece of code.
By giving a PendingIntent to another application, you are granting it the right to perform the operation you have specified as if the other application was yourself (with the same permissions and identity). As such, you should be careful about how you build the PendingIntent: almost always, for example, the base Intent you supply should have the component name explicitly set to one of your own components, to ensure it is ultimately sent there and nowhere else. (For security reasons you should always pass explicit intents to a PendingIntent rather than being implicit.)
http://stackoverflow.com/questions/24257247/differences-between-intent-and-pendingintent

http://codetheory.in/android-pending-intents/

BroadcastReceiver

A BroadcastReceiver is an Android app component that responds to system-wide broadcast announcements….Unlike Activities broadcast receivers do not have any user interface but may create a status bar notification.
http://codetheory.in/android-broadcast-receivers/

Register a BroadcastReceiver

1. Dynamically register an instance of this class with Context.registerReceiver()
2. Statically publish an implementation through the<receiver> tag in your AndroidManifest.xml.

Note: If registering a receiver in your Activity.onResume() implementation, you should unregister it in Activity.onPause(). (You won't receive intents when paused, and this will cut down on unnecessary system overhead). Do not unregister in Activity.onSaveInstanceState(), because this won't be called if the user moves back in the history stack.

Content Provider Basics

https://developer.android.com/guide/topics/providers/content-provider-basics.html
A content provider presents data to external applications as one or more tables that are similar to the tables found in a relational database. A row represents an instance of some type of data the provider collects, and each column in the row represents an individual piece of data collected for an instance.

Accessing a provider

An application accesses the data from a content provider with a ContentResolver client object. This object has methods that call identically-named methods in the provider object, an instance of one of the concrete subclasses of ContentProvider. The ContentResolver methods provide the basic "CRUD" (create, retrieve, update, and delete) functions of persistent storage.

The ContentResolver object in the client application's process and the ContentProvider object in the application that owns the provider automatically handle inter-process communication. ContentProvider also acts as an abstraction layer between its repository of data and the external appearance of data as tables.

For example, to get a list of the words and their locales from the User Dictionary Provider, you call `ContentResolver.query()`. The `query()` method calls the `ContentProvider.query()` method defined by the User Dictionary Provider. The following lines of code show a `ContentResolver.query()` call:

```
// Queries the user dictionary and returns results
mCursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,   // The content URI of the words table
    mProjection,                        // The columns to return for each row
    mSelectionClause                    // Selection criteria
    mSelectionArgs,                     // Selection criteria
    mSortOrder);                        // The sort order for the returned rows
```

Content URIs

A **content URI** is a URI that identifies data in a provider. Content URIs include the symbolic name of the entire provider (its **authority**) and a name that points to a table (a **path**). When you call a client method to access a table in a provider, the content URI for the table is one of the arguments.

In the preceding lines of code, the constant CONTENT_URI contains the content URI of the user dictionary's "words" table. The ContentResolver object parses out the URI's authority, and uses it to "resolve" the provider by comparing the authority to a system table of known providers. The ContentResolver can then dispatch the query arguments to the correct provider.

```
//TODO 这块好多，感觉没法精炼。。。。。
```

```
// I’ll help with that
```

ContentProvider

How to create a ContentProvider?

Data Contract, DB Helper, Database (SQLite), ContentProvider, URIMatcher
Steps:

1. Determine Content URIs (content://com.example.android.app/WEATHER/21210?date=20160623)
    a. Scheme: tells you a URI will be used and the URI refers to a ContentProvider
    b. Authority: A unique string to locate your ContentProvider, normally is the package name of the app.
    c. Location: points to a database table in the app
    d. Query (optional): the data you want to retrieve. Can be a list of records or a specific record.
        i. Dir: content://com.example.android.app/WEATHER/[LOCATION]
        ii. Item: content://com.example.android.app/WEATHER/[LOCATION]/[DATE]
2. Update Contract
    a. A contract class is a `public final` class that contains constant definitions for the URIs, column names, MIME types, and other meta-data that pertain to the provider. The class establishes a contract between the provider and other applications by ensuring that the provider can be correctly accessed even if there are changes to the actual values of URIs, column names, and so forth. https://developer.android.com/guide/topics/providers/content-provider-creating.html#ContractClass
    b.

c.

3. Fill out URIMather
4. Implement functions (onCreate, query, insert, update, delete, getType)

ContentResolver

AlarmManager

http://www.learn-android-easily.com/2013/05/android-alarm-manager_31.html
Many a times we want some task to be performed at some later time in future.
For Example: In SMS Scheduler we want a SMS to be send at some later time, or Task Reminder in which we want to be reminded about a task at a particular time, to implement all these things we use AlarmManager class.

AlarmManager class provides access to the system alarm services. These allow you to schedule your application to be run at some point in the future. When an alarm goes off, the Intent that had been registered for it is broadcast by the system, automatically starting the target application if it is not already running. Registered alarms are retained while the device is asleep (and can optionally wake the device up if they go off during that time), but will be cleared if it is turned off and rebooted.

The Alarm Manager holds a CPU wake lock as long as the alarm receiver's onReceive() method is executing. This guarantees that the phone will not sleep until you have finished handling the broadcast. Once onReceive() returns, the Alarm Manager releases this wake lock. This means that the phone will in some cases sleep as soon as your onReceive() method completes. If your alarm receiver called Context.startService(), it is possible that the phone will sleep before the requested service is launched. To prevent this, your BroadcastReceiver and Service will need to implement a separate wake lock policy to ensure that the phone continues running until the service becomes available.

In the example I will schedule an alarm to send SMS at a particular time in future.
We have two classes
1: MainAcitvity: in this class, we will schedule the alarm to be triggered at particular time .
2: AlarmReciever: when the alarm triggers at scheduled time , this class will receive the alarm, and send the SMS.

AlarmReciever class extends BroadcastReceiver and overrides onRecieve() method. inside onReceive() you can start an activity or service depending on your need like you can start an activity to vibrate phone or to ring the phone

In AndroidManifest:
Permission: <uses-permission android:name="com.android.alarm.permission.SET_ALARM"/>
Register receiver: <receiver android:name=".AlarmReciever"/>

AsyncLoader?

Weak Reference in Android

http://stackoverflow.com/questions/6116395/what-are-the-benefits-to-using-weakreferences
https://community.oracle.com/blogs/enicholas/2006/05/04/understanding-weak-references

Question: So I was thinking to use WeakReferences for all references to the bitmaps used by the views. I have never used a WeakReference and am not sure if this is a good application. Can any body provide an helpful pointers or tips?

Answer: Be careful, this is dangerous in your case. The Garbage Collector could get rid of all your bitmaps while your application may still need them.

The key issue about WeakReference is to understand the difference with hard references. If there is no more hard reference to a bitmap in your application, then the GC is allowed to atomically remove the object from memory and all existing weak reference will instantaneously point to null. In your case, you CANNOT use weak references all over your code.

Here is an idea of the solution. Create a container object that will keep weak references (only) to all your bitmaps. Your views should always reference bitmaps with hard references only. When a view creates a bitmap, it should register it in the container object. When it wants to use a view, it should obtain a hard reference from the container.

Like that, if no views is referring to a bitmap, then the GC will collect the object without side effects for views, since none has a hard reference to it. When using weakly referenced objects, it is good practice to explicitly set hard references to null when you don't need the object anymore.