LINUX KERNEL
low level tasks like hardware Ribers and
and menagement

↓

C/C++ Libs and Android Runtime
(such as Libc, SQLite) ↓ Android libraries and the VM, ART

Your app runs in its own instance of the VM, using classes and services provided here

↓

Application Framework

↓

Application layer
includes every app installed in the device

—

Main activity is launched when you start the app

It may contain a Fragment, a modular container within an activity

↓

we can reference an XML layout from the res/layout folder

When the activity runs, the fragment is created and the resource is inflated (layout inflater)

this converts everything in the XML file to a hierarchy of view objects in memory

When you hit run, your code is compiled into bytecode, that can be ~~run~~ in the Android Virtual Machine. Then that gets installed on the device

In android studio, this is done using gradle

↓

Manages dependencies and allows you to define custom built logic

By selecting make project will start gradle

project

↓

Built using gradle

↓

Bytecode, resources, Manifest

↓

APK

↓

for higher APIs it

↓

pushed to the running device Android Debug Bridge

Lets you interact and debug apps in any device, physical or virtual

By clicking on and the Fragment name Android Studio will show you that Fragment in the layout editor

Every view extends the view base class

Your layout has to resolve ~~adapt to each display~~.

Always choose the simpler that'll get the job done

Go for Linear/Relative/Grid layouts

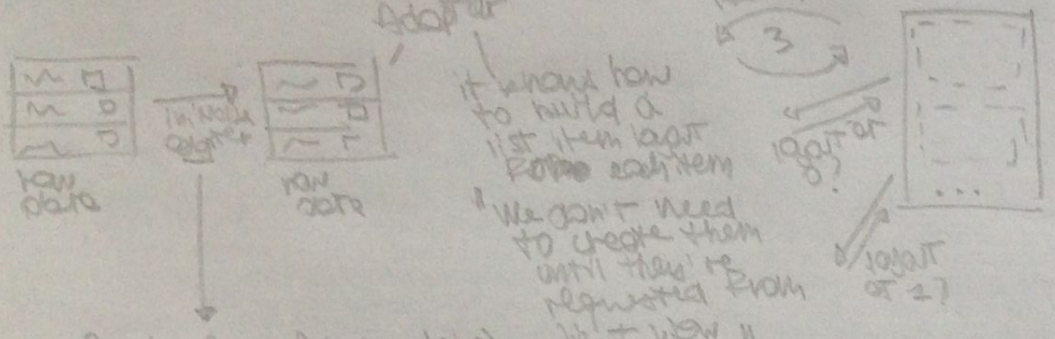
They resize and adapt to any screen.

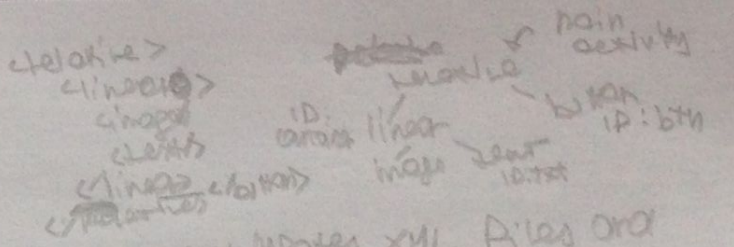
Adapter view classes inside items

- Frame layouts simple layouts with single or stack views
- Linear for stacking vertically or horizontally
- Relative is complicated views are positioned relative to other views or containers

That all descend from the ViewGroup class, designed to contain and give order to child views

Given some sample data, how to populate the list view





the system parses and inflates XML files and turns them into a full view hierarchy

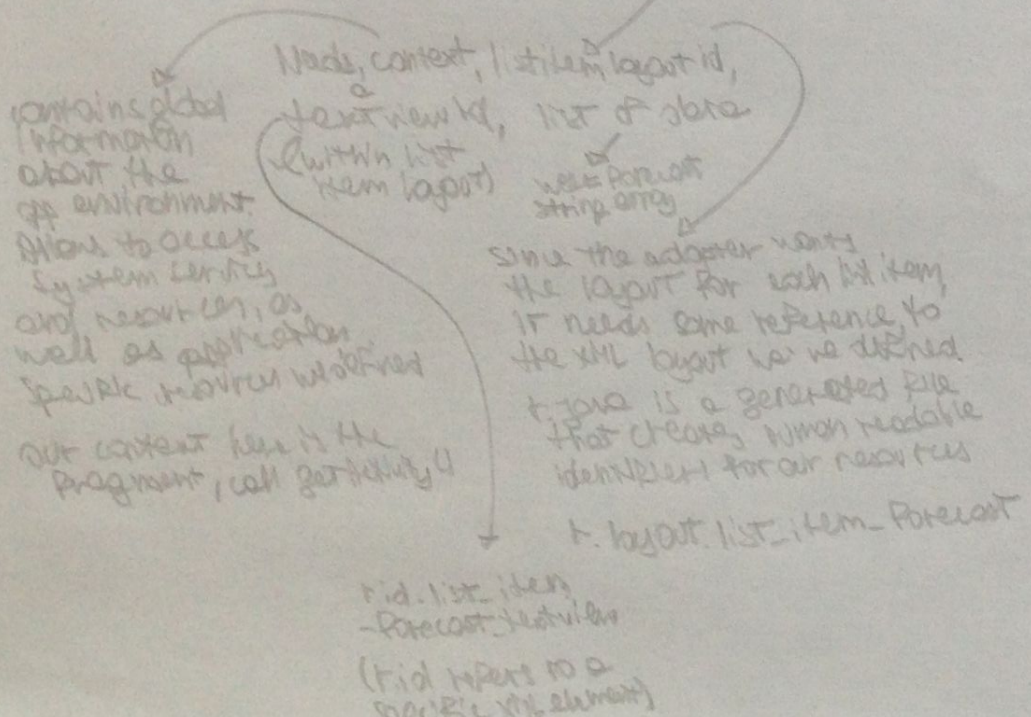
to have a reference to button, we call

b2 (Button) findViewById (R.id.btn);

container = (Linear...) findViewById (R.id.container

if you already have a reference to container, you can t = (text...) container.findViewById(R.id.text);

Create ArrayAdapter<String> in PlaceholderFragment onCreateView



to use http to send and receive data, we have two clients on Android

HttpURLConnection or HttpClient (apache)

- general purpose, lightweight, optimized to suit the needs of most Android apps

—

log a log error messages, with the adb command line tool, type adb logcat to see the debug and error messages printed to the logcat

—

Even the most trivial network accesses can take time. So instead of AsyncTask we use a Service.
It is an application component without the UI, that's less likely to be interrupted. Possibly scheduled by using an inexact repeating alarm.

Android has a specialized solution known as SyncAdapter. It is designed especially to schedule your background data syncs as efficiently as possible

Better, if we use Google Cloud Messaging. It lets your backend notify your SyncAdapter of changes in the server side. So you'll be able to initiate network requests when you know there's something to be updated

Android apps run by default on the main thread (UI thread)
It handles all the user input and output.
We want to avoid [redacted] to stutter

Background thread for long running work

Working with threads can be difficult. AsyncTask simplifies it for us.
With your own thread you have to manage it yourself and you cannot directly update the UI from a background thread
this is done for you with AsyncTask

The problem with AsyncTask is that the thread is tied to a UI component, in this case an activity. So if the activity is terminated the thread will also be terminated.
It should run for less

[redacted] with 7

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            Bitmap b = loadImageFromNetwork(...);
            mImageView.post(new Runnable() {
                public void run() {
                    mImageView.setImageBitmap(bitmap);
                }
            });
        }
    }).start();
}
```

By using post you can update the UI from background thread

```
public void onClick(View v) {
    new DownloadImageTask().execute(...);
}

private class DownloadImageTask
        extends AsyncTask<String, Void, Bitmap> {

    protected Bitmap doInBackground(String... urls) {
        return loadImageFromNetwork(urls[0]);
    }

    protected void onPostExecute(Bitmap result) {
        mImageView.setImageBitmap(result);
    }
}
```

URL

~~To update a listview~~ when new data is modified, the adapter will notify the listview when we use adapter.notifyDataSetChanged.

ArrayAdapter calls that method everytime you add or remove elements from it

—

menu options can be add in the XML file in res/menu for your given activity or fragment. <item> You can add an menu item by using the tag, the file will have the same name of your activity or fragment.

When added to a fragment, make sure to call setHasOptionsMenu(true) on the onCreate method. Overwrite onCreateOptionsMenu and in there inflate the xml resource file (inflater.inflate(R.menu.menu_..., menu)

—

Applications run in their own instance of the VM and have their own memory. To access other applications you must request permission. Permissions are declared in AndroidManifest.xml. request only the permissions you need.

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

In a list view, a list item is clicked, we want something to happen to implement that behaviour we use setOnItemClickListener. We should register an onItemClickListener via that method. Then, when an item is clicked, the callback is invoked

When a list item is clicked, we can show a Toast. It is a pop-up that displays a message for a few seconds. It can be useful for debugging to indicate status.

setItemOnClickListener is called within onCreateView, inside it we added clickListener, which we can instantiate in place.

```
listView.setOnItemClickListener(AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> adapter, View view, int i, long l)
    {
    }
});
```

—

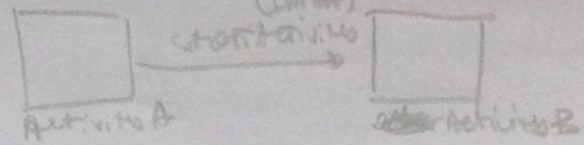
When creating a new activity, you're going to be offered for the hierarchical parent of it, that derives the up button next to the app icon and the left-pointing arrow. It is not a back button as it brings to your previous viewed screen.

—

```
@Override
public View onCreateView(...) {
    ...
    Intent intent = getActivity().getIntent()
    if(intent != null && intent.hasExtra(Intent.EXTRA_TEXT))
    { String message = intent.getStringExtra(Intent.EXTRA_TEXT);
    ((TextView) rootView.findViewById(R.id.detail_text))
        .setText(message);
    }
```

Intents supported by native apps can be found in "Common Intents Page"

To move from one activity to the other we call startActivity

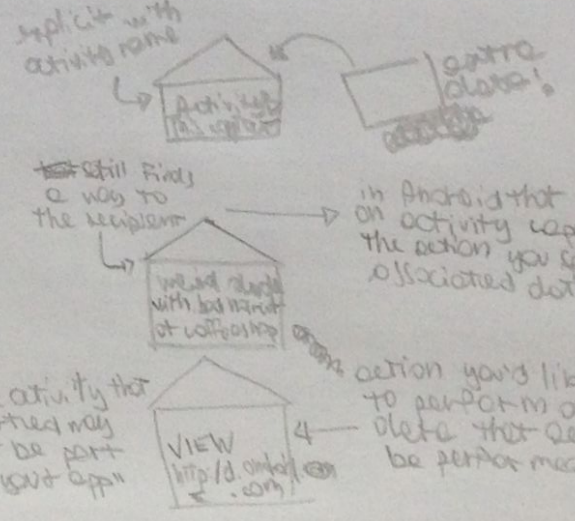


When using startActivity, rather than specifying the class name to start directly, we use an Intent object. We use it whenever we need app components to communicate with each other, or the system.

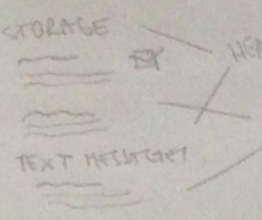
With the Intent explicitly indicating the target using the context and a class name of the activity.

There intents that use the name of the component you're targeting directly. They are called explicit intents

Intents are like envelopes that include which component you want to deliver to. And there's room for a small amount of data to be delivered



Android provides a way to show a hierarchy of preferences



common types are checkboxPreference ListPreference EditTextPreference

A preference can provide UI to allow the user to modify that setting.

Once the user changes the setting, it gets saved in the SharedPreferences file. This stores a bunch of key-value pairs of primitive data

Then at any time you can fetch the value of this shared preference as long as you pass the name of the key that you want

You can also use SharedPreferences attribute the context to get settings as well

To create a preference, define a preference XML file with root element PreferenceScreen with inside preference types. That file should go into a new resource folder with resource type XML.

Into that create your preferences file.

If you want your app to do something just after starting it, code that behaviour in the onStart method.

Dynamically registering will only receive the intent when the app is running.

Manifest will work for receiving for your broadcasts

Then you specify the intent filter

```
<intent-filter>
  <action .../>
</intent-filter>
```

in app as well

```
intentFilter =
  new IntentFilter
  ("action.key");
```

then register

You can register either in your manifest

```
<receiver
  android:name="MyReceiver">
  <intent-filter>
  </intent-filter>
</receiver>
```

or register it directly within your activity

```
registerReceiver(
  myReceiver,
  intentFilter);
```

by including an intent filter with the manifest

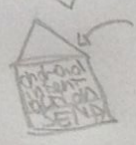
```
<intent-filter
  <action: name="android.intent.action.VIEW"
  <data android:scheme="geo" />
</intent-filter>
```

Indicates that this app is capable of performing this VIEW action on all data which has a geo scheme (optional attribute) kind of data

When you do an implicit intent, android checks the intent filters of every action that's installed on the device, until it finds a match.

The share intent is the most used with that intent, Android manages it for you by showing a list of choices for you.

We use Android action provider, it uses the action send intent



included either as text or URI to binary data

The MIME type indicates what the level points to

To broadcast a message to more apps we use sendBroadcast to send an intent. This is received by broadcast receivers. Using intent filters they indicate which broadcast they're interested in.

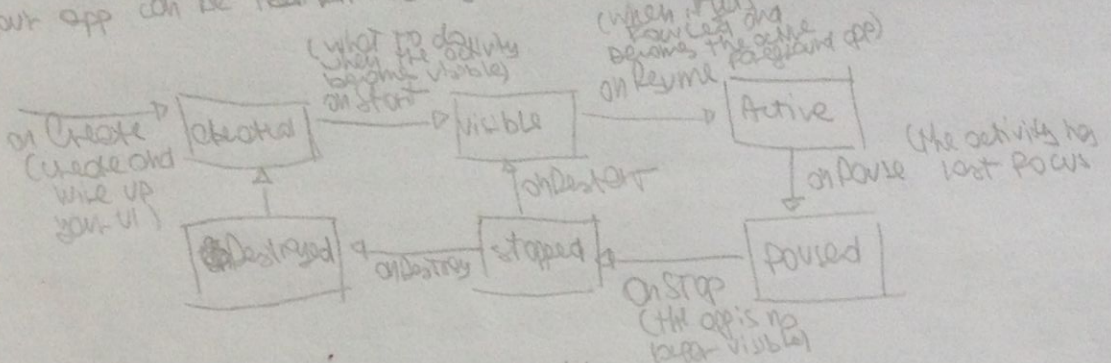
To receive:

```
public class MyReceiver extends BroadcastReceiver {

  @Override
  public void onReceive(...,...) {
    // handle received intent
  }
}
```

```
@Override
public boolean onCreateOptionsMenu(Menu) {
  getMenuInflater().inflate(R.menu.);
  MenuItem item = menu.findItem(R.id.);
  ShareActionProvider
    shareProvider = (ShareActionProvider)
      item.getActionProvider();
  return true
}

private void setShareIntent(Intent) {
  if (intent != null)
    mShareActionProvider.setShareIntent(
      shareIntent);
}
```

```
menu
  item
    android:showAsAction="ifRoom"
    android:actionProviderClass=
      "android.widget.ShareActionProvider" />
```

When having lot of apps in your backstack can use the limited resources your device has Android does it for us by killing low priority applications that you haven't used in a while.
Your app can be killed at any time

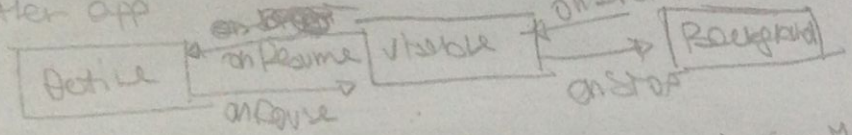


When the app is created, it quickly moves through create, start, resume
the move will move through that cycle several times

—

Active is when the app is in the foreground and has focus, here can receive input from user events and no other activities are occurring it

OnPause is called and the active lifetime ends, or soon as your activity is partially obscured (like when another activity tries to fulfill an implicit intent). You can pause classes that update your UI in onPause but not the ones showing your UI so it is still visible.
The visible lifetime ends when it is completely obscured by another app



Android apps can rely on onStop being called before the app is at risk of being terminated.
But if you're supporting pre-Honeycomb devices, you should prepare for termination on onPause

Android does everything it can to make resource limitations on the device invisible to the user. That means the foreground app is running smoothly and switching between apps seamlessly. That means killing apps in the background that it does not if it needs their resources.
In onPause and onStop we need to clean up any resources

Any task that needs to keep happening, even when your activity is paused, shouldn't really happen within an activity at all

To present the app with the same UI they had when they left or restart, To help, Android has some handlers, specifically for persisting state.
onSaveInstanceState is called immediately before onPause.
onRestoreInstanceState is called immediately after onCreate, But only if the app is being restored, after being terminated by the system.

You can read the bundle of state information saved the last time the app was moved from foreground, for storing to the same state it was the last time the user saw it.

We want to save data before the application is closed. Fetching data again is not good for user experience, and it will drain battery, user could spend internet data, server requests increases, offline users.

—

SharedPreferences, provide a general framework that allows you to save and retrieve persistent key value pairs of primitive data. Android do support for files

When you need more flexibility we use SQLite.
It provides a light weight relational database. That helps you organize and find data easily

—

In the database we'll define a contract, which is an agreement between the data model, storage and views, presentation describing how information is accessed. That agreement is implemented in a contract class.
This class usually contains a list of constants, typically db table column names, that are used to access data from the database, within the UI of an application

—

To create a db we extend SQLiteOpenHelper which helps creating a db and with versioning.
Create a class to extend SQLiteOpenHelper, pass our context, db name, (null) and version in the constructor.
In the onCreate, here, SQL and the contract are used together.
It is called the FIRST time the db is used. Call db.execSQL and create the tables you need

Android has an built in testing framework that allows us to write a test APK that executes a Junit test that all into classes in our main APK.

You use it by extending AndroidTestCase. Now you can override the setup method. It uses the same names a junit.

By extending TestSuite, you can have code to include all test classes in your package into a suite of tests that Junit will run.
This way you can add tests JUST by adding a new class file to the test directory.
TestSuite doesn't project specific. You can just copy the one in sunshine.

Each test should have a check (use an assert) to see if the program applies the correct output. You can just call "fail" if a certain code path should never have been reached

—

onUpgrade, in the DBHelper, is called when the db has already been created, but the version has changed.
SQLiteOpenHelper knows the version has changed because the version passed into the constructor has changed.
Make sure to change the version when you make changes to the database tables

—

We can get an instance of our database from our SQLiteOpenHelper. The instance is a SQLiteDatabase object.
getWritableDatabase() if also writing on the db.
ContentValues are used to represent the row we want to insert. It contains a set of key, value pairs, you've got to add a pair we call db.insert with the columns value to create the new row, it returns the new row-id.
If there's an error, it will return -1

Now we can query the db and receive a Cursor back. A query method is just a helper function to make it easier to execute a select statement. ~~query changes statement depending on the number of parameters we send to it~~

A cursor enables to traversal over the rows and columns of our query result set.
You can use moveToFirst or moveToNext() to change row and getColumnIndex to receive the index of the column you are in
Then you can call get methods and use that index to to retrieve values on that column index

ContentProvider, it allows to share your data across app boundaries by abstracting the underlying data source (SQLite, File or anything else) so that other apps can access it without really understanding how you stored it

By using content providers it's easier to switch out the data source, and much for someone else to manage the UI without having to understand your data storage implementation.

On the UI layer, it is a generic method that returns cursors

If you want to interact with anything outside of your app, you'll need it for that too.

ContentProviders are also needed when using CursorLoader and SyncAdapter and similar APIs for syncing and querying data. Those allow you to efficiently sync with your server, load data in your UI layer by using content observers which updates your UI automatically when the underlying data changes

—

We implement a content provider by extending the ContentProvider class

1 Determine the UI you'll need to support
2 Update the contract to include URIs
3 Fill out the UriMatcher to support those URIs
4 Implement the required functions

URI to pass to ContentProvider

content://com.example.android.sunshine.app/weather/94043

scheme: identifies the provider you'll use

content://com.example.android.sunshine.app/weather/94043

authority: unique string used to locate your content provider (it's the provider package name)

location, in this case a table

optional query, it could be followed by a question mark=

Universal Resource Identifier, they identify a resource (on Web, it's usually that identifies content with a http or https schema)

ContentResolver = getContentResolver();
resolver

Cursor cursor = resolver.query(UserDictionary.Words.CONTENT_URI, null, null, null, null);

App → Content Resolver → content provider ↑ user dictionary data

This is the content provider for Dictionary, there are other for apps such as calendar, alarms and many others.

Those parameters say which rows and which columns you want to include in the list. Null says you want to load all data

getCount() returns the # of rows

moveToNext() to move through its data through returns bool.

To take a value you use .getInt(index) of the column where that value is. Or you can use .getColumnIndex("") takes the name of the column as an argument

Helps your app communicate with the content provider (dictionary) your interested in. It helps your app connect with the content provider by looking at whatever request your app sends to the content resolver. It also allows you to refer to more content providers

On resolver you can now read, add (query insert) change, delete data update delete

methods used to do it, on resolver.

By calling resolver.query, it is calling the corresponding query method in the content provider

Always remember to call cursor.close(). This avoid memory leakage, and eventually need for your app to crash

...
while(cursor.moveToNext())
{ // iterate through all data
}
...

CursorAdapter it's like an Array Adapter but we use a cursor to populate list view

SimpleCursorAdapter adapter = new ...(this, R.layout.list_item
LAYOUT_TO_BE_BOUND,
columns,
LAYOUT_ITEMS_TO_FILL, 0);
listView.setAdapter(adapter);

flags

You can define the URI you'll come use in the contract!

Android provider support to append an ID to the URI path, to indicate a specific record

After our initial query, should register a content observer against the URI, so that when the db is updated, the data it will requery and update the UI

UriMatcher helps us write a content provider.
Content Providers implement functionality based on URIs passed to them.
It will implement four types of URIs, each type will be used to execute different types of operations against the underlying db.
Content providers tie each URI type to an integer constant.
These constant are already added when you extend ContentProvider

URI TYPES:
PATH
PATH/#
PATH/*
PATH/*/OTHER/#
* STRING
# NUMBER

Android provides URI matcher to help match incoming URIs to the content provider integer constants.
It passed each type of URI is passed to our content provider we can perform the requested operation. We can use the integer constants in switch statements

You must register your content provider in your manifest. Now you can use content resolver to refer to it.
It locates our class using the content authority and makes direct calls to the weather provider on our behalf.

Now we fill out the content provider w/ the functions query, insert, update, delete, getType

bulkInsert is an optional method that allows you to put a bunch of inserts into a single transaction.
This is much faster than doing it individually

Loaders are the best practice implementation for asynchronous data loading within activity or fragment.

When you create a Loader, it creates an AsyncTask to load data on the background thread, it then syncs with the UI thread when the initial loading is complete. Can be set up to monitor the underlying data, and deliver any updates to the UI thread.

CursorLoader is an implementation of the AsyncTaskLoader designed to query content providers and return a cursor which you can then bind to the UI. That cursor will be automatically updated whenever the content provider changes, based on changes to the underlying database.

It will reconnect to the last returned cursor after being recreated, after a configuration change (such as screen rotation)

By not using content providers, you cannot use content loaders and you get to create your own loader by extending AsyncTaskLoader directly

---

Loaders are registered by ID with a LoaderManager, which allows them to live beyond the lifecycle of the activity or fragment & they're associated with

AsyncTaskLoader implements the same functionality of AsyncTask, but because it's a loader it's lifecycle is different. In AsyncTaskLoader, once we restore the device, the LoaderManager will make sure the old loader is connected to the AsyncTaskLoader equivalent on postExecute. The loader keeps running in onLoadInBackground. Once it finishes, the activity gets notified through onLoadFinished

CursorLoader is derived from AsyncTaskLoader but has an additional optimization, if the cursor has already finished loading before the activity restarts, it recognizes this in the loader initialization and the cursor is immediately delivered to the UI

To create a cursor loader. First create an integer constant for your loader ID, then implement loader callbacks. Finally initialise the loader with the LoaderManager.

- private static final int MY_LOADER_ID = [my ID];
  (A single activity can have multiple loaders, and they're differentiated by these IDs)

- Loader callbacks are onCreateLoader, onLoaderFinished and onLoaderReset.
  When using cursor loaders, the parameterized type (<String>) is cursor.
  In onCreateLoader we create and return our cursor loader → Finished
  onLoadComplete is called when our load is complete and our data is ready. To use the data in our CursorAdapter we just call swapCursor. Here we also perform any other UI updates we'd want to perform when our data is ready
  onLoaderReset is only called when our loader is being destroyed
  We need to remove all references to our loader data. We call
  CursorAdapter
  swapCursor(null);
  We use LoaderManager to initialise our loader
  getLoaderManager().initLoader([loader ID], [Bundle], [loader callbacks]);
  Optional
  When using it on a fragment we initialise it in the onActivityCreated method

RECAP

In order to build our UI we use views. Views are rectangles on the screen (which however may not be visible)

It handles drawing and event handling and is extended by all the basic widgets.

Views here
google.com/design/spec/components/bottom-sheets.html

ViewGroup is a container for children views. There's FrameLayout, a child that gets added will be default positioned in the top left corner of the view group. If another view is added it'll overlap the first one. Typically we have one child per frame layout

LinearLayout is composed of children either in horizontal row or vertical column. We can also specify layout_weight which allows us to distribute the width or height of a parent across the children

RelativeLayout where we can specify that a view should to the parent's top, right, bottom or left edge and that one view should be relative to another view

In a GridLayout the view fills out cells in a grid. You can also have views that span multiple cells.

A viewgroup is a view, so in our code when we refer to a list item layout as being a single view, that just means the root view of the whole view hierarchy, for that layout. A viewgroup can be nested in another viewgroups.

A view can look the same on a device, when either it is "match_parent", or "wrap_content". That's because we cannot see the boundaries. You'll see the difference when you'll have other children on that view.

Gravity = center, will center the content within the TextView, layout_gravity=center will center the view within the parent.

Padding=x shifts the content inside by x amount. layout_margin=x the parent will be interpreted as this layout param, it adds a margin of x all around the TextView

A view can be visible, invisible or gone

—

In ForecastAdapter. For each row of the cursor we're going to go through and extract out the values. We call it binding the data to the views because we take the value from each column and set it to a different view. For next row we do the same until we have a list item for each row in the cursor.

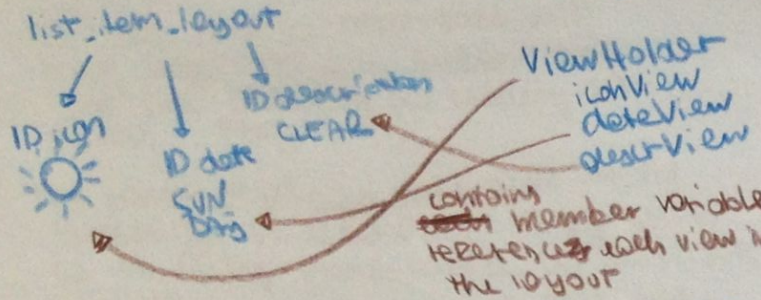
From CursorAdapter we're extending it we need to override bindView and newView. bindView tells us an existing list item layout and updates it with the data from the cursor. In newView we inflate the list item layout and in bindView we bind data to the individual views

—

ItemViewTypes, This is a general concept for all adapters. Instead of just having about one item view type which yields the same layout for all list items, the adapter can declare multiple item view types, so it will have different list item layouts.
For a given row in the cursor, the ForecastAdapter needs to decide if it should be a regular list item or a today list item. Then it will inflate and bind data for the appropriate layout.

1 Set getViewTypeCount() to return two. After averring it.
2 Override getItemViewType(position) based on the passed position
" each view type is represented by an integer, we can bind them in constant attributes
3 In newView(..) get the viewType and layout id and pass them to LayoutInflater.from(context).(—);

To remove unnecessary findViewById calls, we can use the ViewHolder pattern



The ViewHolder is stored in the tag field of the view. That way you use it again by just setting data onto these fields without finding the views again

A ViewHolder is just a pattern, given a list item layout we do all the findViewById calls of the child views in the constructor. By
In newView, inflate the layout and pass it to the ViewHolder.
You can now set the ViewHolder as the tag of the view.
In the bindView method we load the tag from the view to get back the ViewHolder

—

Inflating complex layouts can be expensive, could impact performances and responsiveness of your app.
The rules of thumb are keep your layout shallow and wide (more siblings and fewer children, and avoid having excessive views.
Also avoid using the frame layout as the root for layout that will always be inflated as a child into another one. In these circumstances, the merge tag is a better alternative

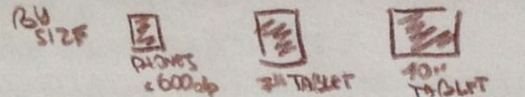
You can use Lint to optimize your layout

—

~~To~~ ~~make~~ your UI must be responsive. Apps can adapt by using multi pane layouts.
- Combine multiple views together

To make your app responsive across devices

By SIZE



res/layout/main_activity.xml <600dp
res/layout-sw600dp/main_activity.xml >600dp
res/layout-sw720dp/ —

sw<N>p specifies the smallest of the screen's two sides. That way you can ignore orientation

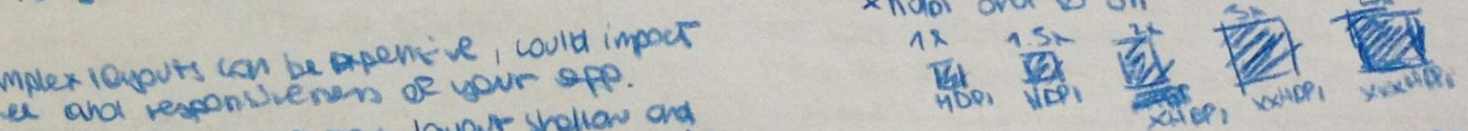
By DENSITY

	LDPI	MDPI	HDPI	XHDPI	...
DPI:	~120	~160	~240	~320	

"Whenever you need to convert from dp to pixels, we use getDisplayMetrics().density to get the screen density and convert pixels to dp's"

"ViewConfiguration lets you access common distances, speeds, times used by the Android system. Such as scroll threshold"

To translate an image to the different dpi's. We start from MDPI, which equals pixels. Then multiply by 1.5 for hdpi, then 2 for xhdpi and so on



"When building an APK for a target screen, the asset packaging tool can strip out the drawables for resolutions you don't need, but if it's in the mipmap folder, these assets will stay in the APK regardless of the target resolution"

↑ We use mipmap for the launcher icon.

Declare which screens your app supports in your manifest

<supports-screens android:...>

"If your app supports all screens you don't need that attribute as it is used as filtering for your app against unsupported devices"

At runtime android will check our layout configuration. And then load the right layout stuff (language) and drawables.

You can also chain these things such as a specific layout for a given language or a screen size for some device configuration.

Like with orientation, that's why android recommends if it must replace those resources.

Building a Flexible UI.

We can have a multipane with both tablet and phone layouts and just one Fragment per activity on phone

The power of Fragments goes beyond grouping UI elements. They allow us to fully modularize our activities, including the lifecycle events they receive in the app store that they maintain.

They were released in Honeycomb, along with tablet support. So that we could use multipane layout.

Fragment lifecycle

the basic lifecycle events are the same as an activity. it moves through start, resume, and stop with the activity owns

In most cases you just put in Fragment lifecycle handlers, what you've put in the activity's corresponding handlers. With some exceptions:

UIs are constructed in onCreateView instead of onCreate. (inflated)

onCreateView is used to look up with any data sources and return them to the activity as well.

A corresponding parent onDestroyView handler, is called immediately before the Fragment is added to the backstack

The FragmentManager can add any Fragment transactions, adding, removing, or replacing Fragments to the backstack with a single parent activity's actions.

... Fragments can move through the cycle, multiple times, independent from the host activity.

In onDestroyView you clean up resources specific to the UI (bitmaps, whatever) As soon the Fragment returns to the backstack, onCreateView is called and you can recreate the UI and reconnect data sources, before your Fragment transitions.

Since a Fragment only exists within an activity, we need callbacks to tell us when a Fragment is attached or detached from its parent.

In onAttach you can get a reference to parent activity. While onAttach is the last thing that happens in onActivityCreated, nothing on Fragment that the parent activity has completed its onCreate handler. In that callback we can safely interact with its UI. Once the Fragment is no longer visible, there's a chance it will be terminated, that happen in onStop (in the activity) and after onDestroyView

---

Fragments are also used to create logical modules, we call them non-UI Fragments.

Since they're non-visual, There's no need to recreate them every time the UI needs updating.

Within onCreate of the Fragment, call setRetainInstance passing in True, and return null in onCreateView. Now any connections, threads or tasks which don't need to be interrupted!

layout is the standard layout folder. We can support additional layouts by adding folders. For instance layout-sw600dp contains layouts for devices where the smallest width (sw) is 600dp. In that way we specify the layout qualifiers the current orientation. We can use w, like layout-w600dp, to specifically target with . For whenever the screen provides at least 600dp or within. You can do the same with height by using h.