Android Widget Framework

Each Activity‘s view hierarchy starts at class ViewRoot (ViewRoot.java). ViewRoot.performTraversals measures, layouts and draws the views as stated on the next two slides. Therefore it calls the following three methods on its only child, the inflated window layout that contains the Activity‘s content:

- measure (lines 763, 950 and 976)
- layout (line 996)
- draw (line 1114)

How Android Draws Views I

When an Activity receives focus, it will be requested to draw its layout. The Android framework will handle the procedure for drawing, but the Activity must provide the root node of its layout hierarchy.
Drawing begins with the root node of the layout. It is requested to measure and draw the layout tree. Drawing is handled by walking the tree and rendering each View that intersects the invalid region. In turn, each View group is responsible for requesting each of its children to be drawn (with the draw() method) and each View is responsible for drawing itself. Because the tree is traversed in-order, this means that parents will be drawn before (i.e., behind) their children, with siblings drawn in the order they appear in the tree.
The framework will not draw Views that are not in the invalid region, and also will take care of drawing the Views background for you.
You can force a View to draw, by calling invalidate().
Drawing the layout is a two pass process: a measure pass and a layout pass. The measuring pass is implemented in measure(int, int) and is a top-down traversal of the View tree. Each View pushes dimension specifications down the tree during the recursion. At the end of the measure pass, every View has stored its measurements. The second pass happens in layout(int, int, int, int) and is also top-down. During this pass each parent is responsible for positioning all of its children using the sizes computed in the measure pass.
When a View's measure() method returns, its getMeasuredWidth() and getMeasuredHeight() values must be set, along with those for all of that View's descendants. A View's measured width and measured height values must respect the constraints imposed by the View's parents. This guarantees that at the end of the measure pass, all parents accept all of their children's measurements. A parent View may call measure() more than once on its children. For example, the parent may measure each child once with unspecified dimensions to find out how big they want to be, then call measure() on them again with actual numbers if the sum of all the children's unconstrained sizes is too big or too small (i.e., if the children don't agree among themselves as to how much space they each get, the parent will intervene and set the rules on the second pass).
To initiate a layout, call requestLayout(). This method is typically called by a View on itself when it believes that is can no longer fit within its current bounds.
The measure pass uses two classes to communicate dimensions. The View.MeasureSpec class is used by Views to tell their parents how they want to be measured and positioned. The base LayoutParams class just describes how big the View wants to be for both width and height. For each dimension, it can specify one of:

How Android Draws Views II

an exact number,
FILL_PARENT, which means the View wants to be as big as its parent (minus padding)
WRAP_CONTENT, which means that the View wants to be just big enough to enclose its content (plus padding).
There are subclasses of LayoutParams for different subclasses of ViewGroup. For example, RelativeLayout has its own subclass of LayoutParams, which includes the ability to center child Views horizontally and vertically.
MeasureSpecs are used to push requirements down the tree from parent to child. A MeasureSpec can be in one of three modes:
UNSPECIFIED: This is used by a parent to determine the desired dimension of a child View. For example, a LinearLayout may call measure() on its child with the height set to *UNSPECIFIED* and a width of *EXACTLY* 240 to find out how tall the child View wants to be given a width of 240 pixels.
EXACTLY: This is used by the parent to impose an exact size on the child. The child must use this size, and guarantee that all of its descendants will fit within this size.
AT_MOST: This is used by the parent to impose a maximum size on the child. The child must guarantee that it and all of its descendants will fit within this size.

http://developer.android.com/guide/topics/ui/how-android-draws.html

measure - onMeasure

The View class (View.java) finds out how big a view should be (width and height). Therefore View.measure calls the View‘s onMeasure method that has to call View.setMeasuredDimension at the end to set the measured dimension.
A ViewGroup (ViewGroup.java) is a layout manager, e.g. class LinearLayout. ViewGroup is extended from class View and overwrites the onMeasure method. ViewGroup.onMeasure then measures all its children and adds its own padding sizes to the measurement → we get the width and height of a ViewGroup like LinearLayout.
At the end of a measurement setMeasuredDimension is called and the View‘s `mMeasuredWidth` and `mMeasuredHeight` are set accordingly.

layout - onLayout

The View class sets the positions of all its child‘s using the layout method. If the layout of a view has changed its onLayout method is also called. This is necessary to relayout the child‘s of a ViewGroup in case the ViewGroup‘s layout has changed. The View.layout methods assign the view a new position by calling the View.setFrame method. View.setFrame then sets `mLeft`, `mTop`, `mRight` and `mBottom` of the View class.

→ onLayout is overwritten by all layout managers (e.g. LinearLayout) so that they are able to position their children. OnLayout is not necessary in widgets like TextView, ImageView, etc. because they get their position from the parent layout manager.

draw - onDraw - dispatchDraw

View.draw first draws its content by calling onDraw and then draws its children by calling dispatchDraw. → onDraw is for the widget‘s content and dispatchDraw is for the children.
ViewGroup therefore overwrites the dispatchDraw method of class View to draw all its children. E.g. a ListView inherits from ViewGroup and also overwrites the dispatchDraw method. But ListView.dispatchDraw only draws the list seperators, the children are drawn by ViewGroup.dispatchDraw by calling super.dispatchDraw from ListView.dispatchDraw.

A widget‘s onDraw method (like TextView or ImageView) draws its content by using the provided canvas. The canvas is already translated to the child‘s position (client area) by the ViewGroup‘s dispatchDraw method. Therefore the widget only has to add its local padding and draw its content.

ViewGroup.dispatchDraw iterates over all its children and calls drawChild for each of them. ViewGroup.drawChild translates the canvas to the child‘s client area before calling the child‘s draw/dispachDraw method. Therefore ViewGroup uses the child‘s `mLeft`, `mTop`, `mRight` and `mBottom` values that are set during the layout phase.
If commenting out `canvas.translate` after `if (hasNoCache)` in ViewGroup.drawChild, all children are drawn at the upper left corner of the screen.

Android Widget Framework

View → TextView, ImageView, Button, ImageButton, …
ViewGroup → LinearLayout, ListView, GridView, …

Android Widget Framework by Example

Before ViewRoot calls measure it gets the root measure spec by ViewRoot.getRootMeasureSpec. This method generates the measure spec accordingly to the layout params.
- MeasureSpec.EXACTLY + size for LayoutParams.FILL_PARENT
- MeasureSpec.AT_MOST + size for LayoutParams.WRAP_CONTENT

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent">
<TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" />
<ImageView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" />
</LinearLayout>
```

LinearLayout.onMeasure
LinearLayout.measureVertical
→ for each child: LinearLayout.measureChildBeforeLayout
ViewGroup.measureChildWithMargins
→ sets the TextView‘s and ImageView‘s measure spec to MeasureSpec.AT_MOST and the remaining height of the LinearLayout
View.measure (child.measure), e.g. ImageView.measure
ImageView.onMeasure
View.setMeasuredDimension: width and height of the image
View.getMeasuredHeight → get the measured height calculated by the widget (e.g. ImageView)
→ take the current height (mTotalLength) into account for the next widget in vertical order in the call to LinearLayout.measureChildBeforeLayout

Android ListView by Example I

ListView class hierarchy:

```
ListView
AbsListView
     AdapterView<ListAdapter>
          ViewGroup
               View
```

ListView: ListView draws the list dividers in dispatchDraw and then delegates the reset to AbsListView.dispatchDraw. ListView also implements the scrolling behaviour that is triggered by key events. The scrolling behaviour that is triggered by touch events is implemented in AbsListView.

AbsListView: AbsListView holds common functionality for ListView and GridView. It draws the list selector in dispatchDraw and then delegates the rest to ViewGroup.dispatchDraw. It implements the scrolling behaviour that is initiated by touch events and by the public API methods `smoothScrollToPosition`, `smoothScrollBy`, etc.

AdapterView: AdapterView connects the widget to its data. Therefore is creates child views for all items using specific adapters. It also manages OnItemClickListeners etc..

ViewGroup: ViewGroup draws the child views (items) of a ListView in its dispatchDraw method. **A ListView‘s ViewGroup only contains the visible items of the list. Invisible items are put into the AbsListView.RecycleBin and are reused later.**

View: View contains common functionality for all views.

Android ListView by Example II

How a list draws itself:

`View.draw` (first draws the list background, then the child views and at last the scrollbars)
`ListView.dispatchDraw` (draws the list dividers)
`AbsListView.dispatchDraw` (draws the list selector)
`ViewGroup.dispatchDraw` (draws all child views by calling ViewGroup.drawChild for each of them)
`ViewGroup.drawChild` (Draw a child of this ViewGroup. This method is responsible for getting the canvas in the right state. This includes clipping, translating so that the child's scrolled origin is at 0, 0, and applying any animation transformations)
→ also calls the child's `computeScroll` callback before doing the canvas translation.
`View.draw` or `View.dispatchDraw` (draws the child itself)

How does `invalidate` work?

`View.invalidate` → checks if the view is not already invalidated (if the `DRAWN` flag of `mPrivateFlags` is not set anymore) and if so calculates the dirty rect for this view. Then the view's parent is used to call `invalidateChild` with this (child) view and the dirty rect as arguments. Also the view is marked as invalid be clearing the `DRAWN` flag.
`ViewGroup.invalidateChild` → is called in a loop for all parents of the child view until ViewRoot is reached
`ViewGroup.invalidateChildInParent` → called with the location and dirty rect of the child view
`ViewRoot.invalidateChildInParent`
`ViewRoot.invalidateChild` → sets the ViewRoots dirty rect (`mDirty`) of class ViewRoot and calls scheduleTraversals which posts a message to call performTraversals. ViewRoot.performTraversals triggers the whole view hierarchy to be redrawn with the dirty rect as the clipping area of the canvas.

→ The `DRAWN` flag of `mPrivateFlags` is set by `ViewGroup.drawChild` as soon as possible to enable further successful calls to `inavlidate`.

Android ListView by Example III

AdapterView enables the list to bind to serveral data sources for information retrieval.

Therefore AbsListView.obtainView creates application specific list items using some adapter (e.g. BaseAdapter). AbsListView.obtainView checks if there is any view in the RecycleBin that can be reused and then calls the getView method of the application‘s adapter to create or adjust it.

AdapterView overwrites all `addView` and `removeView` methods (of base class ViewGroup) and throws an UnsupportedOperationException if called → adding childs to a ListView is only possible using an adapter. ListView uses ViewGroup.attachViewToParent to add items to the list.

AdapterView also contains methods like:
- getSelectedItem
- getItemIdAtPosition
- getItemAtPosition
- setOnItemClickListener
- etc.

ListView initialization
Initially a ListView is filled during the layout phase.

AbsListView.onLayout
    ListView.layoutChildren
        ListView.fillFromTop

Android ListView by Example IV

ListView scrolling behaviour for key events
→ this implementation scrolls the list without doing any animations
→ the scrolling is done one item at a time

ListView.onKeyDown(KEYCODE_DPAD_DOWN)
- ListView.commonKey
  - ListView.arrowScroll
    - ListView.arrowScrollImpl
      - ListView.amountToScroll → determine how much pixels to scroll to get to the next visible view
      - ListView.scrollListItemsBy
        - ViewGroup.offsetChildrenTopAndBottom → offset the vertical location of all childs by the specified amount of pixels
        - ListView.addViewBelow
          - AbsListView.obtainView → create or adjust the new list item
          - ListView.setupChild → adds a child view to the list and positions it properly
            - ViewGroup.attachViewToParent
        - ViewGroup.detachViewFromParent → remove all top child views (items) that are no longer visible
        - AbsListView.RecycleBin.addScrapView → add the now invisible child views to the recycle bin

Android ListView by Example V

ListView scrolling behaviour for touch events (and the `smoothScrollToPosition` API call)
→ this implementation scrolls the list with animations

AbsListView.smoothScrollToPosition
    AbsListView.PositionScroller.start → posts the PoritionScroller runnable into the MQ.
AbsListView.PositionScroller.run
    AbsListView.smoothScrollBy
        AbsListView.FlingRunnable.startScroll
            Scroller.start → starts the scroller to compute scroll positions for a specified distance and duration
        → posts the FlingRunnable into the MQ to scroll the list by one item
    → post the PositionScroller runnable again to the MQ until the desired position is reached
FlingRunnable.run → scrolls the list by one item
    Scroller.computeScrollOffset → computes the scroll offsets for this point in time
    AbsListView.trackMotionScroll
        AbsListView.RecycleBin.addScrapView → add invisible child views (items) to the recycle bin
        ViewGroup.detachViewsFromParent
        ViewGroup.offsetChildrenTopAndBottom → offset the vertical location of all visible childs by the specified amount of pixels
        invalidate
        ListView.fillGap
            ListView.fillDown
                ListView.makeAndAddView
                    AbsListView.obtainView → create or adjust the new list item
                    ListView.setupChild → adds a child view to the list and positions it properly
        (AbsListView.positionSelector → sets the new selector rect that is drawn during the next drawing phase)
    → post this runnable again to the MQ until the Scroller is done and the desired position is reached
    → the animation is drawn as fast as possible without any sleeps

Android Launcher UI WorkspaceView

Writeing a widget that is bigger than a single screen and flinging between the screens is pretty easy.
This UI pattern is used by Launchers and the Google News and Weather app.
Just take a look into the WorkspaceView project.

First you have to create an instance of class WorkspaceView and add several childs to it. Every child has a fullscreen layout in this example.
When the screen has to smoothly scroll to the left or to the right just start the Scroller to compute the required offsets and call `invalidate`. While drawing a child `ViewGroup.drawChild` calls the `computeScroll` callback of class WorkspaceView which computes the new scroll offsets and sets them using `scrollTo`. So all childs are now drawn using the computed `mScrollX` and `mScrollY` offsets. The scroll offsets are applied to each child‘s canvas before being drawn in `ViewGroup.drawChild`. This is done as fast as possible until `Scroller.computeScrollOffsets` (in `WorkspaceView.computeScroll`) returns false.

That‘s all!