About This Document

This article provides information about the UI tips and tricks in Android. This article describes about various widgets and layouts that are useful to make UI in applications. This article also covers the tools that are helpful to develop the UI screens.

Scope

This article is intended for Android developers wishing to develop mobile applications. It assumes basic knowledge of Android and Java programming languages. This article provides information about various tools that are useful to improve the application performance.

Introduction

This article is concentrates on the following elements to improve the performance of your applications. This article describes various widgets and layouts that are useful to make UI in applications. This article also covers the tools that are helpful to develop the UI screens.

ListView

The most widely used widget is ListView/GridView in Android. ListView/GridView is the best widget when you need to show more icons in a view. Let's take a look at the ListView's performance.

The below code snippet shows the getView() method in a ListView.

```
        public View getView(int position, View convertView, ViewGroup parent) {
                convertView = mInflater.inflate(R.layout.list_item_icon_text, null);
                ((TextView) convertView.findViewById(R.id.text))
                                .setText(data[position]);
                ((ImageView) convertView.findViewById(R.id.icon)).setImageBitmap(mIcon);
                .
                .
                .
                return convertView;
        }
```

The above mentioned implementation for the getView() method is not the best, because the implementation is always inflates the layout every time when ever a view needs to return.

A better-modified version for the above implementation is given below

```
        public View getView(int position, View convertView, ViewGroup parent) {
                if (convertView == null)
                        convertView = mInflater.inflate(R.layout.list_item_icon_text,
null);
                ((TextView) convertView.findViewById(R.id.text))
                                .setText(data[position]);
                ((ImageView) convertView.findViewById(R.id.icon)).setImageBitmap(mIcon);
                .
```

```
                .
                .
                return convertView;
        }
```

The above implementation only inflates the layout whenever the convertview is null. It improves the listview performance a lot more than the previous implementation, but there is still another way to implement the getView() method.

A modified version for the above implementation is given below.

```
        static class ViewHolder {
                TextView text;
                ImageView icon;
        }

        public View getView(int position, View convertView, ViewGroup parent) {
                ViewHolder holder;
                if (convertView == null) {
                        convertView = mInflater.inflate(R.layout.list_item_icon_text,
null);
                        holder = new ViewHolder();
                        holder.text = (TextView) convertView.findViewById(R.id.text);
                        holder.icon = (ImageView) convertView.findViewById(R.id.icon);
                        convertView.setTag(holder);
                } else {
                        holder = (ViewHolder) convertView.getTag();
                }
                holder.text.setText(DATA[position]);
                holder.icon.setImageBitmap(mIcon);
                return convertView;
        }
```

The above mentioned implementation is the best to implement the getView() method in listview. This implementation uses the ViewHolder class to improve the performance of the listview. The below figure shows the listview performance for the above mentioned 3 scenarios respectively.

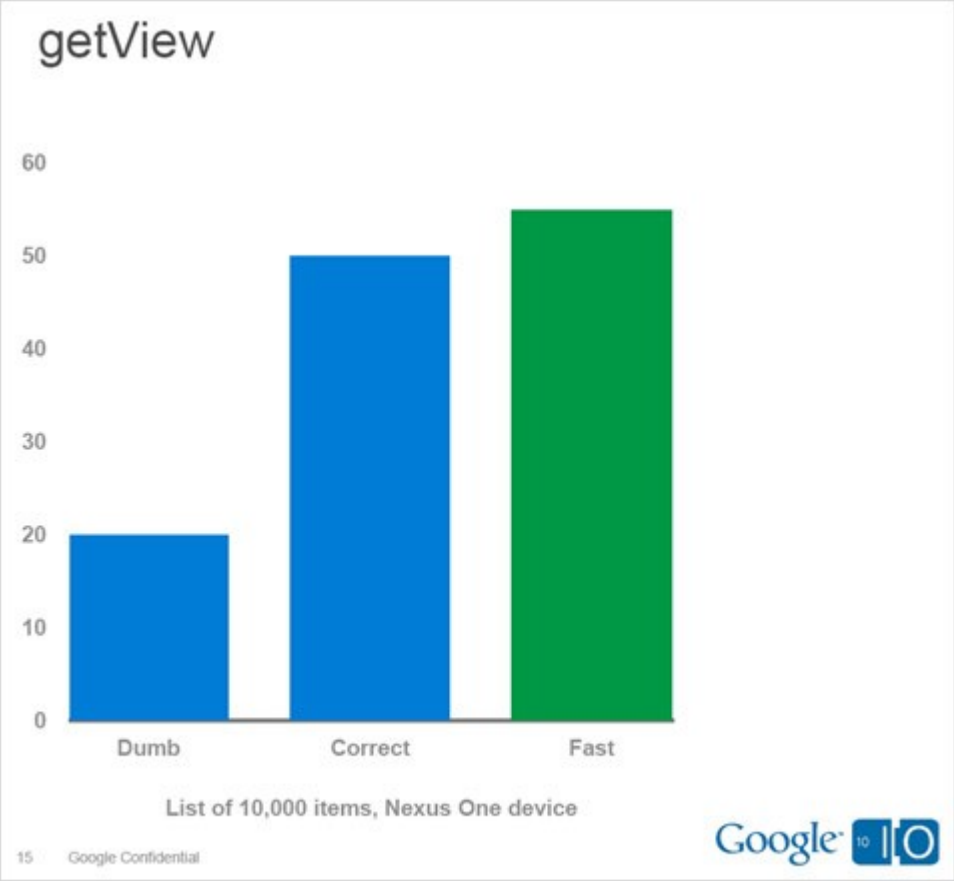

Figure 0: GetView Status

The following code shows ListView having a method to show the empty layout whenever the adaptor class does not have anything to show.

```
listView.setEmptyView(view)
```

Tools

The UI performance will suffer if more layouts are used unnecessarily. The Android sdk provides tools for checking the layouts used in the application screens.

layoutopt

The Andorid sdk provides the layoutopt tool for the effective usage of layouts in the UI screens. The tool is available in tools folder.

```
<AndroidSDK\tools>
```

The way to use the tool is

```
<AndroidSDK\tools>layoutopt [directories|files]
```

The below figure shows the output for some files.

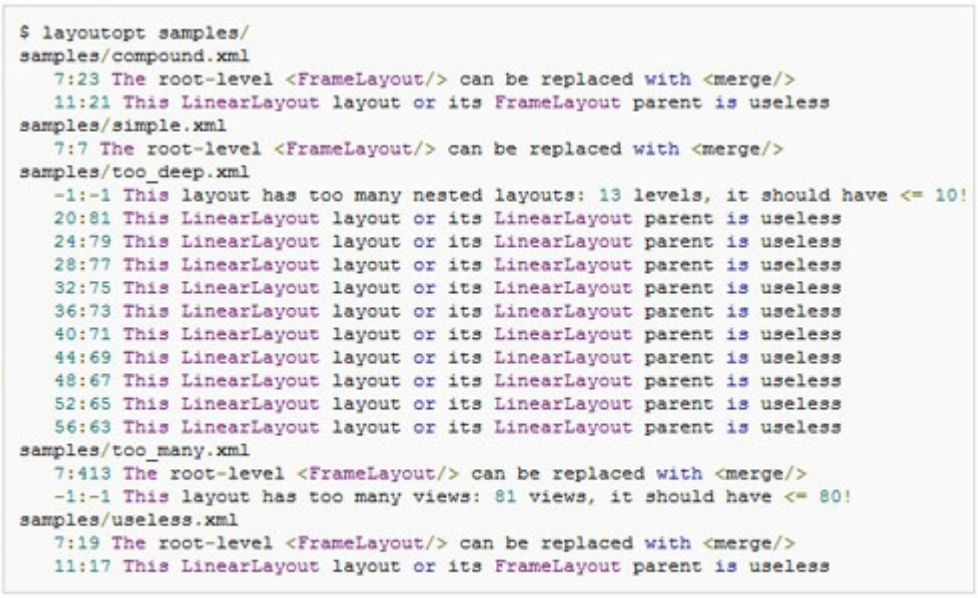

Figure 1: lauotopt

As shown in the above figure, the lauotopt tool suggests removing the useless layouts used in the application screens.

hierarchy viewer

The Android sdk provides one more tool for checking the layouts used in UI screens at runtime. The tool is available in the tools folder.

```
<AndroidSDK\tools>
```

The way to use the tool is as follows

```
<AndroidSDK\tools>hierarchyviewer
```

After executing the above command, a window will open with all the available target devices currently running. Then the user will need to select the particular target and select the particular process . After selecting the Load View Hierarchy button the process will load the hierarchy view in the opened window. The below figure shows hierarchy view for our sample screen.

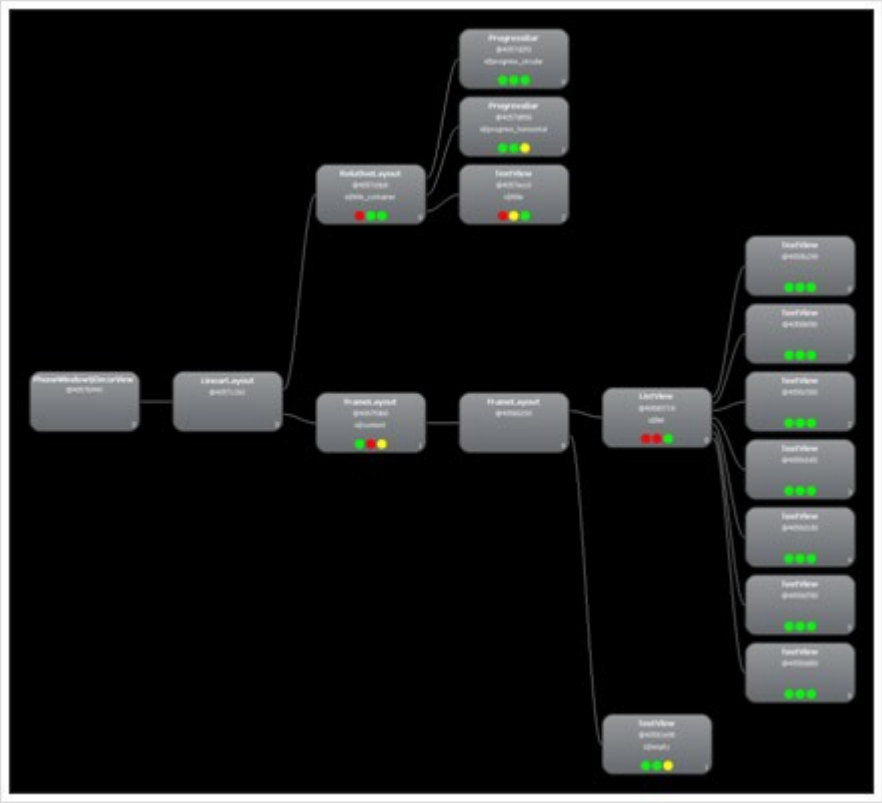

Figure 2: Hierarchy viewer

Layouts

There are so many layouts are available in Android for making UI screens. Here are a few useful tags listed to know how to efficiently create the UI screens.

Tag-include

So many times in development, you will need to show the same layout in the header and footer information. Android provides an <include> </include> tag to reuse the existing layouts. The below example shows the usage of the include tag.

```
<LinearLayout
                        android:id="@+id/mainparent"
                    android:layout_width="fill_parent"
                    android:layout_height="fill_parent">

                        <include layout="@layout/header" />
                        ......
                    <include layout="@layout/footer"/>
                <LinearLayout>
```

Tag-ViewStub

A ViewStub is an invisible, zero-sized View that can be used to lazily inflate layout resources at runtime. When a ViewStub is made visible, or when inflate() is invoked, the layout resource is inflated.

```
<ViewStub android:id="@+id/stub_import"
android:inflatedId="@+id/panel_import"
android:layout="@layout/controls"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:layout_gravity="bottom" />
```

The below figure shows the use of ViewStub.

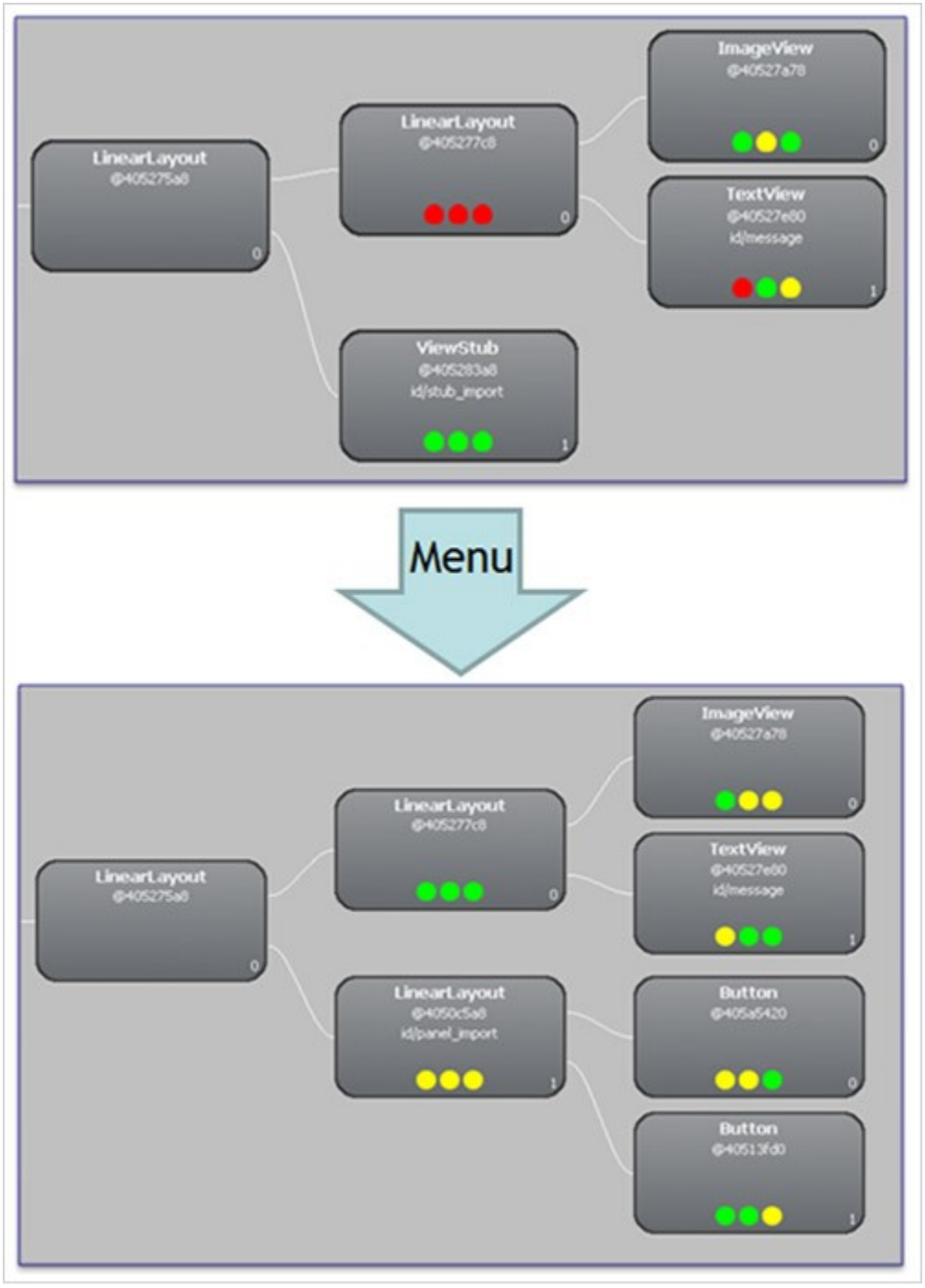

Figure 3: ViewStub

Tag-merge

Merge is an alternative root element that is not drawn in the layout hierarchy. Using this as the root element is useful when you know that this layout will be placed into a layout that already contains the appropriate parent View to contain the children of the <merge> element. The below code snippet shows the merge tag in usage.

```
<merge xmlns:android="http://schemas.android.com/apk/res/android">
                <ImageButton
                          android:layout_width="fill_parent"
                          android:layout_height="fill_parent"
                          android:src="@drawable/golden_gate" />

                <ImageButton
                          android:layout_width="wrap_content"
                          android:layout_height="wrap_content"
                          android:src="@drawable/golden_gate" />
                </merge>



                <merge>
                   <include />
                  <include />
                         ------
                <merge/>
```

The below figure shows the use of merge tag.

By using LinearLayout

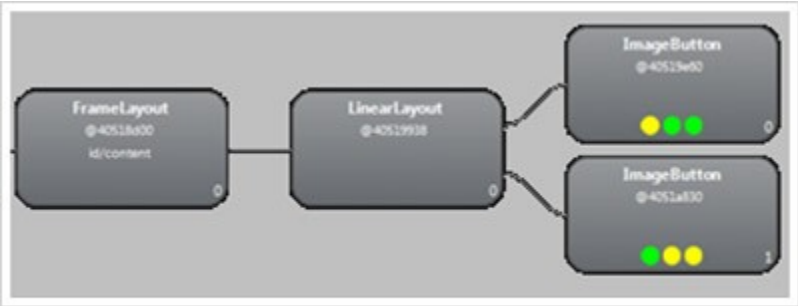

Figure 4: By LinearLayout

By using merge

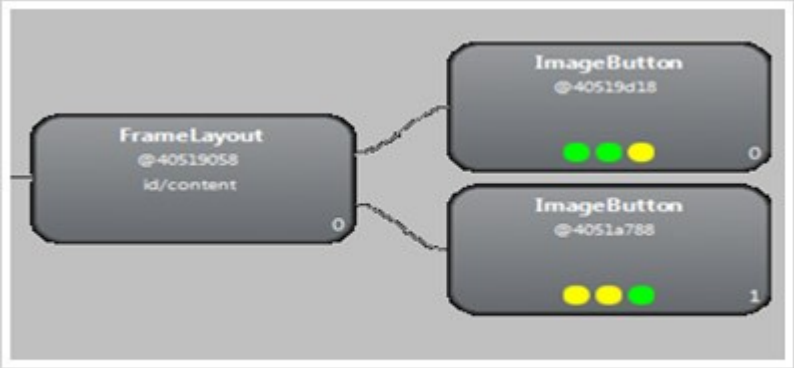

Figure 5: By Merge

LinearLayout vs RelativeLayout

LinearLayout

LinearLayout aligns all the children in a single direction — vertically or horizontally, depending on how you define the orientation attribute. All children are stacked one after the other, so a vertical list will only have one child per row, no matter how wide they are, and a horizontal list will only be one row high (the height of the tallest child, plus padding).

RelativeLayout

RelativeLayout lets child views specify their position relative to the parent view or to each other (specified by ID). So you can align two elements by right border, or make one below another, centered in the screen, centered left, and so on.

Lets examine a example by using Linearlayout and RelativeLayout. The below figure shows the requirement.

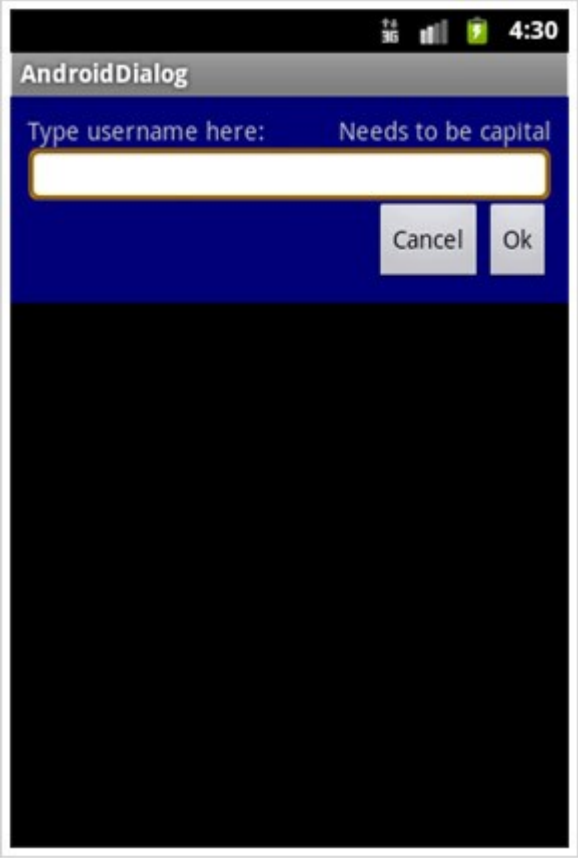

Figure 6: Layout example

The below figures shows the hierarchyview by using the Linearlayout and RelativeLayout for the shown example.

LinearLayout

Figure 7: LinearLayout Hierarchy Viewer

RelativeLayout

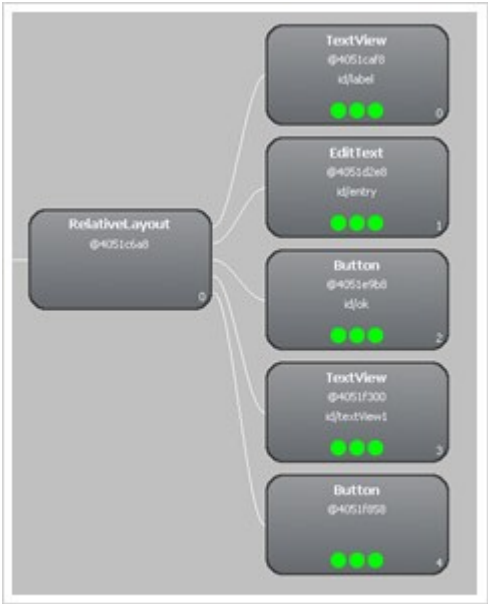

Figure 8: RelativeLayout Hierarchy Viewer

By examining the above hierarchyview anyone can see that RelativeLayout is more efficient than the LinearLayout. But its not the case in real time. The usecase of the layout purely depends on the requirement.

Finally let’s take a look on refreshing the views in UI screens.

Refreshing View

In general, for refreshing the UI invalidate () is called which will refresh the whole canvas screen. To refresh a fixed known area in the canvas screen use invalidate (Rect dirty). This dirty variable is used to specify the rectangle representing the bounds of the dirty region. Hence invalidate (Rect dirty) makes refreshing your UI much faster by refreshing only the specified region.

Ref:http://developer.samsung.com/android/technical-docs/Android-UI-Tips-and-Tricks#