REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Programmatic Layouts

Objectives After this section, you will be able to:

- Create activity layouts programmatically
- Incorporate layout resources into a programmatic layout
- Account for a device's screen resolution when programmatically setting screen dimensions

1.1 Declarative vs. Programmatic Layouts

In most situations, it's best to define an activity's layout declaratively with an XML layout resource rather than programmatically creating and configuring views.

- The declarative approach requires less code.
- The Eclipse ADT plugin provide a graphical layout editor allowing you to preview your layout as you create it.
- It's easier to provide alternate layouts to support multiple screen sizes and orientations.

On occasion, you might want to create all or part of a layout programmatically:

- An extremely simple layout (perhaps one `View`) that is the same for all screen sizes and orientations can be easy to create programmatically.
- You might want to update your layout dynamically at runtime to add or remove components.

Although the programmatic approach provides slightly better performance than the declarative approach, you shouldn't take the programmatic approach solely for performance. Layout resources — as with all of Android's XML-based resources — are pre-parsed into an efficient representation when the application is packaged into an APK file. Therefore the process of inflating the layout resource at runtime doesn't take significantly longer than creating the view hierarchy programmatically.

Additionally, virtually all activities create their layouts in the `onCreate()` method, which is invoked only once on creation of the `Activity` object. Therefore, the extra overhead of the layout resource inflation is incurred only on activity creation, not each time the activity becomes visible.

The other overhead of the declarative approach comes from searching the layout hierarchy for a view when you invoke `findViewById`. However, you can mitigate this by doing the lookup once in the activity's `onCreate()` method and caching the view references in instance fields.

1.2 Programmatic Layouts: The Basics

To create a layout programmatically, you must:

- Create create one or more views.
- Set the properties of those views.
- If you have more than one view, create one or more ViewGroups to contain the views.
- Create a single-rooted view hierarchy by adding the views as children to the ViewGroups, providing *layout parameters* to determine the size and position of each child view.
- Invoke `Activity.setContentView(View)`, passing the root view of your view hierarchy.

1.3 The Simplest Case: A One-View Layout

Creating a layout consisting of a single view is easy. For example:

```
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView tv = new TextView(this);
        tv.setText("Hello, world!");
        setContentView(tv);
}
```

In this example, the layout consists of a single `TextView`.

- By default, `setContentView()` sets both the height and width of the view to `MATCH_PARENT`, causing it to fill the entire activity window.

Note
Rather than using a hard-coded string, you should always define and use string resources for any user-visible string so that you have the option of localizing your app.

1.4 Setting Properties Programmatically

Any view property that you can set in an XML layout resource can be set programmatically.

- The reference page for each view class contains a table mapping the XML attributes to corresponding view methods.
- Remember that many attributes and methods may be inherited from ancestor classes, including the `View` base class.

For many properties, there is a simple 1-to-1 mapping from XML attributes to Java methods.

- For example, `android:visibility` is equivalent to `setVisibility()`; `android:gravity` is equivalent to `setGravity` `android:textColor` is equivalent to `setTextColor()`.

However, some XML attributes don't have a simple mapping to an equivalent Java method.

- For example, the `android:maxLength` attribute for `TextView` controls the maximum number of characters that can be entered into an editable `TextView` (in most cases, actually an instance of the `EditText` subclass). But programmatically, this is just one example of an *input filter* that you can install with the `setFilters()` method.

1.5 Setting Properties Programmatically

Building on the previous example, you could change the appearance of the `TextView` by setting some properties:

```
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView tv = new TextView(this);
        tv.setText("Hello, world!");
        tv.setTextColor(Color.WHITE);
        tv.setBackgroundColor(Color.BLUE);
        tv.setTextSize(20.0f); // Measured in sp units
        setContentView(tv);
}
```

1.6 Creating a `ViewGroup`

For a more complex layout, you need to create an instance of a `ViewGroup` subclass.

- You might need to set properties, like the orientation of a `LinearLayout`.
- Use the overloaded `addView()` method to add child views to the `ViewGroup`.
- Set the `ViewGroup` as the activity's content view.

By default, a new child view is added after all existing children of the `ViewGroup`.

- Optionally, you can specify a 0-based index of where you want to insert the new child.
- Existing child are "moved down" to make room for the new child.

Although the basic version of the `addView()` method adds a child view with a set of default layout parameters for that `ViewGroup` subclass, you typically want to provide an explicit set of layout parameters.

1.7 `ViewGroup` Layout Parameters

The layout parameters of a child view are specified by an instance of a `LayoutParams` object.

`ViewGroup.LayoutParams` provides a base set of parameters, including:

- `height`, `width` — The height and width of the view, expressed as `ViewGroup.MATCH_PARENT`, `ViewGroup.WRAP_CONTENT` or an integer number of pixels.

`ViewGroup.MarginLayoutParams` extends `ViewGroup.LayoutParams` with margin-related parameters:

- `bottomMargin`, `leftMargin`, `rightMargin`, `topMargin` — Margins expressed as an integer number of pixels.

`LinearLayout.LayoutParams` extends `ViewGroup.MarginLayoutParams` with additional parameters:

`gravity` — The gravity for the view, expressed as a constant from the `android.view.Gravity` class.

`weight` — The weight for allocation of extra space, expressed as a `float`.

1.8 A `LinearLayout` Example

The follow shows an example of using a `LinearLayout` to contain two children, and some simple examples of using layout parameters:

```
import static android.view.ViewGroup.LayoutParams.MATCH_PARENT;
import static android.view.ViewGroup.LayoutParams.WRAP_CONTENT;
import android.view.Gravity;

// ...

public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Main LinearLayout configuration
        mLayoutMain = new LinearLayout(this);
        mLayoutMain.setOrientation(LinearLayout.VERTICAL); // ❶
        setContentView(mLayoutMain);

        // Title TextView configuration
        mTextTitle = new TextView(this);
```

```
        mTextTitle.setTextSize(25.0f);  // 25sp
        mTextTitle.setText(R.string.activity_title);
        mTextTitle.setGravity(Gravity.CENTER_HORIZONTAL); // ❷

        // Title TextView layout
        mLayoutMain.addView(mTextTitle, MATCH_PARENT, WRAP_CONTENT); // ❸

        // Content EditText configuration
        mEditContent = new EditText(this);
        mEditContent.setHint(R.string.edit_content_hint);

        // Content EditText layout
        LinearLayout.LayoutParams editContentLayout
                = new LinearLayout.LayoutParams(MATCH_PARENT, 0, 1.0f); // ❹
        editContentLayout.topMargin = 5; // ❺
        mLayoutMain.addView(mEditContent, editContentLayout);
}
```

❶ Set the `LinearLayout` to a vertical orientation.

❷ Horizontally center the content of the `TextView`.

❸ This overloaded form of `addView()` sets the `height` to `MATCH_PARENT` and the `width` to `WRAP_CONTENT`.

❹ Create a `LayoutParams` object for the `EditText` with a `height` of `MATCH_PARENT`, a `width` of `0px`, and a `weight` of `1.0`.

❺ Include a `5px` top margin for the `EditText` layout.

1.9 "Cheating:" Inflating a Layout Resource

Sometimes you might want to simplicity of a declarative layout, but need to create the layout dynamically at runtime.

- In that case, you can use a `LayoutInflater` to inflate a layout at runtime.
- The layout resource you inflate can be an entire activity layout, or just a portion of it.
- You typically obtain a `LayoutInflator` by invoking `Activity.getLayoutInflator()`.

The `LayoutInflator.inflate()` method accepts the following arguments:

- The ID of a layout resource
- A reference to a `ViewGroup` that serves as a parent of the inflated layout hierarchy for the purposes of resolving layout parameters
- Optionally a boolean value to indicate whether the layout hierarchy should be attached to the parent `ViewGroup` (true, the default) or not (false)

1.10 Retaining View State by Assigning IDs

By default, a system runtime configuration change (for example, changing the orientation of the device) results in the system destroys and then re-creating activities.

- Before destroying an activity, the system invokes `Activity.onSaveInstanceState()` to store activity state information in a `Bundle`.
- The `Bundle` is provided to the new activity instance as an argument to its `onCreate()` and `onRestoreInstanceState()` methods.

The default implementation of `Activity.onSaveInstanceState()` traverses the activity's view hierarchy to store state information of the views.

- The method retains state information *only* for those views that have their ID properties set!
- The default `Activity.onRestoreInstanceState()` method then restores the state information to views with an ID.

To retain the state of a programmatically created view, you must invoke `View.setId(int)` to assign an integer ID to the view.

- The ID must be a positive integer value.
- The ID should be unique within the view hierarchy.

1.11 Retrieving Display Metrics

Many of the size layout parameters are expressed in pixel values.

- This can result in layouts that don't adapt well to different screen resolutions unless you scale your sizes based on the resolution.

You can determine the device's screen resolution by obtaining a `DisplayMetrics` object as follows:

```
DisplayMetrics metrics = new DisplayMetrics();
getWindowManager().getDefaultDisplay().getMetrics(metrics);
```

The `DisplayMetrics` object provides several public fields, including:

`density`	A `float` representing a scaling factor for the display relative to a 160 dpi base resolution
`scaledDensity`	A `float` representing a scaling factor for fonts on the display; similar to `density`, but also incorporating the user font size preference
`heightPixels`	The height of the screen in pixels
`widthPixels`	The width of the screen in pixels

1.12 Topic Summary

You should now be able to:

- Create activity layouts programmatically
- Incorporate layout resources into a programmatic layout
- Account for a device's screen resolution when programmatically setting screen dimensions