0

Java Essentials for Android Course

Labs >

Lab 2: Joke List

Intro

In this lab we will be learning how to use and extend the Android user interface library. In a number of ways it is very similar to the Java Swing library, and perhaps just as many ways different. While being familiar with Swing may help in some situations, it is not necessary. It is important to note that this lab is meant to be done in order, from start to finish. Each activity builds on the previous one, so skipping over earlier activities in the lab may cause you to miss an important lesson that you should be using in later activities.

Objectives

At the end of this lab you will be expected to know:

- *How to create an Android Test Project and test Android Activities.*
- *What Views, View Groups, Layouts, and Widgets are and how they relate to each other.*
- *How to declare layouts dynamically at runtime.*
- *How to reference resources in code and from other resource layout files.*

- *How to use Android's system debug output monitor LogCat for debugging.*
- *How to use Events and Event Listeners.*

Activities

For this lab we will be creating a "Joke List" application. It is a simple app that allows a user to view and edit a list of jokes. All tasks for this lab will be based off of this application. Over the course of the lab you will be iteratively refining and adding functionality to the Joke List app. With each iteration you will be either improving upon the previous iteration's functionality, or you will be implementing the same functionality in a different way.

IMPORTANT:
You will be given a Skeleton Project to work with. This project contains all of the java and resource files you will need to complete the lab. Some method stubs, member variables, and resource values and ids have been added as well. It is important that you not change the names of these methods, variables, and resource values and ids for the sake of testing.

In the Skeleton Project will be several test files. These unit tests will be used to evaluate the correctness of your lab. You have complete access to these test cases during development, which gives you the ability to run these tests yourself. You will learn how to create an Android Test Project, and will insert these test files into it for testing purposes.

Contents

1. Setting Up

1.1 Creating the Project

To begin, you will need to download and extract the skeleton project for the JokeList application.

- Click Here to download the skeleton project, then extract it, making sure to preserve the folder structure.
    - *Take note of the path to the root folder of the skeleton project. You may prefer to extract it to your Eclipse workspace directory if you want to keep all Eclipse projects in the same place.*

Next you will need to set up a "Joke List" Android project for this app, and the skeleton project will serve as that Android project. Since the skeleton project was created in Eclipse, the easiest thing to do is to import this project into Eclipse.

- Select **File -> Import...**
- In the Import Wizard, expand **General** and select **Existing Projects into Workspace**. Click **Next**.
- In the **Import Project** wizard, click **select root directory** and click **Browse**. Select the root directory of the skeleton project that you extracted. Make sure the project appears in the *Projects* area and is checked, then click **Finish**.
- Click on the project name in the Package Explorer. Select File -> Rename and change the name of your project to **lab2<userid>** where <userid> is your user id (e.g. jsmith).

1.2 Fill in the Joke Class

Throughout the lab you will be working with the Joke object class. It will serve as the data behind the Android components that will be visible on the application screen. You can see that it is a Plain Old Java Object--or POJO for short. All it has is

Open the **Joke.java** file under the **src** folder in the **edu.calpoly.android.lab2** package. At first glance, look at the member variables. This class will be used to encapsulate two items pertaining to jokes:

1. **String m_strJoke:** This represents the actual text of the joke.
2. **int m_nRating:** This represents a rating that can be assigned to a

joke. There are three possible values that the rating can take:

  - **UNRATED:** indicates no rating has been assigned to this joke by the user.
  - **LIKE:** indicates the user liked this joke.
  - **DISLIKE:** indicates the user did not like this joke.

Begin by filling in this class:

- Fill in all methods marked //TODO.

  - *Read the comments if you are confused as to the purpose of any of the methods.*

  - *Hint: When trying to quickly type out class variables and methods in Eclipse, using the **this** keyword may help since it brings up an auto-complete menu, plus it's not a bad idea to use it out of good practice. Read more about the this keyword here.*

  - *Hint: You can see all **//TODO** instances in Eclipse by viewing the left- and right-hand side of the Java file editor window for Joke.java. On the left-hand side, a blue checkbox appears next to each **//TODO** in the file. This is similar to how a yellow warning light bulb and red 'x' appear next to each line that has a warning and an error, respectively. The right-hand side shows a more condensed version of these indicators.*

    - *Hint: If you see a yellow and red 'x' light bulb indicator appearing next to an error, it means that Eclipse may be able to help you resolve the problem easily. Hovering over the indicator will tell you the error, and clicking on the indicator will roll out any suggestions Eclipse has for resolving the error. Use this feature judiciously, as it is helpful but does not always provide the most appropriate* *suggestions!*

1.3 Create Android Test Project

To make sure the Joke class has been implemented correctly, several unit test files have been provided to you in the Skeleton Project, under the **test** folder. Now we will place these test files in a proper Android Test Project. But first, we must create the Android Test Project.

- In Eclipse, choose the menu item **File** -> **New** -> **Project...**
- Expand the Android folder, choose **Android Test Project** and click **Next**.
- For Project Name, put lab2test<userid> to indicate that this project is a test project for the Skeleton Project.
- You may change the location that the Android Test Project is stored. It is best to store it at the same level/location as the Skeleton Project (not inside of it, but at the same directory).
- For Test Target, make sure **An existing Android project** is selected, then select your Skeleton Project for this lab (**lab2<userid>**) in the list of Android projects that appear.
- Leave the Build Target as it is, and then click **Finish**.

```
Why are we creating a brand new project just for testing?

This is one of two ways to execute this practice. We will demonstrate
and stick with the other option in the next lab and the remainder of
the labs. However, the choice of how to run tests when doing your own
Android development is up to you. This question is addressed further
```

```
here.
```

Now you have an empty Android Test Project in your Package Explorer as well. It looks very similar in structure to the Skeleton Project; this is done on purpose. Check the **src** folder in the Android Test Project and you will notice a new package: **edu.calpoly.android.lab2.test**. It is empty, but we will now fill it with tests!

- In your Skeleton Project, expand the **test** folder and select both JokeTest.java and SimpleJokeList.java. Copy (right-click on one of the files, and select **Copy**) both files.
- In your Android Test Project, expand the **src** folder and click on the package **edu.calpoly.android.lab2.test**, then Paste (right-click on the package, and select **Paste**) both files into the package.

There, now the tests are all set up. Now we'll take a quick look at the settings for running tests. For now, we will just test the Joke.java file to make sure it is implemented correctly.

- Right click on the Android Test Project folder in the Package Explorer (**lab2test<userid>**) and select **Run As** -> **Run Configurations...**
- Make a new run configuration of type **Android JUnit Test**.
- In the Test tab on the right side of the Run Configurations window that appears, select **Run all tests in the selected project, or package** and make sure **lab2test<userid>** is selected. Don't close the Run Configurations window yet.

    - *NOTE: If you'd like to keep track of your Run Configurations, you should name them according to what project they pertain to. You can name each configuration after the lab you are working on, or however else is convenient for you to remember.*

- Now the Android Test Project is set to run all tests in all test files present. Next, choose the Target for running the tests. ADT assumes you will be running the tests on an Android device. This is not the case for the Joke tests, but when you run the tests for SimpleJokeList that is exactly what will be happening, so you will set the configurations to do just that.
    - In the **Target** tab, choose the appropriate device to run tests with. Depending on whether you use a physical device, an AVD, or want to be given the choice each time, choose the appropriate option.

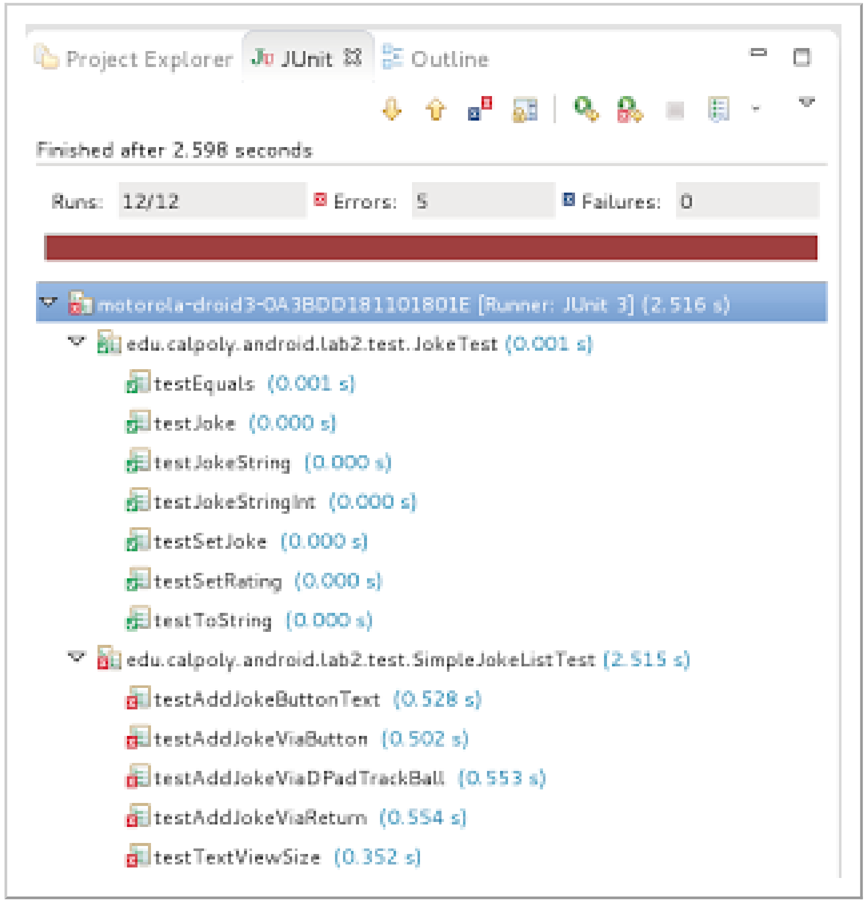
- Click **Apply** to save your changes, then click **Close**.

Now when you click Run after clicking on or anywhere inside the **lab2test<userid>** project in the Package Explorer, the application will attempt to initialize and deploy, and tests will be run. So what are we waiting for? Let's run them!

- In the Android Test Project, right-click the **JokeTest.java** file under the **src** folder in the **edu.calpoly.android.lab2.test** package **and select** select **Run as -> Android JUnit Test**.

   - *Note: It may take several attempts, if you are booting up an AVD.*

This should open up a JUnit tab next to the Package Explorer. You'll see some red--this is expected, since you haven't implemented the SimpleJokeList.java file yet. If you've implemented every method in the Joke.java file correctly, you should see something similar to the following (click image for full size):



Project Explorer | JUnit | Outline

```
Finished after 2.598 seconds

Runs: 12/12    Errors: 5    Failures: 0

motorola-droid3-0A3BDD181101801E [Runner: JUnit 3] (2.516 s)
  edu.calpoly.android.lab2.test.JokeTest (0.001 s)
    testEquals (0.001 s)
    testJoke (0.000 s)
    testJokeString (0.000 s)
    testJokeStringInt (0.000 s)
    testSetJoke (0.000 s)
    testSetRating (0.000 s)
    testToString (0.000 s)
  edu.calpoly.android.lab2.test.SimpleJokeListTest (2.515 s)
    testAddJokeButtonText (0.528 s)
    testAddJokeViaButton (0.502 s)
    testAddJokeViaDPadTrackBall (0.553 s)
    testAddJokeViaReturn (0.554 s)
    testTextViewSize (0.352 s)
```

The tests in the above image were initiated using a Droid 3 physical device.

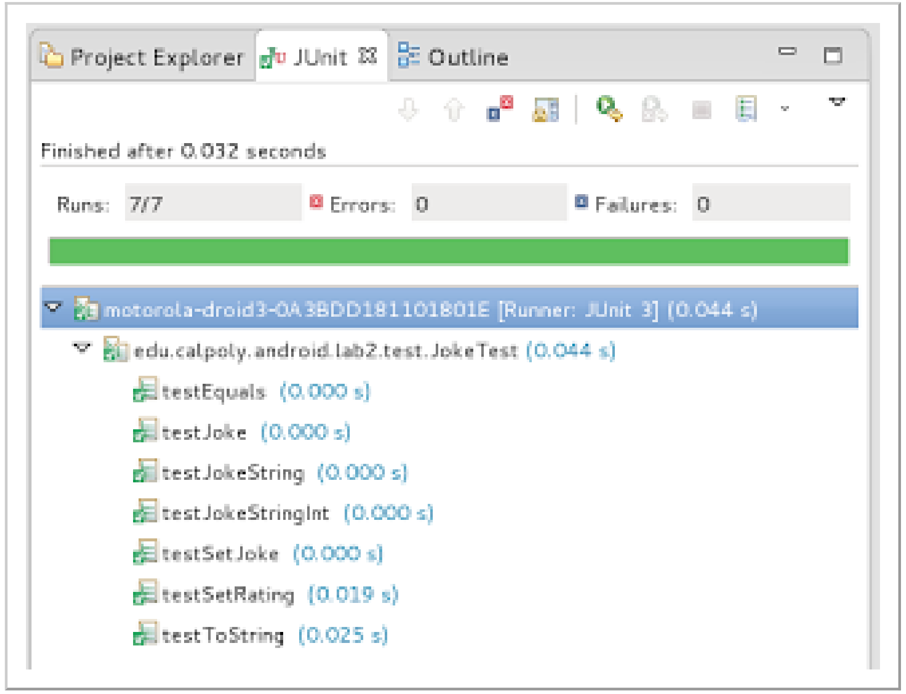
Now let's just run the JokeTest tests, since we know that SimpleJokeListTest is not implemented yet.

- In the **src** folder of the Android Test Project, right-click on **JokeTest.java** and choose **Run As** -> **Android JUnit Test**.
    - The reason why you want to avoid Run As -> JUnit Test is because dealing with multiple launchers is potentially a nightmare. You may try this option, but it is likely that you will

receive NullPointerExceptions and/or core dumps. If this happens, right-click on **JokeTest.java** and choose **Run As -> Run Configurations...** and you will see the new configuration for running just JokeTest. Delete it and run it as indicated above.

- You will likely see a dialogue box pop up that asks you to specify a launcher. Check the **Use configuration specific settings** box and choose **Eclipse JUnit Launcher**, then click **OK**.

This will only run **JokeTest.java** and ignore **SimpleJokeListTest.java**. If you've implemented every method in the Joke.java file correctly, you should see the following (click image for full size):



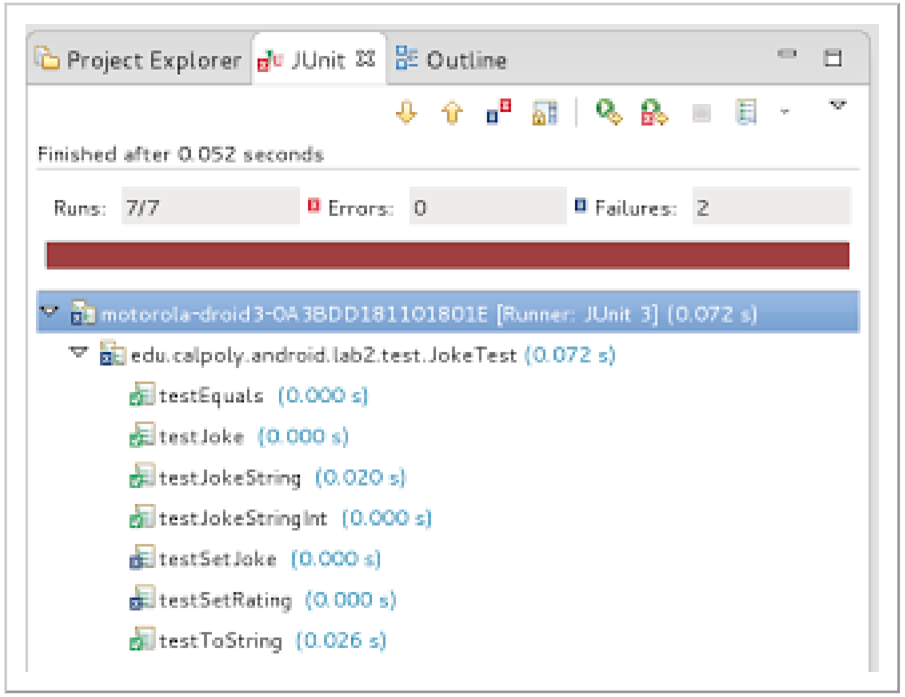
Finished after 0.032 seconds

Runs: 7/7 Errors: 0 Failures: 0

```
motorola-droid3-0A3BDD181101801E [Runner: JUnit 3] (0.044 s)
  edu.calpoly.android.lab2.test.JokeTest (0.044 s)
    testEquals (0.000 s)
    testJoke (0.000 s)
    testJokeString (0.000 s)
    testJokeStringInt (0.000 s)
    testSetJoke (0.000 s)
    testSetRating (0.019 s)
    testToString (0.025 s)
```

Success!

If you see a red bar across the top, this means that you've failed one or more tests. The tab should show how many tests failed, and which tests you failed:


Project Explorer JUnit Outline

Finished after 0.052 seconds

Runs: 7/7 Errors: 0 Failures: 2

```
motorola-droid3-0A3BDD181101801E [Runner: JUnit 3] (0.072 s)
  edu.calpoly.android.lab2.test.JokeTest (0.072 s)
    testEquals (0.000 s)
    testJoke (0.000 s)
    testJokeString (0.020 s)
    testJokeStringInt (0.000 s)
    testSetJoke (0.000 s)
    testSetRating (0.000 s)
    testToString (0.026 s)
```

Need to fix up a few things.

Feel free to open up the JokeTest.java file to look at what exactly is being tested. Make the appropriate corrections to Joke.java and rerun the tests until you've passed all of them. We'll worry about the SimpleJokeListTest.java tests later.

1.4 Retrieving the Joke Resources Strings

The skeleton project has been pre-populated with an array of three different String resources you can use as sample jokes. For a complete background on Resources and how to properly use them you can read the Resources Overview.

- Back in your skeleton project, open up **res/values/strings.xml** in the XML editor to view the joke resources.
    - Notice the `<string-array name="jokeList">` element. This is how you declare an array of strings in strings.xml.
        - Notice the **name** attribute, which is set to "jokeList".
            - *When you add an element to a resource file, ADT will automatically add a static constant with this variable name to the R.java file. You can then use this constant as an identifier for retrieving the resource element through a Resource Object.*
- Open up the **R.java** file under the **gen** folder in the **edu.calpoly.android.lab2** package.
    - Notice that R is a static class with a static subclass for each type of resource element.
    - If you were to add a string resource element and give it a **name** attribute, a static constant with this name gets added to the **R.string** class. Arrays get added to the **R.array** class, drawables get added to the **R.drawable** class, etc.
        - The constant has the same name as the name attribute. Layouts are slightly different in that you don't have to specify a name attribute; the name of the layout constant will be automatically generated in R.java as the name of the XML layout file (e.g., **layout_helloworld.xml** would appear in R.java as **layout_helloworld**).

        - The constant contains the resource id that you can use to retrieve the resource.

You need to display these jokes when the application starts up. When an Activity first starts up, its **onCreate()** method is always called. This method allows you to initialize the Activity. Right now you will only be initializing local variables to hold the jokes that you just entered:

- Open **SimpleJokeList.java**.

- In the **public void onCreate(Bundle savedInstanceState)** method:

    - Notice the `super.onCreate(savedInstance)` call. This is crucial, **always** include this call before anything else in the method. If you don't your Activity won't work.

    - Make a call to `this.getResources()`, which will return a Resources object.

        - The Resources class provides an interface for retrieving resources by their **resourceID**.

        - **resourceID's** can be found in the static **R** class, under their respective resource subclasses (which are named after their resource type), under the name given to them in the resource file: `R.array.jokeList`

    - Initialize the class variable **m_arrJokeList** with a new instance of an ArrayList of Jokes.

    - Retrieve the array of joke strings by calling `getStringArray(R.array.jokeList)` on the resource object.

    - For each of these strings make a call to **addJoke()**, which will initialize Joke objects and place them in **m_arrJokeList**

(You will have to fill in the **addJoke()** method with minimal code for now to make it add a single Joke to the list of Jokes, and call this for each String in the array of joke strings obtained earlier).

When creating a new Activity you will almost always override the **onCreate()** method. The **onCreate()** method, as you will see later, is the place where you will be creating your User Interfaces, binding data to different UI controls, and starting helper threads (this will not be covered in this lab). Also, take note of the "Bundle savedInstanceState" parameter that gets passed in. This variable contains information on the previous state of the UI. As you may have assumed, this means that you are able to save the state of the UI before it closes so that it can be initialized to the same state the next time it is created.

2. LogCat

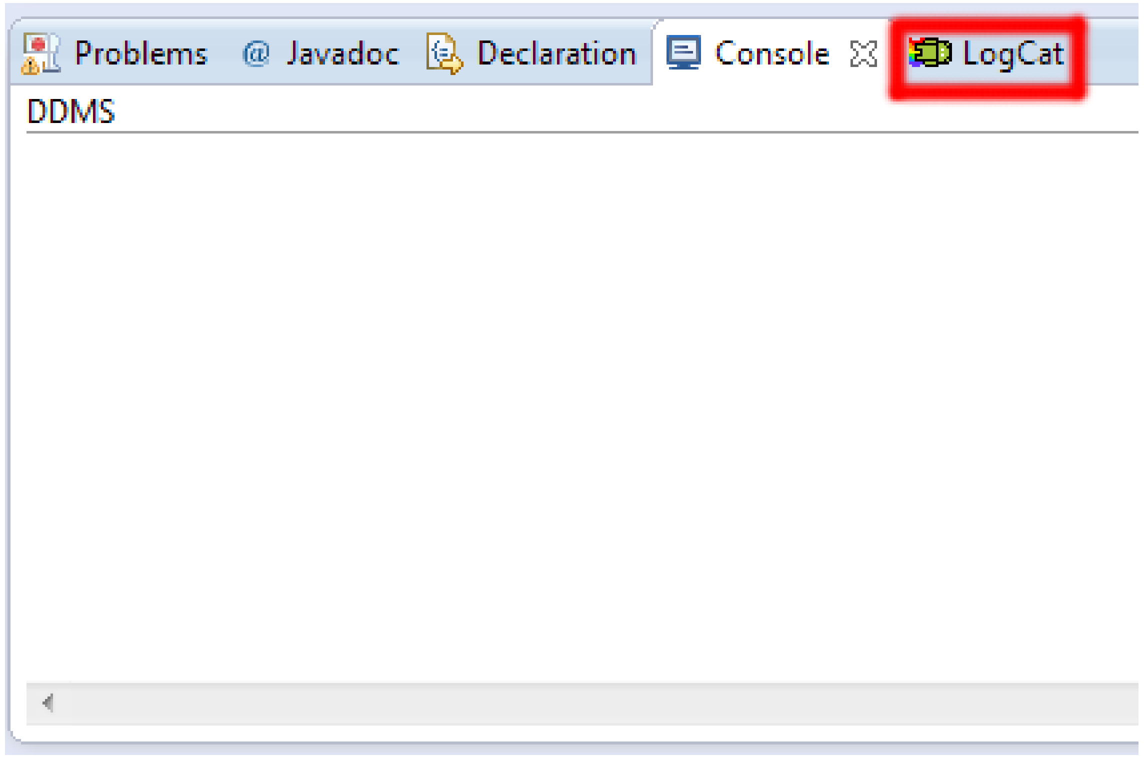
Wouldn't it be wonderful to debug your Android code while programming? Android has something for that, too.

Android uses a special debugging framework called LogCat, which will show every message broadcast from every Activity on the Android device in real time! You can use this framework to send messages from your own application containing various useful information for debugging purposes, such as print statements.

Why use a special debugging framework? It allows for more flexible, fleshed-out feedback tied directly to Android. It helps contain all debugging in a single location, instead of spread out across multiple areas. For example, inserting *print* statements into code (a very common, effective debug method) to print messages out to the Console (usually located in the bottom tabbed window in Eclipse) will reveal nothing when running an Android application. In the context of Android, what you usually see in the Console is status regarding Activity launching, but not any debug statements. Any Console output expected from an Android application will not appear! (Unless you are using an AVD).

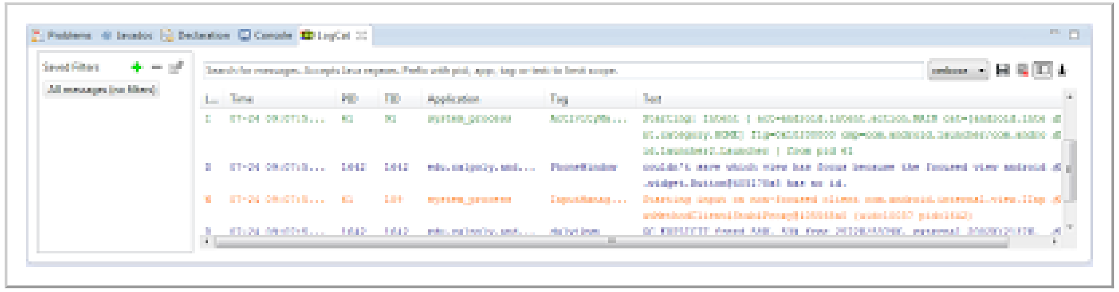
Note: For more in-depth debugging of Android applications in Eclipse, read more about the Dalvik Debug Monitor Server (DDMS) and *Java Debugger.*

- Run your skeleton project as an Android Application. In Eclipse, you will see a tab that contains several listings including Problems, Console and the one we are after, LogCat (see image below):



The LogCat tab.

You will see activity such as this in the LogCat tab *(Click image for larger view)*:



Every message being sent from every process on the device, with details.

This is where every single message from the device will appear, including messages from your application (this includes fatal errors, which appear in red).

- Click the **Clear Log** button (upper-right corner) to clear the LogCat message window.

- Rerun your application, and observe the LogCat message window filling up.

    - Notice the different attributes of each message, such as Level, Time, Tag and Text. Reading the Text of a message may give some insight as to what's going on when the message is sent, but chances are good you won't understand a majority of them right now.

    - Notice the color of each message. Each color corresponds to a type or *Level* of message, such as blue (D) for Debug, green (I) for Info, red (E) for Error, etc. Depending on what messages you want to see, you can filter them by choosing a message

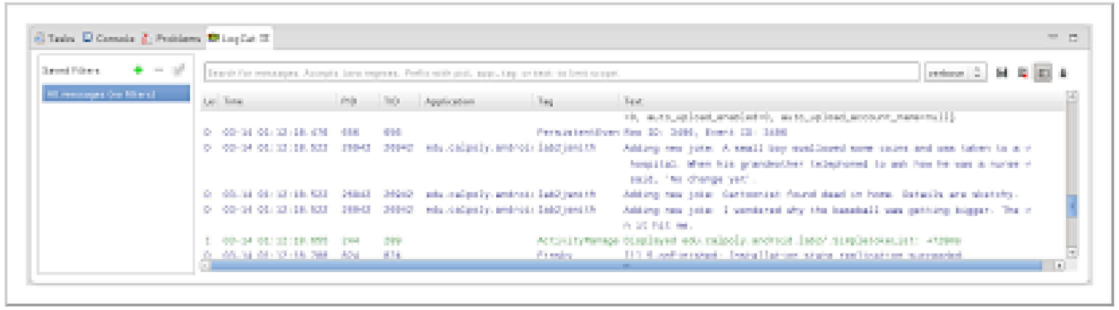
Level from the dropdown menu near the upper-right corner (**verbose** by default, which reports just about every message). For more information about Message levels, read up on them here.

  - You can also save logs of selected messages to an external file, or search for specific messages given certain prefixes for additional filtering such as **text:**. You can even create and save preset filters to search for messages specific to a tag, message, PID, Application and/or log level of your choice!

Now you will make your application generate some messages. You will send a message to LogCat for each Joke that is added to the Joke List using the Log class.

- In SimpleJokeList.java, just before adding each joke to the Joke list, add a *Debug* Log call using **Log.d()**, passing in "lab2<userid>" as the tag and the text of the joke (something like "Adding new joke: " + strJoke) as the message.

- Clear LogCat of messages, then rerun your application. You should see the three Debug level messages appear in LogCat like in the image below (click image for larger view):


Simple and elegant.

If you are ever confused on what your application is doing wrong, consider using LogCat even to just display information about your variables or to simply echo messages to make sure certain methods or lines of code are being reached.

Note: Be mindful of what message Level you use if you go on to publish applications! ***Debug*** *is a safe Level for avoiding accidental publication of debug-level information.*

3. Brief Background on View Classes

In Android, a user interface is a hierarchy composed of different View objects. The View class serves as the base class for all graphical elements, of which there are two main types:

- **Widgets:** Can either be individual, or groups of UI elements. These are things like buttons, text fields, and labels. Widgets directly extend the View class.

- **Layouts:** Provide a means of arranging UI elements on the screen. These are things like a table layout or a linear layout. Layouts extend the ViewGroup class, which in turn extends the View class.

Layouts are all subclasses of the ViewGroup class and their main purpose is to control the position of all the child views they contain. Layouts can be nested within other layouts as well, to create complex user interfaces. Some of the common layout objects you will use are:

- **FrameLayout:** Takes a single view object and simply pins it to the upper left hand corner of the screen. Each child view that is added is drawn on top of the previous one, causing it to be completely or partially obscured without very careful screen size and view hierarchy arrangement.

- **LinearLayout:** Has a list of child views and draws them sequentially in a single direction, either horizontally or vertically. You have the option of assigning a weight value for each child view which determines how much it is allowed to grow if there is extra space.

- **TableLayout:** Positions its child views in a grid of rows and columns. A row is a child view specified by the TableRow class. TableRows can have zero or more cells and can contain empty cells. However, a cell cannot span multiple columns. Each cell is itself another view, like a Button, and can be set to shrink or grow.

- **RelativeLayout:** Positions its child views relative to other child views or the parent view. For example, two child views can be left-justified in the parent, or one child view can be made to be below another.

- **AbsoluteLayout:** Has an absolute (x,y) coordinate position for each child view. This is a rigid layout that pins all child views down exactly where they are specified. Using this does not allow the user interface to adjust for different screen sizes and resolutions.

- *See Common Layout Objects for picture examples and more information about these Layouts and more.*

Declaring the layouts for your user interface can be done dynamically (in code), statically (via an XML resource file, such as in Lab 1), or any combination of the two. In the following section you will build a set of user interfaces dynamically in code. In future labs, you will build a set of user interfaces in XML for static use as this is an overall better practice. It's good to know both ways!

4. SimpleJokeList

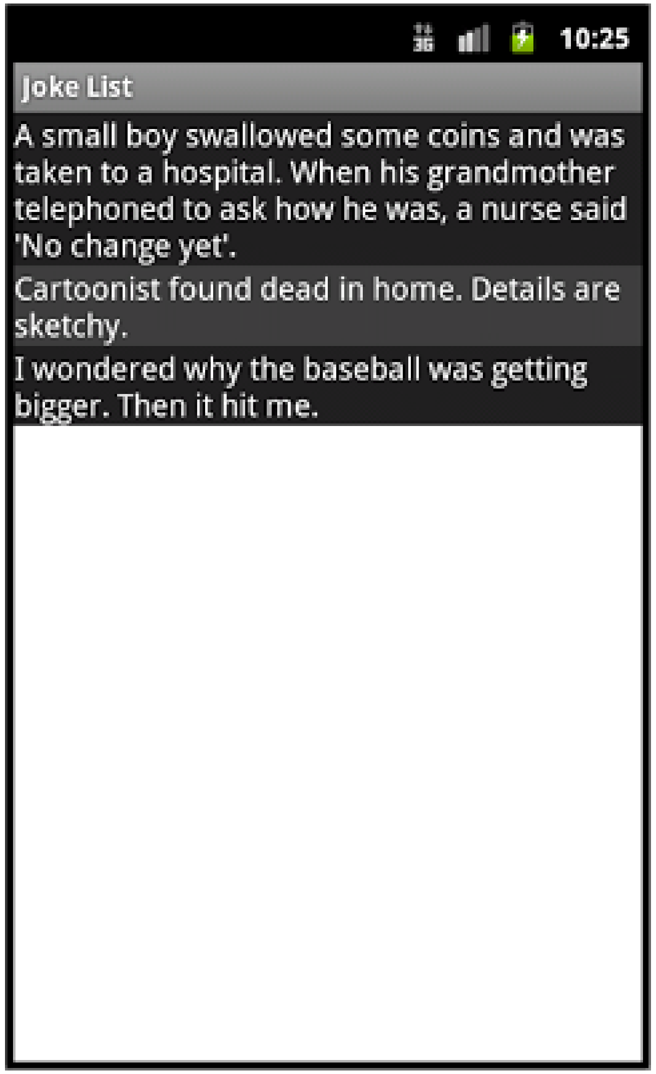
Work done in this section will be limited to the SimpleJokeList.java file. SimpleJokeList is an Activity class which displays a vertical scrollable list of all the Jokes in **m_arrJokeList**. Additionally, it has the ability to add jokes to **m_arrJokeList** via a text field and add a button that floats at the top of the screen.

4.1 Declaring Dynamic Layouts in Code

If you've ever built UIs in Java using Swing, this should be somewhat familiar. You generally want to try to define the layouts for your interface using XML when possible, but there are plenty of instances where it is still necessary to do it at run-time in code. For instance, you may want to dynamically change the layout based on some type of user input. Nevertheless, working with the actual code is good practice for understanding how the different Layout classes work and what sorts of interfaces all the various controls offer.

4.1.1 Display a Scrollable Vertical List

Your first Task is to simply display each of the jokes in the **strings.xml** resource file in a scrollable vertical list. When finished your application should look something like this:



A small boy swallowed some coins and was taken to a hospital. When his grandmother telephoned to ask how he was, a nurse said 'No change yet'.

Cartoonist found dead in home. Details are sketchy.

I wondered why the baseball was getting bigger. Then it hit me.

Looks simple, doesn't it?

Fill in the **initLayout()** method in SimpleJokeList.java:

- Initialize the **m_vwJokeLayout** LinearLayout member variable.
    - *When constructing View objects, you generally have to pass a **Context** object into the constructor. A Context object provides the functionality for accessing resources (like our jokeList), databases, and preferences. The*

    - *Activity class inherits from Context, which is how we were able to retrieve our jokeList by calling this.getResources().*

    - *Hint: You may pass in the Activity itself (**this**) when you need to pass in a Context object.*

- The LinearLayout class displays its child views horizontally by default. Make sure to change this to vertical using the **setOrientation()** method, passing in the LinearLayout.VERTICAL constant.

    - *LinearLayout documentation can be found here.*

- Create and initialize a local ScrollView object.

    - *A ScrollView is merely a FrameLayout that becomes scrollable when its child views are larger than the screen area. It generally has a single child view, which in our case will be another layout manager.*

- Add the LinearLayout to the ScrollView by calling its **addView()** method.

- Call **setContentView()**, passing in your ScrollView.

    - *setContentView provides the content which the Activity is supposed to display. Our UI is a hierarchical structure of nested components. We want to pass in the top-level, or root, element to this method. In our case this is the ScrollView.*

- Make a call to **initLayout()** in your **onCreate()** method and place it beneath the **super.onCreate()** call.

Update the **addJoke()** method:

- Add the joke text to a new TextView object by calling its **setText()** method.

- Add the TextView to **m_vwJokeLayout**.

Your **AndroidManifest.xml** is currently set up so that the SimpleJokeList Activity will launch on startup. Modify the Run Configuration to specifically launch the SimpleJokeList Activity:

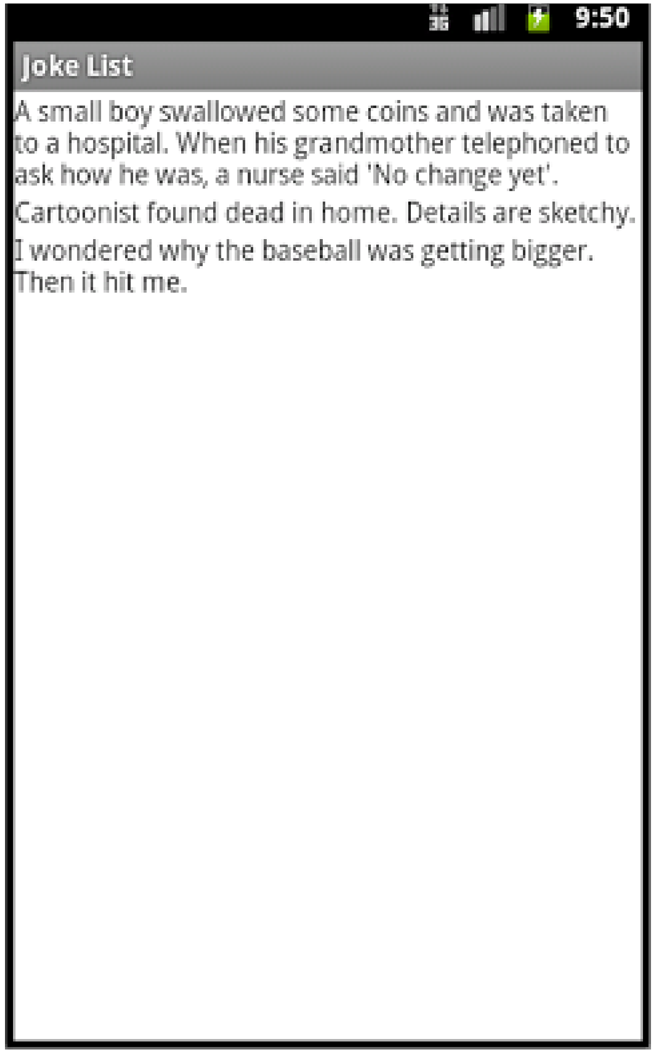
- Select **Run** -> **Run Configurations...** from the menu, or right click on the project in the Package Explorer and select **Run As** -> **Run Configurations...** from the menu.

- If there is no Run Configuration yet for this lab, perform the following:

    - Select the **Android Application** item from the list on the left.
    - Click the **New launch configuration** icon in the top-left corner of the Run Configurations window.
    - Give the configuration a name and the name of your android project.

- Select the **Launch** radio button and select the **SimpleJokeList** activity from the dropdown list.

- Make adjustments for running this application in the Target tab depending on your target Android device and preferences, then click **Apply** and then **Run** to verify that your list of jokes shows up.


9:50

Joke List

A small boy swallowed some coins and was taken to a hospital. When his grandmother telephoned to ask how he was, a nurse said 'No change yet'.

Cartoonist found dead in home. Details are sketchy.

I wondered why the baseball was getting bigger. Then it hit me.

It's a start, but it's a bit hard to tell where one joke ends and another begins...

The text is a little small, so use TextView's **setTextSize(...)** to increase the size of all jokes to 16 font. Note that there are two different method signatures for **setTextSize()**. Feel free to mess around with the font size and different TypedValue constants until the text looks large enough on the device you are running this application on.

- *For the images of the application in this lab, the remainder of the pictures use **COMPLEX_UNIT_PX** and **16** as the input of this method since one of the unit tests checks for this unit size. PX sets the text to use **raw pixels** for the text size, but the size of the joke text on your device may be too big or small with these parameters. If you need to increase the joke text size on your device further, feel free to change the unit test to reflect this as covered in this step.*

It's somewhat hard to distinguish one joke from another, so let's alternate the background colors of each joke. We'll store the color values as resources:

- Like you did in Lab 1, create a new XML Values file in your project called **colors.xml**.

You can use either the Resource Editor or you can directly add the following XML to the file:

```
<color name="light">#1F1F1F</color>
<color name="dark">#3D3D3D</color>
```

  - *If adding the XML yourself, make sure to nest the color tags inside the <resources> tags.*

Back in SimpleJokeList, modify the background color of the TextViews by calling **setBackgroundColor** as you create them in your **addJoke(...)** method.

- Alternate between setting the background color to **dark** and **light**.

- *Hint: First you must initialize the variables. Retrieve the colors in the same way you retrieved the joke strings. Notice the convenient class variables **m_nDarkColor** and **m_nLightColor**, just waiting to be used...*

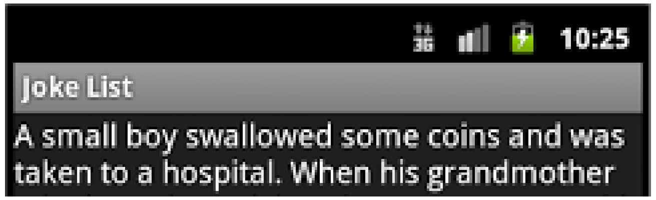
if you run the application now, the text may be too faded to read easily. Let's make the text completely white.

- Add another color resource to colors.xml called text, with the value #FFFFFF.

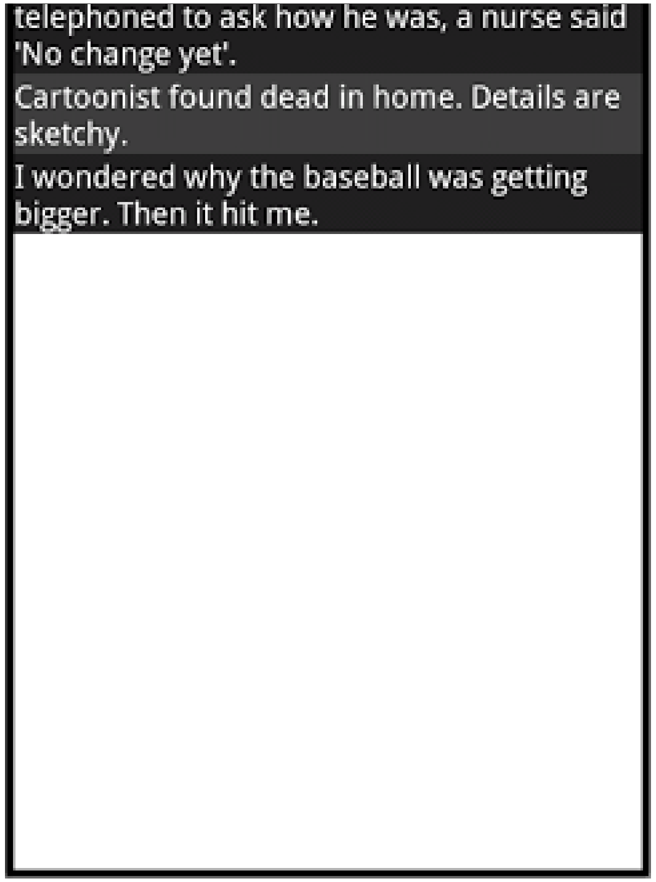
- Add a new class variable in SimpleJokeList.java of type int named m_nTextColor, similar to the already existing **m_nDarkColor** and **m_nLightColor** variables. Initialize it the same way, as well.

- Every time a new Joke is added to the list, change the text color to **m_nTextColor**.

Run the application now, and the Jokes should be much more distinguishable with proper text coloring:



A small boy swallowed some coins and was taken to a hospital. When his grandmother



telephoned to ask how he was, a nurse said 'No change yet'.

Cartoonist found dead in home. Details are sketchy.

I wondered why the baseball was getting bigger. Then it hit me.

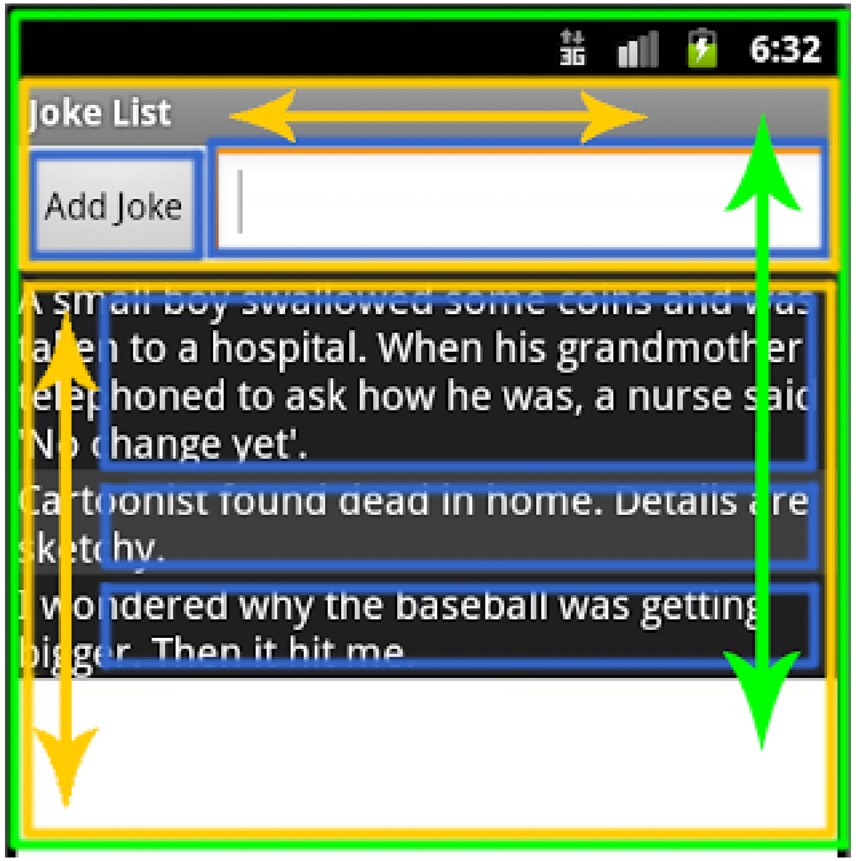
Now that's MUCH better!

Feel free to experiment with the text color and alternating background colors (you can even add more) so it is pleasant and satisfying for your own viewing purposes.

4.1.2 Adding Your Own Jokes

In this next task we will give the user the ability to enter their own jokes by adding a text field and a button to the UI. The text field and button should float at the top of the screen, above the list of jokes. When you scroll through the jokes, the text field and button should not scroll, but remain at the top of the screen.

In order to accomplish this we will walk through a process of nesting LinearLayouts within each other to achieve the desired effect. However, feel free to experiment and use whatever type of ViewGroup Layout you want, so long as the functionality and appearance remains the same. In the figure shown below, you can see how a ***root*** *vertically-oriented LinearLayout* containing a *horizontally-oriented LinearLayout* and the *ScrollView* of jokes can be combined to achieve the effect we want.



A small boy swallowed some coins and was taken to a hospital. When his grandmother telephoned to ask how he was, a nurse said 'No change yet'.

Cartoonist found dead in home. Details are sketchy.

I wondered why the baseball was getting bigger. Then it hit me.

Green: Root ViewGroup. Gold: Child ViewGroups. Blue: Child View components.

Note that the app title, Joke List, is not actually a part of the first of two nested LinearLayouts. The first child ViewGroup in the above image was only extended to make room for the orientation arrow.

In this step you will only be updating your Layout. The new "Add Joke" button and text field won't work yet. You will add code to hook up those controls in the next section.

- *All of the work here will be done in the **initLayout()** method.*
- Declare a local vertically-oriented LinearLayout object, this will be your root ViewGroup *(displayed in green in the figure above)*.
- Declare a local horizontally-oriented LinearLayout object, this will be the ViewGroup containing the "Add Joke" button and text field.
    - Initialize your m_vwJokeButton member variable, setting its text to Add Joke.
        - Add the Button to the horizontal LinearLayout.
    - Initialize your **m_vwJokeEditText** member variable.
        - call **m_vwJokeEditText.setLayoutParams()**, passing in an appropriately initialized LinearLayout.LayoutParams object so that the EditText will take up all of the extra available width and height.
        - call **m_vwJokeEditText.setInputType()**, passing in **InputType.TYPE_CLASS_TEXT**. This will ensure that clicking the Done/Enter key on the soft keyboard works properly on old and new devices.
        - *For more information see the Android Developer Guide and Documentation on:*
            - Layout Parameters
            - How Android Draws Views
            - LinearLayout.LayoutParams

        - Add the EditText to the horizontal LinearLayout.
    -
- Add your horizontal LinearLayout to your root ViewGroup.
- After your declaration and initialization of your ScrollView ViewGroup, add the ScrollView to your root ViewGroup.
- Change your call to **setContentView()** to pass in your new root ViewGroup.

Run your application to test that you have properly setup your layout. You should see a text field and button at the top, followed by the list of jokes below. Trying to add jokes with the Add Joke button won't work yet.

4.2 Simple UI Event Listeners

Now that your UI has the components necessary to allow users to enter new jokes, you need to hook these components up. In this section you will define and register event listeners to handle adding new jokes.

Read the Android Developers Guide on Handling UI Events to get a detailed background on event handlers and listeners. In short, you use event listeners to respond to user interaction with your UI. There are two general steps for doing this. The first is to define an object that implements an interface for the type of interaction you want to respond. The second step is to register that object with the UI control you want to respond to. For example, to react to a particular Button being clicked, you have to register an object that implements the OnClickListener interface with the Button you want to listen to.

4.2.1 Adding a Button Listener

Setup the **onClickListener** for the "Add Joke" Button. When the user hits the "Add Joke" Button a new Joke object should be initialized with the text from the EditText field and added to your joke list. If the EditText is empty, then the Button should do nothing. In **initAddJokeListeners()**:

- Call **setOnClickListener** method for you **m_vwJokeButton** member variable. Pass in a reference to what's called an *Anonymous Inner Class* that implements the OnClickListener interface.
    - *If you are unfamiliar with anonymous inner classes that's alright. Essentially, its an inline way of creating a one-time-use class that implements some interface. You declare the class and instantiate it in one motion. You can read more about them from Sun's Java Tutorials on Anonymous Inner Classes.*
    - Copy the following code:

```
m_vwJokeButton.setOnClickListener(new
OnClickListener() {


 public void onClick(View view) {

        //Implement code to add a new joke here...

 }
});
```

    - Fill in the **onClick** method in the anonymous inner class that you created to add a new joke to the list.
        - Retrieve the text entered by the user into the EditText by calling **getText().toString()** (*Check for*

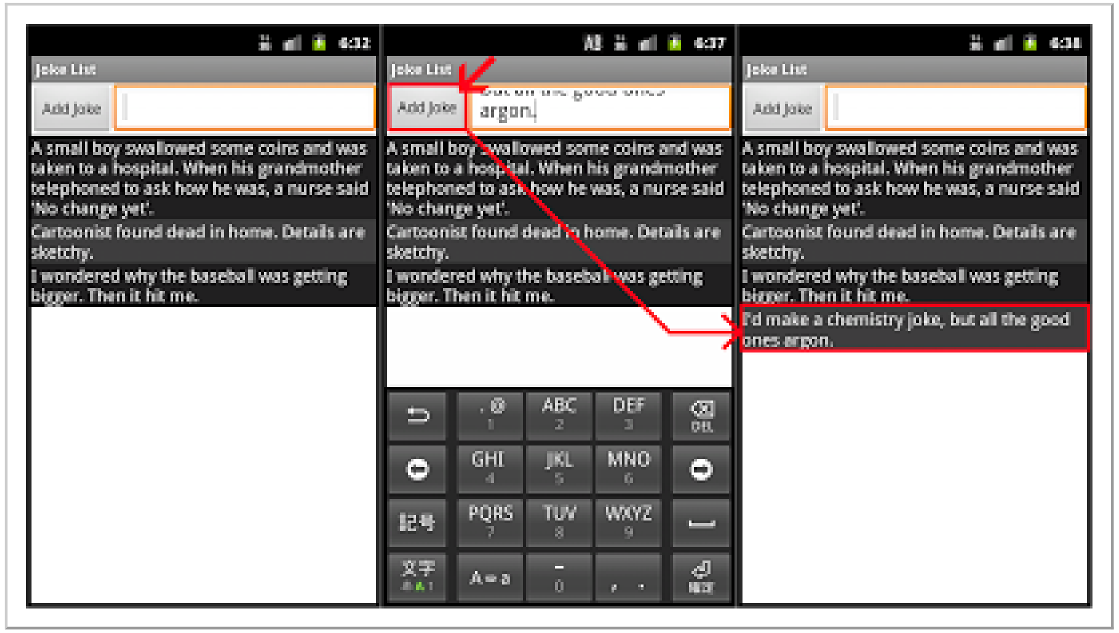
null and empty strings here).

        - Clear the text in the EditText.
        - Call your **addJoke** method
        - Add the following two lines of code to hide the Soft Keyboard that appears when the EditText has focus:

    -
      ```
      InputMethodManager imm = (InputMethodManager)
        getSystemService(Context.INPUT_METHOD_SERVICE);

      imm.hideSoftInputFromWindow(m_vwJokeEditText.getWindowToken(),
      0);
      ```
    - Call **initAddJokeListeners()** from **onCreate()**.

- Run your application to ensure that the "Add Joke" Button functions properly now (click image for full view):


Joke List

Add Joke

A small boy swallowed some coins and was taken to a hospital. When his grandmother telephoned to ask how he was, a nurse said 'No change yet'.

Cartoonist found dead in home. Details are sketchy.

I wondered why the baseball was getting bigger. Then it hit me.

Joke List

Add Joke | but all the good ones argon.

A small boy swallowed some coins and was taken to a hospital. When his grandmother telephoned to ask how he was, a nurse said 'No change yet'.

Cartoonist found dead in home. Details are sketchy.

I wondered why the baseball was getting bigger. Then it hit me.

Joke List

Add Joke

A small boy swallowed some coins and was taken to a hospital. When his grandmother telephoned to ask how he was, a nurse said 'No change yet'.

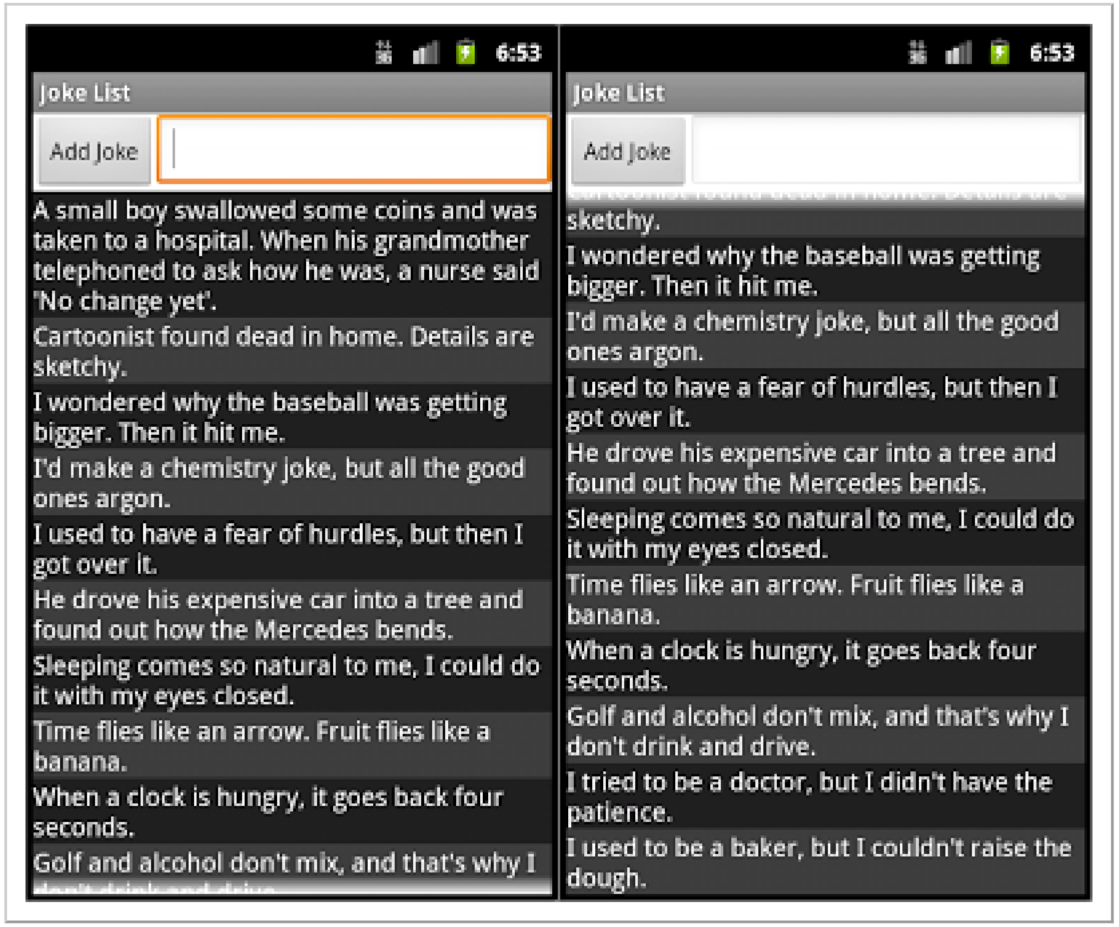
Cartoonist found dead in home. Details are sketchy.

I wondered why the baseball was getting bigger. Then it hit me.

I'd make a chemistry joke, but all the good ones argon.

- You may notice that if you change the orientation of your device, or the onscreen keyboard appears or disappears, all jokes you add will go away. This is expected for now, and will be resolved in a future lab.

- Remember to make sure the scrolling feature works if you add enough jokes, and doesn't cause the screen to automatically scroll down (click image for full view):



A small boy swallowed some coins and was taken to a hospital. When his grandmother telephoned to ask how he was, a nurse said 'No change yet'.
Cartoonist found dead in home. Details are sketchy.
I wondered why the baseball was getting bigger. Then it hit me.
I'd make a chemistry joke, but all the good ones argon.
I used to have a fear of hurdles, but then I got over it.
He drove his expensive car into a tree and found out how the Mercedes bends.
Sleeping comes so natural to me, I could do it with my eyes closed.
Time flies like an arrow. Fruit flies like a banana.
When a clock is hungry, it goes back four seconds.
Golf and alcohol don't mix, and that's why I

sketchy.
I wondered why the baseball was getting bigger. Then it hit me.
I'd make a chemistry joke, but all the good ones argon.
I used to have a fear of hurdles, but then I got over it.
He drove his expensive car into a tree and found out how the Mercedes bends.
Sleeping comes so natural to me, I could do it with my eyes closed.
Time flies like an arrow. Fruit flies like a banana.
When a clock is hungry, it goes back four seconds.
Golf and alcohol don't mix, and that's why I don't drink and drive.
I tried to be a doctor, but I didn't have the patience.
I used to be a baker, but I couldn't raise the dough.

- In this situation you had two options for where you could implement the code to handle the "Add Joke" button's onClick event. Each has its benefits and drawbacks:
  - Use an **Anonymous Inner Class**:
    - +: This option is a little cleaner and more readable since the event handling logic for different elements is self contained and referenced where it is used.
    - +: Since this event listener is only being used for

one UI control, you are free to make certain assumptions when implementing it. For instance, you know exactly which View generated the event (the Button).

        - -: This will consume more resources because this instantiates a new object, and class information for the object will need to be stored as well.

    - Make SimpleJokeList implement the **OnClickListener** interface:

        - +/-: In this option you would be able to register SimpleJokeList to respond to Click events from many different UI Controls. However, you would need to add conditional logic for determining which view fired the onClick event.

        - +: This has a clear performance benefit in that you don't have to load extra classes, which reduces your memory footprint and load time.

        - +: This is actually very common in Android applications due to the need to design for performance imposed by limited resources.

4.2.2 Adding Key Listeners

Your next task is to add event listeners that respond to certain keys being pressed on your own. When the user is entering their joke into the EditText and presses either the "Enter/Return" key or the "DPad-Center/Track-Ball" key, the application should respond exactly the same as it does when the "Add Joke" Button is pressed.

Add this code to the **initAddJokeListeners()** method. Remember that after the joke gets added, the soft-keyboard should disappear. You'll want to use an anonymous inner class, declaring a new OnKeyListener() object and implementing the onKey(View, int, KeyEvent) method.

Hints:

- Android OnKeyListener Interface Documentation
- Android KeyEvent Constants Documentation: ACTION_DOWN, ACTION_UP, & KEYCODEs

Note: Now you probably see why we chose to not make SimpleJokeList implement the OnClickListener interface in the previous step. It is messy if SimpleJokeList acts as the listener handler for too many objects at once, especially if those objects are different types. If we have a large number of Buttons (5 for example) and we want all button objects to be handled differently when clicked, we cannot easily do that without forcing our onClick() method implementation to check to see which button was clicked and adding separate logic for each button in the same method. Using anonymous inner classes is much cleaner in this situation, especially since we don't have too many objects to worry about.

4.2.3 Run SimpleJokeList Tests

- Run the **SimpleJokeListTest.java** Unit Tests to ensure that you have properly filled in this class.

    - Start the emulator or connect your device.

    - Turn off the keyguard by pressing the MENU button, sliding to unlock the home screen, etc. *This is necessary for some of the UI tests to work.*

- Run the tests using the Run button, or right-click the Android Test Project and select **Run as** -> **Android JUnit Test**.
    - *You will likely see a series of screens appearing on your device, running your application and performing actions such as adding jokes.*
- One complete, this should open up a JUnit Tab if it isn't already open. If you see the green bar, you've passed all the tests. Because you set the Test Project up earlier, it should run Unit Tests in both JokeTest.java and SimpleJokeListTest.java.
    - Note that the final Unit Test in SimpleJokeListTest.java, **testTextViewSize()**, may be failing if you changed the TextView's size in **addJoke()** to anything other than 16 (**setTextSize(TypedValue.COMPLEX_UNIT_PX, 16);**). In this case, feel free to change the assertEquals() statement inside the Unit Test to match the size you are using (look at the error message in the Failure Trace window below the JUnit window in Eclipse). The green bar is satisfying, and it would be a shame to deny you of 100% success because of any slight differences in appearance!

5. Deliverables

To complete this lab you will be required to:

1. Put your entire project directory into a .zip or .tar file, similar to the stub you were given. Submit the archive to PolyLearn. This effectively provides time-stamped evidence that you submitted the lab on time should there be any discrepancy later on in the quarter. The name of your archive should be **lab2<userid>** with a **.zip** or **.tar** extension. So if your username is jsmith and you

created a zip file, then your file would be named **lab2jsmith.zip**.

2. Load your app on a mobile device and bring it to class on the due date to demo for full credit.
3. Complete the survey for Lab2: https://www.surveymonkey.com/s/436F13Lab2

Primary Authors: James Reed and Kennedy Owen
Adviser: Dr. David Janzen

Comments

You do not have permission to add comments.