Android Application Programming Tutorial

Karl Cronburg, Lianne Lairmore, Kyle Peters, Raina Masand, & Tom Kim

December 3, 2012

1 Development Environment:

Development of Android applications is possible on any and all computing platforms (Linux, Windows, Mac, etc.). This tutorial was written using Ubuntu with the Eclipse IDE already installed. If you do not already have eclipse, please download and install it (Google **Eclipse download** and it should be the first result).

1.1 Android Software Development Kit (SDK):

Note: The Android SDK requires anywhere from 5 to 15 GB of disk space, depending on how many images you download.

To acquire the Android SDK, go to `http://developer.android.com/sdk/index.html#download`, select "I agree", and choose the appropriate version (64 or 32 bit) then click the download button. Upon completion of the download, cd into your Download directory, run **unzip** on the zip file downloaded, and cd into that directory (android-sdk-linux).

To update the SDK (download appropriate image files for various versions of Android), run **./tools/android&**. This will open up a GUI displaying a list of versions you can download. For the purposes of this tutorial, you need only make sure the **Tools** and **Android 4.2** items are selected. Once these are selected, click **Install XXX packages** (where XXX is some number of packages). Then click the **Accept All** radio button, and click the **Install** button.

At the bottom of the GUI you should now see a progress bar. Once the download(s) and installation(s) complete, you can close the GUI and move on to the next step.

Throughout this tutorial you will at times need to use a terminal and various executables located in sub-directories of the directory you just downloaded. Throughout this tutorial we will assume you have added something like the following to your **bashrc** file (or other appropriate rc file):

```
export ANDROID_SDK=/home/user/Downloads/android-sdk-linux
export PATH=$PATH:$ANDROID_SDK/tools:$ANDROID_SDK/platform-tools
```

1.2 Eclipse:

For this tutorial, we assume you already have eclipse installed, and are running it in a Linux environment. In order to integrate the Android SDK with eclipse, you must install a plugin. To do so, open eclipse, select the **Help** → **Install New Software...** menu. In the window which pops up, click the **Add** button and add a repository named **Android ADT** at location `https://dl-ssl.google.com/android/eclipse/`.

Once the location is added, select it in the **Work with** drop-down menu. Then click the **Select All** button, which should select **Developer Tools** and **NDK Plugins** as listed under **Name**. You should now have a window similar to the one shown in Figure 1. To continue, click **Next**, click **Accept All** and then click **Install**. Click **Ok** to any and all Security Warnings that pop up. Once the installation completes, select **Restart Now** to allow the plugin to take effect.

1.3 Command Line (ant):

Although Eclipse is a wonderful open-source tool, it has its quirks at times. An in-depth tutorial for managing projects from the command line is located at `http://developer.android.com/tools/projects/projects-cmdline.html`. Using the various programs in the SDK one can create, build, and run a project all from the command-line and using your favorite text / code editor (vim, emacs, gedit, ...).

Note: The SDK and Eclipse are more than capable of dealing with any mixture of command-line and Eclipse development. A developer can start out writing a project in Eclipse, and then decide to work from the command-line as needed or vice-versa.

2 Running "Hello World":

Now it's time to make your first app! In eclipse open select File → New → Android Application Project. This will bring up a window to create a new app. The Application Name will be the name of the application. The project name will be the

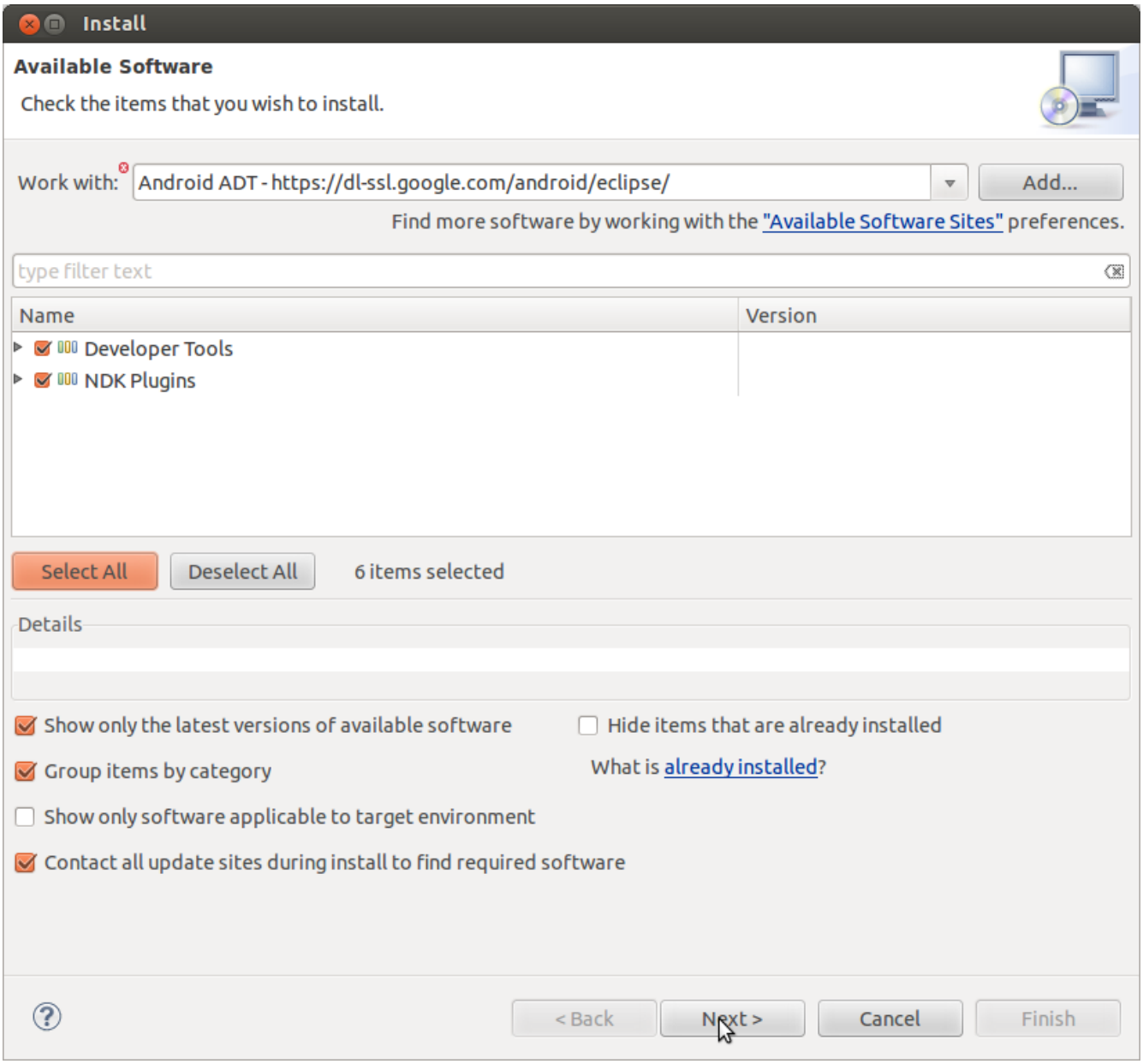

Figure 1: The pop up menu in Eclipse used to install the Android SDK plugin.

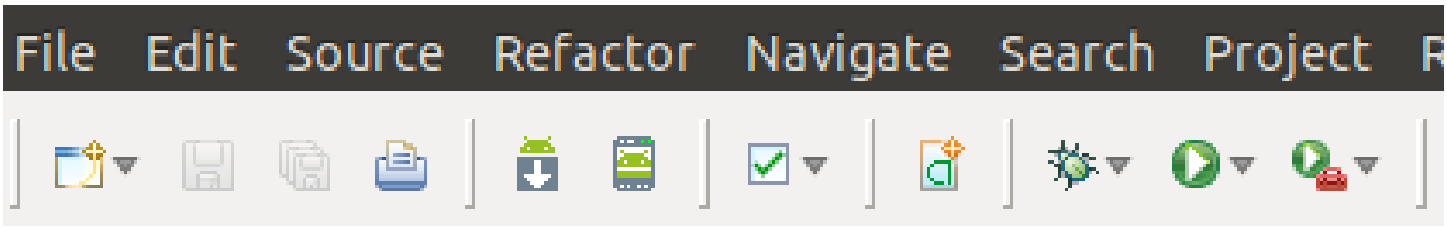

Figure 2: This is what your bar of menus / buttons should look like after installing the plugin. The icon with the green android in a box opens up the SDK Manager (same as running **android** command) and the icon with a green android inside a device / phone opens up the Android Virtual Device Manager (AVD).

```
1
2 # Asus USB Vendor:
3 SUBSYSTEM=="usb", ATTR{idVendor}=="0b05", MODE="0666", GROUP="karl"
4
5 # Motorola:
6 SUBSYSTEM=="usb", ATTR{idVendor}=="22b8", MODE="0666", GROUP="karl"
7
```

Figure 3: Example of the **51-android.rules** file located in **/etc/udev/rules.d**. This file tells the OS which users have which permissions for which devices. Multiple rules can be defined, one on each line, with lines beginning with a # (hash-sign) ignored. In this case, Asus devices (eg Nexus 7) and Motorola devices (eg Motorola Droid) can be read from and written to (**MODE="0666"**) by the user **karl**.

name of the folder created for the project and the Package Name is the package for the app. Eclipse will automatically fill the Project Name and Package Name as you type in the Application Name input.

For this app fill the Application Name as HelloWorld, the other fields should fill themselves in. Push the Next button to get to the next menu. This menu allows you to create an icon for your app. There are three options for creating icons, use a previous image, create an icon from clip art, or use text for the icon. For our project just use the default and select the Next option at the bottom of the page. The next menu allows you to create an Activity (this will be explained more later). The default setting should be to create a Blank Activity. Select the Next option to move on. The next menu will prompt you for the name of the new Activity to be created. An Activity will be the controller class if you were to think of your activity in MVC (Model View Controller) terms. It will control the view of the application. The text boxes should already be filled in with "MainActivity" which is what we will use. Select the Finish Options to create the new app.

2.1 Android Virtual Device (AVD) / Emulator:

You have now created an Android Application. It is now time to run it! Eclipse comes with its own built in emulator. Before you can use it you have to set up a phone to run your application. On the top tool bar there should be a button that looks like a green android in a device, which is to the right of the android with a down arrow and to the left of the check mark button as seen in Figure 2. Click this button to set up the emulator. Select the New... button on the right of the menu that pops up.

You are now setting up a device to run an application on. First give a name to the device. Choose a target API (might only be one option, if not choose any option for now). Next choose how much memory is on the SD card, lets say 100 MiB. Now you should be able to select Creat AVD at the bottom of the pop up. Now you should select the device you just made and select Start... from the right side of the pop up. Another pop up will show up; select Launch to continue. The emulator should now be starting. It takes the "device" some time to load but eventually it will show you the home screen of an android device. Your application hasn't been loaded on the emulator yet, we will do that now.

Go back to the eclipse window. Close out of the emulator windows if they are still open. Select your project folder on the left hand side of the window. Now select the green arrow without a bug or red box on it (as seen in Figure 2). It will ask what you want to run the application as. Select the Android Application option and push Ok. The app should now be running on the emulator (if any other devices are connected there might be an option menu to choose what to run it on, select the emulator). Bring up the emulator window and you will see MainActivity across the top and Hello world! centered on the screen. Congratulations! You are now an android application developer!

2.2 Real Android Device:

If you have your own android device you may want to run your application on that instead of an emulator. The primary benefit of doing so is that you get to see your application in action on a real device. As well, the application will generally run faster (limitations of any AVD) As running your application on a real device is OS dependent, the following instructions are for setting up your system to handle Android devices in recent versions of Linux (tested in Ubuntu >= 10.04 and RHEL 6).

First, your application must be declared as debuggable by adding **android:debuggable="true"** to the <**application**> element of your AndroidManifest.xml file. Remember to disable this tag before building your application for release!

Next, enable **USB Debugging** on your device by going to either **Settings** → **Developer options** or **Settings** → **Applications** → **Development** depending on your Android OS.

Note: If you are working on a Bucknell machine in either 164 or 167, you can skip the rest of this section.

Now you need to enable detection of your device by Linux. To do so, create a file called **/etc/udev/rules.d/51-android.rules** as root (**touch 51-android.rules && chmod 644 51-android.rules**), and add the following line:

```
SUBSYSTEM=="usb", ATTR{idVendor}=="22b8", MODE="0666", GROUP="user"
```

where you replace **user** with your username on the system (this will give read / write permissions of the USB device to your user) and change **22b8** with the USB vendor ID for your particular device. Go to `http://developer.android.com/tools/device.html` for a list of IDs. See Figure 3 for a working example.

You will now want to test your ability to connect to the device. Close all virtual devices (AVDs) and connect only the device you wish to run your app on. Now, execute the command **adb devices**. If you get back something like the following:

```
List of devices attached
1C9BAC121B743112    device
```

then you have properly set up your device for installation of applications. You will now want to cd into the home directory of your project, and execute the following:

```
$ android update project -p .
$ adb install my.app
```

where you replace **my.app** with the name given to your application during initial setup. If all goes well, your application should now be uploaded and installed onto your phone, and automatically opened to the main view. If anything goes wrong, or you ever need to install the application from a different computer, run **adb uninstall my.app** while the device is connected.

2.3 LogCat Debugging:

While developing an application for android you may run into problems and have to debug your code. The System.out.print() and System.err.print() methods do not show up in the console window so another tool has to be used. Eclipse has LogCat which can be used to debug your application. If LogCat isn't already running as a tab on the bottom window in eclipse then open it now by selecting Window → Show View → LogCat. If LogCat isn't one of the options in the list you may need to then select Other and type in "logcat" into the text box on the popup window.

Now that LogCat is open in your screen you can look through the messages (there should be messages in the window if you have successfully started your HelloWorld application on the emulator). The messages are color coded with each level as its own color. You probably just need to know that errors are red and developer messages are blue for now. Messages will continue to be displayed as long as the emulator is on or the device is plugged in. To stop LogCat from automatically scrolling you can use the right down arrow button. You can also filter the type of messages you want to view using the drop down menu that says verbose by default. To create personal filters for the debugger select the + button, on the left side of the message window. The window that pops up will allow you to filter by any of the options given. Tags will be explained in a minute but are commonly the class name. For more information about using LogCat you can go to `http://www.droidnova.com/debugging-in-android-using-eclipse,541.html`

LogCat let's you see logged messages, but how do you make log messages? In the Android API there is a class named Log which allow developers to log. There are five static methods that can be used to log events in an app. This tutorial will show you how to use Log.d(). Log.d() is the method for printing debug messages which can be viewed from LogCat. You can view the documentation for Log at `http://developer.android.com/reference/android/util/Log.html#d%28java.lang.String,%20java.lang.String%29`. The simplest way to use the logging features is to just add

```
Log.d("MethodName","This is a debugging message");
```

By using the method name as the tag string it is easier to set filters for the methods you are debugging. The second string doesn't have to be a string constant, it can contain a variable that you need to check. The the benefit of using Log.d() messages is that when building the project for deployment they will be left out. This is a nice feature since it means you won't have to remove your debugging statements when you submit the app.

3 First Activity:

3.1 XML Layout Files:

Although its nice that our app works right out of the box let's make it do something. The first thing we should do is change the layout. A layout is an XML file which defines what the application looks like. Referencing the MVC model, the layout is the View. Actually a layout extends the View class so it really is a View. On the left side of eclipse is the applications files. Open the res folder followed by the layout folder and double click the main_activity.xml file (the file might already be open from when it was created).

There are two options when working with layouts and views in eclipse. There is a GUI interface to edit layouts and there is the actual XML file. The GUI has its limitations and a combination of both is usually the best option. In this example we will just use XML. On the bottom of the window their should be a tab to view the XML document. There should already be code that looks like this

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="@string/hello_world"
        tools:context=".MainActivity" />

</RelativeLayout>
```

You can see there is a text view centered in the layout with a string hello_world just like what was seen on the emulator. We are going to modify this slightly for our example. First we are going to add a id tag to the TextView by adding *android:id="@+id/text"*. TextView should now look like this

```
 <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="@string/hello_world"
        android:id="@+id/text"
        tools:context=".MainActivity" />
```

We will use the id to reference the TextView in the Activity and in the layout. Next let's add a button below the text view. Below the TextView element add a Button element (still inside the RelativeLayout element).

```
<Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_below="@id/text"
        android:text="@string/change"
        android:onClick="changeString"
        />
```

The tag layout_below uses the id from another element to place the button element. If you save now it will give you an error because we haven't added change to the string file yet. Let's do that now.

Inside the res directory open values directory and inside that open the string.xml file. Again, eclipse gives you the option to use their interface or work with the XML yourself. Switch to the xml viewing mode and add the line

```
   <string name="change">Change</string>
```

inside the resources element, below the other strings. Save this file and return to the main_activity.xml file. There should not be any more errors in the XML file. If you were to run the app now you would see a text saying "Hello world!" with a button that says "Change" under it. We haven't defined "changeString" method yet so pushing the button crashes the app. Let's go fix that.

3.2 Java Activity:

In your project directory their should be a src folder. Navigate into it and open the package folder and then open the MainActivity class. The MainActivity class should have two methods already in it, onCreate() and onCreateOptionsMenu(). The onCreate method is called when the Activity starts and the onCreateOptionsMenu method is called when the device specific menu button is selected.

Right now we don't need to add to either of these methods. We are going to make a new method called changeString(). In the MainActivity class add the this method:

```
public void changeString(View view){
 TextView text = (TextView) findViewById(R.id.text);
 text.setText("Goodbye");
}
```

If TextView is underlined just highlight it and press ctr+shift+o. This will have eclipse add the appropriate imports. When interfacing methods with views in the layout the type signature is to return void and take a View as the parameter. In the above code we use the findViewById() method to retrieve the view from the layout. If you remember before we set the id of the TextView element to text. The R class is automatically generated by eclipse and can be used to reference objects in the layout files. Try running the app now. When you push the button what happens?

Now we could extend this method to change back to "Hello world!". First lets make a class variable called hello which is a boolean and set it to true.

```
public class MainActivity extends Activity {
boolean hello = true;
...
```

Next lets add an if statement before changing the text to "Goodbye" and add an else statement which returns the text to what it was previously.

```
public void changeString(View view){
 TextView text = (TextView) findViewById(R.id.text);
 if(hello){
        text.setText("Goodbye");
        hello = false;
 }else{
        text.setText(R.string.hello_world);
        hello = true;
 }
}
```

As you can see, we can reference strings from the string.xml file not just in the layouts but also from the java classes.

There are more methods that can be overwritten when creating an activity. The Android API has a method for all points in an activities life time. For example onPause() is called when the system is paused and might be a good time for an app to save any current data. A full list of an activities life time can be found at `http://developer.android.com/reference/android/app/Activity.html`

3.3 The Android Manifest:

All applications have a file called AndroidManifest.xml which contain the running information for the application. Go to the project folder and open it now. Go to the xml viewing tab on the bottom like the other xml files. One thing in the AndroidManifest is the minimum and target SDKs for the application. It also contains the permissions needed for the application to run. These permissions will be prompted to the user when they download the application from Google Play. There will be more information about permissions in the next section. In the application element the icon and the label of the application is defined along with all the activities in the application. MainActivity should be listed in this section. Under MainActivity you can see something called a LAUNCHER. This designates the activity as the one which will launch the application.

3.4 String Constants (strings.xml):

We have already looked used the string.xml file a little but lets look what we can really do with it. Having a string.xml file is a really nice feature in Android app development. Not only does it mean only editing strings in one location in the code but it also allows for multiple language development.

Open the string.xml file back up. Instead of adding strings, let's add a string array. Add this to your string.xml file

```
<string-array name="list">
       <item>First</item>
       <item>Second</item>
       <item>Last</item>
   </string-array>
```

Now go back to the MainActivity class. Let's use our new string array. Change the changeString method to have this

```
    public void changeString(View view){
     Resources res = getResources();
     String[] list = res.getStringArray(R.array.list);
     TextView text = (TextView) findViewById(R.id.text);
     text.setText(list[index]);
     index = (index +1) % 3;
    }
```

Now run the program. The string should now rotate through the strings in the string array.

4 Dialogs, Toasts, and More

4.1 Dialogs

In most apps there is a need to use a dialog box. Dialog boxes are new views that do not take up a whole screen and don't start a new activity. We will now add a settings dialog box to our app. Eclipse automatically creates a Settings option in the activity menu. You can view this by pressing the "Menu" button on the right side of the emulator. It is to the right of the home button. When you select the settings button nothing happens. We are going to change that.

In the res folder their is a menu sub-folder. Inside that folder is an activity_main.xml file. This is the menu that you see when you press the menu button. As you can see there is a setting item in the view. Just add the line

```
android:onClick="showMenu"
```

as a tag in the item element. Like before we will now have to define showMenu like we did for changeString. Go back to the MainActivity class and create a showMenu method which returns void and takes in a MenuItem (unlike before in which it took a View as the parameter). The class we are going to use to make a view is AlertDialog.Builder. First we will create a builder and add the necessary components. Then we will create the dialog and lastly show it.
Let's create a dialog that sets the background color for out app. First create a builder

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
```

Next lets create a title for our dialog

```
builder.setTitle("Background Color");
```

Next we need to create the options. First we need a string array in the strings.xml file. Open strings.xml and add

```
   <string-array name="colors">
       <item>White</item>
       <item>Red</item>
       <item>Yellow</item>
   </string-array>
```

There is one more thing we need to do which is to give the RelativeLayout in the layout folder an id. Add the tag

```
android:id="@+id/background"
```

in the RelativeLayout element. Now back in the showMenu method in the MainActivity class we should add items to the dialog.

```
builder.setItems(R.array.colors, new DialogInterface.OnClickListener() {
       public void onClick(DialogInterface dialog, int which) {
        View v = findViewById(R.id.background);
        switch (which){
        case 0:v.setBackgroundColor(0xFFFFFFFF);break;
        case 1:v.setBackgroundColor(0xFFFF0000);break;
        case 2:v.setBackgroundColor(0xFFFFFF00);break;
        }
       }
});
```

The setItems takes two parameters, a resource int representing a string array and a DialogInterface.OnClickListener. In this line we are creating a OnClickListener for the items which will call onClick when any item is selected. The int which is the index of the item chosen. All layouts are extensions of the View class so we will call RelativeLayout a view. We retrieve the background layout with the findViewById method and store it in v. When the first item is selected (white) set the background color to 0xFFFFFFFF. The colors are hex values with the form 0xAARRGGBB with a being Alpha, R being red, G being green, and B being blue. Case 1 results in red and case 2 results in yellow. Do not run this code yet. We are forgetting a very important step. We need to build and show the dialog.

Building and showing the dialog is easy and can be accomplished just with the line

```
builder.create().show();
```

Now run the app. When it loads select the Menu button on the right. The Settings option should pop up on a menu. Select Settings and you will see your new dialog.

4.2 Toats

If you application doesn't need input from the user but wants to notify them of something you may want to use a toast. The best way to explain a toast is to just show you! After dialogs, toasts are easy. We will add one to when we push the Change button. Go into MainActivity and scroll to your changeString method.

Maybe we should warn the user what the next string is going to be that way they will know if they want to change it. All we would need to do is add the line

```
Toast.makeText(this, "The next string is going to be "+list[index], Toast.LENGTH_SHORT).show();
```

after updating index. It's as easy as that! The first argument is just the context you want the toast to appear on. The second argument is the string which you would like to share. The last argument is the length of the toast; there is Toast.LENGTH_SHORT and Toast.LENGTH_LONG. Toasts can also be used in basic debugging too. If you use toasts to debug you will have to remove them when you are done unlike the debug log messages.

4.3 Intents

The last thing we are going to introduce in this tutorial is the idea of an Intent. In our app so far there is only one activity. In most app there are at least two if not many more. Let's add another activity to our app.

To add an activity you can right click on the HelloWorld folder then select New → Other. Alternatively you can select the button on the furthest left of the menu bar which looks like a window. Both of these options will bring you to a dialog to select a new file type.Select Android and then Android Activity and click the Next button. This menu should look familiar. As before just press Next again. Give this activity the name of SecondActivity. Next to the Hierarchical Parent line push the "..." button. Start to type in MainActivity until you see the class you already created. Select it and press okay. When you are back to the previous menu select Finish. Eclipse will automatically open you up to the layout page.

Now we have to get to the new activity. It would be really silly though if we change the color on the first activity but not the second. Let's set it up so we can later change its background color. In the new layout file go to the xml view. Let's add

```
android:id="@+id/secondActivity"
```

to the RelativeLayout element. Next we have to modify activity_main.xml so we can get to our new activity. Lets add the button

```
<Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_alignParentBottom="true"
        android:text="@string/next"
        android:onClick="next"/>
```

to the xml file. We will also have to add

```
<string name="next">Next Activity</string>
```

to the strings.xml file. Now that we have a button we can add the method it calls when touched. Create a next method which returns void and has one parameter that is a View. Now inside that method we are going to make an Intent and use it to start another activity.

```
public void next(View view){
     Intent intent = new Intent(this, SecondActivity.class);
     startActivity(intent);
    }
```

You can now run the app. There should be a button that says "Next Activity". When you push it you will go to another activity. You can use the back button to return to your first activity. Now it looks pretty silly if your first activity is one color and your second one is another. We should fix that now.
What we need to do is send a message to the second activity. The first step is to name this message. Add the following line at the top of the MainActivity class as a class variable

```
public final static String EXTRA_MESSAGE = "com.example.helloworld.COLOR";
```

Next add the class variable

```
String color = "white";
```

as another class variable. Next you have to update color every time showMenu is called, so in case 0 color = "white" and so on. The last thing that we need to add to MainActivity is the line

```
     intent.putExtra(EXTRA_MESSAGE, color);
```

between creating the intent and starting the next activity. Next we need to receive the message in SecondActivity.
Open SecondActivity and add the following code at the end of the onCreate method

```
Intent intent = getIntent();
        String color = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
        View v = findViewById(R.id.secondActivity);
        if(color.equals("white")){
         v.setBackgroundColor(0xFFFFFFFF);
        }else if(color.equals("red")){
         v.setBackgroundColor(0xFFFF0000);
        }else if(color.equals("yellow")){
         v.setBackgroundColor(0xFFFFFF00);
        }
```

You may need to comment out the line

```
        getActionBar().setDisplayHomeAsUpEnabled(true);
```

to get the app to run. Now test the app. Change the color in the first activity and change to the second activity.

That is it for our application demo. Next will be a short introduction to adding your app to the store.

5 App Store Submission:

5.1 Developer Account:

There will come a time when you feel that the application you have created is ready to be open to the world. Rather than telling your friends to download the SDK and install your application by command line (**adb install MyProject**) or stealing

their phone and secretly installing the application yourself, you can publish your application to the Google App / Play Store. To do so, you must visit `https://play.google.com/apps/publish/signup` to sign up for a Google Developer account ($25 registration fee).

Once you have an account, you will then want to go through Google's submission checklist as well as reading up on their application guidelines (`http://developer.android.com/distribute/googleplay/publish/preparing.html`). Read on to learn how to prepare your application locally for distribution.

5.2 Building For Release:

Note: If you have added **android:debuggable="true"** to your AndroidManifest.xml in order to run your application on a real device, you should delete it at this time before building your application.

The packages which Eclipse and ant build for you during development of an app are different from a package you build when creating a distributable version of your app. During development, a "debugging" version of the app is created. To build a release version of your app, you run the command **ant release** (instead of **ant debug**). This creates an APK for your application, located in the **bin** directory, called **MyProject-unsigned.apk**.

5.3 Application Signing:

Now that you have **MyProject-unisgned.apk** located in your bin folder, you now want to **sign** your application. Signing an application verifies that you are the proper owner of your application. If someone hacks your Google account and tries to upload a malicious version of your app, Google will see that the application being update is not signed with the correct public key and will take action.

To generate a key for signing your application for the first time, run:

```
$ keytool -genkey -v -keystore my.app.keystore -alias my.app -keyalg RSA -keysize 2048 -validity 10000
```

The above command verbosely generates a private / public RSA key pair of key size 2048 bits aliased by **my.app** with a validity of 10,000 days (27 years). Feel free to change around the arguments given, however a good validity should be longer than the life expectancy of the app.

Now that you have a private key, you are ready to sign your application. To do so run the following script:

```
#!/bin/bash

# Generate APK:
android update project -p .
ant release

# Sign the APK:
jarsigner -verbose -storepass MyPassword -sigalg MD5withRSA -digestalg SHA1 \
-keystore my.app.keystore bin/MyApp-release-unsigned.apk my.app
mv bin/MyApp-release-unsigned.apk bin/MyApp-release-signed.apk

# Align the APK:
rm -f bin/MyApp.apk
zipalign -v 4 bin/MyApp-release-signed.apk bin/MyApp.apk

# Verify your signature after alignment:
jarsigner -verify bin/MyApp.apk
```

If the last line of output says **jar verified.** then you have successfully built and signed your first application! For further description of application signing, visit `http://developer.android.com/tools/publishing/app-signing.html`.