


Android Development Lab

Today, you (and possibly a partner) will explore using Eclipse to develop Android applications. We will look at a typical development workflow - creating a new project, running an application on the emulator, editing code in both Java and XML, and debugging and testing your application. At the end of each part, you should show your work to me so you can be checked off. If you get stuck or are confused, be sure to ask for help.

Part 1: Creating and running an Activity

Open Eclipse - on the lab machines, there is a shortcut under Start → Class Files. Construct a new Android project by going to File → New → Other, and selecting Android → Android Project. You can also press the new Android Project button -  - on the toolbar. Name your project “AndroidLab.” Use the following values for the other fields:


- Build Target: Android 1.5. This is the Android version that you want your application built against. You should always use the target with the lowest platform version possible, unless you need APIs that were introduced in a newer version of the SDK. In the official Android reference documentation, every function is annotated with “Since: API Level X,” so you know the minimum version that you need:

```
public void addHeaderView (View v) Since: API Level 1
```

Add a fixed view to appear at the top of the list. If addHeaderView is called more than once the views will appear in the order they were

- Application name: AndroidLab. This is the human-readable name for your application, which will appear on the device (or emulator).
- Package name: cornell.cs2046.androidlab. This is just like any other Java package name, and should be unique for a given application.
- Create Activity: Make sure this box is checked, and name the Activity AndroidLabActivity.
- Min SDK Version: 3. This represents the oldest version of the SDK that your application will run on. In general, this should just be equal to the API level for the build target that you selected.

Your project will now appear in the Package Explorer pane in Eclipse, and will be compiled in a few seconds. If you see a red ! or x on the project icon, go to Project → Clean Project, and select the AndroidLab project to clean. Your project should now compile properly - there is an issue with the Lab machines that seems to cause this to happen.

Now, run the application in an emulator. With the project selected under Package Explorer, go to Run → Run or press the  icon, select “Android Application” from the menu that appears, and hit OK. In a few moments, a preconfigured Android 1.5 emulator should start to boot up. The emulator will take about a minute to load - while you wait, you can flag me down so I can check you off. Once the emulator boots, the screen may be locked, in which case, you should press the Menu key. After you unlock the screen and the Activity is launched, you should see “Hello World, AndroidLabActivity” on the screen.

Once you’ve been checked off, move on to the next part. Make sure you leave the emulator open - from now on, when you run your Activity, Eclipse will detect the running instance of the emulator and run the Activity on that instead of launching a new emulator, which should save you a fair amount of time.

Part 2: Work with Layout and Code Editors

Let’s add a button to the screen which pops up a dialog box. First, we should create a string with the text that will appear on the button. Open up `res/values/strings.xml`. This will launch a graphical editor for the strings file. You can see the two strings that Eclipse created by default: `hello`, which contains the message that appeared when you ran the app, and `app_name`, which is the name of your application. On the bottom of the editor window, you should see two tabs - Resources, which you are currently on, and `strings.xml`, which lets you edit the raw XML of the strings file.

Using either the graphical editor or the text editor, whichever you are more comfortable with, create a new string named “`buttontext`” with value “Launch Dialog”, and save the file.

Now, let’s add a Button to the default layout. Open `res/layout/main.xml` which will open a similar two-tab editor. The graphical editor for layouts is probably more difficult to work with than the raw XML, but you are welcome to explore using it. Add a button with the following attributes:

- `android:id="@+id/dialogButton"`. Recall that this is the ID we will use to refer to this button in the code.
- `android:text="@string/buttontext"`. This sets the label of the button.
- `android:layout_width = android:layout_height = "wrap_content"`. This sets the button’s dimensions to the natural size needed to contain its label.
- `android:layout_marginTop="20dip"`. This puts a small margin between the text and the button.

We now add code to have the button launch a dialog when the button is pressed. Open `AndroidLabActivity.java` by expanding the `src/` folder and opening the `cornell.cs2046.androidlab` package. You will see the default `onCreate` method that Eclipse creates for you.

Type the following line of code below the call to `setContentView`:

```
Button button = (Button) findViewById(R.id.dialogButton);
```

You should notice two things as you type:

- Eclipse should automatically complete `R.id.dialogButton` as you are typing it. This content assist is incredibly useful when writing code, and there are many different instances in which Eclipse will help you.
- Once the line is typed, the two instances of `Button` will be underlined Red, signifying a compiler error. If you look at the Problems tab at the bottom of Eclipse, you should see that “Button cannot be resolved to a type”. This is because the top of the source is missing the line:

```
import android.widget.Button;
```

You can type this line manually; however, Eclipse has a keyboard shortcut, Ctrl-Shift-O, which will automatically fill in any missing imports and delete unused imports. When writing code, you will frequently need to import more Android classes; this shortcut will come in handy in managing the imports.

Now, we will set the button’s handler for click events. Type the following:

```
button.setOnClickListener(new OnClickListener() {  
});
```

You should find that Eclipse automatically completes the function name, and that `OnClickListener` is once again underlined red. This time, when you press Ctrl-Shift-O, Eclipse will offer you a choice of which `OnClickListener` to import. In this case, we want `android.view.View.OnClickListener`. However, once you import the class, the function will still be underlined red. What is wrong now?

The issue is that your `OnClickListener` does not have an implementation for the `onClick` method, which is required to implement the `OnClickListener` interface. Hold your mouse over `OnClickListener()` and a small box should pop up offering a fix, which is to add the unimplemented method. Clicking this should automatically fill in a skeleton `onClick` method for you.

Now, replace the automatically generated method stub with the following:

```
AlertDialog.Builder builder = new AlertDialog.Builder(AndroidLabActivity.this);  
builder.setTitle("Hello")  
    .setMessage("This is a dialog!")  
    .setNegativeButton("Close", null)  
    .create().show();
```

This code will use the `AlertDialog.Builder` class to construct an alert dialog and display it on the screen. Builders let you chain together functions like `setTitle` and `setMessage` in one long sequence, as seen in this code snippet. The `setNegativeButton` command tells the dialog to show a cancel button with the text “Close.” The second argument is an `onClickListener` for this cancel button; since we don’t need to do anything once the button is pressed, we leave this argument as `null`. Finally, we call `create()`, which builds the `AlertDialog` object, and then use the `show()` method to display the dialog.

When you launch the application, you should see the “Launch Dialog” button, and pressing it should pop up this dialog box:



Once you have reached this point, get checked off by me and move onto the final part.

Part 3: Debugging and Testing Applications


Now, let’s break the application and explore a few ways to debug on Android. In your `onClick` listener, replace `new AlertDialog.Builder(AndroidLabActivity.this)` with `null`, and run the application again. This time, when you press the button, you should see a pretty unhelpful dialog stating that your app crashed. How can we figure out the cause of the problem?

The first resource is to check the device log to see the exception that was thrown. To open this log, go to Window → Show View → Other, and select Android → LogCat. If you look at the device log and scroll up, you should see that your application crashed because of a `NullPointerException`, and you should see the stack trace, which will give you the function which caused the crash and the line number where the crash occurred. This will help you determine where an error has occurred so you can start debugging the issue.



Once you’ve found where an error occurs, a typical programming trick is to print out certain variables to make sure they have values that you’d expect. You can also print messages to the device log, and they will appear in the LogCat window. Right after you initialize builder to `null`, add the following line:

```
Log.e("AndroidLab", "value of builder = " + builder);
```

You should import `android.util.Log`, if you are offered a choice.

Next, create a filter in LogCat so messages from your application are easy to see. Click the  icon in LogCat to create a new filter, with name “Android Lab” and Log Tag “AndroidLab”. Now, when you run your application and press the button, you should see that builder has value null before it crashes.

Sometimes simple print statements are too cumbersome to use for debugging. Eclipse has great built-in debug functionality that works with Android. To test this, first set a breakpoint in the program by selecting the line containing the call to `Log.e` and pressing `Ctrl-Shift-B`. You should now see a little blue dot on the left side of the editor on this line.

Now, launch the app in debug mode by pressing the bug icon -  - next to the run icon. When you press the “Launch Dialog” button, Eclipse will ask you to switch to the debug perspective. Do this, and you will get to inspect all of the variables and their values at the breakpoint. You can see that builder is null in the Variables view. Next, press the Resume button -  - to continue running the application, which will cause the application to crash. Eclipse’s debug functionality is very extensive, and mostly out of the scope of this class, but it can be helpful for you if you cannot figure out what is causing a crash.

Switch back to the Java perspective by going to `Window → Open Perspective → Java`.

Finally, let’s write a test for your application. This test will emulate pressing the “Launch Dialog” button, and will fail if this causes an exception. First, create a new Android Test Project in the same way you created the initial Android project, but using Android Test Project instead of Android Project. Under Test Target, select the existing AndroidLab project. The rest of the form fields should automatically be filled out for you.

Next, create a new Test class by opening the `src/` folder under the AndroidLabTest project, right clicking on the package `cornell.cs2046.androidlab.test`, and adding a new Java class. Call this class `AndroidLabActivityTest`. Type “`ActivityInstrumentationTestCase2`” under Superclass, and press `Ctrl-Space`, which will automatically fill in the full class name for this superclass. Finish the wizard to create the class.

This class is actually generic, and so you should replace “`extends ActivityInstrumentationTestCase2`” with “`extends ActivityInstrumentationTestCase2<AndroidLabActivity>`”. In addition, insert the following constructor, which prepares the test framework for running tests on `AndroidLabActivity`:

```
public AndroidLabActivityTest() {  
    super("cornell.cs2046.androidlab", AndroidLabActivity.class);  
}
```

Now, let’s write a JUnit test. This should be familiar if you’ve used JUnit before, but if not, this part of the lab will teach you all of the necessary steps. Create the following test

method - JUnit automatically combs any method whose name starts with test and runs these methods as tests:

```
@UiThreadTest
public void testDialog() {
    AndroidLabActivity activity = getActivity();
    Button dialogButton = (Button) activity.
        findViewById(cornell.cs2046.androidlab.R.id.dialogButton);
    try {
        dialogButton.performClick();
    } catch (Exception e) {
        fail("Failed to create dialog!");
    }
}
```

The `@UiThreadTest` annotation tells Android that this test should run on your main UI thread, which lets you access the views and widgets in your application. Note that the full name for the R class is needed since the test package has its own R class which is distinct from the R class for the AndroidLab application. Now, hit run, and select “Android JUnit Test” from the dialog that appears. This should run the test and open a JUnit pane which notes that your test failed, with the expected error, “Failed to create dialog!”.

Finally, go back into the `AndroidLabActivity` source, undo your changes to make the application work again, save, and rerun the test. This time, the test should pass. Show me the passed test, and once you are checked off, you have completed the lab session.