Google Android Development

Lesson#4

Debugging

There are two types of "bugs" that can be created during a development cycle – **Run-Time errors** or **Logic errors**.

Run-Time errors – These errors typically occur when something unexpected happens within your code. A good example of that is allowing your users to click a submit button before supplied any input. If that input is used by the code, errors may be produced.

Logic errors – These are nasty! Your program appears to be working but is not displaying the correct results. For example, if 2 + 2 = 5 instead of 4, that would be a logic error.

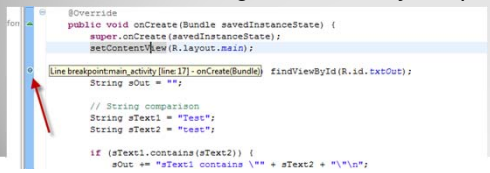
In both cases, you need to find and correct the problem – This is called **debugging**.

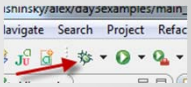
Debugging Your Work

Debugging is typically done by stepping through your code, while keeping an eye out on variable values, and the flow of the application.

Doing so, you can locate bugs, and problems, and get a better understanding of code branching.

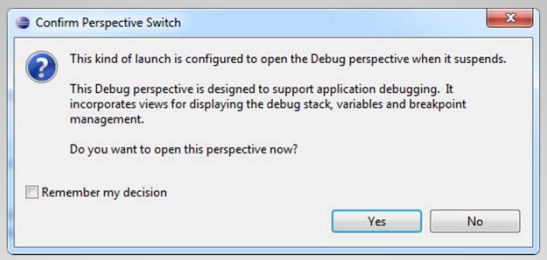
Eclipse provides sufficient features to help you debug your work. To begin debugging, open a project in Eclipse, and create one or more breakpoints within your code. To do so, click on the outer margin in your code window, and look for a small blue dot to appear. This dot is the point at which your code will stop during the debug session. Now, use the debug icon to run your project.



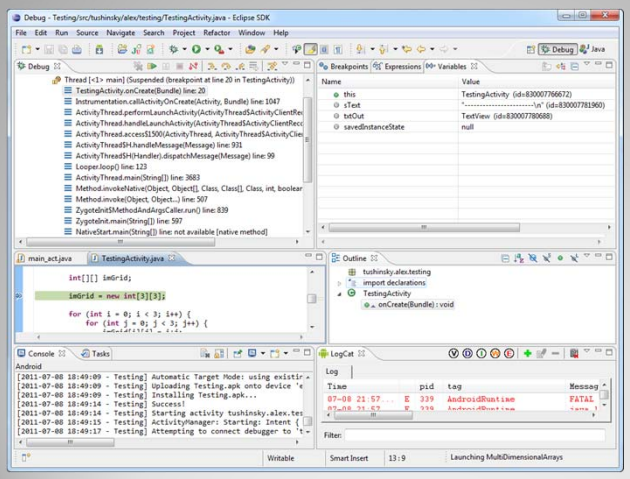


Debugging

Once debugging starts, Eclipse will attempt to open a perspective layout. Allow it to do so.



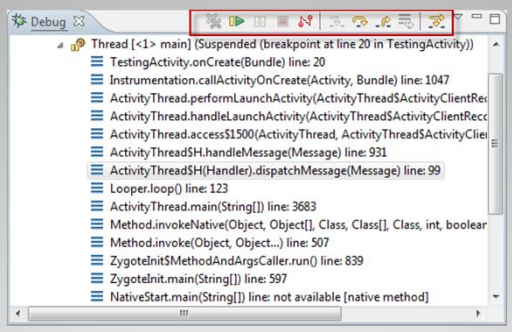
Debugging



The new perspective is opened, and you are now able to step through your code.

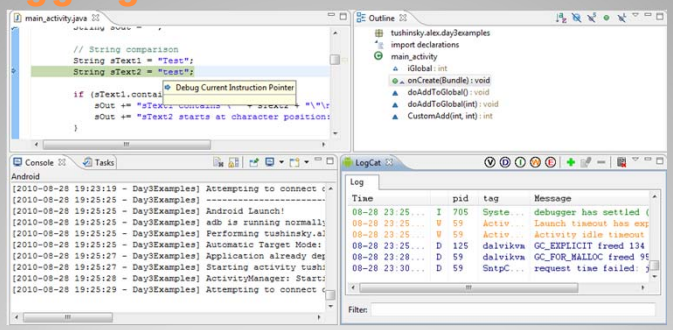
Use **F5** to step into each line, **F6** to step over lengthy methods that you know work correctly.

Debugging



The main debug window provides information about which class and method are currently running. The debug tools (step into, step over, resume, disconnect, etc.) are at the upper right-hand side of this window.

Debugging



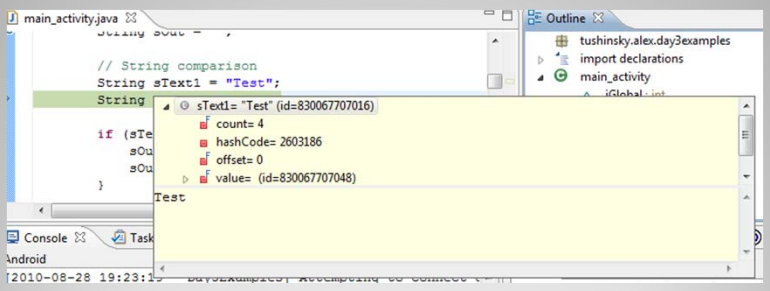
Below the debug window, you'll find your code window, with the statement currently being executed highlighted in green. Across from that is the Outline window which points out which method is currently running, and below are your Console, and LogCat. LogCat is the Android logging system which provides a mechanism for collecting and viewing system debug output. Logs from various applications and portions of the system can be seen here.

Debugging

If you need to switch back to your regular perspective, use the icons on the right-hand side of the IDE to switch between Java and Debug perspectives.

Debugging

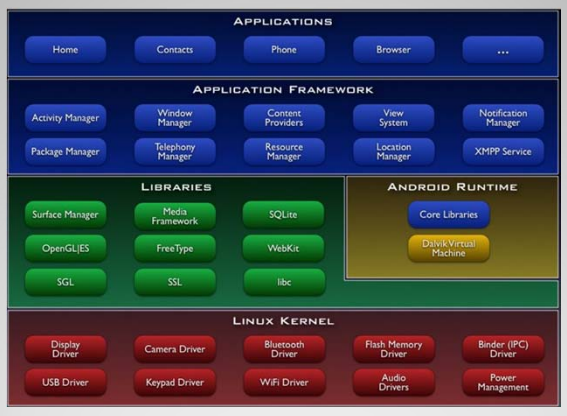
Using F5, you can step through your code. To get a better idea of what values your variables hold, hover over them to expose the tooltip with that information or use the variable window in Eclipse.



Pressing F8 will allow you to resume the execution of your application.

Android

Now that we're familiar with Java and Eclipse, it is time to get a better understanding of Android, and what makes it tick.



Linux Kernel

Android is built on top of very stable and proven technology – Linux. Linux is responsible for managing things such as memory, networking, and various operating system related services and processes.

As a developer, you will never really work with the OS directly.

Native Libraries

Android includes a set of C/C++ libraries used by various components of the Android system. These capabilities are exposed to developers through the Android application framework. Some of the core libraries are listed below:

- **System C library** - a BSD-derived implementation of the standard C system library (libc), tuned for embedded Linux-based devices
- **Media Libraries** - based on PacketVideo's OpenCORE; the libraries support playback and recording of many popular audio and video formats, as well as static image files, including MPEG4, H.264, MP3, AAC, AMR, JPG, and PNG
- **Surface Manager** - manages access to the display subsystem and seamlessly composites 2D and 3D graphic layers from multiple applications
- **LibWebCore** - a modern web browser engine which powers both the Android browser and an embeddable web view
- **SGL** - the underlying 2D graphics engine
- **3D libraries** - an implementation based on OpenGL ES 1.0 APIs; the libraries use either hardware 3D acceleration (where available) or the included, highly optimized 3D software rasterizer
- **FreeType** - bitmap and vector font rendering
- **SQLite** - a powerful and lightweight relational database engine available to all applications

http://developer.android.com/guide/basics/what-is-android.html

Android Runtime

Each application that runs on Android, runs in its own process, using the Dalvik Virtual Machine.

The Dalvik VM, written by Dan Bornstein at Google, is basically Google's implementation of the Java runtime, optimized for smaller, low memory devices, such as phones, and tablets.

Application Framework

The Application Framework provides much functionality necessary for us to build our applications:

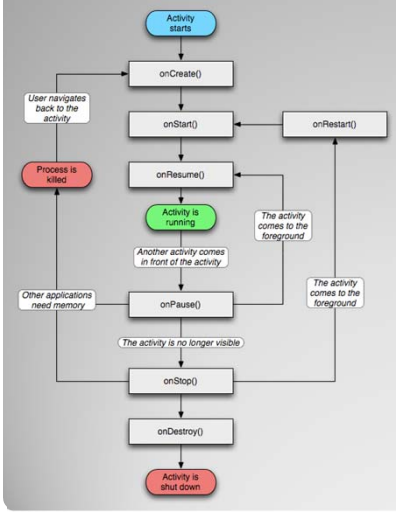
- **Activity Manager** – Controls the lifecycle of our application.
- **Content Providers** – Various objects which allow us to access data from other applications. For example, it is possible to write an app that looks up contact from your phone's contact list.
- **Resource Manager** – Manages all of the resource files (html, graphical, media, etc.) for your application.
- **Location Manager** – GPS related.
- **Notification Manager** – Various alerts and messages.

Applications

There are numerous default applications that are included with Android. Applications such as the Phone Dialer, the default home screen, Contacts, and Phone.

Google also includes their own pack of applications which provide things like the Gmail App, Maps, Market, and several others.

Application Lifecycle



So far, we have been writing our code in a method called onCreate().

Lets take a look at some of the other methods that are available to us.

Methods

onCreate()	Called when the activity is first created. This is where you should do all of your normal static set up — create views, bind data to lists, and so on. This method is passed a Bundle object containing the activity's previous state, if that state was captured. Always followed by onStart().
onRestart()	Called after the activity has been stopped, just prior to it being started again. Always followed by onStart()
onStart()	Called just before the activity becomes visible to the user. Followed by onResume() if the activity comes to the foreground, or onStop() if it becomes hidden.
onResume()	Called just before the activity starts interacting with the user. At this point the activity is at the top of the activity stack, with user input going to it. Always followed by onPause().
onPause()	Called when the system is about to start resuming another activity. This method is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, and so on. It should do whatever it does very quickly, because the next activity will not be resumed until it returns. Followed either by onResume() if the activity returns back to the front, or by onStop() if it becomes invisible to the user.
onStop()	Called when the activity is no longer visible to the user. This may happen because it is being destroyed, or because another activity (either an existing one or a new one) has been resumed and is covering it. Followed either by onRestart() if the activity is coming back to interact with the user, or by onDestroy() if this activity is going away.
onDestroy()	Called before the activity is destroyed. This is the final call that the activity will receive. It could be called either because the activity is finishing (someone called finish() on it), or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between these two scenarios with the isFinishing() method.

http://developer.android.com/guide/topics/fundamentals.html

Application Objects

Here are some basic terms that you should be aware of when it comes to an Android application:

- **Activities** – An activity is a single user screen. Your application may consist of many activities. Each activity must be defined in the AndroidManifest.xml file.
- **Intents** – An intent is a specific action that can be performed within your Activity. You will typically define your own, but you can also use built-in intents. Pretty much anything that the phone can do, can be achieved by calling some specific intent.
- **Services** – A service is a task that runs in the background of the phone.
- **Content Providers** – Content providers provide us with the ability to share data between applications. For example, Google offers a content provider for working with Contacts. This can allow us to build our own custom Contacts app, that still uses the same data as the Phone application.

Additional Resources

Your apps will most likely be more than just the Java code. At the very least, you'll need an icon file. In most cases, you'll have images, html documents, audio, and maybe even video files that are resources to your application.

In Android, all of these resources are stored in the RES folder, which is part of your folder structure in Eclipse. Android's resource compiler (aapt tool) works on these files and produces a class named R that helps you easily reference these items within your application.

AndroidManifest.xml

Every application must have an AndroidManifest.xml file (with precisely that name) in its root directory. The manifest presents essential information about the application to the Android system, information the system must have before it can run any of the application's code. Among other things, the manifest does the following:

- It names the Java package for the application. The package name serves as a unique identifier for the application.
- It describes the components of the application — the activities, services, broadcast receivers, and content providers that the application is composed of. It names the classes that implement each of the components and publishes their capabilities (for example, which Intent messages they can handle). These declarations let the Android system know what the components are and under what conditions they can be launched.
- It determines which processes will host application components.
- It declares which permissions the application must have in order to access protected parts of the API and interact with other applications.
- It also declares the permissions that others are required to have in order to interact with the application's components.
- It lists the Instrumentation classes that provide profiling and other information as the application is running. These declarations are present in the manifest only while the application is being developed and tested; they're removed before the application is published.
- It declares the minimum level of the Android API that the application requires.
- It lists the libraries that the application must be linked against.

http://developer.android.com/guide/topics/manifest/manifest-intro.html

Android Graphical User Interfaces

So far, we've been building applications that are centered around a single activity (we usually call it activity_main). That single activity is responsible for launching our user interface, which it does with this line:

```
setContentView(R.layout.main);
```

Note the main on the end – that refers to the main.xml file, that we minimalistically edit to provide our TextView an id (txtOut).

So our Android applications store their GUIs in XML files. You can build GUIs using code, but Google recommends using the XML way of doing things.

Eclipse provides the basic functionality we need to build our interfaces, although this could be much more improved (and I hope will be).

Layouts and Views

Android relies on a set of objects called Views (widgets) and ViewGroups (layouts).

A View is an individual control such as a textbox, label, button, checkbox, or radiobutton. These are also known as widgets.

A ViewGroup is a special kind of view, that acts like a container. It does not produce anything visual on the screen by itself, but provides a placeholder for where you can place the various views that make up your GUI.

Designing In Eclipse

When you open a layout file (res→layout folder) in Eclipse, you can work with it visually using the "Layout" mode, or edit the XML directly, using the XML tab at the bottom.

You can drag-and-drop layouts or views directly onto the screen, then editing their properties to achieve the designed effect.

You'll quickly find that you can do much of the work visually using the Layout mode, but you'll need to switch back and forth to XML to fix things that you can't control visually.

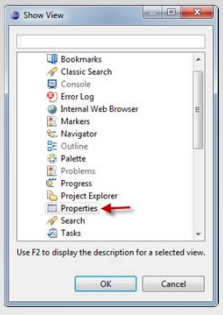
You will also need to work with the Properties panel frequently.

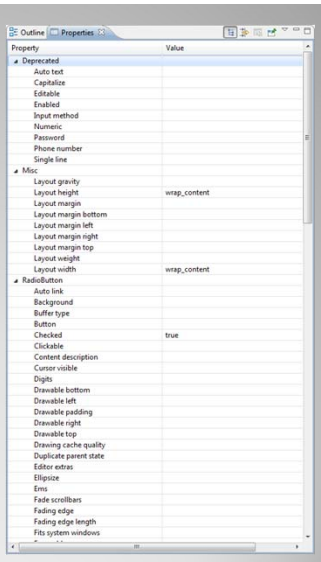
Properties

Most of the properties for each View / ViewGroup can be modified visually as well, using the Properties panel.

To locate the Properties panel, go to:
Window → Show View → Other…

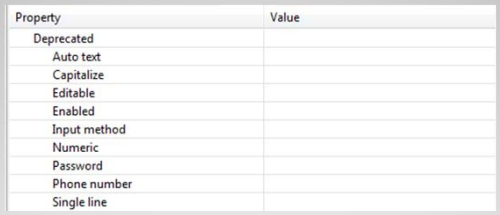
Under **General**, select **Properties**.





Deprecated Properties

Avoid using properties within a section called Deprecated. Deprecated properties will eventually be removed from the Android SDK.



Properties

Each View and Layout will provide its own set of properties, with many properties being common to all controls.

For example, setting the margin for a object can be done by setting a value on Layout_Margin in the Misc section. Similarly, setting an ID for a control can be done by locating the ID property for a given control.

Measuring Things

As you already know, Android provides numerous screen resolutions, and pixel density. This creates a problem for the programmer because the GUIs that are built will display differently at different resolutions and pixel depth.

To solve this problem, Android provides resolution-independent measurements that help solve that problem. The following units of measure are supported, but you should concentrate on sp and dp.

- **px**– Pixels.
- **in**– Inches.
- **mm** – Millimeters.
- **pt** – Points (for fonts). Equal to 1/72 of an inch.
- **dp** – Density Independent Pixel – Use for everything except fonts.
- **dip** – Same as dp.
- **sp** – Scale Independent Pixel – Use for font sizes.

Graphics

When building graphics for your application, be sure to create three versions – one for each screen size (small, normal, large), and density. Alternatively, you can re-use the same graphic, and let Android scale it internally, but you'll have a much better screen if you create the graphics for each resolution.

	Low density (120), ldpi	Medium density (160), mdpi	High density (240), hdpi
Small screen	• QVGA (240x320), 2.6"-3.0" diagonal		
Normal screen	• WQVGA (240x400), 3.2"-3.5" diagonal • FWQVGA (240x432), 3.5"-3.8" diagonal	• HVGA (320x480), 3.0"-3.5" diagonal	• WVGA (480x800), 3.3"-4.0" diagonal • FWVGA (480x854), 3.5"-4.0" diagonal
Large screen		• WVGA (480x800), 4.8"-5.5" diagonal • FWVGA (480x854), 5.0"-5.8" diagonal	

Graphics

For an image that fills the entire screen, start out with a graphic that fits the 854 x 480 resolution. From there you can scale it down to 320 x 480, and 240 x 320 using Photoshop or other graphics application.

This still won't handle all of your resolution scenarios, but will come much closer. Let Android deal with the rest of the minor discrepancies.

Vertical vs. Horizontal

Your phone (and simulator), can change orientation. I can view an application vertically, or horizontally.

As a developer, you have a choice for how to support this:

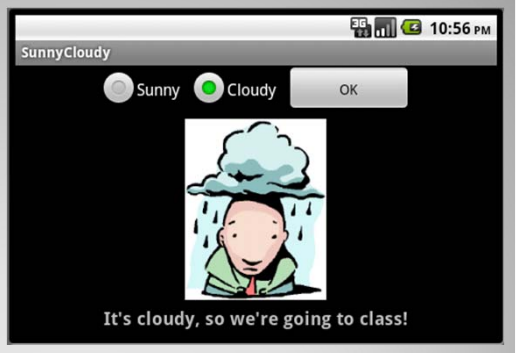
1 – You build one interface only, and disable the other in the Android Manifest file. For example, if you only want to support portrait mode, you can disable the screen from changing in any way when viewed in landscape mode.

2 – You can make your one XML file work to support both orientations. This is typically tricky, and will leave your GUIs sparse, and confusing.

3 – Build multiple versions of your GUI, one for Vertical, and one for Horizontal layout.

Personally, I find that if you can't do what you want with option 1, option 3 is your only other choice.

Our Sunny / Cloudy App



A good example of where a "-land" layout is needed, is our Sunny / Cloudy application. By default, our portrait layout breaks when viewing the application in landscape mode. Creating a –land layout, produces the results above.

Resource Layout Folders

The res project folder is responsible for holding all of your layout, graphical, and media files.

For layouts, you can use the following folders:

- res
  - layout — Normal screen, portrait mode
  - layout-land — Normal screen, landscape mode
  - layout-small — Small screen, portrait mode
  - layout-small-land — Small screen, landscape mode
  - layout-large — Large screen, portrait mode
  - layout-large-land — Large screen, landscape mode

Note: Most of these folders are NOT created for you automatically. In fact the only folder that you'll find by default is "layout". It is up to the developer to determine whether or not you need one of these alternate folders, and to create them using Eclipse.

Resource Graphics Folders

Just like the layout folders, drawable holds the density dependent graphics.

- Res
  - drawable-ldpi — Low density images
  - drawable-mdpi — Medium density images
  - drawable-hdpi — High density images
  - drawable-nodpi — Density independent

Best Practice

According to Google, the following are the best practices for screen independence:

The objective of supporting multiple screens is to create an application that can run properly on any display and function properly on any of the screen configurations listed in Table 1 earlier in this document.

You can easily ensure that your application will display properly on different screens. Here is a quick checklist:

1. Prefer wrap_content, fill_parent and the dip unit to px in XML layout files
2. Avoid AbsoluteLayout
3. Do not use hard coded pixel values in your code
4. Use density and/or resolution specific resources

http://developer.android.com/guide/practices/screens_support.html