Android

Getting Started with Android

To get started with Android, please read the following sections first:

Installing the SDK and Plugin

How to install the Android SDK and Eclipse plugin.

Developing and Debugging

An introduction to developing and debugging Android applications in Eclipse, plus information on using other IDEs.

Hello Android

Writing your first Android Application, the ever popular Hello World, Android style.

Anatomy of an App

A guide to the structure and architecture of an Android Application. This guide will help you understand the pieces that make up an Android app.

Notepad Tutorial

This tutorial document will lead you through constructing a real Android Application: A notepad which can create, edit and delete notes, and covers many of the basic concepts with practical examples.

Development Tools

The command line tools included with the SDK, what they do, and how to use them.

Application Model

A guide to Applications, Tasks, Processes, and Threads. These are the elements that define the way your application is run by the system and presented to the user.

Application Life Cycle

The important life-cycle details for Applications and the Activities running inside of them.

Other Introductory Material

After reading the sections above, the following Getting Started information is also very useful:

Core Packages

These are the basic packages that make up the Android SDK for writing applications. The packages are organized as layers, listed here from lowest-level to highest.

android.util

contains various low-level utility classes, such as specialized container classes, XML utilities, etc.

android.os

provides basic operating system services, message passing, and inter-process communication.

android.graphics

is the core rendering package.

android.text, android.text.method, android.text.style, and android.text.util

supply a rich set of text processing tools, supporting rich text, input methods, etc.

android.database

contains low-level APIs for working with databases.

android.content

provides various services for accessing data on the device: applications installed on the device and their associated resources, and content providers for persistent dynamic data.

android.view

is the core user-interface framework.

android.widget

supplies standard user interface elements (lists, buttons, layout managers, etc) built from the view package.

android.app

provides the high-level application model, implemented using Activities.

Other Notable Packages

These packages provide additional domain-specific features of the Android platform. They are not necessary for basic application development.

android.provider

contains definitions for various standard content providers included with the platform.

android.telephony

provides APIs for interacting with the device's phone stack.

android.webkit

includes various APIs for working with web-based content.

 Build 110632-110632 - 22 Sep 2008 13:34

Android

Installing the SDK

This page describes how to install the Android SDK and set up your development environment. If you haven't downloaded the SDK yet, you can use the link below to get started. Then read the rest of this document to learn how to install, configure, and use the SDK to create Android applications.

Download the SDK

Upgrading?

If you have already developed applications using an earlier version of the SDK, please skip this page and read the **Upgrading the SDK** document.

System and Software Requirements

To develop Android applications using the code and tools in the Android SDK, you need a suitable development computer and development environment, as described below.

Supported Operating Systems:

- Windows XP or Vista
- Mac OS X 10.4.8 or later (x86 only)
- Linux (tested on Linux Ubuntu Dapper Drake)

Supported Development Environments:

- Eclipse IDE
  - Eclipse 3.3 (Europa), 3.4 (Ganymede)
    - Eclipse JDT plugin (included in most Eclipse IDE packages)
    - WST (optional, but needed for the Android Editors feature; included in most Eclipse IDE packages)
  - JDK 5 or JDK 6 (JRE alone is not sufficient)
  - Android Development Tools plugin (optional)
  - **Not** compatible with Gnu Compiler for Java (gcj)
- Other development environments or IDEs
  - JDK 5 or JDK 6 (JRE alone is not sufficient)
  - Apache Ant 1.6.5 or later for Linux and Mac, 1.7 or later for Windows
  - **Not** compatible with Gnu Compiler for Java (gcj)

Note: If JDK is already installed on your development computer, please take a moment to make sure that it meets the version requirements listed above. In particular, note that some Linux distributions may include JDK 1.4 or Gnu Compiler for Java, both of which are not supported for Android development.

Installing the SDK

After downloading the SDK, unpack the .zip archive to a suitable location on your machine. By default, the SDK files are unpacked into a directory named `android_sdk_<platform>_<release>_<build>`. The directory contains the subdirectories `tools/`, `samples/`, and others.

Make a note of the name and location of the unpacked SDK directory on your system — you will need to refer to the SDK directory later, when setting up the Android plugin or using SDK tools.

Optionally, you can add the path to the SDK `tools` directory to your path. As mentioned above, the `tools/` directory is located

in the SDK directory.

- On Linux, edit your ~/.bash_profile or ~/.bashrc file. Look for a line that sets the PATH environment variable and add the full path to the `tools/` directory to it. If you don't see a line setting the path, you can add one:

  ```
  export PATH=${PATH}:<your_sdk_dir>/tools
  ```

- On a Mac, look in your home directory for .bash_profile and proceed as for Linux. You can create the .bash_profile, if you haven't already set one up on your machine.
- On Windows, right click on My Computer, and select Properties. Under the Advanced tab, hit the Environment Variables button, and in the dialog that comes up, double-click on Path under System Variables. Add the full path to the `tools/` directory to the path.

Adding `tools` to your path lets you run Android Debug Bridge (adb) and the other command line tools without needing to supply the full path to the tools directory. Note that, if you update your SDK, you should remember to update your PATH settings to point to the new location, if different.

Installing the Eclipse Plugin (ADT)

If you will be using the Eclipse IDE as your environment for developing Android applications, you can install a custom plugin called Android Development Tools (ADT), which adds integrated support for Android projects and tools. The ADT plugin includes a variety of powerful extensions that make creating, running, and debugging Android applications faster and easier.

If you *will not* be using the Eclipse IDE, you do not need to download or install the ADT plugin.

To download and install the ADT plugin, follow the steps below for your respective Eclipse version.

Eclipse 3.3 (Europa)

1. Start Eclipse, then select **Help** > **Software Updates** > **Find and Install...**.
2. In the dialog that appears, select **Search for new features to install** and click **Next**.
3. Click **New Remote Site**.
4. In the resulting dialog box, enter a name for the remote site (e.g. Android Plugin) and enter this as its URL:

   ```
   https://dl-ssl.google.com/android/eclipse/
   ```

   Click **OK**.
5. You should now see the new site added to the search list (and checked). Click **Finish**.
6. In the subsequent Search Results dialog box, select the checkbox for **Android Plugin** > **Developer Tools**. This will check both features: "Android Developer Tools", and "Android Editors". The Android Editors feature is optional, but recommended. If you choose to install it, you need the WST plugin mentioned earlier in this page. Click **Next**.
7. Read the license agreement and then select **Accept terms of the license agreement**. Click **Next**.
8. Click **Finish**.
9. The ADT plugin is not signed; you can accept the installation anyway by clicking **Install All**.
10. Restart Eclipse.

Eclipse 3.4 (Ganymede)

1. Start Eclipse, then select **Help** > **Software Updates...**.
2. In the dialog that appears, click the **Available Software** tab.
3. Click **Add Site...**
4. Enter this as the Location:

   ```
   https://dl-ssl.google.com/android/eclipse/
   ```

   Click **OK**.
5. Back in the Available Software view, you should see the plugin. Select the checkbox next to *Developer Tools* and click **Install...**
6. On the subsequent Install window, "Android Developer Tools", and "Android Editors" should both be checked. The Android Editors feature is optional, but recommended. If you choose to install it, you need the WST plugin mentioned earlier in this page. Click **Finish**.
7. Restart Eclipse.

After restart, **update your Eclipse preferences** to point to the SDK directory:

1. Select **Window** > **Preferences...** to open the Preferences panel. (Mac OS X: **Eclipse** > **Preferences**)
2. Select **Android** from the left panel.

3. For the SDK Location in the main panel, click **Browse...** and locate the SDK directory.
4. Click **Apply**, then **OK**.

Troubleshooting ADT Installation

If you are having trouble downloading the ADT plugin after following the steps above, here are some suggestions:

- In Step 4, try changing the remote update site URL to use `http`, rather than `https`.
- If you are behind a firewall (such as a corporate firewall), make sure that you have properly configured your proxy settings in Eclipse. In Eclipse 3.3/3.4, you can configure proxy information from the main Eclipse menu in **Window** (on Mac, **Eclipse**) > **Preferences** > **General** > **Network Connections**.

If you are still unable to use Eclipse to download the ADT plugin, follow these steps to download and install the plugin from your computer:

1. Download the ADT zip file (do not unpack it).
2. Follow steps 1 and 2 in the default install instructions (above).
3. In Eclipse 3.3, click **New Archive Site...**.
   In Eclipse 3.4, click **Add Site...**, then **Archive...**
4. Browse and select the downloaded the zip file.
5. Follow the remaining procedures, above, starting from steps 5.

Note that to update your plugin, you will have to follow these steps again instead of the default update instructions.

Note that the "Android Editors" feature of ADT requires several optional Eclipse components (for example, WST). If you encounter an error when installing ADT, your Eclipse installion might not include those components. For information about how to quickly add the necessary components to your Eclipse installation, see the troubleshooting topic ADT Installation Error: "requires plug-in org.eclipse.wst.sse.ui".

Updating the ADT Plugin

In some cases, a new ADT plugin may become available for your existing version of the SDK. You can use the steps below to update the ADT plugin from inside Eclipse.

Eclipse 3.3 (Europa)	Eclipse 3.4 (Ganymede)
1. Select **Help** > **Software Updates** > **Find and Install...**. 2. Select **Search for updates of the currently installed features** and click **Finish**. 3. If an update for ADT is available, select and install. Alternatively, 1. Select **Help** > **Software Updates** > **Manage Configuration**. 2. Navigate down the tree and select **Android Development Tools <version>** 3. Select **Scan for Updates** under **Available Tasks**.	1. Select **Help** > **Software Updates...** 2. Select the **Installed Software** tab. 3. Click **Update...** 4. If an update for ADT is available, select it and click **Finish**.

Installation Notes

Ubuntu Linux Notes

- If you need help installing and configuring Java on your development machine, you might find these resources helpful:
  - https://help.ubuntu.com/community/Java
  - https://help.ubuntu.com/community/JavaInstallation
- Here are the steps to install Java and Eclipse, prior to installing the Android SDK and ADT Plugin.
  1. If you are running a 64-bit distribution on your development machine, you need to install the `ia32-libs` package

using `apt-get:`:

```
apt-get install ia32-libs
```

2. Next, install Java:

```
apt-get install sun-java6-bin
```

3. The Ubuntu package manager does not currently offer an Eclipse 3.3 version for download, so we recommend that you download Eclipse from eclipse.org (http://www.eclipse.org/ downloads/). A Java or RCP version of Eclipse is recommended.
4. Follow the steps given in previous sections to install the SDK and the ADT plugin.

Other Linux Notes

- If you encounter this error when installing the ADT Plugin for Eclipse:

```
An error occurred during provisioning.
Cannot connect to keystore.
JKS
```

your development machine lacks a suitable Java VM. Installing Sun Java 6 will resolve this issue and you can then reinstall the ADT Plugin.

- If JDK is already installed on your development computer, please take a moment to make sure that it meets the version requirements listed at the top of this page. In particular, note that some Linux distributions may include JDK 1.4 or Gnu Compiler for Java, both of which are not supported for Android development.

 Build 110632-110632 - 22 Sep 2008 13:34

Android

Develop and Debug

This page offers an introduction to developing and debugging applications on Android. It teaches how to create, build, run and debug your Android code. Alternatively, you may like to begin with the Hello Android tutorial.

Contents

Developing Android Applications on Eclipse

To begin developing Android applications in the Eclipse IDE, you first create an Android project and then set up a launch configuration. After that, you can write, run, and debug your application.

The sections below provide instructions assuming that you have installed the ADT plugin in your Eclipse environment. If you haven't installed the ADT plugin, you should do that before using the instructions below. See the Installing the Eclipse Plugin (ADT) for more information.

Creating an Android Project

The ADT plugin provides a New Project Wizard that you can use to quickly create an Eclipse project for new or existing code. To create the project, follow these steps:

1. Select **File** > **New** > **Project**
2. Select **Android** > **Android Project**, and press **Next**
3. Select the contents for the project:
   - Select **Create new project in workspace** to start a project for new code.

     Enter the project name, the base package name, the name of a single Activity class to create as a stub .java file, and a name to use for your application.
   - Select **Create project from existing source** to start a project from existing code. Use this option if you want to build and run any of the sample applications included with the SDK. The sample applications are located in the samples/ directory in the SDK.

     Browse to the directory containing the existing source code and click OK. If the directory contains a valid Android manifest file, the ADT plugin fills in the package, activity, and application names for you.
4. Press **Finish**.

The ADT plugin creates the these folders and files for you as appropriate for the type of project:

- src/ A folder that includes your stub .java Activity file.

- res/ A folder for your resources.
- AndroidManifest.xml The manifest for your project.

Creating a Launch Configuration

Before you can run and debug your application in Eclipse, you must create a launch configuration for it. A launch configuration specifies the project to launch, the Activity to start, the emulator options to use, and so on.

To create a launch configuration for the application, follow these steps as appropriate for your Eclipse version:

1. Open the launch configuration manager.
    - In Eclipse 3.3 (Europa), select **Run** > **Open Run Dialog...** or **Run** > **Open Debug Dialog...** as appropriate.
    - In Eclipse 3.4 (Ganymede), select **Run** > **Run Configurations...** or **Run** > **Debug Configurations...** as appropriate.
2. In the project type list on the left, locate the **Android Application** item and double-click it (or right-click > **New**), to create a new launch configuration.
3. Enter a name for your configuration.
4. On the Android tab, browse for the project and Activity to start.
5. On the Target tab, set the desired screen and network properties, as well as any other emulator startup options.
6. You can set additional options on the Common tab as desired.
7. Press **Apply** to save the launch configuration, or press **Run** or **Debug** (as appropriate).

Running and Debugging an Application

Once you've set up the project and launch configuration for your application, you can run or debug it as described below. From the Eclipse main menu, select **Run** > **Run** or **Run** > **Debug** as appropriate, to run or debug the active launch configuration.

Note that the active launch configuration is the one most recently selected in the Run configuration manager. It does not necessarily correspond to the application that is selected in the Eclipse Navigation pane (if any).

To set or change the active launch configuration, use the launch configuration manager. See Creating a Launch Configuration for information about how to access the launch configuration manager..

Running or debugging the application triggers these actions:

- Starts the emulator, if it is not already running.
- Compiles the project, if there have been changes since the last build, and installs the application on the emulator.
- **Run** starts the application.
- **Debug** starts the application in "Wait for debugger" mode, then opens the Debug perspective and attaches the Eclipse Java debugger to the application.

Developing Android Applications with Other IDEs and Tools

The recommended way to develop an Android application is to use Eclipse with the ADT plugin. This plugin provides editing, building, and debugging functionality integrated right into the IDE.

However, if you'd rather develop your application in another IDE, such as IntelliJ, or use Eclipse without the ADT plugin, you can do that instead. The SDK provides the tools you need to set up, build, and debug your application.

Creating an Android Project

The Android SDK includes `activityCreator`, a program that generates a number of stub files for your project, as well as a build file. You can use the program to create an Android project for new code or from existing code, such as the sample applications included in the SDK. For Linux and Mac, the SDK provides `activityCreator.py`, a Python script, and for Windows, `activityCreator.bat`, a batch script. Regardless of platform, you can use `activityCreator` in the same way.

To run `activityCreator` and create an Android project, follow these steps:

1. In the command line, change to the `tools/` directory of the SDK and create a new directory for your project files. If you

are creating a project from existing code, change to the root folder of your application instead.

2. Run `activityCreator`. In the command, you must specify a fully-qualified class name as an argument. If you are creating a project for new code, the class represents the name of a stub class that the script will create. If you are creating a project from existing code, you must specify the name of one Activity class in the package. Command options for the script include:
   - `--out <folder>` which sets the output directory. By default, the output directory is the current directory. If you created a new directory for your project files, use this option to point to it.
   - `--ide intellij`, which generates IntelliJ IDEA project files in the newly created project

Here's an example:

```
~/android_linux_sdk/tools $ ./activityCreator.py --out myproject your.package.name.ActivityName
package: your.package.name
out_dir: myproject
activity_name: ActivityName
~/android_linux_sdk/tools $
```

The `activityCreator` script generates the following files and directories (but will not overwrite existing ones):

- `AndroidManifest.xml` The application manifest file, synced to the specified Activity class for the project.
- `build.xml` An `Ant` file that you can use to build/package the application.
- `src/your/package/name/ActivityName.java` The Activity class you specified on input.
- *`your_activity`*`.iml, `*`your_activity`*`.ipr, `*`your_activity`*`.iws` [*only with the `-ide intelliJ` flag*] intelliJ project files.
- `res/` A directory to hold resources.
- `src/` The source directory.
- `bin/` The output directory for the build script.

You can now move your folder wherever you want for development, but keep in mind that you'll have to use the adb program in the `tools/` folder to send files to the emulator, so you'll need access between your solution and the `tools/` folder.

Also, you should refrain from moving the location of the SDK directory, since this will break the build scripts (they will need to be manually updated to reflect the new SDK location before they will work again).

Building an Android Application

Use the Ant `build.xml` file generated by `activityCreator` to build your application.

1. If you don't have it, you can obtain Ant from the Apache Ant home page. Install it and make sure it is on your executable path.
2. Before calling Ant, you need to declare the JAVA_HOME environment variable to specify the path to where the JDK is installed.

> Note: When installing JDK on Windows, the default is to install in the "Program Files" directory. This location will cause `ant` to fail, because of the space. To fix the problem, you can specify the JAVA_HOME variable like this: `set JAVA_HOME=c:\Prora~1\Java\`. The easiest solution, however, is to install JDK in a non-space directory, for example: `c:\java\jdk1.6.0_02`.

3. If you have not done so already, follow the instructions for Creating a New Project above to set up the project.
4. You can now run the Ant build file by simply typing ant in the same folder as the build.xml file for your project. Each time you change a source file or resource, you should run ant again and it will package up the latest version of the application for you to deploy.

Running an Android Application

To run a compiled application, you will upload the .apk file to the `/data/app/` directory in the emulator using the adb tool as described here:

1. Start the emulator (run `<your_sdk_dir>/tools/emulator` from the command line)
2. On the emulator, navigate to the home screen (it is best not to have that application running when you reinstall it on the emulator; press the **Home** key to navigate away from that application).

3. Run `adb install myproject/bin/<appname>.apk` to upload the executable. So, for example, to install the Lunar Lander sample, navigate in the command line to `<your_sdk_dir>/sample/LunarLander` and type `../../tools/adb install bin/LunarLander.apk`
4. In the emulator, open the list of available applications, and scroll down to select and start your application.

Note: When you install an Activity for the first time, you might have to restart the emulator before it shows up in the application launcher, or other applications can call it. This is because the package manager usually only examines manifests completely on emulator startup.

Attaching a Debugger to Your Application

This section describes how to display debug information on the screen (such as CPU usage), as well as how to hook up your IDE to debug running applications on the emulator.

Attaching a debugger is automated using the Eclipse plugin, but you can configure other IDEs to listen on a debugging port to receive debugging information.

1. **Start the Dalvik Debug Monitor Server (DDMS) tool ,** which acts as a port forwarding service between your IDE and the emulator.
2. **Set optional debugging configurations on your emulator**, such as blocking application startup for an activity until a debugger is attached. Note that many of these debugging options can be used without DDMS, such as displaying CPU usage or screen refresh rate on the emulator.
3. **Configure your IDE to attach to port 8700 for debugging.** We include information on how to set up Eclipse to debug your project.

Configuring your IDE to attach to the debugging port

DDMS will assign a specific debugging port to every virtual machine that it finds on the emulator. You must either attach your IDE to that port (listed on the Info tab for that VM), or you can use a default port 8700 to connect to whatever application is currently selected on the list of discovered virtual machines.

Your IDE should attach to your application running on the emulator, showing you its threads and allowing you to suspend them, inspect their state, and set breakpoints. If you selected "Wait for debugger" in the Development settings panel the application will run when Eclipse connects, so you will need to set any breakpoints you want before connecting.

Changing either the application being debugged or the "Wait for debugger" option causes the system to kill the selected application if it is currently running. You can use this to kill your application if it is in a bad state by simply going to the settings and toggling the checkbox.

Signing Your Applications

The Android system requires that all installed applications are digitally signed — the system will not install or run an application that is not signed appropriately. This applies wherever the Android system is run, whether on an actual device or on the emulator. For this reason, you must set up signing for your application before you will be able to run or debug it on an emulator or device.

The important points to understand about signing Android applications are:

- All applications *must* be signed. The system will not install an application that is not signed.
- You can use self-signed certificates to sign your applications. No certificate authority is needed.
- The system tests a signer certificate's expiration date only at install time. If an application's signer certificate expires after the application is installed, the application will continue to function normally.
- You can use standard tools — Keytool and Jarsigner — to generate keys and sign your application .apk files.

The Android SDK tools assist you in signing your applications when debugging. Both the ADT Plugin for Eclipse and the Ant build tool offer two signing modes — debug mode and release mode.

- In debug mode, the build tools use the Keytool utility, included in the JDK, to create a keystore and key with a known alias and password. At each compilation, the tools then use the debug key to sign the application .apk file. Because the password is known, the tools don't need to prompt you for the keystore/key password each time you compile.
- When your application is ready for release, you compile it in release signing mode. In release mode, the tools compile your .apk without signiing it. You must then use Keytool to generate your own keystore/key and then use the Jarsigner

tool, also included in the JDK, to sign the .apk.

Basic Setup for Signing

To support the generation of a keystore and debug key, you should first make sure that Keytool is available to the SDK build tools. In most cases, you can tell the SDK build tools how to find Keytool by making sure that your JAVA_HOME environment variable is set and that it references a suitable JDK. Alternatively, you can add the JDK version of Keytool to your PATH variable.

If you are developing on a version of Linux that originally came with Gnu Compiler for Java, make sure that the system is using the JDK version of Keytool, rather than the gcj version. If Keytool is already in your PATH, it might be pointing to a symlink at /usr/bin/keytool. In this case, check the symlink target to make sure that it points to the Keytool in the JDK.

Signing in Eclipse/ADT

If you are developing in Eclipse and have set up Keytool as described above, signing in debug mode is enabled by default. When you run or debug your app, ADT signs the .apk for you and installs it on the emulator. No specific action on your part is needed, provided ADT has access to Keytool.

To compile your application in release mode, right-click the project in the Package pane and select Android Tools > Export Application Package. Alternatively, you can follow the "Exporting the unsigned .apk" link in the Manifest Editor overview page. After you have saved the exported .apk, you need to use Jarsigner to sign the .apk with your own key before distribution. If you don't have a key, you can use Keystore to create a keystore and key with all the appropriate fields. If you already have a key, such as a corporate key, you can use that to sign the .apk.

Signing in Ant

If you use Ant to build your .apk files, debug signing mode is enabled by default, assuming that you are using a build.xml file generated by the activitycreator tool included in the latest SDK. When you run Ant against build.xml to compile your app, the build script generates a keystore/key and signs the .apk for you. No specific action on your part is needed.

To compile your application in release mode, all you need to do is specify a build target "release" in the Ant command. For example, if you are running Ant from the directory containing your build.xml file, the command would look like this:

```
ant release
```

The build script compiles the application .apk without signing it. After you have compiled the.apk, you need to use Jarsigner to sign the .apk with your own key before distribution. If you don't have a key, you can use Keystore to create a keystore and key with all the appropriate fields. If you already have a key, such as a corporate key, you can use that to sign the .apk.

Expiry of the Debug Certificate

The self-signed certificate used to sign your application in debug mode (the default on Eclipse/ADT and Ant builds) will have an expiration date of 1 year from its creation date.

When the certificate expires, you will get a build error. On Ant builds, the error looks like this:

```
debug:
[echo] Packaging bin/samples-debug.apk, and signing it with a debug key...
[exec] Debug Certificate expired on 8/4/08 3:43 PM
```

In Eclipse/ADT, you will see a similar error in the Android console.

To fix this problem, simply delete the `debug.keystore` file. On Linux/Mac OSX, the file is stored in `~/.android`. OOn Windows XP, the file is stored in `C:\Documents and Settings\<user>\Local Settings\Application Data\Android`. On Windows Vista, the file is stored in `C:\Users\<user>\AppData\Local\Android`.

The next time you build, the build tools will regenerate a new keystore and debug key.

Note that, if your development machine is using a non-Gregorian locale, the build tools may erroneously generate an already-

expired debug certificate, so that you get an error when trying to compile your application. For workaround information, see the troubleshooting topic I can't compile my app because the build tools generated an expired debug certificate.

Using the ApiDemos Sample Applications

The Android SDK includes a set of sample applications that demonstrate much of the functionality and API usage needed for your applications. The ApiDemos package is preinstalled on the emulator, so you can access it by starting an emulator and sliding open the home screen's application drawer.

You can find the source code corresponding to the ApiDemos apps in `<SDK> /samples/ApiDemos` and look at it to learn more about how it is implemented.

If you want, you can load the ApiDemos sample applications as source projects and modify them, then run them in the emulator. However, to do so, you need to uninstall the preinstalled version of ApiDemos first. If you try to run or modify ApiDemos from your development environment without removing the preinstalled version first, you will get an install error.

For information about how to uninstall and then reinstall ApiDemos so that you can work with them in your development environment, see the troubleshooting topic I can't install ApiDemos apps in my IDE because of a signing error.

Debugging

Android has a fairly extensive set of tools to help you debug your programs:

- **DDMS** - A graphical program that supports port forwarding (so you can set up breakpoints in your code in your IDE), screen captures on the emulator, thread and stack information, and many other features. You can also run logcat to retrieve your Log messages. See the linked topic for more information.
- **logcat** - Dumps a log of system messages. The messages include a stack trace when the emulator throws an error, as well as Log messages. To run logcat, see the linked topic.

  ```
  ...
  I/MemoryDealer( 763): MemoryDealer (this=0x54bda0): Creating 2621440 bytes heap at 0x438db000
  I/Logger( 1858): getView() requesting item number 0
  I/Logger( 1858): getView() requesting item number 1
  I/Logger( 1858): getView() requesting item number 2
  D/ActivityManager( 763): Stopping: HistoryRecord{409dbb20 com.android.home.AllApps}
  ...
  ```

- **Android Log**- A logging class to print out messages to a log file on the emulator. You can read messages in real time if you run logcat on DDMS (covered next). Add a few logging method calls to your code.

  To use the `Log` class, you just call `Log.v()` (verbose), `Log.d()` (debug), `Log.i()` (information), `Log.w()` (warning) or `Log.e` (error) depending on the importance you wish to assign the log message.

  ```
  Log.i("MyActivity", "MyClass.getView() — Requesting item number " + position)
  ```

  You can use logcat to read these messages

- **Traceview** - Android can save a log of method calls and times to a logging file that you can view in a graphical reader called Traceview. See the linked topic for more information.
- **Eclipse plugin** - The Eclipse Android plugin incorporates a number of these tools (ADB, DDMS, logcat output, and other functionality). See the linked topic for more information.
- **Debug and Test Device Settings** - Android exposes several settings that expose useful information such as CPU usage and frame rate. See Debug and Test Settings on the Emulator below.

Also, see the Troubleshooting section of the doc to figure out why your application isn't appearing on the emulator, or why it's not starting.

Debug and Test Settings on the Device

Android lets you set a number of settings that will make it easier to test and debug your applications. To get to the development settings page on the emulator, go to **Dev Tools** > **Development Settings**. This will open the development

settings page with the following options (among others):

- **Debug app** Selects the application that will be debugged. You do not need to set this to attach a debugger, but setting this value has two effects:
  - It will prevent Android from throwing an error if you pause on a breakpoint for a long time while debugging.
  - It will enable you to select the *Wait for Debugger* option to pause application startup until your debugger attaches (described next).
- **Wait for debugger** Blocks the selected application from loading until a debugger attaches. This way you can set a breakpoint in onCreate(), which is important to debug the startup process of an Activity. When you change this option, any currently running instances of the selected application will be killed. In order to check this box, you must have selected a debug application as described in the previous option. You can do the same thing by adding waitForDebugger() to your code.
- **Immediately destroy activities** Tells the system to destroy an activity as soon as it is stopped (as if Android had to reclaim memory). This is very useful for testing the onSaveInstanceState(Bundle) / onCreate(android.os.Bundle) code path, which would otherwise be difficult to force. Choosing this option will probably reveal a number of problems in your application due to not saving state.
- **Show screen updates** Flashes a momentary pink rectangle on any screen sections that are being redrawn. This is very useful for discovering unnecessary screen drawing.
- **Show CPU usage** Displays CPU meters at the top of the screen, showing how much the CPU is being used. The top red bar shows overall CPU usage, and the green bar underneath it shows the CPU time spent in compositing the screen. *Note: You cannot turn this feature off once it is on, without restarting the emulator.*
- **Show background** Displays a background pattern when no activity screens are visible. This typically does not happen, but can happen during debugging.

These settings will be remembered across emulator restarts.

Top Debugging Tips

Quick stack dump

To obtain a stack dump from emulator, you can log in with `adb shell`, use "ps" to find the process you want, and then "kill -3 ". The stack trace appears in the log file.

Displaying useful info on the emulator screen

The device can display useful information such as CPU usage or highlights around redrawn areas. Turn these features on and off in the developer settings window as described in Setting debug and test configurations on the emulator.

Getting system state information from the emulator (dumpstate)

You can access dumpstate information from the Dalvik Debug Monitor Service tool. See dumpsys and dumpstate on the adb topic page.

Getting application state information from the emulator (dumpsys)

You can access dumpsys information from the Dalvik Debug Monitor Service tool. See dumpsys and dumpstate on the adb topic page.

Getting wireless connectivity information

You can get information about wireless connectivity using the Dalvik Debug Monitor Service tool. From the **Device** menu, select "Dump radio state".

Logging Trace Data

You can log method calls and other tracing data in an activity by calling android.os.Debug.startMethodTracing(). See Running the Traceview Debugging Program for details.

Logging Radio Data

By default, radio information is not logged to the system (it is a lot of data). However, you can enable radio logging using the following commands:

```
adb shell
logcat -b radio
```

Running adb

Android ships with a tool called adb that provides various capabilities, including moving and syncing files to the emulator, forwarding ports, and running a UNIX shell on the emulator. See Using adb for details.

Getting screen captures from the emulator

Dalvik Debug Monitor Server (DDMS) can capture screenshots from the emulator.

Using debugging helper classes

Android provides debug helper classes such as util.Log and Debug for your convenience.

Building and Installing an Android Application

Android requires custom build tools to be able to properly build the resource files and other parts of an Android application. Because of this, you must have a specialized build environment for your application.

Custom Android compilation steps include compiling the XML and other resource files, and creating the proper output format. A compiled Android application is an .apk file, which is a compressed file containing .dex files, resource files, raw data files, and other files. You can create a properly structured Android project either from scratch, or from existing source files.

Android does not currently support development of third party applications in native code (C/C++).

The recommended way to develop an Android application is to use Eclipse with the Android plugin, which provides support for building, running, and debugging Android applications.

If you have another IDE, Android provides tools for other IDEs to build and debug Android applications, but they are not as integrated.

Removing an Android Application

To remove an application that you have installed on the emulator, you will need to run adb and delete the .apk file you sent to the emulator when you installed it. Use `adb shell` to drop into a shell on the device as described in the linked topic, navigate to `data/app/`, and then remove the file using `rm your_app.apk`.

Eclipse Tips

Executing arbitrary Java expressions in Eclipse

You can execute arbitrary code when paused at a breakpoint in Eclipse. For example, when in a function with a String argument called "zip", you can get information about packages and call class methods. You can also invoke arbitrary static methods: for example, entering `android.os.Debug.startMethodTracing()` will start dmTrace.

Open a code execution window, select **Window**>**Show View**>**Display** from the main menu to open the Display window, a simple text editor. Type your expression, highlight the text, and click the 'J' icon (or CTRL + SHIFT + D) to run your code. The code runs in the context of the selected thread, which must be stopped at a breakpoint or single-step point. (If you suspend the thread manually, you have to single-step once; this doesn't work if the thread is in Object.wait().)

If you are currently paused on a breakpoint, you can simply highlight and execute a piece of source code by pressing CTRL + SHIFT + D.

You can highlight a block of text within the same scope by pressing ALT +SHIFT + UP ARROW to select larger and larger enclosing blocks, or DOWN ARROW to select smaller blocks.

Here are a few sample inputs and responses in Eclipse using the Display window.

Input	Response
`zip`	`(java.lang.String) /work/device/out/linux-x86-debug/android/app/android_sdk.zip`
`zip.endsWith(".zip")`	`(boolean) true`

`zip.endsWith(".jar")`	`(boolean) false`

You can also execute arbitrary code when not debugging by using a scrapbook page. Search the Eclipse documentation for "scrapbook".

Running DDMS Manually

Although the recommended way to debug is to use the ADT plugin, you can manually run DDMS and configure Eclipse to debug on port 8700. (**Note:** Be sure that you have first started DDMS).

Adding JUnit test classes

In Eclipse/ADT, you can add JUnit test classes to your application. However, you need to set up a custom JUnit configuration before your tests will run properly. For detailed information about how to set up the JUnit configuration, see the troubleshooting topic I can't run a Junit test class in Eclipse.

Build 110632-110632 - 22 Sep 2008 13:34

Android

Hello, Android!

First impressions matter, and as a developer you know that the first impression you get of a development framework is how easy it is to write "Hello, World!" Well, in Android, it's pretty easy. Here's how it looks:

- Create the Project
- Construct the UI
- Run the Code: Hello, Android

The sections below spell it all out in detail.

- Upgrading the UI to an XML Layout
- Debugging Your Project
- Creating a Project without Eclipse

Let's jump in!

Create the Project

Creating the project is as simple as can be. An Eclipse plugin is available making Android development a snap.

You'll need to have a development computer with the Eclipse IDE installed (see System and Software Requirements), and you'll need to install the Android Eclipse Plugin (ADT). Once you have those ready, come back here.

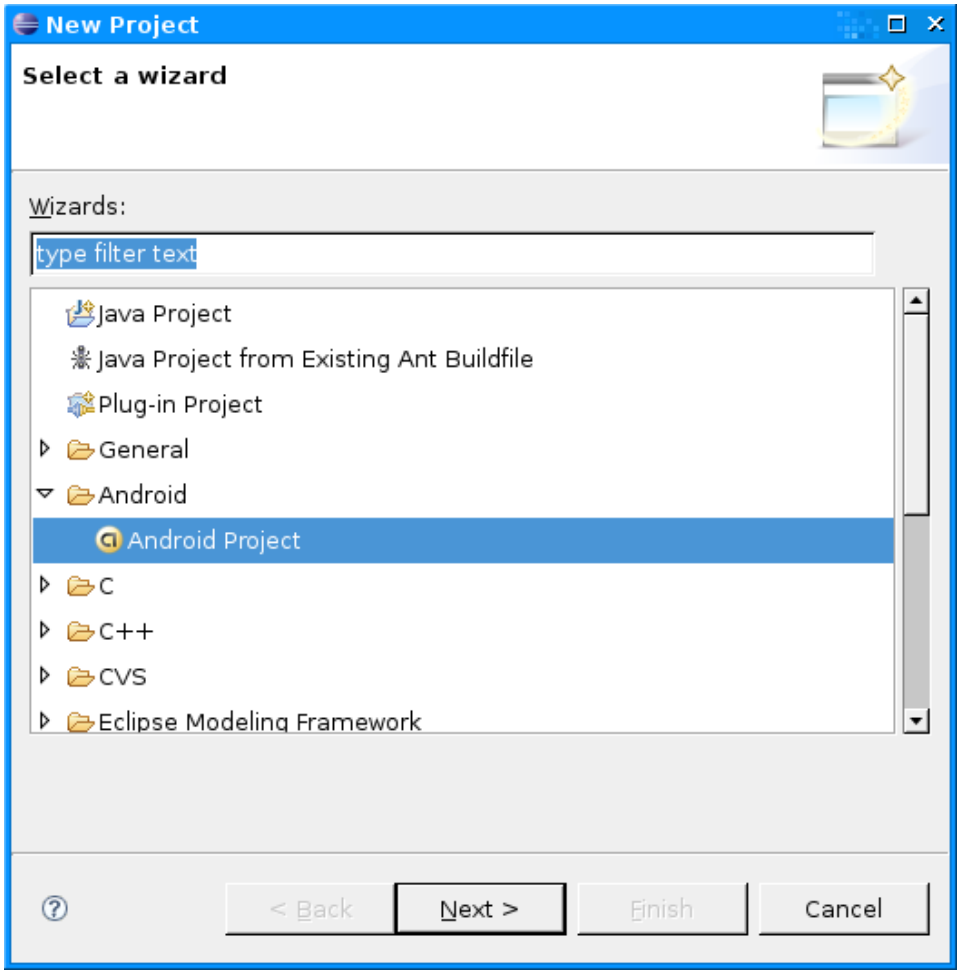
First, here's a high-level summary of how to build "Hello, World!":

1. Create a new "Android Project" via the **File > New > Project** menu.
2. Fill out the project details in the New Android Project dialog.
3. Edit the auto-generated source code template to display some output.

That's it! Next, let's go through each step above in detail.

1. **Create a new Android Project**

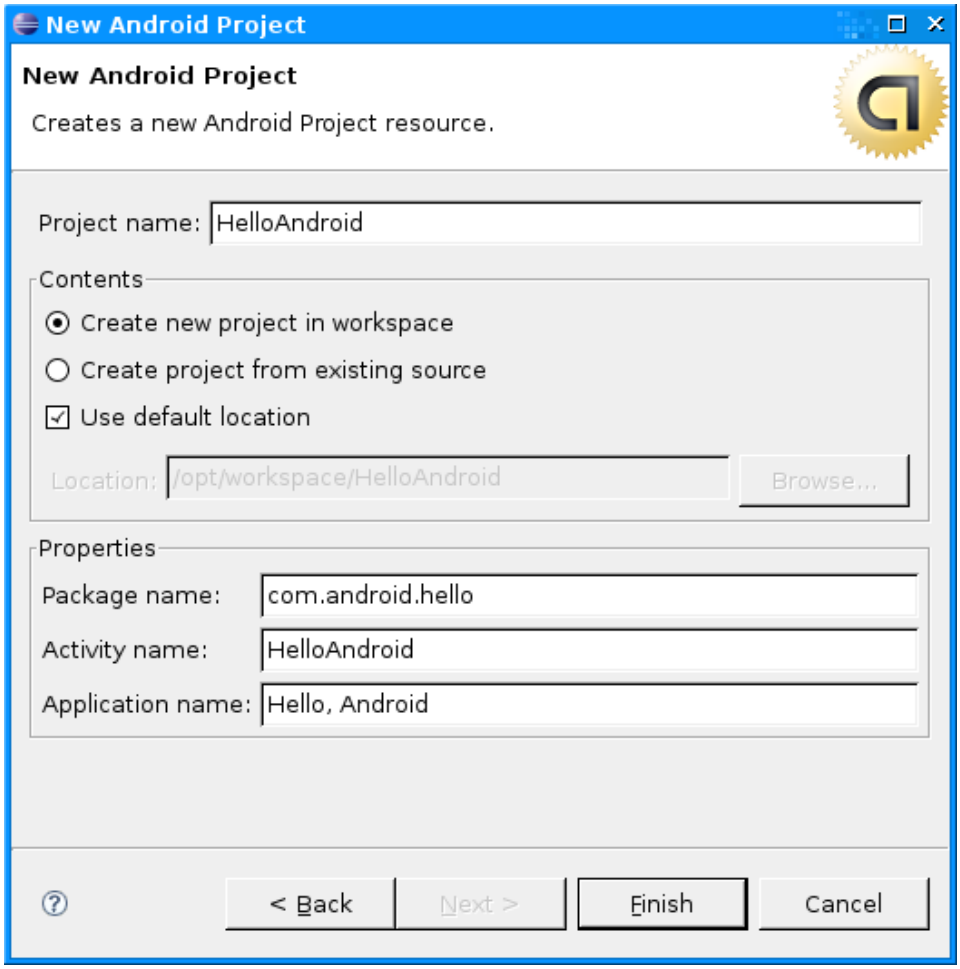
   From Eclipse, select the File > New > Project menu item. If the Android Plugin for Eclipse has been successfully installed, the resulting dialog should have a folder labeled "Android" which should contain a single entry: "Android Project".



Once you've selected "Android Project", click the Next button.

2. **Fill out the project details**

   The next screen allows you to enter the relevant details for your project. Here's an example:



Here's what each field on this screen means:

Project Name	This is the name of the directory or folder on your computer that you want to contain the project.
Package Name	This is the package namespace (following the same rules as for packages in the Java programming language) that you want all your source code to reside under. This also sets the package name under which the stub Activity will be generated. The package name you use in your application must be unique across all packages installed on the system; for this reason, it's very important to use a standard domain-style package for your applications. In the example above, we used the package domain "com.android"; you should use a different one appropriate to your organization.
Activity Name	This is the name for the class stub that will be generated by the plugin. This will be a subclass of Android's Activity class. An Activity is simply a class that can run and do work. It can create a UI if it chooses, but it doesn't need to.
Application Name	This is the human-readable title for your application.

The checkbox for toggling "Use default location" allows you to change the location on disk where the project's files will be generated and stored.

3. **Edit the auto-generated source code**

After the plugin runs, you'll have a class named HelloAndroid (found in your package, HelloAndroid > src > com.android.hello). It should look like this:

```
public class HelloAndroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
```

```
    }
}
```

Now, you *could* run this right away, but let's go a little further, so we understand more about what's happening. So, the next step is to modify some code!

Construct the UI

Take a look at this revised code, below, and make the same changes to your HelloAndroid.java file. We'll dissect it line by line:

```
package com.android.hello;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class HelloAndroid extends Activity {
   /** Called when the activity is first created. */
   @Override
   public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       TextView tv = new TextView(this);
       tv.setText("Hello, Android");
       setContentView(tv);
   }
}
```

Tip: If you forgot to import the TextView package, try this: press **Ctrl-Shift-O** (**Cmd-Shift-O**, on Mac). This is an Eclipse shortcut to organize imports—it identifies missing packages and adds them for you.

In Android, user interfaces are composed of hierarchies of classes called Views. A View is simply a drawable object, such as a radio button, an animation, or (in our case) a text label. The specific name for the View subclass that handles text is simply TextView.

Here's how you construct a TextView:

```
TextView tv = new TextView(this);
```

The argument to TextView's constructor is an Android Context instance. The Context is simply a handle to the system; it provides services like resolving resources, obtaining access to databases and preferences, and so on. The Activity class inherits from Context. Since our HelloAndroid class is a subclass of Activity, it is also a Context, and so we can pass the `this` reference to the TextView.

Once we've constructed the TextView, we need to tell it what to display:

```
tv.setText("Hello, Android");
```

Nothing too surprising there.

At this point, we've constructed a TextView and told it what text to display. The final step is to connect this TextView with the on-screen display, like so:

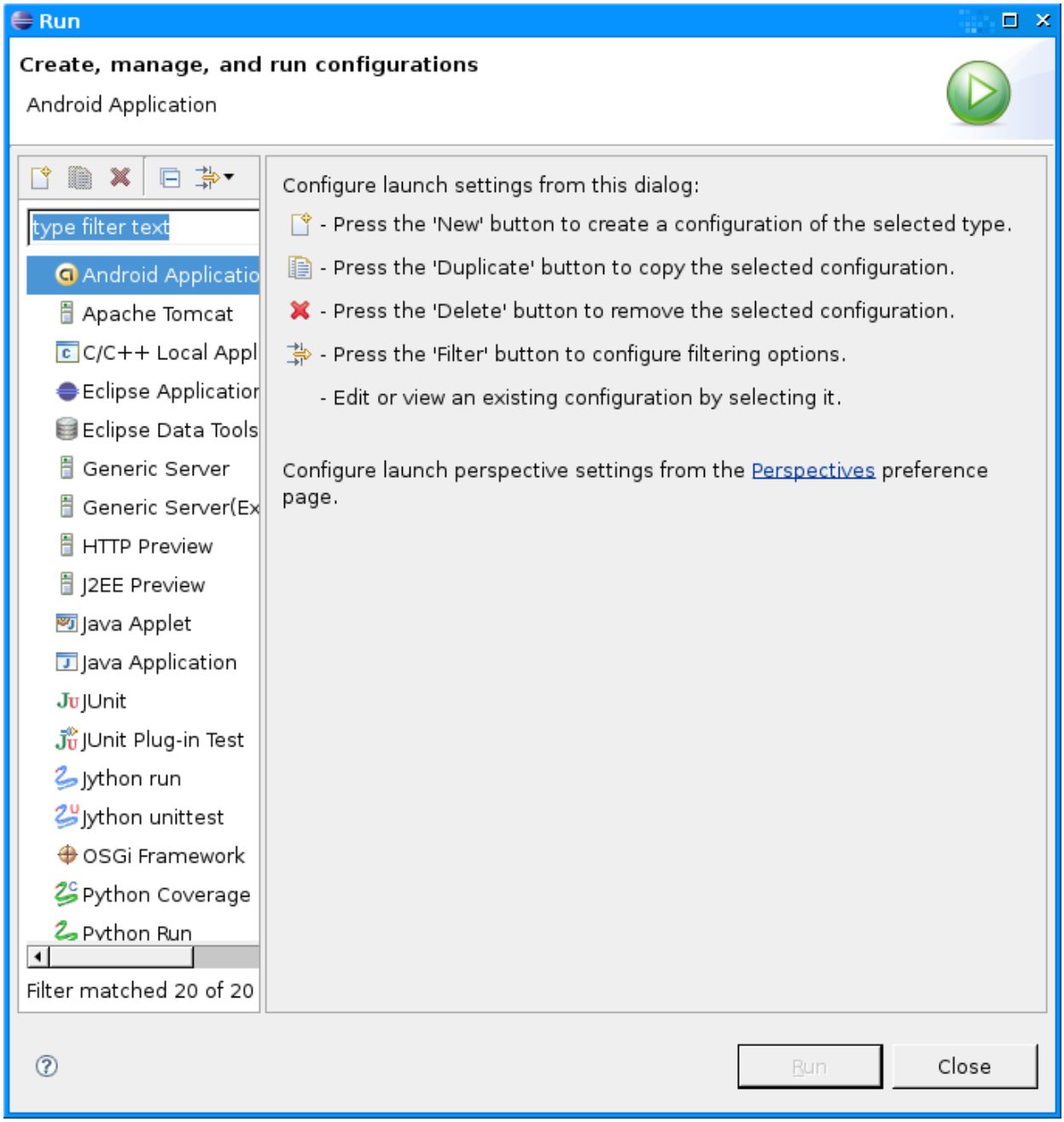
```
setContentView(tv);
```

The `setContentView()` method on Activity indicates to the system which View should be associated with the Activity's UI. If an Activity doesn't call this method, no UI is present at all and the system will display a blank screen. For our purposes, all we want is to display some text, so we pass it the TextView we just created.

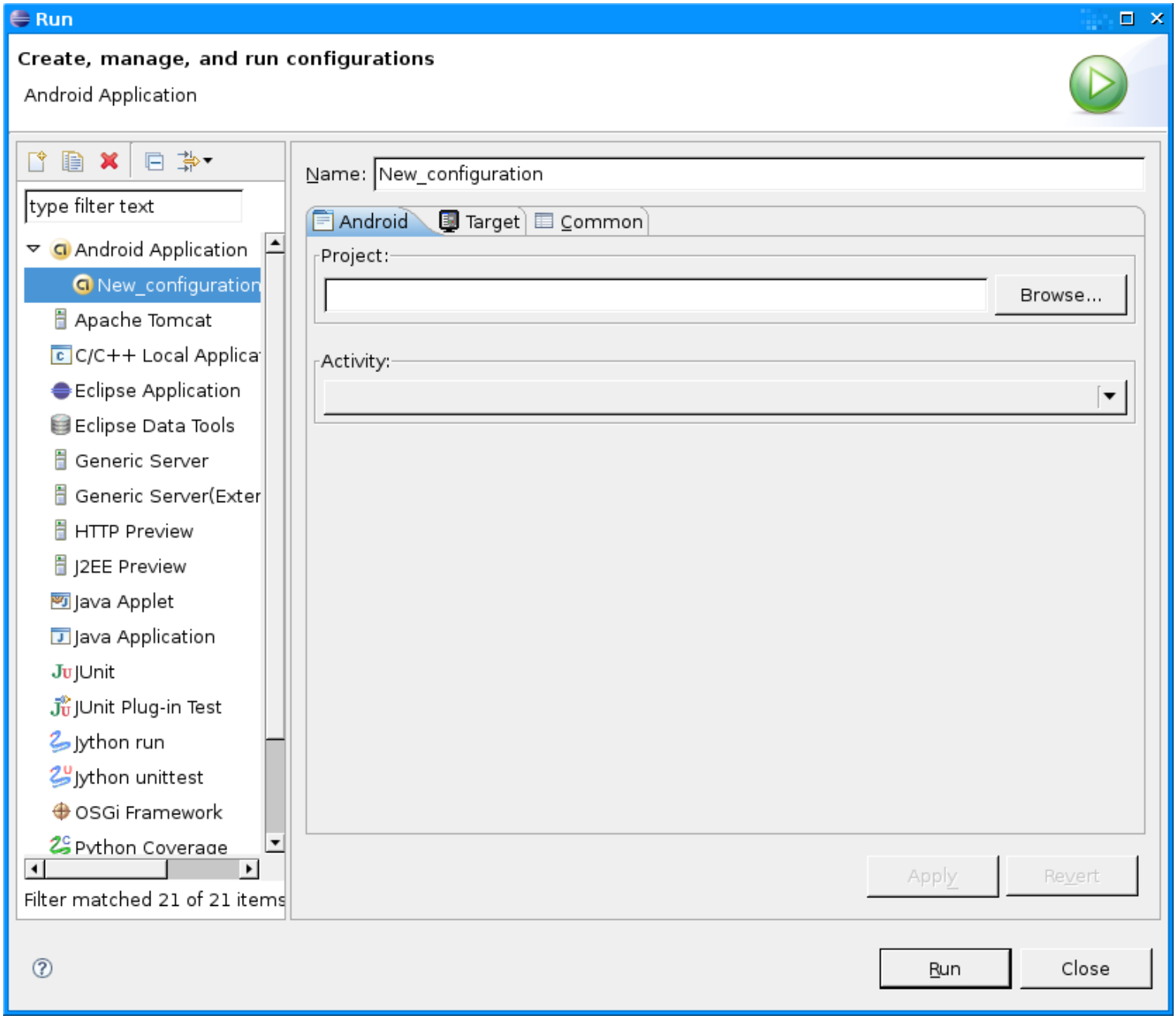
There it is — "Hello, World" in Android! The next step, of course, is to see it running.

Run the Code: Hello, Android

The Eclipse plugin makes it very easy to run your applications. Begin by selecting the **Run > Open Run Dialog** menu entry (in Eclipse 3.4, it's **Run > Run Configurations**). You should see a dialog like this:

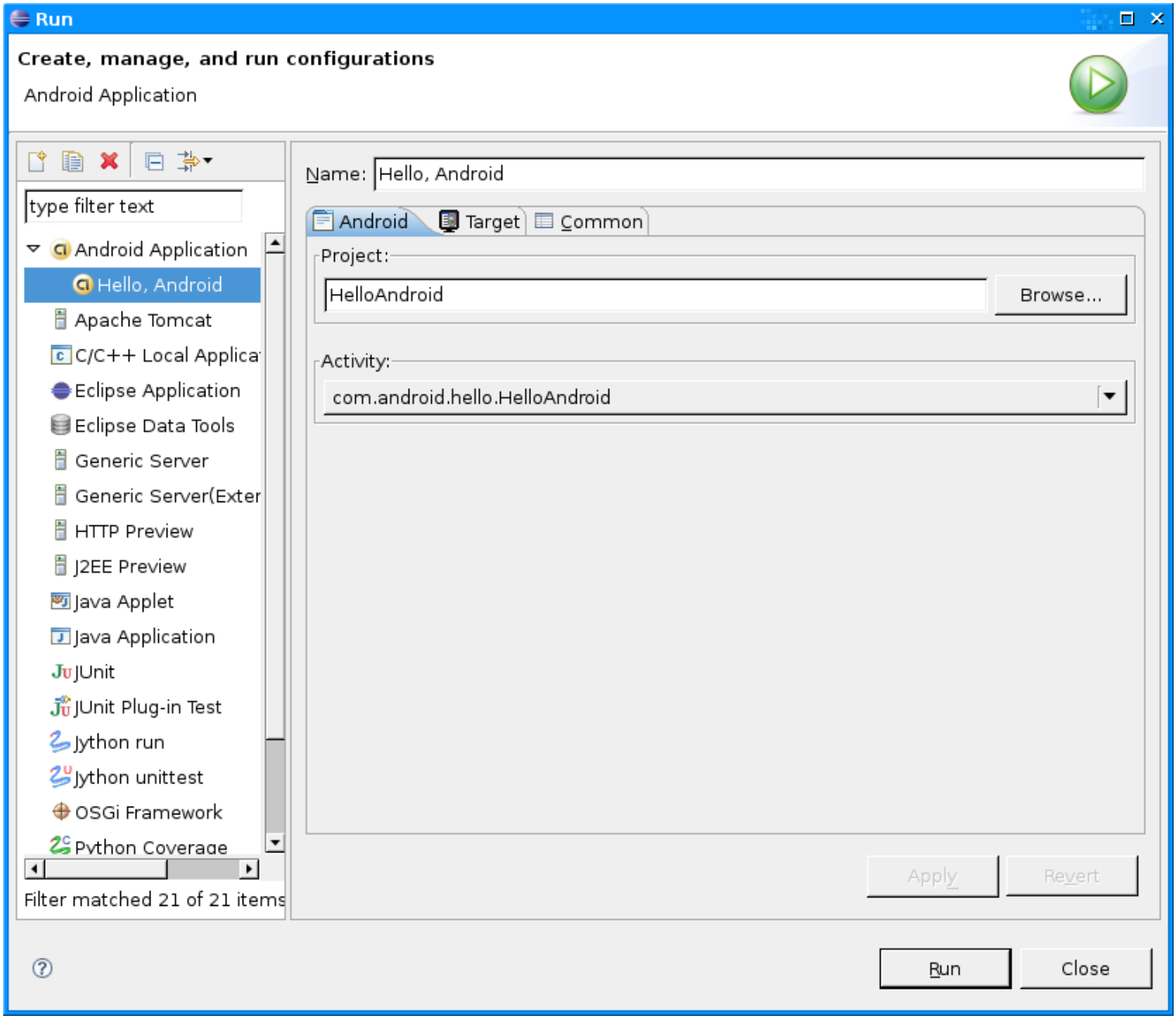


Next, highlight the "Android Application" entry, and then click the icon in the top left corner (the one depicting a sheet of paper with a plus sign in the corner) or simply double-click the "Android Application" entry. You should have a new launcher entry named "New_configuration".

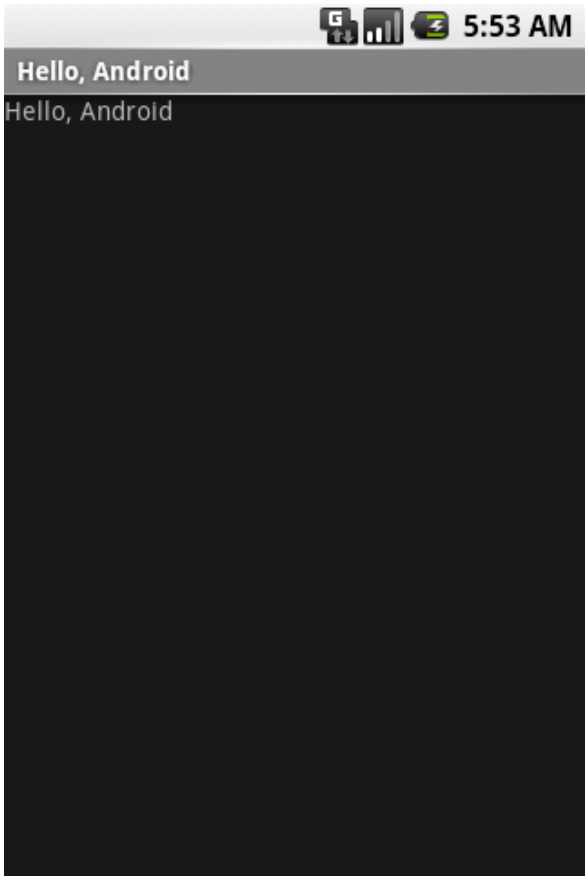


Change the name to something expressive, like "Hello, Android", and then pick your project by clicking the Browse button. (If you have more than one Android project open in Eclipse, be sure to pick the right one.) The plugin will automatically scan your project for Activity subclasses, and add each one it finds to the drop-down list under the "Activity:" label. Since your "Hello, Android" project only has one, it will be the default, and you can simply continue.

Click the "Apply" button. Here's an example:



That's it — you're done! Click the Run button, and the Android Emulator should start. Once it's booted up your application will appear. When all is said and done, you should see something like this:



That's "Hello, World" in Android. Pretty straightforward, eh? The next sections of the tutorial offer more detailed information that you may find valuable as you learn more about Android.

Upgrading the UI to an XML Layout

The "Hello, World" example you just completed uses what we call "programmatic" UI layout. This means that you construct and build your application's UI directly in source code. If you've done much UI programming, you're probably familiar with how brittle that approach can sometimes be: small changes in layout can result in big source-code headaches. It's also very easy to forget to properly connect Views together, which can result in errors in your layout and wasted time debugging your code.

That's why Android provides an alternate UI construction model: XML-based layout files. The easiest way to explain this concept is to show an example. Here's an XML layout file that is identical in behavior to the programmatically-constructed example you just completed:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:text="Hello, Android"/>
```

The general structure of an Android XML layout file is simple. It's a tree of tags, where each tag is the name of a View class. In this example, it's a very simple tree of one element, a TextView. You can use the name of any class that extends View as a tag name in your XML layouts, including custom View classes you define in your own code. This structure makes it very easy to quickly build up UIs, using a much simpler structure and syntax than you would in source code. This model is inspired by the web development model, where you can separate the presentation of your application (its UI) from the application logic used to fetch and fill in data.

In this example, there are also four XML attributes. Here's a summary of what they mean:

Attribute	Meaning
`xmlns:android`	This is an XML namespace declaration that tells the Android tools that you are going to refer to common attributes defined in the Android namespace. The outermost tag in every Android layout file must have this attribute.
`android:layout_width`	This attribute defines how much of the available width on the screen this View should consume. In this case,

	it's our only View so we want it to take up the entire screen, which is what a value of "fill_parent" means.
`android:layout_height`	This is just like android:layout_width, except that it refers to available screen height.
`android:text`	This sets the text that the TextView should contain. In this example, it's our usual "Hello, Android" message.

So, that's what the XML layout looks like, but where do you put it? Under the /res/layout directory in your project. The "res" is short for "resources" and that directory contains all the non-code assets that your application requires. This includes things like images, localized strings, and XML layout files.

The Eclipse plugin creates one of these XML files for you. In our example above, we simply never used it. In the Package Explorer, expand the folder /res/layout, and edit the file main.xml. Replace its contents with the text above and save your changes.

Now open the file named R.java in your source code folder in the Package Explorer. You'll see that it now looks something like this:

```
public final class R {
    public static final class attr {
    };
    public static final class drawable {
        public static final int icon=0x7f020000;
    };
    public static final class layout {
        public static final int main=0x7f030000;
    };
    public static final class string {
        public static final int app_name=0x7f040000;
    };
};
```

A project's R.java file is an index into all the resources defined in the file. You use this class in your source code as a sort of short-hand way to refer to resources you've included in your project. This is particularly powerful with the code-completion features of IDEs like Eclipse because it lets you quickly and interactively locate the specific reference you're looking for.

The important thing to notice for now is the inner class named "layout", and its member field "main". The Eclipse plugin noticed that you added a new XML layout file and then regenerated this R.java file. As you add other resources to your projects you'll see R.java change to keep up.

The last thing you need to do is modify your HelloAndroid source code to use the new XML version of your UI, instead of the hard-coded version. Here's what your new class will look like. As you can see, the source code becomes much simpler:

```
package com.android.hello;

import android.app.Activity;
import android.os.Bundle;

public class HelloAndroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

When you make this change, don't just copy-and-paste it in. Try out the code-completion feature on that R class. You'll probably find that it helps a lot.

Now that you've made this change, go ahead and re-run your application — all you need to do is click the green Run arrow icon, or select **Run > Run History > Hello, Android** from the menu. You should see.... well, exactly the same thing you saw before! After all, the point was to show that the two different layout approaches produce identical results.

There's a lot more to creating these XML layouts, but that's as far as we'll go here. Read the Implementing a User Interface documentation for more information on the power of this approach.

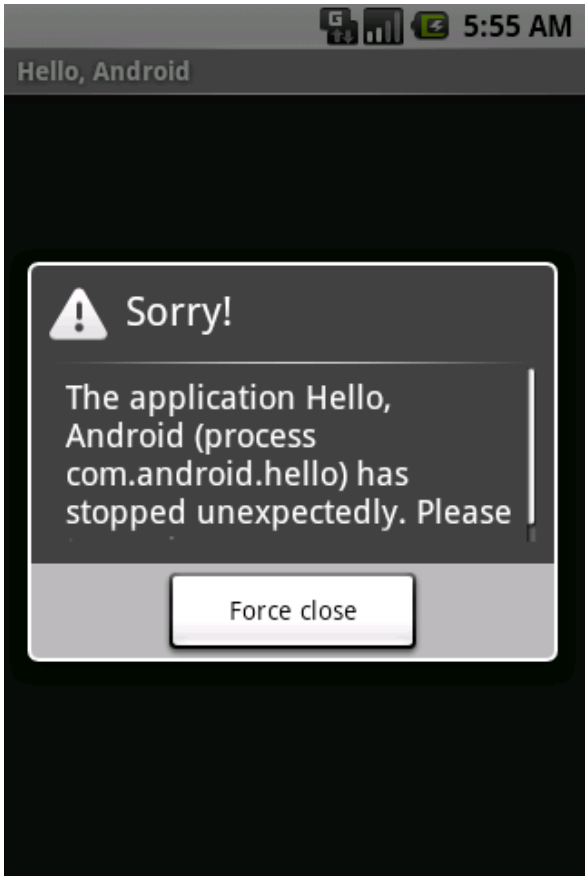
Debugging Your Project

The Android Plugin for Eclipse also has excellent integration with the Eclipse debugger. To demonstrate this, let's introduce a bug into our code. Change your HelloAndroid source code to look like this:

```
package com.android.hello;
```

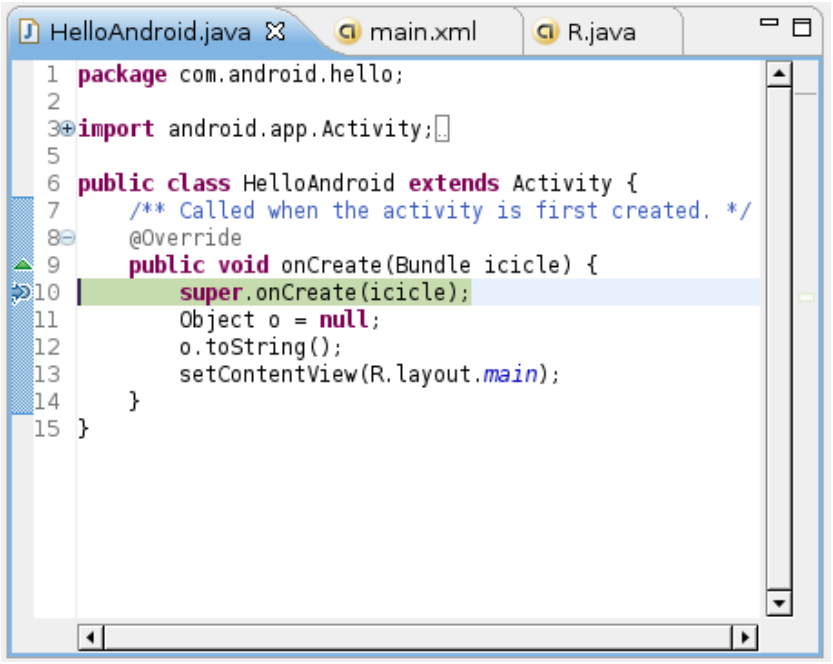
```
import android.app.Activity;
import android.os.Bundle;

public class HelloAndroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Object o = null;
        o.toString();
        setContentView(R.layout.main);
    }
}
```

This change simply introduces a NullPointerException into your code. If you run your application again, you'll eventually see this:



Press "Force Quit" to terminate the application and close the emulator window.

To find out more about the error, set a breakpoint in your source code on the line `Object o = null;` (double-click on the marker bar next to the source code line). Then select **Run > Debug History > Hello, Android** from the menu to enter debug mode. Your app will restart in the emulator, but this time it will suspend when it reaches the breakpoint you set. You can then step through the code in Eclipse's Debug Perspective, just as you would for any other application.



Creating the Project without Eclipse

If you don't use Eclipse (such as if you prefer another IDE, or simply use text editors and command line tools) then the Eclipse plugin can't help you. Don't worry though — you don't lose any functionality just because you don't use Eclipse.

The Android Plugin for Eclipse is really just a wrapper around a set of tools included with the Android SDK. (These tools, like the emulator, aapt, adb, ddms, and others are documented elsewhere.) Thus, it's possible to wrap those tools with another tool, such as an 'ant' build file.

The Android SDK includes a Python script named "activitycreator.py" that can be used to create all the source code and directory stubs for your project, as well as an ant-compatible build.xml file. This allows you to build your project from the command line, or integrate it with the IDE of your choice.

For example, to create a HelloAndroid project similar to the one we just created via Eclipse, you'd use this command:

```
activitycreator.py --out HelloAndroid com.android.hello.HelloAndroid
```

To build the project, you'd then run the command 'ant'. When that command successfully completes, you'll be left with a file named HelloAndroid.apk under the 'bin' directory. That .apk file is an Android Package, and can be installed and run in your emulator using the 'adb' tool.

For more information on how to use these tools, please read the documentation cited above.

Android

Anatomy of an Android Application

There are four building blocks to an Android application:

- Activity
- Broadcast Intent Receiver
- Service
- Content Provider

Not every application needs to have all four, but your application will be written with some combination of these.

Once you have decided what components you need for your application, you should list them in a file called AndroidManifest.xml. This is an XML file where you declare the components of your application and what their capabilities and requirements are. See the Android manifest file documentation for complete details.

Activity

Activities are the most common of the four Android building blocks. An activity is usually a single screen in your application. Each activity is implemented as a single class that extends the Activity base class. Your class will display a user interface composed of Views and respond to events. Most applications consist of multiple screens. For example, a text messaging application might have one screen that shows a list of contacts to send messages to, a second screen to write the message to the chosen contact, and other screens to review old messages or change settings. Each of these screens would be implemented as an activity. Moving to another screen is accomplished by a starting a new activity. In some cases an activity may return a value to the previous activity -- for example an activity that lets the user pick a photo would return the chosen photo to the caller.

When a new screen opens, the previous screen is paused and put onto a history stack. The user can navigate backward through previously opened screens in the history. Screens can also choose to be removed from the history stack when it would be inappropriate for them to remain. Android retains history stacks for each application launched from the home screen.

Intent and Intent Filters

Android uses a special class called an Intent to move from screen to screen. An intent describes what an application wants done. The two most important parts of the intent data structure are the action and the data to act upon. Typical values for action are MAIN (the front door of the application), VIEW, PICK, EDIT, etc. The data is expressed as a URI. For example, to view contact information for a person, you would create an intent with the VIEW action and the data set to a URI representing that person.

There is a related class called an IntentFilter. While an intent is effectively a request to do something, an intent filter is a description of what intents an activity (or BroadcastReceiver, see below) is capable of handling. An activity that is able to display contact information for a person would publish an IntentFilter that said that it knows how to handle the action VIEW when applied to data representing a person. Activities publish their IntentFilters in the AndroidManifest.xml file.

Navigating from screen to screen is accomplished by resolving intents. To navigate forward, an activity calls `startActivity(myIntent)`. The system then looks at the intent filters for all installed applications and picks the activity whose intent filters best matches `myIntent`. The new activity is informed of the intent, which causes it to be launched. The process of resolving intents happens at run time when startActivity is called, which offers two key benefits:

- Activities can reuse functionality from other components simply by making a request in the form of an Intent
- Activities can be replaced at any time by a new Activity with an equivalent IntentFilter

Broadcast Intent Receiver

You can use a BroadcastReceiver when you want code in your application to execute in reaction to an external event, for

example, when the phone rings, or when the data network is available, or when it's midnight. BroadcastReceivers do not display a UI, although they may use the NotificationManager to alert the user if something interesting has happened. BroadcastReceivers are registered in AndroidManifest.xml, but you can also register them from code using `Context.registerReceiver()`. Your application does not have to be running for its BroadcastReceivers to be called; the system will start your application, if necessary, when a BroadcastReceiver is triggered. Applications can also send their own intent broadcasts to others with `Context.sendBroadcast()`.

Service

A Service is code that is long-lived and runs without a UI. A good example of this is a media player playing songs from a play list. In a media player application, there would probably be one or more activities that allow the user to choose songs and start playing them. However, the music playback itself should not be handled by an activity because the user will expect the music to keep playing even after navigating to a new screen. In this case, the media player activity could start a service using `Context.startService()` to run in the background to keep the music going. The system will then keep the music playback service running until it has finished. (You can learn more about the priority given to services in the system by reading Life Cycle of an Android Application.) Note that you can connect to a service (and start it if it's not already running) with the `Context.bindService()` method. When connected to a service, you can communicate with it through an interface exposed by the service. For the music service, this might allow you to pause, rewind, etc.

Content Provider

Applications can store their data in files, an SQLite database, or any other mechanism that makes sense. A content provider, however, is useful if you want your application's data to be shared with other applications. A content provider is a class that implements a standard set of methods to let other applications store and retrieve the type of data that is handled by that content provider.

To get more details on content providers, see Accessing Content Providers.

 Build 110632-110632 - 22 Sep 2008 13:34

Android

Tutorial: A Notepad Application

The tutorial in this section gives you a "hands-on" introduction to the Android framework and the tools you use to build applications on it. Starting from a preconfigured project file, it guides you through the process of developing a simple notepad application and provides concrete examples of how to set up the project, develop the application logic and user interface, and then compile and run the application.

The tutorial presents the notepad application development as a set of exercises (see below), each consisting of several steps. You can follow along with the steps in each exercise and gradually build up and refine your application. The exercises explain each step in detail and provide all the sample code you need to complete the application.

When you are finished with the tutorial, you will have created a functioning Android application and learned in depth about many of the most important concepts in Android development. If you want to add more complex features to your application, you can examine the code in an alternative implementation of a notepad application, in the Sample Code documentation.

Who Should Use this Tutorial

This tutorial is designed for experienced developers, especially those with knowledge of the Java programming language. If you haven't written Java applications before, you can still use the tutorial, but you might need to work at a slower pace.

The tutorial assumes that you have some familiarity with the basic Android application concepts and terminology. If you aren't yet familiar with those, you should read Overview of an Android Application before continuing.

Also note that this tutorial uses the Eclipse development environment, with the Android plugin installed. If you are not using Eclipse, you can follow the exercises and build the application, but you will need to determine how to accomplish the Eclipse-specific steps in your environment.

Preparing for the Exercises

This tutorial builds on the information provided in the Installing the SDK and Hello Android documents, which explain in detail how to set up your development environment for building Android applications. Before you start this tutorial, you should read both these documents, have the SDK installed, and your work environment set up.

To prepare for this lesson:

1. Download the project exercises archive (.zip)
2. Unpack the archive file to a suitable location on your machine
3. Open the `NotepadCodeLab` folder

Inside the `NotepadCodeLab` folder, you should see six project files: `Notepadv1`, `Notepadv2`, `Notepadv3`, `Notepadv1Solution`, `Notepadv2Solution` and `Notepadv3Solution`. The `Notepadv#` projects are the starting points for each of the exercises, while the `Notepadv#Solution` projects are the exercise solutions. If you are having trouble with a particular exercise, you can compare your current work against the exercise solution.

Exercises

The table below lists the tutorial exercises and describes the development areas that each covers. Each exercise assumes that you have completed any previous exercises.

Exercise 1	Start here. Construct a simple notes list that lets the user add new notes but not edit them. Demonstrates the basics of `ListActivity` and creating and handling menu options. Uses a

	SQLite database to store the notes.
Exercise 2	Add a second Activity to the application. Demonstrates constructing a new Activity, adding it to the Android manifest, passing data between the activities, and using more advanced screen layout. Also shows how to invoke another Activity to return a result, using `startActivityForResult()`.
Exercise 3	Add handling of life-cycle events to the application, to let it maintain application state across the life cycle.
Extra Credit	Demonstrates how to use the Eclipse debugger and how you can use it to view life-cycle events as they are generated. This section is optional but highly recommended.

Other Resources and Further Learning

- For a lighter but broader introduction to concepts not covered in the tutorial, take a look at Common Android Tasks.
- The Android SDK includes a variety of fully functioning sample applications that make excellent opportunities for further learning. You can find the sample applications in the `samples/` directory of your downloaded SDK.
- This tutorial draws from the full Notepad application included in the `samples/` directory of the SDK, though it does not match it exactly. When you are done with the tutorial, it is highly recommended that you take a closer look at this version of the Notepad application, as it demonstrates a variety of interesting additions for your application, such as:
  - Setting up a custom striped list for the list of notes.
  - Creating a custom text edit view that overrides the `draw()` method to make it look like a lined notepad.
  - Implementing a full `ContentProvider` for notes.
  - Reverting and discarding edits instead of just automatically saving them.

Build 110632-110632 - 22 Sep 2008 13:34

Android

Development Tools

The Android SDK includes a variety of custom tools that help you develop mobile applications on the Android platform. The most important of these are the Android Emulator and the Android Development Tools plugin for Eclipse, but the SDK also includes a variety of other tools for debugging, packaging, and installing your applications on the emulator.

Android Emulator

A virtual mobile device that runs on your computer. You use the emulator to design, debug, and test your applications in an actual Android run-time environment.

Hierarchy Viewer New!

The Hierarchy Viewer tool allows you to debug and optimize your user interface. It provides a visual representation of your layout's hierarchy of Views and a magnified inspector of the current display with a pixel grid, so you can get your layout just right.

Draw 9-patch New!

The Draw 9-patch tool allows you to easily create a NinePatch graphic using a WYSIWYG editor. It also previews stretched versions of the image, and highlights the area in which content is allowed.

Android Development Tools Plugin for the Eclipse IDE

The ADT plugin adds powerful extensions to the Eclipse integrated environment, making creating and debugging your Android applications easier and faster. If you use Eclipse, the ADT plugin gives you an incredible boost in developing Android applications:

- It gives you access to other Android development tools from inside the Eclipse IDE. For example, ADT lets you access the many capabilities of the DDMS tool — taking screenshots, managing port-forwarding, setting breakpoints, and viewing thread and process information — directly from Eclipse.
- It provides a New Project Wizard, which helps you quickly create and set up all of the basic files you'll need for a new Android application.
- It automates and simplifies the process of building your Android application.
- It provides an Android code editor that helps you write valid XML for your Android manifest and resource files.

For more information about the ADT plugin, including installation instructions, see Installing the ADT Plugin for Eclipse. For a usage example with screenshots, see Hello Android.

Dalvik Debug Monitor Service (ddms)

Integrated with Dalvik, the Android platform's custom VM, this tool lets you manage processes on an emulator or device and assists in debugging. You can use it to kill processes, select a specific process to debug, generate trace data, view heap and thread information, take screenshots of the emulator or device, and more.

Android Debug Bridge (adb)

The adb tool lets you install your application's .apk files on an emulator or device and access the emulator or device from a command line. You can also use it to link a standard debugger to application code running on an Android emulator or device.

Android Asset Packaging Tool (aapt)

The aapt tool lets you create .apk files containing the binaries and resources of Android applications.

Android Interface Description Language (aidl)

Lets you generate code for an interprocess interface, such as what a service might use.

sqlite3

Included as a convenience, this tool lets you access the SQLite data files created and used by Android applications.

Traceview

This tool produces graphical analysis views of trace log data that you can generate from your Android application.

mksdcard

Helps you create a disk image that you can use with the emulator, to simulate the presence of an external storage card (such as an SD card).

dx

The dx tool rewrites .class bytecode into Android bytecode (stored in .dex files.)

UI/Application Exerciser Monkey

The Monkey is a program that runs on your emulator or device and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system- level events. You can use the Monkey to stress-test applications that you are developing, in a random yet repeatable manner.

activitycreator

A script that generates Ant build files that you can use to compile your Android applications. If you are developing on Eclipse with the ADT plugin, you won't need to use this script.

 Build 110632-110632 - 22 Sep 2008 13:34

Android

Android Application Model: Applications, Tasks, Processes, and Threads

In most operating systems, there is a strong 1-to-1 correlation between the executable image (such as the .exe on Windows) that an application lives in, the process it runs in, and the icon and application the user interacts with. In Android these associations are much more fluid, and it is important to understand how the various pieces can be put together.

Because of the flexible nature of Android applications, there is some basic terminology that needs to be understood when implementing the various pieces of an application:

- An **android package** (or **.apk** for short) is the file containing an application's code and its resources. This is the file that an application is distributed in and downloaded by the user when installing that application on their device.
- A **task** is generally what the user perceives as an "application" that can be launched: usually a task has an icon in the home screen through which it is accessed, and it is available as a top-level item that can be brought to the foreground in front of other tasks.
- A **process** is a low-level kernel process in which an application's code is running. Normally all of the code in a .apk is run in one, dedicated process for that .apk; however, the process tag can be used to modify where that code is run, either for the entire .apk or for individual activity, receiver, service, or provider, components.

Tasks

A key point here is: *when the user sees as an "application," what they are actually dealing with is a task*. If you just create a .apk with a number of activities, one of which is a top-level entry point (via an intent-filter for the action `android.intent.action.MAIN` and category `android.intent.category.LAUNCHER`), then there will indeed be one task created for your .apk, and any activities you start from there will also run as part of that task.

A task, then, from the user's perspective your application; and from the application developer's perspective it is one or more activities the user has traversed through in that task and not yet closed, or an activity stack. A new task is created by starting an activity Intent with the Intent.FLAG_ACTIVITY_NEW_TASK flag; this Intent will be used as the root Intent of the task, defining what task it is. Any activity started without this flag will run in the same task as the activity that is starting it (unless that activity has requested a special launch mode, as discussed later). Tasks can be re-ordered: if you use FLAG_ACTIVITY_NEW_TASK but there is already a task running for that Intent, the current task's activity stack will be brought to the foreground instead of starting a new task.

FLAG_ACTIVITY_NEW_TASK must only be used with care: using it says that, from the user's perspective, a new application starts at this point. If this is not the behavior you desire, you should not be creating a new task. In addition, you should only use the new task flag if it is possible for the user to navigate from home back to where they are and launch the same Intent as a new task. Otherwise, if the user presses HOME instead of BACK from the task you have launched, your task and its activities will be ordered behind the home screen without a way to return to them.

Task Affinities

In some cases Android needs to know which task an activity belongs to even when it is not being launched in to a specific task. This is accomplished through task affinities, which provide a unique static name for the task that one or more activities are intended to run in. The default task affinity for an activity is the name of the .apk package name the activity is implemented in. This provides the normally expected behavior, where all of the activities in a particular .apk are part of a single application to the user.

When starting a new activity without the Intent.FLAG_ACTIVITY_NEW_TASK flag, task affinities have no impact on the task the new activity will run in: it will always run in the task of the activity that is starting it. However, if the NEW_TASK flag is being used, then the affinity will be used to determine if a task already exists with the same affinity. If so, that task will be brought to the front and the new activity launched at the top of that task.

This behavior is most useful for situations where you must use the NEW_TASK flag, in particular launching activities from status bar notifications or home screen shortcuts. The result is that, when the user launches your application this way, its current task state will be brought to the foreground, and the activity they now want to look at placed on top of it.

You can assign your own task affinities in your manifest's application tag for all activities in the .apk, or the activity tag of individual activities. Some examples of how this can be used are:

- If your .apk contains multiple top-level applications that the user can launch, then you will probably want to assign different affinities to each of the activities that the users sees for your .apk. A good convention for coming up with distinct names is to append your .apk's package name with a colon separated string. For example, the "com.android.contacts" .apk may have the affinities "com.android.contacts:Dialer" and "com.android.contacts:ContactsList".
- If you are replacing a notification, shortcut, or other such "inner" activity of an application that can be launched from outside of it, you may need to explicitly set the taskAffinity of your replacement activity to be the same as the application you are replacing. For example, if you are replacing the contacts details view (which the user can make and invoke shortcuts to), you would want to set the taskAffinity to "com.android.contacts".

Launch Modes and Launch Flags

The main way you control how activities interact with tasks is through the activity's launchMode attribute and the flags associated with an Intent. These two parameters can work together in various ways to control the outcome of the activity launch, as described in their associated documentation. Here we will look at some common use cases and combinations of these parameters.

The most common launch mode you will use (besides the default `standard` mode) is `singleTop`. This does not have an impact on tasks; it just avoids starting the same activity multiple times on the top of a stack.

The `singleTask` launch mode has a major impact on tasks: it causes the activity to always be started in a new task (or its existing task to be brought to the foreground). Using this mode requires a lot of care in how you interact with the rest of the system, as it impacts every path in to the activity. It should only be used with activities that are front doors to the application (that is, which support the MAIN action and LAUNCHER category).

The `singleInstance` launch mode is even more specialized, and should only be used in applications that are implemented entirely as one activity.

A situation you will often run in to is when another entity (such as the SearchManager or NotificationManager) starts one of your activities. In this case, the Intent.FLAG_ACTIVITY_NEW_TASK flag must be used, because the activity is being started outside of a task (and the application/task may not even exist). As described previously, the standard behavior in this situation is to bring to the foreground the current task matching the new activity's affinity and start the new activity at the top of it. There are, however, other types of behavior that you can implement.

One common approach is to also use the Intent.FLAG_ACTIVITY_CLEAR_TOP flag in conjunction with NEW_TASK. By doing so, if your task is already running, then it will be brought to the foreground, all of the activities on its stack cleared except the root activity, and the root activity's onNewIntent(Intent) called with the Intent being started. Note that the activity often also use the `singleTop` or `singleTask` launch mode when using this approach, so that the current instance is given the new intent instead of requiring that it be destroyed and a new instance started.

Another approach you can take is to set the notification activity's task affinity to the empty string "" (indicating no affinity) and setting the finishOnBackground attribute. This approach is useful if you would like the notification to take the user to a separate activity describing it, rather than return to the application's task. By specifying this attribute, the activity will be finished whether the user leaves it with BACK or HOME; if the attribute isn't specified, pressing HOME will result in the activity and its task remaining in the system, possibly with no way to return to it.

Be sure to read the documentation on the launchMode attribute and the Intent flags for the details on these options.

Processes

In Android, processes are entirely an implementation detail of applications and not something the user is normally aware of. Their main uses are simply:

- Improving stability or security by putting untrusted or unstable code into another process.
- Reducing overhead by running the code of multiple .apks in the same process.

- Helping the system manage resources by putting heavy-weight code in a separate process that can be killed independently of other parts of the application.

As described previously, the process attribute is used to control the process that particular application components run in. Note that this attribute can not be used to violate security of the system: if two .apks that are not sharing the same user ID try to run in the same process, this will not be allowed and different distinct processes will be created for each of them.

See the security document for more information on these security restrictions.

Threads

Every process has one or more threads running in it. In most situations, Android avoids creating additional threads in a process, keeping an application single-threaded unless it creates its own threads. An important repercussion of this is that all calls to Activity, BroadcastReceiver, and Service instances are made only from the main thread of the process they are running in.

Note that a new thread is **not** created for each Activity, BroadcastReceiver, Service, or ContentProvider instance: these application components are instantiated in the desired process (all in the same process unless otherwise specified), in the main thread of that process. This means that none of these components (including services) should perform long or blocking operations (such as networking calls or computation loops) when called by the system, since this will block all other components in the process. You can use the standard library Thread class or Android's HandlerThread convenience class to perform long operations on another thread.

There are a few important exceptions to this threading rule:

- Calls on to an IBinder or interface implemented on an IBinder are dispatched from the thread calling them or a thread pool in the local process if coming from another process, *not* from the main thread of their process. In particular, calls on to the IBinder of a Service will be called this way. (Though calls to methods on Service itself are done from the main thread.) This means that *implementations of IBinder interfaces must always be written in a thread-safe way, since they can be called from any number of arbitrary threads at the same time*.

- Calls to the main methods of ContentProvider are dispatched from the calling thread or main thread as with IBinder. The specific methods are documented in the ContentProvider class. This means that *implementations of these methods must always be written in a thread-safe way, since they can be called from any number of arbitrary threads at the same time*.

- Calls on View and its subclasses are made from the thread that the view's window is running in. Normally this will be the main thread of the process, however if you create a thread and show a window from there then the window's view hierarchy will be called from that thread.

 Build 110632-110632 - 22 Sep 2008 13:34

Android

Life Cycle of an Android Application

In most cases, every Android application runs in its own Linux process. This process is created for the application when some of its code needs to be run, and will remain running until it is no longer needed *and* the system needs to reclaim its memory for use by other applications.

An unusual and fundamental feature of Android is that an application process's lifetime is *not* directly controlled by the application itself. Instead, it is determined by the system through a combination of the parts of the application that the system knows are running, how important these things are to the user, and how much overall memory is available in the system.

It is important that application developers understand how different application components (in particular Activity, Service, and BroadcastReceiver) impact the lifetime of the application's process. **Not using these components correctly can result in the system killing the application's process while it is doing important work.**

A common example of a process life-cycle bug is a BroadcastReceiver that starts a thread when it receives an Intent in its BroadcastReceiver.onReceive() method, and then returns from the function. Once it returns, the system considers the BroadcastReceiver to be no longer active, and thus, its hosting process no longer needed (unless other application components are active in it). So, the system may kill the process at any time to reclaim memory, and in doing so, it terminates the spawned thread running in the process. The solution to this problem is to start a Service from the BroadcastReceiver, so the system knows that there is still active work being done in the process.

To determine which processes should be killed when low on memory, Android places each process into an "importance hierarchy" based on the components running in them and the state of those components. These process types are (in order of importance):

1. A **foreground process** is one that is required for what the user is currently doing. Various application components can cause its containing process to be considered foreground in different ways. A process is considered to be in the foreground if any of the following conditions hold:
    - It is running an Activity at the top of the screen that the user is interacting with (its onResume() method has been called).
    - It has a BroadcastReceiver that is currently running (its BroadcastReceiver.onReceive() method is executing).
    - It has a Service that is currently executing code in one of its callbacks (Service.onCreate(), Service.onStart(), or Service.onDestroy()).

    There will only ever be a few such processes in the system, and these will only be killed as a last resort if memory is so low that not even these processes can continue to run. Generally, at this point, the device has reached a memory paging state, so this action is required in order to keep the user interface responsive.
2. A **visible process** is one holding an Activity that is visible to the user on-screen but not in the foreground (its onPause() method has been called). This may occur, for example, if the foreground Activity is displayed as a dialog that allows the previous Activity to be seen behind it. Such a process is considered extremely important and will not be killed unless doing so is required to keep all foreground processes running.
3. A **service process** is one holding a Service that has been started with the startService() method. Though these processes are not directly visible to the user, they are generally doing things that the user cares about (such as background mp3 playback or background network data upload or download), so the system will always keep such processes running unless there is not enough memory to retain all foreground and visible process.
4. A **background process** is one holding an Activity that is not currently visible to the user (its onStop() method has been called). These processes have no direct impact on the user experience. Provided they implement their Activity life-cycle correctly (see Activity for more details), the system can kill such processes at any time to reclaim memory for one of the three previous processes types. Usually there are many of these processes running, so they are kept in an LRU list to ensure the process that was most recently seen by the user is the last to be killed when running low on memory.
5. An **empty process** is one that doesn't hold any active application components. The only reason to keep such a process around is as a cache to improve startup time the next time a component of its application needs to run. As such, the system will often kill these processes in order to balance overall system resources between these empty cached processes and the underlying kernel caches.

When deciding how to classify a process, the system will base its decision on the most important level found among all the components currently active in the process. See the Activity, Service, and BroadcastReceiver documentation for more detail

on how each of these components contribute to the overall life-cycle of a process. The documentation for each of these classes describes in more detail how they impact the overall life-cycle of their application.

A process's priority may also be increased based on other dependencies a process has to it. For example, if process A has bound to a Service with the Context.BIND_AUTO_CREATE flag or is using a ContentProvider in process B, then process B's classification will always be at least as important as process A's.

Build 110632-110632 - 22 Sep 2008 13:34

Android

Upgrading the SDK

This guide will help you migrate your development environment and applications to the latest version of the SDK. Use this guide if you've been developing applications on a previous version of the Android SDK.

To ensure that your applications are compliant with the Android 1.0 system available on mobile devices, you need to install the new SDK and port your existing Android applications to the updated API. The sections below guide you through the process.

Useful Links

- Overview of Changes

  A high-level look at what's changed in Android, with discussion of how the changes may affect your apps.
- API Diff Report

  A detailed report that lists all the specific changes in the latest SDK.
- Release Notes

  Version details, known issues, and resolved issues.
- Android Developers Group

  A forum where you can discuss migration issues and learn from other Android developers.
- Android Issue Tracker

  If you think you may have found a bug, use the issue tracker to report it.

Install the new SDK

Download the SDK and unpack it into a safe location.

After unpacking the new SDK, you should:

- Wipe your emulator data.

  Some data formats have changed since the last SDK release, so any previously saved data in your emulator must be removed. Open a console/terminal and navigate to the `/tools` directory of your SDK. Launch the emulator with the `-wipe-data` option.

  Windows: `emulator -wipe-data`
  Mac/Linux: `./emulator -wipe-data`
- Update your PATH variable (Mac/Linux; optional).

  If you had previously setup your PATH variable to point to the SDK tools directory, then you'll need to update it to point to the new SDK. E.g., for a `.bashrc` or `.bash_profile` file: `export PATH=$PATH:<your_new_sdk_dir>/tools`

Update your ADT Eclipse Plugin

If you develop on Eclipse and are using the ADT plugin, follow these steps to install the new plugin that accompanies the latest SDK.

Eclipse 3.3 (Europa)	Eclipse 3.4 (Ganymede)
1. Select **Help** > **Software Updates** > **Find and Install...**. 2. Select **Search for updates of the currently installed features** and click **Finish**. 3. If any update for ADT is available, select and install. 4. Restart Eclipse.	1. Select **Help** > **Software Updates...** 2. Select the **Installed Software** tab. 3. Click **Update...** 4. If an update for ADT is available, select it and click **Finish**. 5. Restart Eclipse.

After restart, update your Eclipse preferences to point to the SDK directory:

1. Select **Window** > **Preferences...** to open the Preferences panel. (Mac OSX: **Eclipse** > **Preferences**)
2. Select **Android** from the left panel.
3. For the SDK Location in the main panel, click **Browse...** and locate the SDK directory.
4. Click **Apply**, then **OK**.

Set Up Application Signing

All applications must now be signed before you can install them on the emulator. Both the ADT plugin and the Ant-based build tools support this requirement by signing compiled .apk files with a debug key. To do so, the build tools use the Keytool utility included in the JDK to to create a keystore and a key with a known alias and password. For more information, see Signing Your Applications.

To support signing, you should first make sure that Keytool is available to the SDK build tools. In most cases, you can tell the SDK build tools how to find Keytool by making sure that your JAVA_HOME environment variable is set and that it references a suitable JDK. Alternatively, you can add the JDK version of Keytool to your PATH variable.

If you are developing on a version of Linux that originally came with Gnu Compiler for Java, make sure that the system is using the JDK version of Keytool, rather than the gcj version. If keytool is already in your PATH, it might be pointing to a symlink at /usr/bin/keytool. In this case, check the symlink target to make sure that it points to the keytool in the JDK.

If you use Ant to build your .apk files (rather than ADT for Eclipse), you must regenerate your build.xml file. To do that, follow these steps:

1. In your Android application project directory, locate and delete the current build.xml file.
2. Run activitycreator, directing output to the folder containing your application project.

```
- exec activitycreator --out <project folder> your.activity.YourActivity
```

Run in this way, activityCreator will not erase or create new Java files (or manifest files), provided the activity and package already exists. It is important that the package and the activity are real. The tool creates a new build.xml file, as well as a new directory called "libs" in which to place 3rd jar files, which are now automatically handled by the Ant script.

Migrate your applications

After updating your SDK, you will likely encounter breakages in your code, due to framework and API changes. You'll need to update your code to match changes in the Andriod APIs.

One way to start is to open your project in Eclipse and see where the ADT identifies errors in your application. From there, you can lookup respective changes in the Overview of Changes and API Diffs Report.

If you have additional trouble updating your code, visit the Android Discussion Groups to seek help from other Android developers.

If you have modified one of the ApiDemos applications and would like to migrate it to the new SDK, note that you will need to uninstall the version of ApiDemos that comes preinstalled in the emulator. For more information, or if you encounter an "reinstallation" error when running or installing ApiDemos, see the troubleshooting topic I can't install ApiDemos apps in my IDE because of a signing error for information about how to solve the problem.

Build 110632-110632 - 22 Sep 2008 13:34

Android

Tutorial: Notepad Exercise 1

In this exercise, you will construct a simple notes list that lets the user add new notes but not edit them. The exercise demonstrates:

- *The basics of `ListActivities` and creating and handling menu options.*
- *How to use a SQLite database to store the notes.*
- *How to bind data from a database cursor into a ListView using a SimpleCursorAdapter.*
- *The basics of screen layouts, including how to lay out a list view, how you can add items to the activity menu, and how the activity handles those menu selections.*

[Exercise 1] [Exercise 2] [Exercise 3] [Extra Credit]

Step 1

Open up the `Notepadv1` project in Eclipse.

`Notepadv1` is a project that is provided as a starting point. It takes care of some of the boilerplate work that you have already seen if you followed the Hello Android tutorial.

1. Start a new Android Project by clicking **File** > **New** > **Android Project**.
2. In the New Android Project dialog, select **Create project from existing source**.
3. Click **Browse** and navigate to where you copied the `NotepadCodeLab` (downloaded during setup). Select `Notepadv1` and click **Choose**.
4. You should see `Notepadv1` in the *Project name* and also see the *Location* filled in with the path you selected.
5. Click **Finish**. The `Notepadv1` project should open and be visible in your Eclipse package explorer.

If you see an error about `AndroidManifest.xml`, or some problems related to an Android zip file, right click on the project and select **Android Tools** > **Fix Project Properties**. (The project is looking in the wrong location for the library file, this will fix it for you.)

Step 2

Take a look at the `NotesDbAdapter` class — this class is provided to encapsulate data access to a SQLite database that will hold our notes data and allow us to update it.

At the top of the class are some constant definitions that will be used in the application to look up data from the proper field names in the database. There is also a database creation string defined, which is used to create a new database schema if one doesn't exist already.

Our database will have the name `data`, and have a single table called `notes`, which in turn has three fields: `_id`, `title` and `body`. The `_id` is named with an underscore convention used in a number of places inside the Android SDK and helps keep a track of state. The `_id` usually has to be specified when querying or updating the database (in the column projections and so on). The other two fields are simple text fields that will store data.

Accessing and modifying data

For this exercise, we are using a SQLite database to store our data. This is useful if only *your* application will need to access or modify the data. If you wish for other activities to access or modify the data, you have to expose the data using a ContentProvider.

If you are interested, you can find out more about content providers or the whole subject of Storing, Retrieving, and Exposing Data. The NotePad sample in the `samples/` folder of the SDK also has an example of how to create a ContentProvider.

The constructor for `NotesDbAdapter` takes a Context, which allows it to communicate with aspects of the Android operating system. This is quite common for classes that need to touch the Android system in some way. The Activity class implements the Context class, so usually you will just pass `this` from your Activity, when needing a Context.

The `open()` method calls up an instance of DatabaseHelper, which is our local implementation of the SQLiteOpenHelper class.

It calls `getWritableDatabase()`, which handles creating/opening a database for us.

`close()` just closes the database, releasing resources related to the connection.

`createNote()` takes strings for the title and body of a new note, then creates that note in the database. Assuming the new note is created successfully, the method also returns the row `_id` value for the newly created note.

`deleteNote()` takes a *rowId* for a particular note, and deletes that note from the database.

`fetchAllNotes()` issues a query to return a Cursor over all notes in the database. The `query()` call is worth examination and understanding. The first field is the name of the database table to query (in this case `DATABASE_TABLE` is "notes"). The next is the list of columns we want returned, in this case we want the `_id`, `title` and `body` columns so these are specified in the String array. The remaining fields are, in order: `selection`, `selectionArgs`, `groupBy`, `having` and `orderBy`. Having these all `null` means we want all data, need no grouping, and will take the default order. See SQLiteDatabase for more details.

Note: A Cursor is returned rather than a collection of rows. This allows Android to use resources efficiently -- instead of putting lots of data straight into memory the cursor will retrieve and release data as it is needed, which is much more efficient for tables with lots of rows.

`fetchNote()` is similar to `fetchAllNotes()` but just gets one note with the *rowId* we specify. It uses a slightly different version of the SQLiteDatabase `query()` method. The first parameter (set *true*) indicates that we are interested in one distinct result. The *selection* parameter (the fourth parameter) has been specified to search only for the row "where _id =" the *rowId* we passed in. So we are returned a Cursor on the one row.

And finally, `updateNote()` takes a *rowId*, *title* and *body*, and uses a ContentValues instance to update the note of the given *rowId*.

Step 3

Open the `notepad_list.xml` file in `res/layout` and take a look at it. (You may have to hit the *xml* tab, at the bottom, in order to view the XML markup.)

This is a mostly-empty layout definition file. Here are some things you should know about a layout file:

- All Android layout files must start with the XML header line: `<?xml version="1.0" encoding="utf-8"?>`.
- The next definition will often (but not always) be a layout definition of some kind, in this case a `LinearLayout`.
- The XML namespace of Android should always be defined in the top level component or layout in the XML so that `android:` tags can be used through the rest of the file:

  `xmlns:android="http://schemas.android.com/apk/res/android"`

Layouts and activities

Most Activity classes will have a layout associated with them. The layout will be the "face" of the Activity to the user. In this case our layout will take over the whole screen and provide a list of notes.

Full screen layouts are not the only option for an Activity however. You might also want to use a floating layout (for example, a dialog or alert), or perhaps you don't need a layout at all (the Activity will be invisible to the user unless you specify some kind of layout for it to use).

Step 4

We need to create the layout to hold our list. Add code inside of the `LinearLayout` element so the whole file looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">

  <ListView android:id="@android:id/list"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
  <TextView android:id="@android:id/empty"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/no_notes"/>
```

```
</LinearLayout>
```

- The @ symbol in the id strings of the `ListView` and `TextView` tags means that the XML parser should parse and expand the rest of the id string and use an ID resource.
- The `ListView` and `TextView` can be thought as two alternative views, only one of which will be displayed at once. ListView will be used when there are notes to be shown, while the TextView (which has a default value of "No Notes Yet!" defined as a string resource in `res/values/strings.xml`) will be displayed if there aren't any notes to display.
- The `list` and `empty` IDs are provided for us by the Android platform, so, we must prefix the `id` with `android:` (e.g., `@android:id/list`).
- The View with the `empty` id is used automatically when the ListAdapter has no data for the ListView. The ListAdapter knows to look for this name by default. Alternatively, you could change the default empty view by using setEmptyView(View) on the ListView.

  More broadly, the `android.R` class is a set of predefined resources provided for you by the platform, while your project's `R` class is the set of resources your project has defined. Resources found in the `android.R` resource class can be used in the XML files by using the `android:` name space prefix (as we see here).

Step 5

To make the list of notes in the ListView, we also need to define a View for each row:

1. Create a new file under `res/layout` called `notes_row.xml`.
2. Add the following contents (note: again the XML header is used, and the first node defines the Android XML namespace)

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <TextView android:id="@+id/text1"
       xmlns:android="http://schemas.android.com/apk/res/android"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"/>
   ```

   This is the View that will be used for each notes title row — it has only one text field in it.

   In this case we create a new id called `text1`. The **+** after the @ in the id string indicates that the id should be automatically created as a resource if it does not already exist, so we are defining `text1` on the fly and then using it.
3. Save the file.

Resources and the R class

The folders under res/ in the Eclipse project are for resources. There is a specific structure to the folders and files under res/.

Resources defined in these folders and files will have corresponding entries in the R class allowing them to be easily accessed and used from your application. The R class is automatically generated using the contents of the res/ folder by the eclipse plugin (or by aapt if you use the command line tools). Furthermore, they will be bundled and deployed for you as part of the application.

Open the `R.java` class in the project and look at it, you should see new definitions for `notes_row` and `text1` (our new definitions) meaning we can now gain access to these from the our code.

Step 6

Next, open the `Notepadv1` class in the source. In the following steps, we are going to alter this class to become a list adapter and display our notes, and also allow us to add new notes.

`Notepadv1` will inherit from a subclass of `Activity` called a `ListActivity`, which has extra functionality to accommodate the kinds of things you might want to do with a list, for example: displaying an arbitrary number of list items in rows on the screen, moving through the list items, and allowing them to be selected.

Take a look through the existing code in `Notepadv1` class. There is a currently an unused private field called `mNoteNumber` that we will use to create numbered note titles.

There are also three override methods defined: `onCreate`, `onCreateOptionsMenu` and `onOptionsItemSelected`; we need to fill these out:

- `onCreate()` is called when the activity is started — it is a little like the "main" method for an Activity. We use this to set up resources and state for the activity when it is running.

- `onCreateOptionsMenu()` is used to populate the menu for the Activity. This is shown when the user hits the menu button, and has a list of options they can select (like "Create Note").
- `onOptionsItemSelected()` is the other half of the menu equation, it is used to handle events generated from the menu (e.g., when the user selects the "Create Note" item).

Step 7

Change the inheritance of `Notepadv1` from `Activity` to `ListActivity`:

```
public class Notepadv1 extends ListActivity
```

Note: you will have to import `ListActivity` into the Notepadv1 class using Eclipse, **ctrl-shift-O** on Windows or Linux, or **cmd-shift-O** on the Mac (organize imports) will do this for you after you've written the above change.

Step 8

Fill out the body of the `onCreate()` method.

Here we will set the title for the Activity (shown at the top of the screen), use the `notepad_list` layout we created in XML, set up the `NotesDbAdapter` instance that will access notes data, and populate the list with the available note titles:

1. In the `onCreate` method, call `super()` with the `savedInstanceState` parameter that's passed in.
2. Call `setContentView()` and pass `R.layout.notepad_list`.
3. At the top of the class, create a new private class field called `mDbHelper` of class `NotesDbAdapter`.
4. Back in the `onCreate` method, construct a new `NotesDbAdapter` instance and assign it to the `mDbHelper` field (pass `this` into the constructor for `DBHelper`)
5. Call the `open()` method on `mDbHelper` to open (or create) the database.
6. Finally, call a new method `fillData()`, which will get the data and populate the ListView using the helper — we haven't defined this method yet.

`onCreate()` should now look like this:

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.notepad_list);
        mDbHelper = new NotesDbAdapter(this);
        mDbHelper.open();
        fillData();
    }
```

And be sure you have the `mDbHelper` field definition (right under the mNoteNumber definition):

```
    private NotesDbAdapter mDbHelper;
```

Step 9

Fill out the body of the `onCreateOptionsMenu()` method.

We will now create the "Add Item" button that can be accessed by pressing the menu button on the device. We'll specify that it occupy the first position in the menu.

1. In `strings.xml` resource (under `res/values`), add a new string named "menu_insert" with its value set to `Add Item`:

```
<string name="menu_insert">Add Item</string>
```

More on menus

The notepad application we are constructing only scratches the surface with menus.

You can also add shortcut keys for menu items, create submenus and even add menu items to other applications!.

Then save the file and return to `Notepadv1`.

2. Create a menu position constant at the top of the class:

```
public static final int INSERT_ID = Menu.FIRST;
```

3. In the `onCreateOptionsMenu()` method, change the `super` call so we capture the boolean return as `result`. We'll return this value at the end.
4. Then add the menu item with `menu.add()`.

The whole method should now look like this:

```
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        boolean result = super.onCreateOptionsMenu(menu);
        menu.add(0, INSERT_ID, 0, R.string.menu_insert);
        return result;
    }
```

The arguments passed to `add()` indicate: a group identifier for this menu (none, in this case), a unique ID (defined above), the order of the item (zero indicates no preference), and the resource of the string to use for the item.

Step 10

Fill out the body of the `onOptionsItemSelected()` method:

This is going to handle our new "Add Note" menu item. When this is selected, the `onOptionsItemSelected()` method will be called with the `item.getId()` set to `INSERT_ID` (the constant we used to identify the menu item). We can detect this, and take the appropriate actions:

1. The `super.onOptionsItemSelected(item)` method call goes at the end of this method — we want to catch our events first!
2. Write a switch statement on `item.getItemId()`.

   In the case of *INSERT_ID*, call a new method, `createNote()`, and return true, because we have handled this event and do not want to propagate it through the system.

3. Return the result of the superclass' `onOptionsItemSelected()` method at the end.

The whole `onOptionsItemSelect()` method should now look like this:

```
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
        case INSERT_ID:
            createNote();
            return true;
        }

        return super.onOptionsItemSelected(item);
    }
```

Step 11

Add a new `createNote()` method:

In this first version of our application, `createNote()` is not going to be very useful. We will simply create a new note with a title assigned to it based on a counter ("Note 1", "Note 2"...) and with an empty body. At present we have no way of editing the contents of a note, so for now we will have to be content making one with some default values:

1. Construct the name using "Note" and the counter we defined in the class: `String noteName = "Note " +`

```
mNoteNumber++
```

2. Call `mDbHelper.createNote()` using `noteName` as the title and `""` for the body
3. Call `fillData()` to populate the list of notes (inefficient but simple) — we'll create this method next.

The whole `createNote()` method should look like this:

```
    private void createNote() {
        String noteName = "Note " + mNoteNumber++;
        mDbHelper.createNote(noteName, "");
        fillData();
    }
```

Step 12

Define the `fillData()` method:

This method uses `SimpleCursorAdapter,` which takes a database `Cursor` and binds it to fields provided in the layout. These fields define the row elements of our list (in this case we use the `text1` field in our `notes_row.xml` layout), so this allows us to easily populate the list with entries from our database.

To do this we have to provide a mapping from the `title` field in the returned Cursor, to our `text1` TextView, which is done by defining two arrays: the first a string array with the list of columns to map *from* (just "title" in this case, from the constant `NotesDbAdapter.KEY_TITLE`) and, the second, an int array containing references to the views that we'll bind the data *into* (the `R.id.text1` TextView).

List adapters

Our example uses a SimpleCursorAdapter to bind a database Cursor into a ListView, and this is a common way to use a ListAdapter. Other options exist like ArrayAdapter which can be used to take a List or Array of in-memory data and bind it in to a list as well.

This is a bigger chunk of code, so let's first take a look at it:

```
    private void fillData() {
        // Get all of the notes from the database and create the item list
        Cursor c = mDbHelper.fetchAllNotes();
        startManagingCursor(c);

        String[] from = new String[] { NotesDbAdapter.KEY_TITLE };
        int[] to = new int[] { R.id.text1 };

        // Now create an array adapter and set it to display using our row
        SimpleCursorAdapter notes =
            new SimpleCursorAdapter(this, R.layout.notes_row, c, from, to);
        setListAdapter(notes);
    }
```

Here's what we've done:

1. After obtaining the Cursor from `mDbHelper.fetchAllNotes()`, we use an Activity method called `startManagingCursor()` that allows Android to take care of the Cursor lifecycle instead of us needing to worry about it. (We will cover the implications of the lifecycle in exercise 3, but for now just know that this allows Android to do some of our resource management work for us.)
2. Then we create a string array in which we declare the column(s) we want (just the title, in this case), and an int array that defines the View(s) to which we'd like to bind the columns (these should be in order, respective to the string array, but here we only have one for each).
3. Next is the SimpleCursorAdapter instantiation. Like many classes in Android, the SimpleCursorAdapter needs a Context in order to do its work, so we pass in `this` for the context (since subclasses of Activity implement Context). We pass the `notes_row` View we created as the receptacle for the data, the Cursor we just created, and then our arrays.

In the future, remember that the mapping between the **from** columns and **to** resources is done using the respective ordering of the two arrays. If we had more columns we wanted to bind, and more Views to bind them in to, we would specify them in order, for example we might use `{ NotesDbAdapter.KEY_TITLE, NotesDbAdapter.KEY_BODY }` and `{ R.id.text1, R.id.text2 }` to bind two fields into the row (and we would also need to define text2 in the notes_row.xml, for the body text). This is how you can bind multiple fields into a single row (and get a custom row layout as well).

If you get compiler errors about classes not being found, ctrl-shift-O or (cmd-shift-O on the mac) to organize imports.

Step 13

Run it!

1. Right click on the `Notepadv1` project.
2. From the popup menu, select **Run As** > **Android Application**.
3. If you see a dialog come up, select Android Launcher as the way of running the application (you can also use the link near the top of the dialog to set this as your default for the workspace; this is recommended as it will stop the plugin from asking you this every time).
4. Add new notes by hitting the menu button and selecting *Add Item* from the menu.

Solution and Next Steps

You can see the solution to this class in `Notepadv1Solution` from the zip file to compare with your own.

Once you are ready, move on to Tutorial Exercise 2 to add the ability to create, edit and delete notes.

Back to the Tutorial main page...

 Build 110632-110632 - 22 Sep 2008 13:34

Android

Tutorial: Notepad Exercise 2

In this exercise, you will add a second Activity to your notepad application, to let the user create, edit, and delete notes. The new Activity assumes responsibility for creating new notes by collecting user input and packing it into a return Bundle provided by the intent. This exercise demonstrates:

- *Constructing a new Activity and adding it to the Android manifest*
- *Invoking another Activity asynchronously using* `startActivityForResult()`
- *Passing data between Activity in Bundle objects*
- *How to use a more advanced screen layout*

[Exercise 1] [Exercise 2] [Exercise 3] [Extra Credit]

Step 1

Create a new Android project using the sources from `Notepadv2` under the `NotepadCodeLab` folder, just like you did for the first exercise. If you see an error about `AndroidManifest.xml`, or some problems related to an `android.zip` file, right click on the project and select **Android Tools** > **Fix Project Properties**.

Open the `Notepadv2` project and take a look around:

- Open and look at the `strings.xml` file under `res/values` — there are several new strings which we will use for our new functionality
- Also, open and take a look at the top of the `Notepadv2` class, you will notice several new constants have been defined along with a new `mNotesCursor` field used to hold the cursor we are using.
- Note also that the `fillData()` method has a few more comments and now uses the new field to store the notes Cursor. The `onCreate()` method is unchanged from the first exercise. Also notice that the member field used to store the notes Cursor is now called `mNotesCursor`. The `m` denotes a member field and is part of the Android coding style standards.
- There are also a couple of new overridden methods (`onListItemClick()` and `onActivityResult()`) which we will be filling in below.

Step 2

Add an entry to the menu for deleting a note:

1. In the `onCreateOptionsMenu()` method, add a new line:

```
menu.add(0, DELETE_ID, 0, R.string.menu_delete);
```

2. The whole method should now look like this:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
    menu.add(0, INSERT_ID, 0, R.string.menu_insert);
    menu.add(0, DELETE_ID, 0, R.string.menu_delete);
    return true;
}
```

Step 3

In the `onMenuItemSelected()` method, add a new case for `DELETE_ID`:

```
mDbHelper.deleteNote(getListView().getSelectedItemId());
```

```
fillData();
return true;
```

1. Here, we use the `deleteNote` method to remove the note specified by ID. In order to get the ID, we call `getListView().getSelectedItemId()`.
2. Then we fill the data to keep everything up to date.

The whole method should now look like this:

```
@Override
public boolean onMenuItemSelected(int featureId, MenuItem item) {
    switch(item.getItemId()) {
    case INSERT_ID:
        createNote();
        return true;
    case DELETE_ID:
        mDbHelper.deleteNote(getListView().getSelectedItemId());
        fillData();
        return true;
    }

    return super.onMenuItemSelected(featureId, item);
}
```

Step 4

Fill in the body of the `createNote()` method:

Create a new `Intent` to create a note (`ACTIVITY_CREATE`) using the `NoteEdit` class. Then fire the Intent using the `startActivityForResult()` method call:

```
Intent i = new Intent(this, NoteEdit.class);
startActivityForResult(i, ACTIVITY_CREATE);
```

This form of the Intent call targets a specific class in our Activity, in this case `NoteEdit`. Since the Intent class will need to communicate with the Android operating system to route requests, we also have to provide a Context (`this`).

The `startActivityForResult()` method fires the Intent in a way that causes a method in our Activity to be called when the new Activity is completed. The method in our Activity that receives the callback is called `onActivityResult()` and we will implement it in a later step. The other way to call an Activity is using `startActivity()` but this is a "fire-and-forget" way of calling it — in this manner, our Activity is not informed when the Activity is completed, and there is no way to return result information from the called Activity with `startActivity()`.

Starting Other Activities

In this example our Intent uses a class name specifically. As well as starting intents in classes we already know about, be they in our own application or another application, we can also create Intents without knowing exactly which application will handle it.

For example, we might want to open a page in a browser, and for this we still use an Intent. But instead of specifying a class to handle it, we use a predefined Intent constant, and a content URI that describes what we want to do. See android.content.Intent for more information.

Don't worry about the fact that `NoteEdit` doesn't exist yet, we will fix that soon.

Step 5

Fill in the body of the `onListItemClick()` override.

`onListItemClick()` is a callback method that we'll override. It is called when the user selects an item from the list. It is passed four parameters: the `ListView` object it was invoked from, the `View` inside the `ListView` that was clicked on, the `position` in the list that was clicked, and the `mRowId` of the item that was clicked. In this instance we can ignore the first two parameters (we only have one `ListView` it could be), and we ignore the `mRowId` as well. All we are interested in is the `position` that the user selected. We use this to get the data from the correct row, and bundle it up to send to the `NoteEdit` Activity.

In our implementation of the callback, the method creates an `Intent` to edit the note using the `NoteEdit` class. It then adds data into the extras Bundle of the Intent, which will be passed to the called Activity. We use it to pass in the title and body text, and the `mRowId` for the note we are editing. Finally, it will fire the Intent using the `startActivityForResult()` method call.

Here's the code that belongs in `onListItemClick()`:

```
super.onListItemClick(l, v, position, id);
Cursor c = mNotesCursor;
c.moveToPosition(position);
Intent i = new Intent(this, NoteEdit.class);
i.putExtra(NotesDbAdapter.KEY_ROWID, id);
i.putExtra(NotesDbAdapter.KEY_TITLE, c.getString(
        c.getColumnIndexOrThrow(NotesDbAdapter.KEY_TITLE)));
i.putExtra(NotesDbAdapter.KEY_BODY, c.getString(
        c.getColumnIndexOrThrow(NotesDbAdapter.KEY_BODY)));
startActivityForResult(i, ACTIVITY_EDIT);
```

- `putExtra()` is the method to add items into the extras Bundle to pass in to intent invocations. Here, we are using the Bundle to pass in the title, body and mRowId of the note we want to edit.
- The details of the note are pulled out from our query Cursor, which we move to the proper position for the element that was selected in the list, with the `moveToPosition()` method.
- With the extras added to the Intent, we invoke the Intent on the `NoteEdit` class by passing `startActivityForResult()` the Intent and the request code. (The request code will be returned to `onActivityResult` as the `requestCode` parameter.)

Note: We assign the mNotesCursor field to a local variable at the start of the method. This is done as an optimization of the Android code. Accessing a local variable is much more efficient than accessing a field in the Dalvik VM, so by doing this we make only one access to the field, and five accesses to the local variable, making the routine much more efficient. It is recommended that you use this optimization when possible.

Step 6

The above `createNote()` and `onListItemClick()` methods use an asynchronous Intent invocation. We need a handler for the callback, so here we fill in the body of the `onActivityResult()`.

`onActivityResult()` is the overridden method which will be called when an Activity returns with a result. (Remember, an Activity will only return a result if launched with `startActivityForResult`.) The parameters provided to the callback are:

- `requestCode` — the original request code specified in the Intent invocation (either `ACTIVITY_CREATE` or `ACTIVITY_EDIT` for us).
- `resultCode` — the result (or error code) of the call, this should be zero if everything was OK, but may have a non-zero code indicating that something failed. There are standard result codes available, and you can also create your own constants to indicate specific problems.
- `intent` — this is an Intent created by the Activity returning results. It can be used to return data in the Intent "extras."

The combination of `startActivityForResult()` and `onActivityResult()` can be thought of as an asynchronous RPC (remote procedure call) and forms the recommended way for an Activity to invoke another and share services.

Here's the code that belongs in your `onActivityResult()`:

```
super.onActivityResult(requestCode, resultCode, intent);
Bundle extras = intent.getExtras();

switch(requestCode) {
case ACTIVITY_CREATE:
    String title = extras.getString(NotesDbAdapter.KEY_TITLE);
    String body = extras.getString(NotesDbAdapter.KEY_BODY);
    mDbHelper.createNote(title, body);
    fillData();
    break;
case ACTIVITY_EDIT:
    Long mRowId = extras.getLong(NotesDbAdapter.KEY_ROWID);
    if (mRowId != null) {
        String editTitle = extras.getString(NotesDbAdapter.KEY_TITLE);
        String editBody = extras.getString(NotesDbAdapter.KEY_BODY);
        mDbHelper.updateNote(mRowId, editTitle, editBody);
    }
    fillData();
    break;
```

```
}
```

- We are handling both the `ACTIVITY_CREATE` and `ACTIVITY_EDIT` activity results in this method.
- In the case of a create, we pull the title and body from the extras (retrieved from the returned Intent) and use them to create a new note.
- In the case of an edit, we pull the mRowId as well, and use that to update the note in the database.
- `fillData()` at the end ensures everything is up to date .

Step 7

Open the file `note_edit.xml` that has been provided and take a look at it. This is the UI code for the Note Editor.

This is the most sophisticated UI we have dealt with yet. The file is given to you to avoid problems that may sneak in when typing the code. (The XML is very strict about case sensitivity and structure, mistakes in these are the usual cause of problems with layout.)

There is a new parameter used here that we haven't seen before: `android:layout_weight` (in this case set to use the value 1 in each case).

`layout_weight` is used in LinearLayouts to assign "importance" to Views within the layout. All Views have a default `layout_weight` of zero, meaning they take up only as much room on the screen as they need to be displayed. Assigning a value higher than zero will split up the rest of the available space in the parent View, according to the value of each View's `layout_weight` and its ratio to the overall `layout_weight` specified in the current layout for this and other View elements.

The Art of Layout

The provided note_edit.xml layout file is the most sophisticated one in the application we will be building, but that doesn't mean it is even close to the kind of sophistication you will be likely to want in real Android applications.

Creating a good UI is part art and part science, and the rest is work. Mastering Android layout is an essential part of creating a good looking Android application.

Take a look at the View Gallery for some example layouts and how to use them. The ApiDemos sample project is also a great resource from which to learn how to create different layouts.

To give an example: let's say we have a text label and two text edit elements in a horizontal row. The label has no `layout_weight` specified, so it takes up the minimum space required to render. If the `layout_weight` of each of the two text edit elements is set to 1, the remaining width in the parent layout will be split equally between them (because we claim they are equally important). If the first one has a `layout_weight` of 1 and the second has a `layout_weight` of 2, then one third of the remaining space will be given to the first, and two thirds to the second (because we claim the second one is more important).

This layout also demonstrates how to nest multiple layouts inside each other to achieve a more complex and pleasant layout. In this example, a horizontal linear layout is nested inside the vertical one to allow the title label and text field to be alongside each other, horizontally.

Step 8

Create a `NoteEdit` class that extends `android.app.Activity`.

This is the first time we will have created an Activity without the Android Eclipse plugin doing it for us. When you do so, the `onCreate()` method is not automatically overridden for you. It is hard to imagine an Activity that doesn't override the `onCreate()` method, so this should be the first thing you do.

1. Right click on the `com.android.demo.notepad2` package in the Package Explorer, and select **New** > **Class** from the popup menu.
2. Fill in `NoteEdit` for the `Name:` field in the dialog.
3. In the `Superclass:` field, enter `android.app.Activity` (you can also just type Activity and hit Ctrl-Space on Windows and Linux or Cmd-Space on the Mac, to invoke code assist and find the right package and class).
4. Click **Finish**.
5. In the resulting `NoteEdit` class, right click in the editor window and select **Source** > **Override/Implement Methods...**
6. Scroll down through the checklist in the dialog until you see `onCreate(Bundle)` — and check the box next to it.
7. Click **OK**.

   The method should now appear in your class.

Step 9

Fill in the body of the `onCreate()` method for `NoteEdit`.

This will set the title of our new Activity to say "Edit Note" (one of the strings defined in `strings.xml`). It will also set the content view to use our `note_edit.xml` layout file. We can then grab handles to the title and body text edit views, and the confirm button, so that our class can use them to set and get the note title and body, and attach an event to the confirm button for when it is pressed by the user.

We can then unbundle the values that were passed in to the Activity with the extras Bundle attached to the calling Intent. We'll use them to pre-populate the title and body text edit views so that the user can edit them. Then we will grab and store the `mRowId` so we can keep track of what note the user is editing.

1. Inside `onCreate()`, set up the layout:

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

2. Find the edit and button components we need:

   These are found by the IDs associated to them in the R class, and need to be cast to the right type of `View` (`EditText` for the two text views, and `Button` for the confirm button):

```
mTitleText = (EditText) findViewById(R.id.title);
mBodyText = (EditText) findViewById(R.id.body);
Button confirmButton = (Button) findViewById(R.id.confirm);
```

   Note that `mTitleText` and `mBodyText` are member fields (you need to declare them at the top of the class definition).

3. At the top of the class, declare a `Long mRowId` private field to store the current `mRowId` being edited (if any).
4. Continuing inside `onCreate()`, add code to initialize the `title`, `body` and `mRowId` from the extras Bundle in the Intent (if it is present):

```
mRowId = null;
Bundle extras = getIntent().getExtras();
if (extras != null) {
    String title = extras.getString(NotesDbAdapter.KEY_TITLE);
    String body = extras.getString(NotesDbAdapter.KEY_BODY);
    mRowId = extras.getLong(NotesDbAdapter.KEY_ROWID);

    if (title != null) {
        mTitleText.setText(title);
    }
    if (body != null) {
        mBodyText.setText(body);
    }
}
```

   - We are pulling the `title` and `body` out of the `extras` Bundle that was set from the Intent invocation.
   - We also null-protect the text field setting (i.e., we don't want to set the text fields to null accidentally).

5. Create an `onClickListener()` for the button:

   Listeners can be one of the more confusing aspects of UI implementation, but what we are trying to achieve in this case is simple. We want an `onClick()` method to be called when the user presses the confirm button, and use that to do some work and return the values of the edited note to the Intent caller. We do this using something called an anonymous inner class. This is a bit confusing to look at unless you have seen them before, but all you really need to take away from this is that you can refer to this code in the future to see how to create a listener and attach it to a button. (Listeners are a common idiom in Java development, particularly for user interfaces.) Here's the empty listener:

```
confirmButton.setOnClickListener(new View.OnClickListener() {

    public void onClick(View view) {

    }

});
```

Step 10

Fill in the body of the `onClick()` method in our listener.

This is the code that will be run when the user clicks on the confirm button. We want this to grab the title and body text from the edit text fields, and put them into the return Bundle so that they can be passed back to the Activity that invoked this `NoteEdit` Activity. If the operation is an edit rather than a create, we also want to put the `mRowId` into the Bundle so that the `Notepadv2` class can save the changes back to the correct note.

1. Create a `Bundle` and put the title and body text into it using the constants defined in Notepadv2 as keys:

```
Bundle bundle = new Bundle();

bundle.putString(NotesDbAdapter.KEY_TITLE, mTitleText.getText().toString());
bundle.putString(NotesDbAdapter.KEY_BODY, mBodyText.getText().toString());
if (mRowId != null) {
    bundle.putLong(NotesDbAdapter.KEY_ROWID, mRowId);
}
```

2. Set the result information (the Bundle) in a new Intent and finish the Activity:

```
Intent mIntent = new Intent();
mIntent.putExtras(bundle);
setResult(RESULT_OK, mIntent);
finish();
```

   - The Intent is simply our data carrier that carries our Bundle (with the title, body and mRowId).
   - The `setResult()` method is used to set the result code and return Intent to be passed back to the Intent caller. In this case everything worked, so we return RESULT_OK for the result code.
   - The `finish()` call is used to signal that the Activity is done (like a return call). Anything set in the Result will then be returned to the caller, along with execution control.

The full `onCreate()` method (plus supporting class fields) should now look like this:

```
private EditText mTitleText;
private EditText mBodyText;
private Long mRowId;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.note_edit);

    mTitleText = (EditText) findViewById(R.id.title);
    mBodyText = (EditText) findViewById(R.id.body);

    Button confirmButton = (Button) findViewById(R.id.confirm);

    mRowId = null;
    Bundle extras = getIntent().getExtras();
    if (extras != null) {
        String title = extras.getString(NotesDbAdapter.KEY_TITLE);
        String body = extras.getString(NotesDbAdapter.KEY_BODY);
        mRowId = extras.getLong(NotesDbAdapter.KEY_ROWID);

        if (title != null) {
            mTitleText.setText(title);
        }
        if (body != null) {
            mBodyText.setText(body);
        }
    }

    confirmButton.setOnClickListener(new View.OnClickListener() {

        public void onClick(View view) {
            Bundle bundle = new Bundle();
```

```
            bundle.putString(NotesDbAdapter.KEY_TITLE, mTitleText.getText().toString());
            bundle.putString(NotesDbAdapter.KEY_BODY, mBodyText.getText().toString());
            if (mRowId != null) {
                bundle.putLong(NotesDbAdapter.KEY_ROWID, mRowId);
            }

            Intent mIntent = new Intent();
            mIntent.putExtras(bundle);
            setResult(RESULT_OK, mIntent);
            finish();
        }
    });
}
```

Step 11

Finally, the new Activity has to be defined in the manifest file:

Before the new Activity can be seen by Android, it needs its own Activity entry in the `AndroidManifest.xml` file. This is to let the system know that it is there and can be called. We could also specify which IntentFilters the activity implements here, but we are going to skip this for now and just let Android know that the Activity is defined.

There is a Manifest editor included in the Eclipse plugin that makes it much easier to edit the AndroidManifest file, and we will use this. If you prefer to edit the file directly or are not using the Eclipse plugin, see the box at the end for information on how to do this without using the new Manifest editor.

The All-Important Android Manifest File

The AndroidManifest.xml file is the way in which Android sees your application. This file defines the category of the application, where it shows up (or even if it shows up) in the launcher or settings, what activities, services, and content providers it defines, what intents it can receive, and more.

For more information, see the reference document AndroidManifest.xml

1. Double click on the `AndroidManifest.xml` file in the package explorer to open it.
2. Click the **Application** tab at the bottom of the Manifest editor.
3. Click **Add...** in the Application Nodes section.

   If you see a dialog with radiobuttons at the top, select the top radio button: "Create a new element at the top level, in Application".
4. Make sure "(A) Activity" is selected in the selection pane of the dialog, and click **OK**.
5. Click on the new "Activity" node, in the Application Nodes section, then type `.NoteEdit` into the *Name** field to the right. Press Return/Enter.

The Android Manifest editor helps you add more complex entries into the AndroidManifest.xml file, have a look around at some of the other options available (but be careful not to select them otherwise they will be added to your Manifest). This editor should help you understand and alter the AndroidManifest.xml file as you move on to more advanced Android applications.

If you prefer to edit this file directly, simply open the `AndroidManifest.xml` file and look at the source (use the `AndroidManifest.xml` tab in the eclipse editor to see the source code directly). Then edit the file as follows:
`<activity android:name=".NoteEdit"></activity>`

This should be placed just below the line that reads:
`</activity>` for the `.Notepadv2` activity.

Step 12

Now Run it!

You should now be able to add real notes from the menu, as well as delete an existing one. Notice that in order to delete, you must first use the directional controls on the device to highlight the note. Furthermore, selecting a note title from the list should bring up the note editor to let you edit it. Press confirm when finished to save the changes back to the database.

Solution and Next Steps

You can see the solution to this exercise in `Notepadv2Solution` from the zip file to compare with your own.

Now try editing a note, and then hitting the back button on the emulator instead of the confirm button (the back button is below the menu button). You will see an error come up. Clearly our application still has some problems. Worse still, if you did make some changes and hit the back button, when you go back into the notepad to look at the note you changed, you will find that all your changes have been lost. In the next exercise we will fix these problems.

Once you are ready, move on to Tutorial Exercise 3 where you will fix the problems with the back button and lost edits by introducing a proper life cycle into the NoteEdit Activity.

Back to the Tutorial main page....

Build 110632-110632 - 22 Sep 2008 13:34

Android

Tutorial: Notepad Exercise 3

In this exercise, you will use life-cycle event callbacks to store and retrieve application state data. This exercise demonstrates:

- *Life-cycle events and how your application can use them*
- *Techniques for maintaining application state*

[Exercise 1] [Exercise 2] [Exercise 3] [Extra Credit]

Step 1

Import `Notepadv3` into Eclipse. If you see an error about `AndroidManifest.xml`, or some problems related to an Android zip file, right click on the project and select **Android Tools** > **Fix Project Properties** from the popup menu. The starting point for this exercise is exactly where we left off at the end of the Notepadv2.

The current application has some problems — hitting the back button when editing causes a crash, and anything else that happens during editing will cause the edits to be lost.

To fix this, we will move most of the functionality for creating and editing the note into the NoteEdit class, and introduce a full life cycle for editing notes.

1. Remove the code in `NoteEdit` that parses the title and body from the extras Bundle.

   Instead, we are going to use the `DBHelper` class to access the notes from the database directly. All we need passed into the NoteEdit Activity is a `mRowId` (but only if we are editing, if creating we pass nothing). Remove these lines:

   ```
   String title = extras.getString(NotesDbAdapter.KEY_TITLE);
   String body = extras.getString(NotesDbAdapter.KEY_BODY);
   ```

2. We will also get rid of the properties that were being passed in the `extras` Bundle, which we were using to set the title and body text edit values in the UI. So delete:

   ```
   if (title != null) {
       mTitleText.setText(title);
   }
   if (body != null) {
       mBodyText.setText(body);
   }
   ```

Step 2

Create a class field for a `NotesDbAdapter` at the top of the NoteEdit class:

```
    private NotesDbAdapter mDbHelper;
```

Also add an instance of `NotesDbAdapter` in the `onCreate()` method (right below the `super.onCreate()` call):

```
    mDbHelper = new NotesDbAdapter(this);

    mDbHelper.open();
```

Step 3

In `NoteEdit`, we need to check the *savedInstanceState* for the `mRowId`, in case the note editing contains a saved state in the Bundle, which we should recover (this would happen if our Activity lost focus and then restarted).

1. Replace the code that currently initializes the `mRowId`:

```
        mRowId = null;

        Bundle extras = getIntent().getExtras();
        if (extras != null) {
            mRowId = extras.getLong(NotesDbAdapter.KEY_ROWID);
        }
```

   with this:

```
        mRowId = savedInstanceState != null ? savedInstanceState.getLong(NotesDbAdapter.KEY_ROWID)
                                            : null;
        if (mRowId == null) {
            Bundle extras = getIntent().getExtras();
            mRowId = extras != null ? extras.getLong(NotesDbAdapter.KEY_ROWID)
                                    : null;
        }
```

2. Note the null check for `savedInstanceState`, and we still need to load up `mRowId` from the `extras` Bundle if it is not provided by the `savedInstanceState`. This is a ternary operator shorthand to safely either use the value or null if it is not present.

Step 4

Next, we need to populate the fields based on the `mRowId` if we have it:

```
populateFields();
```

This goes before the `confirmButton.setOnClickListener()` line. We'll define this method in a moment.

Step 5

Get rid of the Bundle creation and Bundle value settings from the `onClick()` handler method. The Activity no longer needs to return any extra information to the caller. And because we no longer have an Intent to return, we'll use the shorter version of `setResult()`:

```
public void onClick(View view) {
    setResult(RESULT_OK);
    finish();
}
```

We will take care of storing the updates or new notes in the database ourselves, using the life-cycle methods.

The whole `onCreate()` method should now look like this:

```
super.onCreate(savedInstanceState);

mDbHelper = new NotesDbAdapter(this);
mDbHelper.open();

setContentView(R.layout.note_edit);

mTitleText = (EditText) findViewById(R.id.title);
mBodyText = (EditText) findViewById(R.id.body);

Button confirmButton = (Button) findViewById(R.id.confirm);
```

```
mRowId = savedInstanceState != null ? savedInstanceState.getLong(NotesDbAdapter.KEY_ROWID)
                                    : null;
if (mRowId == null) {
    Bundle extras = getIntent().getExtras();
    mRowId = extras != null ? extras.getLong(NotesDbAdapter.KEY_ROWID)
                            : null;
}

populateFields();

confirmButton.setOnClickListener(new View.OnClickListener() {

    public void onClick(View view) {
        setResult(RESULT_OK);
        finish();
    }

});
```

Step 6

Define the `populateFields()` method.

```
private void populateFields() {
    if (mRowId != null) {
        Cursor note = mDbHelper.fetchNote(mRowId);
        startManagingCursor(note);
        mTitleText.setText(note.getString(
                    note.getColumnIndexOrThrow(NotesDbAdapter.KEY_TITLE)));
        mBodyText.setText(note.getString(
                note.getColumnIndexOrThrow(NotesDbAdapter.KEY_BODY)));
    }
}
```

This method uses the `NotesDbAdapter.fetchNote()` method to find the right note to edit, then it calls `startManagingCursor()` from the `Activity` class, which is an Android convenience method provided to take care of the Cursor life-cycle. This will release and re-create resources as dictated by the Activity life-cycle, so we don't need to worry about doing that ourselves. After that, we just look up the title and body values from the Cursor and populate the View elements with them.

Step 7

Still in the `NoteEdit` class, we now override the methods `onSaveInstanceState()`, `onPause()` and `onResume()`. These are our life-cycle methods (along with `onCreate()` which we already have).

`onSaveInstanceState()` is called by Android if the Activity is being stopped and **may be killed before it is resumed!** This means it should store any state necessary to re-initialize to the same condition when the Activity is restarted. It is the counterpart to the `onCreate()` method, and in fact the `savedInstanceState` Bundle passed in to `onCreate()` is the same Bundle that you construct as `outState` in the `onSaveInstanceState()` method.

`onPause()` and `onResume()` are also complimentary methods. `onPause()` is always called when the Activity ends, even if we instigated that (with a `finish()` call for example). We will use this to save the current note back to the database. Good practice is to release any resources that can be released during an `onPause()` as well, to take up less resources when in the passive state. For this reason we will close the DBHelper class and set the field to null so that it can be garbage collected if necessary. `onResume()` on the other hand, will re-create the `mDbHelper` instance so we can use it, and then read the note out of the database again and populate the fields.

So, add some space after the `populateFields()` method and add the following

Why handling life-cycle events is important

If you are used to always having control in your applications, you might not understand why all this life-cycle work is necessary. The reason is that in Android, you are not in control of your Activity, the operating system is!

As we have already seen, the Android model is based around activities calling each other. When one Activity calls another, the current Activity is paused at the very least, and may be killed altogether if the system starts to run low on resources. If this happens, your Activity will have to store enough state to come back up later, preferably in the same state it was in when it was killed.

Android has a well-defined life cycle. Life-cycle events can happen even if you are not handing off control to another

Activity explicitly. For example, perhaps a call comes in to the handset. If this happens, and your Activity is running, it will be swapped out while the call Activity takes over.

life-cycle methods:

a. `onSaveInstanceState()`:

```
    @Override
    protected void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        outState.putLong(NotesDbAdapter.KEY_ROWID, mRowId);
    }
```

b. `onPause()`:

```
    @Override
    protected void onPause() {
        super.onPause();
        saveState();
    }
```

We'll define `saveState()` next.

c. `onResume()`:

```
    @Override
    protected void onResume() {
        super.onResume();
        populateFields();
    }
```

Step 8

Define the `saveState()` method to put the data out to the database.

```
    private void saveState() {
       String title = mTitleText.getText().toString();
       String body = mBodyText.getText().toString();

       if (mRowId == null) {
           long id = mDbHelper.createNote(title, body);
           if (id > 0) {
               mRowId = id;
           }
       } else {
           mDbHelper.updateNote(mRowId, title, body);
       }
   }
```

Note that we capture the return value from `createNote()` and if a valid row ID is returned, we store it in the `mRowId` field so that we can update the note in future rather than create a new one (which otherwise might happen if the life-cycle events are triggered).

Step 9

Now pull out the previous handling code from the `onActivityResult()` method in the `Notepadv3` class.

All of the note retrieval and updating now happens within the `NoteEdit` life cycle, so all the `onActivityResult()` method needs to do is update its view of the data, no other work is necessary. The resulting method should look like this:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent intent) {
    super.onActivityResult(requestCode, resultCode, intent);
    fillData();
```

```
}
```

Because the other class now does the work, all this has to do is refresh the data.

Step 10

Also remove the lines which set the title and body from the `onListItemClick()` method (again they are no longer needed, only the `mRowId` is):

```
    Cursor c = mNotesCursor;
    c.moveToPosition(position);
```

and also remove:

```
    i.putExtra(NotesDbAdapter.KEY_TITLE, c.getString(
                    c.getColumnIndex(NotesDbAdapter.KEY_TITLE)));
    i.putExtra(NotesDbAdapter.KEY_BODY, c.getString(
                    c.getColumnIndex(NotesDbAdapter.KEY_BODY)));
```

so that all that should be left in that method is:

```
    super.onListItemClick(l, v, position, id);
    Intent i = new Intent(this, NoteEdit.class);
    i.putExtra(NotesDbAdapter.KEY_ROWID, id);
    startActivityForResult(i, ACTIVITY_EDIT);
```

You can also now remove the mNotesCursor field from the class, and set it back to using a local variable in the `fillData()` method:

```
    Cursor notesCursor = mDbHelper.fetchAllNotes();
```

Note that the `m` in `mNotesCursor` denotes a member field, so when we make `notesCursor` a local variable, we drop the `m`. Remember to rename the other occurrences of `mNotesCursor` in your `fillData()` method.

Run it! (use *Run As -> Android Application* on the project right click menu again)

Solution and Next Steps

You can see the solution to this exercise in `Notepadv3Solution` from the zip file to compare with your own.

When you are ready, move on to the Tutorial Extra Credit exercise, where you can use the Eclipse debugger to examine the life-cycle events as they happen.

Back to the Tutorial main page...

Android

Tutorial: Extra Credit

In this exercise, you will use the debugger to look at the work you did in Exercise 3. This exercise demonstrates:

- *How to set breakpoints to observe execution*
- *How to run your application in debug mode*

[Exercise 1] [Exercise 2] [Exercise 3] [Extra Credit]

Step 1

Using the working `Notepadv3`, put breakpoints in the code at the beginning of the `onCreate()`, `onPause()`, `onSaveInstanceState()` and `onResume()` methods in the `NoteEdit` class (if you are not familiar with Eclipse, just right click in the narrow grey border on the left of the edit window at the line you want a breakpoint, and select *Toggle Breakpoint*, you should see a blue dot appear).

Step 2

Now start the notepad demo in debug mode:

a. Right click on the `Notepadv3` project and from the Debug menu select *Debug As -> Android Application.*
b. The Android emulator should say *"waiting for debugger to connect"* briefly and then run the application.
c. If it gets stuck on the waiting... screen, quit the emulator and Eclipse, from the command line do an `adb kill-server`, and then restart Eclipse and try again.

Step 3

When you edit or create a new note you should see the breakpoints getting hit and the execution stopping.

Step 4

Hit the Resume button to let execution continue (yellow rectangle with a green triangle to its right in the Eclipse toolbars near the top).

Step 5

Experiment a bit with the confirm and back buttons, and try pressing Home and making other mode changes. Watch what life-cycle events are generated and when.

The Android Eclipse plugin not only offers excellent debugging support for your application development, but also superb profiling support. You can also try using Traceview to profile your application. If your application is running too slow, this can help you find the bottlenecks and fix them.

Back to the Tutorial main page...

 Build 110632-110632 - 22 Sep 2008 13:34