http://www.paramiao.com/?p=115
http://www.vogella.com/articles/AndroidBuildAnt/article.html
http://www.androidengineer.com/2010/06/using-ant-to-automate-building-android.html

Using Ant to Automate Building Android Applications

The standard way to develop and deploy Android applications is using Eclipse. This is great because it is free, easy to use, and many Java developers already use Eclipse. To deploy your applications using Eclipse, you simply right-click on the on the project, choose to export the application, and follow the prompts

There are a few things we cannot easily do with this system, though. Using the Eclipse GUI does not allow one to easily:
Add custom build steps.
Use an automated build system.
Use build configurations.
Build the release project with one command.
Fortunately, the Android SDK comes equipped with support for Ant, which is a common build script system popular among Java developers. It is how you can develop Android applications without using Eclipse, if you so desire. This tutorial will show you how to incorporate an Ant build script into your Android projects (even if you still develop with Eclipse), create your release package ready for Marketplace deployment in one step, create a build configuration using a properties file, and alter your source files based on the configuration.

You can use the Ant build script to solve all of the problems listed above. This tutorial expects you to already have your Android SDK setup correctly, and to have Ant installed. It will also help to know a little about Ant if you want to add custom build steps, but you don't really need to know anything to follow the tutorial here.

Although I don't personally use an automated build system for my projects, I do use it to create configuration files and to run custom build scripts. I also believe that it is very important to have a one-step build system, which means that there is only one command to create your final release package (I'll explain why later). You can already run your application in debug mode with Eclipse with one step, but I feel it is important to be able to create the release package in one step as well.

Finally, if this is too much reading for your taste, you can jump straight into the summary for a few simple steps, and download the sample application at the end of the tutorial.

Ant in a nutshell

A build script in Ant is an XML file. The default filename for a Ant build file is build.xml. Build steps in Ant are called tasks, which are defined by targets in the build file. When you build your Android application with the default build script, you would type ant release at the command line. In this case, Ant looks for the default filename build.xml, and release is the target which it builds. The release target builds the application ready for release (as opposed to for debugging). Another example would be ant clean, which cleans the project binaries.

You can do pretty much anything you can imagine with more custom build scripts, from copying files to making network calls. More detail about how to use Ant is beyond the scope of this tutorial, but I will show you some useful tricks.

One custom script which I enjoy very much uses ProGuard to obfuscate and shrink the code. I see the code size of my applications drop by a whopping 50% using it. It helps for users who may think your application is taking too much space on their device. I'll explain how to do this in a future tutorial.
Adding build.xml to an existing project

If you already have a project that you'd like to add the Ant build script to, then there is an easy command line tool you can use. Open up a command prompt and navigate to the base directory of your project. From there, use the command:

```
android update project --path .
```

Here is an example of successful output:

```
>android update project --path .
Updated local.properties
Added file C:\dev\blog\antbuild\build.xml
```

If the android command is not found, then you need to update your path to include the Android tools. On Windows, you can use something like set path=%PATH%;C:\dev\android-sdk-windows\tools (substituting your actual Android installation directory), or even better add it to your path persistently by updating the environment variables through your system properties.

Now you will have a working ant build script in build.xml. You can test your setup by typing ant at the command prompt, and you should receive something similar to the following boilerplate help:

```
>ant
Buildfile: C:\dev\blog\antbuild\build.xml
    [setup] Android SDK Tools Revision 6
    [setup] Project Target: Android 1.5
    [setup] API level: 3
    [setup] WARNING: No minSdkVersion value set.
Application will install on all Android versions.
    [setup] Importing rules file: platforms\android-
3\ant\ant_rules_r2.xml

help:
     [echo] Android Ant Build. Available targets:
     [echo]    help:      Displays this help.
     [echo]    clean:     Removes output files
created by other targets.
     [echo]    compile:   Compiles project's .java
files into .class files.
     [echo]    debug:     Builds the application and
signs it with a debug key.
     [echo]    release:   Builds the application. The
generated apk file must be
     [echo]               signed before it is
published.
     [echo]    install:   Installs/reinstalls the
debug package onto a running
     [echo]               emulator or device.
     [echo]               If the application was
previously installed, the
     [echo]               signatures must match.
     [echo]    uninstall: Uninstalls the application
from a running emulator or
     [echo]               device.

BUILD SUCCESSFUL
```

If the ant command is not found, then you need to update your path. Like above, on Windows use set path=%PATH%;C:\dev\apache-ant-1.8.1\bin (substituting your actual Ant installation directory), or even better update your environment variables.

At this point you should be able to type ant release at the command prompt, which will build the project, placing the unsigned .apk file inside of the bin/ directory.

Note that the output from Ant will show further instructions under -release-nosign: which says to sign the apk manually and to run zipalign. We'll get to these steps later in the signing section below.
Creating a new project with build.xml

If you've already created your project and followed the above instructions, you can skip this section. If not, you can may either create a new Android project using the regular Eclipse method (via New > Other... > Android Project), and follow the instructions in the above section, or you can use the command line as described here.

```
android create project --name YourProjectName --path
C:\dev\YourProject --target android-3 --package
com.company.testproject --activity MainActivity
```

Here is an example of successful output:

```
>android create project --name YourTestProject --path
c:\temp\TestProject --target android-3 --package
com.company.testproject --activity MainActivity

Created project directory: c:\temp\TestProject
Created directory
C:\temp\TestProject\src\com\company\testproject
Added file
c:\temp\TestProject\src\com\company\testproject\MainA
ctivity.java
Created directory C:\temp\TestProject\res
Created directory C:\temp\TestProject\bin
Created directory C:\temp\TestProject\libs
Created directory C:\temp\TestProject\res\values
Added file c:\temp\TestProject\res\values\strings.xml
Created directory C:\temp\TestProject\res\layout
Added file c:\temp\TestProject\res\layout\main.xml
Added file c:\temp\TestProject\AndroidManifest.xml
Added file c:\temp\TestProject\build.xml
```

Note: To see the available targets, use android list target and you should see something like:

```
>android list target
id: 1 or "android-3"
     Name: Android 1.5
     Type: Platform
     API level: 3
     Revision: 4
     Skins: HVGA (default), HVGA-L, HVGA-P, QVGA-L,
QVGA-P
```

In the above case, you can use either 1 or android-3 as the target ID. In the sample project, I chose android-4, which corresponds to Android 1.6.

Once the project is created, you can test if your project build is setup correctly by typing ant at the command line. See the above section for further instructions.

Synchronizing with Eclipse

If you open the Ant build script, build.xml, in Eclipse, you will see an error on the second line of the file at this line: <project name="MainActivity" default="help">. The problem with this line is that it is saying that the default Ant target is "help", but the actual Ant targets used in the build file are imported from another location, which the Eclipse editor does not recognize. The import is done at the line <taskdef name="setup", which imports Ant files from the Android SDK.

Unfortunately, while this error is active in your project, you cannot debug your project from Eclipse, even though the Ant build.xml is not needed. There are two solutions. You can remove default="help" from the file, which will remove the error in Eclipse. If you do this, and type ant at a command prompt without any targets (as opposed to "ant release"), you won't

get the default help. Or, you can copy the imported Ant files directly into your code, which is exactly what you must do if you would like to customize your build. If you follow this tutorial, you won't have to worry about this error. See the Customizing the build section for more information.
Automatically signing your application

Before an application can be delivered to a device, the package must be signed. When debugging using Eclipse, the package is technically signed with a debugging key. (Alternatively, you can build a debug package using ant debug) For actual applications delivered to the Android Marketplace, you need to sign them with a real key. It is useful to put this step into the build process. On top of the ease of automating the process, it allows you to build your application in one step. (One-step builds are a Good Idea)

If you have not already created a key, you can do so automatically using Eclipse (Right click project > Android Tools > Export Signed Application Package...), or follow the instructions here.

Now we must tell the build script about our keystore.

Create a file called build.properties in your project's base directory (in the same directory as build.xml and the other properties files), if it does not already exist. Add the following lines:

```
key.store=keystore
key.alias=www.androidengineer.com
```

Where keystore is the name of your keystore file and change the value of key.alias to your keystore's alias. Now when you run ant release, you will be prompted for your passwords, and the build will automatically sign and zipalign your package.

Of course, having to enter your password doesn't make for a one-step build process. So you could not use this for an automated build machine, for one thing. It also has the disadvantage of requiring you to type the password, which it will display clearly on the screen, which may be a security issue in some circumstances. We can put the passwords into build.properties as well, which will solve the issue:

```
key.store.password=password
key.alias.password=password
```

Caution: There can be several issues with storing the keystore and passwords. Depending on your organization's security policies, you may not be able to store the passwords in version control, or you may not be able to give out the information to all developers who have access to the source. If you want to check in the keystore and the build.properties file, but not the passwords, you can create a separate properties file which could only be allowed on certain machines but not checked in to version control. For example, you could create a secure.properties file which goes on the build machine, but not checked in to version control so all developers wouldn't have access to it; import the extra properties file by adding <property file="secure.properties" /> to build.xml. Finally, you could always build the APKs unsigned with ant release by not adding any information to the properties files. The package built using this method will need to be signed and aligned.

Customizing the build

Now that we've got a working Ant build script, we can create a one-step build. But if we want to customize the build further, we'll have to do a few extra steps. You can do anything with your build that you can do with Ant. There are a few things we'll have to do first.

The Ant targets are actually located in the Android SDK. The targets are what you type after ant on the command line, such as release, clean, etc. To customize the build further, we need to copy the imported targets into our own build file.

If you look in build.xml, you can see the instructions for how to customize your build steps:

The rules file is imported from
<SDK>/platforms/<target_platform>/templates/android_rules.xml
To customize some build steps for your project:
  - copy the content of the main node <project> from android_rules.xml
  - paste it in this build.xml below the <setup /> task.
  - disable the import by changing the setup task below to <setup import="false" />

Find the android_rules.xml file in your Android SDK. For example, mine is located at C:\dev\android-sdk-windows\platforms\android-4\templates. There, copy almost the entire file, excluding the project node (copy below <project name="MainActivity"> to above </project>), and paste it in your build.xml file. Also, change the line <setup /> to <setup import="false"/>.

Now you can change around the build as you please. Test that the build file is still working properly by running a build. For an example of what you can do with the custom build script, see the next section.
Using a Java configuration file

This is a great way to use a build property to affect the source code of your Android application. Imagine a configuration class in your project which sets some variables, such as a debugging flag or a URL string. You probably have a different set of these values when developing than when you release your application. For example, you may turn the logging flag off, or change the URL from a debugging server to a production server.

```
public class Config
{
    /** Whether or not to include logging statements
in the application. */
    public final static boolean LOGGING = true;
}
```

It would be nice to have the above LOGGING flag be set from your build. That way, you can be sure that when you create your release package, all of the code you used for debugging won't be included. For example, you may have debugging log statements like this:

```
if (Config.LOGGING)
{
    Log.d(TAG, "[onCreate] Success");
}
```

You will probably want to leave these statements on during development, but remove them at release. In fact, it is good practice to leave logging statements in your source code. It helps with later maintenance when you, and especially others, need to know how your code works. On the other hand, it is bad practice for an application to litter the Android log with your debugging statements. Using these configuration variables allows you to turn the logging on and off, while still leaving the source code intact.

Another great advantage of using this method of logging is that the bytecode contained within the logging statement can be completely removed by a Java bytecode shrinker such as ProGuard, which can be integrated into your build script. I'll discuss how to do this in a later blog post.

A nice way to set the Config.LOGGING flag is in your build properties. Add the following to build.properties:

```
# Turn on or off logging.
config.logging=true
```

To have this build property be incorporated into our source code, I will use the Ant type filterset with the copy task. What we can do is create a Java template file which has tokens such as @CONFIG.LOGGING@ and copy it to our source directory, replacing the tokens with whatever the build properties values are. For example, in the sample application I have a file called Config.java in the config/ directory.

```
public class Config
{
    /** Whether or not to include logging statements
in the application. */
    public final static boolean LOGGING =
@CONFIG.LOGGING@;
}
```

Please note that this is not a source file, and that config/Config.java is notthe actual file used when compiling the project. The file src/com/yourpackage/Config.java, which is the copied file destination, is what will be used as a source file.

Now I will alter the build file to copy the template file to the source path, but replace @CONFIG.LOGGING with the value of the property config.logging, which is true. I will create an Ant target called config which will copy the above template to the source directory. This will be called before the compile target.

```
    <!-- Copy Config.java to our source tree,
replacing custom tokens with values in
build.properties. The configuration depends on
"clean" because otherwise the build system will not
detect changes in the configuration. -->
  <target name="config">

  <property name="config-target-path"
value="${source.dir}/com/androidengineer/antbuild"/>

  <!-- Copy the configuration file, replacing tokens
in the file. -->
  <copy file="config/Config.java" todir="${config-
target-path}"
        overwrite="true" encoding="utf-8">
   <filterset>
    <filter token="CONFIG.LOGGING"
value="${config.logging}"/>
   </filterset>
  </copy>

  <!-- Now set it to read-only, as we don't want
people accidentally
       editing the wrong one. NOTE: This step is
unnecessary, but I do
       it so the developers remember that this is not
the original file. -->
  <chmod file="${config-target-path}/Config.java"
perm="-w"/>
  <attrib file="${config-target-path}/Config.java"
readonly="true"/>

 </target>
```

To make this Ant target execute before the compile target, we simply add config to the dependency of compile: <target name="compile" depends="config, -resource-src, -aidl". We also make the config target call clean, because otherwise the build system will not detect changes in the configuration, and may not

recompile the proper classes.

Note: The above Ant task sets the target file (in your source directory) to read-only. This is not necessary, but I add it as a precaution to remind me that it is not the original file that I need to edit. When developing, I will change the configuration sometimes without using the build, and Eclipse will automatically change the file from read-only for me. I also do not check in the target file into version control; only the original template and build.properties.
Version control

Do not check in the local.properties file which is generated by the Android build tools. This is noted in the file itself; it sets paths based on the local machine. Do check in the default.properties file, which is used by the Android tools, and build.properties, which is the file which you edit to customize your project's build.

I also don't check in the target Config.java in the source directory, nor anything else is configured by the build. I don't want local changes to propagate to other developers, so I only check in the original template file in the config/ directory.

In my projects, when I release a new version of a project I always check in the properties file and tag it in the source repository with a tag name such as "VERSION_2.0". That way we are certain of what properties the application was built with, and we can reproduce the application exactly as it was released, if we later need to.

Summary

1. At the command line run android create project, or android update project in your project base directory if it already exists.

2. (Optional) Add key.store and key.alias to build.properties if you want to include the signing step in your build.
3. (Optional) Add key.store.password and key.alias.password to build.properties if you want to include the keystore passwords, to make the build run without any further input needed.
4. (Optional) If you would like to further customize the build, copy the SDK Ant build code from <SDK>/platforms/<target_platform>/templates/android_rules.xmlto your local build.xml and change <setup /> to <setup import="false"/>.
5. Use ant release to build your project. It will create the package in bin/.
Sample Application

The sample application is a simple Hello World application, but it also includes the custom build script as described in this tutorial. It also includes the Config.java which is configurable by the build. First, you must run "android update project -p ." from the command line in the project's directory to let the tools set the SDK path in local.properties. Then you can turn on and off logging by changing the value of config.logging in build.properties. Finally, run ant release to build the application, which will create the signed bin/MainActivity-release.apk file ready to be released.

Project source code - antbuild.zip (13.4 Kb)

ant clean
Cleans the project. If you include the all target before clean (ant all clean), other projects are also cleaned. For instance if you clean a test project, the tested project is also cleaned.
ant debug
Builds a debug package. Works on application, library, and test projects and compiles dependencies as needed.

ant emma debug
Builds a test project while building the tested project with instrumentation turned on. This is used to run tests with code coverage enabled.
ant release
Builds a release package.
ant instrument
Builds an instrumented debug package. This is generally called automatically when building a test project with code coverage enabled (with the emma target)
ant <build_target> install
Builds and installs a package. Using install by itself fails.
ant installd
Installs an already compiled debug package. This fails if the .apk is not already built.
ant installr
Installs an already compiled release package. This fails if the .apk is not already built.
ant installt
Installs an already compiled test package. Also installs the .apk of the tested application. This fails if the .apk is not already built.
ant installi
Installs an already compiled instrumented package. This is generally not used manually as it's called when installing a test package. This fails if the .apk is not already built.
ant test
Runs the tests (for test projects). The tested and test .apk files must be previously installed.
ant debug installt test
Builds a test project and the tested project, installs both .apk files, and runs the tests.
ant emma debug install test
Builds a test project and the tested project, installs both .apk files, and runs the tests with code coverage enabled.