ANT

INSTALLING ANT ON WINDOWS

Ant comes installed on Macs and is an easy install on Linux as a package. Unfortunately, it’s a bit of a beast to install on Windows. Before beginning, make sure you have the latest JDK installed. If not, go download it from Sun2 and install it. It’s better to install the JDK instead of just the JRE because some Ant tasks require the JDK.

The first step, as with most software, is to download Ant. Go to the Ant homepage and click to download the binary. Because we’re talking about Windows, choose to download the ZIP file rather than any of the others. Scroll down to where it says “Current release of Ant” and click on the ZIP filename. Once downloaded, unzip the file. You’ll now need to choose a permanent home for Ant on the computer. I tend to use c:\java\ant for simplicity, but you can use whatever you want. I do recommend, however, that the path have no spaces in it (spaces make things more complicated).

Set environment variables. Because you’re installing Ant by hand, you also need to deal with setting environment variables by hand. For Windows XP: To set environment variables on Windows XP, right click on My Computer and select Properties. Then go to the Advanced tab and click the Environment Variables button at the bottom. For Windows 7: To set environment variables on Windows 7, right click on Computer and select Properties. Click on Advanced System Settings and click the Environment Variables button at the bottom.

The dialog for both Windows XP and Windows 7 is the same. Make sure you’re only working on system variables and not user variables. The only environment variable that you absolutely need is JAVA\_HOME, which tells Ant the location of your JRE. If you’ve installed the JDK, this is likely c:\Program Files\Java\jdk1.x.x\jre on Windows XP and c:\Program Files(x86)\Java\jdk1.x.x\jre on Windows 7. You’ll note that both have spaces in their paths, which causes a problem. You need to use the mangled name3 instead of the complete name. So for Windows XP, use C:\Progra~1\Java\jdk1.x.x\jre and for Windows 7, use C:\Progra~2\Java\jdk1.6.0\_26\jre if it’s installed in the Program Files(x86) folder (otherwise use the same as Windows XP).

That alone is enough to get Ant to work, but for convenience, it’s a good idea to add the Ant binary path to the PATH variable. This variable is a semicolon-delimited list of directories to search for executables. To be able to run ant in any directory, Windows needs to know both the location for the ant binary and for the java binary. You’ll need to add both of these to the end of the PATH variable. For Windows XP, you’ll likely add something like this:

;c:\java\ant\bin;C:\Progra~1\Java\jdk1.x.x\jre\bin

For Windows 7, it will look something like this:

;c:\java\ant\bin;C:\Progra~2\Java\jdk1.x.x\jre\bin

Once you’ve done that and applied the changes, you’ll need to restart Windows, open a new command prompt to see if the variables are set properly. You should be able to simply run ant and see something like this:

Buildfile: build.xml does not exist!

Build failed

That means Ant is installed properly and is looking for a build.xml file.

CREATING BUILD FILE

A build file is a simple XML file named build.xml

Add xml declaration in the beginning of the file:

<?xml version="1.0"?>

Each project has its own build file.

The file has a root element named <project>. Give a name to your project:

<project name="My Project">

Each command in the project is called Target. Each Target consists of Tasks. Each target goes in <target> element. Name your target: <target name="info">

If no target will be specified for Ant, use a default target, that is written in <project> element:

<project name="My Project" default="info">

<target name="info">

…

</target>

</project>

Now let's add a Task to our Target. Task is the actual work that Ant is going to do. For example to print out a line to the screen add <echo> task a Target:

<target name="info">

<echo>Hello, World!</echo>

</target>

To run one target before another just use *depends* attribute:

<target name="run" depends="compile">

// run the code

</target>

<target name="info">

// compile the code

</target>

To compile a java file use <javac /> element inside your Target. Specify your source directory with .java files and your compiled files directory, where all .class files will reside. Also include *includeantruntime* attribute set to false. This attribute defines whether to include the Ant run-time libraries in the classpath; defaults to yes, unless build.sysclasspath is set. It is usually best to set this to false so the script's behavior is not sensitive to the environment in which it is run.

<target name="compile">

<javac srcdir="./src" destdir="staging" includeantruntime="false" />

</target>

To run the application use <java> element inside your Target. Use <classpath> element inside the <java> to set the path to your compiled classes. Also, include the package name and the class name of your application into the *classname* attribute of the <java> Target:

<target name="run" depends="compile">

<java classname="ant.Demo.AntDemo">

<classpath path="staging" />

</java>

</target>

SPECIFYING WHICH BUILD FILE TO RUN

Perhaps the most important option for Ant is -buildfile. This option lets you control which build file Ant uses, allowing you to divide the targets of a project into multiple files, and select the appropriate build file depending on your actions. A shortcut to -buildfile is -f. To invoke our existing project, we just name it immediately after the -f or -buildfile argument:

ant -buildfile build.xml compile

This is exactly equivalent to calling ant compile with no file specified. If for some reason the current directory was somewhere in the source tree, which is sometimes the case when you are editing text from a console application such as vi, emacs, or even edit, then you can refer to a build file by passing in the appropriate relative file name for your platform, such as ../../../build.xml or ..\..\..\build.xml. This is fiddly. It is better to use the -find option, which must be followed by the name of a build file. This variant does something very special: it searches up the directory tree to find the first build file in a parent directory of that name, and

invokes it. With this option, when you are deep down the source tree editing files, you can easily invoke the project build with the simple command:

ant -find build.xml

CONTROLLING THE AMOUNT OF INFORMATION PROVIDED

Getting rid of the [java] prefix is easy: we run the build file with the -emacs option; this omits the task-name prefix from all lines printed. The option is called -emacs because the output is now in the emacs format for invoked tools, which enables that and other editors to locate the lines on which errors occurred:

ant -emacs execute

The -quiet option reduces the amount of information to a success message or errors:

ant -quiet execute

One of the attributes of echo is the level attribute: error, warning, info, verbose, and debug control the amount of information that appears. The default value info ensures that echoed messages appear in normal builds, or the two levels of even more information, verbose and debug. By inserting an echo statement into our execute target with the level set to warning, we ensure that even when the build is running in quiet mode the output appears. The Ant task declaration:

<echo level="warning" message="running" />

Another option of immediate relevance is -projecthelp. It lists the main targets in a project, and is invaluable whenever you need to know what targets a build file provides. Ant only lists targets containing the optional description attribute, as these are the targets intended for public consumption.

ant –projecthelp

To see both main and sub targets in a project, you must call Ant with the options -projecthelp and -verbose. The more complex a project is, the more useful the -projecthelp feature becomes.

DATATYPES

Most steps to build a typical Java project deal with files and paths (such as classpaths). Ant provides datatypes to handle these two concepts natively. A set of files is a common entity to manipulate for such tasks as compiling, packaging, copying, deleting, and documenting. Defining a fileset of all .java files, for example, is straightforward:

<fileset dir="src" includes="\*\*/\*.java" id="source.fileset"/>

By providing an id attribute, we are defining a reference. This reference name can be used later wherever a fileset is expected. For example, copying our source code to another directory using the previously defined source.fileset is:

<copy todir="backup">

<fileset refid="source.fileset"/>

</copy>

PROPERTIES

Properties are an important way to customize a build process or to just provide shortcuts for strings that are used repeatedly inside a build file. A property has a name and a value; the name is case-sensitive. Properties may be used in the value of task attributes or in the nested text of tasks that support them. This is done by placing the property name between "${" and "}" in the attribute value. For example, if there is a "builddir" property with the value "build", then this could be used in an attribute like this: ${builddir}/classes. This is resolved at run-time as build/classes.

FILESETS

A fileset is a set of files rooted from a single directory. By default, a fileset specified with only a root directory will include all the files in that entire directory tree, including files in all subdirectories recursively. Let’s copy files from one directory to another:

<copy todir="new\_web">

<fileset dir="web"/>

</copy>

Include all JAR files in the lib directory (nonrecursive, no subdirectories are considered):

<fileset dir="lib">

<include name="\*.jar"/>

</fileset>

Include all .java files below the test directory that end with the word “Test”:

<fileset dir="test">

<include="\*\*/\*Test.java"/>

</fileset>

Include all non-JSP pages in the web directory and below:

<fileset dir="web">

<exclude name="\*\*/\*.jsp"/>

</fileset>

By default, includes and excludes are case-sensitive, but this can be disabled by specifying casesensitive="false". The <include> and <exclude> elements are called patternsets.

PATTERNSETS

Filesets accomplish the include/exclude capability by utilizing another of Ant’s core datatypes: the patternset. A patternset is a collection of file matching patterns. A patternset itself does not refer to any actual files until it is nested in a fileset and therefore rooted at a specific directory.

The pattern matching features are as follows:

\* matches zero or more characters.

? matches a single character.

\*\*, used as the name of a directory, represents matching of all directories from

that point down, matching zero or more directories.

A pattern ending with a trailing / or \ implies a trailing \*\*.

**includes** Comma-separated list of patterns of files that must be included. All files are included when omitted.

**excludes** Comma-separated list of patterns of files that must be excluded. No files (except default excludes) are excluded when omitted.

**includesfile** The name of a file; each line of this file is taken to be an include pattern. You can specify more than one include file by using nested includesfile elements.

**excludesfile** The name of a file; each line of this file is taken to be an exclude pattern. You can specify more than one exclude file by using nested excludesfile elements.

<patternset>

<include name="\*.jsp"/>

</patternset>

The <patternset> element is not always explicitly specified when used within a fileset. A fileset implicitly contains patternsets:

<copy todir="new\_web">

<fileset dir="web" includes="\*\*/\*.jsp"/>

</copy>

EXAMPLE BUILDFILE

<project name="MyProject" default="dist" basedir=".">

<description>

simple example build file

</description>

<!-- set global properties for this build -->

<property name="src" location="src"/>

<property name="build" location="build"/>

<property name="dist" location="dist"/>

<target name="init">

<!-- Create the time stamp -->

<tstamp/>

<!-- Create the build directory structure used by compile -->

<mkdir dir="${build}"/>

</target>

<target name="compile" depends="init"

description="compile the source">

<!-- Compile the java code from ${src} into ${build} -->

<javac srcdir="${src}" destdir="${build}"/>

</target>

<target name="dist" depends="compile"

description="generate the distribution">

<!-- Create the distribution directory -->

<mkdir dir="${dist}/lib"/>

<!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->

<jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>

</target>

<target name="clean"

description="clean up">

<!-- Delete the ${build} and ${dist} directory trees -->

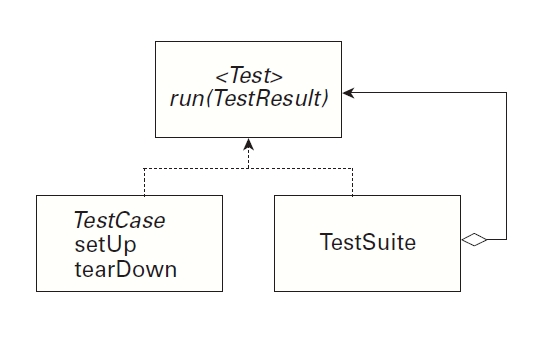
<delete dir="${build}"/>

<delete dir="${dist}"/>

</target>

</project>

JUNIT TESTING

JUnit is a member of the xUnit testing framework family and now the de facto standard testing framework for Java development. JUnit, originally created by Kent Beck and Erich Gamma, is an API that enables developers to create Java test cases. It provides a comprehensive assertion facility to verify expected versus actual results.

JUnit UML diagram depicting the composite pattern utilized by TestCase and TestSuite. A TestSuite contains a collection of tests, which could be either more TestSuites or TestCases, or even classes simply implementing the test interface. Writing a JUnit test case is intentionally designed to be as easy as possible. For a simple test case, you follow three simple steps:

* Create a subclass of junit.framework.TestCase.
* Provide a constructor, accepting a single String name parameter, which calls super(name).
* Implement one or more no-argument void methods prefixed by the word test.

An example is shown in the SimpleTest class code:

package org.example.antbook.junit;

import junit.framework.TestCase;

public class SimpleTest extends TestCase {

public SimpleTest (String name) {

super(name);

}

public void testSomething() {

assertTrue(4 == (2 \* 2));

}

}

TestRunner classes provided by JUnit are used to execute all tests prefixed by the word “test.” The two most popular test runners are a text-based one, junit.textui. TestRunner, and an attractive Swing-based one, junit.swingui.TestRunner.

ASSERTING DESIRED RESULTS

The mechanism by which JUnit determines the success or failure of a test is via assertion statements. An assert is simply a comparison between an expected value and an actual value. There are variants of the assert methods for each primitive datatype and for java.lang.String and java.lang.Object, each with the following signatures:

assertEquals(expected, actual)

assertEquals(String message, expected, actual)

This assertion states that the test expected.equals(actual) returns true, or both objects are null. The equality test for a double also lets you specify a range, to cope with floating point errors better. There are overloaded versions of this method for all Java’s primitive types.

assertSame(Object expected, Object actual),

assertSame(String message, Object expected, Object actual)

Asserts that the two objects are the same. This is a stricter condition than simple equality, as it compares the object identities using expected == actual.

assertNull(Object object),

assertNull(String message, Object object)

This asserts that an object reference equals null.

assertNotNull(Object object),

assertNotNull(String message, Object)

This asserts that an object reference is not null.

assertTrue(boolean condition),

assertTrue(String message, boolean condition)

This assertion fails if the condition is false, printing a message string if supplied. The assertTrue methods were previously named simply assert, but JDK 1.4 introduces a new assert keyword. You may encounter source using the older method names and receive deprecation warnings during compilation.

fail(),

fail(String message)

This forces a failure. This is useful to close off paths through the code that should not be reached.

JUnit uses the term failure for a test that fails expectedly, meaning that an assertion was not valid or a fail was encountered. The term error refers to an unexpected error (such as a NullPointerException).

TESTCASE LIFECYCLE

The lifecycle of a TestCase used by the JUnit framework is as follows:

1. Execute public void setUp().
2. Call a test-prefixed method.
3. Execute public void tearDown().
4. Repeat these steps for each test method.

Any number of test methods can be added to a TestCase, all beginning with the prefix test. The goal is for each test to be small and simple, and tests will usually require instantiating objects. In order to create some objects and preconfigure their state prior to running each individual test method, override the empty TestCase.setUp method, and store state as member variables to your test case class. Use the TestCase.tearDown method to close any open connections or in some way reset state. The setUp and tearDown methods are called before and after every test method is invoked, preventing one test from affecting the behavior of another.

Tests should never make assumptions about the order in which they are called.

WRITING A TESTSUITE

With JUnit’s API, tests can be grouped into a suite by using the TestSuite class. Grouping tests may be a benefit to let you build several individual test cases for a particular subsystem and write an all-inclusive TestSuite that runs them all. A TestSuite also allows specific ordering of tests, which may be important—although ideally the order of tests should not be relevant as each should be able to stand alone. Here is an example of a test suite:

public class AllTests extends TestSuite {

static public Test suite() {

TestSuite suite = new TestSuite();

suite.addTestSuite(SimpleTest.class);

return suite;

}

}