**Tutorial: Hello World with Apache Ant**

This document provides a step by step tutorial for starting java programming with Apache Ant. It does **not** contain deeper knowledge about Java or Ant. This tutorial has the goal to let you see, how to do the easiest steps in Ant.

**Content**

* [Preparing the project](http://ant.apache.org/manual/tutorial-HelloWorldWithAnt.html#prepare)
* [Enhance the build file](http://ant.apache.org/manual/tutorial-HelloWorldWithAnt.html#four-steps)
* [Enhance the build file](http://ant.apache.org/manual/tutorial-HelloWorldWithAnt.html#enhance)
* [Using external libraries](http://ant.apache.org/manual/tutorial-HelloWorldWithAnt.html#ext-libs)
* [Resources](http://ant.apache.org/manual/tutorial-HelloWorldWithAnt.html#resources)

**Preparing the project**

We want to separate the source from the generated files, so our java source files will be in src folder. All generated files should be under build, and there splitted into several subdirectories for the individual steps: classes for our compiled files and jar for our own JAR-file.

We have to create only the src directory. (Because I am working on Windows, here is the win-syntax - translate to your shell):

md src

The following simple Java class just prints a fixed message out to STDOUT, so just write this code into src\oata\HelloWorld.java.

package oata;

public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello World");

}

}

Now just try to compile and run that:

md build\classes

javac -sourcepath src -d build\classes src\oata\HelloWorld.java

java -cp build\classes oata.HelloWorld

which will result in

Hello World

Creating a jar-file is not very difficult. But creating a *startable* jar-file needs more steps: create a manifest-file containing the start class, creating the target directory and archiving the files.

echo Main-Class: oata.HelloWorld>myManifest

md build\jar

jar cfm build\jar\HelloWorld.jar myManifest -C build\classes .

java -jar build\jar\HelloWorld.jar

**Note:** Do not have blanks around the >-sign in the echo Main-Class instruction because it would falsify it!

**Four steps to a running application**

After finishing the java-only step we have to think about our build process. We *have* to compile our code, otherwise we couldn't start the program. Oh - "start" - yes, we could provide a target for that. We *should* package our application. Now it's only one class - but if you want to provide a download, no one would download several hundreds files ... (think about a complex Swing GUI - so let us create a jar file. A startable jar file would be nice ... And it's a good practise to have a "clean" target, which deletes all the generated stuff. Many failures could be solved just by a "clean build".

By default Ant uses build.xml as the name for a buildfile, so our .\build.xml would be:

<project>

<target name="clean">

<delete dir="build"/>

</target>

<target name="compile">

<mkdir dir="build/classes"/>

<javac srcdir="src" destdir="build/classes"/>

</target>

<target name="jar">

<mkdir dir="build/jar"/>

<jar destfile="build/jar/HelloWorld.jar" basedir="build/classes">

<manifest>

<attribute name="Main-Class" value="oata.HelloWorld"/>

</manifest>

</jar>

</target>

<target name="run">

<java jar="build/jar/HelloWorld.jar" fork="true"/>

</target>

</project>

Now you can compile, package and run the application via

ant compile

ant jar

ant run

Or shorter with

ant compile jar run

While having a look at the buildfile, we will see some similar steps between Ant and the java-only commands:

|  |  |
| --- | --- |
| **java-only** | **Ant** |
| md build\classes  javac  -sourcepath src  -d build\classes  src\oata\HelloWorld.java  echo Main-Class: oata.HelloWorld>mf  md build\jar  jar cfm  build\jar\HelloWorld.jar  mf  -C build\classes  .  java -jar build\jar\HelloWorld.jar | <mkdir dir="build/classes"/>  <javac  srcdir="src"  destdir="build/classes"/>  *<!-- automatically detected -->*  *<!-- obsolete; done via manifest tag -->*  <mkdir dir="build/jar"/>  <jar  destfile="build/jar/HelloWorld.jar"  basedir="build/classes">  <manifest>  <attribute name="Main-Class" value="oata.HelloWorld"/>  </manifest>  </jar>  <java jar="build/jar/HelloWorld.jar" fork="true"/> |

**Enhance the build file**

Now we have a working buildfile we could do some enhancements: many time you are referencing the same directories, main-class and jar-name are hard coded, and while invocation you have to remember the right order of build steps.

The first and second point would be addressed with *properties*, the third with a special property - an attribute of the <project>-tag and the fourth problem can be solved using dependencies.

<project name="HelloWorld" basedir="." default="main">

<property name="src.dir" value="src"/>

<property name="build.dir" value="build"/>

<property name="classes.dir" value="${build.dir}/classes"/>

<property name="jar.dir" value="${build.dir}/jar"/>

<property name="main-class" value="oata.HelloWorld"/>

<target name="clean">

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

</target>

<target name="compile">

<mkdir dir="${classes.dir}"/>

<javac srcdir="${src.dir}" destdir="${classes.dir}"/>

</target>

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

<mkdir dir="${jar.dir}"/>

<jar destfile="${jar.dir}/${ant.project.name}.jar" basedir="${classes.dir}">

<manifest>

<attribute name="Main-Class" value="${main-class}"/>

</manifest>

</jar>

</target>

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

<java jar="${jar.dir}/${ant.project.name}.jar" fork="true"/>

</target>

<target name="clean-build" depends="clean,jar"/>

<target name="main" depends="clean,run"/>

</project>

Now it's easier, just do a ant and you will get

Buildfile: build.xml

clean:

compile:

[mkdir] Created dir: C:\...\build\classes

[javac] Compiling 1 source file to C:\...\build\classes

jar:

[mkdir] Created dir: C:\...\build\jar

[jar] Building jar: C:\...\build\jar\HelloWorld.jar

run:

[java] Hello World

main:

BUILD SUCCESSFUL

**Using external libraries**

Somehow told us not to use syso-statements. For log-Statements we should use a Logging-API - customizable on a high degree (including switching off during usual life (= not development) execution). We use Log4J for that, because

* it is not part of the JDK (1.4+) and we want to show how to use external libs
* it can run under JDK 1.2 (as Ant)
* it's highly configurable
* it's from Apache ;-)

We store our external libraries in a new directory lib. Log4J can be [downloaded [1]](http://www.apache.org/dist/logging/log4j/1.2.13/logging-log4j-1.2.13.zip) from Logging's Homepage. Create the lib directory and extract the log4j-1.2.9.jar into that lib-directory. After that we have to modify our java source to use that library and our buildfile so that this library could be accessed during compilation and run.

Working with Log4J is documented inside its manual. Here we use the *MyApp*-example from the [Short Manual [2]](http://logging.apache.org/log4j/docs/manual.html). First we have to modify the java source to use the logging framework:

package oata;

**import org.apache.log4j.Logger;**

**import org.apache.log4j.BasicConfigurator;**

public class HelloWorld {

**static Logger logger = Logger.getLogger(HelloWorld.class);**

public static void main(String[] args) {

**BasicConfigurator.configure();**

**logger.info("Hello World");** // the old SysO-statement

}

}

Most of the modifications are "framework overhead" which has to be done once. The blue line is our "old System-out" statement.

Don't try to run ant - you will only get lot of compiler errors. Log4J is not inside the classpath so we have to do a little work here. But do not change the CLASSPATH environment variable! This is only for this project and maybe you would break other environments (this is one of the most famous mistakes when working with Ant). We introduce Log4J (or to be more precise: all libraries (jar-files) which are somewhere under .\lib) into our buildfile:

<project name="HelloWorld" basedir="." default="main">

...

**<property name="lib.dir" value="lib"/>**

**<path id="classpath">**

**<fileset dir="${lib.dir}" includes="\*\*/\*.jar"/>**

**</path>**

...

<target name="compile">

<mkdir dir="${classes.dir}"/>

<javac srcdir="${src.dir}" destdir="${classes.dir}" **classpathref="classpath"**/>

</target>

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

<java fork="true" **classname="${main-class}"**>

**<classpath>**

**<path refid="classpath"/>**

**<path location="${jar.dir}/${ant.project.name}.jar"/>**

**</classpath>**

</java>

</target>

...

</project>

In this example we start our application not via its Main-Class manifest-attribute, because we could not provide a jarname *and* a classpath. So add our class in the red line to the already defined path and start as usual. Running ant would give (after the usual compile stuff):

[java] 0 [main] INFO oata.HelloWorld - Hello World

What's that?

* *[java]* Ant task running at the moment
* *0* sorry don't know - some Log4J stuff
* *[main]* the running thread from our application
* *INFO* log level of that statement
* *oata.HelloWorld* source of that statement
* *-* separator
* *Hello World* the message

For another layout ... have a look inside Log4J's documentation about using other PatternLayout's.

**Configuration files**

Why we have used Log4J? "It's highly configurable"? No - all is hard coded! But that is not the debt of Log4J - it's ours. We had codedBasicConfigurator.configure(); which implies a simple, but hard coded configuration. More comfortable would be using a property file. In the java source delete the BasicConfiguration-line from the main() method (and the related import-statement). Log4J will search then for a configuration as described in it's manual. Then create a new file src/log4j.properties. That's the default name for Log4J's configuration and using that name would make life easier - not only the framework knows what is inside, you too!

log4j.rootLogger=DEBUG, **stdout**

log4j.appender.**stdout**=org.apache.log4j.ConsoleAppender

log4j.appender.**stdout**.layout=org.apache.log4j.PatternLayout

log4j.appender.**stdout**.layout.ConversionPattern=**%m%n**

This configuration creates an output channel ("Appender") to console named as stdout which prints the message (%m) followed by a line feed (%n) - same as the earlier System.out.println() :-) Oooh kay - but we haven't finished yet. We should deliver the configuration file, too. So we change the buildfile:

...

<target name="compile">

<mkdir dir="${classes.dir}"/>

<javac srcdir="${src.dir}" destdir="${classes.dir}" classpathref="classpath"/>

**<copy todir="${classes.dir}">**

**<fileset dir="${src.dir}" excludes="\*\*/\*.java"/>**

**</copy>**

</target>

...

This copies all resources (as long as they haven't the suffix ".java") to the build directory, so we could start the application from that directory and these files will included into the jar.

**Testing the class**

In this step we will introduce the usage of the JUnit [3] testframework in combination with Ant. Because Ant has a built-in JUnit 3.8.2 you could start directly using it. Write a test class in src\HelloWorldTest.java:

public class HelloWorldTest extends junit.framework.TestCase {

public void testNothing() {

}

public void testWillAlwaysFail() {

fail("An error message");

}

}

Because we dont have real business logic to test, this test class is very small: just show how to start. For further information see the JUnit documentation [3] and the manual of [junit](http://ant.apache.org/manual/Tasks/junit.html) task. Now we add a junit instruction to our buildfile:

...

<path **id="application"** location="${jar.dir}/${ant.project.name}.jar"/>

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

<java fork="true" classname="${main-class}">

<classpath>

<path refid="classpath"/>

**<path refid="application"/>**

</classpath>

</java>

</target>

**<target name="junit" depends="jar">**

**<junit printsummary="yes">**

**<classpath>**

**<path refid="classpath"/>**

**<path refid="application"/>**

**</classpath>**

**<batchtest fork="yes">**

**<fileset dir="${src.dir}" includes="\*Test.java"/>**

**</batchtest>**

**</junit>**

**</target>**

...

We reuse the path to our own jar file as defined in run-target by giving it an ID and making it globally available. The printsummary=yes lets us see more detailed information than just a "FAILED" or "PASSED" message. How much tests failed? Some errors? Printsummary lets us know. The classpath is set up to find our classes. To run tests the batchtest here is used, so you could easily add more test classes in the future just by naming them\*Test.java. This is a common naming scheme.

After a ant junit you'll get:

...

junit:

[junit] Running HelloWorldTest

[junit] Tests run: 2, Failures: 1, Errors: 0, Time elapsed: 0,01 sec

[junit] Test HelloWorldTest FAILED

BUILD SUCCESSFUL

...

We can also produce a report. Something that you (and other) could read after closing the shell .... There are two steps: 1. let <junit> log the information and 2. convert these to something readable (browsable).

...

**<property name="report.dir" value="${build.dir}/junitreport"/>**

...

<target name="junit" depends="jar">

**<mkdir dir="${report.dir}"/>**

<junit printsummary="yes">

<classpath>

<path refid="classpath"/>

<path refid="application"/>

</classpath>

**<formatter type="xml"/>**

<batchtest fork="yes" **todir="${report.dir}"**>

<fileset dir="${src.dir}" includes="\*Test.java"/>

</batchtest>

</junit>

</target>

**<target name="junitreport">**

**<junitreport todir="${report.dir}">**

**<fileset dir="${report.dir}" includes="TEST-\*.xml"/>**

**<report todir="${report.dir}"/>**

**</junitreport>**

**</target>**

Because we would produce a lot of files and these files would be written to the current directory by default, we define a report directory, create it before running the junit and redirect the logging to it. The log format is XML so junitreport could parse it. In a second target junitreport should create a browsable HTML-report for all generated xml-log files in the report directory. Now you can open the ${report.dir}\index.html and see the result (looks something like JavaDoc).  
Personally I use two different targets for junit and junitreport. Generating the HTML report needs some time and you dont need the HTML report just for testing, e.g. if you are fixing an error or a integration server is doing a job.

**Resources**

[1] <http://www.apache.org/dist/logging/log4j/1.2.13/logging-log4j-1.2.13.zip>

[2] <http://logging.apache.org/log4j/docs/manual.html>

[3] <http://www.junit.org/index.htm>

**Tutorial: Writing Tasks**

This document provides a step by step tutorial for writing tasks.

**Content**

* [Set up the build environment](http://ant.apache.org/manual/tutorial-writing-tasks.html#buildenvironment)
* [Write the Task](http://ant.apache.org/manual/tutorial-writing-tasks.html#write1)
* [Use the Task](http://ant.apache.org/manual/tutorial-writing-tasks.html#use1)
* [Integration with TaskAdapter](http://ant.apache.org/manual/tutorial-writing-tasks.html#TaskAdapter)
* [Deriving from Apache Ant's Task](http://ant.apache.org/manual/tutorial-writing-tasks.html#derivingFromTask)
* [Accessing the Task's Project](http://ant.apache.org/manual/tutorial-writing-tasks.html#accessTaskProject)
* [Attributes](http://ant.apache.org/manual/tutorial-writing-tasks.html#attributes)
* [Nested Text](http://ant.apache.org/manual/tutorial-writing-tasks.html#NestedText)
* [Nested Elements](http://ant.apache.org/manual/tutorial-writing-tasks.html#NestedElements)
* [Our task in a little more complex version](http://ant.apache.org/manual/tutorial-writing-tasks.html#complex)
* [Test the Task](http://ant.apache.org/manual/tutorial-writing-tasks.html#TestingTasks)
* [Debugging](http://ant.apache.org/manual/tutorial-writing-tasks.html#Debugging)
* [Resources](http://ant.apache.org/manual/tutorial-writing-tasks.html#resources)

**Set up the build environment**

Apache Ant builds itself, we are using Ant too (why we would write a task if not? :-) therefore we should use Ant for our build.

We choose a directory as root directory. All things will be done here if I say nothing different. I will reference this directory as *root-directory* of our project. In this root-directory we create a text file names *build.xml*. What should Ant do for us?

* compiles my stuff
* make the jar, so that I can deploy it
* clean up everything

So the buildfile contains three targets.

<?xml version="1.0" encoding="ISO-8859-1"?>

<project name="MyTask" basedir="." default="jar">

<target name="clean" description="Delete all generated files">

<delete dir="classes"/>

<delete file="MyTasks.jar"/>

</target>

<target name="compile" description="Compiles the Task">

<javac srcdir="src" destdir="classes"/>

</target>

<target name="jar" description="JARs the Task">

<jar destfile="MyTask.jar" basedir="classes"/>

</target>

</project>

This buildfile uses often the same value (src, classes, MyTask.jar), so we should rewrite that using <property>s. On second there are some handicaps:<javac> requires that the destination directory exists; a call of "clean" with a non existing classes directory will fail; "jar" requires the execution of some steps before. So the refactored code is:

<?xml version="1.0" encoding="ISO-8859-1"?>

<project name="MyTask" basedir="." default="jar">

**<property name="src.dir" value="src"/>**

**<property name="classes.dir" value="classes"/>**

<target name="clean" description="Delete all generated files">

<delete dir="**${classes.dir}**" **failonerror="false"**/>

<delete file="**${ant.project.name}.jar**"/>

</target>

<target name="compile" description="Compiles the Task">

**<mkdir dir="${classes.dir}"/>**

<javac srcdir="**${src.dir}**" destdir="${classes.dir}"/>

</target>

<target name="jar" description="JARs the Task" **depends="compile"**>

<jar destfile="${ant.project.name}.jar" basedir="${classes.dir}"/>

</target>

</project>

*ant.project.name* is one of the [build-in properties [1]](http://ant.apache.org/manual/properties.html#built-in-props) of Ant.

**Write the Task**

Now we write the simplest Task - a HelloWorld-Task (what else?). Create a text file *HelloWorld.java* in the src-directory with:

public class HelloWorld {

public void execute() {

System.out.println("Hello World");

}

}

and we can compile and jar it with ant (default target is "jar" and via its *depends*-clause the "compile" is executed before).

**Use the Task**

But after creating the jar we want to use our new Task. Therefore we need a new target "use". Before we can use our new task we have to declare it with[<taskdef> [2]](http://ant.apache.org/manual/Tasks/taskdef.html). And for easier process we change the default clause:

<?xml version="1.0" encoding="ISO-8859-1"?>

<project name="MyTask" basedir="." default="**use**">

...

**<target name="use" description="Use the Task" depends="jar">**

**<taskdef name="helloworld" classname="HelloWorld" classpath="${ant.project.name}.jar"/>**

**<helloworld/>**

**</target>**

</project>

Important is the *classpath*-attribute. Ant searches in its /lib directory for tasks and our task isn't there. So we have to provide the right location.

Now we can type in ant and all should work ...

Buildfile: build.xml

compile:

[mkdir] Created dir: C:\tmp\anttests\MyFirstTask\classes

[javac] Compiling 1 source file to C:\tmp\anttests\MyFirstTask\classes

jar:

[jar] Building jar: C:\tmp\anttests\MyFirstTask\MyTask.jar

use:

[helloworld] Hello World

BUILD SUCCESSFUL

Total time: 3 seconds

**Integration with TaskAdapter**

Our class has nothing to do with Ant. It extends no superclass and implements no interface. How does Ant know to integrate? Via name convention: our class provides a method with signature public void execute(). This class is wrapped by Ant's org.apache.tools.ant.TaskAdapter which is a task and uses reflection for setting a reference to the project and calling the *execute()* method.

*Setting a reference to the project*? Could be interesting. The Project class gives us some nice abilities: access to Ant's logging facilities getting and setting properties and much more. So we try to use that class:

import org.apache.tools.ant.Project;

public class HelloWorld {

private Project project;

public void setProject(Project proj) {

project = proj;

}

public void execute() {

String message = project.getProperty("ant.project.name");

project.log("Here is project '" + message + "'.", Project.MSG\_INFO);

}

}

and the execution with ant will show us the expected

use:

Here is project 'MyTask'.

**Deriving from Ant's Task**

Ok, that works ... But usually you will extend org.apache.tools.ant.Task. That class is integrated in Ant, get's the project-reference, provides documentation fields, provides easier access to the logging facility and (very useful) gives you the exact location where *in the buildfile* this task instance is used.

Oki-doki - let's us use some of these:

import org.apache.tools.ant.Task;

public class HelloWorld extends Task {

public void execute() {

// use of the reference to Project-instance

String message = getProject().getProperty("ant.project.name");

// Task's log method

log("Here is project '" + message + "'.");

// where this task is used?

log("I am used in: " + getLocation() );

}

}

which gives us when running

use:

[helloworld] Here is project 'MyTask'.

[helloworld] I am used in: C:\tmp\anttests\MyFirstTask\build.xml:23:

**Accessing the Task's Project**

The parent project of your custom task may be accessed through method getProject(). However, do not call this from the custom task constructor, as the return value will be null. Later, when node attributes or text are set, or method execute() is called, the Project object is available.

Here are two useful methods from class Project:

* String getProperty(String propertyName)
* String replaceProperties(String value)

The method replaceProperties() is discussed further in section [Nested Text](http://ant.apache.org/manual/tutorial-writing-tasks.html#NestedText).

**Attributes**

Now we want to specify the text of our message (it seems that we are rewriting the <echo/> task :-). First we well do that with an attribute. It is very easy - for each attribute provide a public void set<attributename>(<type> newValue) method and Ant will do the rest via reflection.

import org.apache.tools.ant.Task;

import org.apache.tools.ant.BuildException;

public class HelloWorld extends Task {

String message;

public void setMessage(String msg) {

message = msg;

}

public void execute() {

if (message==null) {

throw new BuildException("No message set.");

}

log(message);

}

}

Oh, what's that in execute()? Throw a *BuildException*? Yes, that's the usual way to show Ant that something important is missed and complete build should fail. The string provided there is written as build-failes-message. Here it's necessary because the log() method can't handle a *null* value as parameter and throws a NullPointerException. (Of course you can initialize the *message* with a default string.)

After that we have to modify our buildfile:

<target name="use" description="Use the Task" depends="jar">

<taskdef name="helloworld"

classname="HelloWorld"

classpath="${ant.project.name}.jar"/>

<helloworld **message="Hello World"**/>

</target>

That's all.

Some background for working with attributes: Ant supports any of these datatypes as arguments of the set-method:

* elementary data type like *int*, *long*, ...
* its wrapper classes like *java.lang.Integer*, *java.lang.Long*, ...
* *java.lang.String*
* some more classes (e.g. *java.io.File*; see [Manual 'Writing Your Own Task' [3]](http://ant.apache.org/manual/develop.html#set-magic))
* Any Java Object parsed from Ant 1.8's [Property Helper](http://ant.apache.org/manual/Tasks/propertyhelper.html)

Before calling the set-method all properties are resolved. So a <helloworld message="${msg}"/> would not set the message string to "${msg}" if there is a property "msg" with a set value.

**Nested Text**

Maybe you have used the <echo> task in a way like <echo>Hello World</echo>. For that you have to provide a public void addText(String text)method.

...

public class HelloWorld extends Task {

private String message;

...

public void addText(String text) {

message = text;

}

...

}

But here properties are **not** resolved! For resolving properties we have to use Project's replaceProperties(String propname) : String method which takes the property name as argument and returns its value (or ${propname} if not set).

Thus, to replace properties in the nested node text, our method addText() can be written as:

public void addText(String text) {

message = getProject().replaceProperties(text);

}

**Nested Elements**

There are several ways for inserting the ability of handling nested elements. See the [Manual [4]](http://ant.apache.org/manual/develop.html#nested-elements) for other. We use the first way of the three described ways. There are several steps for that:

1. We create a class for collecting all the info the nested element should contain. This class is created by the same rules for attributes and nested elements as for the task (set<attributename>() methods).
2. The task holds multiple instances of this class in a list.
3. A factory method instantiates an object, saves the reference in the list and returns it to Ant Core.
4. The execute() method iterates over the list and evaluates its values.

import java.util.Vector;

import java.util.Iterator;

...

public void execute() {

if (message!=null) log(message);

for (Iterator it=messages.iterator(); it.hasNext(); ) { **// 4**

Message msg = (Message)it.next();

log(msg.getMsg());

}

}

Vector messages = new Vector(); **// 2**

public Message createMessage() { **// 3**

Message msg = new Message();

messages.add(msg);

return msg;

}

public class Message { **// 1**

public Message() {}

String msg;

public void setMsg(String msg) { this.msg = msg; }

public String getMsg() { return msg; }

}

...

Then we can use the new nested element. But where is xml-name for that defined? The mapping XML-name : classname is defined in the factory method: public *classname* create*XML-name*(). Therefore we write in the buildfile

<helloworld>

<message msg="Nested Element 1"/>

<message msg="Nested Element 2"/>

</helloworld>

Note that if you choose to use methods 2 or 3, the class that represents the nested element must be declared as static

**Our task in a little more complex version**

For recapitulation now a little refactored buildfile:

<?xml version="1.0" encoding="ISO-8859-1"?>

<project name="MyTask" basedir="." default="use">

<property name="src.dir" value="src"/>

<property name="classes.dir" value="classes"/>

<target name="clean" description="Delete all generated files">

<delete dir="${classes.dir}" failonerror="false"/>

<delete file="${ant.project.name}.jar"/>

</target>

<target name="compile" description="Compiles the Task">

<mkdir dir="${classes.dir}"/>

<javac srcdir="${src.dir}" destdir="${classes.dir}"/>

</target>

<target name="jar" description="JARs the Task" depends="compile">

<jar destfile="${ant.project.name}.jar" basedir="${classes.dir}"/>

</target>

<target name="use.init"

description="Taskdef the HelloWorld-Task"

depends="jar">

<taskdef name="helloworld"

classname="HelloWorld"

classpath="${ant.project.name}.jar"/>

</target>

<target name="use.without"

description="Use without any"

depends="use.init">

<helloworld/>

</target>

<target name="use.message"

description="Use with attribute 'message'"

depends="use.init">

<helloworld message="attribute-text"/>

</target>

<target name="use.fail"

description="Use with attribute 'fail'"

depends="use.init">

<helloworld fail="true"/>

</target>

<target name="use.nestedText"

description="Use with nested text"

depends="use.init">

<helloworld>nested-text</helloworld>

</target>

<target name="use.nestedElement"

description="Use with nested 'message'"

depends="use.init">

<helloworld>

<message msg="Nested Element 1"/>

<message msg="Nested Element 2"/>

</helloworld>

</target>

<target name="use"

description="Try all (w/out use.fail)"

depends="use.without,use.message,use.nestedText,use.nestedElement"

/>

</project>

And the code of the task:

import org.apache.tools.ant.Task;

import org.apache.tools.ant.BuildException;

import java.util.Vector;

import java.util.Iterator;

/\*\*

\* The task of the tutorial.

\* Print a message or let the build fail.

\* @since 2003-08-19

\*/

public class HelloWorld extends Task {

/\*\* The message to print. As attribute. \*/

String message;

public void setMessage(String msg) {

message = msg;

}

/\*\* Should the build fail? Defaults to *false*. As attribute. \*/

boolean fail = false;

public void setFail(boolean b) {

fail = b;

}

/\*\* Support for nested text. \*/

public void addText(String text) {

message = text;

}

/\*\* Do the work. \*/

public void execute() {

// handle attribute 'fail'

if (fail) throw new BuildException("Fail requested.");

// handle attribute 'message' and nested text

if (message!=null) log(message);

// handle nested elements

for (Iterator it=messages.iterator(); it.hasNext(); ) {

Message msg = (Message)it.next();

log(msg.getMsg());

}

}

/\*\* Store nested 'message's. \*/

Vector messages = new Vector();

/\*\* Factory method for creating nested 'message's. \*/

public Message createMessage() {

Message msg = new Message();

messages.add(msg);

return msg;

}

/\*\* A nested 'message'. \*/

public class Message {

// Bean constructor

public Message() {}

/\*\* Message to print. \*/

String msg;

public void setMsg(String msg) { this.msg = msg; }

public String getMsg() { return msg; }

}

}

And it works:

C:\tmp\anttests\MyFirstTask>ant

Buildfile: build.xml

compile:

[mkdir] Created dir: C:\tmp\anttests\MyFirstTask\classes

[javac] Compiling 1 source file to C:\tmp\anttests\MyFirstTask\classes

jar:

[jar] Building jar: C:\tmp\anttests\MyFirstTask\MyTask.jar

use.init:

use.without:

use.message:

[helloworld] attribute-text

use.nestedText:

[helloworld] nested-text

use.nestedElement:

[helloworld]

[helloworld]

[helloworld]

[helloworld]

[helloworld] Nested Element 1

[helloworld] Nested Element 2

use:

BUILD SUCCESSFUL

Total time: 3 seconds

C:\tmp\anttests\MyFirstTask>ant use.fail

Buildfile: build.xml

compile:

jar:

use.init:

use.fail:

BUILD FAILED

C:\tmp\anttests\MyFirstTask\build.xml:36: Fail requested.

Total time: 1 second

C:\tmp\anttests\MyFirstTask>

Next step: test ...

**Test the Task**

We have written a test already: the use.\* tasks in the buildfile. But its difficult to test that automatically. Common (and in Ant) used is JUnit for that. For testing tasks Ant provides a JUnit Rule org.apache.tools.ant.BuildFileRule. This class provides some for testing tasks useful methods: initialize Ant, load a buildfile, execute targets, capturing debug and run logs ...

In Ant it is usual that the testcase has the same name as the task with a prepending *Test*, therefore we will create a file *HelloWorldTest.java*. Because we have a very small project we can put this file into *src* directory (Ant's own testclasses are in /src/testcases/...). Because we have already written our tests for "hand-test" we can use that for automatic tests, too. But there is one little problem we have to solve: all test supporting classes are not part of the binary distribution of Ant. So you can build the special jar file from source distro with target "test-jar" or you can download a nightly build from[http://gump.covalent.net/jars/latest/ant/ant-testutil.jar [5]](http://gump.covalent.net/jars/latest/ant/ant-testutil.jar).

For executing the test and creating a report we need the optional tasks <junit> and <junitreport>. So we add to the buildfile:

...

<project name="MyTask" basedir="." default="test">

...

<property name="ant.test.lib" value="ant-testutil.jar"/>

<property name="report.dir" value="report"/>

<property name="junit.out.dir.xml" value="${report.dir}/junit/xml"/>

<property name="junit.out.dir.html" value="${report.dir}/junit/html"/>

<path id="classpath.run">

<path path="${java.class.path}"/>

<path location="${ant.project.name}.jar"/>

</path>

<path id="classpath.test">

<path refid="classpath.run"/>

<path location="${ant.test.lib}"/>

</path>

<target name="clean" description="Delete all generated files">

<delete failonerror="false" includeEmptyDirs="true">

<fileset dir="." includes="${ant.project.name}.jar"/>

<fileset dir="${classes.dir}"/>

<fileset dir="${report.dir}"/>

</delete>

</target>

<target name="compile" description="Compiles the Task">

<mkdir dir="${classes.dir}"/>

<javac srcdir="${src.dir}" destdir="${classes.dir}" classpath="${ant.test.lib}"/>

</target>

...

<target name="junit" description="Runs the unit tests" depends="jar">

<delete dir="${junit.out.dir.xml}"/>

<mkdir dir="${junit.out.dir.xml}"/>

<junit printsummary="yes" haltonfailure="no">

<classpath refid="classpath.test"/>

<formatter type="xml"/>

<batchtest fork="yes" todir="${junit.out.dir.xml}">

<fileset dir="${src.dir}" includes="\*\*/\*Test.java"/>

</batchtest>

</junit>

</target>

<target name="junitreport" description="Create a report for the rest result">

<mkdir dir="${junit.out.dir.html}"/>

<junitreport todir="${junit.out.dir.html}">

<fileset dir="${junit.out.dir.xml}">

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

</fileset>

<report format="frames" todir="${junit.out.dir.html}"/>

</junitreport>

</target>

<target name="test"

depends="junit,junitreport"

description="Runs unit tests and creates a report"

/>

...

Back to the *src/HelloWorldTest.java*. We create a class with a public *BuildFileRule* field annotated with JUnit's *@Rule* annotation. As per conventional JUnit4 tests, this class should have no constructors, or a default no-args constructor, setup methods should be annotated with *@Before*, tear down methods annotated with *@After* and any test method annotated with *@Test*.

import org.apache.tools.ant.BuildFileRule;

import org.junit.Assert;

import org.junit.Test;

import org.junit.Before;

import org.junit.Rule;

import org.apache.tools.ant.AntAssert;

import org.apache.tools.ant.BuildException;

public class HelloWorldTest {

@Rule

public final BuildFileRule buildRule = new BuildFileRule();

@Before

public void setUp() {

// initialize Ant

buildRule.configureProject("build.xml");

}

@Test

public void testWithout() {

buildRule.executeTarget("use.without");

assertEquals("Message was logged but should not.", buildRule.getLog(), "");

}

public void testMessage() {

// execute target 'use.nestedText' and expect a message

// 'attribute-text' in the log

buildRule.executeTarget("use.message");

Assert.assertEquals("attribute-text", buildRule.getLog());

}

@Test

public void testFail() {

// execute target 'use.fail' and expect a BuildException

// with text 'Fail requested.'

try {

buildRule.executeTarget("use.fail");

fail("BuildException should have been thrown as task was set to fail");

} catch (BuildException ex) {

Assert.assertEquals("fail requested", ex.getMessage());

}

}

@Test

public void testNestedText() {

buildRule.executeTarget("use.nestedText");

Assert.assertEquals("nested-text", buildRule.getLog());

}

@Test

public void testNestedElement() {

buildRule.executeTarget("use.nestedElement");

AntAssert.assertContains("Nested Element 1", buildRule.getLog());

AntAssert.assertContains("Nested Element 2", buildRule.getLog());

}

}

When starting ant we'll get a short message to STDOUT and a nice HTML-report.

C:\tmp\anttests\MyFirstTask>ant

Buildfile: build.xml

compile:

[mkdir] Created dir: C:\tmp\anttests\MyFirstTask\classes

[javac] Compiling 2 source files to C:\tmp\anttests\MyFirstTask\classes

jar:

[jar] Building jar: C:\tmp\anttests\MyFirstTask\MyTask.jar

junit:

[mkdir] Created dir: C:\tmp\anttests\MyFirstTask\report\junit\xml

[junit] Running HelloWorldTest

[junit] Tests run: 5, Failures: 0, Errors: 0, Time elapsed: 2,334 sec

junitreport:

[mkdir] Created dir: C:\tmp\anttests\MyFirstTask\report\junit\html

[junitreport] Using Xalan version: Xalan Java 2.4.1

[junitreport] Transform time: 661ms

test:

BUILD SUCCESSFUL

Total time: 7 seconds

C:\tmp\anttests\MyFirstTask>

**Debugging**

Try running Ant with the flag -verbose. For more information, try flag -debug.

For deeper issues, you may need to run the custom task code in a Java debugger. First, get the source for Ant and build it with debugging information.

Since Ant is a large project, it can be a little tricky to set the right breakpoints. Here are two important breakpoints for version 1.8:

* Initial main() function: com.apache.tools.ant.launch.Launcher.main()
* Task entry point: com.apache.tools.ant.UnknownElement.execute()

If you need to debug when a task attribute or the text is set, begin by debugging into method execute() of your custom task. Then set breakpoints in other methods. This will ensure the class byte-code has been loaded by the Java VM.

**Resources**

This tutorial and its resources are available via [BugZilla [6]](http://issues.apache.org/bugzilla/show_bug.cgi?id=22570). The ZIP provided there contains

* this initial version of this tutorial
* the buildfile (last version)
* the source of the task (last version)
* the source of the unit test (last version)
* the ant-testutil.jar (nightly build of 2003-08-18)
* generated classes
* generated jar
* generated reports

The last sources and the buildfile are also available [here [7]](http://ant.apache.org/manual/tutorial-writing-tasks-src.zip) inside the manual.

Used Links:  
  [1] <http://ant.apache.org/manual/properties.html#built-in-props>  
  [2] <http://ant.apache.org/manual/Tasks/taskdef.html>  
  [3] <http://ant.apache.org/manual/develop.html#set-magic>  
  [4] <http://ant.apache.org/manual/develop.html#nested-elements>  
  [5] <http://gump.covalent.net/jars/latest/ant/ant-testutil.jar>  
  [6] <http://issues.apache.org/bugzilla/show_bug.cgi?id=22570>  
  [7] [tutorial-writing-tasks-src.zip](http://ant.apache.org/manual/tutorial-writing-tasks-src.zip)

# Tutorial: Tasks using Properties, Filesets & Paths

After reading the tutorial about [writing tasks [1]](http://ant.apache.org/manual/tutorial-writing-tasks.html) this tutorial explains how to get and set properties and how to use nested filesets and paths. Finally it explains how to contribute tasks to Apache Ant.

## Content

* [The goal](http://ant.apache.org/manual/tutorial-tasks-filesets-properties.html#goal)
* [Build environment](http://ant.apache.org/manual/tutorial-tasks-filesets-properties.html#buildenvironment)
* [Property access](http://ant.apache.org/manual/tutorial-tasks-filesets-properties.html#propertyaccess)
* [Using filesets](http://ant.apache.org/manual/tutorial-tasks-filesets-properties.html#filesets)
* [Using nested paths](http://ant.apache.org/manual/tutorial-tasks-filesets-properties.html#path)
* [Returning a list](http://ant.apache.org/manual/tutorial-tasks-filesets-properties.html#returning-list)
* [Documentation](http://ant.apache.org/manual/tutorial-tasks-filesets-properties.html#documentation)
* [Contribute the new task](http://ant.apache.org/manual/tutorial-tasks-filesets-properties.html#contribute)
* [Resources](http://ant.apache.org/manual/tutorial-tasks-filesets-properties.html#resources)

## The goal

The goal is to write a task, which searchs in a path for a file and saves the location of that file in a property.

## Build environment

We can use the buildfile from the other tutorial and modify it a little bit. That's the advantage of using properties - we can reuse nearly the whole script. :-)

<?xml version="1.0" encoding="ISO-8859-1"?>

<project name="**FindTask**" basedir="." default="test">

...

<target name="use.init" description="Taskdef's the **Find**-Task" depends="jar">

<taskdef name="**find**" classname="**Find**" classpath="${ant.project.name}.jar"/>

</target>

**<!-- the other use.\* targets are deleted -->**

...

</project>

The buildfile is in the archive [tutorial-tasks-filesets-properties.zip [2]](http://ant.apache.org/manual/tutorial-tasks-filesets-properties.zip) in /build.xml.01-propertyaccess (future version saved as \*.02..., final version as build.xml; same for sources).

## Property access

Our first step is to set a property to a value and print the value of that property. So our scenario would be

<find property="test" value="test-value"/>

<find print="test"/>

ok, can be rewritten with the core tasks

<property name="test" value="test-value"/>

<echo message="${test}"/>

but I have to start on known ground :-)

So what to do? Handling three attributes (property, value, print) and an execute method. Because this is only an introduction example I don't do much checking:

import org.apache.tools.ant.BuildException;

public class Find extends Task {

private String property;

private String value;

private String print;

public void setProperty(String property) {

this.property = property;

}

// setter for value and print

public void execute() {

if (print != null) {

String propValue = **getProject().getProperty(print)**;

log(propValue);

} else {

if (property == null) throw new BuildException("property not set");

if (value == null) throw new BuildException("value not set");

**getProject().setNewProperty(property, value)**;

}

}

}

As said in the other tutorial, the property access is done via Project instance. We get this instance via the public getProject() method which we inherit from Task (more precise from ProjectComponent). Reading a property is done via getProperty(*propertyname*) (very simple, isn't it?). This property returns the value as String or *null* if not set.  
Setting a property is ... not really difficult, but there is more than one setter. You can use the setProperty() method which will do the job like expected. But there is a golden rule in Ant: *properties are immutable*. And this method sets the property to the specified value - whether it has a value before that or not. So we use another way. setNewProperty() sets the property only if there is no property with that name. Otherwise a message is logged.

*(by the way: a short word to ants "namespaces" (don't be confused with xml namespaces: an <antcall> creates a new space for property names. All properties from the caller are passed to the callee, but the callee can set its own properties without notice by the caller.)*

There are some other setter, too (but I haven't used them, so I can't say something to them, sorry :-)

After putting our two line example from above into a target names use.simple we can call that from our testcase:

import org.junit.Rule;

import org.junit.Test;

import org.junit.Before;

import org.junit.Assert;

import org.apache.tools.ant.BuildFileRule;

public class FindTest {

@Rule

public final BuildFileRule buildRule = new BuildFileRule();

@Before

public void setUp() {

configureProject("build.xml");

}

@Test

public void testSimple() {

buildRule.executeTarget("useSimgle");

**Assert.assertEquals("test-value", buildRule.getLog());**

}

}

and all works fine.

## Using filesets

Ant provides a common way of bundling files: the fileset. Because you are reading this tutorial I think you know them and I don't have to spend more explanations about their usage in buildfiles. Our goal is to search a file in path. And on this step the path is simply a fileset (or more precise: a collection of filesets). So our usage would be

<find file="ant.jar" location="location.ant-jar">

<fileset dir="${ant.home}" includes="\*\*/\*.jar"/>

</find>

What do we need? A task with two attributes (file, location) and nested filesets. Because we had attribute handling already explained in the example above and the handling of nested elements is described in the other tutorial the code should be very easy:

public class Find extends Task {

private String file;

private String location;

private Vector filesets = new Vector();

public void setFile(String file) {

this.file = file;

}

public void setLocation(String location) {

this.location = location;

}

public void addFileset(FileSet fileset) {

filesets.add(fileset);

}

public void execute() {

}

}

Ok - that task wouldn't do very much, but we can use it in the described manner without failure. On next step we have to implement the execute method. And before that we will implement the appropriate testcases (TDD - test driven development).

In the other tutorial we have reused the already written targets of our buildfile. Now we will configure most of the testcases via java code (sometimes it's much easier to write a target than doing it via java coding). What can be tested?

* not valid configured task (missing file, missing location, missing fileset)
* don't find a present file
* behaviour if file can't be found

Maybe you find some more testcases. But this is enough for now.  
For each of these points we create a testXX method.

public class FindTest {

@Rule

public final BuildFileRule buildRule = new BuildFileRule();

... // constructor, setUp as above

@Test

public void testMissingFile() {

**Find find = new Find();**

try {

**find.execute();**

fail("No 'no-file'-exception thrown.");

} catch (Exception e) {

// exception expected

String expected = "file not set";

assertEquals("Wrong exception message.", expected, e.getMessage());

}

}

@Test

public void testMissingLocation() {

Find find = new Find();

**find.setFile("ant.jar");**

try {

find.execute();

fail("No 'no-location'-exception thrown.");

} catch (Exception e) {

... // similar to testMissingFile()

}

}

@Test

public void testMissingFileset() {

Find find = new Find();

find.setFile("ant.jar");

find.setLocation("location.ant-jar");

try {

find.execute();

fail("No 'no-fileset'-exception thrown.");

} catch (Exception e) {

... // similar to testMissingFile()

}

}

@Test

public void testFileNotPresent() {

buildRule.executeTarget("testFileNotPresent");

String result = buildRule.getProject().getProperty("location.ant-jar");

assertNull("Property set to wrong value.", result);

}

@Test

public void testFilePresent() {

buildRule.executeTarget("testFilePresent");

String result = buildRule.getProject().getProperty("location.ant-jar");

assertNotNull("Property not set.", result);

assertTrue("Wrong file found.", result.endsWith("ant.jar"));

}

}

If we run this test class all test cases (except *testFileNotPresent*) fail. Now we can implement our task, so that these test cases will pass.

protected void validate() {

if (file==null) throw new BuildException("file not set");

if (location==null) throw new BuildException("location not set");

if (filesets.size()<1) throw new BuildException("fileset not set");

}

public void execute() {

validate(); // 1

String foundLocation = null;

for(Iterator itFSets = filesets.iterator(); itFSets.hasNext(); ) { // 2

FileSet fs = (FileSet)itFSets.next();

DirectoryScanner ds = fs.getDirectoryScanner(getProject()); // 3

String[] includedFiles = ds.getIncludedFiles();

for(int i=0; i<includedFiles.length; i++) {

String filename = includedFiles[i].replace('\\','/'); // 4

filename = filename.substring(filename.lastIndexOf("/")+1);

if (foundLocation==null && file.equals(filename)) {

File base = ds.getBasedir(); // 5

File found = new File(base, includedFiles[i]);

foundLocation = found.getAbsolutePath();

}

}

}

if (foundLocation!=null) // 6

getProject().setNewProperty(location, foundLocation);

}

On **//1** we check the prerequisites for our task. Doing that in a validate-method is a common way, because we separate the prerequisites from the real work. On **//2** we iterate over all nested filesets. If we don't want to handle multiple filesets, the addFileset() method has to reject the further calls. We can get the result of a fileset via its DirectoryScanner like done in **//3**. After that we create a platform independent String representation of the file path (**//4**, can be done in other ways of course). We have to do the replace(), because we work with a simple string comparison. Ant itself is platform independent and can therefore run on filesystems with slash (/, e.g. Linux) or backslash (\, e.g. Windows) as path separator. Therefore we have to unify that. If we found our file we create an absolute path representation on **//5**, so that we can use that information without knowing the basedir. (This is very important on use with multiple filesets, because they can have different basedirs and the return value of the directory scanner is relative to its basedir.) Finally we store the location of the file as property, if we had found one (**//6**).

Ok, much more easier in this simple case would be to add the *file* as additional *include* element to all filesets. But I wanted to show how to handle complex situations without being complex :-)

The test case uses the ant property *ant.home* as reference. This property is set by the Launcher class which starts ant. We can use that property in our buildfiles as a [build-in property [3]](http://ant.apache.org/manual/properties.html#built-in-props). But if we create a new ant environment we have to set that value for our own. And we use the <junit> task in fork-mode. Therefore we have do modify our buildfile:

<target name="junit" description="Runs the unit tests" depends="jar">

<delete dir="${junit.out.dir.xml}"/>

<mkdir dir="${junit.out.dir.xml}"/>

<junit printsummary="yes" haltonfailure="no">

<classpath refid="classpath.test"/>

**<sysproperty key="ant.home" value="${ant.home}"/>**

<formatter type="xml"/>

<batchtest fork="yes" todir="${junit.out.dir.xml}">

<fileset dir="${src.dir}" includes="\*\*/\*Test.java"/>

</batchtest>

</junit>

</target>

## Using nested paths

A task providing support for filesets is a very comfortable one. But there is another possibility of bundling files: the <path>. Fileset are easy if the files are all under a common base directory. But if this is not the case you have a problem. Another disadvantage is its speed: if you have only a few files in a huge directory structure, why not use a <filelist> instead? <path>s combines these datatypes in that way that a path contains other paths, filesets, dirsets and filelists. This is why [Ant-Contribs [4]](http://ant-contrib.sourceforge.net/) <foreach> task is modified to support paths instead of filesets. So we want that, too.

Changing from fileset to path support is very easy:

***Change java code from:***

private Vector filesets = new Vector();

public void addFileset(FileSet fileset) {

filesets.add(fileset);

}

***to:***

private Vector paths = new Vector(); \*1

public void add**Path**(**Path** path) { \*2

paths.add(path);

}

***and build file from:***

<find file="ant.jar" location="location.ant-jar">

<fileset dir="${ant.home}" includes="\*\*/\*.jar"/>

</find>

***to:***

<find file="ant.jar" location="location.ant-jar">

**<path>** \*3

<fileset dir="${ant.home}" includes="\*\*/\*.jar"/>

</path>

</find>

On **\*1** we rename only the vector. Itï¿½s just for better reading the source. On **\*2** we have to provide the right method: an add*Name*(*Type* t). Therefore replace the fileset with path here. Finally we have to modify our buildfile on **\*3** because our task doesnï¿½t support nested filesets any longer. So we wrap the fileset inside a path.

And now we modify the testcase. Oh, not very much to do :-) Renaming the testMissingFileset() (not really a *must-be* but better itï¿½s named like the think it does) and update the *expected*-String in that method (now a path not set message is expected). The more complex test cases base on the buildscript. So the targets testFileNotPresent and testFilePresent have to be modified in the manner described above.

The test are finished. Now we have to adapt the task implementation. The easiest modification is in the validate() method where we change le last line to if (paths.size()<1) throw new BuildException("path not set");. In the execute() method we have a little more work. ... mmmh ... in reality it's lesser work, because the Path class does the whole DirectoryScanner-handling and creating-absolute-paths stuff for us. So the execute method is just:

public void execute() {

validate();

String foundLocation = null;

for(Iterator itPaths = paths.iterator(); itPaths.hasNext(); ) {

Path path = (**Path**)itPaths.next(); // 1

String[] includedFiles = **path.list()**; // 2

for(int i=0; i<includedFiles.length; i++) {

String filename = includedFiles[i].replace('\\','/');

filename = filename.substring(filename.lastIndexOf("/")+1);

if (foundLocation==null && file.equals(filename)) {

**foundLocation = includedFiles[i];** // 3

}

}

}

if (foundLocation!=null)

getProject().setNewProperty(location, foundLocation);

}

Of course we have to do the typecase to Path on **//1**. On **//2** and **//3** we see that the Path class does the work for us: no DirectoryScanner (was at 2) and no creating of the absolute path (was at 3).

## Returning a list

So far so good. But could a file be on more than one place in the path? - Of course.  
And would it be good to get all of them? - It depends on ...

In this section we will extend that task to support returning a list of all files. Lists as property values are not supported by Ant natively. So we have to see how other tasks use lists. The most famous task using lists is Ant-Contribs <foreach>. All list elements are concatenated and separated with a customizable separator (default ',').

So we do the following:

<find ... **delimiter=""**/> ... </find>

If the delimiter is set we will return all found files as list with that delimiter.

Therefore we have to

* provide a new attribute
* collect more than the first file
* delete duplicates
* create the list if necessary
* return that list

So we add as testcase:

***in the buildfile:***

<target name="test.init">

<mkdir dir="test1/dir11/dir111"/> \*1

<mkdir dir="test1/dir11/dir112"/>

...

<touch file="test1/dir11/dir111/test"/>

<touch file="test1/dir11/dir111/not"/>

...

<touch file="test1/dir13/dir131/not2"/>

<touch file="test1/dir13/dir132/test"/>

<touch file="test1/dir13/dir132/not"/>

<touch file="test1/dir13/dir132/not2"/>

<mkdir dir="test2"/>

<copy todir="test2"> \*2

<fileset dir="test1"/>

</copy>

</target>

<target name="testMultipleFiles" depends="use.init,**test.init**"> \*3

<find file="test" location="location.test" **delimiter=";"**>

<path>

<fileset dir="test1"/>

<fileset dir="test2"/>

</path>

</find>

<delete> \*4

<fileset dir="test1"/>

<fileset dir="test2"/>

</delete>

</target>

***in the test class:***

public void testMultipleFiles() {

executeTarget("testMultipleFiles");

String result = getProject().getProperty("location.test");

assertNotNull("Property not set.", result);

assertTrue("Only one file found.", result.indexOf(";") > -1);

}

Now we need a directory structure where we CAN find files with the same name in different directories. Because we can't sure to have one we create one on **\*1** and **\*2**. And of course we clean up that on **\*4**. The creation can be done inside our test target or in a separate one, which will be better for reuse later (**\*3**).

The task implementation is modified as followed:

private Vector foundFiles = new Vector();

...

private String delimiter = null;

...

public void setDelimiter(String delim) {

delimiter = delim;

}

...

public void execute() {

validate();

// find all files

for(Iterator itPaths = paths.iterator(); itPaths.hasNext(); ) {

Path path = (Path)itPaths.next();

String[] includedFiles = path.list();

for(int i=0; i<includedFiles.length; i++) {

String filename = includedFiles[i].replace('\\','/');

filename = filename.substring(filename.lastIndexOf("/")+1);

if (file.equals(filename) && **!foundFiles.contains(includedFiles[i]**)) { // 1

foundFiles.add(includedFiles[i]);

}

}

}

// create the return value (list/single)

String rv = null;

if (foundFiles.size() > 0) { // 2

if (delimiter==null) {

// only the first

rv = (String)foundFiles.elementAt(0);

} else {

// create list

StringBuffer list = new StringBuffer();

for(Iterator it=foundFiles.iterator(); it.hasNext(); ) { // 3

list.append(it.next());

if (**it.hasNext()**) list.append(delimiter); // 4

}

rv = list.toString();

}

}

// create the property

if (rv!=null)

getProject().setNewProperty(location, rv);

}

The algorithm does: finding all files, creating the return value depending on the users wish, returning the value as property. On **//1** we eliminates the duplicates. **//2** ensures that we create the return value only if we have found one file. On **//3** we iterate over all found files and **//4** ensures that the last entry has no trailing delimiter.

Ok, first searching for all files and then returning only the first one ... You can tune the performance of your own :-)

## Documentation

A task is useless if the only who is able to code the buildfile is the task developer (and he only the next few weeks :-). So documentation is also very important. In which form you do that depends on your favourite. But inside Ant there is a common format and it has advantages if you use that: all task users know that form, this form is requested if you decide to contribute your task. So we will doc our task in that form.

If you have a look at the manual page of the [Java task [5]](http://ant.apache.org/manual/Tasks/java.html) you will see that it:

* is plain html
* starts with the name
* has sections: description, parameters, nested elements, (maybe return codes) and (most important :-) examples
* parameters are listed in a table with columns for attribute name, its description and whether it's required (if you add a feature after an Ant release, provide a since Ant xx statement when it's introduced)
* describe the nested elements (since-statement if necessary)
* provide one or more useful examples; first code, then description.

As a template we have:

<html>

<head>

<meta http-equiv="Content-Language" content="en-us">

<title>**Taskname** Task</title>

</head>

<body>

<h2><a name="**taskname**">**Taskname**</a></h2>

<h3>Description</h3>

<p> **Describe the task.**</p>

<h3>Parameters</h3>

<table border="1" cellpadding="2" cellspacing="0">

<tr>

<td valign="top"><b>Attribute</b></td>

<td valign="top"><b>Description</b></td>

<td align="center" valign="top"><b>Required</b></td>

</tr>

**do this html row for each attribute (including inherited attributes)**

<tr>

<td valign="top">classname</td>

<td valign="top">the Java class to execute.</td>

<td align="center" valign="top">Either jar or classname</td>

</tr>

</table>

<h3>Parameters specified as nested elements</h3>

**Describe each nested element (including inherited)**

<h4>**your nested element**</h4>

<p>**description**</p>

<p><em>since Ant 1.6</em>.</p>

<h3>Examples</h3>

<pre>

**A code sample; don't forget to escape the < of the tags with &lt;**

</pre>

**What should that example do?**

</body>

</html>

Here is an example documentation page for our task:

<html>

<head>

<meta http-equiv="Content-Language" content="en-us">

<title>Find Task</title>

</head>

<body>

<h2><a name="find">Find</a></h2>

<h3>Description</h3>

<p>Searchs in a given path for a file and returns the absolute to it as property.

If delimiter is set this task returns all found locations.</p>

<h3>Parameters</h3>

<table border="1" cellpadding="2" cellspacing="0">

<tr>

<td valign="top"><b>Attribute</b></td>

<td valign="top"><b>Description</b></td>

<td align="center" valign="top"><b>Required</b></td>

</tr>

<tr>

<td valign="top">file</td>

<td valign="top">The name of the file to search.</td>

<td align="center" valign="top">yes</td>

</tr>

<tr>

<td valign="top">location</td>

<td valign="top">The name of the property where to store the location</td>

<td align="center" valign="top">yes</td>

</tr>

<tr>

<td valign="top">delimiter</td>

<td valign="top">A delimiter to use when returning the list</td>

<td align="center" valign="top">only if the list is required</td>

</tr>

</table>

<h3>Parameters specified as nested elements</h3>

<h4>path</h4>

<p>The path where to search the file.</p>

<h3>Examples</h3>

<pre>

<find file="ant.jar" location="loc">

<path>

<fileset dir="${ant.home}"/>

<path>

</find>

</pre>

Searches in Ants home directory for a file <i>ant.jar</i> and stores its location in

property <i>loc</i> (should be ANT\_HOME/bin/ant.jar).

<pre>

<find file="ant.jar" location="loc" delimiter=";">

<path>

<fileset dir="C:/"/>

<path>

</find>

<echo>ant.jar found in: ${loc}</echo>

</pre>

Searches in Windows C: drive for all <i>ant.jar</i> and stores their locations in

property <i>loc</i> delimited with <i>';'</i>. (should need a long time :-)

After that it prints out the result (e.g. C:/ant-1.5.4/bin/ant.jar;C:/ant-1.6/bin/ant.jar).

</body>

</html>

## Contribute the new task

If we decide to contribute our task, we should do some things:

* is our task welcome? :-) Simply ask on the user list
* is the right package used?
* does the code conform to the styleguide?
* do all tests pass?
* does the code compile on JDK 1.2 (and passes all tests there)?
* code under Apache license
* create a patch file
* publishing that patch file

The [Ant Task Guidelines [6]](http://ant.apache.org/ant_task_guidelines.html) support additional information on that.

Now we will check the "Checklist before submitting a new task" described in that guideline.

* Java file begins with Apache license statement. ***must do that***
* Task does not depend on GPL or LGPL code. ***ok***
* Source code complies with style guidelines ***have to check (checkstyle)***
* Code compiles and runs on Java1.2 ***have to try***
* Member variables are private, and provide public accessor methods if access is actually needed. ***have to check (checkstyle)***
* *Maybe* Task has failonerror attribute to control failure behaviour ***hasn't***
* New test cases written and succeed ***passed on JDK 1.4, have to try on JDK 1.2***
* Documentation page written ***ok***
* Example task declarations in the documentation tested. ***ok (used in tests)***
* Patch files generated using cvs diff -u ***to do***
* patch files include a patch to defaults.properties to register the tasks ***to do***
* patch files include a patch to tasklist.html to link to the new task page ***to do***
* Message to dev contains [SUBMIT] and task name in subject ***to do***
* Message body contains a rationale for the task ***to do***
* Message attachments contain the required files -source, documentation, test and patches zipped up to escape the HTML filter. ***to do***

### Package / Directories

This task does not depend on any external library. Therefore we can use this as a core task. This task contains only one class. So we can use the standard package for core tasks: org.apache.tools.ant.taskdefs. Implementations are in the directory src/main, tests in src/testcases and buildfiles for tests in src/etc/testcases.

Now we integrate our work into Ants distribution. So first we do an update of our cvs tree. If not done yet, you have to checkout the ant module from Apaches cvs server as described in [Access the Source Tree (AnonCVS) [7]](http://ant.apache.org/cvs.html) (password is *anoncvs*):

cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic login //1

cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic checkout ant //2

If you have a local copy of Ants sources just do an update

cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic login

cd ant //3

cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic update //4

We use the *-d* flag on **//1** to specify the cvs directory. You can specify the environment variable CVSROOT with that value and after that you havenï¿½t to use that flag any more. On **//2** we get the whole cvs tree of ant. (Sorry, but that uses a lot of time ... 10 up to 30 minutes are not unusual ... but this has to be done only once :-). A cvs update doesn't use a modulename but you have to be inside the directory. Therefore we go into that on **//3** and do the update on **//4**.

Now we will build our Ant distribution and do a test. So we can see if there are any tests failing on our machine. (We can ignore these failing tests on later steps; windows syntax used here- translate to xNIX if needed):

ANTHOME> build // 1

ANTHOME> set ANT\_HOME=%CD%\dist // 2

ANTHOME> ant test -Dtest.haltonfailure=false // 3

First we have to build our Ant distribution (**//1**). On **//2** we set the ANT\_HOME environment variable to the directory where the new created distribution is stored (%CD% is expanded to the current directory on Windows 2000 and XP, on 9x and NT write it out). On **//3** we let Ant do all the tests (which enforced a compile of all tests) without stopping on first failure.

Next we apply our work onto Ants sources. Because we haven't modified any, this is a relative simple step. *(Because I have a local copy of Ant and usually contribute my work, I work on the local copy just from the beginning. The advantage: this step isn't necessary and saves a lot of work if you modify existing source :-)*.

* move the Find.java to ANTHOME/src/main/org/apache/tools/ant/taskdefs/Find.java
* move the FindTest.java to ANTHOME/src/testcases/org/apache/tools/ant/taskdefs/FindTest.java
* move the build.xml to ANTHOME/src/etc/testcases/taskdefs/**find.xml** (!!! renamed !!!)
* add a package org.apache.tools.ant.taskdefs; at the beginning of the two java files
* delete all stuff from find.xml keeping the targets "testFileNotPresent", "testFilePresent", "test.init" and "testMultipleFiles"
* delete the dependency to "use.init" in the find.xml
* in FindTest.java change the line configureProject("build.xml"); to configureProject("src/etc/testcases/taskdefs/find.xml");
* move the find.html to ANTHOME/docs/manual/Tasks/find.html
* add a <a href="Tasks/find.html">Find</a><br> in the ANTHOME/docs/manual/tasklist.html

Now our modifications are done and we will retest it:

ANTHOME> build

ANTHOME> ant run-single-test // 1

-Dtestcase=org.apache.tools.ant.taskdefs.FindTest // 2

-Dtest.haltonfailure=false

Because we only want to test our new class, we use the target for single tests, specify the test to use and configure not to halt on the first failure - we want to see all failures of our own test (**//1 + 2**).

And ... oh, all tests fail: *Ant could not find the task or a class this task relies upon.*

Ok: in the earlier steps we told Ant to use the Find class for the <find> task (remember the <taskdef> statement in the "use.init" target). But now we want to introduce that task as a core task. And nobody wants to taskdef the javac, echo, ... So what to do? The answer is the src/main/.../taskdefs/default.properties. Here is the mapping between taskname and implementing class done. So we add afind=org.apache.tools.ant.taskdefs.Find as the last core task (just before the # optional tasks line). Now a second try:

ANTHOME> build // 1

ANTHOME> ant run-single-test

-Dtestcase=org.apache.tools.ant.taskdefs.FindTest

-Dtest.haltonfailure=false

We have to rebuild (**//1**) Ant because the test look in the %ANT\_HOME%\lib\ant.jar (more precise: on the classpath) for the properties file. And we have only modified it in the source path. So we have to rebuild that jar. But now all tests pass and we check whether our class breaks some other tests.

ANTHOME> ant test -Dtest.haltonfailure=false

Because there are a lot of tests this step requires a little bit of time. So use the *run-single-test* during development and do the *test* only at the end (maybe sometimes during development too). We use the *-Dtest.haltonfailure=false* here because there could be other tests fail and we have to look into them.

This test run should show us two things: our test will run and the number of failing tests is the same as directly after the cvs update (without our modifications).

### Apache license statement

Simply copy the license text from one the other source from the Ant source tree.

### Test on JDK 1.2

Until version 1.5 Ant must be able to run on a JDK 1.1. With version 1.6 this is not a requisite any more. But JDK 1.2 is a must-to-work-with. So we have to test that. You can download older JDKs from [Oracle [8]](http://www.oracle.com/technetwork/java/archive-139210.html).

Clean the ANT\_HOME variable, delete the *build, bootstrap* and *dist* directory and point JAVA\_HOME to the JDK 1.2 home directory. Then do thebuild, set ANT\_HOME and run ant test (like above).

Our test should pass.

### Checkstyle

There are many things we have to ensure. Indentation with 4 spaces, blanks here and there, ... (all described in the [Ant Task Guidelines [6]](http://ant.apache.org/ant_task_guidelines.html) which includes the [Sun code style [9]](http://www.oracle.com/technetwork/java/codeconvtoc-136057.html)). Because there are so many things we would be happy to have a tool for do the checks. There is one: checkstyle. Checkstyle is available at [Sourceforge [10]](http://checkstyle.sourceforge.net/) and Ant provides with the check.xml a buildfile which will do the job for us.

Download it and put the checkstyle-\*-all.jar into your %USERPROFILE%\.ant\lib directory. All jar's stored there are available to Ant so you haven't to add it to you %ANT\_HOME%\lib directory (this feature was added with Ant 1.6).

So we will run the tests with

ANTHOME> ant -f check.xml checkstyle htmlreport

I prefer the HTML report because there are lots of messages and we can navigate faster. Open the ANTHOME/build/reports/checkstyle/html/index.html and navigate to the Find.java. Now we see that there are some errors: missing whitespaces, unused imports, missing javadocs. So we have to do that.

Hint: start at the **buttom** of the file so the line numbers in the report will keep up to date and you will find the next error place much more easier without redoing the checkstyle.

After cleaning up the code according to the messages we delete the reports directory and do a second checkstyle run. Now our task isn't listed. That's fine :-)

### Publish the task

Finally we publish that archive. As described in the [Ant Task Guidelines [7]](http://ant.apache.org/ant_task_guidelines.html) we can post it on the developer mailinglist or we create a BugZilla entry. For both we need some information:

|  |  |  |
| --- | --- | --- |
| **subject** | *short description* | Task for finding files in a path |
| **body** | *more details about the path* | This new task looks inside a nested <path/> for occurrences of a file and stores all locations as a property. See the included manual for details. |
| **attachments** | *all files needed to apply the path* | Archive containing a patch with the new and modified resources |

Sending an email with these information is very easy and I think I haven't to show that. The other way - BugZilla - is slightly more difficult. But it has the advantage that entries will not be forgotten (once per week a report is generated). So I will show this way.

You must have a BugZilla account for that. So open the [BugZilla Main Page [11]](http://issues.apache.org/bugzilla/) and follow the link [Open a new Bugzilla account [12]](http://issues.apache.org/bugzilla/createaccount.cgi) and the steps described there if you haven't one.

1. From the BugZilla main page choose [Enter a new bug report [13]](http://issues.apache.org/bugzilla/enter_bug.cgi)
2. Choose "Ant" as product
3. Version is the last "Alpha (nightly)" (at this time 1.7)
4. Component is "Core tasks"
5. Platform and Severity are ok with "Other" and "Normal"
6. Initial State is ok with "New"
7. Same with the empty "Assigned to"
8. It is not required to add yourself as CC, because you are the reporter and therefore will be informed on changes
9. URL: no url required
10. Summary: add the *subject* from the table
11. Description: add the *body* from the table
12. Then press "Commit"
13. After redirecting to the new created bug entry click "Create a New Attachment"
14. Enter the path to your local path file into "File" or choose it via the "File"'s button.
15. Enter a short description into "Description", so that you could guess, what the path file includes. Here we could add "Initial Patch".
16. The "Content Type" is "auto-detect". You could use the "patch" type, if you only provide a single path file, but we want do upload more that one, included in our patch.zip.
17. Then press "Commit"

Now the new task is uploaded into the bug database.

## Resources

  [1] [tutorial-writing-tasks.html](http://ant.apache.org/manual/tutorial-writing-tasks.html)  
  [2] [tutorial-tasks-filesets-properties.zip](http://ant.apache.org/manual/tutorial-tasks-filesets-properties.zip)  
  [3] [properties.html#built-in-props](http://ant.apache.org/manual/properties.html#built-in-props)  
  [4] <http://ant-contrib.sourceforge.net/>  
  [5] [Tasks/java.html](http://ant.apache.org/manual/Tasks/java.html)  
  [6] <http://ant.apache.org/ant_task_guidelines.html>  
  [7] <http://ant.apache.org/cvs.html>  
  [8] <http://www.oracle.com/technetwork/java/archive-139210.html>  
  [9] <http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>  
  [10] <http://checkstyle.sourceforge.net/>  
  [11] <http://issues.apache.org/bugzilla/>  
  [12] <http://issues.apache.org/bugzilla/createaccount.cgi>  
  [13] <http://issues.apache.org/bugzilla/enter_bug.cgi>