Using Apache Ant

Writing a Simple Buildfile

Apache Ant's buildfiles are written in XML. Each buildfile contains one project and at least one (default) target. Targets contain task elements. Each task element of the buildfile can have an `id` attribute and can later be referred to by the value supplied to this. The value has to be unique. (For additional information, see the Tasks section below.)

Projects

A *project* has three attributes:

Attribute	Description	Required
name	the name of the project.	No
default	the default target to use when no target is supplied.	No; however, **since Ant 1.6.0**, every project includes an implicit target that contains any and all top-level tasks and/or types. This target will always be executed as part of the project's initialization, even when Ant is run with the -projecthelp option.
basedir	the base directory from which all path calculations are done. This attribute might be overridden by setting the "basedir" property beforehand. When this is done, it must be omitted in the project tag. If neither the attribute nor the property have been set, the parent directory of the buildfile will be used. A relative path is resolved relative to the directory containing the build file.	No

Optionally, a description for the project can be provided as a top-level `<description>` element (see the description type).

Each project defines one or more *targets*. A target is a set of *tasks* you want to be executed. When starting Ant, you can select which target(s) you want to have executed. When no target is given, the project's default is used.

Targets

A target can depend on other targets. You might have a target for compiling, for example, and a target for creating a distributable. You can only build a distributable when you have compiled first, so the distribute target *depends on* the compile target. Ant resolves these dependencies.

It should be noted, however, that Ant's `depends` attribute only specifies the *order* in which targets should be executed - it does not affect whether the target that specifies the dependency(s) gets executed if the dependent target(s) did not (need to) run.

More information can be found in the dedicated manual page.

Tasks

A task is a piece of code that can be executed.

A task can have multiple attributes (or arguments, if you prefer). The value of an attribute might contain references to a property. These references will be resolved before the task is executed.

Tasks have a common structure:

```
<name attribute1="value1" attribute2="value2" ... />
```

where *name* is the name of the task, *attributeN* is the attribute name, and *valueN* is the value for this attribute.

There is a set of built-in tasks, but it is also very easy to write your own.

All tasks share a task name attribute. The value of this attribute will be used in the logging messages generated by Ant.

Tasks can be assigned an `id` attribute:

```
<taskname id="taskID" ... />
```

where *taskname* is the name of the task, and *taskID* is a unique identifier for this task. You can refer to the corresponding task object in scripts or other tasks via this name. For example, in scripts you could do:

```
<script ... >
  task1.setFoo("bar");
</script>
```

to set the `foo` attribute of this particular task instance. In another task (written in Java), you can access the instance via `project.getReference("task1")`.

Note[1]: If "task1" has not been run yet, then it has not been configured (ie., no attributes have been set), and if it is going to be configured later, anything you've done to the instance may be overwritten.

Note[2]: Future versions of Ant will most likely *not* be backward-compatible with this behaviour, since there will likely be no task instances at all, only proxies.

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.

In its most simple form properties are defined in the build file (for example by the property task) or might be set outside Ant. 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`.

With Ant 1.8.0 property expansion has become much more powerful than simple key value pairs, more details can be found in the concepts section of this manual.

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>
```

Notice that we are declaring properties outside any target. As of Ant 1.6 all tasks can be declared outside targets (earlier version only allowed `<property>`,`<typedef>` and `<taskdef>`). When you do this they are evaluated before any targets are executed. Some tasks will generate build failures if they are used outside of targets as they may cause infinite loops otherwise (`<antcall>` for example).

We have given some targets descriptions; this causes the `projecthelp` invocation option to list them as public targets with the descriptions; the other target is internal and not listed.

Finally, for this target to work the source in the `src` subdirectory should be stored in a directory tree which matches the package names. Check the `<javac>` task for details.

Token Filters

A project can have a set of tokens that might be automatically expanded if found when a file is copied, when the filtering-copy behavior is selected in the tasks that support this. These might be set in the buildfile by the filter task.

Since this can potentially be a very harmful behavior, the tokens in the files **must** be of the form `@`*token*`@`, where *token* is the token name that is set in the `<filter>` task. This token syntax matches the syntax of other build systems that perform such filtering and remains sufficiently orthogonal to most programming and scripting languages, as well as with documentation systems.

Note: If a token with the format `@`*token*`@` is found in a file, but no filter is associated with that token, no changes take place; therefore, no escaping method is available - but as long as you choose appropriate names for your tokens, this should not cause problems.

Warning: If you copy binary files with filtering turned on, you can corrupt the files. This feature should be used with text files *only*.

Path-like Structures

You can specify `PATH`- and `CLASSPATH`-type references using both "`:`" and "`;`" as separator characters. Ant will convert the separator to the correct character of the current operating system.

Wherever path-like values need to be specified, a nested element can be used. This takes the general form of:

```
<classpath>
  <pathelement path="${classpath}"/>
  <pathelement location="lib/helper.jar"/>
</classpath>
```

The `location` attribute specifies a single file or directory relative to the project's base directory (or an absolute filename), while the `path` attribute accepts colon- or semicolon-separated lists of locations. The `path` attribute is intended to be used with predefined paths - in any other case, multiple elements with `location` attributes should be preferred.

Since Ant 1.8.2 the location attribute can also contain a wildcard in its last path component (i.e. it can end in a "*") in order to support wildcard CLASSPATHs introduced with Java6. Ant will not expand or evaluate the wildcards and the resulting path may not work as anything else but a CLASSPATH - or even as a CLASSPATH for a Java VM prior to Java6.

As a shortcut, the `<classpath>` tag supports `path` and `location` attributes of its own, so:

```
<classpath>
  <pathelement path="${classpath}"/>
</classpath>
```

can be abbreviated to:

```
<classpath path="${classpath}"/>
```

In addition, one or more Resource Collections can be specified as nested elements (these must consist of file-type resources only). Additionally, it should be noted that although resource collections are processed in the order encountered, certain resource collection types such as fileset, dirset and files are undefined in terms of order.

```
<classpath>
  <pathelement path="${classpath}"/>
  <fileset dir="lib">
```

```
      <include name="**/*.jar"/>
    </fileset>
    <pathelement location="classes"/>
    <dirset dir="${build.dir}">
      <include name="apps/**/classes"/>
      <exclude name="apps/**/*Test*"/>
    </dirset>
    <filelist refid="third-party_jars"/>
  </classpath>
```

This builds a path that holds the value of `${classpath}`, followed by all jar files in the `lib` directory, the `classes` directory, all directories named `classes` under the `apps` subdirectory of `${build.dir}`, except those that have the text `Test` in their name, and the files specified in the referenced FileList.

If you want to use the same path-like structure for several tasks, you can define them with a `<path>` element at the same level as *target*s, and reference them via their *id* attribute--see References for an example.

By default a path like structure will re-evaluate all nested resource collections whenever it is used, which may lead to unnecessary re-scanning of the filesystem. Since Ant 1.8.0 path has an optional *cache* attribute, if it is set to true, the path instance will only scan its nested resource collections once and assume it doesn't change during the build anymore (the default for *cache* still is *false*). Even if you are using the path only in a single task it may improve overall performance to set *cache* to *true* if you are using complex nested constructs.

A path-like structure can include a reference to another path-like structure (a path being itself a resource collection) via nested `<path>` elements:

```
  <path id="base.path">
    <pathelement path="${classpath}"/>
    <fileset dir="lib">
      <include name="**/*.jar"/>
    </fileset>
    <pathelement location="classes"/>
  </path>

  <path id="tests.path" cache="true">
    <path refid="base.path"/>
    <pathelement location="testclasses"/>
  </path>
```

The shortcuts previously mentioned for `<classpath>` are also valid for `<path>`.For example:

```
  <path id="base.path">
    <pathelement path="${classpath}"/>
  </path>
```

can be written as:

```
  <path id="base.path" path="${classpath}"/>
```

Path Shortcut

In Ant 1.6 a shortcut for converting paths to OS specific strings in properties has been added. One can use the expression ${toString:*pathreference*} to convert a path element reference to a string that can be used for a path argument. For example:

```
  <path id="lib.path.ref">
    <fileset dir="lib" includes="*.jar"/>
  </path>
  <javac srcdir="src" destdir="classes">
```

```
    <compilerarg arg="-Xbootclasspath/p:${toString:lib.path.ref}"/>
  </javac>
```

Command-line Arguments

Several tasks take arguments that will be passed to another process on the command line. To make it easier to specify arguments that contain space characters, nested `arg` elements can be used.

Attribute	Description	Required
value	a single command-line argument; can contain space characters.	Exactly one of these.
file	The name of a file as a single command-line argument; will be replaced with the absolute filename of the file.	
path	A string that will be treated as a path-like string as a single command-line argument; you can use `;` or `:` as path separators and Ant will convert it to the platform's local conventions.	
pathref	Reference to a path defined elsewhere. Ant will convert it to the platform's local conventions.	
line	a space-delimited list of command-line arguments.	
prefix	A fixed string to be placed in front of the argument. In the case of a line broken into parts, it will be placed in front of every part. *Since Ant 1.8.*	No
suffix	A fixed string to be placed immediately after the argument. In the case of a line broken into parts, it will be placed after every part. *Since Ant 1.8.*	No

It is highly recommended to avoid the `line` version when possible. Ant will try to split the command line in a way similar to what a (Unix) shell would do, but may create something that is very different from what you expect under some circumstances.

Examples

```
  <arg value="-l -a"/>
```

is a single command-line argument containing a space character, *not* separate commands "-l" and "-a".

```
  <arg line="-l -a"/>
```

This is a command line with two separate arguments, "-l" and "-a".

```
  <arg path="/dir;/dir2:\dir3"/>
```

is a single command-line argument with the value `\dir;\dir2;\dir3` on DOS-based systems and `/dir:/dir2:/dir3` on Unix-like systems.

References

Any project element can be assigned an identifier using its `id` attribute. In most cases the element can subsequently be referenced by specifying the `refid` attribute on an element of the same type. This can be useful if you are going to replicate the same snippet of XML over and over again--using a `<classpath>` structure more than once, for example.

The following example:

```
<project ... >
  <target ... >
    <rmic ...>
      <classpath>
        <pathelement location="lib/"/>
        <pathelement path="${java.class.path}/"/>
        <pathelement path="${additional.path}"/>
      </classpath>
    </rmic>
  </target>

  <target ... >
    <javac ...>
      <classpath>
        <pathelement location="lib/"/>
        <pathelement path="${java.class.path}/"/>
        <pathelement path="${additional.path}"/>
      </classpath>
    </javac>
  </target>
</project>
```

could be rewritten as:

```
<project ... >
  <path id="project.class.path">
    <pathelement location="lib/"/>
    <pathelement path="${java.class.path}/"/>
    <pathelement path="${additional.path}"/>
  </path>

  <target ... >
    <rmic ...>
      <classpath refid="project.class.path"/>
    </rmic>
  </target>

  <target ... >
    <javac ...>
      <classpath refid="project.class.path"/>
    </javac>
  </target>
</project>
```

All tasks that use nested elements for PatternSets, FileSets, ZipFileSets or path-like structures accept references to these structures as shown in the examples. Using `refid` on a task will ordinarily have the same effect (referencing a task already declared), but the user should be aware that the interpretation of this attribute is dependent on the implementation of the element upon which it is specified. Some tasks (the property task is a handy example) deliberately assign a different meaning to `refid`.

Use of external tasks

Ant supports a plugin mechanism for using third party tasks. For using them you have to do two steps:

1. place their implementation somewhere where Ant can find them
2. declare them.

Don't add anything to the CLASSPATH environment variable - this is often the reason for very obscure errors. Use Ant's own mechanisms for adding libraries:

- via command line argument `-lib`
- adding to `${user.home}/.ant/lib`
- adding to `${ant.home}/lib`

For the declaration there are several ways:

- declare a single task per using instruction using `<taskdef name="taskname" classname="ImplementationClass"/>`
  `<taskdef name="for" classname="net.sf.antcontrib.logic.For" /> <for ... />`
- declare a bundle of tasks using a properties-file holding these taskname-ImplementationClass-pairs and `<taskdef>`
  `<taskdef resource="net/sf/antcontrib/antcontrib.properties" /> <for ... />`
- declare a bundle of tasks using a xml-file holding these taskname-ImplementationClass-pairs and `<taskdef>`
  `<taskdef resource="net/sf/antcontrib/antlib.xml" /> <for ... />`
- declare a bundle of tasks using a xml-file named antlib.xml, XML-namespace and `antlib:` protocoll handler
  `<project xmlns:ac="antlib:net.sf.antconrib"/> <ac:for ... />`

If you need a special function, you should

1. have a look at this manual, because Ant provides lot of tasks
2. have a look at the external task page in the manual (or better online)
3. have a look at the external task wiki page
4. ask on the Ant user list
5. implement (and share) your own