Phing 2 – User Guide

Andreas Aderhold, Alex Black, Manuel Holtgrewe, Hans Lellelid

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

About this book

Authors

- Andreas Aderhold, andi@binarycloud.com
- Alex Black, enigma@turingstudio.com
- Manuel Holtgrewe, grin@gmx.net
- Hans Lellelid, hans@xmpl.org

Because Phing is based on Ant, parts of this manual are also adapted from the Ant manual (see [ant]).

CVS

$Id: AboutThisBook.html,v 1.3 2003/10/20 18:15:25 hlellelid Exp $

$Revision: 1.3 $

Copyright

License

See the GNU FDL ([gnu−fdl], which is included in this book.

Introduction

What Phing Is

Phing is a project build system based on Apache ant [ant]. You can do anything with Phing that you could do with a traditional build system like Gnu make [gnumake], and Phing's use of simple XML build files and extensible PHP "task" classes make it an easy−to−use and highly flexible build framework.

Because Phing is based on Ant, parts of this manual are also adapted from the ant manual (see [ant]). We are extremely grateful to the folks in the Ant project for creating (and continuing to create) such an inspiring build system model, and for the open−source licensing that makes it possible for us to learn from each other and build increasingly better tools.

Phing & Binarycloud: History

Phing was originally a subproject of Binarycloud. Binarycloud is a highly engineered application framework, designed for use in enterprise environments. Binarycloud uses XML extensively for storing metadata about a project (configuration, nodes, widgets, site structure, etc.). Because Binarycloud is built for PHP, performing extensive XML processing and transformations on each page request is an unrealistic proposition. Phing is used to "compile" the XML metadata into PHP arrays that can be processed without overhead by PHP scripts.

Of course, XML "compilation" is only one of many ways that Binarycloud uses the Phing build system. The Phing build system makes it possible for you to:

- Build multi language pages from one source tree (see binarycloud documentation and [below in this document]),
- Centralize metadata (e.g. your data model) in one XML file and generate several files from that XML with different XSLT.

It the beginning, Binarycloud used the GNU make system; however, this approach had some drawbacks: The "space−before−tab"−problem in makefiles, the fact that it is only natively available for Unix systems etc. So, the need for a better build system arose. Apache Ant was a logical choice −− do to its use of XML build files and modular design. The problem was that Ant is written in Java, so you need to install a JVM on your computer to use it. Besides the need for yet another interpreter (i.e. besides PHP), there was also legal/ideological conflict in requiring a commercial JVM (there were problems with Ant on JVMs other than Sun's) for an LGPL'd Binarycloud.

So, the development of Phing began. Phing is a build system written in PHP and uses the ideas of Ant. The first release was designed & developed simultaneously, and thus not very sophisticated. This original system was quickly pushed to its limits and the need for a better Phing became a priority. Andreas Anderhold, who was responsible for Phing/r1, designed and wrote much of the Phing/r2 that followed. Phing/r2 became the Phing−1.0 that exists today for PHP4.

The current development of Phing is focused on Phing 2, which has involved a number of bugfixes, functionality enhancements, and most significantly a conversion of the codebase to use new PHP5 features such as abstract classes, interfaces, and try/catch/throw exception handling.

How Phing Works

Phing uses XML *buildfiles* that contain a description of the things to do. The buildfile is structured into targets that contain the actual commands to perform (e.g. commands to copy a file, delete a directory, perform a DB query, etc.). So, to use Phing, you would first write your buildfile and then you would run phing, specifying the target in your buildfile that you want to execute.

```
% phing −f mybuildfile.xml mytarget
```

By default Phing will look for a buildfile named *build.xml* (so you don't have to specify the buildfile name unless it is not *build.xml*) and if no target is specified Phing will try to execute the default target, as specified in the <project> tag.

Cool, so how can I help?

Phing is under active development and there are many things to be done. To get involved, you can start by doing the following:

- Read this manual to understand Phing ;−)
- Go to http://phing.tigris.org and subscribe to the "dev" mailing list

Setting−Up Phing

The goal of this chapter is to help you obtain and correctly setup and execute Phing on your operating system. Once you setup Phing properly you shouldn't need to revisit this chapter, unless you're re−installing or moving your installation to another platform.

System Requirements

To use Phing you must have installed PHP version 5.0.0b1 or above compiled −−with−libxml2, as well as −−with−xsl if you want to make use of advanced functionality. At the time of writing PHP5.0.0b2−dev is currently unable to run Phing due to segmentation faults arising somewhere in the XML parsing of the build file.

For more information on PHP and the required modules see the PHP [php] website. For a brief list of software dependencies see below.

Operating Systems

Designed for portability from the get go, Phing runs on all platforms that run PHP. However some advanced funcionality may not work properly or is simply ignored on some platforms (i.e. chmod on the Windows platform).

To get the most out of Phing, a Unix style platform is recommended. Namely: Linux, FreeBSD, OpenBSD, etc.

Software Dependencies

For a detailed list of required sofware and libraries, refer to the following table of Software dependencies

Software Dependencies

Software	Required for	Source
PHP 5.0.0b2	Execution	http://www.php.net

Obtaining Phing

Phing is free software distributed under the terms of the LGPL.

Distribution Files

There are several ways to get a Phing distribution package. If you do not want to participate in developing Phing itself it is recommended that you get the latest snapshot or stable packaged distribution. If you are interested in helping with Phing development, get a CVS revision of the files, as described below.

The easiest way to obtain the distribution package is to visit the Phing website [phing] and download the current distribution package in the format you desire.

As of version 2.0.0b1, you have the option of downloading a PEAR−installable package or the full phing distribution. If you wish to modify phing we suggestion downloading the full version so that you can (e.g.) create your own PEAR package. If you simply wish to use Phing for your own project or need it to build another package, download & install the PEAR package.

Getting A Development Copy From CVS

You are encouraged to contibute to the development of Phing. If you want to participate in Phing development or if you are simply intereseted in the latest features and development, obtain a CVS copy as described in the following paragraphs.

The CVS revisions of Phing are not bullet−proof and may fail to execute properly on your machine. Only obtain the CVS versions if you are absolutely aware of limitations and constraints of such an action. Additionally you should sign up to the development mailinglist to report and notice errors and incompatibilities.

We assume that you are running a Unix style operating system. So we expect the CVS software is installed ant the `cvs` executable is in your system's search path.. However, the steps for a Windows based system are very similar. There is plenty additional resources available on how to use CVS [cvs−howto] and on CVS speciaities on the Tigris project management platfrom [cvs−tigris].

The frist thing you have to do is log onto the CVS server. Therefore type the following line at your command promt:

```
 cvs −d :pserver:guest@cvs.tigris.org:/cvs login
```

If you signed−up as developer at the Tigris website, login with your own password, the same as the one you use to access the site. Or use the password "guest" with username "guest".

If this is the only project you working on, you only need to set the cvsroot once. Thereafter when you log in to this domain, the cvs repository for this project is assumed. If you are working multiple projects, however, you must specify the cvsroot each time you log in to ensure that the cvs repository you are accessing is the right one.

To check out the project source repository an individual module (if you don't need the entire repository), type:

```
cvs −d :pserver:guest@cvs.tigris.org:/cvs checkout phing
```

This command should result in a scrolling list of files being added to the local directory you specified on your machine. Now you are ready to use your favourite file editor to work with individual files and make changes.

The top−level CVS module contains the Phing project website in addition to the phing application; the Phing files, therefore, reside in the phing/ subdirectory:

```
./phing/phing
```

PEAR Install

The easiest way to install Phing is using the PEAR installer. Provided that the package you downloaded is a the PEAR−ready package, you can install Phing simply from the command line (Unix or Windows):

```
$> pear install phing−2.0.0b1−pear.tar.gz
```

The pear installer will check any dependencies and place the phing script (phing or phing.bat) into your PHP script directoy (i.e. where the "pear" script resides).

Non−PEAR Install

If you are not using the PEAR installer, you will need to setup your environment in order to run Phing. The distribution of Phing consists of three directories: *bin* , *docs* and *classes* . Only the *bin* and *classes* directories are required to run Phing. To install Phing, choose a directory and uncompress the distribution file in that directory (you may already have done this in a prior step). This directory will be known as *PHING_HOME* .

On Windows 95 and Windows 98, the script used to launch Phing will have problems if PHING_HOME is a long filepath. This is due to limitations in the OS's handling of the "for" batch−file statement. It is recommended, therefore, that Phing be installed in a short path, such as C:\opt\phing.

Before you can run Phing there is some additional set up you will need to do perform:

- Add the full path to the *bin/* directory to your path.
- Set the *PHING_HOME* environment variable to the directory where you installed Phing. On some operating systems the Phing wrapper scripts can guess PHING_HOME (Unix dialects and Windows NT/2000). However, it is better to not rely on this behavior.
- Set the *PHP_COMMAND* environment variable to where your Php binary is located (including the binary i.e. PHP_COMMAND=/usr/bin/php).
- Set the *PHP_CLASSPATH* environment variable (see the section below). This should be set at least point to PHING_HOME/classes. Alternatively, you can also just add the phing/classes directory to your PHP include_path ini setting.
- Check your php.ini file to make sure that you have the following settings:
    - ♦ max_execution_time = 0 *// unlimited execution time*
    - ♦ memory_limit = 32M *// you may need more memory depending on size of your build files*

If you are using Phing in conjunction with another application, you may need to add additional paths to PHP_CLASSPATH.

Unix

Assuming you are running a Unix dialect operating system with the *bash bourne shell* and Phing is installed in `/opt/phing` . The following sets up the environment properly:

```
  export PHP_COMMAND=/usr/bin/php
  export PHING_HOME=/opt/phing
  export PHP_CLASSPATH=${PHING_HOME}/classes
  export PATH=${PATH}:${PHING_HOME}/bin
```

Windows

On the Windows platfrom, assuming Phing is installed in c:\opt\phing. The following sets up your environment:

```
  set PHP_COMMAND=c:\opt\php\php.exe
  set PHING_HOME=c:\opt\phing
  set PHP_CLASSPATH=c:\opt\phing\classes
  set PATH=%PATH%;%PHING_HOME%\bin
```

Advanced

There are lots of variants that can be used to run/prepare Phing. You need at least the following:

- If you want Phing to be able to use other packages / classes, you can either add them to the PHP_CLASSPATH or to PHP's include_path.
- Some Tasks in phing/tasks/ext may require 3rd party libraries to be installed. Generally, tools with compatible license (and stable releases) are included in phing/lib so that outside dependencies can be avoided. PEAR libs will not, however, be bundled with Phing since they are generally bundled with PHP. If you are using a 3rd party task, see the Task documentation to be aware of any dependencies.

You are now ready to use the `phing` command at your command prompt, from everywhere in your directory tree.

Calling Phing

Now you are prepared to execute Phing on the command line or via script files. The following section briefly describe how to properly execute phing.

Command Line

Phing execution on the command line is simple. Just change to the directory where your buildfile resides and type

```
$ phing [targetname]
```

at the command line (where [targetname] is the target you want to be executed). Optionally, you may specify command line arguments as listed in Appendix A.

Getting Started

Phing buildfiles are written in XML, and so you will need to know at least some basic things about XML to understand the following chapter. There is a lot of information available on the web:

- The Standard Recommendation of XML by the w3c http://www.w3.org/TR/2000/REC−xml: very technical but exhaustive.
- *XML In 10 Points* http://www.w3.org/XML/1999/XML−in−10−points: Quick introduction into XML.
- *A technical introduction to XML* http://www.xml.com/pub/a/98/10/guide0.html: Interesting article by the creator of DocBook.

XML And Phing

A valid Phing buildfile has the following basic structure:

- The document prolog
- Exactly one root element called *<project>* .
- Several Phing *type* elements (i.e. *<property>* , *<fileset>* , *<patternset>* etc.)
- One or more *<target>* elements containing built−in or user defined Phing *task* elements (i.e. *<install>* , *<bcc>* , etc).

Writing A Simple Buildfile

The Foobar project installs some PHP files from a source location to a target location, creates an archive of this files and provides an optional clean−up of the build tree:

```
<?xml version="1.0"?>

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

  <target name="prepare">
    <echo msg="Preparing build..." />
    <mkdir dir="./build" />
  </target>

  <target name="build" depends="prepare">
    <echo>Building...</echo>
    <copy file="./src/File.php" to="./build/File.php"/>
    <copy file="./src/File2.php" to="./build/File2.php"/>
  </target>

  <target name="dist" depends="build">
    <echo message="Creating archive..." />
    <tar outfile="furbee.tar.gz" basedir="./build"/>
  </target>

  <target name="clean">
    <echo msg="Cleaning up..."/>
    <delete file="./build"/>
  </target>
</project>
```

Project Element

The first element after the document prolog is the root element named *<project>* on line 3. This element is a container for all other elements and can/must have the following attributes:

<project> Attributes

Attribute	Meaning	Required
name	The name of the project	No
basedir	The base directory of the project. If none is specified, the current directory is used.	No
default	The default target that is to be executed if no target(s) are specified when calling this build file.	Yes
description	The description of the project.	Yes

Target Element

A target can *depend* on other targets. You might have a target for installing the files in the build tree, for example, and a target for creating a distributable tar.gz archive. You can only build a distributable when you have installed the files first, so the distribute target depends on the install target. Phing resolves these dependencies.

It should be noted, however, that Phing'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.

Phing tries to execute the targets in the depends attribute in the order they appear (from left to right). Keep in mind that it is possible that a target can get executed earlier when an earlier target depends on it, in this case the dependant is only executed once:

```
    <target name="D" depends="C,B,A" />
```

Suppose we want to execute target `D` . From its depends attribute, you might think that first target `C` , then `B` and then `A` is executed. Wrong! `C` depends on `B` , and `B` depends on `A` , so first `A` is executed, then `B` , then `C` , and finally `D` .

A target gets executed only once, even when more than one target depends on it (see the previous example).

The optional description attribute can be used to provide a one−line description of this target, which is printed by the `−projecthelp` command−line option.

Target attributes

You can specify one or more of the following attributes within the target element.

<target> Attributes

Attribute	Meaning	Required
name	The name of the target	Yes
depends	A comma−seperated list of targets this target depends on.	No
if	The name of the *Property* that hast to be set in order for this target to be executed	No
unless	The name of the *Property* that must *not* be set in order for this target to be executed.	

Task Elements

A *task* is a piece of PHP code that can be executed. This code implements a particular action to perform (i.e. install a file). Therefore it must be defined in the buildfile so that it is actually invoked by Phing.

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 core tasks (see Appendix B) along with a number of optional tasks. It is also very easy to write your own tasks (see Extending Phing).

Tasks can be assigned an *id* attribute:

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

By doing this you can refer to specific tasks later on in the code of other tasks.

Property Element

Properties are essentially variables that can be used in the buildfile. These might be set in the buildfile by calling the PropertyTask, or might be set outside Phing on the command line (properties set on the command always line override the ones in the buildfile). A property has a name and a value only. Properties may be used in the value of task attributes. This is done by placing the property name between " `${` " and " `}` " in the attribute value. For example, if there is a BC_BUILD_DIR property with the value 'build', then this could be used in an attribute like this: *${BC_BUILD_DIR}/en* . This is resolved to *build/en*.

It should be noted that if you use a property that is not defined via the property task, the system environment table is searched for this name. For example, if you would use the property `${BCHOME}` and you did not define this prior in the buildfile or at the command line, Phing uses the environment variable `BCHOME` if it exists.

Built−in Properties

Phing provides access to system properties as if they had been defined using a *<property>* task. For example, *${os.name}* expands to the name of the operating system. See Appendix A for a complete list

More Complex Buildfile

```
<?xml version="1.0" ?>

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

  <property environment="env"/>

  <property file="${env.BCHOME}/build.properties"/>

  <property name="package"  value="${phing.project.name}" override="true" />
  <property name="builddir" value="${env.BCHOME}/build/testsite" override="true" />
  <property name="srcdir"   value="${project.basedir}" override="true" />

  <!-- Fileset for all files -->
  <fileset dir="." id="allfiles">
    <include name="**">
  </fileset>

  <!-- Main Target -->
  <target name="main" description="main target">
    <copy todir="${builddir}">
      <fileset refid="allfiles" />
    </copy>
  </target>


  <!-- Rebuild -->
  <target name="rebuild" description="rebuilds this package">
    <delete dir="${builddir}" />
    <phingcall target="main"/>
  </target>
</project>
```

This build file first defines some properties with the *<property>* task call to *PropertyTask*. Then, it defines a fileset and two targets. Let us have a quick rundown of this build file.

The first five tags within the *project* tag define properties. They appear in the three ways this tag can occur:

- The first occurrence of the *property* tag only contains the *environment* tag. This way, it makes all environment variables available as "*env.VARIABLE_NAME*".
- The second *property* tag contains only the *file* attribute. The value has to be a relative or absolute path to a property file (for the format, see Appendix E).

- The other times, the tag has a *name* and a *value* attribute. After the call, the value defined in the attribute *value* is available through the key enclosed in "${" and "}".

The next noticeable thing in the build file is the <*fileset*> tag. It defines a fileset, i.e. a set of multiple files. You can include and exclude Files with *include* and *exclude* tags within the *fileset* tag. For more information concerning Filesets (i.e. Patterns) see Appendix C. The fileset is given an *id* attribute, so it can be referenced later on.

The first task only contains a call to *CopyTask* via <*copy*>. The interesting thing is within the *copy* tag. Here, a fileset task is not written out with nested *include* or *exclude* elements, but via the *refid*, the earlier create Fileset is referenced. This way, you can use a once defined fileset multiple times in your build files.

The only noticeable thing in the second target is the call to *PhingTask* with the <*phingcall*> tag (see Appendix B for more information. The task executes a specified target within the same build file. So, the second target removes the build directory and calls *main* anew so the project is rebuilt.

Project Components

This goal of this chapter is to make you familiar with the basic components of a buildfile. After reading this chapter, you should be able to read and understand the basic structure of any buildfile even if you don't know exactly what the individual pieces do.

For supplemental reference information, you should see Appendix B, Appendix C and Appendix D.

Projects

In the structure of a Phing buildfile, there must be exactly one *Project* defined; the *<project>* tag is the root element of the buildfile, meaning that everything else in the buildfile is contained within the *<project >* element.

```
<?xml version="1.0"?>

<project name="test" description="Simple test build file" default="main" >
  <!-- Everything else here -->
<project>
```

The listing above shows a sample *<project>* tag that has all attributes available for Projects. The *name* and *description* attributes are fairly self−explanatory; the *default* attribute specifies the default *Target* to execute if no target is specified (Targets are described below). For a complete reference, see Appendix D.

Project Components in General

Project components are everything you can be find inside a project. So Targets are project components, as are Tasks, Types, etc. Project components may have attributes and nested tags. Attributes only contain simple values, i.e. strings, integers etc. Nested elements may be complex Phing types (like FileSets) or simple wrapper classes for values with custom keys (see FileSet for example).

Any nested elements must be supported by the class that implements the project component, and because the nested tags are handled by the project component class the same nested tag may have different meanings (and different attributes) depending on the context. So, for example, the nested tag *<param.../>* within the *<phingcall>* tag is handled very differently from the*<param.../>* tag within the *<xsltfilter>* tag −− in the first case setting project properties, in the second case setting XSLT parameters.

Targets

Targets are collections of project components (but not other targets) that are assigned a unique name within their project. A target generally performs a specific task −− or calls other targets that perform specific tasks −− and therefore a target is a bit like a *function* (but a target has no return value).

Targets may *depend* on other targets. For example, if target A depends on a target B, then when target A is called to be executed, target B will be executed first. Phing automatically resolves these dependencies. You cannot have circular references like: "target A depends on target B that depends on target A".

The following code snippet shows an example of the use of targets.

```
<target name="othertask" depends="buildpage" description="Whatever">
  <!-- Task calls here -->
<target>

<target name="buildpage" description="Some description">
  <!-- Task calls here -->
<target>
```

When Phing is asked to execute the *othertask* target, it will see the dependency and execute *buildpage* first. Notice that the the dependency task can be defined after the dependent task.

Tasks

Tasks are responsible for doing the work in Phing. Basically, tasks are the individual actions that your buildfile can perform. For example, tasks exist to copy a file, create a directory, TAR files in a directory. Tasks may also be more complex such as XsltTask which copies a file and transforms the file using XSLT, SmartyTask which does something similar using Smarty templates, or CreoleTask which executes SQL statements against a specified DB. See Appendix B for descriptions of Phing tasks.

Tasks support parameters in the form of:

- Simple parameters (i.e. strings) passed as XML attributes, or
- More complex parameters that are passed by nested tags

Simple parameters are basically strings. For example, if you pass a value *"A simple string."* as a parameter, it is evaluated as a string and accessible as one. You can also reference properties as described in Getting Started.

Note: There are special values that are not mapped to strings, but to boolean values instead. The values *true*, *false*, *yes*, *no*, *on* and *off* are translated to true/false boolean values.

```
<property name="myprop" value="value" override="true"/>
```

However, some tasks support more complex data types as parameters. These are passed to the task with *nested tags*. Consider the following example:

```
<copy>
  <fileset dir=".">
    <include name="**" />
  </fileset>
</copy>
```

Here, *CopyTask* is passed a complex parameter, a Fileset. Tasks may support multiple complex types in addition to simple parameters. Note that the names of the nested tags used to create the complex types depend on the task implementation. Tasks may support default Phing types (see below) or may introduce other types, for example to wrap key/value pairs.

Refer to Appendix B for a list of system tasks and their parameters.

Types

Basics

Besides the simple types (strings, integer, booleans) you can use in the parameters of tasks, there are more complex Phing *Types*. As mentioned above, they are passed to a task by using nesting tags:

```
<task>
  <type />
</task>

<!-- or: -->

<task>
  <type1>
    <subtype1>
      <!-- etc. -->
    </subtype1>
  </type1>
</task>
```

Note that types may consist of multiple nested tags –– and multiple levels of nested tags, as you can see in the second task call above.

Referencing Types

An additional fact about types you should notice is the possibility of *referencing* type instances, i.e. you define your type somewhere in your build file and assign an id to it. Later, you can refer to that type by the id you assigned. Example:

```
<project>
  <fileset id="foo">
    <include name="*.php" />
  </fileset;>

  <!-- Target that uses the type -->
  <target name="foo" >
    <copy todir="/tmp">
      <fileset refid="foo" />
    </copy>
  </target>
</project>
```

As you can see, the type instance is assigned an id with the *id* attribute and later on called by passing a plain *fileset* tag to *CopyTask* that only contains the *refid* attribute.

Basic Types

The following section gives you a quick introduction into the basic Phing types. For a complete reference see Appendix C.

FileSet

FileSets are groups of files. You can include or exclude specific files and patterns to/from a FileSet. The use of patterns is explained below. For a start, look at the following example:

```
<fileset dir="/tmp" id="fileset1">
  <include name="sometemp/file.txt" />
  <include name="othertemp/**" />
  <exclude name="othertemp/file.txt" />
</fileset>

<fileset dir="/home" id="fileset2">
  <include name="foo/**" />
  <include name="bar/**/*.php" />
  <exclude name="foo/tmp/**" />
</fileset>
```

The use of patterns is quite straightforward: If you simply want to match a part of a filename or dirname, you use *****. If you want to include multiple directories and/or files, you use ******. This way, filesets provide an easy but powerful way to include files.

FileList

FileLists, like FileSets, are collections of files; however, a FileList is an explicitly defined list of files −− and the files don't necessarily have to exist on the filesystem.

Besides being able to refer to nonexistent files, another thing that FileLists allow you to do is specify files *in a certain order*. Files in FileSets are ordered based on the OS−level directory listing functions, in some cases you may want to specify a list of files to be processed in a certain order −− e.g. when concatenating files using the *<append>* task.

```
<filelist dir="base/" files="file1.txt,file2.txt,file3.txt"/>

<!−− OR: −−>
<filelist dir="basedir/" listfile="files_to_process.txt"/>
```

FilterChains and Filters

FilterChains can be compared to Unix pipes. Unix pipes add a great deal of flexibility to command line operations; for example, if you wanted to copy just those lines that contained the string *blee* from the first 10 lines of a file called *foo* to a file called *bar*, you could do:

```
cat foo | head −n10 | grep blee > bar
```

Something like this is not possible with the tasks and types that we have learned about thus far, and this is where the incredible usefulness of *FilterChains* becomes apparent. They emulate Unix pipes and provide a powerful dimension of file/stream manipulation for the tasks that support them.

FilterChain usage is quite straightforward: you pass the complex Phing type *filterchain* to a task that supports FilterChains and add individual filters to the FilterChain. In the course of executing the task, the filters are applied (in the order in which they appear in the XML) to the contents of the files that are being manipulated by your task.

```
<filterchain>
  <replacetokens>
    <token key="BC_PATH" value="${top.builddir}/"/>
    <token key="BC_PATH_USER" value="${top.builddir}/testsite/user/${lang}/"/>
  </replacetokens>

  <filterreader classname="phing.filters.TailFilter">
    <param name="lines" value="10"/>
  </filterreader>
</filterchain>
```

The code listing above shows you some example of how to use filter chains. For a complete reference see Appendix C. This filter chain would replace all occurences of *BC_PATH* and *BC_PATH_USER* with the values assigned to them in lines 4 and 5. Additionally, it will only return the last 10 lines of the files.

Notice above that FilterChain filters have a "shorthand" notation and a long, generic notation. Most filters can be described using both of these forms:

```
<replacetokens>
  <token key="BC_PATH" value="${top.builddir}/"/>
  <token key="BC_PATH_USER" value="${top.builddir}/testsite/user/${lang}/"/>
</replacetokens>

<!-- OR: -->

<filterreader classname="phing.filters.ReplaceTokens">
  <param type="token" name="BC_PATH" value="${top.builddir}/"/>
  <param type="token" name="BC_PATH" value="${top.builddir}/testsite/user/${lang}/"/>
</filterreader>
```

As the pipe concept in Unix, the filter concept is quite complex but powerful. To get a better understanding of different filters and how they can be used, take a look at any of the many uses of FilterChains in the build files for the binarycloud [bc] project.

File Mappers

With FilterChains and filters provide a powerful tool for changing *contents* of files, Mappers provide a powerful tool for changing the *names* of files.

To use a Mapper, you must specify a pattern to match on and a replacement pattern that describes how the matched pattern should be transformed. The simplest form is basically no different from the DOS *copy* command:

```
copy *.bat *.txt
```

In Phing this is the *glob* Mapper:

```
<mapper type="glob" from="*.bat" to="*.txt"/>
```

Phing also provides support for more complex mapping using regular expressions:

```
<mapper type="regexp" from="^(.*)\.conf\.xml$$" to="\1.php"/>
```

Consider the example below to see how Mappers can be used in a build file. This example includes some of the other concepts introduced in this chapter, such as FilterChains and FileSets. If you don't understand everything, don't worry. The important point is that Mappers are types too, which can be used in tasks that support them.

```
<copy>
  <fileset dir=".">
    <include name="*.ent.xml" />
  </fileset>

  <mapper type="regexp" from="^(.*)\.ent\.xml$" to="\1.php"/>

  <filterchain>
    <filterreader classname="phing.filters.XsltFilter">
      <param name="style" value="ent2php.xsl" />
    </filterreader>
  </filterchain>
</copy>
```

For a complete reference, see Appendix C.

Extending Phing

Phing was designed to be flexible and easily extensible. Phing's existing core and optional tasks do provide a great deal of flexibility in processing files, performing database actions, and even getting user feedback during a build process. In some cases, however, the existing tasks just won't suffice and because of Phing's open, modular architecture adding exactly the functionality you need is often quite trivial.

In this chapter we'll look primarily at how to create your own tasks, since that is probably the most useful way to extend Phing. We'll also give some more information about Phing's design and inner workings.

Extension Possibilities

There are three main areas where Phing can be extended: tasks, types, mappers. The following sections discuss these options.

Tasks

Tasks are pieces of codes that perform an atomic action like installing a file. Therefore a special worker class hast to be created and stored in a specific location, that actually implements the job. The worker is just the interface to Phing that must fulfill some requirements discussed later in this chapter, however it can − but not necessarily must − use other classes, workers and libraries that aid performing the operations needed.

Types

Extending types is a rare need; nevertheless, you can do it. A possible type you might implement is *urlset*, for example.

You may end up needing a new type for a task you write; for example, if you were writing the XSLTTask you might discover that you needed a special type for XSLTParams (even though in that case you could probably use the generic name/value Parameter type). In cases where the type is really only for a single task, you may want to just define the type class in the same file as the Task class, rather than creating an official stand−alone *Type*.

Mappers

Creating new mappers is also a rare need, since most everything can be handled by the RegexpMapper. The Mapper framework does provide a simple way for defining your own mappers to use instead, however, and mappers implement a very simple interface.

Source Layout

Files And Directories

Before you are going to start to extend Phing let's have a look at the source layout. You should be comfortable with the organization of files witchin the source tree of Phing before start coding. After you extracted the source distribution or checked it out from CVS you should see the following directory structure:

```
$PHING_HOME
  |-- bin
  |-- classes
  |    `-- phing
  |         |-- filters
  |         |    `-- util
  |         |-- mappers
  |         |-- parser
  |         |-- tasks
  |         |    |-- ext
  |         |    |-- system
  |         |    |    `-- condition
  |         |    `-- user
  |         `-- types
  |-- docs
  |    `-- phing_guide
  `-- test
       |-- classes
           `-- etc
```

The following table briefly describes the contents of the major directories:

Phing source tree directories

Directory	Contents
bin	The basic applications (phing, configure) as well as the wrapper scripts for different platforms (currently Unix and Windows).
classes	Repository of all the classes used by Phing. This is the base directory that should be on the PHP include_path. In this directory you will find the subdirectory phing/ with all the Phing relevant classes.
docs	Documentation files. Generated books, online manuals as well as the PHPDoc generated API documentation.
test	A set of testcases for different tasks, mappers and types. If you are developing in CVS you should add a testcase for each implementation you check in.

Currently there is no distinction between the *source* layout and the *build* layout of Phing. The figure above shows the CVS tree that carries some additional files like the Phing website. Later on there may be a buildfile to create a clean distribution tree of Phing itself.

File Naming Conventions

There are some filenaming conventions used by Phing. Here's a quick rundown on the most basic conventions. A more detailed list can be found in [See Naming And Coding Standards]:

- Filenames consist of no more or less than two elements: *name* and *extension* .
- Choose short descriptive filenames (must be less than 31 chars)
- Names must not contain dots.
- Files containing PHP code must end with the extension *.php* .
- There must be only one class per file (no procedural methods allowed, use a separate file for them), with the exception of "inner"−type / helper classes that can be declared in the same file as the "outer" / main class.
- The name portion of the file must be named exactly like the class it contains.
- Buildfiles and configure rulesets must end with the extension *.xml* .

Coding Standards

We are using PEAR coding standards. We are using a less strict version of these standards, but we do insist that new contributions have phpdoc comments and make explicitly declarations about public/protected/private variables and methods. If you have suggestions about improvements to Phing codebase, don't hesitate to let us know.

System Initialization

PHP installations are typically quite customized −− e.g. different memory_limit, execution timeout values, etc. The first thing that Phing does is modify PHP INI variables to create a standard PHP environment. This is performed by the *init layer* of Phing that uses a three−level initialization procedure. It basically consists of three different files:

- Platform specific wrapper scripts in bin/
- Main application in bin/
- Phing class in classes/phing/

At the first look this may seem to be unnecessary overhead. Why three levels of initialization? The main reason why there are several entry points is that Phing is build so that other frontends (e.g. PHP−GTK) could be used in place of the command line.

Wrapper Scripts

This scripts are technical not required but provided for the ease of use. Imagine you have to type every time you want to build your project:

```
php −qC /path/to/phing/bin/phing.php −verbose all distro snapshot
```

Indeed that is not very elegant. Furthermore if you are lax in setting your environment variables these script can guess the proper variables for you. However you should always set them.

The scripts are platform dependent, so you will find shell scripts for *Unix* like platforms (sh) as well as the batch scripts for *Windows* platforms. If you set−up your path properly you can call Phing everywhere in your system with this command−line (referring to the above example):

```
phing −v2 all distro
```

The Main Application (phing.php)

This is basically a wrapper for the Phing class that actually does all the logic for you. If you look at the sourcecode for phing.php you will see that all real initialization is handled in the Phing class. phing.php is simply the commandline entry point for Phing.

The Phing Class

Given that all the prior initialization steps passed successfully the Phing is included and *Phing::startup()* is invoked by the main application script. It sets−up the system components, system constants ini−settings, PEAR and some other stuff. The detailed start−up process is as follows:

- Start Timer
- Set System Constants
- Set Ini−Settings
- Set Include Paths

After the main application completed all operations (successfully or unsuccessfully) it calls *Phing::shutdown(EXIT_CODE)* that takes care of a proper destruction of all objects and a gracefully termination of the program by returning an *exit code* for shell usage (see [See Program Exit Codes] for a list of exit codes).

System Services

The Exception system

Phing uses the PHP5 try/catch/throw Exception system. Phing defines a number of Exception subclasses for more fine−grained handling of Exceptions. Low level Exceptions that cannot be handled will be wrapped in a BuildException and caught by the outer−most catch() {} block.

Build Lifecycle

This section exists to explain −− or try −− how Phing "works". Particularly, how Phing procedes through a build file and invokes tasks and types based on the tags that it encounters.

How Phing Parses Buildfiles

Phing uses an ExpatParser class and PHP's native expat XML functions to handle the parsing of build files. The handler classes all extend the phing.parser.AbstractHandler class. These handler classes "handle" the tags that are found in the buildfile.

Core tasks and datatypes are mapped to XML tag names in the defaults.properties files −− specifically phing/tasks/defaults.properties and phing/types/defaults.properties.

It works roughly like this:

1. phing.parser.RootHandler is registered to handle the buildfile XML document
2. RootHanlder expects to find exactly one element: <project>. RootHandler invokes the ProjectHandler with the attributes from the <project> tag or throws an exception if no <project> is found, or if something else is found instead.
3. ProjectHandler expects to find <target> tags; for these ProjectHandler invokes the TargetHandler. ProjectHandler also has exceptions for handling certain tasks that can be performed at the top−level: <resolve>, <taskdef>, <typedef>, and <property>; for these ProjectHandler invokes the TaskHandler class. If a tag is presented that doesn't match any expected tags, then ProjectHandler assumes it is a datatype and invokes the DataTypeHandler.
4. TargetHandler expects all tags to be either tasks or datatypes and invokes the appropriate handler (based on the mappings provided in the defaults.properties files).
5. Tasks and datatypes can have nested elements, but only if they correspond to a create*() method in the task or datatype class. E.g. a nested <param> tag must correspond to a createParam() method of the task or datatype.

... More to come ...

Writing Tasks

Creating A Task

We will start creating a rather simple task which basically does nothing more than echo a message to the screen. See [below] for the source code and the following [below] for the XML definition that is used for this task.

```
<?php

require_once "phing/Task.php";

class MyEchoTask extends Task {

    /**
     * The message passed in the buildfile.
     */
    private $message = null;

    /**
     * The setter for the attribute "message"
     */
    public function setMessage($str) {
        $this−>message = $str;
    }

    /**
     * The init method: Do init steps.
     */
    public function init() {
      // nothing to do here
    }

    /**
     * The main entry point method.
     */
    public function main() {
      print($this->message);
    }
}

?>
```

This code contains a rather simple, but complete Phing task. It is assumed that the file is named *MyEchoTask.php* and placed in *classes/phing/tasks/my* directory. We'll explain the source code in detail shortly. But first we'd like to discuss how we should register the task to Phing so that it can be executed during the build process.

Using the Task

The task shown [above] must somehow get called by Phing. Therefore it must be made available to Phing so that the buildfile parser is aware a correlating XML element and it's parameters. Have a look at the minimalistic buildfile example given in [the buildfile below] that does exactly this.

```
<?xml version="1.0" ?>

<project name="test" basedir="." default="myecho">
    <taskdef name="myecho" worker="phing.tasks.my.MyEcho" />

    <target name="test.myecho">
      <myecho message="Hello World" />
    </target>
</project>
```

Besides the XML document prolog and the shell elements that are required to properly execute the task (project, target) you'll find the <*taskdef*> element (line 4) that properly registers your custom task to Phing. For a detailed synopsis of the taskdef element see the [description of this task].

Now, as we have registered the task by assigning a name and the worker class ([see source code above]) it is ready for usage within the <*target*> context (line 8). You see that we pass the message that our task should echo to the screen via an XML attribute called "message".

Source Discussion

No that you've got the knowledge to execute the task in a buildfile it's time to discuss how everything works.

Task Structure

All files containing the definition of a task class follow a common well formed structure:

- Package imports to import all required packages
- The class declaration and definition
- The class's properties
- The class's constructor
- Setter methods for each XML attribute
- The *init()* method
- The *main()* method
- Arbitrary *private* (or *protected*) class methods

Package Imports

Always import all the packages/files needed for this task in full written notation. Furthermore you should always import *phing.Task* at the very top of your *import* block. Then import all other required system or proprietary packages. Import works quite similar to PHP's native *include_once* but with some Java−stylish additions providing a file system independent notation.

For a more in−depth explanation of the used package mechanism and the package support API reference, see [package support] For a list of stock packages provided with Phing, see [package list].

Class Declaration

If you look at line 5 in [the source code of the task] you will find the *class declaration*. This will be familiar to you if you are experienced with OOP in PHP (we assume here that you are). Furthermore there are some fine−grained rules you must obey when creating the classes (see also,[naming and coding standards]):

- Your classname must be exactly like the taskname you are going to implement plus the suffix "Task". In our example case the classname is *MyEchoTask* (constructed by the taskname "myecho" plus the suffix "task"). The upper/lower case casing is currently only for better reading. However, it is encouraged that you use it this way.
- The task class you are creating must at least extend "Task" to inherit all task specific methods.

Class Properties

The next lines you are coding are class properties. Most of them are inherited from the Task superclass, so there's not need to redeclare them. Nevertheless you should *declare* the following ones by your own:

- Taskname. Always hard code the *taskname* property that equals the name of the XML element that your task claims. Currently this information is not used − but it will be in the future.
- Your arbitrary properties that reflect the XML attributes/elements which your task accepts.

In the MyEchoTask example the coded properties can be found in lines 7 to 11. Give you properties meaningful descriptive names that clearly state their function within the context. A couple of properties are inherited from the superclass that must not be declared in the properties part of the code.

For a list of inherited properties (most of them are reserved, so be sure not to overwrite them with your own) can be found in the "Phing API Reference" in the docs/api/ directory.

The Constructor

The next block that follows is the class's constructor. It must be present and call at least the constructor or the parent class. Of course, you can add some initialization data here. It is recommended that you *define* your prior declared properties here.

Setter Methods

As you can see in the XML definition of our task ([see buildfile above] , line 9) there is an attribute defined with the task itself, namely "message" with a value of the the text string that our task should echo. The task must somehow become aware of the attribute name and the value. Therefore the *setter methods* exist.

For each attribute you want to import to the task's namespace you have to define a method named exactly after the very attribute plus the string "Set" prepended. This method accepts exactly one parameter that holds the value of the attribute. No you can set the value an class internal property to the value incoming via the setter method.

In out example the setter is named *SetMessage* , because the XML attribute the echo task accepts is "message". SetMessage now takes the string "Hello World" provided by the parser and sets the value of the internal class property *$strMessage* to "Hello World". It is now available to the task for further disposal.

Creator Methods

Creator methods allow you to manage nested XML tags in your new Phing Task.

init() Method

The init method gets called when the <taskname> xml element closes. It must be implemented even if it does nothing like in the example above. You can do init steps here required to setup your task object properly. After calling the Init−Method the task object remains untouched by the parser. Init should not perform operations related somehow to the action the task performs. An example of using init may be cleaning up the $strMessage variable in our example (i.e. trim($strMessage)) or importing additional workers needed for this task.

The init method should return true or an error object evaluated by the governing logic. If you don't implement init method, phing will shout down with a fatal error.

main() Method

There is exactly one entry entry point to execute the task. It is called after the complete buildfile has been parsed and all targets and tasks have been scheduled for execution. From this point forward the very implementation of the tasks action starts. In case of our example a message (imported by the proper setter method) is Logged to the screen through the system's "Logger" service (the very action this task is written for). The *Log()* method−call in this case accepts two parameters: a event constant and the message to log.

For a in−depth list of system constants see See System Constants. For the detailed reference on the system's logger see [REF] and the Phing API docs located in the docs/ subdirectory.

Arbitrary Methods

For the more or less simple cases (as our example) all the logic of the task is coded in the Main() method. However for more complex tasks common sense dictates that particular action should be swapped to smaller, logically contained units of code. The most common way to do this is separating logic into private class methods − and in even more complex tasks in separate libraries.

```
private function myPrivateMethod() {
    // definition
}
```

More reading on this particular topic can be sound in See Naming And Coding Standards.

Summary

You now have learned how to create and use a task. However we guess there are much questions open concerning task development: "How do I use filesets and mapper" or "How do I implement custom nested tags in my task". Most of these concepts and the proper usage will be clear if you continue reading this doc. Additionally you might check out the appendices for the advanced examples (See Advanced Task Example).

Writing Types

You should only create a standalone Type if the Type needs to be shared by more than one Task. If the Type is only needed for a specific Task −− for example to handle a special parameter or other tag needed for that Task −− then the Type class should just be defined within the same file as the Task. (For example, *phing/filters/XSLTFilter.php* also includes an XSLTParam class that is not used anywhere else.)

For cases where you do need a more generic Type defined, you can create your own Type class −− similar to the way a Task is created [Writing Tasks].

Creating a DataType

Type classes need to extend the abstract DataType class. Besides providing a means of categorizing types, the DataType class provides the methods necessary to support the "refid" attribute. (All types can be given an id, and can be referred to later using that id.)

In this example we are creating a DSN type because we have written a number of DB−related Tasks, each of which need to know how to connect to the database; instead of having database parameters for each task, we've created a DSN type so that we can identify the connection params once and then use it in all our db Tasks.

```
require_once "phing/types/DataType.php";

/**
 * This Type represents a DB Connection.
 */
class DSN extends DataType {

  private $url;
  private $username;
  private $password;
  private $persistent = false;

  /**
   * Sets the URL part: mysql://localhost/mydatabase
   */
  public function setUrl($url) {
    $this->url = $url;
  }

  /**
   * Sets username to use in connection.
   */
  public function setUsername($username) {
    $this->username = $username;
  }

         /**
   * Sets password to use in connection.
   */
  public function setPassword($password) {
    $this->password = $password;
  }

  /**
   * Set whether to use persistent connection.
   * @param boolean $persist
   */
  public function setPersistent($persist) {
    $this->persistent = (boolean) $persist;
  }

  public function getUrl(Project $p) {
    if ($this->isReference()) {
      return $this->getRef($p)->getUrl($p);
    }
    return $this->url;
  }

  public function getUsername(Project $p) {
    if ($this->isReference()) {
      return $this->getRef($p)->getUsername($p);
    }
    return $this->username;
  }
  public function getPassword(Project $p) {
    if ($this->isReference()) {
      return $this->getRef($p)->getPassword($p);
    }
    return $this->password;
  }

  public function getPersistent(Project $p) {
    if ($this->isReference()) {
      return $this->getRef($p)->getPersistent($p);
    }
    return $this->persistent;
  }

  /**
   * Gets a combined hash/array for DSN as used by PEAR.
   * @return array
   */
  public function getPEARDSN(Project $p) {
    if ($this->isReference()) {
      return $this->getRef($p)->getPEARDSN($p);
    }

    include_once 'DB.php';
    $dsninfo = DB::parseDSN($this->url);
    $dsninfo['username'] = $this->username;
    $dsninfo['password'] = $this->password;
    $dsninfo['persistent'] = $this->persistent;

    return $dsninfo;
  }

  /**
   * Your datatype must implement this function, which ensures that there
   * are no circular references and that the reference is of the correct
   * type (DSN in this example).
   *
   * @return DSN
   */
  public function getRef(Project $p) {
    if ( !$this->checked ) {
      $stk = array();
      array_push($stk, $this);
      $this->dieOnCircularReference($stk, $p);
    }
    $o = $this->ref->getReferencedObject($p);
    if ( !($o instanceof DSN) ) {
      throw new BuildException($this->ref->getRefId()." doesn't denote a DSN");
    } else {
      return $o;
    }
  }

}
```

Using the DataType

The *TypedefTask* provides a way to "declare" your type so that you can use it in your build file. Here is how you would use this type in order to define a single DSN and use it for multiple tasks. (Of course you could specify the DSN connection params each time, but the premise behind needing a DSN datatype was to avoid specifying the connection parameters for each task.)

```
<?xml version="1.0" ?>

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

  <typedef name="dsn" worker="myapp.types.DSN" />

  <dsn
      id="maindsn"
      url="mysql://localhost/mydatabase"
      username="root"
      password=""
      persistent="false" />

  <target name="main">

    <my-special-db-task>
             <dsn refid="maindsn"/>
    </my-special-db-task>

    <my-other-db-task>
      <dsn refid="maindsn"/>
    </my-other-db-task>

  </target>

</project>
```

Source Discussion

Getters & Setters

You must provide a setter method for every attribute you want to set from the XML build file. It is good practice to also provide a getter method, but in practice you can decide how your tasks will use your task. In the example above, we've provided a getter method for each attribute and we've also provided an additional method: DSN::getPEARDSN() which returns the DSN hash array used by PEAR::DB, PEAR::MDB, and Creole. Depending on the needs of the Tasks using this DataType, we may only wish to provide the getPEARDSN() method rather than a getter for each attribute.

Also important to note is that the getter method needs to check to see whether the current DataType is a reference to a previously defined DataType –– the DataType::isReference() exists for this purpose. For this reason, the getter methods need to be called with the current project, because References are stored relative to a project.

The getRef() Method

The getRef() task needs to be implemented in your Type. This method is responsible for returning a referenced object; it needs to check to make sure the referenced object is of the correct type (i.e. you can't try to refer to a RegularExpresson from a DSN DataType) and that the reference is not circular.

You can probably just copy this method from an existing Type and make the few changes that customize it to your Type.

Writing Mappers

Writing your own filename mapper classes will allow you to control how names are transformed in tasks like CopyTask, MoveTask, XSLTTask, etc. In some cases you may want to extend existing mappers (e.g. creating a GlobMapper that also transforms to uppercase); in other cases, you may simply want to create a very specific name transformation that isn't easily accomplished with other mappers like GlobMapper or RegexpMapper.

Creating a Mapper

Writing filename mappers is simplified by interface support in PHP5. Essentially, your custom filename mapper must implement *phing.mappers.FileNameMapper*. Here's an example of a filename mapper that creates DOS−style file names. For this example, the "to" and "from" attributes are not needed because all files will be transformed. To see the "to" and "from" attributes in action, look at *phing.mappers.GlobMapper* or *phing.mappers.RegexpMapper*.

```
require_once "phing/mappers/FileNameMapper.php";

/**
 * A mapper that makes those ugly DOS filenames.
 */
class DOSMapper implements FileNameMapper {

  /**
   * The main() method actually performs the mapping.
   *
   * In this case we transform the $sourceFilename into
   * a DOS−compatible name.  E.g.
   * ExtendingPhing.html −> EXTENDI~.DOC
   *
   * @param string $sourceFilename The name to be coverted.
   * @return array The matched filenames.
   */
  public function main($sourceFilename) {

    $info = pathinfo($sourceFilename);
    $ext = $info['extension'];
    // get basename w/o extension
    $bname = preg_replace('/\.\w+\$/', '', $info['basename']);

    if (strlen($bname) > 8) {
      $bname = substr($bname,0,7) . '~';
    }

    if (strlen($ext) > 3) {
      $ext = substr($bname,0,3);
    }

    if (!empty($ext)) {
      $res = $bname . '.' . $ext;
    } else {
      $res = $bname;
    }

    return (array) strtoupper($res);
  }

  /**
   * The "from" attribute is not needed here, but method must exist.
   */
  public function setFrom($from) {}

         /**
   * The "from" attribute is not needed here, but method must exist.
   */
  public function setTo($to) {}

}
```

Using the Mapper

Assuming that this mapper is saved to *myapp/mappers/DOSMapper.php* (relative to a path on PHP's *include_path* or in *PHP_CLASSPATH* env variable), then you would refer to it like this in your build file:

```
<mapper classname="myapp.mappers.DOSMapper"/>
```

Appendix A: Fact Sheet

Built−In Properties

Phing Built−In Properties

Property	Contents
application.startdir	Current work directory
host.arch	Name of the host machine. *Not available on Windows machines.*
host.domain	DNS domain name, i.e. php.net. *Not available on Windows machines.*
host.fstype	The type of the filesystem. Possible values are *UNIX*, *WINNT* and *WIN32*
host.machine	System architecture, i.e. *i586*. *Not available on Windows machines.*
host.name	Operating System name as returned by *posix_uname()*. *Not available on Windows machines.*
host.os.release	Operating version release, i.e. *2.2.10*. *Not available on Windows machines.*
host.os.version	Operating system version, i.e. *#4 Tue Jul 20 17:01:36 MEST 1999*. *Not available on Windows machines.*
line.separator	Character(s) that signal the end of a line, "\n" for Linux, "\r\n" for Windows system, "\r" for Macintosh.
php.classpath	The value of the environment variable *PHP_CLASSPATH*
php.version	Version of the PHP interpreter. Same as PHP constant *PHP_VERSION* (see PHP Manual).
phing.buildfile	Full path to current buildfile
phing.id	ID of hte current phing instance
phing.version	Current Phing version. This property equals the value of the PHP constant *PHP_OS* (see PHP Manual. Possible values are *Linux*, *Win32* and *WINNT*, for example.
project.name	Name of the currently processed project.
project.basedir	The current project basedir
project.description	The description of the currently processed project.
user.home	Value of the environment variable *HOME*.

Command Line Arguments

Currently, the following command line arguments listed in table below are currently available.

Phing Command Line Arguments

Parameter	Meaning
−quiet	Quiet operation, no output at all
−verbose	Verbose, give some output
−debug	Output debug information

−buildfile [builfile]	Specify an alternate builfile name. Default is *build.xml*
−logger path.to.Logger	Specify an alternate logger. Default is *phing.listener.DefaultLogger*. Other options include *phing.listener.NoBannerLogger* and *phing.listener.AnsiColorLogger*
−help	Display the help screen
−projecthelp	List the available targets for this project

TODO Complete this

Distribution File Layout

```
$PHING_HOME
  |-- bin
  |-- classes
  |    `-- phing
  |         |-- filters
  |         |    `-- util
  |         |-- mappers
  |         |-- parser
  |         |-- tasks
  |         |    |-- ext
  |         |    |-- system
  |         |    |    `-- condition
  |         |    `-- user
  |         `-- types
  |-- docs
  |    `-- phing_guide
  `-- test
       |-- classes
           `-- etc
```

Program Exit Codes

Phing is script−safe − means that you can execute Phing and Configure within a automated script context. To check back the success of a Phing call it returns an exit code that can be captured by your calling script. The following list gives you details on the used exit codes and their meaning.

Program Exit Codes

Exitcode	**Description**
−2	Environment not properly defined
−1	Parameter error occured, printed help screen
0	Successful execution, no warnings, no errors
1	Successful execution, but warnings occured

The LGPL License

Source http://www.gnu.org/licenses/lgpl.txt

```
                  GNU LESSER GENERAL PUBLIC LICENSE
                       Version 2.1, February 1999

 Copyright (C) 1991, 1999 Free Software Foundation, Inc.
     59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL.  It also counts
 as the successor of the GNU Library Public License, version 2, hence
 the version number 2.1.]

                            Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
Licenses are intended to guarantee your freedom to share and change
free software--to make sure the software is free for all its users.

  This license, the Lesser General Public License, applies to some
specially designated software packages--typically libraries--of the
Free Software Foundation and other authors who decide to use it.  You
can use it too, but we suggest you first think carefully about whether
this license or the ordinary General Public License is the better
strategy to use in any particular case, based on the explanations below.

  When we speak of free software, we are referring to freedom of use,
not price.  Our General Public Licenses are designed to make sure that
you have the freedom to distribute copies of free software (and charge
for this service if you wish); that you receive source code or can get
it if you want it; that you can change the software and use pieces of
it in new free programs; and that you are informed that you can do
these things.

  To protect your rights, we need to make restrictions that forbid
distributors to deny you these rights or to ask you to surrender these
rights.  These restrictions translate to certain responsibilities for
you if you distribute copies of the library or if you modify it.

  For example, if you distribute copies of the library, whether gratis
or for a fee, you must give the recipients all the rights that we gave
you.  You must make sure that they, too, receive or can get the source
code.  If you link other code with the library, you must provide
complete object files to the recipients, so that they can relink them
with the library after making changes to the library and recompiling
it.  And you must show them these terms so they know their rights.

  We protect your rights with a two-step method: (1) we copyright the
library, and (2) we offer you this license, which gives you legal
permission to copy, distribute and/or modify the library.

  To protect each distributor, we want to make it very clear that
```

```
there is no warranty for the free library.  Also, if the library is
modified by someone else and passed on, the recipients should know
that what they have is not the original version, so that the original
author's reputation will not be affected by problems that might be
introduced by others.

  Finally, software patents pose a constant threat to the existence of
any free program.  We wish to make sure that a company cannot
effectively restrict the users of a free program by obtaining a
restrictive license from a patent holder.  Therefore, we insist that
any patent license obtained for a version of the library must be
consistent with the full freedom of use specified in this license.

  Most GNU software, including some libraries, is covered by the
ordinary GNU General Public License.  This license, the GNU Lesser
General Public License, applies to certain designated libraries, and
is quite different from the ordinary General Public License.  We use
this license for certain libraries in order to permit linking those
libraries into non-free programs.

  When a program is linked with a library, whether statically or using
a shared library, the combination of the two is legally speaking a
combined work, a derivative of the original library.  The ordinary
General Public License therefore permits such linking only if the
entire combination fits its criteria of freedom.  The Lesser General
Public License permits more lax criteria for linking other code with
the library.

  We call this license the "Lesser" General Public License because it
does Less to protect the user's freedom than the ordinary General
Public License.  It also provides other free software developers Less
of an advantage over competing non-free programs.  These disadvantages
are the reason we use the ordinary General Public License for many
libraries.  However, the Lesser license provides advantages in certain
special circumstances.

  For example, on rare occasions, there may be a special need to
encourage the widest possible use of a certain library, so that it becomes
a de-facto standard.  To achieve this, non-free programs must be
allowed to use the library.  A more frequent case is that a free
library does the same job as widely used non-free libraries.  In this
case, there is little to gain by limiting the free library to free
software only, so we use the Lesser General Public License.

  In other cases, permission to use a particular library in non-free
programs enables a greater number of people to use a large body of
free software.  For example, permission to use the GNU C Library in
non-free programs enables many more people to use the whole GNU
operating system, as well as its variant, the GNU/Linux operating
system.

  Although the Lesser General Public License is Less protective of the
users' freedom, it does ensure that the user of a program that is
linked with the Library has the freedom and the wherewithal to run
that program using a modified version of the Library.

  The precise terms and conditions for copying, distribution and
```

```
    a) The modified work must itself be a software library.

    b) You must cause the files modified to carry prominent notices
    stating that you changed the files and the date of any change.

    c) You must cause the whole of the work to be licensed at no
    charge to all third parties under the terms of this License.

    d) If a facility in the modified Library refers to a function or a
    table of data to be supplied by an application program that uses
    the facility, other than as an argument passed when the facility
    is invoked, then you must make a good faith effort to ensure that,
    in the event an application does not supply such function or
    table, the facility still operates, and performs whatever part of
    its purpose remains meaningful.

    (For example, a function in a library to compute square roots has
    a purpose that is entirely well-defined independent of the
    application.  Therefore, Subsection 2d requires that any
    application-supplied function or table used by this function must
    be optional: if the application does not supply it, the square
    root function must still compute square roots.)

These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Library,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
distribute the same sections as part of a whole which is a work based
on the Library, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote
it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Library.

In addition, mere aggregation of another work not based on the Library
with the Library (or with a work based on the Library) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

  3. You may opt to apply the terms of the ordinary GNU General Public
License instead of this License to a given copy of the Library.  To do
this, you must alter all the notices that refer to this License, so
that they refer to the ordinary GNU General Public License, version 2,
instead of to this License.  (If a newer version than version 2 of the
ordinary GNU General Public License has appeared, then you can specify
that version instead if you wish.)  Do not make any other change in
these notices.

  Once this change is made in a given copy, it is irreversible for
that copy, so the ordinary GNU General Public License applies to all
subsequent copies and derivative works made from that copy.
```

```
LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING
RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A
FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF
SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
DAMAGES.

                     END OF TERMS AND CONDITIONS

           How to Apply These Terms to Your New Libraries

  If you develop a new library, and you want it to be of the greatest
possible use to the public, we recommend making it free software that
everyone can redistribute and change.  You can do so by permitting
redistribution under these terms (or, alternatively, under the terms of the
ordinary General Public License).

  To apply these terms, attach the following notices to the library.  It is
safest to attach them to the start of each source file to most effectively
convey the exclusion of warranty; and each file should have at least the
"copyright" line and a pointer to where the full notice is found.

    <one line to give the library's name and a brief idea of what it does.>
    Copyright (C) <year>  <name of author>

    This library is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    This library is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with this library; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the library, if
necessary.  Here is a sample; alter the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the
  library `Frob' (a library for tweaking knobs) written by James Random Hacker.

  <signature of Ty Coon>, 1 April 1990
  Ty Coon, President of Vice

That's all there is to it!
```

Appendix B: Core Tasks

This appendix contains a reference of all core tasks, i.e. all tasks that are needed to build a basic project. If you are looking for binarycloud related tasks, look in appendix ?.

This reference lists the tasks alphabetically by the name of the classes that implement the tasks. So if you are searching for the reference to the `<copy>` tag, for example, you will want to look at the reference of CopyTask.

AdhocTaskdefTask

The AdhocTaskdefTask allows you to define a task within your build file.

```
<target name="main"
        description="==>test AdhocTask ">

                <adhoc-task name="foo"><![CDATA[
                        class FooTest extends Task {
                                private $bar;

                                function setBar($bar) {
                                        $this->bar = $bar;
                                }

                                function main() {
                                        $this->log("In FooTest: " . $this->bar);
                                }
                        }
                ]]></adhoc-task>

                <foo bar="B.L.I.N.G"/>
</target>
```

Note that you should use <![CDATA[...]]> so that you don't have to quote entities within your *<adhoc-task></adhoc-task>* tags.

Parameters

Name	Type	Description	Default	Required
name	String	Name of XML tag that will represent this task.	n/a	Yes

AdhocTypedefTask

The AdhocTaskdefTask allows you to define a datatype within your build file.

```
<target name="main"
        description="==>test AdhocType">

                <adhoc-type name="dsn"><![CDATA[
                        class CreoleDSN extends DataType {
                                private $url;

                                function setUrl($url) {
                                        $this->url = $url;
                                }

                                function getUrl() {
                                return $this->url;
                                }
                        }
                ]]></adhoc-type>

     <!-- creole-sql task doesn't exist; just an example -->
     <creole-sql file="test.sql">
       <dsn url="mysql://root@localhost/test"/>
                </creole-sql>

</target>
```

Note that you should use <![CDATA[...]]> so that you don't have to quote entities within your *<adhoc−type></adhoc−type>* tags.

Parameters

Name	Type	Description	Default	Required
name	String	Name of XML tag that will represent this datatype..	n/a	Yes

AppendTask

The Append Task appends text or contents of files to a specified file.

```
<append destFile="${process.outputfile}">
  <filterchain>
    <xsltfilter style="${process.stylesheet}">
             <param name="mode" expression="${process.xslt.mode}"/>
    </xsltfilter>
  </filterchain>
  <filelist dir="book/" listfile="book/PhingGuide.book"/>
</append>
```

In the example above, AppendTask is reading a filename from *book/PhingGuide.book*, processing the file contents with XSLT, and then appending the result to the file located at *${process.outputfile}*. This is a real example from the build file used to generate this book!

Parameters

Name	Type	Description	Default	Required
destFile	File	Path of file to which text should be appended.	n/a	Yes
file	File	Path to file that should be appended to destFile.	n/a	Yes
text	String	Some literal text to append to file.	*n/a*	No

Supported Nested Tags

- FileList
- FilterChain

AvailableTask

Available Task tests if a resource/file is set and sets a certain property to a certain value if it exists.

```
<available file="/tmp/test.txt" property="test_txt_exists" value="Yes"/>
```

```
<available file="/home/foo" type="dir" property="properties.yetanother" />
```

```
<available file="/home/foo/bar" property="foo.bar" value="Well, yes" />
```

Here, *AvailableTask* first checks for the existance of either file or directory named *test.txt* in */tmp*. Then, it checks for the directory *foo* in */home* and then for the file or directory *bar* in */home/foo*. If */tmp/test.txt* is found, the property *test_txt_exists* is set to *"Yes"*, if */home/foo* is found and a directory, *properties.yetanother* is set to *"true"* (default). If */home/foo/bar* exists, *AvailableTask* will set *foo.bar* to *"Well, yes"*.

Parameters

Name	Type	Description	Default	Required
property	string	Name of the property that is to be set.	n/a	Yes
value	String	The value the propert is to be set to.	*"true"*	No
file	String	File/directory to check existance.	n/a	Yes (or *resource*)
resource	String	Path of the resource to look for.	n/a	Yes (or *file*)
type	String (file|dir)	Determines if *AvailableTask* should look for a file or a directory at the position set by *file*. If empty, it checks for either file or directory.	n/a	No

CallTargetTask

The CallTargetTask calls a target from the same project. A <*project*> tag may contain <*property*> tags that define new properties. In the following example, the properties *property1* and *foo* are defined and only accessible inside the called target.

However, this will only work if the properties are not yet set outside the `"phingcall"` tag.

Example

```
<target name="foo">
    <phingcall target="bar">
        <property name="property1" value="aaaaa" />
        <property name="foo" value="baz" />
    </phingcall>
</target>

<target name="bar" depends="init">
    <echo message="prop is ${property1} ${foo}" />
</target>
```

Parameters

Name	Type/Values	Description	Default	Required
target	string	The name of the target in the same project that is to be called.	n/a	Yes

CopyTask

The phing Copy Task. Copies a file or directory to a new file or directory. Files are only copied if the source file is newer than the destination file, or when the destination file does not exist. It is possible to explictly overwrite existing files.

Example

On the one hand, CopyTask can be used to copy file by file:

```
<copy file="somefile.txt" tofile="/tmp/anotherfile.bak" overwrite="true"/>
```

Additionally, *CopyTask* supports Filesets, i.e. you can easily include/exclude one or more files. For more information, see Appendix C. Mappers and Filterchains are also supported by *CopyTask*, so you can do almost everything that needs processing the content of the files or the filename.

Notice: CopyTask does not allow self copying, i.e. copying a file to the same name for security reasons.

```
<copy todir="/tmp/backup" >
  <fileset dir=".">
    <include name="**/*.txt">
    <include name="**/*.doc">
    <include name="**/*.swx">
  </fileset>
</copy>
```

Attributes

Name	Type	Description	Default	Required
file	String	The source file.	Yes	
tofile	String	The destination the file is to be written to. *tofile* specifies a *full* filename. If you only want to specify a directory to copy to, use *todir*. Either this or the *todir* attribute is required.	n/a	Yes (or *todir*)

todir	String	The directory the file is to be copied to. The file will have the same name of the source file. If you want to specify a different name, use *tofile*.	n/a	Yes (or *tofile*)
overwrite	Boolean	If set to true, the target file will be overwritten.	false	No
tstamp	Boolean	If set to *true*, the new file will have the same mtime as the old one.	false	No
includeemptydirs	Boolean	If set to *true*, also empty directories are copied.	true	No

Supported Nested Tags

- Fileset
- Filterchain
- Mapper

DeleteTask

Deletes a file or directory, or set of files defined by a fileset. See Appendix C for information on Filesets.

Example

```
<-- Delete a specific file from a directory -->
<delete file="foo.bar" dir="/tmp" />

<-- Delete a directory -->
<delete dir="/tmp/darl" includeemptydirs="true" verbose="true" failonerror="true" />
```

Attributes

Name	Type	Description	Default	Required
file	String	The file that is to be deleted. You either have to specify this attribute, or *dir* or both.	n/a	Yes
dir	String	The directory that is to be deleted. You either have to specify this attribute, or *file* or both.	n/a	Yes (or *file*)
verbose	Boolean	Used to force listing of all names of deleted files.	n/a	Yes
quiet	Boolean	If the file does not exist, do not display a diagnostic message or modify the exit status to reflect an error. This means that if a file or directory cannot be deleted, then no error is reported. This setting emulates the −f option to the Unix *rm* command. Default is false meaning things are verbose	n/a	Yes
failonerror	Boolean	If this attribute is set to *true*, DeleteTask will verbose on errors but the build process will not be stopped.	true	Yes
includeemptydirs	Boolean	Determines if empty directories are also to be deleted.	false	False

Supported Nested Tags

- Fileset

EchoTask

This task simply verboses a string.

Example

```
<echo msg="Phing rocks!" />

<echo message="Binarycloud, too." />

<echo>And don't forget Propel.</echo>
```

Attributes

Name	Type	Description	Default	Required
msg	String	The string that is to be send to the output.	n/a	Yes
message	String	Alias for *msg*.	n/a	Yes

ExecTask

Executes a shell command. You can use this to quickly add a new command to Phing. However, if you want to use this regularly, you should think about writing a Task for it.

Example

```
<-- List the contents of "/home". -->
<exec command="ls -l" dir="/home" />

<-- Start the make process in "/usr/src/php-4.0". -->
<exec command="make" dir="/usr/src/php-4.0" />

<-- List the contents of "/tmp" out to a file. -->
<exec command="ls -l" "/tmp > foo.out" escape="false" />
```

Attributes

Name	Type	Description	Default	Required
command	String	The command that is to be executed.	n/a	Yes
dir	String	The directory the command is to be executed in.	n/a	Yes
os	String	Only execute if os.name contains specified text.	n/a	No

escape	Boolean	By default, we escape shell metacharacters before executing. Setting this to false will disable this precaution.	TRUE	No

ForeachTask

The foreach task iterates over a list, a list of paths, or both. If both, list and paths, are specified, the list will be evaluated first. Nested paths are evaluated in the order they appear in the task.

Example

```
<!-- loop through languages, and call buildlang task with setted param -->
<foreach list="languages" param="lang" target="buildlang" />
```

Attributes

Name	Type	Description	Default	Required
list	string	The list of values to process, with the delimiter character, indicated by the "delimiter" attribute, separating each value.	n/a	Yes
target	string	The target to call for each token, passing the token as the parameter with the name indicated by the "param" attribute.	n/a	Yes
param	string	The name of the parameter to pass the tokens in as to the target.	n/a	Yes
delimiter	string	The delimiter string that separates the values in the "list" parameter. The default is ",".	,	No

InputTask

The *InputTask* can be used to interactively set property values based on input from the console (or other Reader).

Example

```
<echo>HTML pages installing to: ${documentRoot}</echo>
<echo>PHP classes installing to: ${servletDirectory}</echo>

<input propertyname="documentRoot">Web application document root</input>
<input propertyname="servletDirectory"
  defaultValue="/usr/servlets" promptChar="?">PHP classes install dir</input>

<echo>HTML pages installed to ${documentRoot}</echo>
<echo>PHP classes installed to ${servletDirectory}</echo>
```

Attributes

Name	Type	Description	Default	Required
propertyName	String	The name of the property to set.	n/a	Yes
defaultValue	String	The default value to be set if no new value is provided.	n/a	Yes

message	String	Prompt text (same as CDATA).	n/a	No
promptChar	String	The prompt character to follow prompt text.	n/a	No

MkdirTask

Create a directory.

Example

```
<-- Create a temp directory -->
<mkdir dir="/tmp/foo" />

<-- Using mkdir with a property -->
<mkdir dir="{$dirs.install}/tmp" />
```

Attributes

Name	Type	Description	Default	Required
dir	String	The directory that is to be created.	n/a	Yes

MoveTask

Moves a file or directory to a new file or directory. By default, the destination file is overwritten if it already exists. When overwrite is turned off, then files are only moved if the source file is newer than the destination file, or when the destination file does not exist.

Source files and directories are only deleted if the file or directory has been copied to the destination successfully.

Example

```
<-- The following will move the file "somefile.txt" to "/tmp" and
    change its filename to "anotherfile.bak". It will overwrite
    an existing file. -->
<move file="somefile.txt" tofile="/tmp/anotherfile.bak" overwrite="true"/>

<-- This will move the "/tmp" directory to "/home/default/tmp",
    preserving the directory name. So the final name is
    "/home/default/tmp/tmp". Empty directories are also copied -->
<move file="/tmp" todir="/home/default/tmp" includeemptydirs="true" />
```

Attributes and Nested Elements

For further documentation, see CopyTask, since MoveTask only is a child of CopyTask and inherits all attributes.

PhingTask

This task calls another build file. You may specify the target that is to be called within the build file. Additionally, the `<phing>` Tag may contain `<property>` Tags (see PropertyTask).

Example

```
<-- Call target "xslttest" from buildfile "alternativebuildfile.xml" -->
 <phing phingfile="alternativebuild.xml" inheritRefs="true" target="xslttest" />

<-- Do a more complex call -->
<phing phingfile="somebuild.xml" target="sometarget">
  <property name="foo" value="bar">
  <property name="anotherone" value="32">
</phing>
```

Attributes

Name	Type	Description	Default	Required
inheritAll	Boolean	If true, pass all properties to the new phing project.	true	No
inheritRefs	Boolean	If true, pass all references to the new phing project.	false	No
dir	String	The directory to use as a base directory for the new phing project. Default is the current project's basedir, unless inheritall has been set to *false*, in which case it doesn't have a default value. This will override the basedir setting of the called project.	n/a	No
phingFile	String	The build file to use. Defaults to "build.xml". This file is expected to be a filename relative to the dir attribute given.	n/a	Yes
target	String	The target of the new Phing project to execute. Default is the new project's default target.	n/a	No

Supported Nested Tags

- Fileset

Base directory of the new project

The base directory of the new project is set dependant on the *dir* and the *inheritAll* attribute. This is important to keep in mind or else you might run into bugs in your build.xml's. The following table shows when which value is used:

***dir* attribute**	***inheritAll* attribute**	**new project's basedir**
value provided	true	value of *dir* attribute
value provided	false	value of *dir* attribute
omitted	true	basedir of calling task (the build file containing the *<phing>* call.

omitted	false	basedir attribute of the <*project*> element of the new project

PearPackageTask

With the PearPackageTask, you can create a package.xml which can be installed using the PEAR installer. Use this in conjunction with the TarTask to completely script the building of a PEAR pacakge.

This task uses the PEAR_PackageFileManager class. In order to be maximally flexible, the majority of options are set generically (using <*option*> tag) and are set using PEAR_PackageFileManager::setOptions(). Use the <*mapping*> tag to represent complex values (which are turned into associative arrays and also set using setOptions() method).

Example

```
<pearpkg name="phing" dir="${build.src.dir}" destFile="${build.base.dir}/package.xml">
<fileset>
  <include name="**"/>
</fileset>
<option name="notes">Sample release notes here.</option>
<option name="description">Package description</option>
<option name="summary">Short description</option>
<option name="version" value="2.0.0b1"/>
<option name="state" value="beta"/>
 <mapping name="maintainers">
  <element>
   <element key="handle" value="hlellelid"/>
   <element key="name" value="Hans"/>
   <element key="email" value="hans@xmpl.org"/>
   <element key="role" value="lead"/>
  </element>
</mapping>
</pearpkg>
```

Attributes

Name	Type	Description	Default	Required
name	String	The name of the PEAR package.	n/a	Yes
dir	String	The base directory of files to add to package.	n/a	Yes
destFile	String	The file to create.	package.xml in base directory	No

Supported Nested Tags

- fileset
- option
- mapping

PhpEvalTask

With the PhpEvalTask, you can set a property to the results of evaluating a PHP expression or the result returned by a function/method call.

Examples

```
<php function="crypt" returnProperty="enc_passwd">
  <param value="${auth.root_passwd}"/>
</php>

<php expression="3 + 4" returnProperty="sum"/>
```

Attributes

Name	Type	Description	Default	Required
function	String	The name of the Property.	n/a	One of these is required.
expression	String	The expression to evaluate.	n/a	
class	String	The static class which contains function.	n/a	No
returnProperty	String	The name of the property to set with result of expression or function call.	n/a	No

Supported Nested Tags

- param

PropertyTask

With PropertyTask, you can define *user* properties in your build file.

Example

```
<property name="strings.test" value="Harr harr, more power!" />
<echo message="{$strings.text}" />

<property name="foo.bar" value="Yet another property..." />
<echo message="{$foo.bar}" />
```

Attributes

Name	Type	Description	Default	Required
name	String	The name of the Property.	n/a	Yes
value	String	The value of the Property.	n/a	Yes
override	Boolean	Whether to force override of existing value.	false	No

ReflexiveTask

The ReflexiveTask performs operations on files. It is essentially a convenient way to transform (using filter chains) files without copying them.

Example

```
<reflexive>
  <fileset dir=".">
    <include pattern="*.html">
  </fileset>
  <filterchain>
    <replaceregexp>
      <regexp pattern="\n\r" replace="\n"/>
    </replaceregexp>
  </filterchain>
</reflexive>
```

Attributes

Name	Type	Description	Default	Required
file	String	A single file to be processed.	n/a	Yes (unless <fileset> provided)

Supported Nested Tags:

- fileset
- filterchain

ResolvePathTask

The ResolvePathTask turns a relative path into an absolute path, with respect to specified directory or the project basedir (if no dir attribute specified).

This task is useful for turning a user−defined relative path into an absolute path in cases where buildfiles will be called in different directories. Without this task, buildfiles lower in the directory tree would mis−interpret the user−defined relative paths.

Example

```
<property name="relative_path" value="./dirname"/>

<resolve propertyName="absolute_path" file="${relative_path}"/>

<echo>Resolved [absolute] path: ${absolute_path}</echo>
```

Attributes

Name	Type	Description	Default	Required
file	String	The file or directory path to resolve.	n/a	Yes
dir	File	The base directory to use when resolving "file".	project.basedir	No
propertyName	String	The name of the property to set with resolved (absolute) path.	n/a	Yes

TarTask

The *TarTask* creates a tarball from a fileset or directory.

Example

```
<tar destfile="phing.tar" basedir="." compression="gzip">
 <fileset dir=".">
         <include name="**/**" />
 </fileset>
</tar>
```

The above example uses a fileset to determine which files to include in the archive.

Attributes

Name	Type	Description	Default	Required
destfile	String	Tarball filename	n/a	Yes
basedir	String	Base directory to tar (if no fileset specified, entire directory contents will be included in tar)	none	No
compression	String	Type of compression to use (gzip, bzip2, none)	none	No

Supported Nested Tags

- fileset

TaskdefTask

With the TaskdefTask you can import a user task into your buildfile.

Example

```
<!-- Includes the Task named "ValidateHTMLTask" and makes it available by
     <validatehtml> -->
<taskdef classname="user.tasks.ValidateHTMLTask" name="validatehtml" />

<!-- Includes the Task "RebootTask" from "user/sometasks" somewhere inside
     the $PHP_CLASSPATH -->
<taskdef classname="user.sometasks.RebootTask" name="reboot" />
```

Attributes

Name	Type	Description	Default	Required
classname	String	The path to the class that defines the TaskClass.	n/a	Yes
name	String	The name the task is available as after importing. If you specify "validate", for example, you can access the task imported here with `<validate>`.	n/a	Yes
classpath	String	The classpath to use when including classes. This is added to PHP's include_path.	n/a	No
classpathref	String	Reference to classpath to use when including classes. This is added to PHP's include_path.	n/a	No

Supported Nested Tags

- classpath

TouchTask

The TouchTask works like the Unix *touch* command: It sets the modtime of a file to a specific time. Default is the current time.

Example

```
<touch file="README.txt" millis="102134111" />
```

```
<touch file="COPYING.lib" datetime="10/10/1999 09:31 AM" />
```

Attributes

Name	Type	Description	Default	Required
file	String	The file which time is to be changed.	n/a	No
datetime	DateTime	The date and time the mtime of the file is to be set to. The format is "MM/DD/YYYY HH:MM AM or PM"	*now*	No
millis	Integer	The millisecons since midnight Jan 1 1970 (Unix epoche).	*now*	No

TypedefTask

With the TypedefTask you can import a user task into your buildfile.

Example

```
<!-- Includes the Type named "CustomProject" and makes it available by
     <cproject> -->
<taskdef classname="user.types.CustomProject" name="cproject" />
```

Attributes

Name	Type	Description	Default	Required
classname	String	The path to the class that defines the type class.	n/a	Yes
name	String	The name the type is available as after importing. If you specify "cproject", for example, you can access the type imported here with `<cproject>`.	n/a	Yes
classpath	String	The classpath to use when including classes. This is added to PHP's include_path.	n/a	No
classpathref	String	Reference to classpath to use when including classes. This is added to PHP's include_path.	n/a	No

Supported Nested Tags

- classpath

UpToDateTask

Available Task tests if a resource/file is set and sets a certain property to a certain value if it exists.

```
<uptodate property="propelBuild.notRequired" targetfile="${deploy}\propelClasses.tgz" >
  <srcfiles dir= "${src}/propel" includes="**/*.php"/>
</uptodate>
```

sets the property *propelBuild.notRequired* to true if the *${deploy}/propelClasses.tgz* file is more up−to−date than any of the PHP class files in the *${src}/propel* directory.

Parameters

Name	Type	Description	Default	Required
property	string	Name of the property that is to be set.	n/a	Yes
value	String	The value the propert is to be set to.	*"true"*	No
srcfile	String	The file to check against target file(s).	n/a	Yes (or nested *srcfiles*)
targetfile	String	The file for which we want to determine the status.	n/a	Yes (or nested *mapper*)

Supported Nested Tags

- FileSet
- Mapper

XsltTask

With *XsltTask*, you can run a XSLT tranformation on an XML file. Actually, *XsltTask* extends *CopyTask*, so you can use all the elements allowed there.

Example

```
<!-- Transform docbook with an imaginary XSLT file -->
<xslt todir="/srv/docs/phing"tyle="dbk2html.xslt" >
  <fileset dir=".">
    <include name="**/*.xml" />
  </fileset>
</xslt>
```

Attributes

Name	Type	Description	Default	Required
style	String	The path where the Xslt file is located	n/a	Yes

Note: You can also use all the attributes available for CopyTask.

Suppported Nested Elements

Note: You can use all the elements also available for CopyTask.

Additionally, you can use *<param>* tags with a *name* and a *value* attribute. These parameters are then available from within the xsl style sheet.

Appendix C: Core Types

This appendix contains a reference of the system *data types* contained in Phing.

FileList

FileLists offer a way to represent a specific list of files. Unlike FileSets, FileLists may contain files that do not exist on the filesystem. Also, FileLists can represent files in a specific order −− whereas FileSets represent files in whichever order they are returned by the filesystem.

Usage Examples

```
<filelist dir="/etc" files="httpd/conf/httpd.conf,php.ini"/>
```

Or you can use a *listfile*, which is expected to contain one filename per line:

```
<filelist dir="conf/" listfile="ini_files.txt"/>
```

This will grab each file as listed in *ini_files.txt*. This can be useful if one task compiles a list of files to process and another task needs to read in that list and perform some action to those files.

Attributes

Attributes for the *<fileset>* tag

Name	Type	Description	Default	Required
dir	String	The directory, to which the paths given in *files* or *listfile* are relative.	n/a	Yes
files	String	Comma or space−separated list of files.	n/a	Yes (or *listfile)*
listfile	String	A text file with one filename per line.	n/a	Yes (or *files)*

FileSet

Filesets offer a easy and straigtforward way to include files. You can include/exclude files in/from a fileset using the *<include>*/*<exclude>* tags. In patterns, one asterisk (*****) maps to a part of a file/directory name within a directory level. Two asterisks (******) may include above the "border" of the directory separator.

Examples

- *test*.xml* will include *test_42.xml*, but it will not include *test/some.xml*.
- *test**.xml* fits to *test_42.xml* as well as to *test/bla.xml*, for example.
- ***/*.ent.xml* fits to all files that end with *ent.xml* in all subdirectories of the directory specified with the *dir* attribute of the *<fileset>* tag. However, it will not include any files that are directly in the base

directory of the file set.

Usage Example

```
<fileset dir="/etc" >
  <include name="httpd/**" />
  <include name="php.ini" />
</fileset>
```

This will include the apache configuration and PHP configuration file from */etc*.

Attributes

Attributes for the <*fileset*> tag

Name	Type	Description	Default	Required
dir	String	The directory, the paths given in *include*/*exclude* are relative to.	n/a	Yes

The only tags that are supported by *Fileset* are the <*include*> and the <*exclude*> tags. These tags must have a *name* attribute that contains the pattern to include/exclude.

Path / Classpath

The Path data type can be used to respresent path structures. In many cases the path type will be used for nested <classpath> tags. E.g.

```
<path id="project.class.path">
  <pathelement dir="lib/"/>
  <pathelement dir="ext/"/>
</path>

<target name="blah">
  <taskdef name="mytask" path="myapp.phing.tasks.MyTask">
    <classpath refid="project.class.path"/>
  </taskdef>
</target>
```

Attributes for <*path*> tag

Name	Type	Description	Default	Required
dir	String	Specific path to directory	n/a	No
path	String	A path (which contains multiple locations separated by path.separator) to add.	n/a	No

Nested Tags

The <*path*> tag supports nested <*fileset*> and <*dirset*> tags.

Core Filters

Filters have to be defined within a *<filterchain>* context to work. Example:

```
<filterchain>
  <expandproperties />
</filterchain>
```

There are two ways to use a filter: System filters (the ones shipped with Phing) can be used with their own tag name, such as *<xsltfilter>*, *<expandpropertyfilter>* or *<tabtospaces>*. Another way is to use the *<filterreader>* tag.

PhingFilterReader

The PhingFilterReader is used when you want to use filters that are not directly available through their own tag. Example:

```
<filterchain>
  <filterreader classname="phing.filter.ReplaceTokens">
    <-- other way to set attributes -->
    <param name="begintoken" value="@@" />
    <param name="endtoken" value="@@" />

    <-- other way to set nested tags -->
    <param type="token" key="bar" value="foo" />
  </filterreader>
</filterchain>
```

In the *filterreader* tag you have to specify the path the class is in. The *FilterReader* will then load this class and pass the parameters to the loaded filter. There are two types of parameters: First, you can pass "normal" parameters to the loaded filter. That means, you can pass parameters as if they were attributes. If you want to do this, you only specify the *name* and *value* attributes in the *param* tag. You can also pass nested elements to the filter. Then, you have to specify the *type* attribute. This attribute specifies the name of the nested tag.

The result of the example above is identical with the following code:

```
<filterchain>
  <replacetokens begintoken="@@" endtoken="@@">
    <token key="bar" value="foo" />
  </replacetokens>
</filterchain>
```

Attributes

Attributes for *<filterreader>*

Name	Type	Description	Default	Required
classname	String	Name of class to use (in dot−path notation).	n/a	Yes

classpath	String	The classpath to use when including classes. This is added to PHP's include_path.	n/a	No
classpathref	String	Reference to classpath to use when including classes. This is added to PHP's include_path.	n/a	No

Nested Tags

The *PhingFilterReader* supports nested <classpath>.

Advanced

In order to support the <filterreader ... /> sytax, your class must extend the BaseParamFilterReader class. Most of the filters that are bundled with Phing can be invoked using this syntax. The noteable exception (at time of writing) is the ReplaceRegexp filter, which expects find/replace parameters that do not fit the name/value mold. For this reason, you must always use the shorthand <replaceregexp .../> to invoke this filter.

ExpandProperties

The ExpandProperties simply replaces property names with their property values. For example, if you have the following in your build file:

```
<property name="description.txt" value="This is a text file" />

<copy todir="/tmp">
  <filterchain>
    <expandproperties />
  </filterchain>

  <fileset dir=".">
    <include name="**" />
  </fileset>
</copy>
```

And the string *${description.txt}* it will be replaced by *This is a text file*.

HeadFilter

This filter reads the first *n* lines of a file; the others are not further passed through the filter chain. Usage example:

```
<filterchain>
  <headfilter lines="20" />
</filterchain>
```

Attributes

Attributes for the *<headfilter>* tag

Name	Type	Description	Default	Required
lines	Integer	Number of lines to read.	10	No

Line Contains

This filter is only "permeable" for lines that contain the expression given as parameter. For example, the following filterchain would only let all the lines pass that contain *class*:

```
<filterchain>
  <linecontains>
    <contains value="class" />
  </linecontains>
</filterchain>
```

Nested Tags

The *linecontains* tag must contain one or more *contains* tags. Latter must have a *value* attribute that has to be set to the string the line has to contain to be let through.

LineContainsRegexp

This filter is similar to LineContains but you can specify regular expressions instead of simple strings.

```
<filterchain>
  <linecontainsregexp>
    <regexp pattern="foo(.*)bar" />
  </linecontainsregexp>
</filterchain>
```

Nested Tags

The *LineContains* filter has to contain at least one *regexp* tag. This must have a *pattern* attribute that is set to a regular expression.

PrefixLines

This filter adds a prefix to every line. The following example will add the string *foo:* in front of every line.

```
<filterchain>
  <prefixlines prefix="foo: " />
</filterchain>
```

Attributes

Attributes for the <*prefixlines*> tag

Name	Type	Description	Default	Required

prefix	string	Strint to prepend to every line.	n/a	Yes

ReplaceTokens

The *ReplaceTokens* filter will replace certain tokens. Tokens are strings enclosed in special characters. If you want to replace *##BCHOME##* by the path to the directory set in the environment variable *BCHOME*, you could do the following:

```
<property environment="env" />

<filterchain>
  <replacetokens begintoken="##" endtoken="##">
    <token key="BCHOME" value="${env.BCHOME}" />
  </replacetokens>
</filterchain>
```

Attributes

Attributes for the *<replacetokens>* tag

Name	Type	Description	Default	Required
begintoken	string	The string that marks the beginning of a token.	@	No
endtoken	string	The string that marks the end of a token.	@	No

Nested Tags

The *ReplaceTokens* filter must contain one or more *token* tags. These must have a *key* and a *value* attribute.

ReplaceRegexp

The *ReplaceRegexp* filter will perform a regexp find/replace on the input stream. For example, if you want to replace ANT with Phing (ignoring case) and you want to replace references to *.java with *.php:

```
<filterchain>
  <replaceregexp>
        <regexp pattern="ANT" replace="Phing" ignoreCase="true"/>
        <regexp pattern="(\w+)\.java" replace="\1.php"/>
  </replaceregexp>
</filterchain>
```

Nested Tags

The *ReplaceTokens* filter must contain one or more *regexp* tags. These must have *pattern* and *replace* attributes –– and optionally the *ignoreCase* attribute.

StripLineBreaks

The *StripLineBreaks* filter removes all linebreaks from the stream passed through the filter chain.

```
<filterchain>
  <striplinebreaks />
</filterchain>
```

StripLineComments

The *StripLineComments* filter removes all line comments from the stream passed through the filter chain:

```
<filterchain>
  <striplinecomments>
    <comment value="#" />
    <comment value="--" />
    <comment value="//" />
  </striplinecomments>
</filterchain>
```

Nested Tags

The *striplinecomments* tag must contain one or more *comment* tags. These must have a *value* attribute that specifies the character(s) that start a line comment.

StripPhpComments

The *StripPhpComment* filter removes all PHP comments from the stream passed through the filter.

```
<filterchain>
  <stripphpcomments />
</filterchain>
```

TabToSpaces

The TabToSpaces filter replaces all tab characters with a given count of space characters.

```
<filterchain>
  <tabtospaces tablength="8" />
</filterchain>
```

Attributes

Attributes for the <*tabtospaces*> filter

Name	Type	Description	Default	Required

tablength	Integer	The number of space characters that a tab is to represent.	8	No

TailFilter

Similar to HeadFilter, this filter reads the last *n* lines of a file; the others are not further passed through the filter chain. Usage example:

```
<filterchain>
  <tailfilter lines="20" />
</filterchain>
```

Attributes

Attributes for the <*tailfilter*> tag

Name	Type	Description	Default	Required
lines	Integer	Number of lines from the back to read.	10	No

XsltFilter

The *XsltFilter* applies a XSL template to the stream. Though you can use this filter directly, you should use XsltTask which is shortcut to the following lines:

```
<filterchain>
  <xsltfilter style="somexslt.xsl" />
</filterchain>
```

Attributes

Attributes for the <*xsltfilter*> tag

Name	Type	Description	Default	Required
style	String	The XSLT stylesheet to use for transformation.	n/a	Yes
html	Boolean	Whether to parse the input as HTML (using libxml2 DOMDocument::loadHTML()).	false	No

Nested Tags

The *XsltFilter* filter may contain one or more *param* tags to pass any XSLT parameters to the stylesheet. These *param* tags must have *name* and *expression* attributes.

Core Mappers

While filters are applied to the *content* of files, Mappers are applied to the *filenames*. All mappers have the same API, i.e. the way you use them is the same:

```
<mapper type="mappername" from="frompattern" to="topattern" />
```

Attributes

Attributes for the *<mapper>* tag

Name	Type	Description	Default	Required
type	String	Type of the mapper.	n/a	Yes
from	String	The pattern the filename is to be matched to. The exact meaning is dependent on the implementation of the mapper.	n/a	depends on the implementation of the mapper
to	String	The pattern according to which the filename is to be changed to. Here, the usage is dependent on the implementation of the mapper, too.	n/a	depends on the implementation of the mapper

FlattenMapper

The *FlattenMapper* removes the directories from a filename and solely returns the filename.

```
<copy todir="/tmp">
  <mapper type="flatten" />

  <fileset refid="someid" />
</copy>
```

This code will copy all files in the fileset to /tmp. All files will be in the target directory.

Examples

```
<mapper type="flatten" />
```

Applying the mapper, you will get the following results from the following filenames:

From	To
test.txt	test.txt
./foo/bar/test.bak	test.bak

GlobMapper

The *GlobMapper* works like the *copy* command in DOS:

```
<copy todir="/tmp">
  <mapper type="glob" from="*.php" to="*.php.bak"/>

  <fileset refid="someid" />
</copy>
```

This will change the extension of all files matching the pattern **.php* to *.php.bak.*

Examples

```
<mapper type="glob" from="*txt" to="*txt.bak"/>
```

Applying the mapper, you will get the following results from the following filenames:

From	To
test.txt	test.txt.bak
./foo/bar/test.txt	./foo/bar/test.txt.bak
mytxt	mytxt.bak
SomeClass.php	*ignored*, SomeClass.php

IdentityMapper

The *IdentityMapper* will not change anything on the source filenames.

MergeMapper

The *MergeMapper* changes all source filenames to the same filename.

Examples

```
<mapper type="merge" to="test.tar"/>
```

Applying the mapper, you will get the following results from the following filenames:

From	To
test.txt	test.tar
./foo/bar/test.txt	test.tar
mytxt	test.tar
SomeClass.php	test.tar

RegexpMapper

The *RegexpMapper* changes filenames according to a pattern defined by a regular expression. This is the most powerful mapper and you should be able to use it for every possible application.

Examples

```
<mapper type="regexp" from="^(.*)\.conf\.xml" to="\1.php"/>
```

The mapper as above will do the following mappings:

From	To
test.txt	*ignore*, test.txt
./foo/bar/test.conf.xml	./foo/bar/test.php
someconf.conf.xml	someconf.php

Appendix D: Project Components

This file will give you a quick introduction and a reference of the things that you may see in a build files besides tasks and types.

Phing Projects

Projects are the outermost container for everything in build files. The *<project>* tag also is the root tag in build files. It contains the name, the directory, a short description and a default target.

Project may contain task calls and targets (see below).

Example

```
<?xml version="1.0" ?>

<project name="TestProject" basedir="." default="main"
         description="This is a test project to show how to use projects ;-)">

  <!-- Everything else goes here -->

</project>
```

Attributes

Name	Type	Description	Default	Required
basedir	String	The base directory of the project, i.e. the directory all paths are relative to.	n/a	Yes
default	String	The name of the target that is executed if none is explicitely specified when calling Phing	all	No
description	String	A free text description of the project	n/a	No
name	String	Name of the project	n/a	Yes

Targets

Example

```
<target if="lang" unless="lang.en" depends="foo1,foo2"
       name="main" description="This is an example target" >

  <!-- everything else goes here -->

</target>
```

The target defined in the example above is only executed, if the property *${lang}* is set and the property *${lang.en}* is not set. Additionally, id depends on the targets *foo1* and *foo2*. That means, the targets *foo1* and *foo2* are executed before the target *main* is executed. The name of the target is *main* and it also has a description.

Attributes

Name	Type	Description	Default	Required
depends	String	One or more names of targets that have to be executed before this target can be executed.	n/a	No
description	String	A free text description of the target.	n/a	No
if	String	The name of the property that is to be set if the target is to be executed.	n/a	No
name	String	The name of the target	n/a	Yes
unless	String	The name of the property that is to be set if the target is not to be executed.	n/a	No

Appendix E: File Formats

Build File Format

The following XML file shows a skelleton build file, that only contains a project and a target. See the references for the Phing Types and Tasks for information on how to use them.

```
<?xml version="1.0" ?>

<!--
  The root tag of each build file must be a "project" tag.
-->
<project name="(projectname)" [basedir="(projectbasedir)"]
         [default="(targetname)"] [description="(projectdescription)"]>

  <!--
    Type and task calls here, i.e. filesets, patternsets,
    CopyTask calls etc.
  -->

  <target name="(targetname)" [depends="targetname1,targetname2"]
          [if="(ifproperty)"] [unless="(unlessproperty)"]>
    <!--
      Type and task calls here, i.e. filesets, patternsets,
      CopyTask calls, etc.
    -->
  </target>

  <!--
    More targets here
  -->
</project>
```

Property File Format

Property Files define properties. Properties are stored in key/value pairs and may only contain plain text. The suffix of these files should be *.properties*, the default Property File for a Build File is *build.properties*

```
# Property files contain key/value pairs
key=value

# Property keys may contain alphanumeric chars and colons, but
# not special chars. This way you can create pseudo-namespaces
myapp.window.hsize=300
myapp.window.vsize=200
myapp.window.xpos=10
myapp.window.ypos=100

# You can refer to values of other properties by enclosing their
# keys in "${}".
text.width=${myapp.window.hsize}
```

```
# Everything behind the equal sign is the value, you do
# not have to enclose strings:
text=This is some text, Your OS is ${php.os}

# I guess that is all there is to property files
```

Bibliography

International Standards

[osi−model]
OSI (Open System Interconnect) Model

◊ http://www.iso.org
◊ http://www.instantweb.com/foldoc/foldoc.cgi?OSI

[xml10−spec]
W3C XML 1.0 Specifications

◊ http://www.w3.org/XML/

[unicode]
Unicode

◊ http://www.unicode.org

Licenses

[gnu−lgpl]
The GPL (Gnu Lesser Public License)

◊ http://www.gnu.org/licenses/lgpl.html

[gnu−fdl]
The Gnu FDL (Free Documentation License), the license used for this documentation

◊ http://www.gnu.org/licenses/fdl.html

Open Source Projects

[bc]
Binarycloud

◊ http://www.binarycloud.com
◊ http://binarycloud.tigris.org

[w3c−tidy]
HTMLTidy, a W3C (x)HTML and XML syntax checker and code beautifier

◊ http://www.w3c.org/People/Ragget/tidy/

[phpdoc]
The PHPDoc Project

◊ http://www.phpdoc.de

[phpclasses]
Manuel Lemos' PHPClasses Repository

◊ http://www.phpclasses.org

[pear]
PEAR (Php Extension Archive Repository)

◊ http://pear.php.net

[ant]
Ant, a Java Build Tool, the main inspiration for Phing

◊ http://ant.apache.org

[gnumake]
GNU make, an inspiration for Phing

◊ http://www.gnu.org/software/make/make.html

[pollo]
Pollo, a visual editor for XML files. A schema to edit phing build files is shipped with Phing.

◊ http://pollo.sourceforge.net

[gingerall]
Ginger Alliance − Home Of Sablotorn

◊ http://www.gingerall.com

[php]
The PHP homepage − PHP Hypertext Preprocessor

◊ http://www.php.net

[gnu]
The GNU (GNU's Not Unix) Organization

◊ http://www.gnu.org

[phing]
Phing (PHing Is Not Gnumake)

◊ http://phing.info

Manuals

[cvs−howto]
Short manuals for CVS

◊ http://www.ucolick.org/~de/CVSbeginner.html

[cvs−tigris]
CVS and tigris.org

◊ http://binarycloud.tigris.org/project/www/docs/ddUsingCVS_command−line.html

Other Resources

[javadoc]
Sun Javadoc

◊ http://java.sun.com/j2se/javadoc/

[zend]
Zend Technologies, Ltd.

◊ http://www.zend.com