



Quick Start Guide

JWare/AntXtras Foundation

Release v0.5



iDare Media, Inc.
Attn: JWare Projects
1133 Broadway, Suite 706
New York, NY. 10010
jware.idaremedia.com

Simone Cato
Version 3.0, Revised May 2005

This document is copyright 2002-2005, iDare Media, Inc.

The following are trademarks of other organizations:

- Java, J2EE, J2SE, Sun, Sun Microsystems and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.
- HTML, XML, XSL, and related standards are trademarks or registered trademarks of the Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, or Keio University on behalf of the World Wide Web Consortium.
- UNIX is a registered trademark of The Open Group.
- Linux is freely distributed under the GNU General Public License (GPL). The term "Linux" is a registered trademark of Linus Torvalds, the original author of the Linux kernel.
- Windows, Microsoft, and Win32 are registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.
- All other trademarks cited in this document are the property of their respective owners.

Table of Contents

Introduction	5
Terms Of Use.....	5
Installation.....	6
Overview.....	7
Fixture-Control.....	7
Execution-Rules	8
Flow-Control.....	9
Feedback (Log4Ant).....	10
Helpers	10
Why Read The Rest Of This Guide?	11
Basic Concepts Explained	12
Build Iteration	12
Remote Observability	12
URL-Based Locations.....	13
Scoped Configuration.....	13
TaskSets	13
Variable Properties	14
Flexible Value Substitution	14
Environment Snapshots.....	15
Noise Levels	16
Step Launchers (aka Project Cloning Trickery).....	16
Build File Excerpts.....	17
Example: Putting AntX In Its Own Namespace.....	17
Example: Enforcing Certain Requirements.....	17
Example: Defaulting Non-Essential Properties (1)	18
Example: Defaulting Non-Essential Properties (2)	18
Example: Declaring The AntX Step-Launcher Construct.....	19
Example: Creating Nested (Hidden) Targets.....	19
Example: Conditionally Executing Tasksets Within A Target (1).....	20
Example: Conditionally Executing Tasksets Within A Target (2).....	21
Example: Defining Error Handling For A Set of Tasks (1).....	22
Example: Defining Error Handling For A Set of Tasks (2).....	22
Example: Using Variable Properties.....	23
Example: Creating A Self-Verifying Test Build File	24
Example: Doing Lots Of Stuff At Once.....	24
AntX User Properties.....	27
Examples	27
Internal AntX Properties.....	27
AntX Task and Macro Definitions	28
AntX Type Definitions.....	30
AntX Condition Definitions	31

Introduction

We originally created the JWare/AntXtras packages as part of our internal build environment. Because a large percentage of the Ant components were generic and could be used by other Ant users, we released them under the JWare open-source umbrella. The JWare/AntXtras *Foundation* package (AntXtras or AntX) includes our most independent and reusable Ant components divided into five main categories: *Fixture-Control*, *Execution-Rules*, *Flow-Control*, *Feedback*, and *Helpers*. You can select individual AntX components or use the entire package in your own Ant build scripts.

This document will introduce you to AntX: it will give you an overview of the fundamental AntX design concepts, describe briefly the most important AntX components, and demonstrate some common uses of AntX in a build program. Read the “JWare/AntXtras Foundation User Guide” for comprehensive documentation for all AntX tasks, types, and options. Read the last three sections of this document for the standard AntX component definitions; you can use this information to change any AntX task, type, or condition name mapping to your liking. Refer to your distribution’s AntX API javadocs and test suite source code for information about using the AntX Java packages to write your own tasks.

AntX and its documentation are written for experienced Ant users and developers. AntX is not intended for someone learning Ant or for someone evaluating Ant (unless you are concerned about the issues that AntX specifically addresses).

Terms Of Use

JWare/AntXtras Foundation, binary and source form, is released under the Free Software Foundation’s GNU LGPL; a copy of this license can be found on the Free Software Foundation’s website at <http://www.fsf.org/licenses/lgpl.html>. *Please read the LGPL carefully before using any of the JWare/AntXtras source in your own application.*

JWare/AntXtras Foundation uses software developed by and on behalf of the Apache Software Foundation, <http://www.apache.org/>.

The JWare website and JWare Series documentation are copyright iDare Media, Inc. and are not released under the GNU LGPL. Redistribution without the express permission of iDare Media, Inc. is forbidden.

Installation

The JWare/AntXtras Foundation (AntX) installation is similar to that of any optional Ant package. The following instructions describe how to install and verify AntX in your Ant runtime environment.

1. If you haven't already done so, download and install an [Ant distribution](#)—at least version 1.6.2. Verify that Ant is properly installed. AntX only uses components of the standard Ant distribution; optional tasks are not required.
2. Download, verify, and extract an [AntX distribution](#). For the first-time user we suggest you download the binary distribution that includes several sample build files— the samples can be used to verify the installation is working properly.

F If you must manually generate all binaries for your environment, download the source-only distribution. The included “ez-build.xml” [Ant](#) build file can generate a default distribution from the source. Be sure you configure the “ez-build.properties” file for your environment. You must use javadoc version 1.4 or later to generate the javadocs using the ez-build.xml Ant file.

In the remaining steps we will use `<ANTX_DIR>` to refer to the directory into which the AntX distribution was extracted or built.

3. Update your Ant runtime environment to include the two AntX jar files `<ANTX_DIR>/lib/JWare_apis.jar` and `<ANTX_DIR>/lib/AntX_tasks.jar` in your Ant classpath. There are several ways of telling Ant about third-party jar files; the easiest method is to copy the AntX jars files into your Ant distribution's lib directory. A safer approach is to install AntX in its own location and update the CLASSPATH used when you run Ant (for example, by using the `-lib` option to run Ant or a custom `$HOME/.antrc` file).
4. Verify the AntX tasks are accessible from Ant. The easiest way to do this is to run Ant against one of the sample or test build files from the distribution. From within the `<ANTX_DIR>/etc` directory, run `'ant -f antx-install-check.xml'`. This sample build file declares several AntX task and type definitions. If Ant is unable to locate the AntX classes, even this simple build file will fail.
5. Read the rest of this guide for an overview of the most useful AntX tasks.
6. Include the AntX components in your build file(s) using a standard `<taskdef>` declaration like: `<taskdef resource="com/idaremedia/antx/antlib.xml"/>`.
7. Start using AntX!

Overview

The JWare/AntXtras Foundation tasks (AntX) are divided into five main categories: *Fixture-Control*, *Execution-Rules*, *Flow-Control*, *Feedback*, and *Helpers*. The *main* components in each category are described below. Read the last three sections of this document for a complete listing of all available tasks, types, conditions, macrodefs, and presetdefs. In the following descriptions, the term “taskset” refers to the standard AntX task container; see the “Basic Concepts Explained” section for additional information.

Fixture-Control

The Fixture-Control components give you an informal framework within which you can define your build’s execution environment globally or at a local, taskset-specific level. Because AntX does not require a particular project structure, the fixture components are loosely coupled. You can use the fixture-control items as part of a new build system or to support an existing build environment’s architecture.

Task/Type Name	Description
msgsbundle, strings, properties, etc.	Shareable, structured items containing build fixture data (messages, URLs, files, etc.). Because structured items are usually verified as they are created instead of when they are (first) used, you notice invalid declarations or modifications immediately.
managebundles, manageuris, etc.	Administrator tasks that let you define primordial build fixture settings. Like the standard <code><typedef></code> you can use any administrator task as part of an <code><antlib></code> script that is loaded by your build’s initialization.
autorun	A taskset that you include in a standard <code>antlib</code> . The taskset’s contents are automatically executed when the <code>antlib</code> is loaded.
isolate, overlay	Two tasksets that let you control what a set of nested tasks see and affect in the active project’s environment. The <code><isolate></code> taskset lets you create a “bubble” that shields the project from most modifications caused by the nested tasks. The <code><overlay></code> task lets you hide, overload, or add selective bits of fixture (like properties) that the nested tasks might use.
overlay-msgs, overlay-emit	Configuration tasksets that let you overlay custom types of information for enclosed task.
assign, assignimport, unassign	Tasks to create and edit variable properties (also called exported properties) across task, target, and (sub)project boundaries. Also very useful when you need to create automated test scripts or to time the duration of tasks.
datadef	A task that lets you insert data objects into the project’s reference table without having to predeclare a global id regardless of whether the enclosing target is ever run.
copyproperty, copyreference, copymsg	Tasks to copy resource messages, other properties, variables, and references between standard project properties. Useful when combined with AntX’s flow-control tasks to pass-along fixture information to sub-projects.

Task/Type Name	Description
capturelogs, captureoutputs, copylogged, evaluatelogged	Tasks that capture and save task outputs and byproducts for further processing or examination. These tasks are also very useful when you want to capture and verify task output in automated test scripts.

Execution-Rules

The Execution-Rules components supplement the standard `<condition>`, `<available>`, and `<fail>` tasks to create a powerful rules framework that makes writing self-verifying build scripts very easy. The main build rules assertion tasks, `<assert>` and `<assertlogged>`, are also very useful for testing your own scripts or testing third-party tasks.

While the AntX rules framework include a large collection of new condition tests, its main tasks are completely compatible with Ant's builtin conditions and any custom conditions that implement the standard Ant condition interfaces. The AntX sister project AntXtras/Oofs also includes a large set of new conditions for use with AntX.

Task/Type Name	Description
assert, fixturecheck	A combination of the <code><condition></code> and <code><fail></code> tasks; if the condition is <i>false</i> the build process is immediately stopped. These tasks are designed to provide an extensive range of test conditions with minimal build script. Assertions can be disabled; fixture checks cannot.
batchchecks	A wrapper taskset for <code><assert></code> s and <code><fixturecheck></code> s that lets you execute a set of checks in a single pass. If at least one check fails, the batch operation fails after all checks performed.
prefer	A combination of the <code><condition></code> and <code><property></code> tasks; if the condition is <i>false</i> , the task will (re) set a property to some default value. This task allows a build script to specify default values for undefined build properties while issuing the appropriate warnings to the build's monitor.
rule, rulemacro	A shareable condition definition comprised of either all requirements (<code><require></code>) or all preferences (<code><prefer></code>). A rule is only useful if referenced by an AntX build-rule task. A <code>rulemacro</code> lets you parameterize a rule.
tally	A collection of condition evaluators. Extends the standard <code><condition></code> to include all of the additional AntX conditions and support true as well as false update properties. You can also run tallies repeatedly from AntX value URIs.
tallyset	A shareable collection of is-available type conditions. Only useful if referenced by an AntX <code><tally></code> task or another <code><tallyset></code> .
matches	A condition that updates a project property if a value matches a specified regular expression (or not). Can also be used as an embedded condition or run repeatedly from AntX value URIs.
criteria	A reusable test definition that can contain items other than conditions. Useful if you want to define non-trivial loop termination tests that are called repeatedly and have side-effects.

Flow-Control

The Flow-Control tasks help you to manage your build process's execution flow by providing several looping and error handling constructs missing from the standard Ant tasks. The looping tasks do not intend to make Ant a scripting language; instead, their main goal is to make writing *templated* build scripts easier. Similarly, many of the tasks' error-handling features aim to make writing *error-tolerant* and auto-correcting build scripts easier.

Task/Type Name	Description
protect	A taskset that captures build exceptions generated from nested tasks and allows the script-author to specify how to handle the exception. Similar in functionality to the Java language <i>try-catch-finally</i> construct.
do	A taskset that will be executed only if a set of conditions have been satisfied. Similar to an <i>if-then</i> Java-style construct but with the form of the standard Ant <i>if/unless</i> construct.
dowhile	A taskset that will be executed as long as a referenced condition returns <code>true</code> . Similar in functionality to the <i>while</i> Java-style construct.
domatch	A taskset that tries to match a selection criterion to a set of named, nested tasksets. If a choice is matched, the <code><domatch></code> executes the selected taskset's nested tasks. Similar in functionality to the Java language <i>switch-case-default</i> construct.
step	A nested target. Only useful when combined with either the AntX <code><call></code> , <code><callinline></code> , or <code><callforeach></code> flow-control tasks.
call, callinline	Tasks that execute a sequence of steps, targets, or macros. Provides a set of error-handling options if one or more call-targets fail. Supplements the standard <code><antcall></code> task.
callforeach	An extension of the base <code><call></code> and <code><callinline></code> tasks that adds several looping options. Includes support for iterating a series of strings, a <code>FileSet</code> , a <code>DirSet</code> , a <code>Path</code> , or a bounded integer range among others.
foreach	An alternative to <code><callforeach></code> that defines the tasks to be executed within the <code>foreach</code> block itself. Similar in form to the Java language looping/block constructs. You should use this task to minimize the number of top-level targets created just to support a looping construct.
stop or stopbuild	Like a <code><fail></code> but with support for resource bundle-based messages and custom build exception types.

Feedback (Log4Ant)

The Feedback components help you create a robust logging and reporting framework for your build process. The framework supports resource bundle-based message templates and includes a complete bridge between Ant and another Apache project, [Log4j](#). The feedback framework can be extended to support other logging or reporting systems like Sun's J2SE 1.4 logging package.

Task/Type Name	Description
show	A <code>ResourceBundle/MessageFormat</code> version of <code><echo></code> . Message templates are located in external properties resource bundles (files, URLs, or class resources). Supports up to two dynamic message arguments.
emit, checkpoint	Tasks that broadcast build information to external build monitors and reporting systems. A checkpoint is a specialized <code><emit></code> that is used to broadcast diagnostic tracing information. Off the shelf, works with most Log4J appenders and viewers like "Chainsaw".
emitconfiguration	A shareable data object that defines a set of feedback configuration that any other feedback component can use to broadcast information. By default works with a Log4j-based logging system but is easily extended for other logging systems.
emitlogs	A special taskset that repeats messages logged to the standard Ant infrastructure to any listening build monitors. Basically lets you extend Ant logging to include any Log4J appender or viewer.
emitmappings	A shareable data object that maps Ant log messages to specific logging system groupings. Messages are mapped based on their origin (like target name and task class), or by their contents (using regular expression matching). Supplements the <code><emitlogs></code> task.

Helpers

The Helpers provide useful standalone types and utility tasks for build scripts and other task implementations. Several helpers were developed as diagnostic or test aids so their usefulness in regular build files is minimal.

Task/Type Name	Description
printenv [†] , print [†]	Tasks to display specified project properties, variable properties, references, and other fixture information. Also useful for debugging the <code><rules></code> type and the <code><tally></code> task.
tempdir, newtempdir	Tasks that locate a suitable temporary directory into which scratch files and directories can be safely created. The new directories are based on the platform-specific temporary directory (for example, <code>\$TMPDIR</code> on Unix systems).
newfile, newtempfile, mkdirs	Tasks that create new [temporary] files and directories. New temporary files are automatically marked for deletion on normal termination of the Ant runtime. Can also initialize the new file's contents from a source prototype file or resource. Usually used with the <code><tempdir></code> task.

Task/Type Name	Description
printerror [†]	Task that displays contents of an <code>ErrorSnapshot</code> . An <code>ErrorSnapshot</code> is an AntX data type used to capture failure information within a build process.
evaluatelogged	Task that evaluates a set of captured log information for some script-supplied acceptance criteria. Usually used to determine if a task such as <code><javadoc></code> or <code><javac></code> has issued non-fatal warnings that should be captured.
listdir	Task that generates a directory listing and saves it into a local project string list. You can then use the list like you would an statically defined string list.
vendorinfo	Task that copies product version and build information to a set of project properties. Currently works for any active JWare component; easily extended to include other projects.

[†]These tasks are designed as diagnostic and test helpers.

Why Read The Rest Of This Guide?

Because, in a nutshell, all the other AntX documentation assumes you have. Instead of reiterating the same underlying design principles and terminology in all the documentation, the remainder of this document is the only source for this information.

The next section, “[Basic Concepts Explained](#)” describes some fundamental concepts that will help you understand why the various tasks are designed and implemented the way they are. The AntX User Guide relates these concepts to specific build problems we were trying to solve. The “[Build File Excerpts](#)” section highlights how AntX tasks, in combination with standard Ant tasks, can be used to implement certain functionality in a build system. (The AntX User Guide includes additional task-specific examples). And finally, the [last four sections](#) list the complete set of AntX task, type, condition, and user property definitions.

If you intend to extend AntX programmatically, you should also read the AntX javadoc package descriptions for complete documentation of all task options, support classes, and implementation restrictions.

Basic Concepts Explained

The following are some basic concepts embodied in the JWare/AntXtras Foundation product. These ideas (and our terminology) are explained here to help you understand the companion to this document “The JWare/AntX Foundation User Guide” and the AntX source code’s structure (relevant only if you develop with AntX or one of its associated projects).

Build Iteration

In much of the public documentation, Ant is largely described in terms of a project with discreet targets and tasks. Ant targets and projects are effectively static; they don’t do anything by themselves. When a user runs Ant, the project and the selected target(s) combine and become the local fixture in which sets of instructions, the Ant tasks, are executed. At least, this is Ant’s view of the universe.

From the Ant user’s perspective, the project but especially the selected target’s name represents *the overall task to be accomplished, or goal*. This is why user-selected Ant targets are often named to reflect the reason for or side effects of executing the target’s tasks. So, interestingly, from the user’s point of view *the selected target is the task*—that Ant actually executes instructions across multiple JVMs, projects, and other targets is of little importance as long as the original request is serviced.

AntX calls this request servicing the *build iteration*. A single build-iteration begins when the user invokes Ant. A build-iteration transcends the multiple Ant projects, targets, and tasks that might be created and used before it reaches its end. The JWare/AntXtras Foundation project contains rudimentary support classes that ensure that iteration-scoped information can be created and properly managed. However, only a few foundation tasks (called configuration tasks) actually embody any iteration level control. To really leverage the value of the build-iteration you should look at some of the AntX-based projects on the JWare website (www.jware.info).

Remote Observability

One of the primary objectives of the current AntX architecture is to facilitate the ability to observe a remote build-iteration as it begins, progresses, and ends. Such an ability allows you to both monitor automated builds for problems as well as create other applications that utilize the by-products generated by the various steps executed during a build-iteration.

A build-iteration’s feedback comes from two sources: the Ant runtime implementation, and the explicit information broadcast by you the script author. When Ant displays every target that is called, that is one form of feedback. When Ant executes an `<echo>` instruction that you have included, that is another form of feedback. AntX is primarily concerned with the second of the two forms. Because AntX does not change the core Ant implementation, it cannot alter when or what messages are emitted from Ant’s infrastructure. These messages must be controlled with the facilities that Ant itself provides: configurable build listeners, various run options, etc.

AntX contains several feedback tasks of which `<show>`, `<emit>`, `<emitlogs>`, `<print>`, and `<checkpoint>` are the most important. Collectively these tasks allow *you* to issue checkpoint events and to describe a build-iteration’s state in great detail to any build monitor.

URL-Based Locations

To ensure that you can use AntX in a distributed (and diverse) build environment, almost every AntX component that accepts data or configuration from an external source, lets you specify that source using a standard URL format or Java resource format. This flexibility lets you execute your AntX scripts with data that resides in local files, in classpath-specified jar files, on remote HTTP servers, or any another other datastore that you can access using a URL-encoded extraction string (like a database with a `jdbc://URL`). Many of AntX tasks also let you save location information in the form of URLs.

Scoped Configuration

By default, the Ant environment provides you with a single collection of write-once/read-many-times controls, called properties, as the primary tool to configure the build fixture.¹ Generally, global properties are a good thing. However, there are times that, as a build script author, you want to affect only the build fixture for a specific set of tasks without affecting subsequently executed tasks. AntX calls this sort of control, *scoped configuration*. Scoped configuration is most often applied at the build iteration level where it affects enclosed, executed tasks, regardless of their declared project or target. AntX includes several scoped configuration tasks; for example `<msgconfigure>` and `<isolate>`.

Do not confuse AntX's scoped configuration with Ant macros (`<macrodef>`s) and preset-definitions (`<predef>`s). Both these Ant items let you create custom, optionally parameterized, tasks and meta-tasks without proliferating "internal" top-level targets in an attempt to create some level of modularity (heavy use of `<antcall>` is a usual sign you've started doing this). In contrast, AntX's scoped configuration tasks help you alter a build's fixture temporarily, at runtime, for whatever reason; they do not make Ant task definition easier in any way. AntX configuration tasks *can* help you modularize your build script's *workflow* particularly if your build involves running many independent sub-builds.

Bubbles and Nets

Within AntX there are two special kinds of scoped configuration tasks: *bubbles* and *nets*. You use bubbles to isolate the effect of the enclosed tasks on the project's environment without creating a sub-project that must reparse the original build file. The AntX `<isolate>` task is one example of a bubble task. Nets, on the other hand, are used to filter existing or overlay custom fixture information read by the enclosed tasks. Unlike a bubble task, a net task will allow certain modifications through to the current project's environment while it blocks other information from being seen or modified. The AntX `<overlay>` task is an example of a net task.

TaskSets

Many of the AntX tasks are defined as task containers we call *tasksets*. A taskset (itself a task) usually implements some kind of configuration or enforces some kind of control over its nested tasks; we call these nested tasks the taskset's *scope of control*. Because tasksets are also tasks, tasksets can be nested.

Ordinarily for a taskset to be effective at least one nested task must be defined. However, most tasksets will function as harmless no-ops if left empty; those that do not will generate a build

¹ Actually there are three kinds of properties: user, inherited, and regular for lack of another term. These properties primarily differ in how Ant propagates them to child projects. Their other differences are not formally exposed through standard Ant tasks in any way. Most tasks create regular properties. From a property-read perspective (the build script author's perspective), the three types appear as a single source.

exception when executed. Every AntX scoped configuration task is a taskset; its configuration only applies to its nested tasks. Empty configuration tasksets do not modify the build fixture in any way.

A taskset's nested tasks are not configured until the controlling taskset says so. This means property expansion may not occur until the taskset is actually executed which is usually the desired behavior. Tasksets that are selectively executed by other tasks as called *quiet* tasksets. One example of a quiet taskset is the `<iferror>` taskset which is executed only if its controlling `<protect>` taskset detects a build failure.

Starting with Ant 1.6, task containers can contain freestanding data declarations, like classpaths or filters, as well as other tasks. Just like data declared at the top-level or target-level, declarations that include an Ant "id" attribute are visible to all tasks within the same project; however, the surrounding taskset must be executed in order for the data to be defined fully.

Conditional Execution

Ant supports a rudimentary, conditional execution capability based on the existence of a property (or lack thereof). Conditional execution is supported by all Ant targets and a few standard tasks using two attributes: "if" and "unless". AntX supports and extends this simple capability with three new flow-control tasksets: `<do>`, `<domatch>`, and `<dowhile>`—no other AntX flow-control task supports conditional execution directly unless it's necessary to maintain compatibility with existing Ant tasks.

We limit condition execution to just these tasksets for two reasons. First, it keeps things simple for engineers writing new build scripts and for those maintaining old ones. To introduce and alter a condition for running one or more tasks, you simply drop them inside one of the AntX "do" tasksets with the appropriate condition parameters attached. Second, it keeps things simple for us, the AntX developers, because it limits the number of classes we must update and test to support new (or remove old) condition parameters.

Variable Properties

Ant's project properties are write-once; once defined, they cannot be modified or removed. Generally, write-once properties suit their intended purpose but they can be troublesome in many situations. The AntX "bubbles" and "nets" task containers alleviate some of the problems, but some issues remain. In particular, creating test build scripts for new or third-party tasks would be much easier if we could "suspend" the write-once characteristic of properties. Well we couldn't, so we did the next best thing and created a script-accessible mutable property called a *variable property*.

For us, variable properties are used primarily to diagnose build scripts or to test task implementations. However, they solve some other common build problems like manipulating loop variables or controlling build meta data like build numbers, task duration, and status flags. In general, if an AntX task can read or create property values, it can also read and write variable property values instead.

Variable properties are also called *exported properties* because they can exist at the build iteration level, the thread level, or the project level (like references).

Flexible Value Substitution

The standard Ant environment supports a simple property-based value substitution mechanism. Any component's attributes can be defined in terms of existing property values but *the source*

properties must exist when the attributes' owning element is configured. While tolerable for task declarations that are often configured just before being executed(once), this limitation significantly reduces the flexibility of globally declared, shareable data objects. Because Ant's data objects are never executed, they are configured *as their script declarations are parsed*². This means if you reference a property in a global data item's attributes, that property must exist before the data item is configured, not before it is used.

Ant 1.6 introduced task macros and delayed configuration for target-bound items which alleviated some of the obvious "chicken before the egg" problems with simple property substitution. But neither macros nor delayed configuration address the need for per-use property substitution in global data objects. Even some of the new data types, like `<propertyset>`, have task-specific "recalculate" options to workaround the problem.

AntX components extend the standard Ant substitution mechanism with something called **Flexible Value declarations**. Flexible values enable attribute value substitution to occur when a data item is used or when a task is executed. Moreover, an attribute's value can be defined as a literal value, a property value, a variable property value, or a reference's string representation. This flexibility allows you to define shared data like templates. The actual values of properties, variables, etc. are determined when the data is used, not by its location in the build script.

Value URIs

Starting with 0.5, AntX now supports a special kind of property value called a **value URI**. A value URI is a short hand way of executing a builtin Ant or AntX function using the standard property reference syntax. Among its many uses, value URIs let you chain task calls without creating lots of transient project properties to hold intermediate values. The final string value of the value URI is whatever the executed function returns; for example, the `$shorttime: value URI` will return a formatted time string of the current time while the `$filenotempty: value URI` returns "true" if the named file is empty or does not exist and "false" otherwise. AntX comes with a small set of builtin value URI interpreters but you can extend the set with your own interpreter classes; see the AntX javadocs for additional information.

Environment Snapshots

AntX aims to make writing error-tolerant and auto-correcting build scripts easier and it provides several error-handling mechanisms with its components to this end. However, when a build-stopping error does occur, AntX also provides a device that captures environment information (including the actual build exception) and passes that information to logs, build monitors, and any custom build listeners. This device is called an environment or **error snapshot** and is usually created by the AntX `<protect>` taskset when it detects a build exception has been thrown by one of its nested tasks.

Although it is not required, you can explicitly define what an error snapshot contains and what bits of a snapshot gets passed-on to monitors and other build listeners. Currently snapshots can contain build-iteration fixture information like execution-location, properties, and references. Every AntX build-feedback task can display or broadcast error snapshots.

² Starting with Ant 1.6, the build file is parsed completely then the top-level components are configured or executed before the default or selected target is run. However, the effect is the same, properties must be defined before the data objects that reference them.

Noise Levels

AntX provides several build-feedback tasks (`show`, `emit`, `print`, etc.). Associated with each of these feedback tasks is a *noise level*—the level of importance associated with the event that triggered the feedback. Like the standard Ant `<echo>` task, the term “noise level” is often abbreviated to just “level.” AntX supports six noise levels: `FATAL`, `ERROR`, `WARNING`, `INFO`, `VERBOSE`, and `DEBUG`. These six levels represent a super-set of the set supported by the standard Ant logging infrastructure. When using the Ant logging system, the AntX `FATAL` level is mapped to the Ant `ERROR` level.

Within the Ant runtime, a noise level is only used to filter output to the log streams. However, when AntX is combined with a logging system like Log4J, the noise level can trigger more complex responses like email notification, or persistence in a build report database. In this scenario, how the build monitor responds to a build event is configured independent of the build script and its command options (which could be fixed if the system is automated).

Step Launchers (aka Project Cloning Trickery)

The current Ant `<antcall>` task is build file-dependent—the live project cannot be copied unless the Ant runtime has access to the original parsed build file (as stored in the `ant.file` property). Manipulations to the in-memory `Project` object, particularly the dynamic addition or removal of targets or tasks, are lost to every `<antcall>` declaration or any custom task based on `<antcall>`.³

Many of the AntX flow-control tasks extend the standard `<antcall>` task and have inherited this limitation. (Ant has no easy way to copy a live project, so *not* extending `<antcall>` does not help.) As a work-around the affected flow-control tasks rely on a special target/task combination called a *Step Launcher*. A Step Launcher is an AntX construct that serves as a call pipeline from a project to its forked projects. It allows the caller (an AntX flow-control task) to execute a specific target *or embedded taskset* in another, independent, project. This work-around is very effective but has one regrettable user-requirement: in order to use the affected flow-control tasks, you must include the special step launcher construct in your build file ([an example](#) of how to do this is included in the “Build File Excerpts” section).

Hopefully, Ant 2 will provide better support for dynamic customization of live projects. Until then, the Step Launcher is AntX’s answer to the problem.

³ Actually the Ant documentation does not detail how the target project is created except to describe how properties and references are passed-through from the caller. However, the implementation “duplicates” a project by parsing the file defined by `ant.file`, not by cloning or copying the memory-based object.

Build File Excerpts

The remainder of this section lists several excerpts of build files that use JWare/AntXtras Foundation tasks. These examples will demonstrate some of the benefits of adding AntX to your Ant tools repository. Some examples are followed by a “Points Of Interest” list that highlights items that might be otherwise overlooked. These examples do not constitute a tutorial—see the “The JWare/AntXtras Foundation User Guide” for complete task-specific examples.

Example: Putting AntX In Its Own Namespace

Here is an example that assigns the AntX library to its own namespace behind the prefix “ax”. In this script, all AntX tasks *and their nested elements* must be prefixed with “ax”. The AntX jar files must be part of Ant’s default classpath (see the Ant manual for details about the special “antlib:” namespace construct).

```
<project name="MyProject" basedir="." default="whatever"
  xmlns:ax="antlib:com.idaremedia.antx" ...> (a)
  ...
  <ax:managebundle resource="{basedir}/msgs.properties"/> (b)
  <ax:assert allset="src,lib,etc" msgid="err.misin.vars"/>
  ...
  <target name="whatever">
    <ax:show msgid="greetings" msgarg1="{DSTAMP}"/>
    ...
  </target>
</project>
```

Points Of Interest

- a) The AntX library’s namespace is declared using Ant’s special “antlib” syntax.
- b) The AntX library is actually loaded the first time the prefix is used.

Example: Enforcing Certain Requirements

Here is a build script excerpt that ensures certain properties are not defined at build-iteration start if a clean distribution is being generated. The rationale is that if these properties exist, the integrity of the build’s output is not guaranteed to be clean.

```
<rule id="clean.dist.required"> (a)
  <require failproperty="dirty.build" msgid="err.dirty.env"> (b)
    <allset malformed="reject">
      <property name="dist.type"/>
      <property name="build.type"/>
      <property name="defaults.disabled"/>
    </allset>
    <equals match="public" property="dist.type"/>
    <noneset>
      <property name="build.number">
      <property name="disable.scm">
      <property name="disable.clean">
      <property name="disable.docs">
    </noneset>
  </require>
</rule>
...
target name="dist" depends="init,..."> (c)
...
</target>
...
```

```

<target name="clean-dist">
  <assert ruleid="clean.dist.required"/> (d)
  <callinlined targets="dist"/>
</target>

```

Points Of Interest

- clean.dist.required is a reusable rule for what defines clean.
- The dirty.build flag is included to let the build script itself be tested.
- The standard distribution target dist still exists.
- The only difference between clean-dist and dist is the enforcement of the rule.

Example: Defaulting Non-Essential Properties (1)

Here is a build script excerpt that defines an internal target that ensures the dist.type property is always defined to a known value if the build allows default values. Presumably a user could execute the script with defaults off to check that he has explicitly defined all the prerequisite build properties (as should be the case for a production candidate build).

```

<target name="-default-disttype" unless="defaults.disabled"> (a)
  <assert isset="defaults.disttype"/>
  <matches pattern="internal|local|public" property="dist.type"
    falseproperty="broken.disttype"/>
  <prefer isnotset="broken.disttype" msgid="warn.unknown.disttype" (b)
    falseproperty="dist.type" default="\${defaults.disttype}"/>
</target>

```

Points Of Interest

- The target is only executed if defaults are enabled for the build-iteration.
- A warning is always sent to the Ant logging system and any attached build monitors if the dist.type needs to be fixed.

Example: Defaulting Non-Essential Properties (2)

Here is a build script excerpt that defaults certain javadoc-related properties as the normal behavior; no warnings are issued. The user can still customize these properties, but it is not required.

```

<rule id="apidocs-labels-defined"> (a)
  <fixturecheck allset="module_name,module_version"/> (b)
  <prefer isnotwhitespace="doc.title" isa="property"
    falseproperty="doc.title" default="\${module_name} API Documentation"/>
  <prefer isnotwhitespace="doc.header" isa="property"
    falseproperty="doc.header" default="\${module_name}-\${module_version}"/>
  <prefer isnotwhitespace="doc.footer" isa="property"
    falseproperty="doc.footer" default="\${module_name}-\${module_version}"/>
</rule>
...
<target name="api-docs" depends="-init-buildenv,...">
  <prefer ruleid="apidocs-labels-defined"/>
  <javadoc ...>
    <header><![CDATA[\${doc.header}]]></header>
    ...
  </javadoc>
</target>

```

Points Of Interest

- While this can be done using standard <property> declarations, the effect is not exactly the same: the whitespace check is not possible and the preference does not try to blindly

- (re) define a property (depending on the write-once characteristic of properties to block overwrites).
- b) The rule depends on the “module_name” and “module_version” properties and regular Ant property substitution. The <fixturecheck> ensures these base fixture properties exist.

Example: Declaring The AntX Step-Launcher Construct

Below are several examples that show different ways to declare the special step-launcher target that several AntX flow-control tasks need to execute embedded steps (see next example). You must install this target *in any project* that uses the <foreach> task or the <call> or <callforeach> tasks with embedded <step>s. (Otherwise, you do *not* need to do any of this.)

The first script excerpt uses the AntX step-installer antlib.xml file to automatically define the special target. The script installs AntX into the default namespace.

```
<taskdef resource="com/idaremedia/antx/install/antlib.xml" />
```

This next excerpt assigns AntX components to their own namespace but can still use the step-installer antlib to automatically define the special target.

```
<project name="Root" default="compile" xmlns:ax="antlib:com.idaremedia.antx.install">
  ...
  <ax:foreach name="hello" i="i" in="1,10" mode="local">
    <echo message="{i}" />
  </ax:foreach>
  ...
</project>
```

This last excerpt does not use either AntX antlib.xml file for whatever reason, but creates a custom task definition and calls it manually.

```
<project name="Root" basedir="." default="compile">
  <taskdef
    name="ax-install-steprunner"
    classname="com.idaremedia.antx.flowcontrol.call.StepLauncherInstallTask" />
  <ax-install-steprunner />
  ...
</project>
```

Example: Creating Nested (Hidden) Targets

Here is a build file excerpt that defines several build process steps inside a single public target. The target’s implementation is modular without exposing its internals as public-facing Ant targets. As shown, a user can only build against the actual Ant target, prepare-sources.

```
<target name="prepare-sources" unless="disable.fetch">
  <step name="verify"> (a)
    <assert msgid="err.prep.required.properties">
      <isboolean property="is.public.build" />
      <isset property="SCM_TAG_DATE" />
      <isnotwhitespace property="modules" />
      <assert isdirectory="{originals}" />
    </assert>
  </step>
  <step name="scrub"> (a)
    <assert isdirectory="{work}" />
    <delete quiet="true" includeEmptyDirs="true">
      <fileset dir="{work}/src" />
      <fileset dir="{work}/examples" />
    </delete>
  </step>
</target>
```

```

</step>
<step name="checkout"> (a)
  < cvs command="checkout -A" dest="{originals}"
    failonerror="{is.public.build}"
    package="{module_id}" tag="{cvstag}" />
  < echo message="{SCM_TAG_DATE}"
    file="{originals}/{module_id}/checkout.last" />
</step>
<step name="merge"> (a)
  < copy filtering="true" todir="{work}" includeEmptyDirs="false">
    < fileset dir="{originals}/{module_id}/java">
      < include name="src" />
      < include name="examples" />
    < /fileset>
    < filterset refid="copyfilters.exported.srcfiles" />
  < /copy>
</steps>
< copyproperty name="cvstag" value="" unless="cvstag" />
< callinline steps="verify,scrub" />
< callforeach i="module_id" list="{modules}" mode="local" (b)
  steps="checkout,merge" />
</target>

```

Points Of Interest

- a) None of the nested steps create any properties or references that the owning target needs. Because steps can be run isolated from the caller's environment, properties and references created by a step are not necessarily seen by the caller's project. (This is not true for AntX variable properties that are bound to the build-iteration.) Also note that property substitution occurs before each step is executed not when the step is encountered in the target.
- b) The `verify` and `scrub` steps are called once, but the `checkout` and `merge` steps are called for each module in the `modules` property. In this example, all steps are executed within the current project, no child projects are created.

Example: Conditionally Executing Tasksets Within A Target (1)

Here is an example top-level scriptlet `initalways` that uses some AntX flow-control tasks to execute a group of tasks only if certain properties are defined to a particular value.

```

<do taskname="initalways">
  <tstamp/>
  <managebundle file="resources/msgs.properties"/> (a)
  <assert ruleid="required.build.properties"/>

  <domatch property="dist.type"> (b)
    <equals value="local"> (b)
      <property name="defaults.isDebugEnabled" value="true"/>
      <property name="defaults.isopt" value="false"/>
      <property name="defaults.access" value="package"/>
    </equals>
    <equals value="public"> (b)
      <property name="defaults.isDebugEnabled" value="false"/>
      <property name="defaults.isopt" value="true"/>
      <property name="defaults.access" value="public"/>
      <property name="is.public.build" value="true"/>
    </equals>
    <default>
      <stop msgid="err.unknown.disttype" msgarg1="{dist.type}"/> (c)
    </default>
  </domatch>
  ...
</do>

```

Points Of Interest

- a) The build's resource bundle-based messages are defined as early as possible in the build-iteration so other tasks can use them. Ideally this scriptlet is executed immediately after the required task definitions have occurred.
- b) The value of the `dist.type` property determines what tasks get executed next.
- c) The build stops immediately if the `dist.type` property is undefined or not recognized.

Example: Conditionally Executing Tasksets Within A Target (2)

Here is an internal target, `-run-acceptance-tests`, that runs a set of acceptance tests if either of two testing engines is present (*Junit* or something called *PET*). The build script enforces the rule that for public distribution builds some kind-of acceptance tests must be run; for other build types the target's caller gets to decide what to do if no testing engine is present. (Note some of the 'is-available' tallying has been omitted for space considerations.)

```
<target name="-run-acceptance-tests" depends="-tally-environment">
  <assert ruleid="required.tests.properties"/>
  <do if="junit.present" unless="disable.junit" > (a)
    <assert isclass="junit.framework.Assert"/>
    <emit msgid="msg.running.acceptance.junit"/>
    <junit .../>
    <junitreport .../>
  </do>
  <do if="pet.present" unless="disable.pet" > (a)
    <prefer httpalive="{build.PET.server}" falseproperty="pet.upload.disable"/>
    <emit msgid="msg.preparing.PET.files"/>
    <pet-prepare .../>
    <do unless="pet.upload.disable">
      <pet-run .../>
    </do>
  </do>
  <tally trueproperty="no.test.engines" > (b)
    <noneset>
      <property name="junit.present"/>
      <property name="pet.present"/>
    </noneset>
  </tally>
  <do if="no.test.engines">
    <emit msgid="err.no.tests.engines" level="warning"/>
    <tally logic="or" trueproperty="require.test.engines" > (c)
      <isset property="is.public.build"/>
      <equals match="public" property="dist.type"/>
    </tally>
    <stop if="require.test.engines" msgid="err.signoff.required" fatal="yes"/>
  </do>
</target>
```

Points Of Interest

- a) Specific tests are run if their support environments are present and that test type has not been explicitly disabled.
- b) Only the `*.present` flags are checked; the `disable.*` flags are not. This example works like the previous example that disallowed certain disable flags if a clean public distribution build was required so redundant checks are not necessary.
- c) The tallied property `no.test.engines` is set for a non-public build caller to see.

Example: Defining Error Handling For A Set of Tasks (1)

Here is an example macro `getversion` that tries to retrieve a product's version information from a CVS repository. If it cannot get the information, it defines different error handling for production builds and internal developer builds.

```
<presetdef name="co-version-file">
  <cvs command="update -A" package="{module}" dest="{originals}" .../>
</presetdef>

<macrodef name="getversion">
  <attribute name="module"/>
  <attribute name="property" default="build.version"/>
  <sequential>
    <assert isdirectory="{originals}"/>
    <property name="@{property}.F"
      value="{originals}/{module}/version.in"/>
    <domatch property="dist.type">
      <like value="public|production">
        <protect> (a)
          <co-version-file/>
          <loadfile property="@{property}" srcfile="@{property}.F"/>
          <iferror capturesnapshotid="last.err"> (a)
            <emit level="fatal" thrown="last.err" msgid="err.vers"/>
          </iferror>
        </protect>
      </like>
      <like value="local|internal">
        <protect haltiferror="no"> (b)
          <co-version-file/>
          <loadfile property="@{property}" srcfile="@{property}.F"/>
          <iferror capturesnapshotid="last.err"> (b)
            <emit level="error" thrown="last.err" msgid="err.vers"/>
            <property name="@{property}" value="0.00.0 dev0 (test)"/>
          </iferror>
        </protect>
      </like>
      <otherwise>
        <stop msgid="err.unknown.disttype" msgarg1="{dist.type}" fatal="yes"/>
      </otherwise>
    </domatch>
  </sequential>
</macrodef>
```

Points Of Interest

- a) For a public distribution build, if the `cvs` or `loadfile` tasks generate an error, an error snapshot is captured and sent to any listening build monitors (log4j-based or other). The build is stopped (the `<protect>`'s "haltiferror" flag is on by default).
- b) For a local or developer build, if the `cvs` or `loadfile` tasks generate an error, an error snapshot is also captured and broadcast, **but the build does not stop**. Instead it defaults the product number to a "0.00.0 dev0 (test)" version and continues.

Example: Defining Error Handling For A Set of Tasks (2)

Here is an example target `compile-subprojects` that tries to compile a set of sub-projects while maintaining a single overall `build.number` using a combination of the `<protect>` task and AntX's variable properties. If all the sub-projects compile successfully the `build.number` is incremented by one, otherwise it is rolled back by one.

```
<target name="compile-subprojects" depends="...">
  <assert isset="build.number.F" msgid="err.need.buildnum.file"/>
  <do unless="build.number">
    <buildnumber file="{build.number.F}"/>
  </do>
```

```

<assert isnumeric="${build.number}" gte="0" msgid="err.bad.bn">
<assign var="bn" fromproperty="build.number" scope="project"/> (a)

<protect>
  <checkpoint msgid="subbuilds.started"/>
  <subant genericantfile="build.xml" inheritall="no"> (b)
    <property name="build.number" value="${build.number}"/>
    <filelist dir="${basedir}" files="${modulelist}"/>
  </subant>
  <checkpoint msgid="subbuilds.finished"/>

  <iferror> (c)
    <emit level="warning" msgid="warn.rollback.buildnum"/>
    <assign var="bn" op="-" scope="project"/>
  </iferror>
  <always> (d)
    <assign var="bn" op="++" scope="project" copyproperty="next.bn"/>
    <propertyfile file="${build.number.F}" comment="...">
      <key name="build.number" value="${next.bn}"/>
    </propertyfile>
    <emit msgid="rollover.buildnum" msgarg1="${next.bn}"/>
  </always>
</protect>
</target>

```

Points Of Interest

- a) Initializes a local **variable** build number from the **property** `build.number`. AntX build-iteration variables exist in a different namespace from regular Ant properties.
- b) Each sub-project may try to read its own build number. Because of the way the `<buildnumber>` task works, each time the file is loaded the number contained in the file is increased by one. The `<always>` handler will correct this undesirable side effect.
- c) This particular script wants the build number to rollback in the event of any build error. So if any build error occurs, the `<iferror>` handler rolls back the top-level `build.number` by one in preparation for the next step (d), which is always performed.
- d) The build number file is maintained according to the top-level `build.number`. If an error has occurred the final build number will be same as when the target was called (thanks to the `<iferror>` manipulation); otherwise it is incremented **by one**.

Example: Using Variable Properties

Here is a build file excerpt that uses AntX variable properties to approximate how long it takes to compile a set of java files. The information is displayed on the build machine's console.

```

<target name="compile-apis" depends="...">
  <show msgid="banner.compiling"/>
  <assign var="time" op="now"/> (a)
  <copyproperty name="compile.start" variable="time" transform="datetime"/>

  <javac srcdir="${module_src}" destdir="${module_classes}"
    debug="${javac.isdebug}" optimize="${javac.isopt}"
    deprecation="${javac.isdebug}"
    classpath="${build.classpath}"
    includes="**/*.java"
    excludes="**/tests*/,**/doc-files/">
    <compilerarg compiler="jikes" value="+Z"/>
  </javac>

  <assign var="time" op="-now"/> (b)
  <copyproperty name="compile.duration" variable="time" transform="duration"/>
  <show msgid="banner.compiling.done"
    msgarg1="${compile.start}" msgarg2="${compile.duration}"/> (c)
</target>

```

Points Of Interest

- a) The start time is captured to both the `time` variable and the (immutable) property `compile.start`.
- b) The compile's duration is based on the timestamp stored in the `time` variable.
- c) Both the start-time and the duration can be displayed in the build-iteration's logs.

Example: Creating A Self-Verifying Test Build File

Here is an excerpt from a test build file for the AntX `<protect>` task implementation; the target "testIfErrorBeforeAlways" verifies that the nested `<iferror>` is always executed before the nested `<always>` block. Note that this target is a complete self-verifying test case. The sample shows how the various assertion tasks can help test other task implementations as well as regular build files.

```
<target name="testIfErrorBeforeAlways">
  <assert isnotset="I.died"/> (a)
  <capturelogs>
    <protect haltiferror="no">
      <iferror failproperty="I.died" capturethrown="the.err"/>
      <always>
        <assert isset="I.died"/>
        <assert isref="the.err" iskindof="true" class="java.lang.Exception"/>
        <echo message="helloworld(((ALWAYS)))"/>
      </always>
      <assert istrue="false"/> <!--I WILL FAIL à (b)
    </protect>
    <assertlogged value="helloworld(((ALWAYS)))"/>
  </capturelogs>
  <printerror snapshotid="the.err" if="tests.verbose"/> (d)
</target>
```

Points Of Interest

- a) Verifies the test fixture is as expected.
- b) This should fail and never get to I.
- c) The `<never>` ensures the requirement posed by (b).
- d) This lets me manually examine what's been generated by (b).

Example: Doing Lots Of Stuff At Once

Here is an example that previews two versions of a single source directory ("debug" and "optimize"). For each build type, the `preview.signoff` target compiles the common source with different compiler options and runs any embedded unit tests (whose results are optionally converted to an HTML-based report). To use this target a caller must supply three properties: `module_src` (location of java files), `stem` (package prefix or "*" for all), and `build.resources` (location of build resource files).

```
<presetdef name="itid-stamp">
  <tstamp>
    <format property="_ITID" pattern="MMM-dd-yyyy'_ 'hhmm"/>
  </tstamp>
</presetdef>
...
<target name="preview.signoff">
  <copyproperty name="_source" value="${module_src}" unless="_source"/>
  <assert isdirectory="${_source}"/>

  <itid-stamp/>
  <tempdir subdirectory="preview_${_ITID}" pathproperty="preview.root"/>
```

```

<show msgid="info.preview.where" msgarg1="${preview.root}"/>
<foreach name="compile-test" i="build.type" list="debug,optimize"
  tryeach="true" failproperty="notclean-compile-test">

  <show msgid="banner.start.${build.type}"/>
  <property name="_rootdir" value="${preview.root}/${build.type}"/>

  <domatch property="build.type">
    <equals value="debug">
      <property name="javac.isdebug" value="true"/>
      <property name="javac.isopt" value="false"/>
      <property name="javac.classpath" value="${java.class.path}"/>
    </equals>
    <equals value="optimize">
      <property name="javac.isdebug" value="false"/>
      <property name="javac.isopt" value="true"/>
      <property name="javac.classpath" value="${java.class.path}"/>
    </equals>
  </domatch>

  <property name="_envfile" value="${_rootdir}/.antenv"/>
  <printenv properties="all" variables="all" tofile="${_envfile}"/>

  <property name="_classes" value="${_rootdir}/classes"/>
  <mkdir dir="${_classes}"/>

  <javac srcdir="${_source}" destdir="${_classes}"
    debug="${javac.isdebug}" deprecation="${javac.isdebug}"
    optimize="${javac.isopt}"
    depend="false"
    classpath="${javac.classpath}"
    includes="${stem}/**/*.java"
    excludes="**/*.*/**,*/doc-files"/>

  <property name="_results" value="${_rootdir}/results"/>
  <mkdir dir="${_results}"/>

  <junit fork="yes">
    <classpath>
      <pathelement location="${_classes}"/>
      <pathelement path="${javac.classpath}"/>
    </classpath>
    <formatter type="xml"/>
    <batchtest todir="${_results}"
      failureproperty="notclean-junit">
      <fileset dir="${_classes}">
        <include name="${stem}/**/*.Test.class"/>
        <exclude name="**/*Skeleton*.class"/>
      </fileset>
    </batchtest>
  </junit>
  <do if="notclean-junit">
    <assign var="notclean-junit" value="true"/>
  </do>

  <do ifTrue="enable.preview.reports">
    <assert isdirectory="${build.resources}/xsl"/>
    <property name="_reports" value="${_rootdir}/reports"/>
    <mkdir dir="${_reports}"/>
    <junitreport todir="${_results}">
      <fileset dir="${_reports}">
        <include name="TEST-*.xml"/>
      </fileset>
      <report format="frames" todir="${_reports}"
        styledir="${build.resources}/xsl"/>
    </junitreport>
  </do>
</foreach>

```

```
<tally trueproperty="notclean-preview" logic="or">
  <issettrue variable="notclean-junit"/>
  <issettrue property="notclean-compile-test"/>
</tally>
<stop if="notclean-preview" msgid="err.preview.unsuccessful"/>
</target>
```

AntX User Properties

The table below contains the list of system properties used by various AntX classes. Every AntX user property is prefixed with the marker string: “jware.antx.” For brevity this prefix is omitted in the following table but must be included in actual use.

Property	Description	Defaults
noiselevel	Default noise level used by feedback tasks. Used to determine at which level messages will be logged and/or reported.	“info”
noiselevel.false.prefers	Default noise level used by <prefer> tasks to emit messages for unmet preferences.	“warning”
defaults.haltiferror.flag	Default setting for the standard AntX “haltiferror” parameter.	“yes”
default.messages.bundle	Resource used as default (root) messages resource bundle.	None
tempobject.prefix	Prefix used to name temporary objects created by various temporary object factory tasks.	“qat”
defaults.asserts.flag	Toggle to switch AntX assertions (<assert>) off.	“yes”
allow.property.msgids	Determines if the project’s properties will be checked (first) for message ids. Set to either “yes” to enable property checks, or “no” to disable checks.	“no”
root.category	Log4j category used by feedback subsystem if none specified.	None
defaults.passwordfile	Location of simple (cleartext) password file in format of a standard Properties file.	None
strict.scriptcheck.enabled	Controls whether lenient AntX tasksets will try to verify their nested tasks at load time. Turn on if running AntX’s test scripts.	“no”

Examples

To set the default AntX noise level to VERBOSE:

```
<property name="jware.antx.noiselevel" value="verbose"/>
...
<show message="A verbose message"/>
```

Internal AntX Properties

In their implementation, some AntX flow-control tasks use properties to communicate between objects that reside in different projects. All of these properties are prefixed with “jware.antx.internal_.” Please do not explicitly define or use any of these properties as user properties; doing so will cause the AntX tasks to behave in unexpected ways.

AntX Task and Macro Definitions

The table below contains the default task definitions included with the JWare/AntXtras Foundation distribution. To use these definitions as-is load the “antlib.xml” file into your Ant runtime. This file is located in the com/idaremedia/antx/install directory of the distribution’s main jar file “AntX_tasks.jar.” So, if the main jar file is already in your Ant runtime’s classpath, you define the following (namespace URI omitted for brevity):

```
<taskdef resource="com/idaremedia/antx/install/antlib.xml"/>
```

Category	Task Name	Class Name
Execution-Rules	assert	com.idaremedia.antx.condition.solo.AssertTask
	prefer	com.idaremedia.antx.condition.solo.PreferTask
	tally	com.idaremedia.antx.condition.solo.TallyTask
	matches	com.idaremedia.antx.condition.solo.MatchesTask
	fixturecheck	com.idaremedia.antx.condition.solo.VerifyFixture
	rulemacro	com.idaremedia.antx.condition.solo.RuleMacroDef
	batchchecks	com.idaremedia.antx.condition.solo.BatchChecksTaskSet
	never ¹	com.idaremedia.antx.condition.solo.NeverTask
Flow-Control	do	com.idaremedia.antx.flowcontrol.ConditionalTaskSet
	dowhile	com.idaremedia.antx.flowcontrol.RepeatedTaskSet
	stop	com.idaremedia.antx.flowcontrol.StopTask
	protect	com.idaremedia.antx.flowcontrol.wrap.ProtectedTaskSet
	iferror ¹	com.idaremedia.antx.flowcontrol.wrap.IfErrorTask
	always ¹	com.idaremedia.antx.flowcontrol.wrap.AlwaysTask
	step	com.idaremedia.antx.flowcontrol.call.InlineStep
	callinline	com.idaremedia.antx.flowcontrol.call.LocalTargetTask
	call	com.idaremedia.antx.flowcontrol.call.CallOnceTask
	callforeach	com.idaremedia.antx.flowcontrol.call.CallForEachTask
	foreach	com.idaremedia.antx.flowcontrol.call.InlineForEachTask
	domatch	com.idaremedia.antx.flowcontrol.match.SwitchTask
	equals ²	com.idaremedia.antx.flowcontrol.match.MatchEquals
	true ²	<i>presetdef</i> for <equals value="true"/>
	like ²	com.idaremedia.antx.flowcontrol.match.MatchLike
	meets ²	com.idaremedia.antx.flowcontrol.match.MatchCondition
default ²	com.idaremedia.antx.flowcontrol.match.MatchAll	
otherwise ²	com.idaremedia.antx.flowcontrol.match.MatchAll	
Fixture	assign	com.idaremedia.antx.solo.ExportTask
	assignimport	com.idaremedia.antx.solo.ImportTask
	unassign	com.idaremedia.antx.solo.UnassignTask
	copyproperty	com.idaremedia.antx.solo.CopyPropertyTask
	copymsg	com.idaremedia.antx.init.CopyMsgTask

Category	Task Name	Class Name
	copyreference	com.idaremedia.antx.solo.CopyReferenceTask
	datadef	com.idaremedia.antx.fixture.helpers.DataDefTask
	locals ³	com.idaremedia.antx.flowcontrol.wrap.Locals
	isolate	com.idaremedia.antx.flowcontrol.wrap.IsolatedTaskSet
	macrolocals ³	<i>presetdef</i> for <locals name="_macrolocals_"/>
	macroisolate	<i>presetdef</i> for <isolate block="_macrolocals_"/>
	overlay	com.idaremedia.antx.solo.LocalFixtureTaskSet
	overlay-emit	com.idaremedia.antx.feedback.EmitConfigureTask
	overlay-msgs	com.idaremedia.antx.init.UISMConfigureTask
	capturelogs	com.idaremedia.antx.capture.CaptureLogsTask
	captureoutput	com.idaremedia.antx.capture.CaptureStreamsTask
	copylogged	com.idaremedia.antx.capture.CopyLoggedTask
	clearlogged	com.idaremedia.antx.capture.ClearLoggedTask
	managebundles	com.idaremedia.antx.init.InitUISMTask
	manageuris	com.idaremedia.antx.solo.ValueURIManagerTask
	manageprinters	com.idaremedia.antx.print.ManagePrintersTask
Feedback	show	com.idaremedia.antx.starters.MsgTask
	emit	com.idaremedia.antx.feedback.EmitTask
	checkpoint	com.idaremedia.antx.feedback.CheckpointTask
	emitlogs	com.idaremedia.antx.feedback.EmitLogsTask
Helpers	printenv [†]	com.idaremedia.antx.print.EchoItemsTask
	print [†]	com.idaremedia.antx.print.PrintTask
	vendorinfo	com.idaremedia.antx.solo.VendorInfoTask
	tempdir	com.idaremedia.antx.mktemp.TempLocator
	parentdir	com.idaremedia.antx.mktemp.ParentDirTask
	newtempfile	com.idaremedia.antx.mktemp.MkTempFile
	newfile	com.idaremedia.antx.mktemp.MkFileTask
	mkdirs	com.idaremedia.antx.mktemp.MkdirsTask
	truncatefiles	com.idaremedia.antx.mktemp.TruncateFileTask
	newtempdir	com.idaremedia.antx.mktemp.MkTempDirectory
	listdir	com.idaremedia.antx.mktemp.ListDirTask
	evaluatelogs	com.idaremedia.antx.capture.InterpretLoggedTask
	assertlogged [†]	com.idaremedia.antx.capture.AssertLoggedTask
	printerror [†]	com.idaremedia.antx.print.EchoErrorTask
	printbundle [†]	com.idaremedia.antx.init.EchoBundleTask

[†]These tasks are designed for use with test build scripts or diagnostic feedback.

¹These tasks function only when nested in a <protect> taskset.

²These tasks function only when nested in a <domatch> taskset.

³This task functions only with an <isolate> taskset.

AntX Type Definitions

The table below contains the default type definitions included with the JWare/AntXtras Foundation distribution. To use these definitions as-is load the “antlib.xml” file into your Ant runtime. This file is located in the `com/idaremedia/antx/install` directory of the distribution’s main jar file “AntX_tasks.jar.” So, if the main jar file is already in your Ant runtime’s classpath, you define (namespace URI omitted for brevity):

```
<taskdef resource="com/idaremedia/antx/install/antlib.xml"/>
```

Category	Type Name	Class Name
Build-Rules	rule	<code>com.idaremedia.antx.condition.solo.BuildRule</code>
	tallyset	<code>com.idaremedia.antx.condition.solo.TallySet</code>
	criteria	<code>com.idaremedia.antx.condition.solo.ExecuteCriteria</code>
Feedback	emitconfiguration	<code>com.idaremedia.antx.feedback.EmitConfigurationType</code>
	emitmappings	<code>com.idaremedia.antx.feedback.GroupingMapper</code>
	printer	<code>com.idaremedia.antx.print.PrinterMapping</code>
	printer-registry	<code>com.idaremedia.antx.print.PrinterRegistry</code>
	errorsnapshot	<code>com.idaremedia.antx.ErrorSnapshot</code>
Fixture	strings	<code>com.idaremedia.antx.solo.StringList</code>
	urls	<code>com.idaremedia.antx.solo.UrlList</code>
	properties	<code>com.idaremedia.antx.solo.PropertiesList</code>
	msgsbundle	<code>com.idaremedia.antx.init.UISMBundle</code>

AntX Condition Definitions

The table below contains the default conditions included with the JWare/AntXtras Foundation distribution. To use these definitions as-is load the condition “antlib.xml” file into your Ant runtime. This file is located in the `com/idaremedia/antx/condition` directory of the distribution’s main jar file “AntX_tasks.jar.” So, if the main jar file is already in your Ant runtime’s classpath, you define (namespace URI omitted for brevity):

```
<typedef resource="com/idaremedia/antx/condition/antlib.xml"/>
```

Condition Name	Class Name
isset	com.idaremedia.antx.condition.IsSet
isnotset	com.idaremedia.antx.condition.IsNotSet
issettrue	com.idaremedia.antx.condition.IsSetTrue
isallset	com.idaremedia.antx.condition.AllSet
isalltrue	com.idaremedia.antx.condition.AllSetTrue
isnonaset	com.idaremedia.antx.condition.NoneSet
isanyset	com.idaremedia.antx.condition.AnySet
isanytrue	com.idaremedia.antx.condition.AnySetTrue
isboolean	com.idaremedia.antx.condition.IsBoolean
isnumeric	com.idaremedia.antx.condition.IsNumeric
isnotwhitespace	com.idaremedia.antx.condition.IsNotWhitespace
ismatch	com.idaremedia.antx.condition.Matches
isequal	com.idaremedia.antx.condition.StringEquals
isantversion	com.idaremedia.antx.condition.IsAntVersion
isclass	com.idaremedia.antx.condition.IsClass
isresource	com.idaremedia.antx.condition.IsResource
isreference	com.idaremedia.antx.condition.IsReference
isdirectory	com.idaremedia.antx.condition.IsDirectory
isfilenotempty	com.idaremedia.antx.condition.FileNotEmpty