

Extending Ant

Steve Loughran
stevel@apache.org

Extending Ant, for Apachecon 2007 Europe

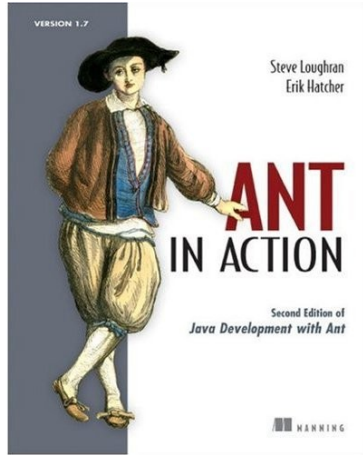
© 2007 Apache Software Foundation.

This presentation is released under the Apache 2.0 License.

It is currently hosted in Ant's svn repository at

http://svn.apache.org/repos/asf/ant/core/trunk/docs/slides/extending_ant.odp

About the speaker



Research on deployment at HP
Laboratories:
<http://smartfrog.org/>

Ant team member

Shipping in May/June:
Ant in Action!



<http://antbook.org/>

<http://smartfrog.org/>

All about me. By day: a research scientist at HPLabs, Bristol, UK. Out of hours Ant dev team (it's a long story), and co-author of Ant in Action, which is shipping in May 2007!

Today's Topic: Extending Ant

Inline scripting

Ant Tasks

Conditions

Ant Types and Resources

Embedded Ant

Non-XML syntaxes

Cutting your own Ant distribution

Out of scope for today
Ask on dev@ant.apache.org

Ant is extensible in many, many ways. You can use it in your own code, you can add new classes, and types for it. Want to use it under an IDE with custom input and output. There's hooks for that. Want to subclass Main and handle things differently. Use the -main option on the command line.

Being open source, Ant's interiors are there for inspection and extension. We try not to hide anything important, and encourage people to use it whenever there is a problem for which Ant is the appropriate solution.

This talk is going to look at some of the common ways to extend Ant, but it is only a fraction of the possibilities.

Before we begin

Ant 1.7 + source
Apache BSF
jython, jruby,
groovy, javascript
netREXX,...
or
Java1.6

```
> ant -diagnostics
----- Ant diagnostics report -----
Apache Ant version 1.7.0

bsf-2.3.0.jar (175348 bytes)
jruby-0.8.3.jar (1088261 bytes)
js-1.6R3.jar (708578 bytes)
jython-2.1.jar (719950 bytes)

java.vm.version : 1.6.0-b105
```

Problem

Generate a random number as part of the build

<scriptdef> declares scripted tasks

```
<scriptdef language="javascript" manager="javax"
  name="random">
  <attribute name="max"/>
  <attribute name="property"/>
```

Script Language

All attributes are optional

```
var max=attributes.get("max")
var property=attributes.get("property")
if(max==null || property==null) {
  self.fail("'property' or 'max' is not set")
} else {
  var result=java.util.Random().nextInt(max)
  self.log("Generated random number " + result)
  project.setNewProperty(property, result)
}
```

Java API

Ant API

```
</scriptdef>
```

This is an Ant task, inline.

Once declared you can use it anywhere you would a normal task, and it will generally behave just like a normal Ant task, even though those are normally written in Java. One thing about this declaration; the URI attribute, which puts it in a different namespace.

This example uses Jython, which retains Python's indentation rules. Even when embedded inside an Ant task, Jython source must be indented against the left margin. Watch out for XML editors that try to make your source look pretty, as they will break your code.

What is in scope in <scriptdef>?

<code>self</code>	the active subclass of <code>ScriptDefBase</code>
<code>self.text</code>	any nested text
<code>attributes</code>	map of all attributes
<code>elements</code>	map of all elements
<code>project</code>	the current project

These are the variables in scope inside a script. They give you access to the project, and everything behind it.

The `self` variable lets you get at the nested text, and it also has a set of methods that you can use. Look at the javadocs of `ScriptDefBase` to see what there is

<scriptdef> tasks *are* Ant Tasks

```
<target name="testRandom">  
    <random max="20" property="result"/>  
    <echo>Random number is ${result}</echo>  
</target>
```

```
> ant testRandom  
  
Buildfile: build.xml  
  
testRandom:  
    [random] Generated random number 8  
    [echo] Random number is 8  
  
BUILD SUCCESSFUL  
Total time: 1 second
```

Once you have written a task, it is now usable as any other Ant task. Users do not need to care how they are written, though stack traces when things go wrong sometimes make it very clear that something funny is happening underneath.

You do need to make sure that all users have the right script engines, but if you target Java1.6 and use JavaScript as the script language, you get everything right out of the box.

Yes, but do they work?

```
<random max="20" property="result"/>
```

```
<random max="20"/>
```

No working tasks without tests!

If you are writing tasks, be it in Java or a scripting language, you need a way of making sure they work. For a long time, the only way to test Ant tasks was to use the (unreleased, unsupported) ant-testutils.jar library to run a project under JUnit, with various assertions in the test methods.

This is now obsolete. The preferred method to test now is with inline assertions in the build file, with targets that test a different aspect of the task.

Of course, they need to be run. By Hand? Or by a test runner?

Imagine: test targets with assertions

```
<target name="testRandomTask">
  <random max="20" property="result"/>
  <echo>Random number is ${result}</echo>
  <au:assertPropertySet name="result"/>
  <au:assertLogContains
    text="Generated random number"/>
</target>

<target name="testRandomTaskNoProperty">
  <au:expectfailure expectedMessage="not set">
    <random max="20"/>
  </au:expectfailure>
</target>
```

If you are writing tasks, be it in Java or a scripting language, you need a way of making sure they work. For a long time, the only way to test Ant tasks was to use the (unreleased, unsupported) ant-testutils.jar library to run a project under JUnit, with various assertions in the test methods.

This is now obsolete. The preferred method to test now is with inline assertions in the build file, with targets that test a different aspect of the task.

Of course, they need to be run. By Hand? Or by a test runner?

AntUnit

```
<target name="antunit"
  xmlns:au="antlib:org.apache.ant.antunit">
  <au:antunit>
    <fileset file="${ant.file}"/>
    <au:plainlistener/>
  </au:antunit>
</target>
```

```
>ant antunit
Buildfile: build.xml

antunit:
[au:antunit] Build File: /home/ant/script/build.xml
[au:antunit] Tests run: 3, Failures: 0, Errors: 0,
              Time elapsed: 0.057 sec
[au:antunit] Target: testRandomTask took 0.038 sec
[au:antunit] Target: testRandomTaskNoProperty took 0.018 sec

BUILD SUCCESSFUL
```

<http://ant.apache.org/antlibs/antunit/>

AntUnit is the new, supported framework for running tests. Here we are, running the tests we've just written. All of them worked. This is cool. We have a build file that defines a task, and it contains the unit tests that actually validate it.

There is no excuse for any Ant task to not have unit tests. Even if functional testing is hard, you can at least check that the task handles invalid arguments by failing.

AntUnit is JUnit for Ant

`<antunit>` can test any number of nested files

All targets matching `test?*` are run

setup and teardown targets for every test

plain text or XML output

Assert state of build and file system

`<expectfailure>`

probes fault handling.

```
<assertTrue>
<assertFalse>
<assertEquals>
<assertPropertySet>
<assertPropertyEquals>
<assertPropertyContains>
<assertFileExists>
<assertFileDoesntExist>
<assertDestIsUptodate>
<assertDestIsOutofdate>
<assertFilesMatch>
<assertFilesDiffer>
<assertReferenceSet>
<assertReferenceIsType>
<assertLogContains>
```

AntUnit's architecture bears more than a passing resemblance to JUnit. It contains the assertions listed here, most of which are just `<presetdef>` extensions of `<assertTrue>`, a task that tasks a nested condition and fails if the condition is not met.

One special is `<assertLogContains>`, which checks that the log contains a specific string. This only works under an Antunit run, where the log is diverted to a buffer. The rest can be used in any build file.

The `<antunit>` task will run against any number of build files passed in as parameters; the output can be plaintext or XML, the latter a format you can feed through `<junitreport>`.

Ant uses AntUnit to test its own tasks, it is a supported product. Use it!

Nested Elements in <scriptdef>

```
<scriptdef language="ruby" name="nested"
  uri="http://antbook.org/script">
  <element name="classpath" type="path"/>
  paths=$elements.get("classpath")
  if paths==nil then
    $self.fail("no classpath")
  end
  for path in paths
    $self.log(path.toString())
  end
</scriptdef>
```

ant type;
use classname to give a full
Java class name

```
<target name="testNested"
  xmlns:s="http://antbook.org/script">
  <s:nested>
    <classpath path=".:${user.home}"/>
    <classpath path="${ant.file}" />
  </s:nested>
</target>
```

Nested elements have to be declared with their type.

your script should check for mandatory elements being supplied; call `self.fail()` if not, to print a message.

`<scriptdef>` best practises

Use `<scriptdef>` first!

Java 1.6+: target JavaScript

Test with `<antunit>`

Declare tasks into new namespaces

XML Namespaces

Tasks and types are declared in Ant's main namespace
Unless you set the `uri` attribute of any `-def` task
(`<scriptdef>`, `<typedef>`, `<presetdef>`, ...)
Private namespaces give isolation from the rest of Ant.

```
<target name="testRandom"
  xmlns:s="http://antbook.org/script">

  <s:random max="20" property="result"/>

  <echo>Random number is ${result}</echo>
</target>
```

Ant is more flexible about naming than most XML languages
—you don't need to declare children or attributes
in the same namespace as the parent

Namespaces came about in XML to let you mix different XML vocabularies in the same document.

Ant uses them to let you keep declared tasks separate from the built in ones, or those of third parties. This stops confusion about which version of `<jspc>` or `<deploy>` will get used.

Now, most people don't understand XML namespaces properly. If you meet someone who says they does, ask them whether the following XSD gives you an ambiguous particle error or not

```
<xsd:choice>
  <xsd:any namespace="##other" processContents="lax" />
  <xsd:any namespace="##local" processContents="lax" />
</xsd:choice>
```

The answer is "msxml thinks `##local` is in `##other`, but Xerces doesn't", which shows that even XML parser teams have different opinions.

To reduce support calls, Ant has a more relaxed view of matching than most XML languages:

If a task in a namespace supports a nested element or attribute, then Ant will match it with one declared in the same namespace, or one not declared in any namespace (`##local`). You only need to put the task/type definition in a namespace, not any of its children.

Other scriptable things

<code><script></code>	Inline script (obsolete)
<code><scriptfilter></code>	Inline filter of native/Java I/O
<code><scriptcondition></code>	Scripted condition (set <code>self.value</code> to true/false)
<code><scriptselector></code>	File selection logic in a script
<code><scriptmapper></code>	Filename mapping for <code><copy></code> , <code><uptodate></code> , <code><apply></code> ...

use in emergency, but they have limited reuse except through
build file sharing

Writing a “classic” Java task

```
public class ResourceSizeTask extends Task {  
    private String property;  
    private Union resources = new Union();  
  
    public void execute() {  
        if (property == null) {  
            throw new BuildException("No property");  
        }  
    }  
}
```

extend org.apache.tools.ant.Task
override public void execute()
throw BuildException **when things go wrong**

Even though

Public setter methods become attributes

```
public void setProperty(String property) {  
    this.property = property;  
}  
  
public void setFile(File file)  
public void setLimit(int limit)  
public void setClasspath(Path path)  
public void setFailonerror(boolean flag)
```

Ant expands properties then converts the string to the required type

Anything with a String constructor is supported

Files and paths are resolved to absolute paths

Overloaded methods? String comes last

Add elements through add() and create()

```
public void addSrc(FileSet fileset) {
    resources.add(fileset);
}

public Path createClasspath() {
    Path p=new Path(getProject());
    resources.add(p);
    return p;
}

public void add(ResourceCollection rc) {
    resources.add(rc);
}
```

Compile, <taskdef>, then use

```
<taskdef name="filesize"
  uri="http://antbook.org/"
  classname="org.antbook.apachecon.ResourceSizeTask"
  classpath="${build.classes.dir}"/>

<target name="testPath" xmlns:book="http://antbook.org/">
  <book:filesize property="size">
    <path path="${java.class.path}"/>
  </book:filesize>
  <au:assertPropertySet name="size"/>
</target>
```

Nested Text


```
package org.antbook.apachecon;
import org.apache.tools.ant.Task;

public class MessageTask extends Task {

    private String text = "";

    public void addText(String text) {
        this.text = getProject().replaceProperties(text);
    }

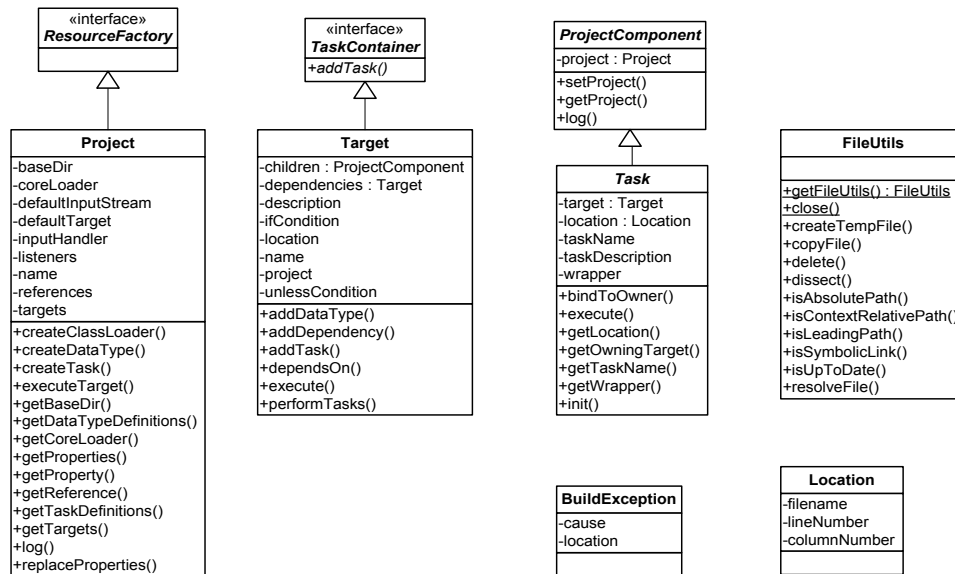
    public void execute() {
        log(text);
    }
}
```



explicitly expand
properties

Once you forget to expand properties (e.g. <sql>), you cannot patch it back in without running the risk breaking build files out in the field.

Ant's Main Classes



Resources

Resources are Ant datatypes that can be declared and cross-referenced across tasks

Resources are sources and sinks of data

Some resources are *Touchable*

Resources may be out of date or not found

Resources provide task authors with a way of modelling data, and of integrating with existing tasks that work with resources

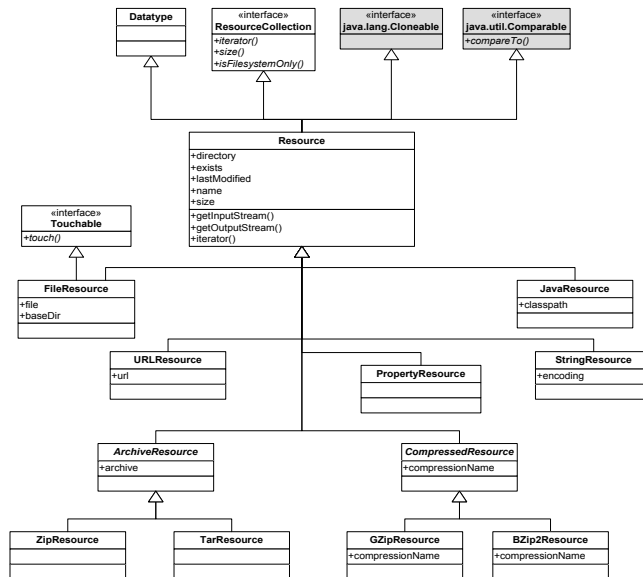
Resources are something that came with Ant 1.7; a fairly complex object model for how to provide a source or sink of data.

Resource-enabled tasks (of which `<copy>` and some of the archive tasks are the main examples) can take any source of data, regardless of its origin, and feed it in to the task.

Expect more Ant tasks to be resource enabled over time. For Ant1.7.1 I'm thinking we could res-enable the script tasks, to load source files from resources. If we res-enable `<import>`, we could even load Ant files from inside Zip files, remote web sites, etc, etc.

Come along to the Ant BOF for extra info.

Resources



This is the UML diagram for resources. UML always scares me, and UML diagrams like this even more. Finally, a bit of Ant that uses interfaces and a class hierarchy, rather than introspection.

Let's create a simple resource to generate a random stream of text, as big as you ask.

A simple resource

```
public class RandomResource extends Resource
    implements Cloneable {

    public boolean isExists() {
        return true;
    }

    public long getLastModified() {
        return UNKNOWN_DATETIME;
    }

    public InputStream getInputStream() throws IOException {
        if(isReference()) {
            RandomResource that;
            that = (RandomResource) getCheckedRef();
            return that.getInputStream();
        } else {
            //TODO
        }
    }
}
```

timestamp logic

reference resolution

return the data

This is the outline of the resource. We extend Ant's resource type, and implement a few methods.

First: stub the existence (always) and timestamp methods (assume always out of date), then the real work

getInputStream() is called to get an input stream. We have to do two things

1. check if we are a reference, if so, resolve it to the real instance, which must be of the same type as us.
2. Do the work; return an inputStream of some type.

For the random number generator we add an attribute to spec the limit of the resource (infinite resources can cause the build to appear to never complete), and provide an InputStream class that returns a random number stream using the Java crypto API.

Use <typedef> to define resources

```
<typedef name="rdata"
  uri="antlib:org.antbook.resources"
  classname="org.antbook.apachecon.RandomResource" />

<copy todir="build">
  <res:rdata name="random.bin" length="8192"/>
</copy>

<res:rdata length="10" id="sharedResource"/>

<loadresource property="random.property">
  <resource refid="sharedResource"/>
</loadresource>
<echo>random=${random.property}
```

Resources are just Java classes; we have to compile them and declare using <typedef>

<typedef> is the supertype of <taskdef>...it looks at the interfaces/classes a type declares and determines what it is. Everything but tasks get declared with <typedef>.

Once declared, you get to use it. Here I've stuck it in a new XMLNS, and use it inside a couple of copies. One inline, one via a reference.

The <loadresource> task loads a resource into a property...it actually gets a second instance of the resource class. We have to resolve the external resource at run time.

Demo

```
> ant demo
Buildfile: build.xml

demo:
    [echo]
random=yijyaeaxakikeybgbfvvhbyottpqtnvpauiemymnibbancaxcolbdptvbeyuhhqj
msroanrjjsmnocyqhyoibysugdwfsqsecsnugcijnjndhuuodbjoiiknsutukfhwrtos
afbujkhvgaeypfagrnnvcwqtkxjicxyuxnkqikujjtmwopkemeiwsitjpuieqxp ehdfvkw
drdtspbbftrjipnwvfwviooxwhfhsllkfxbeywwucfykglccoakyvrmncvwhmpycsojqbnf
kogrlkutuyugklmqkoyludubsaumcpvirgtjwghsukiphippruonyekcqdklkuwlruesse
vkbffgrljeiotgohcfjuftnplvitkfcrrbsmrevhlonsjojgogkrvtcrborxexxlnpkrjva
ovgqusombwyuxorlilavjkbwgjkkfuxvsknmvtgxdbcddmgqufifehyfugvirofybecfrsm
ejhkxrbgwmppxkucrelggfllqchuamadseihfmuefcavmwgasdncqfejt fombgsiqhnfaig
pyfjtjuglfttrjksnnvcwskdrjggilgogvubbwghgoefivsqntdimlgmntggghshoqgdeal
kjfpbcmoadcexraveoglcqdfdmyskngyfxtgqwlmobuvphxywkdpaeketobferskqcbtpc
xxvfvaonkiymweeosgnceynernu

BUILD SUCCESSFUL
```

This is our resource at work. Testing a random stream of data is left as an exercise for the reader, but be assured, tests are possible, especially if the type supports a seed attribute.

Resource best practises

Use resources to add new data
sources/destinations to existing tasks

Declare resources into new namespaces

Test with <antunit>

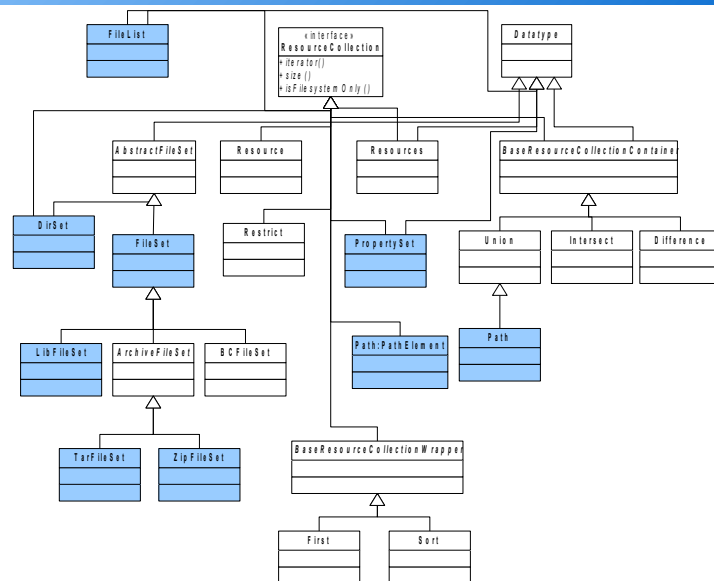
Package as an antlib

Resources can retrofit interesting new data sources to existing tasks; with a bit of work, <copy> can be made to do everything (actually, with a bit of work, copy can probably become Turing-equivalent, but we aren't going to go there).

Declare resources in new namespaces. This makes it easier to inject into existing classes; test this with antunit.

finally, package as antlibs -we will get to those next.

Resource Collections



This is the UML diagram for resource collections.

Resources are designed to aggregate, in particular

- * we've merged in all the existing collections like filesets, paths and filelists, into the hierarchy. They are all just resource collections returning resources of specific types.
- * there are general purpose union/intersection and difference collections that merge unions together.
- * every resource is itself a resource collection

Resource-enabling a task

```
public class ResourceCount extends Task {  
    private Union resources = new Union();  
  
    public void add(ResourceCollection rc) {  
        resources.add(rc);  
    }  
  
    public void execute() {  
        int count = 0;  
        Iterator element = resources.iterator();  
        while (element.hasNext()) {  
            Resource resource = (Resource) element.next();  
            log(resource.toString(), Project.MSG_VERBOSE);  
            count++;  
        }  
        log("count="+count);  
    }  
}
```

Store in a Union

Accept any collection

iterate!

To enable resources in a task, you just add an `add()` method that accepts a collection. That will give you any resource collection, including a `Resource` and subclasses.

Add these to a `Union` and you are storing stuff for reuse.

To get the resources back later, call `iterator()` and get an iterator through every single resource. That's it.

If you only want to support specific interfaces or classes, you need to cast and fail if you get the wrong type. But if you can, try and avoid that, as it makes your code less flexible.

Adding more resource support is on the Ant team's todo list; <scriptdef> got it the last week in April. It lets you declare a script on the classpath, a remote URL, anywhere else -fantastic flexibility with no need in the code to care. All it does is call `getInputStream`, letting the resources do the work.

Conditions

```
public class isEven extends ProjectComponent
    implements Condition {

    protected int value;

    public void setValue(int v) {
        value = v;
    }

    public boolean eval() {
        return (value & 1) == 0;
    }

}
```

A condition is a component that implements

`Condition` **and** `Condition.eval()`

Declare it with `<typedef>`

Conditions are really fun. Anything you can make conditional, you can use to control `<fail>`, `<conditon>`, `<waitfor>` and anything else 'conditional'.

write a `ProjectComponent` that implements the `Condition` interface

(or add to an existing task, resource, etc)

Add any attributes or elements you want.

Implement boolean `eval()`,

Return true if the condition holds

That's it! To test, use `<assertTrue>`; and `<assertFalse>` you can just embed it inside.

Turning a JAR into an antlib

Add `antlib.xml` to your package/JAR

Implicit loading via the XML namespace URL

```
xmlns:lib="antlib:org.antbook.resources"
```

```
<antlib>
  <taskdef name="filesize"
    classname="org.antbook.tasks.filesize.ResourceSizeTask"/>

  <typedef name="rdata"
    classname="org.antbook.resources.RandomResource" />
</antlib>
```

declare tasks, types, presets, macros, scriptdefs

I've mentioned antlib a few times -what is one?

An Antlib is a JAR file that contains an XML file listing what Ant tasks and types it provides. This XML file is read by Ant to make the tasks and types visible in a new namespace.

The thought of another XML configuration file is likely to make people, especially those who work with Spring, run screaming, but this is a special XML file: it is a list of Ant tasks. That's right: an `antlib.xml` file is a special subset of a `build.xml` file.

You can use `<scriptdef>`, `<typedef>`, `<taskdef>`, `<macrodef>`, `<presetdef>` or any other task that implements a special interface to indicate that it can be hosted in Antlib. This is best restricted to tasks that declare things, not tasks that have side effects.

Using an antlib

Implicit loading via the XML namespace URL

```
xmlns:res="antlib:org.antbook.resources"

<project name="filesize" default="default"
  xmlns:au="antlib:org.apache.ant.antunit"
  xmlns:res="antlib:org.antbook.resources">

  <res:rdata name="random.bin" size="8192"/>
</project>
```

or by explicitly loading

```
<typedef
  onerror="failall"
  uri="antlib:org.antbook.resources"
  classpath="{resources.jar}"/>
```

There's two ways to load an Antlib.

-implicit. this is the nice way to load third party libs that are on Ant's classpath you just declare the antlib: URI in the namespace declaration, and they are loaded automatically.

If the lib is there, you can use anything declared in the namespace. If the libfile is missing, you don't see an error until any of the XML elements are actually used.

-The other way to load a library is explicitly, through a `<typedef>` command that lists the URI to use, and the classpath of the library (including any other dependencies). This is good for where you set up the path using Ivy, or when you build the project itself. The `onerror="failall"` attribute says "if there is an error reading this file, fail now, not later". It catches problems in the typedef task, rather than when an attempt is made to use something in the namespace.

Summary

Ant is a collection of Java classes that are designed to be extended

Extend Ant through script, Java tasks, resources

Ant: conditions, selectors, filters, new antlib-types.

Embrace AntUnit!

See the Ant source, Ant in Action, and the Ant mailing list for more details.

We've just done a quick tour of Ant's facilities, stuff we haven't looked at all are conditions



Vragen?

Questions?

Embedded Ant

Don't call Ant `Main.main(String args[])` directly
Create a new Project and execute it

```
Project project = new Project();
project.init();
DefaultLogger logger = new DefaultLogger();
project.addBuildListener(logger);
logger.setOutputPrintStream(System.out);
logger.setErrorPrintStream(System.err);
logger.setMessageOutputLevel(Project.MSG_INFO);
System.setOut(
    new PrintStream(new DemuxOutputStream(project, false)));
System.setErr(
    new PrintStream(new DemuxOutputStream(project, true)));
project.fireBuildStarted();
```

Ask for help on developer@ant.apache.org

How to embed Ant?

Do not use Ant's main, which assumes it is in charge of the runtime, exits at the end of the run, overwrites your stdout and stderr channels, etc.

Create a new project, hook up any IO streams you want to a logger, then tell the project the build has begun. Here we are creating a default logger to the existing output channels, log at -info level and then put a multiplexed output stream in place of the old channels. `DemuxOutputStream` catches all output, buffers each thread's writes up to the end of the line, then routes it to the project, which logs it under the identity of the current thread. It's how Ant makes output of code run under Ant appear inside a task. Setting up the output is entirely optional -but useful to see what is going on.

Cutting your own Ant distribution

- Don't break existing Ant installations
- Don't be incompatible with normal Ant releases
- Don't hide the Ant classes in a different JAR.
- Build with all the optional libraries present
- Rename ant.bat and ant.sh
- Create custom equivalents of ANT_HOME, ANT_OPTS, ANT_ARGS env variables.
- Document what you've done

WebLogic and WebSphere: please help us here!

Next extra topic: redistributing Ant yourself. Lots of things do this, either embedded, or with a copy of Ant somewhere.

Here are some best practises. Remember it's the Ant team that get the support calls when builds fail because there is another copy of Ant on the classpath. We tell them to delete your product and try again.

If your tool (e.g. Forrest) fail because you've shipped an older copy of Ant and yet you still read in ANT_OPTS in forrest.bat, well, that's your support problem, but its trivial to avoid (ask for different env variables)