NewAntFeaturesInDetail Import

the task's manual page

<import> lets you import build file snippets much like entity includes did before Ant 1.6, but it also adds some new features on top of this.

<import> vs entity includes

- Files suitable for <import> must be legal Ant build files, i.e. they have to be well-formed XML documents and use <project> as their root element. This means that - unlike the snippets you use in entity includes - the included file can easily be edited and (partly) validated with XML aware tools.
- You can use Ant properties in the file name for the piece to include in <import>. With entity includes you've been limited to static file names and hard-coded (relative) paths.
- You don't have to worry about the way your XML parser resolves relative file names.
- Entity includes require a DOCTYPE declaration and thus have their roots in DTD systems. If you have an XML-Schema or Relax-NG gramar for Ant build files, the two are hard to combine.
- You must not use <import> inside a <target> (or <sequential>), while there is no such limitation for entity includes.

<import> is a boon to multi-project builds because it lets you keep all dependencies in a single DAG (c.f: "Recursive Make Considered Harmful"), in a multi-project build.

For example, suppose build.xml does an <import> of two projects named "x", and "y", and that both "x" and "y" have targets named "moo". Any target in any of the projects can make an explicit reference to target "x.moo" or "y.moo" in its depends clause. This feature is wonderful, but is woefully under-documented, and has least two nasty flaws:

- Names of targets within a project aren't bound tighter than names external to a build file. Therefore, if project "x" is imported first, then project "y", and a target in "y" depends on "moo", then ant will interpret that as "x.moo", NOT "y.moo". That's particularly scary when you consider that the world of targets that can potentially be imported is huge and a dependency hijacking could be both silent and in a remote file elsewhere in a big build.
- Targets in the top-level don't get a magical *projectname.targetname* alias. You can only refer to them by their unadorned *targetname*. This is evil because what happens if a target in the top level gets moved elsewhere? At that point you're right back to the first problem mentioned of being subject to silent dependency hijacking. As of Ant 1.7, you cannot refer to the targets in the main build file explicitly, so there's no workaround for this one other than knowing about it & being careful.

How does an imported file load a resource relative to itself?

By using the magic `ant.file.`*projectname* property.

Assume build.xml loads config/common.xml, the latter making use of a collocated properties file config/common.properties. The way <import> is currently implemented (I argumented against it, but that's beside the point), common.xml cannot simply do a `<property file="common.properties"/>` as if it was stand-alone, because the project's 'basedir' will correspond to the top-most importing build file. Ant does on the other hand store the absolute pathname of imported build files in a magic property of the form ant.file.*projectname*, where *projectname* is an imported project's name (as defined by the 'name' attribute of the top-level <project> element, and ***NOT*** the imported file name!). You can thus <dirname> that absolute filename, and use the resulting directory to locate the resource relatively to the imported build file. Here's a full example on Windows:

```
C:\oss\org_apache\antx\import16> type build.xml
<project name="main" default="build">
  <import file="config/common.xml" />

  <target name="build" depends="common.build">
    <echo>main-build called.</echo>
    <echo>version = ${version}</echo>
  </target>
</project>

C:\oss\org_apache\antx\import16> type config\common.xml
<project name="common" default="build">
```

*

```
  <dirname property="mybasedir" file="${ant.file.common}" />
  <property file="${mybasedir}/common.properties" />
```

*

```
  <target name="build" depends="compile, test">
    <echo>common-build called.</echo>
    <echo>version = ${version}</echo>
  </target>

  <target name="compile">
    <echo>common-compile called.</echo>
    <echo>version = ${version}</echo>
  </target>

  <target name="test">
    <echo>common-test called.</echo>
    <echo>version = ${version}</echo>
  </target>
</project>

C:\oss\org_apache\antx\import16> type config\common.properties
version = 1.0

C:\oss\org_apache\antx\import16> ant
Buildfile: build.xml

compile:
     [echo] common-compile called.
     [echo] version = 1.0

test:
     [echo] common-test called.
     [echo] version = 1.0

common.build:
     [echo] common-build called.
     [echo] version = 1.0

build:
     [echo] main-build called.
     [echo] version = 1.0

BUILD SUCCESSFUL
Total time: 0 seconds
```

Since an imported build file *projectname* is used both in itself (to be able to locate resources relative to itself as demonstrated above) but also potentially in all the files that will import it (to refer to its overriden targets), changing the *projectname* of an imported build file will almost always break its _client_s, i.e. those build files that import it. This is a mistake IMHO, and breaks encapsulation (again, I argued against this, but no avail.) One should therefore select those project names carefully, lest one wants to expose itself to quite of bit of refactoring.

Overriding targets in the imported build file

If a target is present in both your main build file and the one that you import, the one from your main file takes precedence.

This means that you can write a generic build file and if you just have to tweak it a little, you can do so by overriding a target. You even have access to the imported target via *projectname.targetname*.

For example, let's take the simple build file from Ant's manual. Suppose it is called "example.xml".

```
<project name="MyProject" default="dist" basedir=".">
    <description>
        simple example build file
    </description>
  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist"  location="dist"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
        description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
        description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
  </target>

  <target name="clean"
        description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

Now, let's assume you need to "rmic" some classes - a convenient way to do that would be right after the javac task in the compile target.

You could do

```
<project basedir="." default="dist">

  <import file="example.xml"/>

  <target name="compile" depends="init"
        description="compile the source and rmic some classes" >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
    <rmic base="${build}" verify="true"/>
  </target>
</project>
```

and your rmic task has been injected. You can even do

```
<project basedir="." default="dist">

  <import file="example.xml"/>

  <target name="compile" depends="MyProject.compile"
        description="rmic some classes" >
    <rmic base="${build}" verify="true"/>
  </target>
</project>
```

and use your own compile target just to post-process the original compile target.