Software Development

COMP220/COMP285

Seb Coope

Ant: Structured Build

These slides are mainly based on "Java Development with Ant" - E. Hatcher & S.Loughran. Manning Publications, 2003

Imposing Structure

- In the build process discussed in the previous lecture and considered in a **Lab** sessions:
- *source files*, *output files*, and *build file* were in the ***same directory***.
- These were files **`Main.java, Main.class`** and **`build.xml`** in

`C:\Antbook\ch02\firstbuild`

(In **Labs** you will use **`H:`** instead of **`C:`**)

Imposing Structure

- In a bigger project, things could get out of hand.
- We want to ***automate the cleanup*** in **Ant**.
- If done wrong, this could ***accidentally delete source files***.
- Thus, let us ***separate*** ***source*** and ***generated*** files into different directories.

Imposing Structure

- We also want to place **Java** *source file* into a **Java** ***package***.
- We want to *create* a **JAR** *file* containing the compiled code.
- We should be *able to* ***clean up*** *the directories* with compiled files and this **JAR** file before starting the next build.
- Hence, use (de facto) ***standard directory names*** as in the next table:

STANDARD DIRECTORY NAMES

Directory name	Function
`src`	source files
`build\classes` (or **`bin`** – a default in **Eclipse**)	intermediate output (generated; cleanable)
`dist`	distributable files (generated; cleanable)

Let these directories be **sub-directories** of a new **base directory:**

```
C:\Antbook\ch02\secondbuild
```

STANDARD DIRECTORY NAMES

Thus, we will have the following

directory structure:

```
C:\Antbook\ch02\secondbuild\src
                           \build\classes
                           \dist
```

which will be *further extended.*

Please relate all the following considerations and your Lab exercises exactly with the directory structure suggested in these slides

(except your **personal** directory structure under `src` related with your **personal** packages – to be discussed below).

Packages

- Keep files together with close associations
- Related to scope
- Stops naming collisions
- Relation with domain names
  - Reversed
    - comp220.csc.liv.ac.uk
  - Package could be
    - uk.ac.liv.csc.comp220.utils

Laying out the source directories and source files

- When working on a project it makes sense to ***add*** some (*your personal*) ***package declaration*** to our java classes such as **`Main.java`** considered in the previous lecture (and a **Lab**):

```
package org.example.antbook.lesson1; //package declaration
public class Main {
    public static void main(String args[]) {
        for(int i=0;i<args.length;i++) {
            System.out.println(args[i]);
        }
    }
}
```

- In this case we should also ***put*** so modified **`Main.java`** ***into*** the sub-…-sub-directory *of* **`src`**: `src`**`\org\example\antbook\lesson1`**

  corresponding to (or matching) the above ***package name***.

Laying out the source directories and source files

- Recall that **Java** *packages* and "contained" in these packages **Java** *classes* are always organized in such a way:
  - ***package names*** of **Java *classes*** exactly match *directory system* where to find them (also in **JAR** files).
- That is, we use traditional ***agreement*** that any **Java** class declaring its package

```
package aaa.bbb.ccc;
```

MUST be contained in a corresponding sub-sub-…-directory

```
...\aaa\bbb\ccc
```

Laying out the source directories and source files

- Recall also that ***packages*** *give an appropriate* ***level of access*** to their classes and methods.

- It is quite reasonable that ***related classes*** are ***in the same package*** (i.e. have the same package declaration).

- Packages also allow to use "qualified" ***class*** and ***method names*** specific to these packages like

```
org.example.antbook.lesson1.Main
```

- ***without any conflict*** even with possibly identical names in some other projects within other packages.

Compilation from **command line**

TRY the following:

- Compile our renewed java class with the ***full name***

```
C:\Antbook\ch02\secondbuild\src\
   org\example\antbook\lesson1\Main.java
```

by the command

```
C:\Antbook\ch02\secondbuild\src>javac -d ..\build\classes
org\example\antbook\lesson1\Main.java
```

- **`javac`** option **`-d ..\build\classes`** specifies ***directory*** (full or relative name) *where to place **generated** class files .*
- ***`org\example\antbook\lesson1`***\**`Main.java`**

  is ***source** file to be compiled:*
- this should be either ***full name*** or a ***name relative*** to the current directory **`C:\Antbook\ch02\secondbuild\src`**.

Compilation from **command line**

- Then the resulting compiled **`Main.class`** will have the corresponding full path:

```
C:\Antbook\ch02\secondbuild\build\classes\
    org\example\antbook\lesson1\Main.class
```

provided that the subdirectories

```
build\classes
```

exist.

Check that existence of these directories is really necessary!

Compilation from **command line**

- **Summarise** that, the general form of ***compile command*** is:

```
javac -d [where to compile] [what to compile]
```

- ***Package declaration*** `aaa.bbb.ccc` of the compiled class shows the ***path*** `aaa\bbb\ccc` for the compiled class ***relatively*** *to* `[where to compile]`

- **WHAT TO CHANGE** above in the command

```
javac -d ..\build\classes
  org\example\antbook\lesson1\Main.java
```

if you compile from `secondbuild` **instead of** `src`**??**

Compilation from **command line**

- **TRY** the same command again, but with ***commented*** package declaration in **`Main.java`**:

```
// package org.example.antbook.lesson1;
```

In which directory the compiled `Main.class` will appear? (Use **time stamps** to identify.)

- **TRY** the same command again with a different package declaration in the class **`Main.java`**

```
package org.example2.antbook.lesson1;
```

- **Where now the compiled class `Main.class` will appear?** (Use **time stamps** to identify.)
- **RECOVER the original package name!!!**

Summary on compiling with package declarations in source files

- The general form of ***compile command*** is:

```
javac -d [directory where to compile]
   [what to compile]
```

- Package declaration in a `.java` file says ***where to put*** *the resulting compiled class* ***relatively*** to a directory (`-d [directory]`).

- When the **Java** compiler compiles the files, it *always places the output files* ***in a directory path that matches the package declaration***.

<javac> task, **laying out directories** and **dependency checking**

Recall that ***in*** **Ant *build file:***

- <javac> task means *compiling*.

- The next time <javac> task runs, it does *dependency checking* :
  - *looks* at the *directories tree of generated class files* and
  - *compares* them to the source files
    - - whether they are *up-to date* and should be recompiled.

- When doing *dependency checking*, it relies on ***matching*** the *source tree* to the *destination tree*.

<javac> task, **laying out directories** and **dependency checking**

- For **Java** source *dependency checking* to work, you **MUST** lay out source **Java** files in a *directory tree* that ***matches*** the *package declarations in the source files*.

- When the source file has **no package** declaration (the **empty** package) you **must place this file in the base of the source tree** (typically, the directory `src`).

- If **Ant** keeps on ***unnecessary recompiling*** your **Java** files every time you do a build, it is probably ***because you have not placed them correctly in the package hierarchy***.

- ***Unnecessary recompiling***, even if done automatically, ***is time consuming!***

- That is, **if** the package name `aaa.bbb.ccc` of the source `File.java` **does not match** with its directory path, e.g., `...\aaa\bbbbb\ccc\File.java` (by your mistake) then the directory path leading to `...\aaa\bbb\ccc\File.class` will be ***different*** from that of the source and `<javac>` will ***not be able to compare time stamps*** of these files.

Comments on laying out directories

- It *may seem inconvenient* to rearrange your files into a system of subdirectories.

- But *on a large project, such a layout is critical* to separating and organizing classes.

- Modern **I**ntegrated **D**evelopment **E**nvironments (**IDE**) (such as **Eclipse**) also *prefer and supports using this layout* structure, as does the underlying **Java** *compiler* `javac` (as we have already seen).

- Recall again that *correctly laid out directories* also serve *for dependency checking* of <javac> which consists in comparing the timestamps of the source and destination files. In big projects this *saves time!*

Adding dependency checks

- Besides dependency check by `<javac>` task, there is an additional `<depend>` task to do more *advanced dependency checking*.
- The `<javac>` dependency logic (to insure that out of date classes are **not** recompiled during incremental builds) implements a *rudimentary* check that only passes `.java` files to the compiler if the corresponding `.class` file is older or nonexistent.
- It *does not rebuild classes when the files that they depend upon change*, such as a *parent class* or an *imported class*.
- The latter problem is resolved by the `<depend>` task

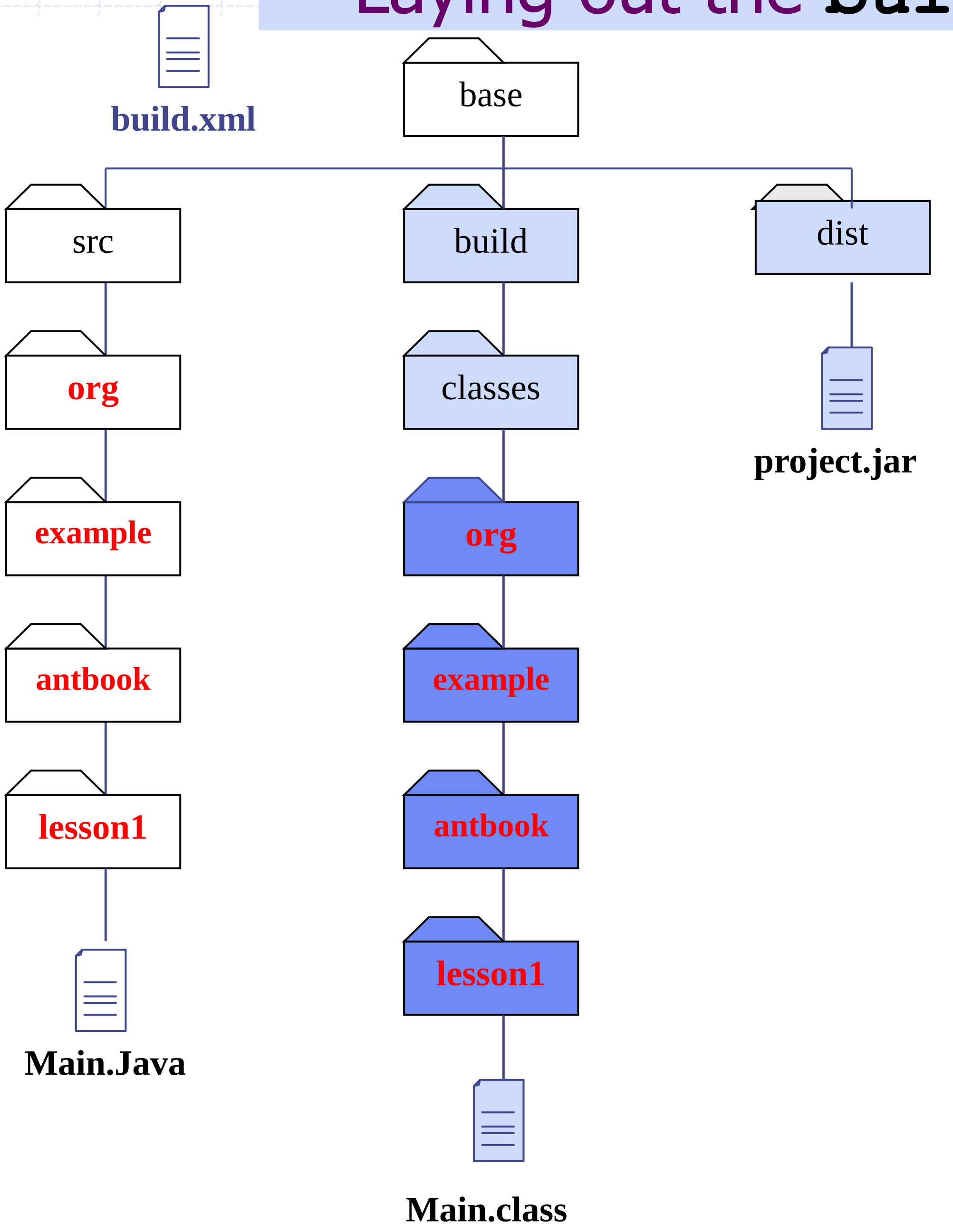
Laying out the `build` and `dist` directories

Imagine that you have a *huge project* where we should *create*

- many ***intermediate*** files,
- as well as ***delivered*** or ***deployed files***

Taking into account our previous discussion, the directories for a project may look like in the following slide.

Laying out the `build` and `dist` directories



A ***source*** tree (branch) is *separated* from the ***build*** and ***distribution*** output.

All the ***shaded*** directories and files are ***created by* Ant** during the build ***automatically!***

Note, that in our case **`base`** directory is **`secondbuild`**

Red colour and **darker shade** correspond to the ***package name***

The directory layout

- Assume putting all ***intermediate*** files into the **`build`** directory tree.
- Recall that **Java compiler** lays out compiled **`*.class`** files into a (darker shaded above) directory structure that ***matches the package declarations*** in the source files.
- The ***compiler will create the appropriate directories*** under **`build/classes`** subdirectory automatically, so we *do not need to create them manually* and therefore to bother too much about this.
- After deciding on ***package name*** and creating corresponding directories under **`src`**, **we need to prescribe *in the build file*** (only once) that the following directories be generated:
  - the top level **`build`** directory, and the **`classes`** subdirectory,
  - as well as the **`dist`** directly for deployed (**JAR**, **Zip**, **tar**, **WAR**, etc.) *archive files*.

The directory layout

- Note that the **`dist`** directories are usually much simpler than the intermediate file directories under **`build`**.
- All these (shaded above) files and directories will be created ***automatically*** (and even can be automatically deleted before any new build) by **Ant.**
- So, we are not worrying about them.
- We only should write in the build file:

```
<mkdir dir="build/classes" />
<mkdir dir="dist" />
```

Create the build file structured.xml

C:\Antbook\ch02\secondbuild\structured.xml

```
<?xml version="1.0" ?>
<project name="structured" default="archive" >

  <target name="init">
    <mkdir dir="build/classes" />
    <mkdir dir="dist" />
  </target>

  <target name="compile" depends="init" >
    <javac srcdir="src"
           destdir="build/classes"
           includeAntRuntime="no"/>
  </target>
```

Creates the output directories

Compiles into the output directories

(continues)

Creating the build file `structured.xml`

(continued)

```
  <target name="archive" depends="compile" >
    <jar destfile="dist/project.jar"
         basedir="build/classes" />
  </target>

  <target name="clean" depends="init">
    <delete dir="build" />
    <delete dir="dist"  />
  </target>

</project>
```

Creates the **Java archive** from all compiled classes

Cleans the output directories

(when invoked)

Creating the build file `structured.xml` (continued)

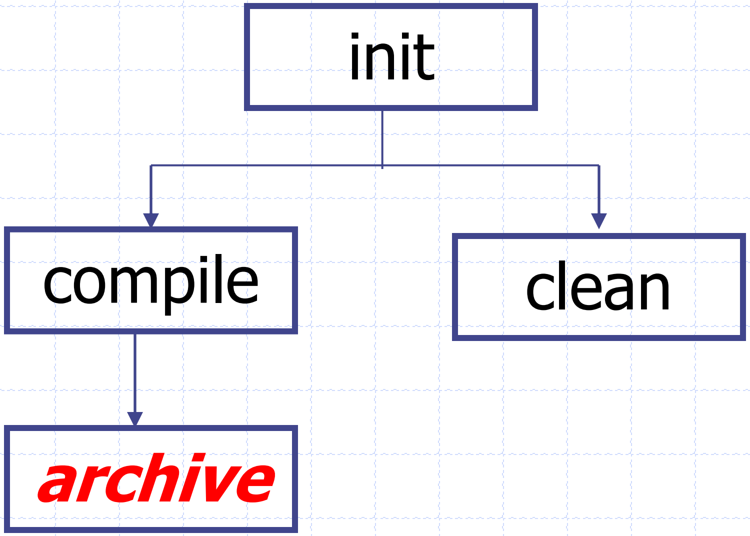
- The meaning of all these targets is evident.
- Let us note only that:

  - the **Ant** task **`<javac>`** ***compiles all Java source from*** **`src`** ***directory and all its subdirectories***.

  - the **Ant** task **`<jar>`** creates **JAR** file containing ***all files in and below*** the **`build/classes`** directory, which in this case means:

    all **`*.class`** files created by compile target.

Target dependencies in **structured.xml**



Here **clean** ***depends*** upon **init**;

archive ***depends*** on **compile** directly and on **init** indirectly (via **compile**).

The target **archive** is declared as ***default*** in the build file **structured.xml**

Running the new build file

- Recall that the command

```
ant
```

runs, *by default*, the build file named as **`build.xml`**

- In the case of a different build file, like our **`structured.xml`**, we should use the following form of the command:

```
ant -f structured.xml
```

Running the new build file

- Since the target **`archive`** is *default* one in **`structured.xml`**, the command

```
ant -f structured.xml
```

will run only the *chain of targets*

```
init -> compile -> archive
```

`structured.xml` **calls** **`archive`** which **calls** **`compile`** which **calls** **`init`**

- In this run **`clean`** target will **not** be executed since it **will not be called**.

Now,

- **DELETE** **`build`** and **`dist`** directories ***to start on a clean space***, and
- **RUN** the above command.

Running the build

```
C:\Antbook\ch02\secondbuild>ant -f structured.xml
Buildfile: C:\Antbook\ch02\secondbuild\structured.xml

init:
    [mkdir] Created dir:
C:\Antbook\ch02\secondbuild\build\classes
    [mkdir] Created dir: C:\Antbook\ch02\secondbuild\dist

compile:
    [javac] Compiling 1 source file to
C:\Antbook\ch02\secondbuild\build\classes

archive:
      [jar] Building jar:
C:\Antbook\ch02\secondbuild\dist\project.jar

BUILD SUCCESSFUL
Total time: 3 seconds
```

Rerunning the build again

```
C:\Antbook\ch02\secondbuild>ant -f structured.xml
Buildfile:
   C:\Antbook\ch02\secondbuild\structured.xml

init:

compile:

archive:

BUILD SUCCESSFUL
Total time: 1 second
```

Why no **real** action, but **`BUILD SUCCESSFUL`**?

Rerunning the build again

- **None** of the tasks **`<mkdir>`, `<javac>`, `<jar>`** say that they are doing any ***real*** work.
- All of these tasks check their dependencies:
  - **`<mkdir>`** does not create directories that already exist;
  - **`<javac>`** compares source and class file timestamps:
    - if **up to date** – do actually nothing;

Rerunning the build again

- `<jar>` compares the time of all files to be added to the archive with the time of the `.jar` file itself.

- If the resulting files are up to date, these tasks, although invoked,

  do actually nothing.

- **TRY** the same in `-verbose` or `–v` mode to see the similar comments from **Ant**.

Clean it!

- Finally, **TRY** the command

```
ant -f structured.xml clean
```

which *deletes* **`build`** and **`dist`** directories.

- Hence, you can start build process again on a clean place (after changing something in your source files under **`src`**).

What if…?

- **What if** our subdirectories under `src` were laid out wrongly, not according to the package declaration in `Main.java`?
- **TRY** to *change* the package declaration *in our source file*

```
src\org\example\antbook\lesson1\Main.java
```

to **package** org.**example2**.antbook.lesson1**;**

- and ***RUN repeatedly*** the command

```
ant -f structured.xml
```

- Check that **Ant** really keeps on *unnecessary recompiling* `Main.java` every time you do a build *because you have not placed them correctly in the package hierarchy*.
- **RECOVER the original package name!!!**

Multiple targets **on the command line**

The command with multiple ***targets*** as arguments

```
ant -f structured.xml compile archive
```

is ***equivalent*** to running **Ant *twice:***

```
ant -f structured.xml compile

ant -f structured.xml archive
```

The resulting sequence of targets will be:

`init -> compile,` and then
`init -> compile -> archive`

Thus, **for multiple targets called, repetitions** of targets are possible in the resulting sequence**!!**

TRY it!

Multiple dependencies **in build file**

- When a target lists ***multiple dependencies***

```
<target name= "all" depends= "archive,clean" />
```

then **Ant** executes them ***in the order listed*** **:**

- It first *calls* `archive` and then `clean`.
- Note that `archive` will also call its dependencies, that is, `init` -> `compile` -> `archive` will be executed.
- Then `clean` will be called, but ***now*** `init` will ***not*** be repeated:

```
init -> compile -> archive -> clean -> all
```

When **Ant** build file runs in itself, i.e. ***one*** or ***no*** targets is called ***from the command line***, then ***targets are not repeated***

- **TRY** to check this by ***adding*** the above target `all` to `structured.xml;` use the command

```
ant -f structured.xml all
```

Running Java Program from inside Ant

- We now have a structured build process that compiles **Java** files and creates the **JAR** file from the **Java** compiled classes.

- The next question is:

  How to **run** a **Java** program with Ant?

First, Executing **from Command Line**

To ***execute*** our program `Main.class` we should first ***compile*** `Main.java`. (See `Main.java` on **Slide 7**)

Then we could just call our program `Main.class` as usually ***from the command line (or on console)*** by stating

- the ***classpath*** (showing where to find `Main.class`),
- the ***qualified class name*** (using the ***package name***) and
- the ***arguments*** `"a"`, `"b"`, and `"."`:

```
C:\Antbook\ch02\secondbuild>java -cp build\classes
org.example.antbook.lesson1.Main a b .
a
b
.
```

Three inputs

Three identical outputs

This program `Main.class` just ***types the argument values.***

Why execute from inside **Ant**?

Running this program ***from the build file*** provides some ***benefits*** *in comparison with command line* :

- ***no need to split*** program ***compilation*** *from* ***execution***
- a target to run ***depends upon the compilation*** target, so we know we always
  - ***run the latest version*** of the code
- ***easy to pass complex arguments*** to the program
- ***easier to set up the classpath***
- the program can run inside **Ant**'s own **JVM:**
  - it ***loads faster***

Adding an **execute target**

- **Extend** the previous build file **`structured.xml`** to **new file** **`execute.xml`** by adding target

```
<target name="execute" depends="compile">
  <java
    classname="org.example.antbook.lesson1.Main"
    classpath="build/classes" >
    <arg value="a"/>
    <arg value="b"/>
    <arg file="."/>
  </java>
</target>
```

See below on the difference between **`value`** and **`file`** attributes of **`<arg>`**

- **`<java>`** task ***executes*** the program **`Main.class`** with the arguments specified.

`<arg>` tags

- `<arg value="somevalue">` adds a command-line ***argument*** `somevalue`.

- The action of this task is evident (with the **`value`** attribute).

- The last argument is ***of another kind:***

  `<arg file="."/>`

  It tells **Ant** to *resolve* the **`file`** attribute `"."` (meaning "this directory") to an *absolute* ***build file location*** (more precisely, to an *absolute* ***base directory location***) and consider this location *as an argument value* before calling the program.

- The latter *differs from* ***the ordinary***

  `<arg value="."/>`

  used implicitly in the above command line running (Slide 38).

Running `<java>` task in the `<execute>` target

```
C:\Antbook\ch02\secondbuild>ant -f execute.xml execute
Buildfile: C:\Antbook\ch02\secondbuild\execute.xml

init:

compile:

execute:
     [java] a
     [java] b
     [java] C:\Antbook\ch02\secondbuild

BUILD SUCCESSFUL
Total time: 1 second
```

TRY it! **Try it also** with

`<arg file="abcd/pqr.txt"/>`

For the Lab:
Getting information about the project

`-projecthelp` lists the **main** and **other** targets in a project build file.

```
C:\Antbook\ch02\secondbuild>ant -projecthelp -f execute.xml
Buildfile: C:\Antbook\ch02\secondbuild\execute.xml

Main targets:

Other targets:

 archive
 clean
 compile
 execute
 init
Default target: archive
```

Here **Ant** lists ***no*** **main** targets because

main targets are those which contain the optional **`description`** **attribute**, as these are the

targets intended for public consumption.

For the Lab:
Getting information about the project

The above example is *not very informative*, which is our fault for ***not documenting*** the file.

Add a `description` attribute to each target of **`execute.xml`**, such as

```
description= "Compiles the source code"
```

for the **`compile`** target.

Add also a **`<description>`** element right under the **`<project>`** opening tag.

Look at the resulting **`build.xml`** file (downloadable from corresponding **Lab** Web page).

For the Lab:
Getting information about the project

Note, that **build.xml** differs from **execute.xml** only

1. by such *descriptions*
2. by declaring **execute** as a *default target*, and
3. by changing the *project name* with **secondbuild**

PUT this **build.xml** in **C:\Antbook\ch02\secondbuild** directory.

TRY the command

```
ant -projecthelp
```

(calling by default **build.xml**).

Compare the result with the previous command (from the previous slide)

```
ant -projecthelp -f execute.xml:
```

For the Lab:
Getting information about the project

```
C:\Antbook\ch02\secondbuild>ant -projecthelp
Buildfile: C:\Antbook\ch02\secondbuild\build.xml
Compiles and runs a simple program
Main targets:

 archive   Creates the JAR file
 clean     Removes the temporary directories used
 compile   Compiles the source code
 execute   runs the program
Default target: execute
```

- "Described" targets are listed as "Main targets‟ now.
- Other "sub targets" are hidden from view.
- Use **`–verbose`** (or **`–v`**) to see these **`Other targets`** as well