Engineering Software Development in Java

Lecture Notes for ENCE 688R, Civil Information Systems

Spring Semester, 2013

Mark Austin,
Department of Civil and Enviromental Engineering,
University of Maryland,
College Park,
Maryland 20742, U.S.A.

Contents

9 Working with Packages, JAR and Ant **1**
9.1 Working with Packages 1
Organizing Files into Packages 1
Setting the CLASSPATH 2
Program Compilation 2
Application 1. Two kinds of Apple, Fruit and Computers 2
9.2 Working with JAR Files 7
Creating JAR files 7
Inspecting the Contents of a JAR File 7
Using the code within a JAR file 8
Extracting the Contents from a JAR file 8
JAR Files as Applications 9
9.3 Program Compilation with Ant 10
9.4 Application 2: Basic Program Compilation with Ant 10
Compiling the Program with Ant 12
Creating JAR Files with Ant 13
Running Programs with Ant 14
9.5 Application 3: Not so Basic Program Compilation with Ant 14
9.6 Application 4: Dealing with many Jar Files 18
9.7 Application 5: Using Ant to Move Files Around 19
9.8 Application 6: Using Ant to Run Programs having Arguments 19

References **21**

I Appendices **22**

Index **23**

Chapter 9

Working with Packages, JAR and Ant

Java provides a number of mechanisms for organizing classes into packages and then making the available for use via JAR files.

9.1 Working with Packages

Definition. A package is simply a group of classes. Packages are a convenient mechanism for ...

> **... organizing code and, in the case or team development of code, separating the work of one developer from another.**

Organizing Files into Packages

All of the source code files in a package must be located in a subdirectory that matches the full name of the package. For example, source code files in the sub-directory

```
computer/
```

are put in the package computer. Simple put the statement

```
package computer;
```

before the first executable statement in each source code file. Similarly, files in computer/old would have the package name

```
package computer.old;
```

Thus a hierarchy of directories will store a hierarchy of packages.

Note. To keep package names unique, Oracle recommends that package names be organized according to the organization in which the code is being developed. Suppose, for example, that we are developing a graphics package under the auspecies of

```
http://www.cee.umd.edu
```

The corresponding package name would be:

```
package edu.umd.cee.graphics;
```

Setting the CLASSPATH

In most applications the hierarchy of source code files will not branch from the root directory. Therefore, we need to tell Java where to find the hierarchy of packages.

In UNIX, the PATH environmental variable contains a list of directories that will be searched to find an appropriate file. The CLASSPATH environemt variable is to Java what PATH is to UNIX.

In UNIX/Linux/Mac OS X, you can explicitly type:

```
prompt >> setenv CLASSPATH /User/austin/java.d/package-and-jar.d/
```

or

```
prompt >> setenv CLASSPATH $PWD
```

For situations where packages are located in multiple subdirectory hierarchies, a list of paths can be specified. Suppose, for example, the directories are:

```
/User/austin/project1
/User/austin/project2
```

The appropriate command is:

```
prompt >> setenv CLASSPATH /User/austin/project1:/User/austin/project2
```

Directories are separated by a colon(:) in UNIX/Linux and Mac OS X, a semi-colon (;) in Windows.

Program Compilation

With the CLASSPATH in place, simply type:

```
prompt >> javac *.java
```

and the source code files will be compiled.

Remark. Hint for Program Design. Keep data private. This ensure that two variables with the same name, but in different packages will not have any unintended interactions.

Application 1. Two kinds of Apple, Fruit and Computers

Let's consider a program that works with two kinds of apple, fruit and computers, and draws upon classes in two packages. The directory and source code file names are:

```
Package                  Directory and Source Code
--------------------------------------------------
Base directory
   package fruit
```

```
                              /fruit/Apple.java
                              /fruit/Orange.java
        package computer
                              /computer/Apple.java
     --------------------------------------------------
```

Within the package fruit, the details of Apple.java and Orange.java are as follows:

source code

```
/*
 *  ===================================================
 *  Apple.java: An apple is a piece of fruit...
 *  ===================================================
 */

package fruit;

public class Apple {

      // Constructor

      public Apple() {}

      // Return description the fruit...

      public String toString() {
         String s = "An Apple is a type of Fruit!!!";
         return s;
      }

      public static void main ( String [] args ) {
         Apple a = new Apple();
         System.out.println( a );
      }
}
```

and ...

source code

```
/*
 *  ===================================================
 *  Orange.java: An orange is a piece of fruit...
 *  ===================================================
 */

package fruit;

public class Orange {

      // Constructor
      public Orange() {}

      // Return description the fruit...

      public String toString() {
         String s = "An Orange is a type of Fruit!!!";
         return s;
      }

      public static void main ( String [] args ) {
         Orange a = new Orange();
         System.out.println( a );
      }
}
```

And for the computer package, Apple.java contains:

source code

```
/*
 *  ===================================================
 *  Apple.java: Apple is a computer company ....
 *  ===================================================
 */

package computer;

public class Apple {

      // Constructor

      public Apple() {}

      // Return description the fruit...

      public String toString() {
         String s = "Apple is a computer company!!!";
         return s;
      }

      public static void main ( String [] args ) {
         Apple a = new Apple();
         System.out.println( a );
      }
}
```

The key point here is the Apple.java appears in both packages! Finally, we have a test program that imports both packages and then calls methods in the appropriate classes.

Apple Test Program. Here are the test program details:

source code

```
/*
 *  ===================================================
 *  TestApple.java: The Apple Test program ....
 *  ===================================================
 */

import fruit.*;
import computer.*;

public class TestApple {
   public static void main ( String [] args ) {

      // Create an apple using the complete path name...

      fruit.Apple a = new fruit.Apple();
      System.out.println( a );

      // Create an orange using the abbreviated path name.

      Orange or = new Orange();
      System.out.println( or );

      // Create an apple computer object .....

      computer.Apple c = new computer.Apple();
      System.out.println( c );
   }
}
```

Basic Program Compilation

To compile the program, first we set the classpath:

```
prompt >> setenv CLASSPATH $PWD
```

Then simply type:

```
prompt >> javac TestApple.java
```

The files before and after compilation are as follows:

```
Before Compilation              After Compilation
-------------------------------------------------
TestApple.java                     TestApple.java
fruit/Apple.java                 fruit/Apple.java
fruit/Orange.java               fruit/Orange.java
computer/Apple.java           computer/Apple.java
```

```
                                              TestApple.class
                                            fruit/Apple.class
                                           fruit/Orange.class
                                         computer/Apple.class
     -------------------------------------------------
```

Notice that the import statements

```
     import fruit.*;
     import computer.*;
```

in TestApple.java allow the java compiler to find and compile the Apple.java and Orange.java source code files in directories fruit and computer.

Running the Program

```
     prompt >> java TestApple
```

9.2 Working with JAR Files

A JAR (Java Archive) file is simply a ZIP file that contains classes, possibly other files that a program may need (e.g., image and sound files), and a manifest file describing the special features of the archive.

All of these files can be downloaded with a single HTTP request to the the server. Use of the ZIP compression format reduces the download time.

Creating JAR files

The most common command for creating a new JAR file has the format:

```
prompt >> jar cvf NameOfJARFile File1 File2 File3 ....
```

The **cvf** command line argument says:

- We wish to create (c) a JAR file.
- We wish the command to be verbose (v).
- We are designating the filename of the JAR file.

Example 1. From the base directory

```
prompt >> jar cvf Fruit.jar fruit/Apple.class fruit/Orange.class
```

creates a the JAR file Fruit.jar. An equivalent command is

```
prompt >> jar cvf Fruit.jar fruit/*.class
```

In either case, the output is:

```
Script started on Sat May  6 13:50:53 2006
prompt >>
prompt >> jar cvf Fruit.jar fruit/Apple.class fruit/Orange.class
added manifest
adding: Apple.class(in = 537) (out= 352)(deflated 34%)
adding: Orange.class(in = 540) (out= 352)(deflated 34%)
prompt >>
prompt >> exit
Script done on Sat May  6 13:51:31 2006
```

Inspecting the Contents of a JAR File

The syntax for inspecting the contents of the JAR file is:

```
prompt >> jar tvf NameOfJARFile
```

Here the command line options ”vf” are as previously explained. The command line option ”t” indicates that we want to see the table of contents for the jar file. e.g.,

```
prompt >> jar tvf Fruit.jar
```

produces the output:

```
Script started on Sat May  6 13:57:49 2006
prompt >>
prompt >> jar tvf Fruit.jar
     0 Sat May 06 13:51:24 EDT 2006 META-INF/
    70 Sat May 06 13:51:24 EDT 2006 META-INF/MANIFEST.MF
   537 Fri May 05 18:39:14 EDT 2006 Apple.class
   540 Fri May 05 18:39:14 EDT 2006 Orange.class
prompt >>
prompt >> exit
Script done on Sat May  6 13:58:07 2006
```

Using the code within a JAR file

To use the code within a jar file we follow the syntax:

```
prompt >> java -cp path-to-the-jar-file class-containing-the-main-method
```

Example. For our fruit application, both the Apple and Orange classes have main methods. Now suppose that the Fruit.jar file is moved to a bin directory (i.e., relative to the base directory, the JAR file is located in bin/Fruit.jar). To run the class file from the base directory

```
prompt >> java -cp bin/Fruit.jar Apple.class
```

Extracting the Contents from a JAR file

Syntax:

```
prompt >> java xvf NameOfTarFile
```

Example.

```
prompt >>
prompt >> jar xvf Fruit.jar
    created: META-INF/
  extracted: META-INF/MANIFEST.MF
  extracted: Apple.class
  extracted: Orange.class
prompt >>
```

The MANIFEST.MF file is located in the META-INF sub-directory.

JAR Files as Applications

You can run JAR-packaged applications with the Java interpreter. The basic command is:

```
prompt >> java -jar jar-file
```

The -jar flag tells the interpreter that the application is packaged in the JAR file format. You can only specify one JAR file, which must contain all the application-specific code. Before this command will work, however, the runtime environment needs to know which class within the JAR file is the application's entry point.

To indicate which class is the application's entry point, you must add a Main-Class header to the JAR file's manifest. The header takes the form:

Main-Class: classname

The header's value, classname, is the name of the class that's the application's entry point.

Example. The contents of mainclass.mf are:

```
1. Main-Class: fruit.Apple
2.
```

It's important to have a black line in mainclass.mf Now, type:

```
prompt >> jar cmf mainclass.mf Fruit.jar fruit/*.class
prompt >> java -jar Fruit.jar
An Apple is a type of Fruit!!!
prompt >>
```

9.3 Program Compilation with Ant

Ant (Another Neat Tool) is a ...

... platform-independent scripting tool for automating build processes.

It provides similar functionality to the UNIX utility program **Make**, but is implemented in Java, requires the Java Platform, and is ideal for building Java projects [1].

Ant supports a large number of built-in compilation tasks including creation/removal of directories, copying files from one location to another, setting of the classpaths, and compilation of java source code files.

Fast Answers to Frequently-Asked Questions:

1. **Where can I get Ant?**

   See http://ant.apache.org/resources.html

2. **What is the Relationship between Ant and Netbeans and Eclipse?**

   All of the main Java IDEs ship with Ant. So, for example, the Eclipse IDE can build and execute Ant scripts. The NetBeans IDE uses Ant for its internal build system. If you don't want to deal with the complexities of Ant, consider downloading and learning either Eclipse or Netbeans. One benefit in installing Ant this way is that you will get the most recent release of Ant at the time the IDE was released.

3. **I have a Windows computer. Can I run Ant?**

   Yes. For details, see: http://ant.apache.org/manual/install.html The following sets up the environment:

```
set ANT_HOME=c:\ant
set JAVA_HOME=c:\jdk-1.5.0.05
set PATH=%PATH%;%ANT_HOME%\bin
```

9.4 Application 2: Basic Program Compilation with Ant

To see how Ant works in practice, lets repeat Application 1 (Two kinds of Fruit, Apples and Computers), but use Ant to automate the program compilation. The packages fruit (containing Apple.java and Orange.java) and computer (containing Apple.java) will be as previously described. The test program is TestApple.java.

Abbreviated details of the manual approach to program compilation are,

```
prompt >> setenv CLASSPATH $PWD
prompt >> javac TestApple.java
```

Now let's repeat the compilation, but this time use a buildfile and ant. The details of **build.xml** are as follows:

source code

```
<project>

    <target name="clean">
        <delete dir="build"/>
    </target>

    <target name="compile">
        <mkdir dir="build/classes"/>
        <javac srcdir="fruit" destdir="build/classes"/>
    </target>

    <target name="jar">
        <mkdir dir="build/jar"/>
        <jar destfile="build/jar/Fruit.jar" basedir="build/classes">
            <manifest>
                <attribute name="Main-Class" value="fruit.Apple"/>
            </manifest>
        </jar>
    </target>

    <target name="run">
        <java jar="build/jar/Fruit.jar" fork="true"/>
    </target>

</project>
```

The key points to note are as follows:

1. Ant uses XML to describe the build process and its dependencies, and by default, the XML file is named build.xml.

2. This script file has four targets:

```
clean   -- remove the contents of the build directory
compile -- compile the source code files
    jar -- generate jar files from the class files
    run -- execute the program (needs a manifest file).
```

3. Within each target are the actions that Ant must take to build that target; these are performed using built-in tasks. For example, to compile the java program, ant will first create a `build/classes` directory (Ant will only do so if it does not already exist), and then invoke the java compiler `javac` to compile the java source code files in `fruit`, and place the compiled bytecodes in `build/classes`.

Therefore, in our first example, the build and compilation tasks are `mkdir` and `javac`.

Compiling the Program with Ant

To compile the program, just type:

```
    Script started on Sun May  7 10:59:08 2006
    prompt >>
    prompt >> ant compile
    Buildfile: build.xml

    compile:
        [mkdir] Created dir:
                /Users/austin/java.d/package-ant.d/build/classes
        [javac] Compiling 2 source files to
                /Users/austin/java.d/package-ant.d/build/classes

    BUILD SUCCESSFUL
    Total time: 2 seconds
    prompt >>
```

The file structure after compilation is:

```
prompt >> ls -lsR
total 40
16 -rw-r--r--   1 austin  austin  7036 May  7 10:57 README.txt
 8 -rw-r--r--   1 austin  austin   693 May  7 10:28 TestApple.java
 0 drwxr-xr-x   3 austin  austin   102 May  7 10:59 build
 8 -rw-r--r--   1 austin  austin   593 May  7 10:39 build.xml
 0 drwxr-xr-x   4 austin  austin   136 May  7 10:28 computer
 0 drwxr-xr-x   4 austin  austin   136 May  7 10:54 fruit
 8 -rw-r--r--   1 austin  austin    25 May  7 10:28 mainclass.mf

./build:
total 0
0 drwxr-xr-x   3 austin  austin  102 May  7 10:59 classes

./build/classes:
total 0
0 drwxr-xr-x   4 austin  austin  136 May  7 10:59 fruit

./build/classes/fruit:
total 16
8 -rw-r--r--   1 austin  austin  437 May  7 10:59 Apple.class
8 -rw-r--r--   1 austin  austin  439 May  7 10:59 Orange.class

./computer:
total 16
8 -rw-r--r--   1 austin  austin  542 May  7 10:28 Apple.java

./fruit:
total 16
8 -rw-r--r--   1 austin  austin  538 May  7 10:28 Apple.java
8 -rw-r--r--   1 austin  austin  544 May  7 10:28 Orange.java
prompt >>
prompt >> exit
Script done on Sun May  7 10:59:41 2006
```

The source codee files `fruit/Apple.java` and `fruit/Orange.java` are compiled into byte-codes `fruit/Apple.class` and `fruit/Orange.class`, respectively. Also, notice that the java compiler is instructed to only look in `fruit/` for source code files, so `computer/Apple.java` remains uncompiled. There are two ways of solving this problem:

1. Broaden the compile directive to look in both `/fruit` and `/compile`, or

2. Create separate compile directives for the fruit and computer programs.

Creating JAR Files with Ant

Just type:

```
     prompt >> ant jar
```

The compilation output and abbreviated file structure is as follows:

```
Script started on Sun May  7 11:06:05 2006
prompt >> ant jar
Buildfile: build.xml

jar:
    [mkdir] Created dir:
            /Users/austin/java.d/package-ant.d/build/jar
      [jar] Building jar:
            /Users/austin/java.d/package-ant.d/build/jar/Fruit.jar

BUILD SUCCESSFUL
Total time: 1 second
prompt >>

prompt >> ls -lsR
total 40
 0 drwxr-xr-x   4 austin  austin   136 May  7 11:06 build

./build:
total 0
0 drwxr-xr-x   3 austin  austin  102 May  7 10:59 classes
0 drwxr-xr-x   3 austin  austin  102 May  7 11:06 jar

./build/jar:
total 8
8 -rw-r--r--   1 austin  austin  1242 May  7 11:06 Fruit.jar

prompt >>
prompt >> exit
Script done on Sun May  7 11:06:34 2006
```

Running Programs with Ant

```
     prompt >> ant run
```

This generates the output:

```
Script started on Sun May  7 11:09:39 2006
prompt >> ant run
Buildfile: build.xml

run:
     [java] An Apple is a type of Fruit!!!

BUILD SUCCESSFUL
Total time: 1 second
prompt >>
prompt >> exit
Script done on Sun May  7 11:09:47 2006
```

The equivalent keyboard command is:

```
     prompt >> java -jar build/jar/Fruit.jar
```

Combining Steps: Compile, jar and run

All three steps can be achieved in one step:

```
     prompt >> ant compile jar run
```

9.5 Application 3: Not so Basic Program Compilation with Ant

Application 2 requires that the software developer be intimately involved with each step of the program development (e.g., clean, then compile, then run). In our third application, we ask Ant to step things up a bit.

1. The source code files (e.g., in /src folder) are separated from the compile files (e.g., in /build folder).

1. The Apple source code files (e.g., /src/fruit/Apple.java) are separated from the test folder (e.g., /test/TestApple.java).

2. Dependencies among various aspects of the compilation are introduced (e.g., you cannot run a program unless it has already been compiled).

Before compilation the files are as follows:

```
total 16
8 -rwxr-xr-x  1 austin  staff  1783 Oct 10 17:28 build.xml
0 drwxr-xr-x  5 austin  staff   170 Oct 10 10:51 src
0 -rw-r--r--  1 austin  staff     0 Oct 10 17:31 temp
0 drwxr-xr-x  3 austin  staff   102 Oct 10 17:24 test
```

```
./src:
total 0
0 drwxr-xr-x  4 austin  staff  136 Oct 10 10:12 computer
0 drwxr-xr-x  6 austin  staff  204 Oct 10 16:32 fruit
0 drwxr-xr-x  6 austin  staff  204 Oct 10 11:00 music

./src/computer:
total 16
8 -rwxr-xr-x  1 austin  staff  542 Oct 10 10:12 Apple.java

./src/fruit:
total 32
8 -rwxr-xr-x  1 austin  staff  538 Oct 10 10:12 Apple.java
8 -rwxr-xr-x  1 austin  staff  544 Oct 10 10:12 Orange.java

./src/music:
total 32
8 -rwxr-xr-x  1 austin  staff  638 Oct 10 10:53 Apple.java
8 -rwxr-xr-x  1 austin  staff  775 Oct 10 11:00 Beatles.java

./test:
total 8
8 -rwxr-xr-x  1 austin  staff  814 Oct 10 17:24 TestApple.java
```

Ant works with the build file:

source code

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

    <target name="clean">
        <delete dir="build"/>
        <delete dir="lib"/>
    </target>

    <target name="compile">
        <mkdir dir="build/classes/fruit"/>
        <javac srcdir="src/fruit" destdir="build/classes"/>
        <mkdir dir="build/classes/computer"/>
        <javac srcdir="src/computer" destdir="build/classes"/>
        <mkdir dir="build/classes/music"/>
        <javac srcdir="src/music" destdir="build/classes"/>
    </target>

    <target name="jar" depends="compile">
        <mkdir dir="lib"/>
        <jar destfile="lib/Fruit.jar" basedir="build/classes">
            <manifest>
                <attribute name="Main-Class" value="fruit.Apple"/>
            </manifest>
        </jar>
        <jar destfile="lib/Computer.jar" basedir="build/classes">
            <manifest>
                <attribute name="Main-Class" value="computer.Apple"/>
            </manifest>
        </jar>
        <jar destfile="lib/Music.jar" basedir="build/classes">
            <manifest>
                <attribute name="Main-Class" value="music.Apple"/>
            </manifest>
        </jar>
    </target>

    <target name="run" depends="jar">
        <java jar="lib/Fruit.jar"    fork="true"/>
        <java jar="lib/Computer.jar" fork="true"/>
        <java jar="lib/Music.jar"    fork="true"/>
    </target>

    <target name="test" depends="jar">
        <mkdir dir="build/classes/test"/>
        <javac srcdir="test" destdir="build/classes"/>
        <jar destfile="lib/TestApple.jar" basedir="build/classes">
            <manifest>
                <attribute name="Main-Class" value="TestApple"/>
            </manifest>
        </jar>
        <java jar="lib/TestApple.jar"    fork="true"/>
    </target>

</project>
```

There are five targets:

1. `<target name="clean">` removes the class files and the library of jar files (stored in lib/*.jar).
2. `<target name="compile">` compiles source code in the directories:

    ```
    src/fruit,
    src/computer, and
    src/music.
    ```

    The bytecodes are put in

    ```
    build/classes/fruit
    build/classes/computer
    build/classes/music
    ```

3. `<target name="jar" depends="compile">` creates jar files for each of the three cases and puts them in the lib/ directory; i.e.,

    ```
    lib/Fruit.jar,
    lib/Computer.jar,
    lib/Music.jar,
    ```

4. **`<target name="run" depends="jar">`** executes the main methods in each of the jar files:

```
lib/Fruit.jar,
lib/Computer.jar,
lib/Music.jar,
```

5. **`<target name="test" depends="jar">`** compile the source code

```
test/TestApple.java
```

   into the jar file

```
test/TestApple.jar
```

   Execute the main method in test/TestApple.jar.

Dependencies between the targets:

```
<target name="test" depends="jar">
<target name="run" depends="jar">
<target name="jar" depends="compile">
```

Therefore, you should be able to clean the system (i.e., ant clean) and then run the ”test” target. This gives:

```
Buildfile: build.xml

compile:
    [mkdir] Created dir: /Users/austin/java.d/package-ant2.d/build/classes/fruit
    [javac] Compiling 2 source files to /Users/austin/java.d/package-ant2.d/build/classes
    [mkdir] Created dir: /Users/austin/java.d/package-ant2.d/build/classes/computer
    [javac] Compiling 1 source file to /Users/austin/java.d/package-ant2.d/build/classes
    [mkdir] Created dir: /Users/austin/java.d/package-ant2.d/build/classes/music
    [javac] Compiling 2 source files to /Users/austin/java.d/package-ant2.d/build/classes

jar:
    [mkdir] Created dir: /Users/austin/java.d/package-ant2.d/lib
      [jar] Building jar: /Users/austin/java.d/package-ant2.d/lib/Fruit.jar
      [jar] Building jar: /Users/austin/java.d/package-ant2.d/lib/Computer.jar
      [jar] Building jar: /Users/austin/java.d/package-ant2.d/lib/Music.jar

test:
    [mkdir] Created dir: /Users/austin/java.d/package-ant2.d/build/classes/test
    [javac] Compiling 1 source file to /Users/austin/java.d/package-ant2.d/build/classes
      [jar] Building jar: /Users/austin/java.d/package-ant2.d/lib/TestApple.jar
     [java] An Apple is a type of Fruit!!!
     [java] An Orange is a type of Fruit!!!
     [java] Apple is a computer company!!!
     [java] Apple is a recording company!!!
     [java] The Beatles: John, Paul, George, Ringo
     [java]
     [java]

BUILD SUCCESSFUL
Total time: 1 second
```

9.6 Application 4: Dealing with many Jar Files

Now let's deal the problem of compiling a program that has many (perhaps dozens) or jar files. Rather than explicity list every single filename, we can ...

... use regular expressions to vastly simplify the collection of jar file names.

Consider:

```
total 3320
   0 drwxr-xr-x@  4 austin  staff     136 Aug 13  2012 antlr
 272 -rwxr-xr-x@  1 austin  staff  136065 Aug 13  2012 aterm-java-1.6.jar
   0 drwxr-xr-x  19 austin  staff     646 Aug 13  2012 jena
   0 drwxr-xr-x   5 austin  staff     170 Aug 13  2012 jgrapht
1224 -rwxr-xr-x@  1 austin  staff  622820 Aug 13  2012 pellet-core.jar
 520 -rwxr-xr-x@  1 austin  staff  266122 Aug 13  2012 pellet-datatypes.jar
 104 -rwxr-xr-x@  1 austin  staff   52156 Aug 13  2012 pellet-el.jar
   0 drwxr-xr-x   4 austin  staff     136 Aug 13  2012 pellet-jena
 792 -rwxr-xr-x@  1 austin  staff  404602 Aug 13  2012 pellet-query.jar
 408 -rwxr-xr-x@  1 austin  staff  207217 Aug 13  2012 pellet-rules.jar

./antlr:
total 312
  8 -rwxr-xr-x@ 1 austin  staff    1429 Aug 13  2012 LICENSE.txt
304 -rwxr-xr-x@ 1 austin  staff  151989 Aug 13  2012 antlr-runtime-3.2.jar

./jena:
total 22904
3288 -rwxr-xr-x@ 1 austin  staff  1680523 Aug 13  2012 arq-2.8.4.jar
   8 -rwxr-xr-x@ 1 austin  staff     1682 Aug 13  2012 copyright.txt
6320 -rwxr-xr-x@ 1 austin  staff  3233439 Aug 13  2012 icu4j-3.4.4.jar
 416 -rwxr-xr-x@ 1 austin  staff   210961 Aug 13  2012 iri-0.8-sources.jar
 304 -rwxr-xr-x@ 1 austin  staff   151589 Aug 13  2012 iri-0.8.jar
2944 -rwxr-xr-x@ 1 austin  staff  1506473 Aug 13  2012 jena-2.6.3-tests.jar
3712 -rwxr-xr-x@ 1 austin  staff  1900385 Aug 13  2012 jena-2.6.3.jar
 392 -rwxr-xr-x@ 1 austin  staff   198945 Aug 13  2012 junit-4.5.jar
 704 -rwxr-xr-x@ 1 austin  staff   358180 Aug 13  2012 log4j-1.2.13.jar
1304 -rwxr-xr-x  1 austin  staff   665064 Aug 13  2012 lucene-core-2.3.1.jar
  48 -rwxr-xr-x@ 1 austin  staff    23445 Aug 13  2012 slf4j-api-1.5.8.jar
  24 -rwxr-xr-x@ 1 austin  staff     9679 Aug 13  2012 slf4j-log4j12-1.5.8.jar
  56 -rwxr-xr-x@ 1 austin  staff    26514 Aug 13  2012 stax-api-1.0.1.jar
   8 -rwxr-xr-x@ 1 austin  staff        6 Aug 13  2012 version.txt
1024 -rwxr-xr-x@ 1 austin  staff   524224 Aug 13  2012 wstx-asl-3.2.9.jar
2352 -rwxr-xr-x@ 1 austin  staff  1203860 Aug 13  2012 xercesImpl-2.7.1.jar
```

Now we can simply write:

```
<!-- Define properties and filesets for classpath -->

<property name="lib.dir"   value="lib" />
<property name="build.dir" value="build" />

<path id="files-classpath">
    <fileset dir="${lib.dir}"         includes="**/*.jar"/>
    <fileset dir="${lib.dir}/jena"    includes="**/*.jar"/>
    <fileset dir="${lib.dir}/antlr"   includes="**/*.jar"/>
    <pathelement path="${build.dir}"/>
</path>
```

to include all of the jar files in the lib folder, plus the classes positioned in the build directory.

9.7 Application 5: Using Ant to Move Files Around

In the script:

```
<!-- ========================================== -->
<!-- Compile Java source code.                  -->
<!-- ========================================== -->

<target name="compile" description="Compile java source code">
    <mkdir dir="build" />
    <mkdir dir="build/demo" />
    <javac srcdir="./src" destdir="./build"
           classpath    = "{java.class.path}"
           classpathref = "files-classpath"
           fork="true"/>

    <mkdir dir ="build/demo/data" />
    <copy todir="build/demo/data">
        <fileset dir="src/demo/data" />
    </copy>
</target>
```

a set of data files is copied from directory (folder) `src/demo/data` to directory `build/demo/data`.

9.8 Application 6: Using Ant to Run Programs having Arguments

Sometimes java application programs will import files, either from pre-defined directories (e.g., images files), or as data files specified as a program argument. The latter can be incorporated into ant files, e.g.,

```
<!-- Program 02: Run demo.DateTimeBrowser -->

<target name="run02" depends = "compile" description="Run demo.DateTimeBrowser">
    <java classname = "demo.DateTimeBrowser"
          classpath    = "{java.class.path}"
          classpathref = "files-classpath" fork="true">
```

```
              <arg value="./build/demo/data/DateTimeFile.dat"/>
        </java>
    </target>
```

executes the program `demo.DateTimeBrowser` with the program argument:

```
./build/demo/data/DateTimeFile.dat.
```

Bibliography

[1] Holzner S. *Ant - The Definitive Guide (2nd Edition)*. O'Reilly Media, 2005.

Part I

Appendices

Index

Ant, 10
- creating JAR files, 13
- program compilation, 10–13
- running programs, 14–20

CLASSPATH, 2

Eclipse, 10

JAR files, 7–9
Java
- ant, 10
- package, 1

Java Archive (JAR) files, 7

MANIFEST file, 8

Netbeans, 10