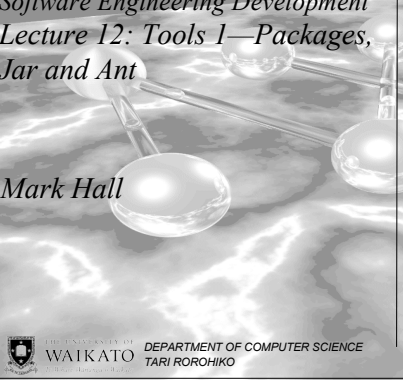
COMP241
Software Engineering Development
Lecture 12: Tools 1—Packages, Jar and Ant

Mark Hall



- Packages
  - Naming conventions
- Access modifiers
- JAR
  - Exectable jar
- Project directory structure
- Ant
  - build.xml
  - Targets
  - Tasks
  - Properties
  - Paths

DEPARTMENT OF COMPUTER SCIENCE
TARI ROROHIKO

Packages

- A package is a set of classes that have been grouped together
- The Java API is a huge collection of packages
  - `java.lang`
  - `java.net`
  - `java.awt.event`
  - `org.img.CORBA`

Packages

- Packages represent a **hierarchical directory structure**
- Use the `import` statement to use classes from another package
  - Note, if you don't import you can still use a class from another package but you will have to provide the *fully qualified* name
    E.g. `javax.swing.JPanel j = new javax.swing.JPanel();`
- Use the `package` statement to put a class into a package
- If the `package` statement is missing, your class is put into the "default package".

Naming conventions

- The prefix of a package name is always written in all lowercase ASCII letters
  - Recommended that it should be one of the top level domain names (com, edu, gov, mil, net, org) or two letter country codes
- Subsequent components of the package name vary according to an organization's own internal naming conventions
  - Might specify that certain directory name components be division, department, project, etc. names
- E.g. `com.sun.eng`, `com.apple.quicktime.v2`

Access Modifiers

- Access modifiers control who may access
  - classes
  - methods
  - data members
- Java has four access modifiers
  - `public`
  - `protected`
  - *(friendly)*
  - `private`

Meaning of Access Modifiers

Keyword	**Allow access from**
`public`	anywhere
`protected`	the same package, or any subclass of the present class
`(friendly)`	anywhere in the same package
`private`	only the same class

JAR-Java ARchive

- Benefits
  - Several files in a single archive
  - Simpler file transfers
  - Compression
  - Security
  - Portability

Basic Usage of `jar`

Operation	**Command**
Create archive	`jar cf` <*archive*> <*files*>
View archive	`jar tf` <*archive*>
Extract contents of archive	`jar xf` <*archive*>
Extract specific files	`jar xf` <*archive*> <*files*>
Create archive with manifest	`jar cfm` <*archive*> <*manifest*> <*files*>

The Manifest File

- Each JAR archive includes a file
  `META-INF/MANIFEST.MF`
- Contains Meta-Informaton, e.g. about
  - Main class
  - Download extensions
  - Vendor
  - Cryptographic signatures
- Useful to create an *executable* JAR file:
  `java -jar` <*archive*>`.jar`

Creating an Executable JAR File

- Create a manifest template (e.g. manifest.mf) file that contains:
  - `Main-Class: <fully qualified name of class>`
    E.g. `Main-Class: org.groovy.MyFunkyApplication`
- Create the jar file
  `jar cfm funkyapp.jar manifest.mf -C classes org`

Directory Structure

Simple:

- All files in the same directories
  `.java`
  `.class`
  `.html`
- Hard to maintain
- How to clean up?

Advanced:

- Different directories for files of different type
  - `src/` — `.java` files
  - `classes/` — `.class` files
  - `jar/` — JAR files
  - Etc.

Maven Directory Structure

```
project/
  LICENSE.txt
  README.txt
  build.xml                    ANT build file
  src/
      main/
            java/              Application/Library sources
            resources/         Application/Library resources
            config/            Configuration files
            webapps/           Web application sources
      test/
            java/              Test sources
            resources/         Test resources
  target/                      Holds the output of the build
```

Ant

Ant — the Java Build Tool

- Automate the build process
- Like `make`, but specifically designed for Java
- XML build file
- Platform-independent

Ant build file: "build.xml"

```
<project name="comp241"
         default="jar">
  <target name="compile">
     . . .
  </target>
  <target name="jar"
          depends="compile">
     . . .
  </target>
</project>
```

XML Elements:
<name attrib="value">…</name>
Or
<name attrib="value"/>

Targets

- A target is a set of tasks you want to be executed
- Each project defines one or more targets
- A target can *depend* on other targets
- Each target gets executed only once, even when more than one target depends on it
- Commonly used target names:
  - init, compile, jar (dist), run, test, clean

Targets (example)

```
<project name="comp241"
         default="jar">
  <target name="compile">
     . . .
  </target>
  <target name="jar"
    description="Create jar file"
    depends="compile">
     . . .
  </target>
  <target name="doc"
    depends="site, api">
     . . .
  </target>
</project>
```


Tasks

- A task is a piece of code that can be executed
- Common structure:

```
<name attribute1="value1" attribute2="value2 …/>
```

- Useful built-in tasks:
  - `javac` Compiles a Java source tree
  - `jar` Jars a set of files
  - `java` Executes a Java class
  - `mkdir` Creates a directory

Tasks (example)

```
<javac srcdir="src"
       destdir="classes"
       classpath="jar/xyz.jar"
       debug="on"
       source="1.5"/>
```

Compiles all .java files under the src directory, and stores the .class files in the classes directory. The classpath used includes jar/xyz.jar, and compiling with debug information is on. The source level is 1.5.

Properties

- A project can have a set of properties (set in the build file by the properties task, or outside ant)
- A property has a name (case-sensitive) and a value
- Use a property by placing its name between "${" and "}"
- Built-in properties
  - basedir, ant.file, ant.version etc.

Path-like structures

- The location attribute of the <pathelement> tag specifies a single file or directory relative to the projects base directory (or an absolute name)
- The path attribute of the <pathelement> tag accepts colon- or semicolon-separated lists of locations
- Othe possible tags include <dirset>, <fileset> and <filelist>

Classpath example

```
<classpath>
  <pathelement path="${classpath}"/>
  <fileset dir="lib">
      <include name="**/*.jar"/>
  </fileset>
  <pathelement location="classes"/>
  <dirset dir="${build.dir}">
      <include name="apps/**/classes"/>
      <exclude name="apps/**/Test/**"/>
  </dirset>
</classpath>
```

Builds a path that holds the value of ${classpath}, followed by all jar files in the lib directory, the classes directory, all directories named classes under the apps subdirectory of property build.dir, except those that have the text Test in their name.