Debian policy for Java

Niels Thykier

Ola Lundqvist

Stephane Bortzmeyer

Debian policy for Java

by Niels Thykier, Ola Lundqvist, Stephane Bortzmeyer, and

$Revision:$ $Date:$ Edition

Abstract

This is the java policy for Debian. It begins with a background description, continues with the real policy, some issues to discuss and ends with some advices to java packagers.

The policy covers java virtual machines, java compilers, java programs and java libraries.

Table of Contents

Chapter 1. Background

There are several "subpolicies" in Debian. They all want to make the Debian Policy (http://www.debian.org/doc/debian-policy) more precise when it comes to a specific subject. See the Emacs subpolicy in package emacsen-common for instance. As far as I know, the only subpolicy for a programming language, is that of Perl (http://non-us.debian.org/~hertzog/perl-policy.html/).

Feel free to report comments, suggestions and/or disagreements to the java-common package (<java-common@packages.debian.org>) or the Debian Java mailing list <debian-java@lists.debian.org>. Change requests should be sent as a bug to the java-common package.

Chapter 2. Policy

Virtual packages are created: *java1-runtime* and *java2-runtime*.

Packages written in Java are separated in two categories: programs and libraries. Programs are intended to be run by end-users. Libraries are intended to help programs to run and to be used by developers.

Both *must* be shipped as Java bytecode (`*.class` files, packaged in a `*.jar` archive) and with an "Architecture: all". There are rare exceptions to this such as Eclipse SWT. Exceptions to this rule can only be granted by the Java Team. Requests *must* be sent to `<debian-java@lists.debian.org>`.

The Java bytecode *may* additionally be shipped as machine code, as produced for example by the GNU Compiler for Java, in a separate architecture-specific package.

Programs and libraries *should* enable JUnit tests, if these are present. The build *may* ignore test failures.

2.1. Virtual machines

Java virtual machines *must* depend on java-common. They can also provide the runtime environment that the package contains (*java1-runtime* and/or *java2-runtime*). If it does not provide the files itself it *must* depend on the needed runtime environment.

Packages that contain a runtime conforming to the Java 1.1 specification *should* provide *java1-runtime*. Packages that contain a runtime conforming to the Java 2 specification *should* provide *java2-runtime*. If a package conforms to both, then it *should* provide both; however, packages that do not implement the methods from Java 1.1 that have been deprecated in Java 2 *must not* provide *java1-runtime*.

They *should* use `/etc/alternatives` for the name 'java' if they are command-line compatible with the Sun's java program.

They *should* have a CLASSPATH predefined which include the needed runtime environment.

If a given source (like the JDK does) brings both a compiler and a virtual machine, you *may* name the compiler package xxxx-dev.

Some Java classes implement their routines using a "native" language (such as C). This native code is compiled and stored in dynamic libraries (such as JNI modules) that are loaded at runtime. If a virtual machine supports native code, it *must* include the directory `/usr/lib/jni` in its search path for these dynamic libraries.

2.2. Java programs

Programs *must* have one or more executables in one or more of the directories defined by 9.1 (http://www.debian.org/doc/debian-policy/ch-opersys.html#s9.1) of the Debian Policy. These *must* either be a wrapper script or a symlink to an executable jar. In any case, they *must* run without specific environment variables (see Policy 10.9 (http://www.debian.org/doc/debian-policy/ch-opersys.html#s10.9)), for instance CLASSPATH. They *must* respect the Policy rules for executables (for instance a manual page per executable, see Policy 13.1 (http://www.debian.org/doc/debian-policy/ch-docs.html#s13.1)).

If the package installs a jar (or a symlink to a jar) as an executable the package *must* have an absolute dependency on jarwrapper or an equivalent package, which allows jar files to be executed directly from PATH like a normal program. The package *must* also ensure that the correct class is used as main-class. As an example jarwrapper uses the Main-Class attribute from the main part of the Manifest of the jar file to determine this.

Additional classes in the package must be packaged in one or more JARs which can be put into /usr/share/java (if they are intended to be used by other programs) or into a private directory in /usr/share/<package>.

Programs *must* depend on the needed runtime environment (*java1-runtime* and/or *java2-runtime*).

There is no naming rules for programs, they are ordinary programs, from the user point of view.

2.3. Java libraries

Libraries are not separated between developers (-dev) and users versions, since this is meaningless in Java.

Java libraries packages *must* be named libXXX[version]-java (without the brackets), where the version part is optional and *should* only contain the necessary part. The version part *should* only be used to avoid naming collisions. The XXX part is the actual package name used in the text below.

Their classes *must* be in `jar` archive(s) in the directory `/usr/share/java`, with the name `packagename[-extraname]-fullversion.jar`. The extraname is optional and used internally within the package to separate the different jars provided by the package. The fullversion is the version of that jar file. In some cases that is not the same as the package version.

Some package *must* also provide a symbolic link from `packagename-extraname.jar` to the most compatible version of the available `packagename-extraname-version.jar` files.

Class files in a java library *must* be built with debug symbols.

All jar files *must* have a well-documented CLASSPATH, so that developers should know what to add to their wrappers.

This applies only to libraries, *not* to the core classes provided by a the runtime environment.

Some Java libraries rely on code written in a "native" language, such as JNI (Java Native Interface) code. This native code is compiled into separate dynamic libraries which are loaded by the Java virtual machine at runtime.

If a Java library relies on native code, the dynamic libraries containing this compiled native code *should* be installed into the directory `/usr/lib/jni`. These dynamic libraries *should* be shipped in a separate architecture-specific package named libXXX[version]-jni. The package containing the Java bytecode (generally libXXX[version]-java) *should* depend on this package.

There may be situations, such as with very small packages, where it is better to bundle the Java code and the native code together into a single package. Such packages *should* be architecture-specific and follow the usual libXXX[version]-java naming convention.

Java library packages *should* compile the javadoc API of the library. The API *must* link against the javadoc API of the libraries it depends on. This includes the core java classes, which are provided by *default-jdk-doc*. The API *must* be registered with doc-base and *must* be installed in `/usr/share/doc/<package>/api/` or `/usr/share/doc/<package>/api-<component>/`.

The API *must* be placed in a separate doc package. This package *must* recommend the doc packages it was linked against.

2.4. Native Java Bytecode (gcj packages)

Java bytecode compiled into native code is referred to as gcj-code and packages containing gcj-code as gcj-packages.

gcj-packages has been added in order to improve performance of Java libraries and programs. This is particularly useful on architectures where the JVM does not have a JIT. However, this performance comes at the cost of size, extra compilation time and creates architecture dependent packages.

Packages *must not* ship gcj-code without the permission of the Java team (<`debian-java@lists.debian.org`>). Source packages that shipped gcj-packages as of March 22nd,2010, have been given this permission through the ratification of this policy.

A request for permission to add gcj should convince the Java Team that the performance boost of adding the gcj-packages out-weights the disadvantages.

Source packages compiling gcj-packages *must* Build-Depend on *gcj-native-helper* (formerly known as *default-jdk-builddep*). The gcj-code *should* only be shipped for a selected set of architectures.

The gcj-code *must* be installed in `/usr/lib/gcj/` and shipped separately from the original jar file. The gcj-package *must* also install the classmap file generated by aot-compile in `/usr/share/gcj/classmap.d/`.

The gcj-package *must* call rebuild-gcj-db in the postinst and postrm script, if rebuild-gcj-db is present.

The gcj-package *must* depend on the package providing the jar file, it is a native compilation. The package containing the jar file *must* declare either a Suggests or a Recommends relationship on the gcj-package.

2.5. Main, contrib or non-free

About politics: packaging Java stuff changes nothing to the rules Debian uses to find if a program is free or not. Since there are not many free Java tools, keep in mind the following:

- If your source package can compile (correctly) only with non-free tools, it cannot go to main. If your package itself is free, it *must* go to contrib.
- If your binary package can run only with non-free virtual machines (GNU-Classpath (http://www.gnu.org/software/classpath) has a list of free versions), it cannot go to main. If your package itself is free, it *must* go to contrib. Since java libraries do not have a runtime dependency, this rule does not apply to them.

Chapter 3. Issues to discuss

The following points are discussions about the policy, either because they have to be studied more, or are controversial.

- Name and existence of the repository. It was removed in the latest version.
- The symbolic links in `/usr/share/java` be made by a script instead, similar to the c-libraries.
- Core classes (`java.*`). More study needed.
- Sun's Community Source Licence. Can we use it? How? The 2.3 version of the text can be found here (http://wwws.sun.com/software/java2/license.html).
- All jars must have a good CLASSPATH documentation, but how should it be documented. The best solution is probably in some computer parsable format to make it even easier for the developer.

  It should exist some tool to parse this. How should it work?

  Should the tool also be used to create the necessary symbolic links needed by servlets under tomcat?

- Should there be a default CLASSPATH, similar to a repository? Which jars should be included in that? A standard and one optional part? If there are a default CLASSPATH (in the wrapper) how should it be overridden?
- How to check for a good enough JVM, and to select a proper one to use. Are `/etc/alternatives` not good enough?
- Should the JVM internal classes be possible to override entirely and how?

Chapter 4. Advices to Java packagers

Warning: These are just advices, they are not part of the policy.

- Be sure to manage all dependencies by hand in `debian/control`. Debian development tools cannot find them automatically like they do with C programs and libraries (or like dh_perl does it for Perl, a volunteer to write dh_java would be welcome).
- You can suppress many calls in `debian/rules` which are meaningless for Java, like dh_strip and dh_shlibdeps.
- Source package handling is painful, since most Java upstream programs come with `.class` files. I suggest to make a new `.orig` tarball after cleaning them, otherwise, dpkg-source will complain.
- Java properties files are probably better under `/etc` and flagged as configuration files (this will be integrated in the policy, one day).
- Java packages *should* be built with *default-jdk* if possible.