Jalopy and CheckStyle

features … installing … configuring … using

Mark Volkmann

Partner

Object Computing, Inc.

1/9/03

OBJECT COMPUTING, INC.

Jalopy and CheckStyle Common Characteristics

- Open source
- Available at SourceForge
- Can be run from command-line
- Can be run from an Ant build file
- Can be integrated with an IDE
  - not covered here since there are too many variations in IDE setup
  - see documentation of Jalopy and CheckStyle for instructions on how to integrate with a particular IDE

Jalopy Features

- Reformats code to a single style
- BSD license
- Usage scenario
  - when developers get code from the source repository they use Jalopy to reformat it to their preferred style
  - before they commit changes to the source repository they run Jalopy again to reformat the code to a style selected for the project
- Source repositories
  - consider hooking Jalopy into check-in/commit functionality of repository so that developers can't bypass this step
  - putting code back in a common format is essential so that reasonable diffs can be performed to determine changes made between revisions

Jalopy Features (Cont'd)

- Can control
  - brace placement
  - whitespace usage / indentation / line wrapping
  - code order
  - import optimization
  - and more
- IDE Plug-ins
  - Eclipse 2.0
  - JBuilder 5.0 or above
  - JDeveloper 9i (Oracle)
  - jEdit 4.1pre1 or above
  - NetBeans 3.3 or above
  - Sun ONE Studio 4 (based on NetBeans)

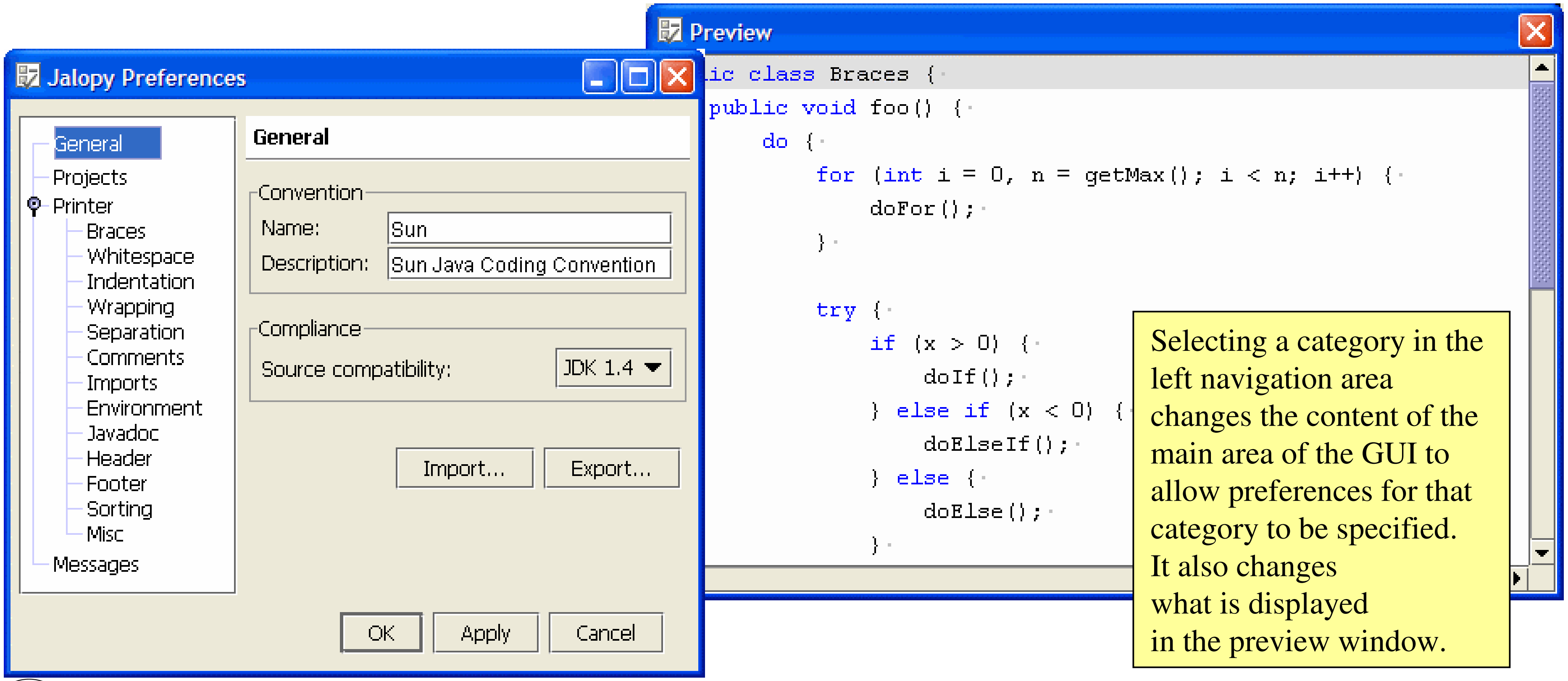
• three possible settings: disabled, expand and collapse
• when set to **expand**
all wildcard imports are replaced by explicit imports
• when set to **collapse**,
multiple explicit imports for the same package are replaced by a single wildcard import for the package

Installing Jalopy

- Download from
  - http://jalopy.sourceforge.net
- To install
  - unzip
  - add its bin directory to PATH environment variable
    - only needed to run from command-line

Configuring Jalopy

- To modify coding style enforced by Jalopy
  - run preferences script (either .bat or .sh) in bin directory
  - opens a Swing GUI where coding style preferences can be specified



General

Convention

Name: Sun

Description: Sun Java Coding Convention

Compliance

Source compatibility: JDK 1.4

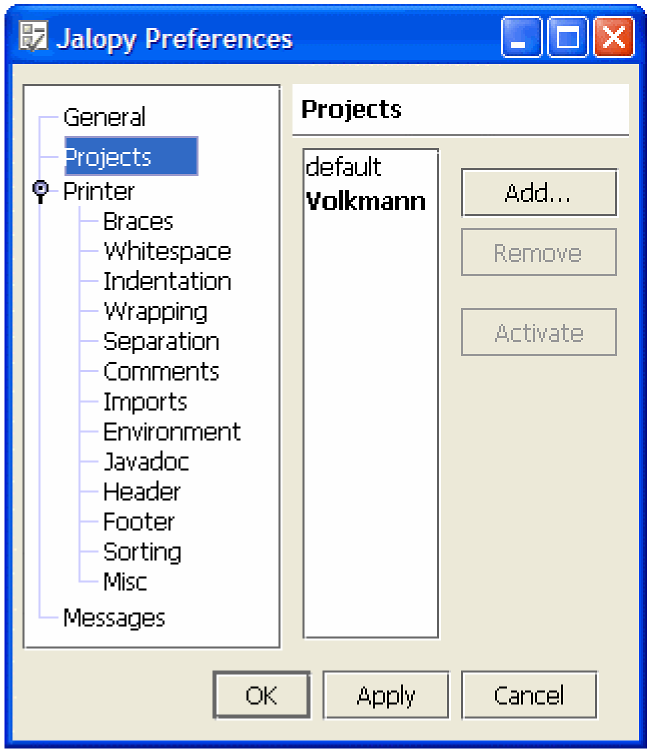
```
ic class Braces {
public void foo() {
    do {
        for (int i = 0, n = getMax(); i < n; i++) {
            doFor();
        }

        try {
            if (x > 0) {
                doIf();
            } else if (x < 0) {
                doElseIf();
            } else {
                doElse();
            }
```

Selecting a category in the left navigation area changes the content of the main area of the GUI to allow preferences for that category to be specified. It also changes what is displayed in the preview window.

Jalopy Projects

- A Jalopy Project is a set of preferences
  - multiple projects can be created
- Saved in separate, binary files
  - this makes using the GUI the only practical way to view and modify preferences
  - under Windows XP, these are stored under C:\Documents and Settings*user-name*\.jalopy
- To view current list of projects
  - click on the "Projects" category
- Default project
  - is created first time GUI is used



Jalopy Projects (Cont’d)

- Active project (in bold)
  - affected by changes made under “Printer” category
  - contains preferences used when Jalopy is run
  - to change, select project name and click “**Activate**” button
- New projects
  - click “Add…” button to create
  - remember to “**Activate**” it to specify preferences for it and use it!

Running Jalopy From Command-line

- Add bin\jalopy-console-version.jar to CLASSPATH
- Enter commands with the following format
  - `jalopy [`*`options`*`]` *`args`*
  - when running under Linux, it may be necessary to
    - change .sh script files to UNIX format
    - add getopt-*version*.jar to CLASSPATH

Common Jalopy Command-line Usages

- Run on a single source file and overwrite it

```
jalopy classname.java
```

- Run on a single source file and avoid overwriting

```
jalopy classname.java > classname.new
```

redirects output to a different file

- Run on all source files below current directory and overwrite

```
jalopy -r .
```

- Run on all source files below current directory and avoid overwriting

```
jalopy -r . -d output-directory
```

redirects output to a different directory

- Get help on all available options

```
jalopy -h
```

Running Jalopy From Ant

- To enable Ant to find the Jalopy JAR files
  - either copy them to the Ant lib directory
    or insert the following taskdef into Ant build file

```
<taskdef name="jalopy"
  classname="de.hunsicker.jalopy.plugin.ant.AntPlugin">
  <classpath>
    <fileset dir="jalopy-dir/bin">
      <include name="*.jar"/>
    </fileset>
  </classpath>
</taskdef>
```

Running Jalopy From Ant (Cont'd)

- Example Ant target
  - runs Jalopy on all Java source files in a project

```
<target name="format" depends="compile">
  <taskdef name="jalopy"
    classname="de.hunsicker.jalopy.plugin.ant.AntPlugin"/>
  <jalopy classpathref="classpath">
    <fileset dir="${src.dir}">
      <include name="**/*.java"/>
    </fileset>
  </jalopy>
</target>
```

depends on the compile target because formatting code that contains syntax errors may produce undesirable results

use this if previous taskdef is not used and Jalopy JAR files are in Ant lib directory

id of a path element that is also used to compile source code

depending on compile and specifying classpath are only necessary to take advantange of import optimization

- For details on other jalopy task attributes
  - see docs\plugin-ant-usage.html

CheckStyle Features

- Checks conformance of Java source code against a set of coding standards that are highly configurable
- GNU Lesser General Public Licence (LGPL)
- Highlights of what it can check
  - javadoc comments
    - reports classes, interfaces, fields and methods that don't have a javadoc comment
    - can specify minimum visibility scope that requires them (for example, protected)
    - default scope is private which requires javadoc on everything
  - naming conventions
    - reports names that don't conform to specified conventions
    - checks names of every package, class, interface, constant, static field, instance field, method, parameter and local variable
  - headers
    - reports source files that don't begin with a specified header which is typically used to provide copyright information

CheckStyle Features (Cont'd)

- imports
  - reports imports that aren't needed
  - reports imports with restricted package prefixes (defaults to "sun")
- size violations
  - reports lines that are too long (default limit is 80 characters)
  - reports methods with too many lines of code (default limit is 150)
  - reports constructors with too many lines of code (default limit is 150)
  - reports sources files with too many lines of code (default limit is 2000)
  - reports methods and constructors that take too many parameters (default limit is 7)
  - reports casts and commas not followed by a space
  - reports periods preceded or followed by a space
  - reports incorrect spacing around parentheses
  - reports incorrect line wrapping of expressions containing operators
- whitespace
  - reports files that contain tab characters
  - reports keywords not surrounded by spaces
    - `if`, `for`, `while`, `do`, `catch`, `synchronized` and `return`

CheckStyle Features (Cont'd)

- modifiers
  - reports wrong order of modifiers (`public`, `protected`, `private`, `abstract`, `static`, `final`, `transient`, `volatile`, `synchronized`, `native` and `strictfp`)
  - reports use of unnecessary public and abstract modifiers in interfaces
  - reports non-private fields
- blocks
  - reports missing braces
  - reports empty blocks (can require that they at least contain a comment)
  - reports non-conforming placement of left and right braces
- and more
  - reports comments containing "`TODO:`"

Installing CheckStyle

- Download from
  - http://checkstyle.sourceforge.net
- To install
  - unzip
  - add checkstyle-all-*version*.jar to CLASSPATH environment variable

Configuring CheckStyle

- To modify coding standards checked by CheckStyle
  - create a Java property file describing alternate settings
  - an example of a CheckStyle property is "`checkstyle.maxlinelen`" which defaults to 80
  - for details on specific properties that can be set, see docs\config.html
  - the property file to be used can be specified in a command-line option or in the "`properties`" attribute of the "`checkstyle`" Ant task

Running CheckStyle From Command-line

- Enter commands with one of the following formats

```
java com.puppycrawl.tools.checkstyle.Main
  [options] [source-files]
```

or

```
java -jar %CHECKSTYLE_HOME%\checkstyle-all-2.4.jar
  [options] [source-files]
```

  – difference between them is that the first requires CLASSPATH environment variable to be set and the second doesn't

CheckStyle Options

- `-f xml`
  - specifies that XML output should be generated instead of plain text which is the default
  - contrib directory contains XSLT stylesheets that can be used to transform the XML output into HTML
- `-p` *`property-file`*
  - specifies a property file which configures checks that will be performed
- `-o` *`file`*
  - specifies name of output file
  - if not used, output is written to stdout
- `-r` *`directory`*
  - specifies directory containing the source files to be checked
  - all .java files in and below that directory will be checked

Running CheckStyle From Ant

- To enable Ant to find the CheckStyle JAR file
  - either copy it to the Ant lib directory
    or insert the following taskdef into Ant build file

```
<taskdef name="checkstyle"
  classname="com.puppycrawl.tools.checkstyle.Main"
  classpath="checkstyle-dir/checkstyle-all-2.4.jar.jar"/>
```

Running CheckStyle From Ant (Cont'd)

- Example Ant target
  - runs CheckStyle on all Java source files in a project and produces an HTML report

```
<target name="check">
  <taskdef name="checkstyle"
    classname="com.puppycrawl.tools.checkstyle.CheckStyleTask"/>
  <checkstyle properties="checkstyle.properties"
    failOnViolation="false" failureProperty="check.failed">
    <fileset dir="${src.dir}" includes="**/*.java"/>
    <formatter type="xml" toFile="checkstyle.xml"/>
  </checkstyle>
  <xslt in="checkstyle.xml" out="checkstyle.html"
    style="checkstyle-dir/contrib/checkstyle-noframes.xsl"/>
</target>
```

use this if previous taskdef is not used and CheckStyle JAR files are in Ant lib directory

- Setting `failOnViolation` to false causes it to continue checking source files after one has failed to pass all the checks.
- `failureProperty` specifies an Ant property to set if one or more checks fail.
- This can be used in subsequent Ant targets to prevent them from running.

Summary

- The combination of these tools can go a long way toward allowing tolerance of personal coding styles while still delivering a consistent code base that conforms to project guidelines