Jalopy - User’s Guide v. 1.9.3

Jalopy - User's Guide v. 1.9.3

Contents

Acknowledgments

First and foremost we wish to thank the creators of the free software libraries we use. Jalopy includes source code and artwork developed by

- the ANTLR Project
- the Apache Software Foundation
- Ayman Al-Sairafi
- Bernhard Picher
- the Eclipse Project
- Dr. Heinz M. Kabutz
- the JDBM Project
- the JDOM Project
- Jean-Marie Dautelle
- Karsten Lentzsch
- the Object Web Consortium
- Simon Tuffs
- Sun Microsystems, Inc.

Please refer to Appendix A, *Library Dependencies* for a more detailed list and the individual licensing terms.

We would like to say a big thanks to those who contributed code during the Open Source days. Thanks also to all users and customers who provided feedback, submitted bug reports and suggested new features.

Introduction

Jalopy is an advanced source code formatter for the Sun Java™ Programming Language. It can automate all aspects of code layout, like indentation, aligning, line wrapping, brace styling, grouping and sorting. Without effort you can achieve a consistent coding style across your development team, or present your code in different shapes for purposes like code review or customer shipment.

Jalopy is written in Java™ and provides many Plug-ins to integrate the formatting engine into some of the most popular Java applications, including Ant, Eclipse, IntelliJ IDEA, Sun Java Studio Creator, Borland JBuilder, CodeGear JBuilder, Oracle JDeveloper, jEdit, NetBeans, Maven, IBM Rational Application Developer (RAD), Sun ONE Studio and IBM Websphere Application Developer (WSAD) and others, but can be used standalone as well.

> **How much is good layout worth?**
>
> Our studies support the claim that knowledge of programming plans and rules of programming discourse can have a significant impact on programming comprehension. In their book called *[The] Elements of [Programming] Style*, Kernighan and Plauger also identify what we would call discourse rules. Our empirical results put teeth into these rules: It is not merely a matter of aesthetics that programs should be written in a particular style. Rather there is a psychological basis for writing programs in a conventional manner: programmers have strong expectations that other programmers will follow these discourse rules. If the rules are violated, then the utility afforded by the expectations that programmers have built up over time is effectively nullified. The results from the experiments with novice and advanced student programmers and with professional programmers described in this paper provide clear support for this claim.
>
> —Elliot Soloway and Kate Ehrlich

What does it do for you?

- **Jalopy accurately represents the logical structure of your code**. That's the primary purpose of any source code formatting. Indentation, white space and line wrapping are used in a sensible way to show the structure of your code.
- **Jalopy consistently represents the logical structure of your code**. Nearly impossible to achieve manually. Scope levels are always correctly indented, braces and brackets are always found at the same places.
- **Jalopy improves the readability of your code**. Your code layout will always meet the expectations laid out by your team no matter when a file was written or by whom.
- **Jalopy makes your code withstand modifications**. A developer don't have to care anymore whether modifying one line of code may require modifying several others in order to achieve consistency.
- **Jalopy increases the productivity of your developers**. They can concentrate on the design and implementation issues, instead of spending time struggling with the code style. Developers may even write code in whatever style they prefer, formatting it before they submit files to the repository.

How can it be used?

- **Jalopy is especially well-suited for client-side use by developers**. It tightly integrates with all common Java IDE applications to serve as an integral part of any source code editing.
- **Jalopy can be added into your SCM**. Your sources are formatted on the server side before they are committed to the repository. This can currently be achieved using custom scripting with the Console Plug-in.
- **Jalopy supports usage as part of your build process**. Your sources may be formatted after every compilation, or on the checkout/checkin process. Jalopy already provides support for the well-known Ant and Maven build tools and can be easily integrated with many others.

Part I. Core

This part of the manual covers the core Jalopy engine: generic installation and usage instructions along with a detailed discussion of the available options to customize application behavior and formatting output.

Chapter 1. Installation

Describes the steps necessary to install a Jalopy release.

1.1 System requirements

Jalopy will run on any Pentium-class machine with a minimum of 256 MB RAM. You may succeed with less, but it's not recommended for a good user experience. Depending on your options, you need between 10-20 MB free disk space for the installation files. During runtime, additional space is required for settings, caches and backup, typically between 5-50 MB, but this again depends on your project size and setup.

The supported operating systems are: Linux, Mac OS X (x86 only), Solaris, Unix, Windows XP or later. Jalopy *should* run on all platforms that provide a suitable Java VM.

Jalopy requires a properly configured Java JDK version 1.4 or later on your system. We recommend to use a more recent version for best performance.

1.2 Prerequisites

Installation should be preferably performed below the user directory, when possible. If you should need to install into a different target directory, please make sure that the setup wizard is invoked with sufficient user privileges, because it may need to create directories below certain application folders that might not be accessible as normal user.

If you choose to install any of the provided IDE plug-ins, you must close any running target application(s) prior to installation. Otherwise setup may fail, because necessary files cannot be installed or obsolete files cannot be removed.

Download Online Help

During installation you will be asked whether you want to download and install the online help for the preferences dialog. If you're installing on a machine without Internet access, you can download the help file separately from http://www.triemax.com/download/jalopy-help-1.9.3.jar and either place it along the directory where the installer sits—it will then be picked up and installed automatically by the installer. Or copy the file into the Jalopy settings directory, e.g. on a typical Windows XP system to `C:\Documents and Settings\John Doo\.jalopy\1.9.3\jalopy-help-1.9.3.jar`. You can find more information about the Jalopy settings directory in Section 2.1, "Overview".

The setup wizard will install the online help always into the Jalopy settings directory as described above. But if you're doing a custom installation, because you want the software to be available for all your developers without requiring them to do any extra work on their client machines, you can place the file into the same directory where the binaries have been installed, e.g. if the software was installed into `C:\Program Files\jalopy`, the binaries can be found in `C:\Program Files\jalopy\lib`. Just copy the help file into this directory and it will be available to all users of the binary.

1.3 Wizard Installation

Jalopy comes as a compressed JAR Archive (JAR) that contains all necessary application files. The JAR is executable and provides a graphical setup wizard that lets you install the software in a few easy steps. If you're about to install Jalopy for the first time, wizard installation is highly recommended. When upgrading, it is usually much simpler to perform a silent install as described in Section 1.4, "Silent Installation".

WARNING Please note that there is a common bug with Microsoft Internet Explorer that sometimes renames the provided JAR file to one with the .zip extension. If the downloaded file ends with .zip, simply rename the extension to make the installer work.

Step 1: Startup

To start the setup wizard, you may open a shell and type

```
% java -jar jalopy-setup-1.9.3_156.jar
```

at the command line. But with modern Java virtual machines, it is usually possible to launch the installer by just double-clicking the downloaded JAR file from within your file manager. If your system should be not configured this way, you can always resort to the manual invocation as described above.

Step 2: Welcome

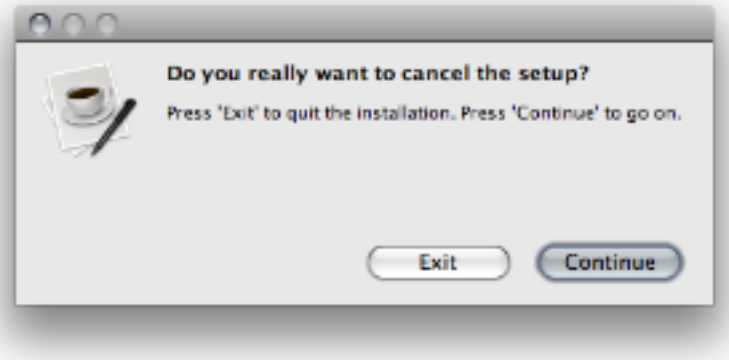
After you've invoked the setup wizard, the welcome screen will appear shortly.

Figure 1.1. Setup Wizard Welcome Screen

Each wizard page contains a button bar at the bottom that provides the available page actions. To proceed to the next page, click the *Next* button. When available, you can press the *Back* button to return to the previous page and alter your selections.

You can use the *Cancel* button at any time to abort the installation. A dialog will appear that asks for confirmation.

Figure 1.2. Cancel Setup Wizard

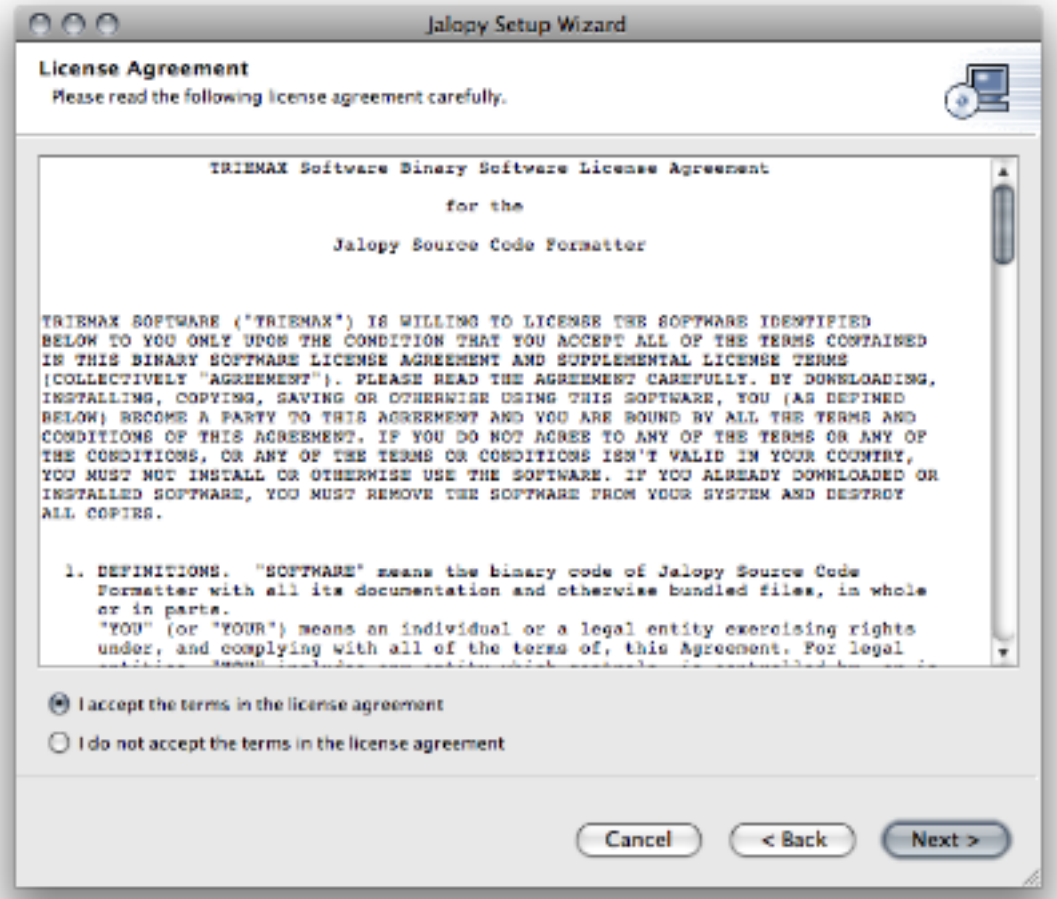


Press the *Exit* button to terminate the setup wizard. Press *Continue* to close the dialog and continue with the installation.

Step 3: License Agreement

Pressing the *Next* button on the Welcome Page will display the License Agreement.

Figure 1.3. Setup Wizard License Agreement

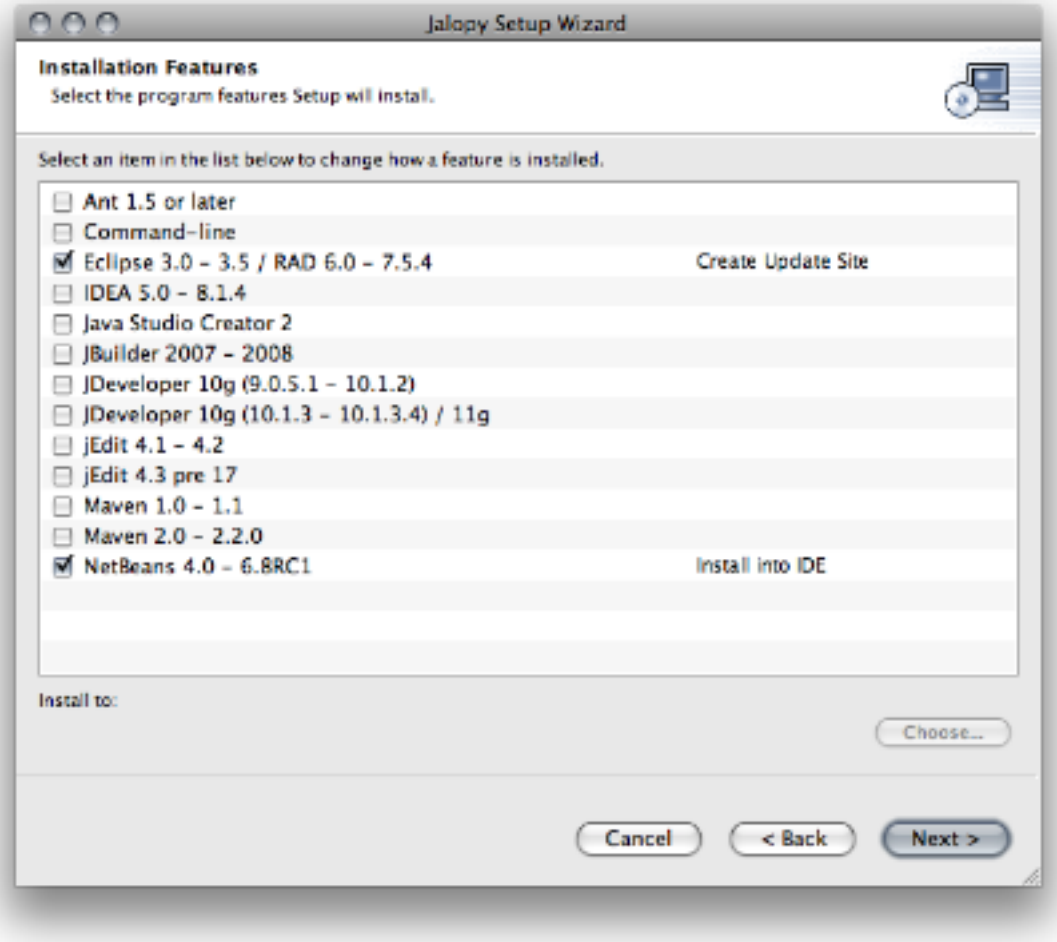


Read the terms carefully! You need to accept the license agreement before you can proceed with the installation. Select the *I accept the terms in the license agreement* item and press the *Next* button to proceed.

Step 4: Installation Features

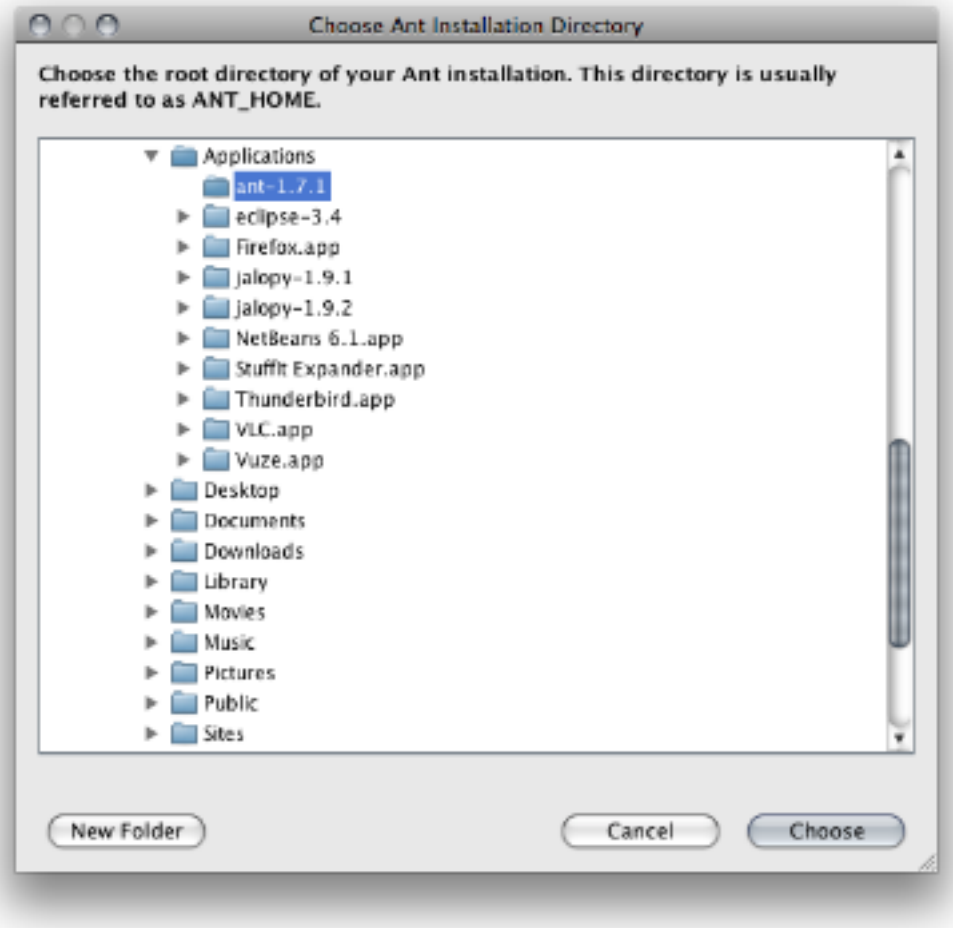
Pressing the *Next* button on the License Terms page will bring up the Installation Features page that lets you select what program features setup should install and how.

Figure 1.4. Setup Wizard Installation Features



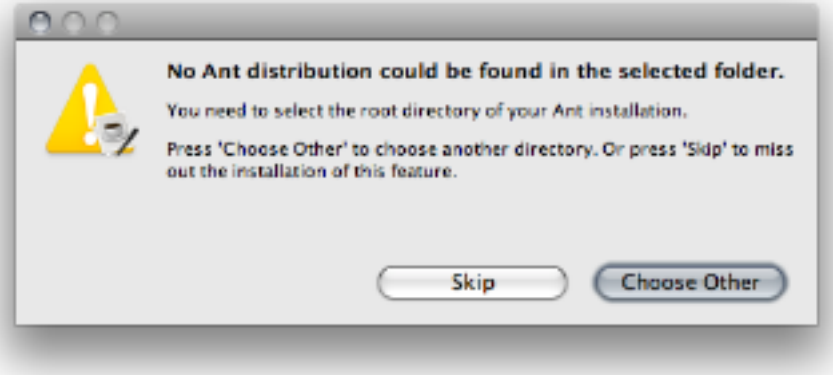
Select any of the check boxes to mark a feature for installation. Should you ever run the installer again, your choices from the last session are remembered and the screen configured accordingly. When you first select a feature, a file chooser pops up that lets you select the target directory for the specific application. This is usually the root installation or settings directory of the application.

Figure 1.5. Choose Ant Installation Directory



The wizard verifies your selection and ensures valid target directories. If validation should fail, a dialog pops up to inform you about the situation.

Figure 1.6. Installation Directory Verification Failed



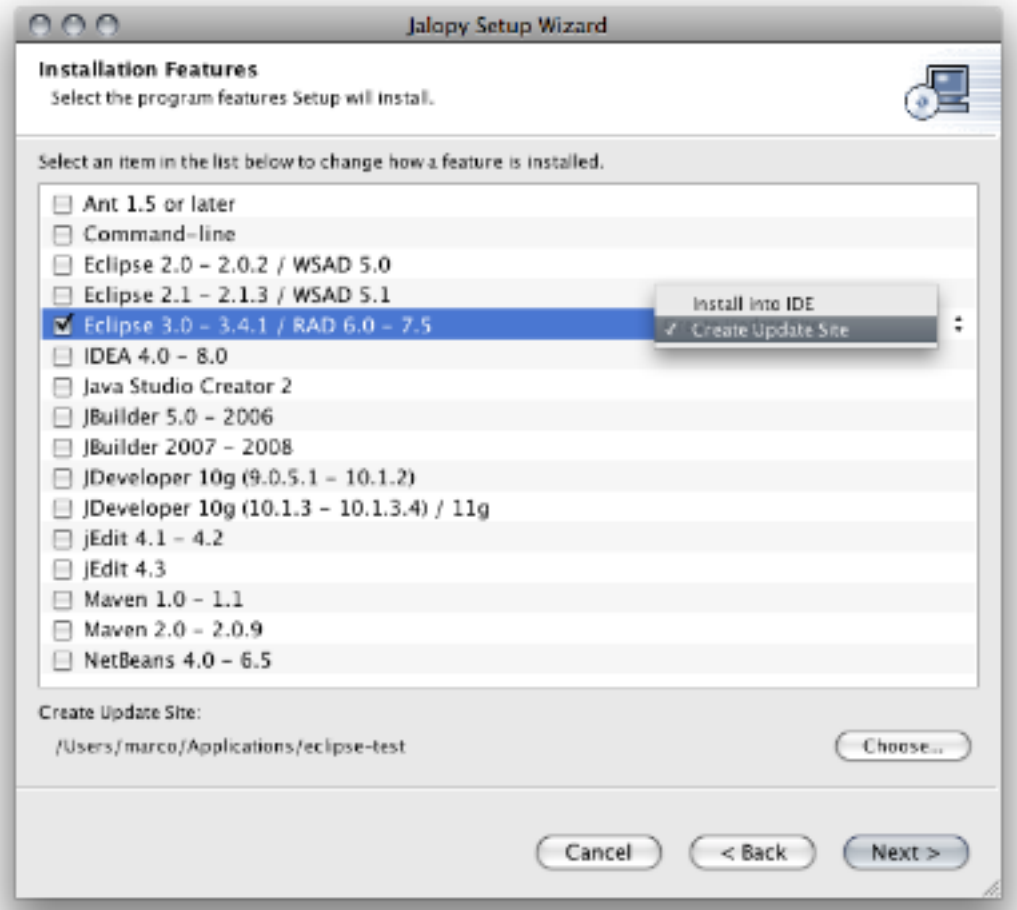
Press the *Choose Other* button if you want to select a different directory. Otherwise, if you want to leave out the feature, you can press the *Skip* button to end the directory chooser and return to the feature selection screen (or the next directory chooser dialog).

Once a target directory has been set for a feature, it will be displayed below the feature list. You can change the target directory for a feature at any time. First select the corresponding item in the list and then use the *Choose...* button at the bottom to specify the

target directory. Please note that when multiple check boxes are selected, multiple directory choosers will appear one after another. Don't be confused, just look at the title of each file chooser to see what application directory is required.

Because it can be more convenient to leverage the plug-in manager facility of the IDE, for some plug-ins you can create an IDE specific bundle that can later be installed using the IDE specific provisions. To change the installation target, click the current target item behind the installation feature and choose one of the available options.

Figure 1.7. Choose Eclipse Installation Target

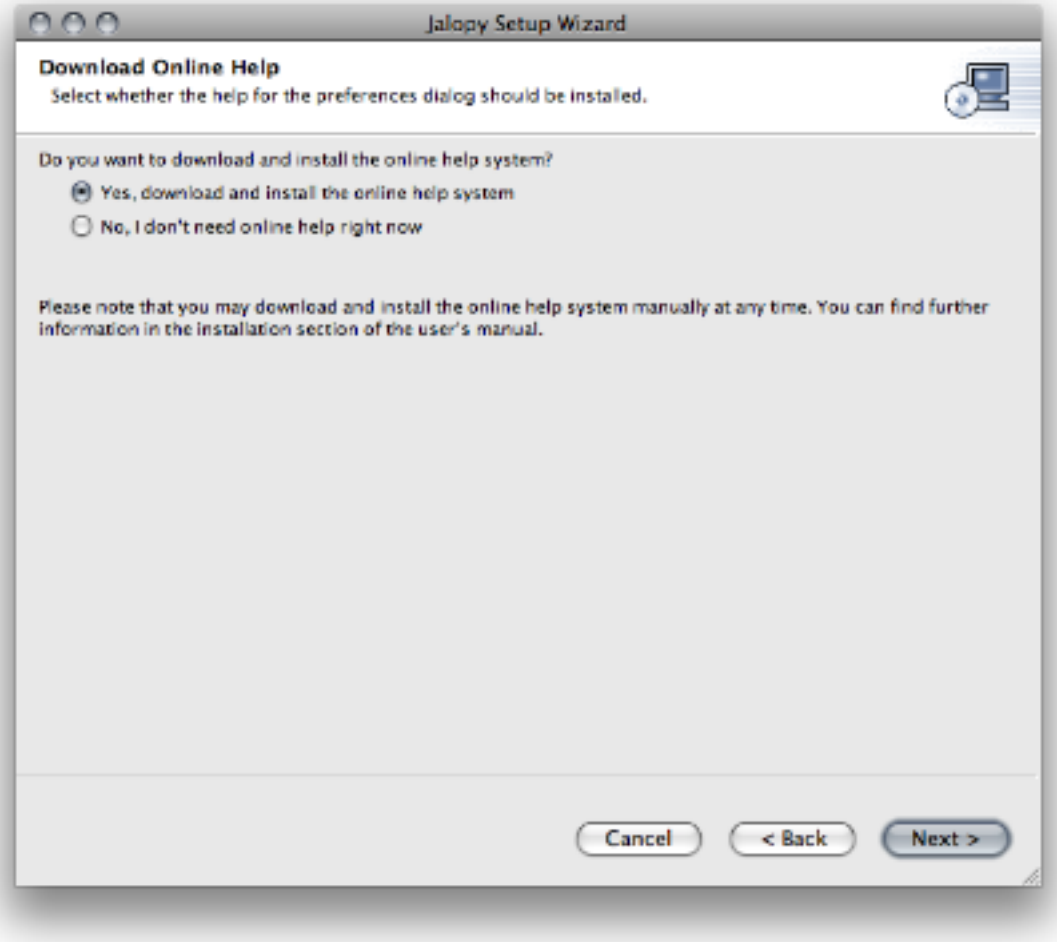


When the installation target has been set to the application specific update manager format, the installation will create an achive bundle in the specified directory, but the plug-in itself won't be installed into the target application. You will need to use the IDE update manager to perform plug-in installation after the setup wizard has finished.

Step 5: Online Help System (optional)

When the installer can connect with the Internet, and no up-to-date help can be found on your system, you will be presented with the option to download and install the online help system for the GUI.

Figure 1.8. Setup Wizard Download Online Help



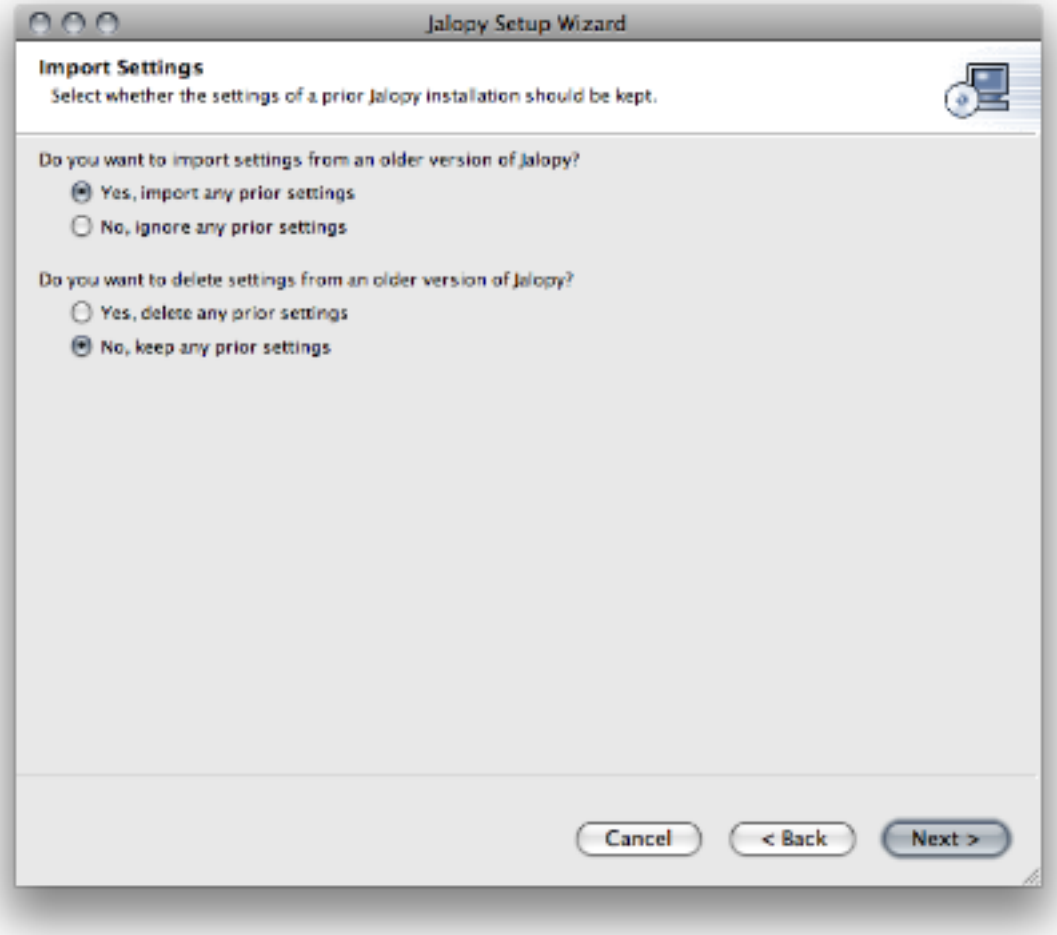
It is recommended to let the installer handle help installation, but when you're installing on a machine without Internet access, you can download the help file separately from http://www.triemax.com/download/jalopy-help-1.9.3.jar and either place it along the directory where the installer sits - it will then be picked up and installed automatically by the installer (the Download Online Help screen does not appear in such a case).

Or copy the file into the Jalopy settings directory, e.g. on a typical Windows XP system to `C:\Documents and Settings\John Doo\.jalopy\1.9.3\jalopy-help-1.9.3.jar`.

Step 6: Settings Import (optional)

In case an older Jalopy release could be found on your machine, the wizard lets you choose whether the settings of the prior version should be imported during installation.

Figure 1.9. Setup Wizard Import Settings



Select the *Yes, import my settings* option to have your settings imported or *No, ignore any prior settings* to start with the defaults. Additionally, you can control whether your prior settings should be deleted or kept. Select *Yes, delete any prior settings* to delete your existing settings. Or choose *No, keep any prior settings* to leave any present settings untouched.

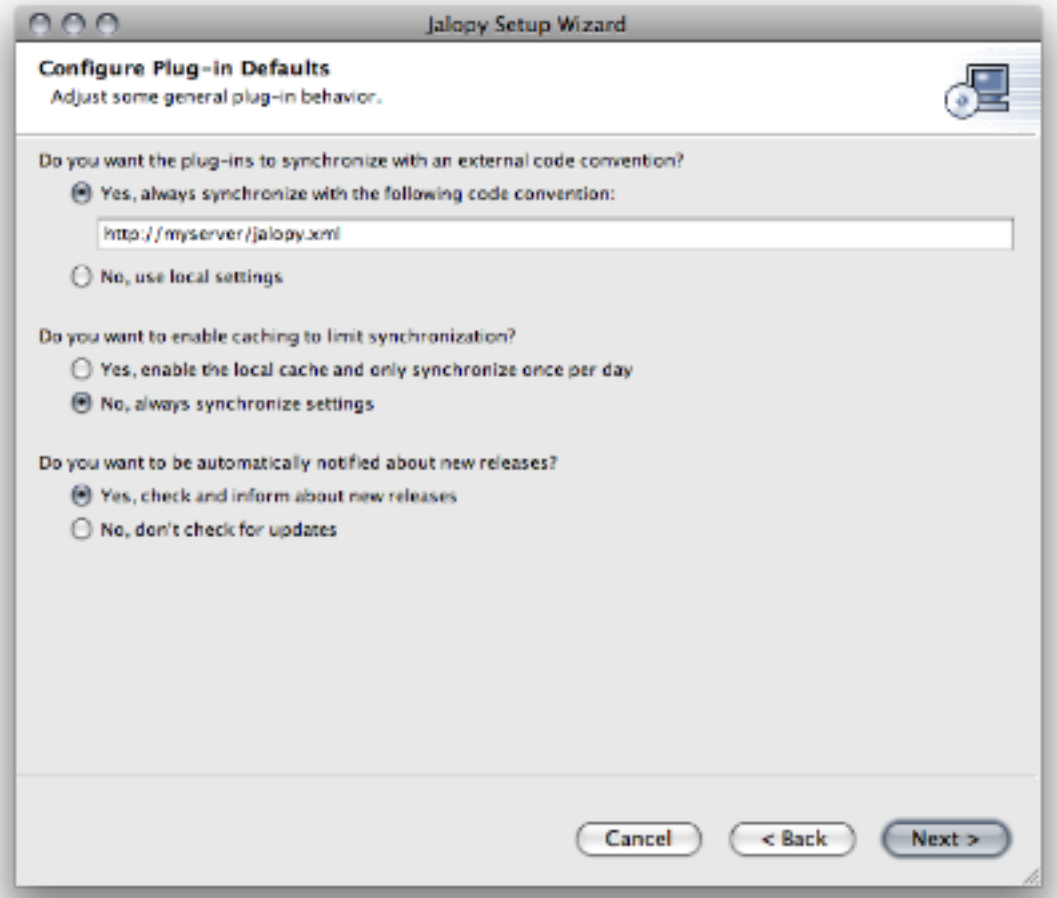
IMPORTANT Please note that the settings of the Jalopy Open Source versions up to 1.0b11 are always removed during installation. If you want to keep such settings, make a backup of your settings directory before you start the setup routine. Detailed information about the Jalopy settings system can be found in Chapter 2, *Configuration*.

Step 7: Configure Plug-in Defaults

The installer lets you pre-configure some IDE plug-in preferences, to possibly eliminate the need for client configuration.

NOTE The specified defaults only apply for the Eclipse, IDEA, JBuilder, JDeveloper, jEdit and NetBeans plug-ins. You still have to configure the headless plug-ins as usual, which is more appropriate for their use cases.

Figure 1.10. Plug-in Defaults Screen



When using Jalopy in a team, it is often mandatory to share a common code convention to achieve a consistent code layout style. There are multiple ways to achieve this goal, but the best and most convenient approach is to embedd this information right into the IDE plug-ins. This way developers must not know nor care how to configure Jalopy—it will automatically pick up its settings upon first installation. You can of course adjust the defaults later using the IDE preferences dialog.

To specify a shared code convention, select the *Yes, always synchronize with the following code convention* radio button and enter the path of the shared code convention. This can either be a file system path or a web url.

By default, Jalopy checks the specified settings file for changes each time it is about to format. This can be prohibitive when the code convention sits on a server without fast network access. To avoid long delays in such situations, you can enable local caching. Jalopy then only checks once per day for changes. To enable local caching, select the *Yes, enable the local cache and only synchronize once per day* radio button.

In order to keep keep track of updates, the IDE plug-in can notify about the avalailibilty of new releases. When a new release becomes available, they display a notification dialog that provides access to the release notes. If you don't want to be notified about updates, select the *No, don't check for updates* radio button.

Step 8: Confirmation

When all configuration is done, the installation summary dialog is displayed. Please review your choices and press the *Install* button to start the installation.

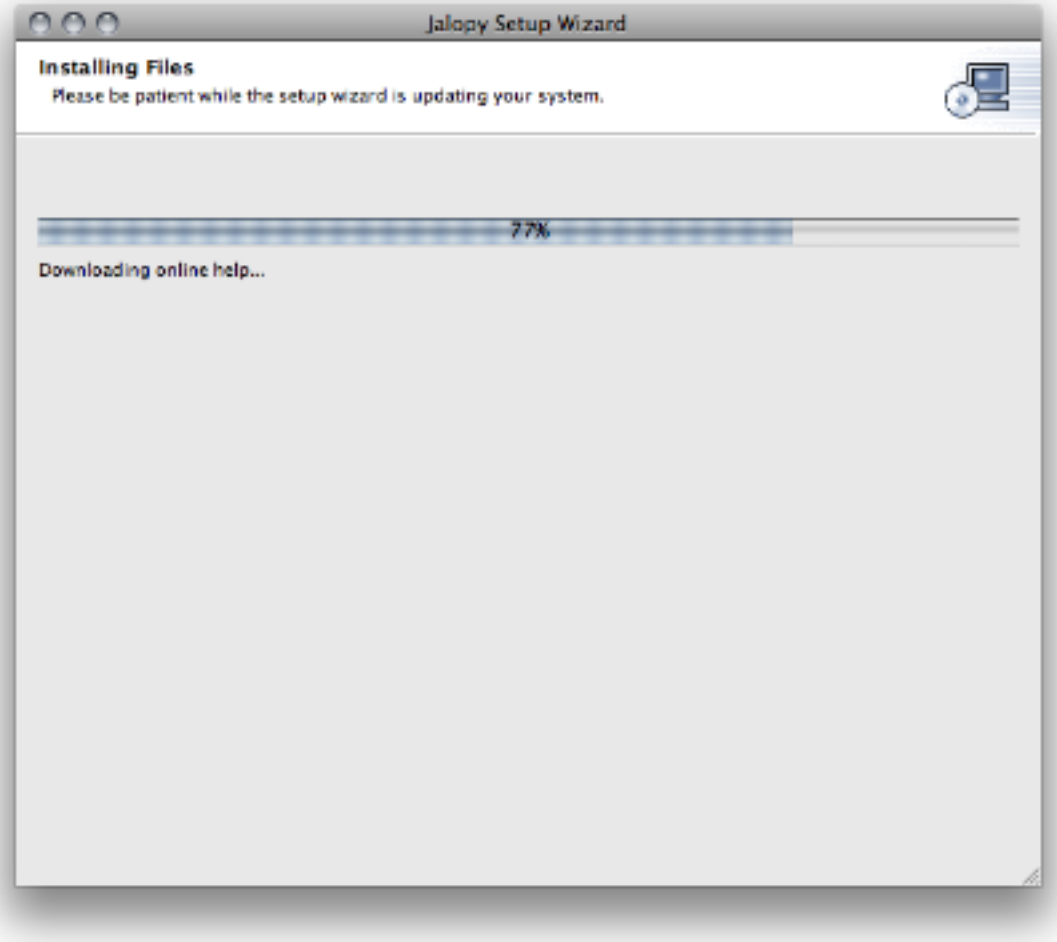
Figure 1.11. Ready Screen

IMPORTANT Please note that installation cannot be canceled once started. You should make sure that you've selected all desired features and configured the target locations correctly before you start the installation. You can of course, re-run the installer at any time in case you need to perform installation with different settings. If you think that you've made a wrong choice during the information gathering, press the *Back* button to flip through the pages and review your settings.

Step 9: Installation

Once the insallation has started, a progress dialog informs you about the pending installation steps.

Figure 1.12. Progress Screen



The installation process might take a while, please be patient until the installer has finished updating your system. Especially the installation of the Eclipse plug-in can be very time consuming on big installations.

Step 9: Results

When installation has been finished, the finish screen appears.

Figure 1.13. Finish Screen

You can press the *Show Report* to review the installation log. In case something should have went wrong during the installation, please provide support with this log information. Press the *Finish* button to close the setup wizard.

1.4 Silent Installation

The executable JAR file contains built-in support for silent (unattended) installation. A normal wizard install guides the user through different graphical dialog boxes and expects some input. However, a silent install does not prompt the user for input. Instead it receives the required setup data from a configuration driver file that provides the information the user would otherwise enter as responses to dialog boxes.

The setup configuration driver file uses the standard java.util.Properties format. It consists of key/value pairs representing the data entries. Performing a wizard install automatically creates (or updates) a configuration driver file in the Jalopy settings directory that reflects the information given during the last setup session.

To perform a silent install, open a shell and type

```
% java -jar jalopy-setup-1.9.3_156.jar --silent
```

at the command line. This will perform installation with the data gathered from your last installation session. But the installer supports a few more options to control the setup process. These are described below.

Options

Table 1.1. Install Wizard command-line options

Option	Long Option	Arguments	Description	Since
-c	--config	<filepath>	Specifies the absolute path to the setup configuration driver file to use for the installation, e.g. `/home/John Doo/tools/jalopy-install.ini`. When omitted, the settings of the last installation run will be used when available	1.4
-h	--help		Displays a short help	1.4
-l	--log		Specifies the directory where the log file should be written. When omitted, the log file is stored in the Jalopy settings directory (Section 2.1, "Overview")	1.4
-s	--silent		Performs silent install	1.4
	--update-center		Creates a NetBeans update center	1.9.2
	--update-site		Creates an Eclipse update site	1.9.2

Example configuration driver file

Below you find an annotated sample configuration driver file, that explains all available keys and the possible values.

Example 1.1. Sample configuration driver file

```
#Jalopy installation data
#Fri Dec 03 09:14:37 CET 2004

delete.settings=false
import.settings=false
download.help=true

install.ant=true
install.ant.dir=/Home/John Doo/Applications/ant-1.7.1

install.console=true
install.console.dir=/Home/John Doo/Applications/jalopy

install.eclipse=true
install.eclipse.dir=/Home/John Doo/Applications/eclipse-3.4.2

install.idea=true
install.idea.dir=/Home/John Doo/Applications/idea-8.0

install.jdeveloper=true
install.jdeveloper.dir=/Home/John Doo/Applications/jdeveloper-11g

install.jedit=true
install.jedit.dir=/Home/John Doo/Applications/jedit-4.3

install.maven=true
install.maven.dir=/Home/John Doo/.m2

install.netbeans=true
install.netbeans.dir=/Home/John Doo/Applications/netbeans-6.5
```

1 Header comment that contains the last modification date of the file

2 delete.settings = true|false

Indicates wether the settings of a prior version should be removed.

"false" to keep prior settings, "true" to remove them

3 import.settings = true|false

Indicates wether the settings of a prior version should be imported.

"false" to ignore prior settings, "true" to import them

4 install.[appKey] = true|false

Indicates wether the specified application plug-in should be installed.

"false" means that the plug-in won't be installed, "true" installs the plug-in. The valid appKeys are ant, console, eclipse, idea, idea3.x, jbuilder, jdeveloper, jbuilder, jedit, netbeans, netbeans3.4.

5 install.[appKey].dir = absolute file path

Specifies the absolute file path of the root application directory. The file path is stored in platform notation.

1.5 Manual Installation

It is often possible to install Jalopy manually yoursef, but this might require certain non-trivial tasks, especially for the IDE plug-ins. It is therefore recommended to at least initially use the installer to perform installation on a test system and extract the necessary information for your custom deployment procedure. Please contact support if you need any specific assistance.

IMPORTANT Wizard installation is mandatory with the trial version

Chapter 2. Configuration

Provides a detailed discussion of the Jalopy settings system and all available options to configure formatting output.

2.1 Overview

Jalopy stores all settings below its own settings directory. This directory is normally located under the user home directory and shared by all provided Plug-ins. The table below shows the typical locations for the common operating systems.

Table 2.1. Typical settings directories for user "John Doo"

Operating System	Jalopy Settings Directory
Linux	/home/John Doo/.jalopy/
Mac OS X	/Users/John Doo/.jalopy/
Solaris	/export/home/John Doo/.jalopy/
Windows Vista	C:\Users\John Doo\AppData\Roaming\.jalopy\
Windows XP	C:\Documents and Settings\John Doo\.jalopy\

Substitute "John Doo" with your user name. Please consult your operating s ystem documentation if your system uses different paths for the user directories.

In order to provide version interoperability between releases, the settings of each release are stored in subdirectories named after the version number of an individual release, e.g. `C:\Documents and Settings\John Doo\.jalopy\1.9.3\`. Each settings configuration uses a distinct folder, e.g. the default settings for user John Doo (on Windows XP) are stored in `C:\Documents and Settings\John Doo\.jalopy\1.9.3\default\`.

A settings configuration is called a *profile* and stores the actual code convention as well as user-specific data like file and dialog histories. See Section 2.1.1.1, "Main window" for more information. Please note that you can always use settings of prior versions with the most recent release, but it is generally not recommended nor supported to try vice-versa, as there is no guarantee that it will work this way round. Wizard installation will let you update your settings automatically when upgrading, see Section 1.3, "Wizard Installation". Code convention related settings are usually shared using a textual XML format, see Section 2.1.1.8, "Export code convention" for more information.

If need be, you can reconfigure the root directory to your own liking by pointing the Java system property "triemax.jalopy.home" to the folder name where settings should be stored. The Java launcher provides the standard -D option to define system properties. If the path is a string that contains spaces, you must enclose it with double quotes:

```
% java -Dtriemax.jalopy.home="/Users/John/Library/Application Support/Jalopy" [...]
```

When using an IDE or build tool, you might be required to use a different mechanism to define system properties. Please refer to the user documentation of the tool vendor for specific instructions.

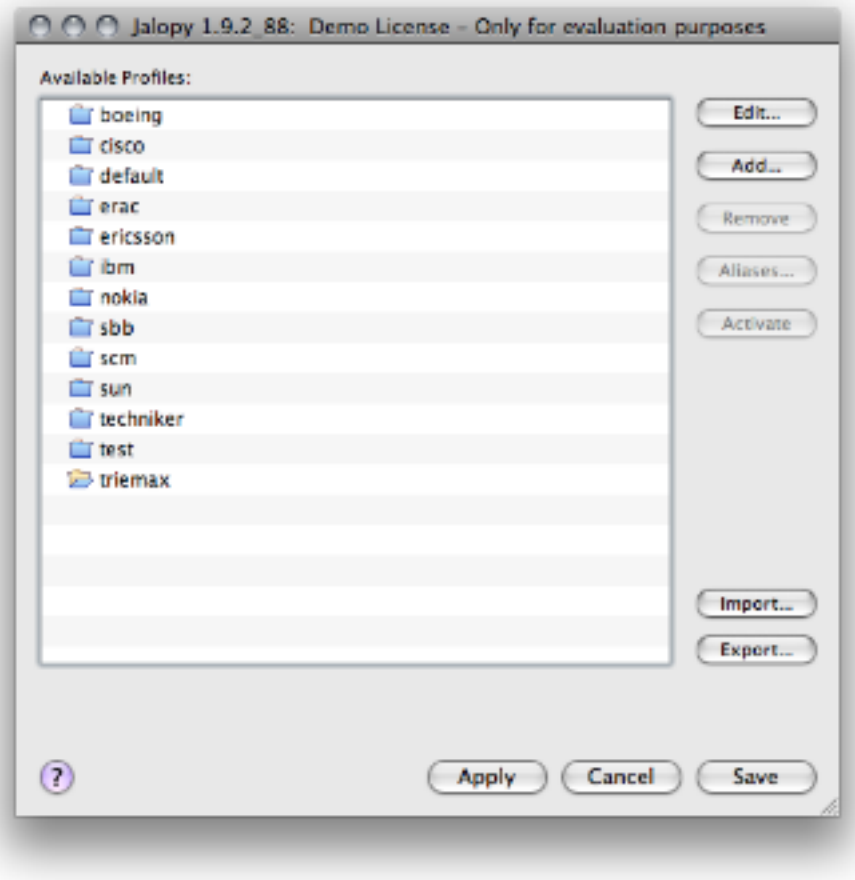
2.1.1 Preferences GUI

Settings are stored in binary files, that are not directly editable. Instead, a graphical user interface with a preview facility is provided to let you easily configure the settings. The GUI consists of several individual windows that can be freely arranged on your desktop. The Main window provides profile management and is the only window that appears after program start. From there you can access all other windows, namely the Configuration window to edit all formatting options, the Preview window that gives you an immediate layout preview reflecting the current settings, and the Help window that assists you at any time with complete documentation. The GUI may be either invoked directly on the command-line or from within your IDE. Please refer to the individual Plug-in chapters in Part II, "Plug-ins" for information on how to display it from the application you're using.

2.1.1.1 Main window

The Main window is the first window that appears and provides the means to manage several code convention profiles. Please note that when using one of the IDE Plug-ins, the contents of the Main window will be integrated into the IDE preferences dialog and therefore the appearance somewhat differs from IDE to IDE, but the functionality explained below is always present.

Figure 2.1. Main window



As explained in Section 2.1, "Overview", Jalopy stores code convention settings in profiles. The list component displays all currently known profiles. Click an entry to see what actions are available. Depending on the state and type of a profile, not all actions might be available all the time.

In Figure 2.1, “Main window” above, the active profile is selected and therefore the removal and activation buttons are deactivated. Editing, adding, importing and exporting is always possible.

The Main window always appears centered on the same monitor where it was invoked from.

Button bar

The Main window provides a button bar at the bottom that lets you perform different actions.

Help

The *Help* button displays the online help window. The keyboard shortcut for this action is *F1*. Please note that the help button is only available if the online help has been installed as outlined in the “Installation instructions”.

Save

The *Save* button lets you persist any unsaved changes made during a configuration session and closes the Main window.

Cancel

The *Cancel* button closes the Main window but any unsaved settings changes made during a configuration session are ignored.

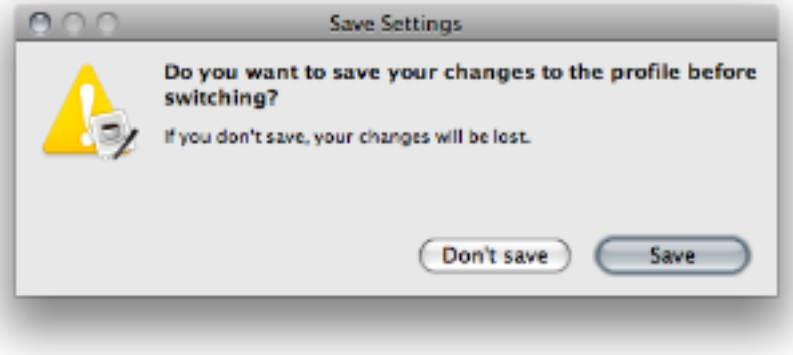
Apply

The *Apply* button persists any unsaved changes made during a configuration session.

2.1.1.2 Editing profiles

To edit an existing profile, select the profile in the list and press the *Edit...* button. If the selected profile is not the currently active one, the selected profile will be automatically activated. If the settings of the priorly activated profile have been altered, you will be asked whether you want to have your changes persisted before switching.

Figure 2.2. Save Profile Changes



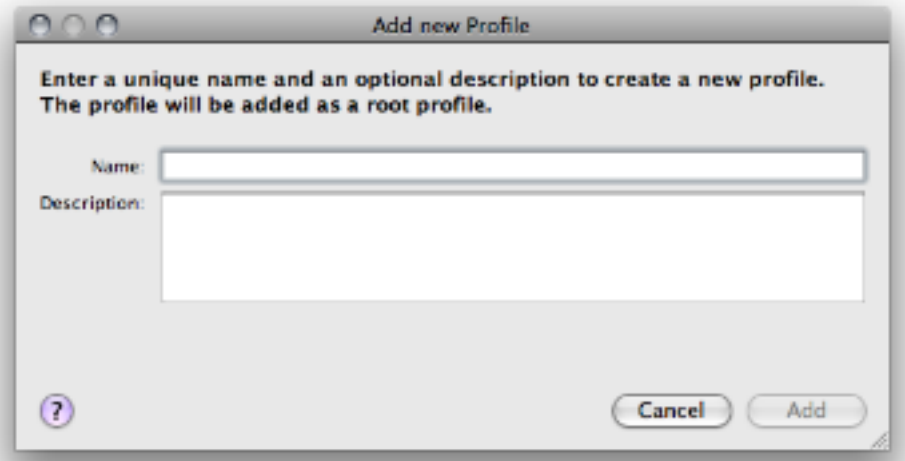
Press the *Save* button, if your settings should be saved. Otherwise press *Don’t save* to ignore any chances that were made to the profile. The Configuration window appears along with the Preview window, and here you can alter all available options to configure formatting

output. Any changes you make are directly reflected in the Preview window, but you must explicitly save your changes in the Main window. The Configuration window is explained in detail in the section called “Configuration window” below.

2.1.1.3 Adding profiles

To add a new profile click the *Add...* button. A dialog will appear that lets you create the new profile.

Figure 2.3. Add new Profile Dialog



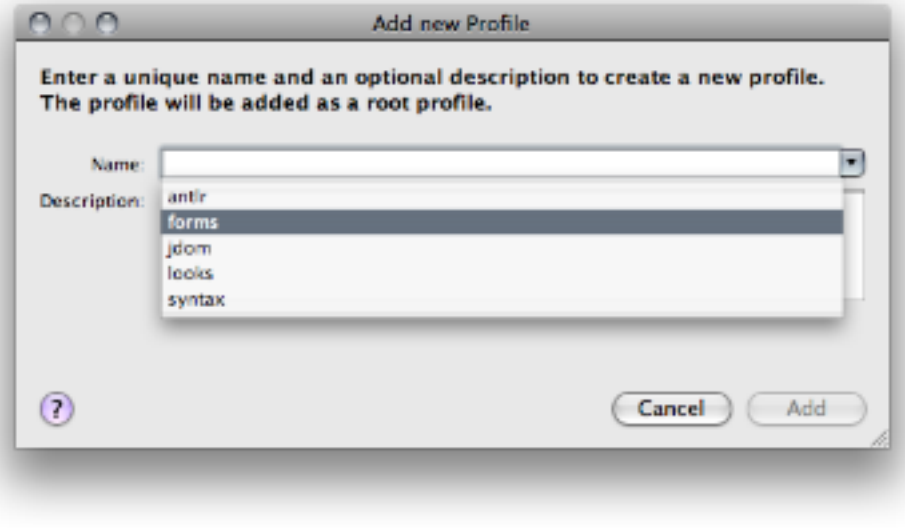
You need to enter the profile name, might add an optional informative description and if a profile is currently selected in the profiles view, you can choose whether you want to create a nested profile by selecting the parent profile. A nested profile will automatically adapt any changes made to its parent profile(s).

Name

The profile name needs to be unique and will be used as the name of the disk folder where all profile information will be stored. Therefore, you should avoid characters that your platform does not allow to be used in file paths.

As a convenience, when invoked from within one of the supported IDEs, the dialog provides a combo box with the names of all projects currently available in the IDE that have no corresponding Jalopy profile.

Figure 2.4. Add new Profile from within IDE Plug-in



Description

The optional description must be no longer than 256 characters and can be freely chosen. It will be displayed in a tool tip when the mouse is moved over an entry. Name and description are available for inclusion in templates. See Section 2.4.3, "Local variables" for more information.

Parent Profile

In order to provide the ability to easily manage profiles that largely share the same settings, you can create *nested profiles*. A nested profile will automatically adapt any changes made to its parent profile(s). This feature is only available with Jalopy 1.7 or later.

The typical example would be a number of different projects that should receive the same formatting style, but require different headers. In order to setup such a scenario, you would create a master profile where you configure all shared settings, and afterwards create different nested profiles for each project where you define the individual headers. Later on, if you want to apply any changes to the formatting style of all projects, you would only alter the master profile and the changes will be propagated to the nested profiles automatically.

In order to create a nested profile, simply choose the parent profile here. If you choose "None", the new profile will be created as a root profile. Please note that this option is only available when a profile is currently selected in the list view. When you add a new profile, the settings of the currently selected profile will be used to create the new profile. If you create a nested profile, the selected profile will be the parent profile. For every profile you define, a new subdirectory is created below the main settings directory where all related files will be stored.

2.1.1.4 Removing profiles

To remove an existing profile, select an entry or multiple entries in the list and press the *Remove* button. The profile folders will be removed on disk and the selected entries disappear. A profile may only be removed if it is not active. The default profile cannot be removed. Please note that if a selected profile contains any nested profiles, removing the profile will cause all nested profiles to be removed as well!

2.1.1.5 Activating profiles

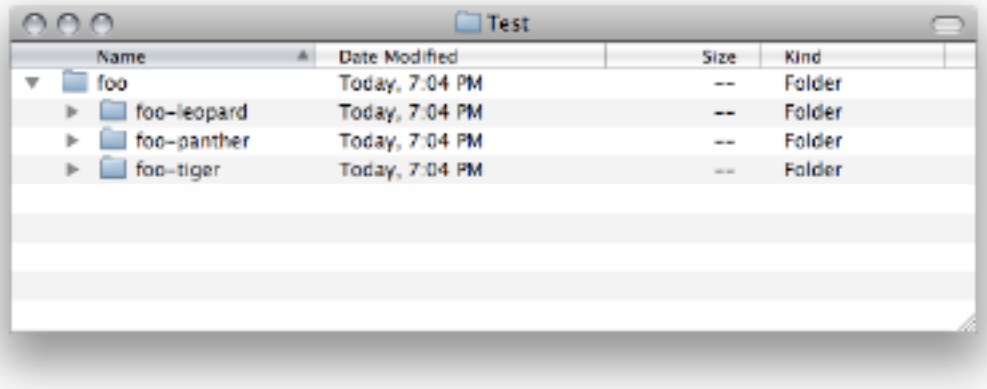
To activate an existing profile, select an entry in the list and press the *Activate* button. The stored settings will become active and the preferences dialog updated accordingly.

2.1.1.6 Defining aliases

Depending on the type and size of your projects and the provisions of your IDE, it might be necessary to create several project modules in order to manage your codebase efficiently. In such a case all related modules should still receive the same code style.

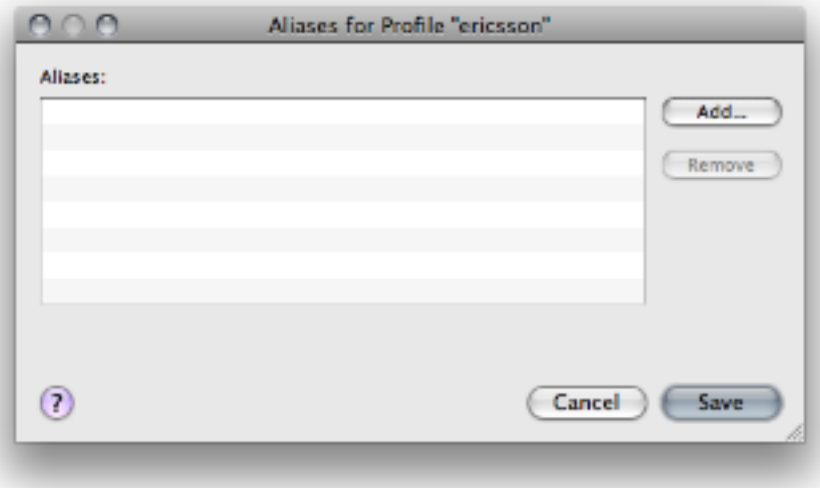
To achieve and manage such a uniform style easily, you can map modules to one (logical) Jalopy profile that defines the code style. Make sure that the Auto-switch feature (see below) is enabled and Jalopy will automatically use the correct settings for each module. Say you have a project “foo” which consists of three modules.

Figure 2.5. Sample Project With 3 Modules



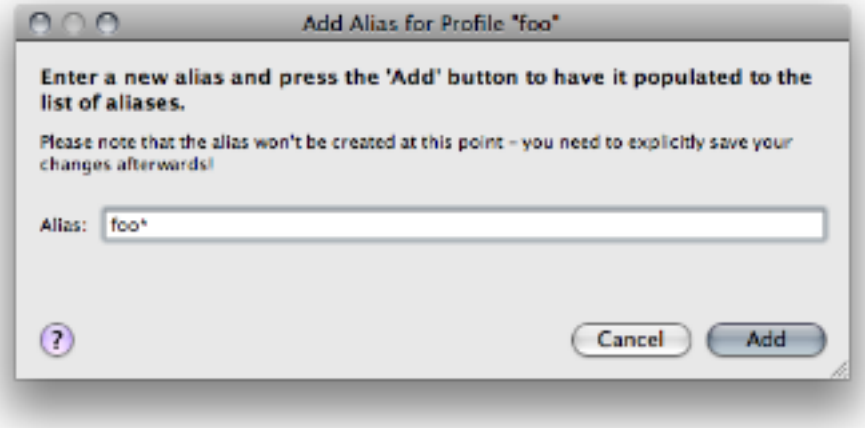
To map these modules to one Jalopy profile, choose the target profile in the list and press the *Aliases...* button. A dialog will be displayed that shows all defined aliases for the profile and lets you alter the alias definitions.

Figure 2.6. Profile Aliases Dialog



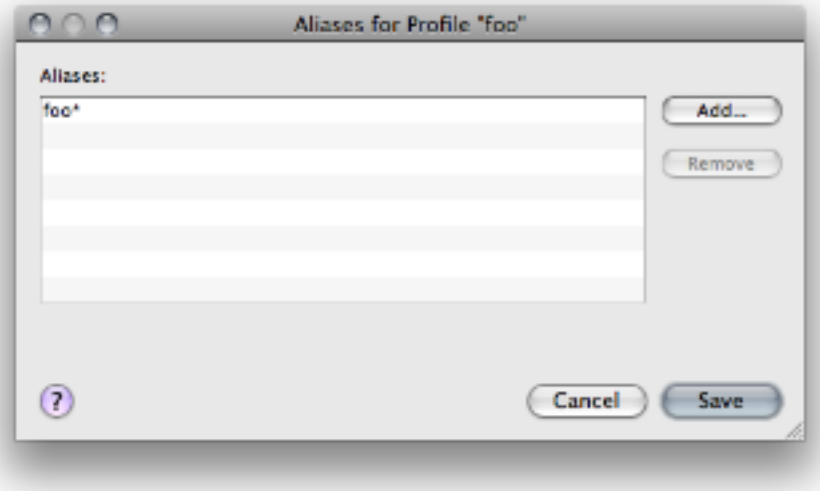
Press the *Add...* button to add a new alias for the profile.

Figure 2.7. Add new Profile Alias



You can either add the names of all modules as new aliases or when the modules share a common prefix—like in our example—use a wildcard alias to point to all modules in just one step. Simply put the * wildcard after the prefix and press the *Add* button. The alias is then displayed on the list, but has not yet been created. You need to explicitly press the *Save* button to have your changes applied and any new aliases created or existing aliases removed.

Figure 2.8. Profile Aliases



Press the *Save* button to save your changes or *Cancel* if you want to ignore any changes. Please note that the alias information of a profile is displayed as part of the tooltip (in square brackets). Move the mouse over a list entry, and the tooltip will appear shortly.

Since 1.2

2.1.1.7 Import code convention

Use the *Import...* button to import an already-saved code convention. Since version 1.6, Jalopy also supports importing of Checkstyle configurations (version 3.5 or later). Jalopy is able to import code conventions from both local and distributed locations. Just specify a valid Internet address (either starting with `http://`, `https://` or `www.`) for the latter.

Since Jalopy 1.7, exported code conventions store the names of their profile. During import it is therefore possible to recreate the profile structure. When importing a single

profile and the original profile does not already exist, you will be asked whether it should be created and the exported settings imported into this profile. Otherwise, the settings will be imported into the currently active profile. Checkstyle configurations will always be imported into the currently active profile. Importing a file that contains several code conventions, will always recreate the original profiles if they should not already exist.

Versions prior to 1.0b8 stored the backup directory always as an absolute file. Therefore after importing a very old code convention, you should check whether this directory points to your preferred backup directory. This advice holds true even for later versions in case you've changed any of the default directories (backup, history, message log).

Import Checkstyle configurations

Importing Checkstyle configurations only means a best effort. There is no guarantee that the resulting Jalopy code convention exactly matches your style preferences because some Checkstyle modules might be ambiguous or missing at all. E.g. take the following `ParenPad` module configuration:

```
<module name="ParenPad">
    <property name="tokens" value="METHOD_CALL" />
    <property name="option" value="nospace" />
</module>
```

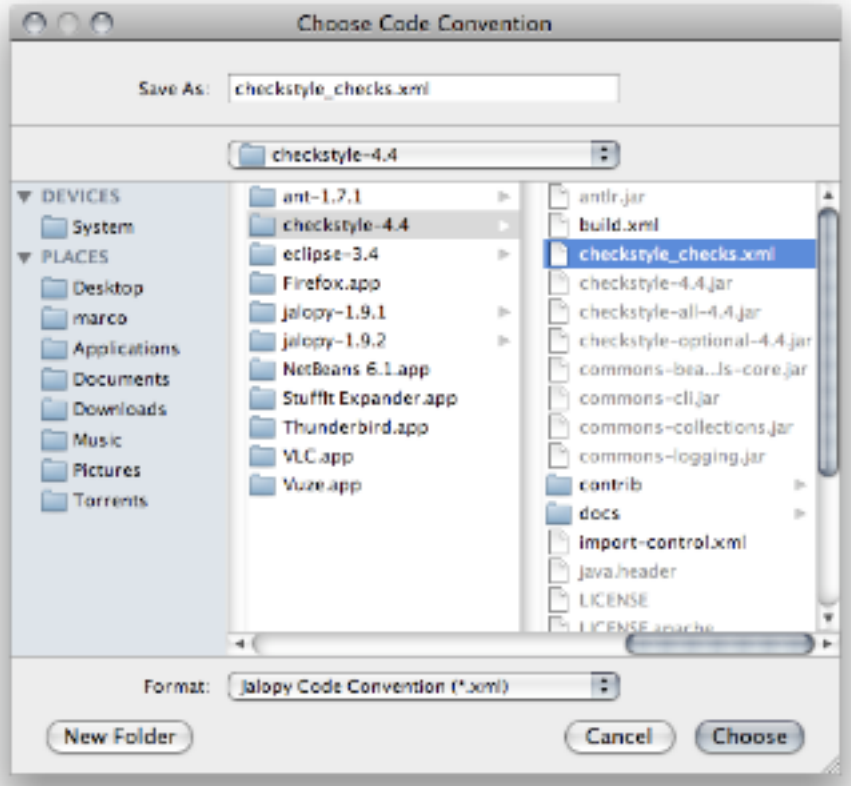
It only defines a white space check for method call parentheses, but does not express the preference for other parentheses. It could be "space" or "nospace". In such a case, Jalopy will assume the default value Checkstyle uses when no token is defined ("nospace" in the example).

Another common case is the <whitespaceAfter> module. Without any tokens defined, it will check for white space after three tokens (comma, semi, type cast parenthesis). But what if you limit the check to only two tokens (comma, semi)? Does it mean that no white space should appear after the right parenthesis of type casts? Or should it be allowed? Checkstyle accepts both, but Jalopy will assume that you don't want white space after the token in such a case.

The same problem appears when a Checkstyle module is not contained in your configuration. Jalopy can't interfere any preferences in such a case and assumes the default settings of an empty module config. In general, importing works better the more Checkstyle modules are defined. It is recommended that you test the resulting Jalopy code convention against the Checkstyle configuration after importing. Just format some source files into a temporary directory and run Checkstyle to check for any style violations. This way you can be sure that the import covered all your preferences.

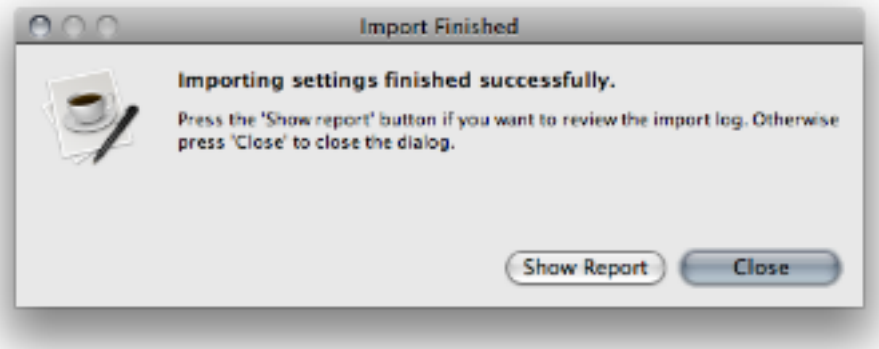
To import a Checkstyle configuration you need to press the *Import...* button and enter or select the configuration file that should be imported. The file dialog provides a file filter for Checkstyle configurations, but because all configuration files use the `.xml` extension, it actually doesn't matter what file filter is selected.

Figure 2.9. Choose Checkstyle Configuration



After the import has finished, a confirmation dialog appears that lets you display a report of the imported modules.

Figure 2.10. Import Checkstyle Configuration Confirmation



2.1.1.8 Export code convention

Use the *Export...* button to save your settings as a new code convention. Select the profiles that should be exported in the list, and press the *Export...* button to choose a file to export to. You may select multiple profiles that should be exported into just one file.

Please note that if a nested profile is selected, by default all parent and child profiles will be exported as well. If you really only want to export the profiles that have been selected in the list, hold down the *Ctrl* key when pressing the *Export...* button.

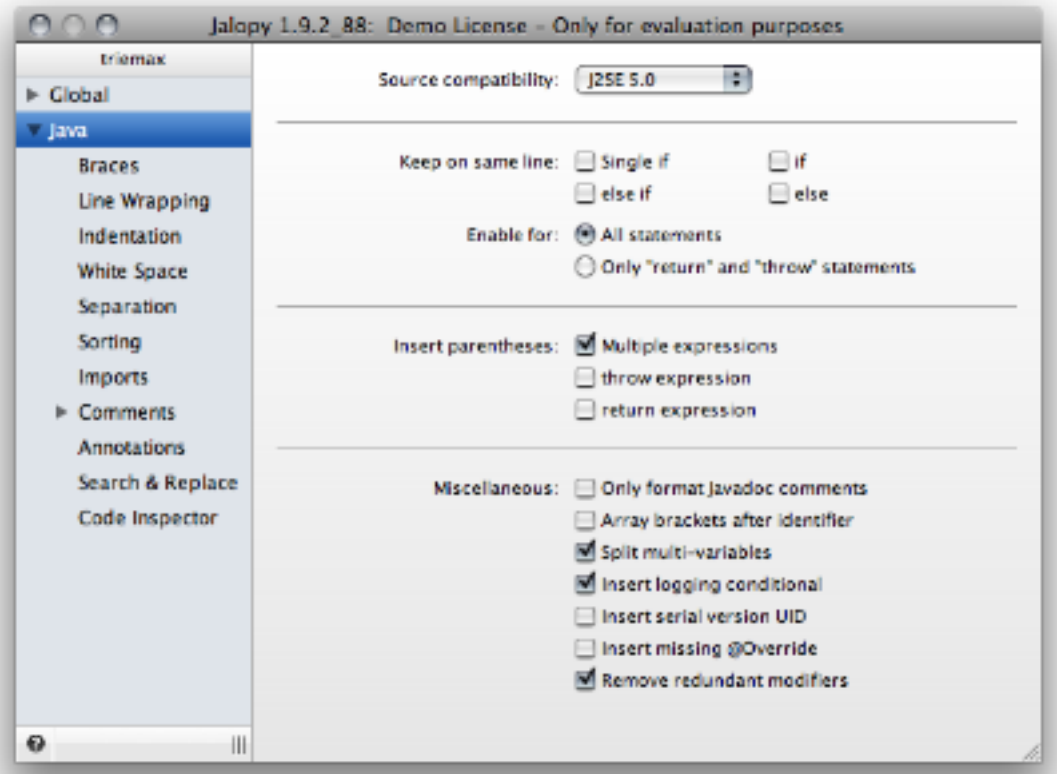
In order to be able to share settings across different systems and users, file paths should be stored relative to make the code convention portable. Jalopy therefore exports all file paths below its settings directory as relative file paths. History, backup and message log directories are by default set to paths below the Jalopy settings directory and are therefore correctly handled by the export. If you should have specified custom file paths here, you should check these paths and adjust them when necessary, after a code convention has been imported.

Please note that exporting only covers the actual code convention settings. All other profile data (history, backup, logs, reports) is ignored. If you really need to share *all* profile data, just copy the whole settings directory or selected profile folders over.

Configuration window

The Configuration window provides a tree view on the left that lets you navigate between the different preferences screens and the current preference screen displayed on the right that provides the actual options to configure the current profile. As a hint, the name of the current profile is displayed at the top left, above the tree. The Configuration window is invoked from the Main window by pressing the *Edit..* button and automatically restores its position from the last session.

Figure 2.11. Configuration window



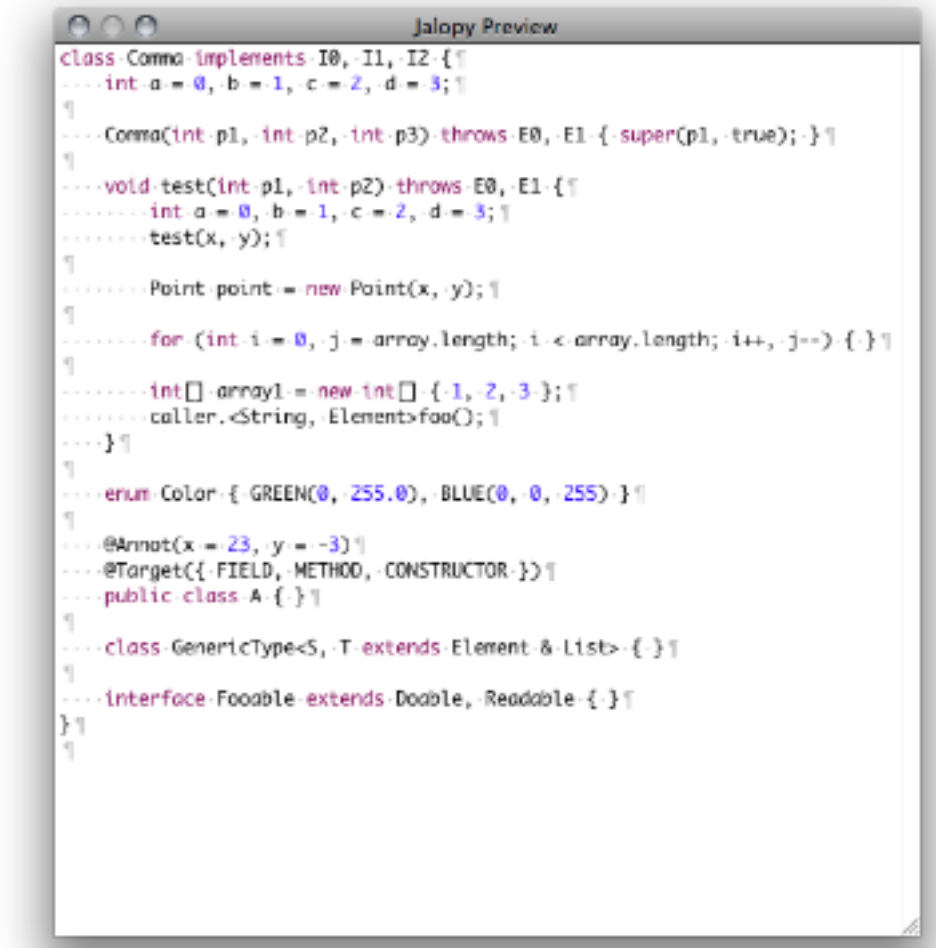
To navigate between the different available screens, you can use the tree view on the left that provides access to all screens or you can cycle between the different screens by pressing *Ctrl+Left* (previous screen) or *Ctrl+Right* (next screen). On Mac OS X you use *Cmd+Left* and *Cmd+Right* instead.

NOTE Closing the Configuration window does *not* alter your profile. Any changes you might have made are not immediately saved, but temporarily recorded until you explicitly save or apply them from within the Main window

Preview window

The Preview window provides a sneak preview of the formatting style of the currently chosen preferences.

Figure 2.12. Preview window



The Preview window normally displays a short sample file that changes with each preference page and only contains elements that would be affected by the options of the active preferences page. But you can display a file of your choice by selecting File > Open... and type or browse the file you wish to be used in the preview. Alternatively, you can simply drag & drop a file to the preview text area. The custom file will then be used for all preferences pages until you explicitly close it via File > Close (which would restore the system preview file), or choose another custom file. It's also possible to automatically have the currently opened file picked up when using an IDE Plug-in. Please refer to Section 2.2.1.1, "Use current file in preview" for more information about this feature.

To visualize indentation behavior you can control the display of the usually hidden whitespace characters TAB and SPACE and EOL by selecting View > Show Whitespace

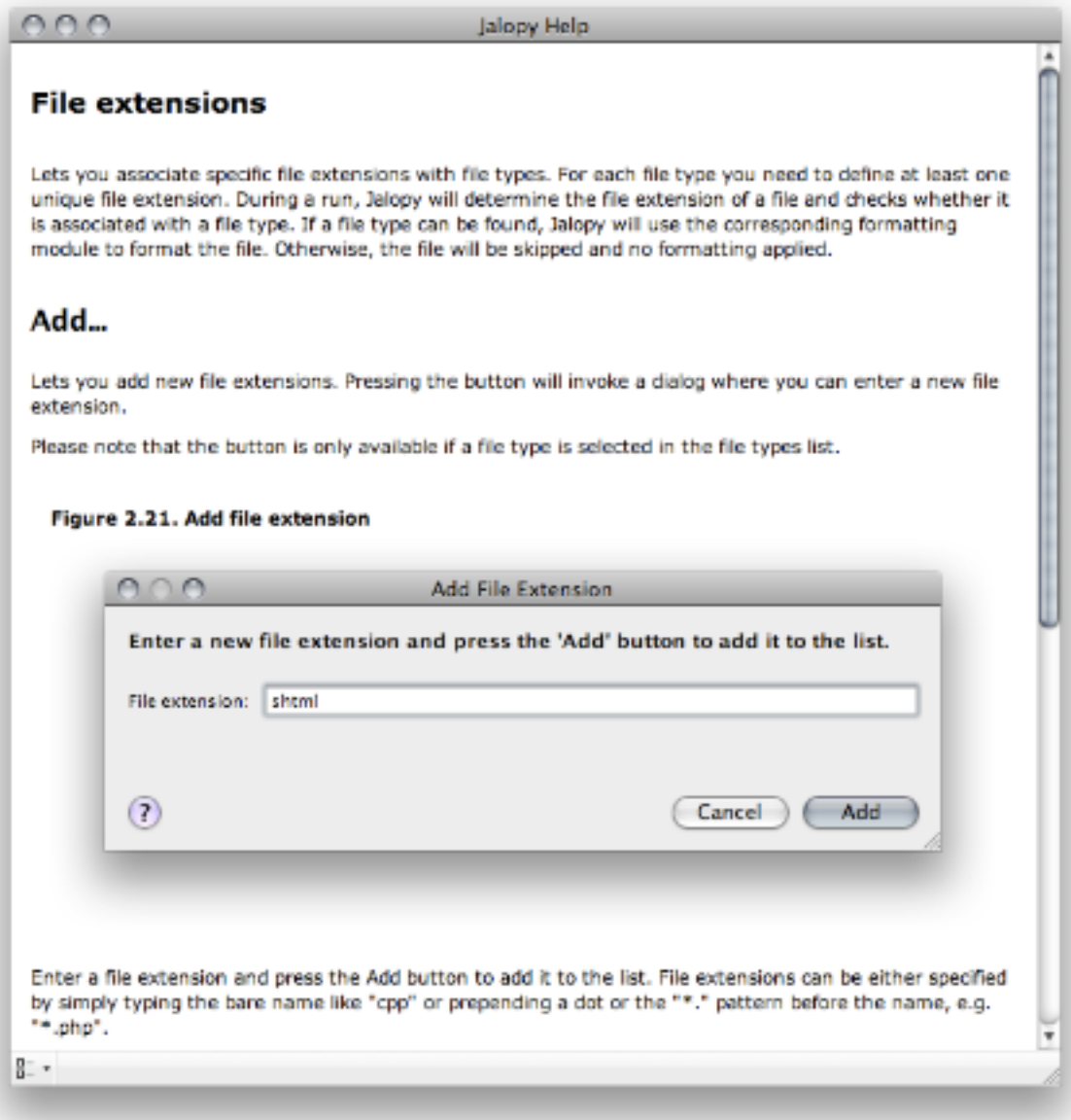
Characters and/or View > Show EOL Characters. Please note that on Mac OS X the menu actions are available through the global menu bar, while on other platforms the menu is attached to the Preview window.

Help window

The Help window lets you browse, search, and print system documentation. Please remember that you might need to install support files manually if the software was not installed using the Setup Wizard. Please refer to Chapter 1, *Installation* for more information about the installation options.

You invoke the Help window by either pressing the *F1* key at any time or by clicking the help button in a window or dialog. Please be aware that the Help window is the most prominent application window and always sits on top of all other windows.

Figure 2.13. Help window



The Help window is directly connected with the current application window or dialog and changes its contents whenever you move the mouse over a component of the application window or dialog. This way you are always presented with the most relevant information

when working with the application. But the Help window also provides different navigation views to access all available help topics in a more traditional manner. To display or switch views, you can choose one of the available options in the pop-up menu at the bottom left of the Help window.

Content view

The Content view provides a hierarchal tree view of all available help topics. Explore the topic tree to find the information you are looking for. To view a topic, click the link in the topic tree. You can use the *Forward* and *Back* buttons in the button bar at the top of the Help window to go to topics you have already visited. They behave the same way back and forward buttons work in an web browser.

Index view

The Index view provides a searchable index of the help contents. Enter a keyword in the search field and successively press the *Enter* key to display the topics that match the given search term. Directly selecting an entry in the index will display the associated topic.

Favorites view

The Favorites view lets you add and organize bookmarks for topics. You might want to add bookmarks for frequently accessed topics. To add a bookmark you select a help topic in the content view, then switch to the favorites view, open the context menu (right-click the mouse) and select the Add menu item.

2.1.2 Settings files

A synopsis of the used files is given in the table below.

Table 2.2. Settings files

Name	Purpose
alias.dat	Stores the alias names of a profile
export.dat	Stores the file history of the last ten exported code conventions
history.dat	Stores the history information of all processed files
import.dat	Stores the file history of the last ten imported code conventions
log.dat	Stores the file history of the last ten log files
page.dat	Stores the information of the last displayed settings page
project.dat	Stores the information of a profile
settings.dat	Stores the current code convention settings

The group of settings stored in `settings.dat` that describe the style of a source file is called a *code convention*. You can share code conventions using a textual XML format. See Section 2.1.1.7, “Import code convention” and Section 2.1.1.8, “Export code convention” for more information.

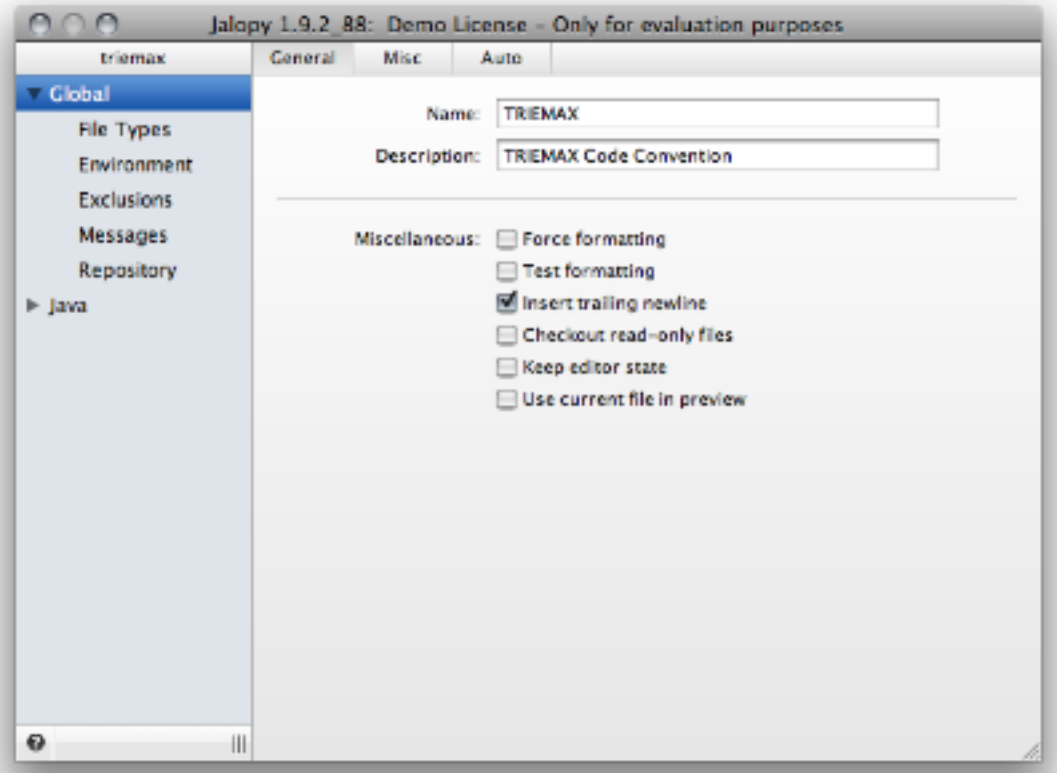
2.2 Global

Lets you configure the global settings that apply to all supported languages.

2.2.1 General

Lets you control some general preferences.

Figure 2.14. General preferences page



Name

The name of the code convention. This should be a short and unique name that easily identifies a code convention.

Description

Stores a short description for the code convention. The optional description may be used to provide a more detailed explanation of a code convention.

Name and description may be inserted into source files during formatting. See Section 2.4, “Environment” for more information.

2.2.1.1 Miscellaneous

Lets you control miscellaneous options that doesn't fit elsewhere.

Force formatting

Jalopy can keep track of which files have been formatted previously (Refer to Section 2.2.2.1, “History” for more information about this feature). If the history is enabled, Jalopy won't format files that have not changed since the last formatting. However, you can disable this check to force a reformat. For example, you might need to update the copyright notice for the whole code base. Enabling this switch ensures that all source files are always formatted.

Test formatting

When enabled, formatting output is not written to disk and/or opened editors are not updated. This may be worthwhile when you want to determine what files cause warnings or errors during formatting. This way Jalopy can be used somewhat similar to a coding style checker (see Section 2.8.19, "Code Inspector" for the available style checks). This option is mostly useful for batch mode processing, therefore it can be enabled from the Console, Ant or Maven Plug-ins directly, and should be normally left disabled here. Refer to Section 2.6.2, "Logging" for more information about the possible logging options.

Since 1.0

Insert trailing newline

When enabled, Jalopy inserts an empty line at the end of every file. This may help to avoid problems with certain text formatters and processors. Note that Jalopy always inserts at least one empty line after footers, so there is no real need (but it doesn't hurt) to check the mark in case footers are used. See Section 2.8.16, "Footer" for more information on footers.

Example 2.1. Trimmed EOF

```
package foo;¶
¶
class Foo {¶
}¶
```

Example 2.2. Trailing newline EOF

```
package foo;¶
¶
class Foo {¶
}¶
¶
```

Checkout read-only files

When enabled, Jalopy tries to checkout read-only files when it detects that a file is under source control. Such behavior should be the default with most SCM providers anyway, but if you happen to use a SCM system that does not work this way, this option might come to the rescue. Please note that this feature is currently only available with the Eclipse and IntelliJ IDEA Plug-ins.

Since 1.9.2

Keep editor state

When enabled, Jalopy will keep the current editor state when formatting editor contents. If an editor is currently dirty, i.e. contains unsaved changes, Jalopy will only update the editor. Otherwise the file is changed on disk as well. Please note that depending on your IDE an undo might not be possible when this option has been enabled.

Since 1.9

Use current file in preview

When enabled, Jalopy will use the source file that is currently opened in the editor as the preview file for the configuration dialog. Otherwise custom code snippets are used instead. The preview uses the actual editor file, not the current editor contents. One therefore needs

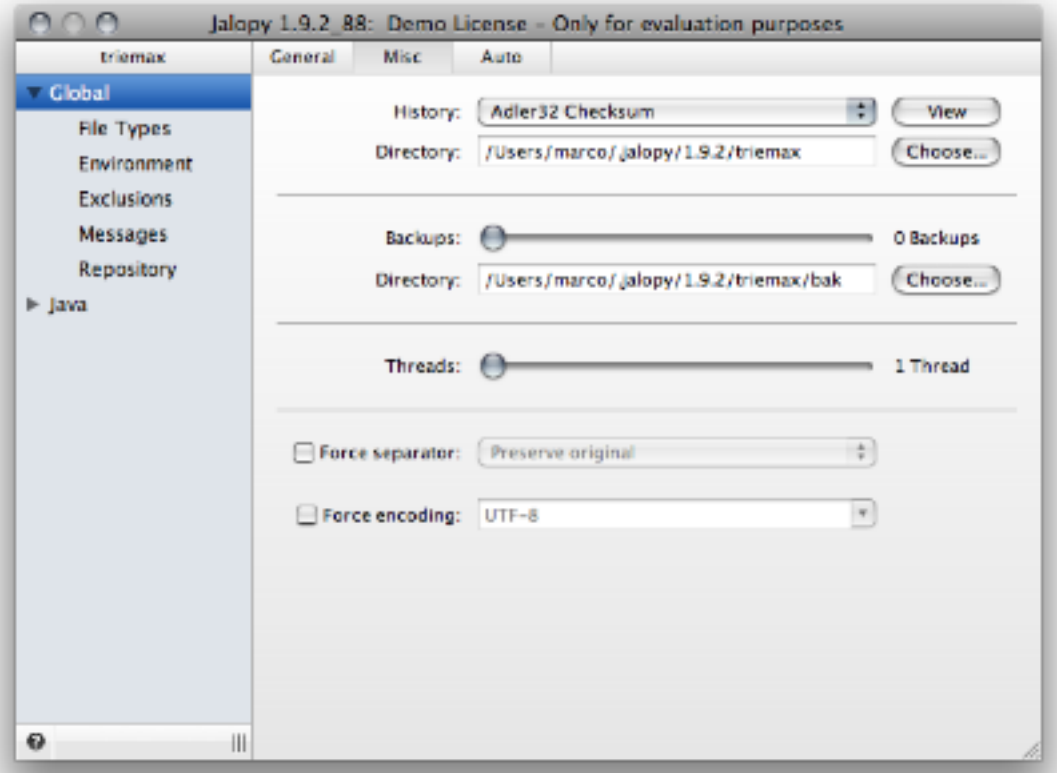
to persist any changes before they show up in the preview. Please note that you need to close and re-open the configuration dialog before an option change will take effect.

Since 1.9

2.2.2 Misc

Lets you control miscellaneous settings.

Figure 2.15. General Misc settings page



2.2.2.1 History

In order to efficiently use formatting of projects with several developers, it is important to be able to only format files which have changed. Jalopy provides a simple way to accomplish this by calculating checksums. This stops formatting files that have just been updated from source control from having being formatted (and time stamps updated) and thus prevents checking-in files that actually weren't touched by the developer later on.

To enable the history feature, select your preferred checksum method from the combo box on the left. Adler32 is faster, CRC32 is slightly more accurate. The history information of previous formatting runs will be saved in a file `"history.dat"`. Since Jalopy 1.0.3, you can specify the directory where the file is actually stored. The default is to store the history file in the current profile directory. The history file will grow over time, especially if one manages several big projects which share the same profile. As all history entries are read into memory at startup, it could eat up quite a bit of resources. Therefore a simple history viewer is provided which enables you to selectively remove obsolete entries if need be.

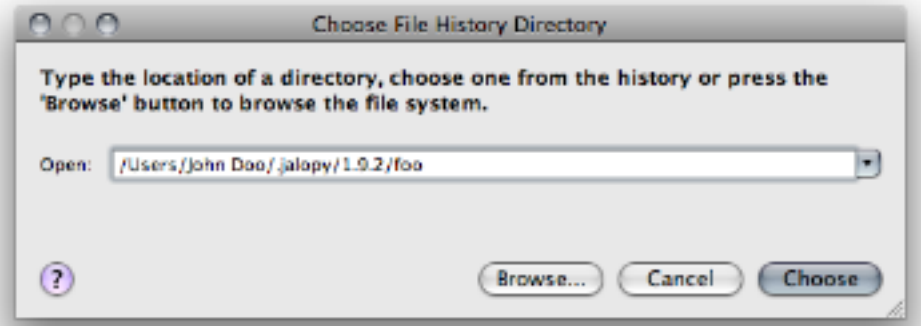
View

You can use the *View* button to display the history viewer. Entries can be selectively removed via the pop-up menu.

Directory

To change the directory where the history data is stored, press the *Choose...* button. A dialog appears that lets you enter a new directory or—in case the history directory was already changed—select one from out of the last ten chosen history directories.

Figure 2.16. Choose history directory



Either enter a directory in the text field directly, or press the *Browse...* button to invoke a directory browser that lets you search the file system for an existing folder or create a new one.

Since 1.0.3

2.2.2.2 Backup

For security reasons, Jalopy creates a backup copy before it overwrites a file so the file may be restored in case a severe error occurred during the write process. The original file is stored in the backup directory and normally removed after the newly formatted file has been successfully written.

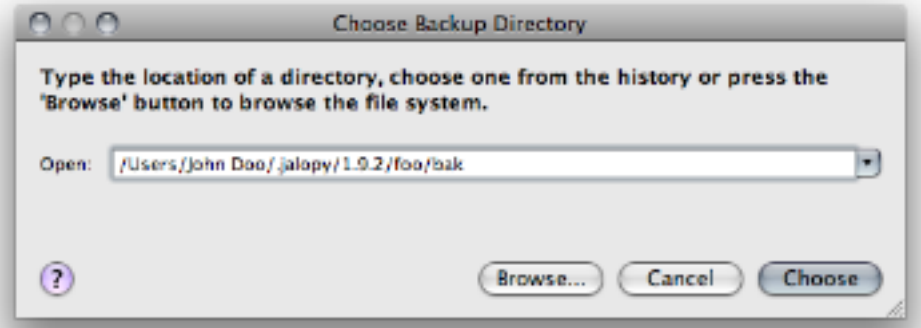
Level

The backup level defines how many numbered backups should be retained (up to 30). The default is to never keep any backups (i.e. the backup level is set to '0'). Use the slider to set the number of backups you want to keep.

Directory

Specifies the directory where file backups are stored. You should leave this setting untouched in order to make your code convention portable across different systems and platforms (See Section 2.1.1.8, “Export code convention” for more information about possible portability issues). To change the backup directory, click on the *Choose...* button. A dialog appears that lets you enter a new directory or in case the backup directory was already changed, select one from out of the last 10 chosen backup directories.

Figure 2.17. Choose backup directory



2.2.2.3 Threads

During batch-formatting, Jalopy can divide the work onto several processors and cores to speed up processing. If you run Jalopy on a multi-processor or multi-core system, use the slider to set the corresponding number of processors or cores (or the multiplier when using a multi-processor system with several cores).

2.2.2.4 Force separator

Lets you specify whether and what line ending character(s) should be forced. Enabling the check box causes the selected line separator to be forced for newly formatted files. You can choose from one of the two common platform styles (Unix, Windows) to enforce a specific line terminator. Or select *Platform default* if you want to obey the line terminator of your platform. Choosing *Preserve original* keeps the style of the source files, but please note that Jalopy does not support mixed line separators. It will use the style of the first line separator found in a source file for the complete file!

When left disabled, the default behavior depends on the used Plug-in: The Ant, Console and Maven Plug-ins preserve the original line separator by default (this may be overridden via the “fileformat” attribute, the “format” command-line option, or the “fileFormat” parameter). The Eclipse 2.x and NetBeans Plug-ins preserve the original format, too. All other Plug-ins use the corresponding IDE setting (sometimes called line terminator or end-of-line characters).

Since 1.2.1

2.2.2.5 Force Encoding

Lets you specify a specific output encoding to be used to write files. Enabling the check box causes the selected encoding to be forced for newly formatted files. You can either choose from one of the platform supported encodings or specify a specific encoding yourself.

When left disabled, the behavior depends on the used Plug-in. When using one of the IDE Plug-ins, the file encoding as specified in the IDE would be used. But the Ant, Console or Maven Plug-ins would use the platform default encoding instead.

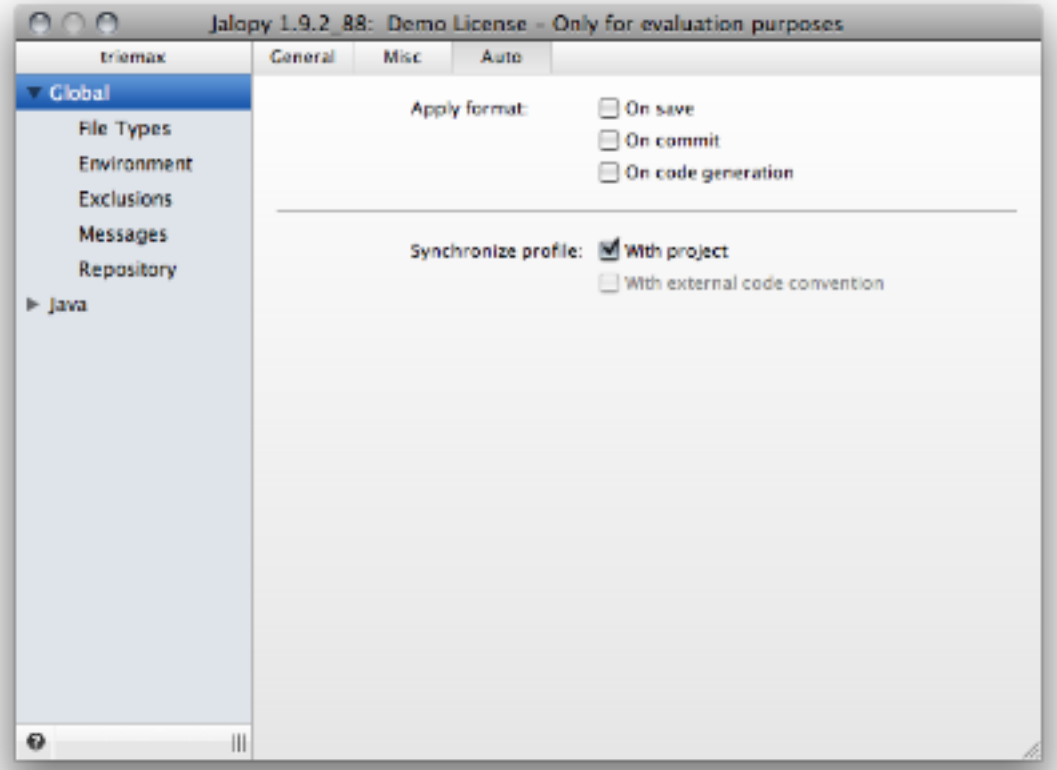
Since 1.9.1

TIP The Non-IDE Plug-ins allow you to control input and output encoding via configuration attributes/options directly. Please refer to the documentation of the individual Plug-ins for more information.

2.2.3 Auto

Lets you control the auto-format settings.

Figure 2.18. Auto format settings page



On save

When enabled, formatting is performed whenever a dirty file is saved.

Since 1.0.3

On commit

When enabled, files are formatted prior to be committed back to the source code management system (SCM). This feature is currently only available with IDEA 5.1 or later.

Since 1.8

On code generation

When enabled, source files are automatically formatted after they were generated from the model. This feature is currently only available with IBM Rational Systems Developer.

Since 1.9.1

With project

When enabled, Jalopy tries to activate the profile with either the same name as or aliased by the current IDE project before a file gets formatted. If no corresponding profile exists for the current IDE project, formatting uses the settings of the active Jalopy profile. Please note that for this feature to work efficiently, all profiles should have their auto-switch option enabled!

Since 1.0.2

With external code convention

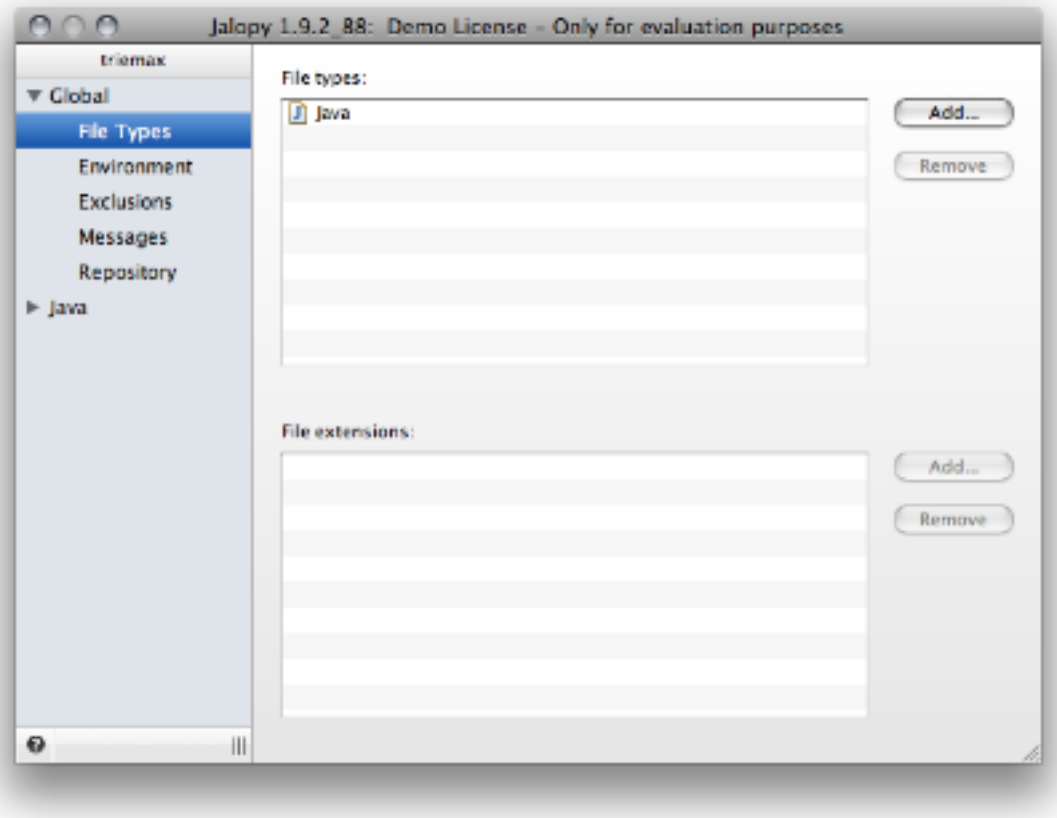
When you're working with a group of developers, it can be very useful to share specific code conventions with your team. When this option is enabled, Jalopy checks upon every invocation whether any changes to the shared code convention(s) were applied and if so, updates the local profile accordingly before the sources are formatted. This way all members of the team always use the same coding style without any further user intervention required. Please note that the check box is only available after a code convention has been imported.

Since 1.0

2.3 File Types

Jalopy provides formatting support for different file types. On the *File Types* settings page you can specify what file types should be supported and what files belong to a specific file type.

Figure 2.19. File types settings page



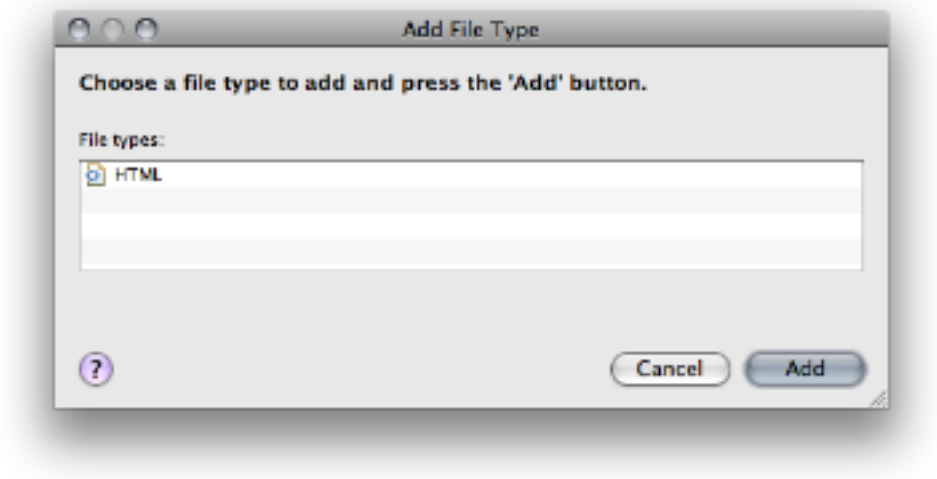
2.3.1 File types

Lets you enable/disable support for specific file types. Formatting is enabled by adding a file type to the list and disabled by removing it from the list. After a file type has been added, all associated files are formatted by the corresponding formatting module.

Add...

Lets you add new file types. Pressing the button will invoke a dialog where you can select from a fixed list of available file types to add.

Figure 2.20. Add file type



Select one or several file types and press the *Add* button to add them to the list. Please note that Jalopy will automatically register some well known file extensions for each file type you add. You might want to review these mappings and adjust them at your wish. Please refer to Section 2.3.2, “File extensions” below.

Remove

Lets you remove an already defined file type and thus disable formatting for the file type. Select one or several file types and press the *Remove* button to alter the list. Please note that the button is only available if at least one file type is selected.

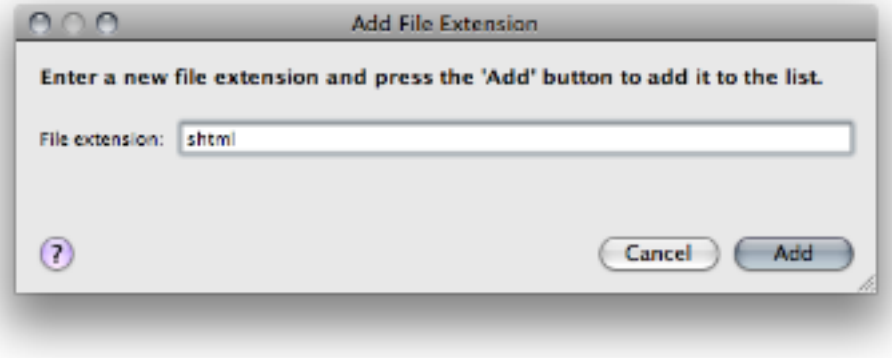
2.3.2 File extensions

Lets you associate specific file extensions with file types. For each file type you need to define at least one unique file extension. During a run, Jalopy will determine the file extension of a file and checks whether it is associated with a file type. If a file type can be found, Jalopy will use the corresponding formatting module to format the file. Otherwise, the file will be skipped and no formatting applied.

Add...

Lets you add new file extensions. Pressing the button will invoke a dialog where you can enter a new file extension. Please note that the button is only available if a file type is selected in the file types list.

Figure 2.21. Add file extension



Enter a file extension and press the *Add* button to add it to the list. File extensions can be either specified by simply typing the bare name like "cpp" or prepending a dot or the "*." pattern before the name, e.g. "*.php".

Remove

Lets you remove an already defined file extension. Select one or several file extensions and press the *Remove* button to alter the list. Each file type must have at least one associated file extension in order to enable formatting for the file type. If no file extension is associated with a file type, no formatting will be applied for the file type! Please note that the button is only available if both a file type is selected in the file types list and at least one file extension in the file extensions list.

2.4 Environment

Lets you specify, view and adjust environment variables. Environment variables are simple key/value pairs and can be used in header, footer and Javadoc templates to form expressions that will be resolved during formatting. Embedded strings of the form `$variable$` are replaced with their corresponding value. This process is called *variable interpolation*.

Valid keys take the form `[a-zA-Z_][a-zA-Z0-9_.-]*` and are case-sensitive. Values can be freely chosen.

Example 2.3. Header template with environment variables expressions

```
//-------------------------------------------------------------
// file :       $file.name$
// project:     $project$
//
// create:      date:       $date$
//              by:         $author$
//
//-------------------------------------------------------------
// copyright:   BSJT Software License (see class documentation)
//-------------------------------------------------------------
```

Example 2.4. Sample environment variables

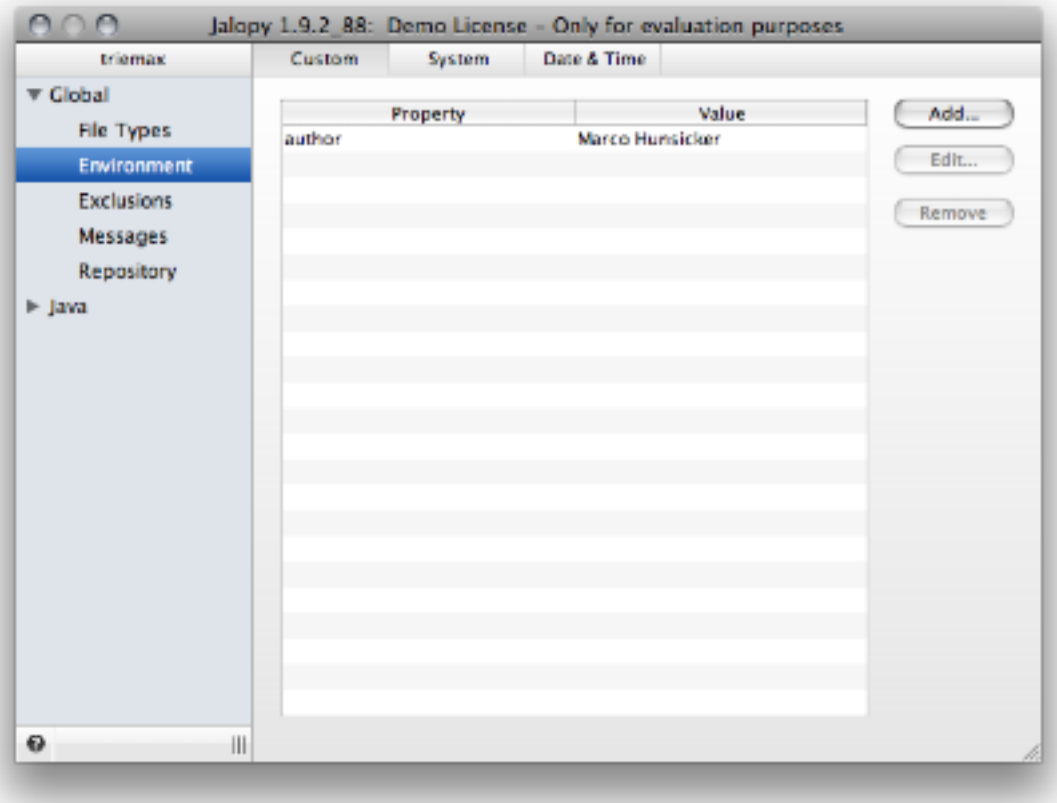
```
author = John Doo
project.description = Nukes: The OpenSource CMS
```

Jalopy lets you define custom variables, but also provides access to the Java environment variables as well as some Jalopy-specific variables that are generally useful for common source formatting needs.

2.4.1 Custom environment variables

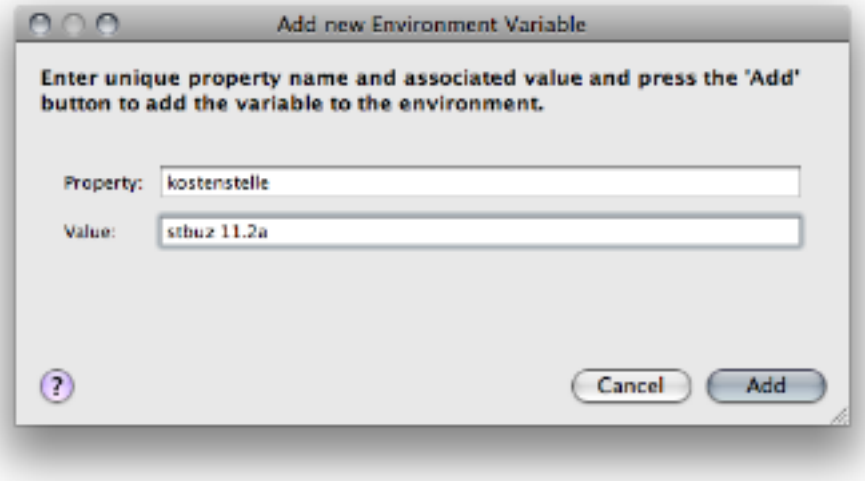
Lets you view and/or modify the custom environment variables.

Figure 2.22. Custom Environment settings page



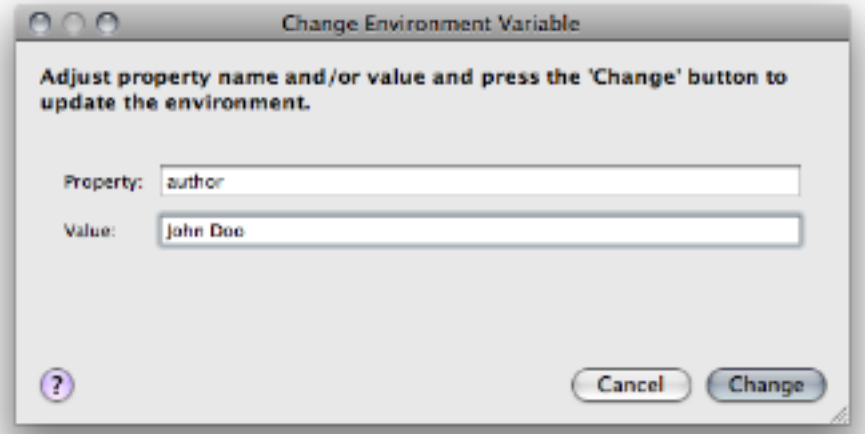
Use the *Add...* and *Remove* buttons to add or remove items to and from the list.

Figure 2.23. Add Custom Environment Variable



Select a variable and press the *Change...* button if you want to adjust an existing environment variable.

Figure 2.24. Change Custom Environment Variable



Local Overrides

Custom user variables are stored as part of your code convention and are therefore shared across a whole team. If you need to define user-specific information, e.g. to automatically add the name of the developer who creates a class, this is possible via the local overrides file `".user.properties"`.

When found in the Jalopy settings directory, the specified variables will override any other custom or system variables. The overrides file uses the common java.util.Properties format.

Table 2.3. Typical .overrides locations for user "John Doo"

Operating System	Jalopy .overrides Location
Linux	/home/John Doo/.jalopy/.user.properties
Mac OS X	/Users/John Doo/.jalopy/.user.properties
Solaris	/export/home/John Doo/.jalopy/.user.properties
Windows Vista	C:\Users\John Doo\AppData\Roaming\.jalopy\.user.properties
Windows XP	C:\Documents and Settings\John Doo\.jalopy\.user.properties

Please consult your operating system documentation if your system uses different paths for the user directories. Detailed information about the Jalopy settings directory can be found in Chapter 2, *Configuration*.

Since 1.6

Example 2.5. Local Overrides File

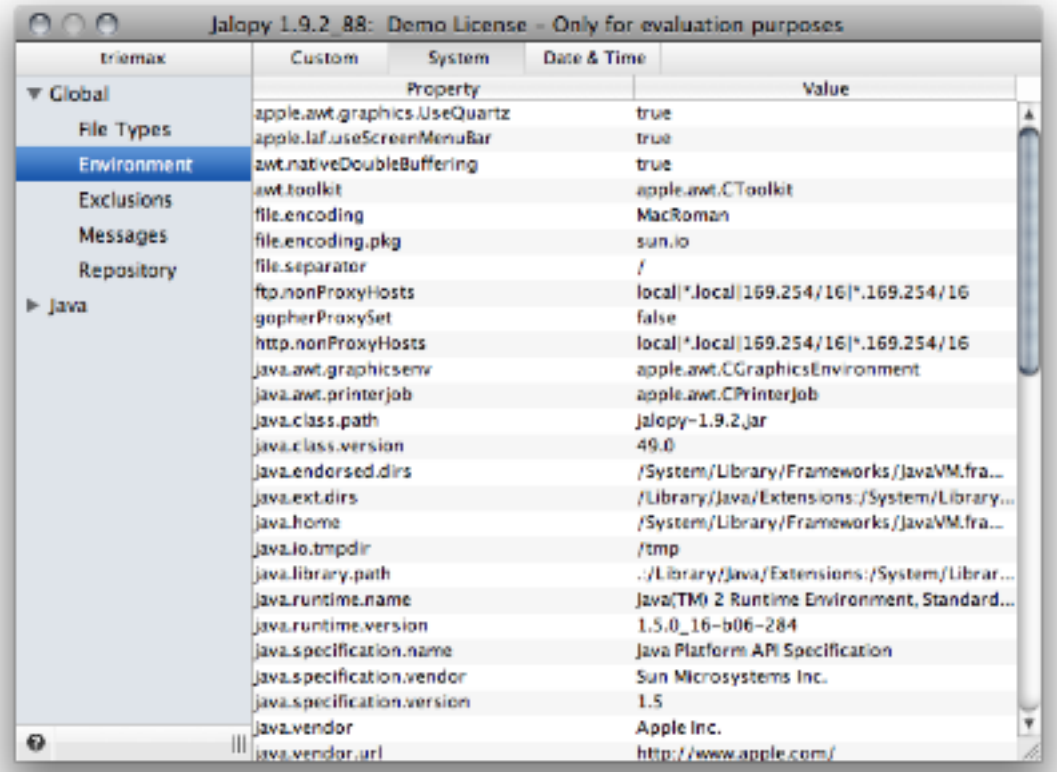
```
author=John Doo
division=IT_DEV_AR
```

The above example would define/override the variables "author" and "division". Please note that the environment may be overwritten manually using the Console and Ant Plug-ins as well. Please refer to the corresponding Plug-in documentation.

2.4.2 System environment variables

Lets you view your Java system environment variables. All system environment variables are automatically available as well, but cannot be changed from within Jalopy.

Figure 2.25. System environment variables



2.4.3 Local environment variables

Additionally, Jalopy provides some local variables that are automatically set depending on the execution context. The current list of valid local variables reads as follows:

Table 2.4. Local environment variables

Name	Description	Scope	Since
file	The absolute path of the currently processed source file (e.g. `/usr/projects/test/MyFile.java`)	global	1.0
file.name	The name of the currently processed source file (e.g. `MyFile.java`)	global	1.0
file.format	A string representation of the line ending character(s) used to write a file (UNIX, DOS or MAC)	global	1.0
convention	The name of the currently active code convention (as specified in the settings)	global	1.0
convention.desc	The description of the currently active code convention (as specified in the settings)	global	1.0
project	The name of the currently active project/profile. For IDE Plug-ins this resolves to the current IDE project, otherwise the active Jalopy profile name is used	global	1.0.1
project.desc	The description of the currently active Jalopy profile	global	1.0.1

Name	Description	Scope	Since
tab.size	The current indentation setting (as specified in the settings)	global	1.0
date	The current date, formatted in the style specified in the Date/Time settings (see Section 2.4.5, "Date/Time" below)	global	1.0
date.year	The current year	global	1.0
date.long	The current date, formatted as `java.text.DateFormat` LONG style	global	1.0
date.full	The current date, formatted as `java.text.DateFormat` FULL style	global	1.0
time	The current time, formatted in the style specified in the Date/Time settings (see Section 2.4.5, "Date/Time" below)	global	1.0
time.long	The current time, formatted as `java.text.DateFormat` LONG style	global	1.0
time.full	The current time, formatted as `java.text.DateFormat` FULL style	global	1.0
package.name	The package name of the currently processed source file (e.g. com.foo.mypackage)	global	1.0
class.name	Holds the name of the currently processed class, interface or enum	Javadoc class, interface, field, constructor, method, getter, setter	1.0
field.name	Holds the name of the currently processed field	Javadoc field	1.0
field.type	Holds the type name of the currently processed field	Javadoc field	1.0
method.name	Holds the name of the currently processed method	Javadoc method	1.0
param.name	Holds the name of the currently processed Javadoc parameter tag	Javadoc constructor, method	1.0
param.type	Holds the type of the currently processed Javadoc parameter tag	Javadoc constructor, method	1.0
exception.type	Holds the type of the currently processed throws clause member	Javadoc constructor, method	1.0
return.type	Holds the return type of the currently processed method	Javadoc method	1.0
property.name	Holds the property name of the currently processed getter/setter method. You can control the behavior during variable interpolation with the "Format bean property" option	Javadoc getter/setter	1.1
Property.Name	Holds the capitalized property name of the currently processed getter/setter method	Javadoc getter/setter	1.9.3

2.4.4 Usage

Once defined, variables can be enclosed with dollar signs to form variable expressions and embedded in comment templates. Variable expressions take the form `$[a-zA-Z_][a-zA-Z0-9_.-]*$`.

Example 2.6. Sample variable expressions

```
$author$
$project$
```

During emitting, these expressions will be interpolated and the value of the variable inserted into the source file.

Example 2.7. Header template with environment variable expressions

```
//-------------------------------------------------------------
// file :       $file.name$
// project:     $project$
//
// last change: date:       $Date$
//              by:         $author$
//              revision:   $Revision$
//-------------------------------------------------------------
// copyright:   BSJT Software License (see class documentation)
//-------------------------------------------------------------
```

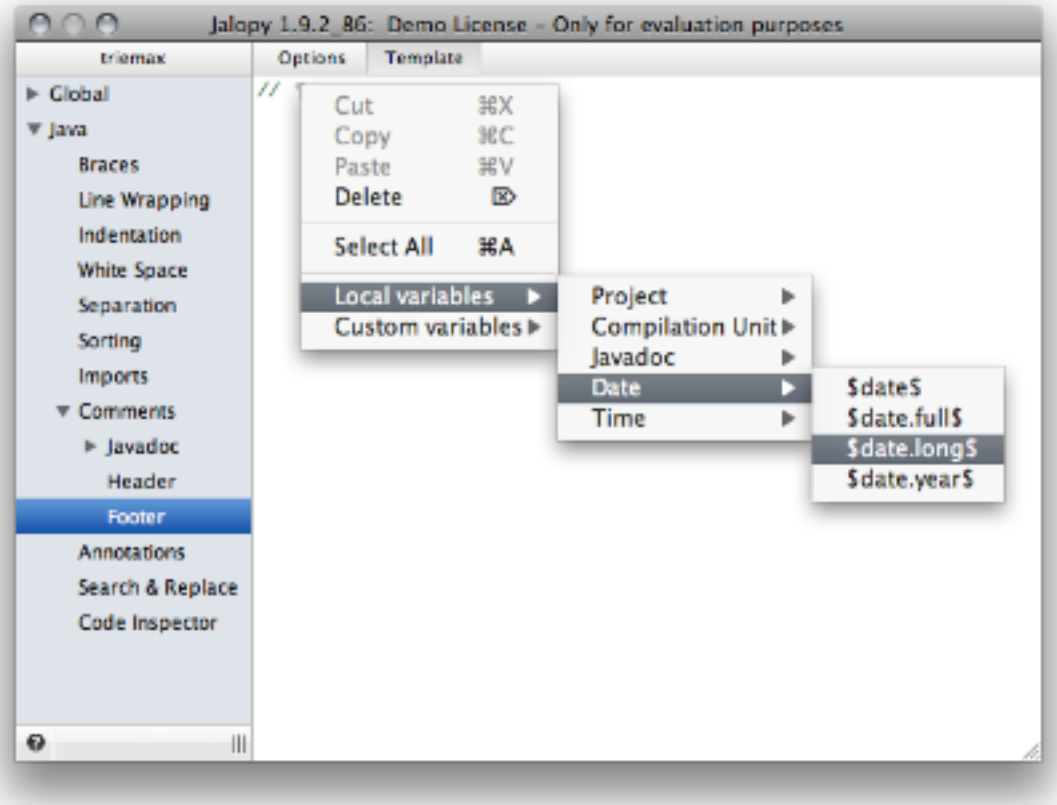
Example 2.8. Header after interpolation

```
//-------------------------------------------------------------
// file :       Byte.java
// project:     bsjt-rt
//
// last change: date:       $Date$
//              by:         John Doo
//              revision:   $Revision$
//-------------------------------------------------------------
// copyright:   BSJT Software License (see class documentation)
//-------------------------------------------------------------
```

As you can see in the above example, if a variable is not defined, Jalopy won't touch the expression and simply preserves the original content. This way, the formatter works nicely with other source code tools and SCM products.

The available user and local environment variables are provided from within the context menu of the text component when customizing the different templates.

Figure 2.26. Insert variables via context menu

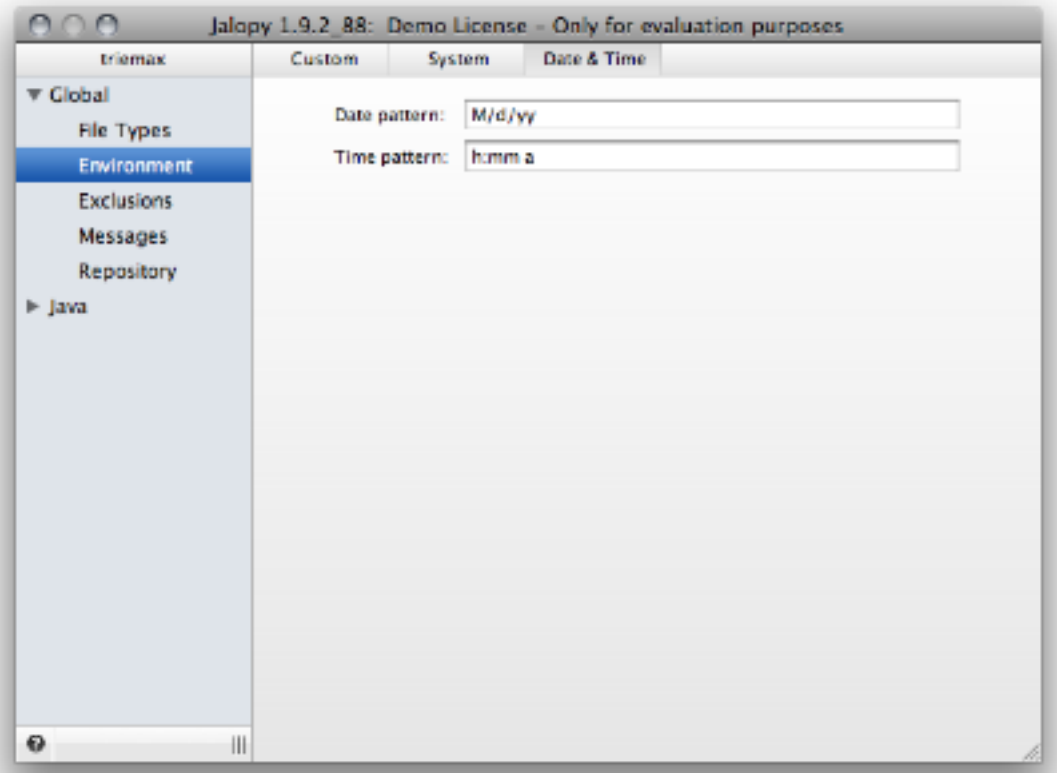


Please see Section 2.4.1, “Custom variables”, Section 2.4.2, “System variables” and Section 2.4.3, “Local variables” for descriptions of the different available variable types.

2.4.5 Date & Time settings

Lets you define the patterns that should be used for the `$date$` and `$time$` expressions (see Section 2.4.3, “Local variables”).

Figure 2.27. Environment Date & Time settings page



For a detailed description, the list of available patterns and further examples, please refer to the Javadoc for the `java.text.SimpleDateFormat` class.

Date pattern

Lets you define the pattern that is used for the `$date$` variable expression. The default pattern is `M/d/yy`, which translates to something like `07/23/09`.

Since 1.2

Time pattern

Lets you define the pattern that is used for the `$time$` variable expression. The default pattern is `h:mm a`, which translates to something like `02:56 PM`.

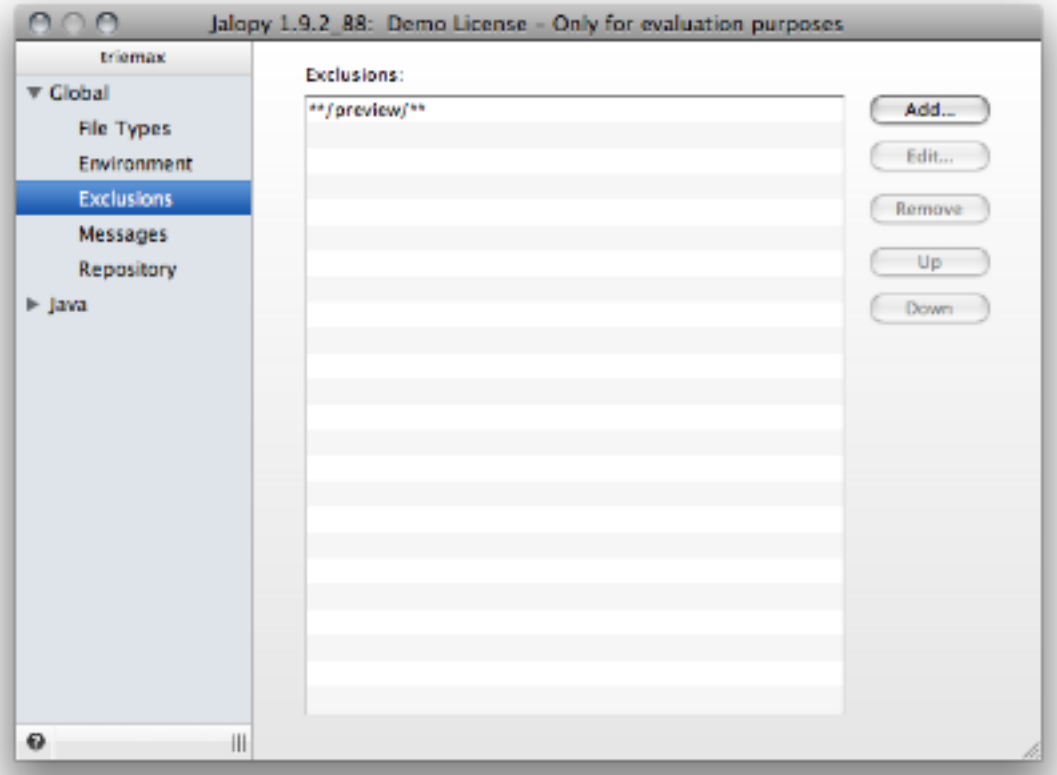
Since 1.2

2.5 Exclusions

Lets you configure exclusion patterns that can be used to omit certain directories or individual files from formatting.

Since 1.9.1

Figure 2.28. Exclusions settings page



2.5.1 Exclusion patterns

Lets you define exclusion patterns that should be used for resource matching. You can specify an arbitrary amount of exclusion patterns. The patterns will be checked in sequence and a file will only be formatted if none of the patterns matches the absolute file or parent directory path. The list component displays all patterns currently defined. Use the button bar on the right to add, remove or change patterns and define the order in which the patterns should be checked.

Add...

Lets you add new exclusion patterns. Pressing the button will invoke a new dialog where you can enter the pattern. The used pattern syntax looks very much like the patterns used in UNIX: * matches zero or more characters, ? matches one character. For example: to match all Java files in a specific directory, you could use /test/foo/*.java. To match all files whose name starts with “Foo”, you could use /test/foo/Foo.*.

To make things a bit more flexible, there is one extra feature which makes it possible to match multiple directory levels. This can be used to match a complete directory tree, or a file anywhere in the directory tree. To do this, ** must be used as the name of a directory. When

** is used as the name of a directory in the pattern, it matches zero or more directories. For example: /test/** matches all files and directories under /test/, such as /test/Foo.java or /test/foo/bar/Bar.java, but not /Foo.java.

Most often you probably want to exclude whole directories from being formatted, e.g. to exclude some test data from being processed. This might be achieved by defining a pattern like **/test/** to exclude everything below directories named "test". If you only want to exclude something in the project foo it might look like this: **/foo/**/testdata/**.

Please note that patterns only allow forward slashes. Any backslashes will be automatically replaced. The pattern matching itself is platform-agnostic and patterns match even on platforms that don't use the forward slash as the file separator.

Change...

Lets you alter an already defined exclusion pattern. The button is only available if an item is currently selected in the pattern list. Pressing the button will invoke a new dialog where you can change the exclusion pattern for the currently selected item in the pattern list.

Remove

Lets you remove an already defined exclusion pattern. The button is only available if an item is currently selected in the pattern list.

Up

Lets you change the position of an already defined exclusion pattern in the pattern list. The button is only available if an item is currently selected in the pattern list and this is not the topmost item.

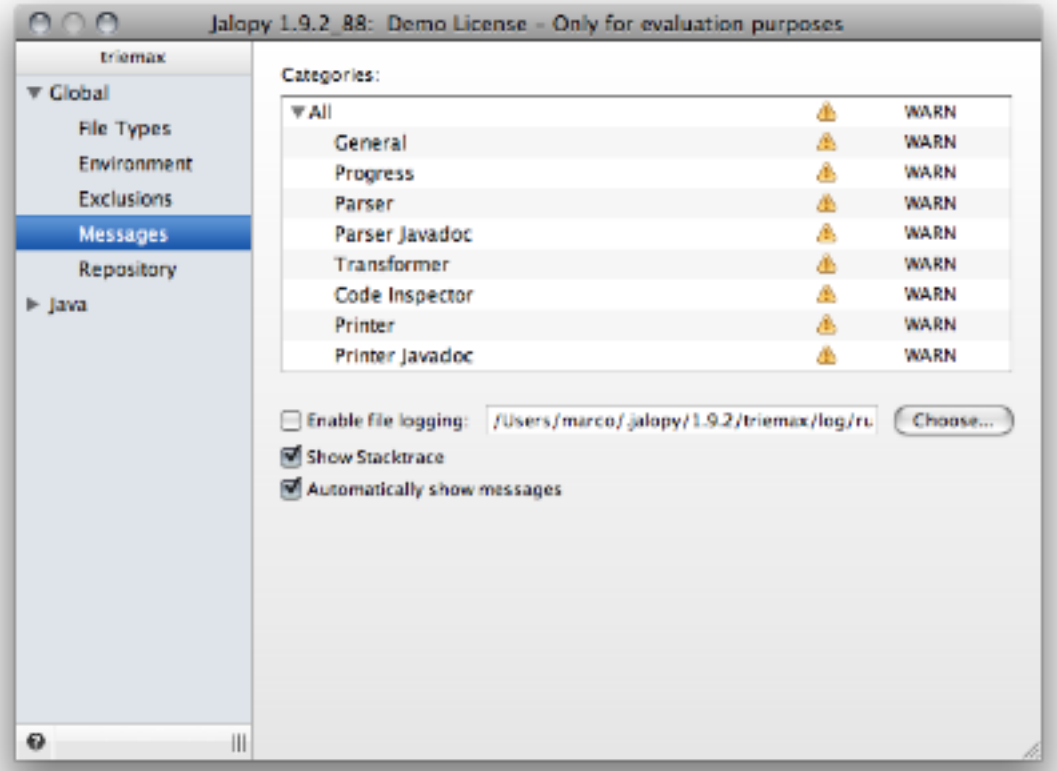
Down

Lets you change the position of an already defined exclusion pattern in the pattern list. The button is only available if an item is currently selected in the pattern list and this is not the last item.

2.6 Messages

Controls the Jalopy message output. Given the sensitive nature of automatic source code processing, it is quite important to keep informed about what is going on during the formatting process. The default configuration should be sufficient for most users, so there should seldom arise the need to change anything here.

Figure 2.29. Messages settings page



2.6.1 Categories

Jalopy provides different message categories for setting the logging verbosity level. Changing the threshold from *ERROR* to *DEBUG* successively displays more (albeit not necessarily more useful) messages.

1. General

   General purpose chain for I/O activity and all sorts of stuff that doesn't fit elsewhere.

2. Parsing

   Message chain for messages related to the language parsing.

3. Javadoc parsing

   Message chain for messages related to the parsing of Javadoc comments.

4. Transforming

   Message chain for messages related to the post-processing of the generated parse tree.

5. Printing

   Message chain for the main printing engine.

6. Javadoc Printing

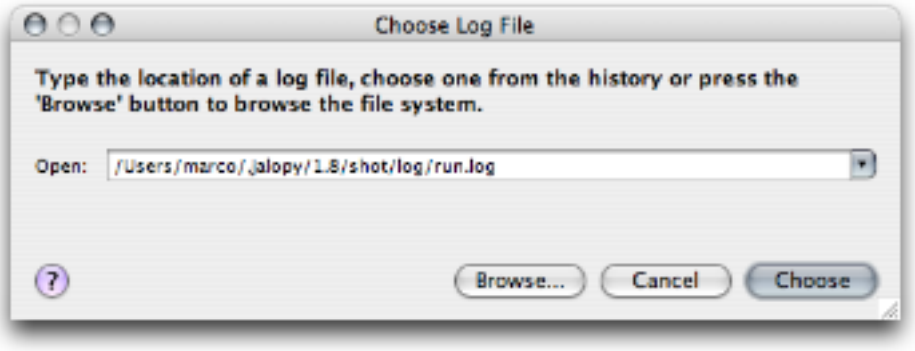
Message chain for Javadoc printing related messages.

2.6.2 Logging

Use logfile

Normally, messages are only printed to the console or the message view of your IDE, here you can define that all messages should be written to a log file as well. This is most useful when you use Jalopy as part of an automated build process. Press the *Choose...* button to select the file you wish to write logging output to.

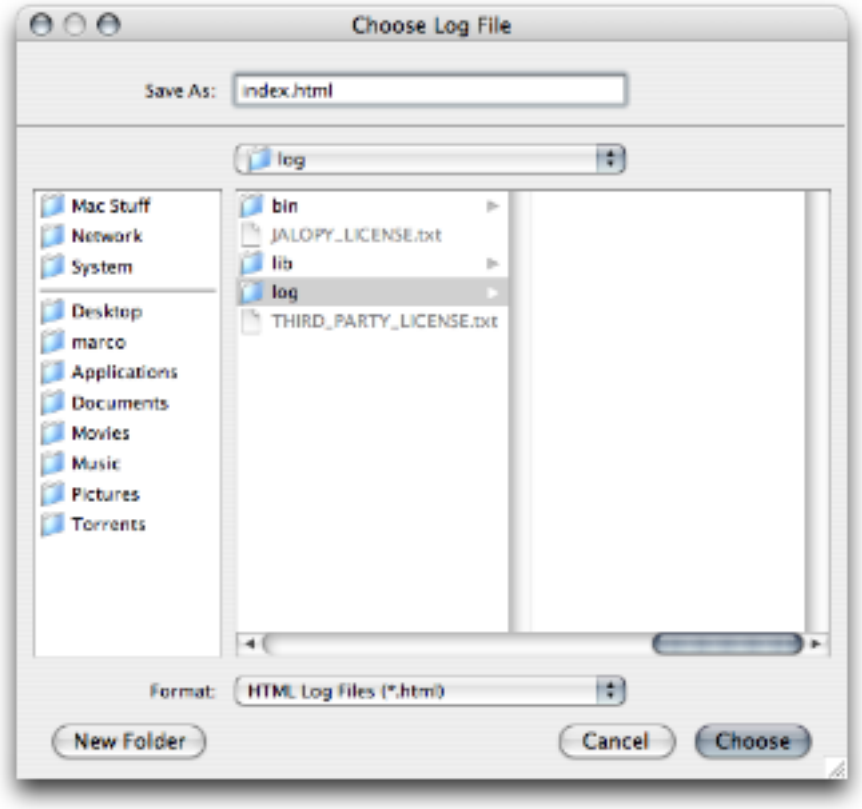
Figure 2.30. Choose Log File



You can either directly enter a location into the combo box, choose one from the history, or press the *Choose...* button to display a file chooser that lets you interactively select a file.

Jalopy supports three different log file formats. A custom text format *(.log)*, an HTML format *(.html)* and a flat XML format *(.xml)*. These formats can only be selected via the file chooser.

Figure 2.31. Choose Log File format



Both the custom text format and the XML output are simple flat file formats, but the HTML option produces an hierarchical report similar to that generated by the Javadoc tool. Please note that this feature can be enabled from the Ant, Console and Maven Plug-ins directly, so it should be normally left disabled here.

Since 1.0

2.6.3 Misc

Show stacktrace

Enables or disables the inclusion of the current execution stack trace with error messages. This proves useful in case you need to file a bug report as it reveals the source of the error.

Automatically show messages

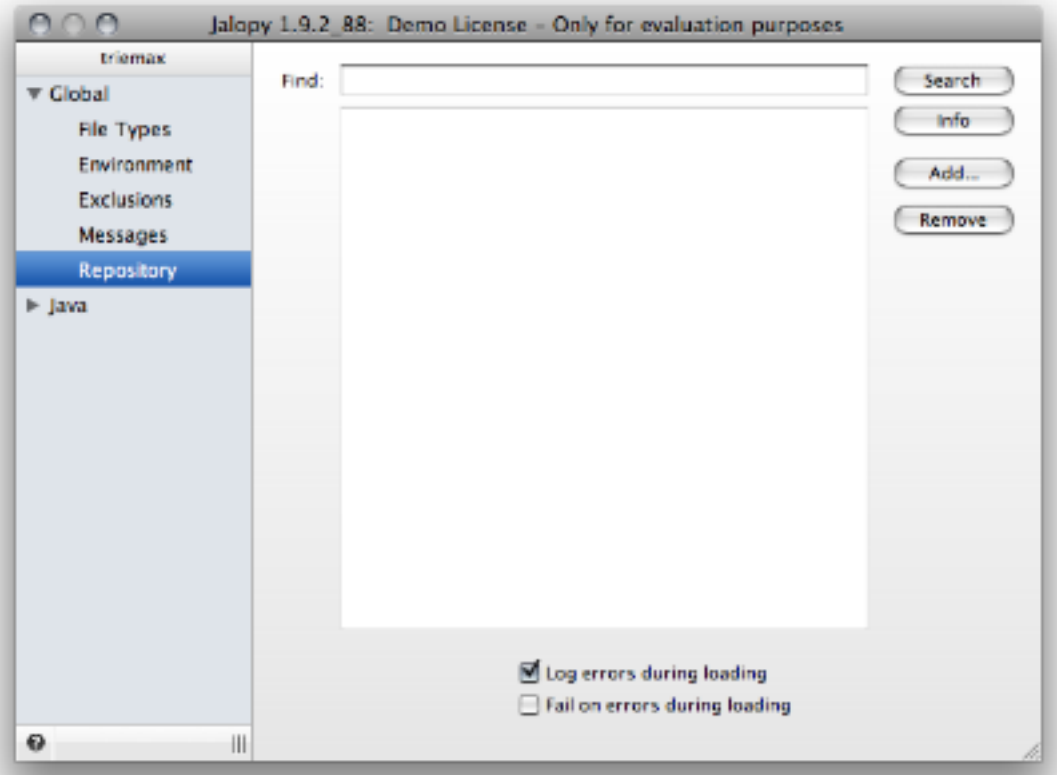
When enabled, messages are automatically displayed when a formatting run finished. Otherwise you manually need to activate the corresponding view in order to access messages. This option only applies for the IDE Plug-ins.

2.7 Repository

For certain features, Jalopy needs to know about the referenced types. When used inside an IDE, Jalopy uses the provided infrastructure to access this information. But when run standalone, such information in maintained in a simple database, called the type repository. The repository page provides the means to maintain the database. But as an end user you

usually don't have to deal with the repository directly and the provided functionality is purely for forensic purposes.

Figure 2.32. Repository settings page



The type repository is only necessary for the Ant, Console and Maven Plug-ins and used for features that require type resolution like the import optimization. Unlike all other functionality which works purely on the source code level, type information is extracted by analyzing the binary files of a project. It is therefore a necessity that the class path is correctly configured in order to be able to process all needed files.

Since 1.0.2

2.7.1 Searching the repository

To query the type repository for a specific type or package name, enter the information in the text field and press the *Search* button. This is mostly useful only during developing of the application and might never be used by end users. You can either search for a single type name (e.g. "String"), a qualified type name (e.g. "java.lang.String") or a package name (e.g. "java.lang").

2.7.2 Displaying info about the repository

To display some statistics about the type repository, press the *Info* button.

2.7.3 Adding libraries to the repository

To manually add a class library or directory to the repository, click on the *Add...* button, select the library to add and click *Add*. This is most useful only during testing and debugging, and does not provide much pratical need for the casual end-user.

2.7.4 Removing the repository

To remove the type repository from disk, press the *Remove* button. The database will be closed if it is currently open and all stored information will be erased.

2.7.5 Initialization

During the initialization of the type repository all project class files are analyzed. Jalopy uses a byte code reader (ASM) and attempts to gather both the class name and all its referenced super class names for a given file. For 3rd party libraries, it can be possible that not all super classes can be loaded (i.e. are defined or even be part of the project). In most cases this should not be problematic, but it still potentially could hinder the successful execution of the services that rely on the type repository later on. Therefore, users can control the behavior of the repository here and specify how problems should be handled.

Log errors during loading

When enabled, Jalopy logs a warning when a class could not be fully analyzed. By default, this option is enabled as it is recommended that you manually verify that none of the mentioned files might pose problems later on. The type repository will be initialized despite the problems. All dependent features will be available.

Since 1.0.3

Fail on errors during loading

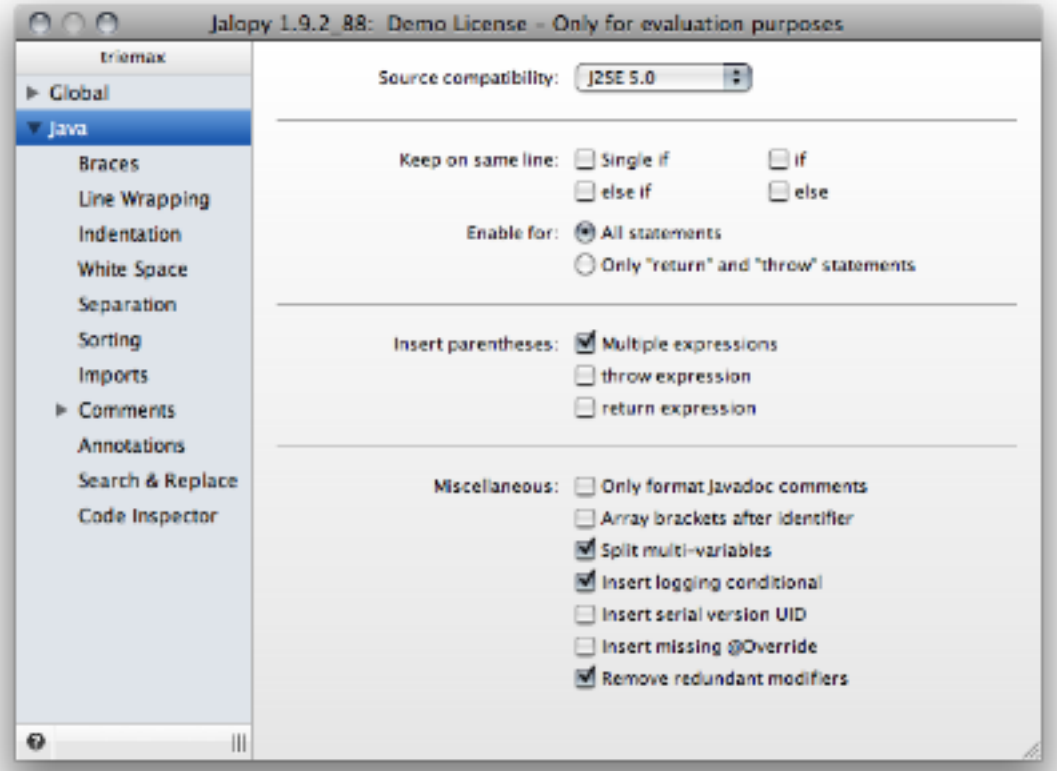
When enabled, the type repository will not be initialized when an error occurs. All dependent features will be disabled.

Since 1.0.3

2.8 Java

Lets you control all Java related settings.

Figure 2.33. Java settings page



2.8.1 Source compatibility

Lets you specify the Java platform compliance level.

Table 2.5. Compliance levels

Level	Description
Java SE 6	`assert` and `enum` are recognized as reserved keywords. Section 2.8.5, "Insert @Override annotation" may be enabled to have missing annotations inserted for overridden and implemented methods
J2SE 5.0	`assert` and `enum` are recognized as reserved keywords. Section 2.8.5, "Insert @Override annotation" may be enabled to have missing annotations inserted for overridden methods
J2SE 1.4	`assert` is recognized as a reserved keyword
J2SE 1.3	`assert` and `enum` are valid identifiers that can be used to name variables and/or methods

It is highly recommended to avoid strings that have become reserved keywords when targeting older Java releases. It's therefore probably a good idea to stick with at least J2SE 5.0 compliance.

2.8.2 Keep on same line

Lets you print certain statements on just one line when possible, i.e. if they don't exceed the maximal line length.

Single if

When enabled, prints single `if` statements in one line when possible.

Since 1.2

Example 2.9. Standard single if block

```
if (cond)                                   |
    return getMagicNumber();                |
                                            |
...                                         |
```

Example 2.10. Compact single if block

```
if (cond) return getMagicNumber();          |
                                            |
...                                         |
```

if

When enabled, prints the `if` part of if/else statements in one line when possible.

Since 1.2

Example 2.11. Standard if/else block

```
if (cond)                                   |
    return getMagicNumber();                |
else                                        |
    return -1                               |
```

Example 2.12. Compact if of if/else block

```
if (cond) return getMagicNumber();          |
else                                        |
    return -1;                              |
```

Please note that when the `if` part has been printed on one line, the following `else if` or `else` part always starts on a new line.

else if

When enabled, prints the `else if` part of if/else statements in one line when possible.

Example 2.13. Standard else of if/else block

```
if (cond1)                                                  |
    return getMagicNumber();                                |
else if (cond2)                                             |
    throw new IllegalStateException();                      |
```

Example 2.14. Compact else if block

```
if (cond1)                                                  |
    return getMagicNumber();                                |
else if (cond2) throw new IllegalStateException();          |
```

Please note that when the `else if` part has been printed on one line, the following `else` part always starts on a new line.

Since 1.2

Example 2.15. Compact if block

```
if (cond1)                                                  |
    return getMagicNumber();                                |
else if (cond2) throw new IllegalStateException();          |
else                                                        |
    return 0;                                               |
```

else

When enabled, prints the `else` part of if/else statements on one line when possible.

Since 1.2

Example 2.16. Standard else block

```
if (cond1)                                                    |
    return getMagicNumber();                                  |
else if (cond2)                                               |
    throw new IllegalStateException();                        |
else                                                          |
    return 0                                                  |
```

Example 2.17. Compact else block

```
if (cond1)                                                    |
    return getMagicNumber();                                  |
else if (cond2)                                               |
    throw new IllegalStateException();                        |
else return 0;                                                |
```

Enable for

You can narrow the scope for the above mentioned options by selecting whether all statements should be printed on one line when possible or only `throw` or `return` statements.

Since 1.2

Example 2.18. All statements on same line

```
if (true) return result;
else if (false) System.out.println("unexpected condition");
else throw new Error();
```

Example 2.19. Only throw and return statements on same line

```
if (true) return result;
else if (false)
    System.out.println("unexpected condition");
else throw new Error();
```

2.8.3 Insert parentheses

Lets you insert superfluous parentheses automatically to avoid any doubts about how an expression is evaluated.

Multiple expressions

When enabled, Jalopy inserts parentheses around expressions that involve more than two terms in order to clarify their precedence. It is always good advise to use more parentheses than you think you need. They may not be required, but they add clarity and don't cost anything.

Example 2.20. How is this expression evaluated?

```
int result = 12 + 4 % 3 * 7 / 8;
```

Example 2.21. How is this expression evaluated? (continued)

```
int result = 12 + (4 % 3 * 7 / 8);
```

throw expression

Lets you insert parentheses around the throw expression to treat the statement like a function call.

Since 1.6

Example 2.22. Throw statement

```
if ( condition )
    throw new IllegalStateException();
```

Example 2.23. Throw statement with parentheses

```
if ( condition )
    throw ( new IllegalStateException() );
```

return expression

Lets you insert parentheses around the return expression to treat the statement like a function call.

Since 1.6

Example 2.24. return statement

```
if ( condition)
    return true;
```

Example 2.25. return statement with parentheses

```
if ( condition)
    return ( true );
```

2.8.4 Miscellaneous

Only format Javadoc comments

When enabled, only Javadoc comments are formatted according to the current Javadoc settings. Any surrounding source code is left untouched. When you enable this option, the GUI switches its mode and hides all non-Javadoc related options. In order to display the full set of options again, you have to disable the Javadoc-only option. You can control the style of Javadoc comments through the Javadoc settings pages.

Since 1.8

Array brackets after identifier

Lets you choose where the brackets of array types should be placed. By default, Jalopy prints the square brackets right after the array type.

Example 2.26. Array brackets after type

```
int[] a;
```

But C/C++ programmers may expect them to appear after the identifier instead.

Example 2.27. Array brackets after identifier

```
int a[];
```

Note that Java allows some strange freedom in the way arrays can be defined. Array brackets may not only appear after either the type or an identifier, but a mixed style is also allowed

(though not recommended). Jalopy handles all styles well, but is only able to move the brackets if the dimension of all array declarators is equal.

Example 2.28. Mixed array notation with equal dimensions

```
float[] f[][], g[][], h[][];
```

Jalopy would print the above example as

Example 2.29. Mixed array notation with equal dimensions after formatting

```
float[][][] f, g, h;             // print brackets after type

float f[][][], g[][][], h[][][]; // print brackets after identifier
```

Mixed array declarators with different dimensions will be printed *as-is*.

Example 2.30. Mixed array notation with different dimensions

```
float[][] f[][], g[][][], h[];
```

Split multi-variables

When enabled, multi-variables are split into several variable declarations. Otherwise multi-variables are kept and printed according to the current settings.

Since 1.0.1

Example 2.31. Multi-variable

```
BigInteger q = null, p = null, g = null;
```

Example 2.32. Splitted multi-variable

```
BigInteger q = null;
BigInteger p = null;
BigInteger g = null;
```

Remove redundant modifiers

The Java Language specification allows certain modifiers that are redundant and should be avoided. Enabling this option will ensure that these modifiers are removed where present. The modifiers that will be removed are:

- the `abstract` modifier of interface declarations (see Java Language specification, section 9.1.1).

  ```
  public abstract interface Fooable { }
  ```

- the `abstract` and `public` modifiers of method declarations in interfaces (see the Java Language specification, section 9.4).

  ```
  public interface Fooable {
      public abstract reportFoo();
  }
  ```

- the `final` modifier of method declarations that are either declared `private` or members of class or enum declarations that are declared `final` (see the Java Language specification, section 8.4.3.3).

  ```
  public class Foo {
      private final performFooOperation() { }
  }
  ```

```
public final class AnotherFoo {
    public final performAnotherFooOperation() { }
}
```

- the `public`, `static` and `final` modifiers of field declarations in interfaces (see the Java Language specification, section 9.3).

```
public interface Foo {
    public static final int FOO_CONSTANT = 1;
}
```

Since 1.5

2.8.5 Code Generation

Insert serial version UID

Common sense dictates to declare an explicit serial version UID in every serializable class to eliminate the serial version UID as a potential source of incompatibility (with the additional benefit of a small performance gain). When this option is enabled and the class directly derives from either `java.io.Serializable` or `java.io.Externalizable`, Jalopy inserts a serial version UID for the class.

Example 2.33. Serial version UID

```
import java.io.Serializable;

public class Coordinate implements Serializable {
    private final static long serialVersionUID = -8973068452784520619L;

    ...
}
```

You can choose whether you want to have a default serial version UID added or whether the serial version id should be generated from the actual class content. Please note that when choosing the latter, you need to make sure that the source file has been compiled before formatting is applied, because here the serial version UID is computed from the byte code.

Insert @Override annotation

When enabled, formatting automatically adds the @Override marker annotation for methods that override a method from a superclass. The @Override annotation (introduced with J2SE 5.0) greatly reduces the chance of accidentally overloading when you really want to override. The @Override annotation tells the compiler that you intend to override a method from a superclass. If you don't get the parameter list quite right so that you're really overloading the method name, the compiler emits a compile-time error. It's therefore good practice to always use the annotation when you override a method to eliminate potential bugs.

Since 1.8

Example 2.34. @Override annotation

```
public class Parent {
  int i = 0;
  void doSomething (int k) {
    i = k;
  }
}

class Child extends Parent {
  @Override
  void doSomething (long k) {
    i = 2 * k;
  }
}
```

This option is only available if the Java compliance level is set to J2SE 5.0 or higher. The chosen compliance level affects the scope of the annotation. With J2SE 5.0, annotations may only be inserted for methods that override a method in a super class. But since Java SE 6.0, the override annotation is also allowed to annotate methods that implement an interface. Depending on the chosen compliance level, Jalopy therefore performs the corresponding action.

Insert logging conditional

Typically, logging systems have a method that submits a logging message like

```
logger.debug("some message: " + someVar);
```

This is fine, but if the debug level is set such that this message will *NOT* display, then time is wasted doing the string marshalling. Thus, the preferred way to do this is

```
if (logger.isDebugEnabled()) {
    logger.debug("some message: " + someVar);
}
```

which will only use CPU time if the log message is needed. Enabling this switch will ensure that every logging call with the debug level set will be enclosed with the conditional expression.

Use this feature with care! The current implementation only supports the Jakarta Log4J toolkit and is somewhat weak in that every method call named *debug* is treated as a logging call which could be incorrect in your application. However, it works fine for the *l7dlog* calls.

Insert implicit constructor

If you don't define a constructor for a class, a default parameterless constructor is automatically created by the compiler. To make this provision more visible, you can let Jalopy insert the implicit constructor automatically for top-level classes.

Since 1.9.3

Example 2.35. Class without constructor

```
public class Watch {
}
```

Example 2.36. Class with default constructor

```
public class Watch {
    public Watch() {
    }
}
```

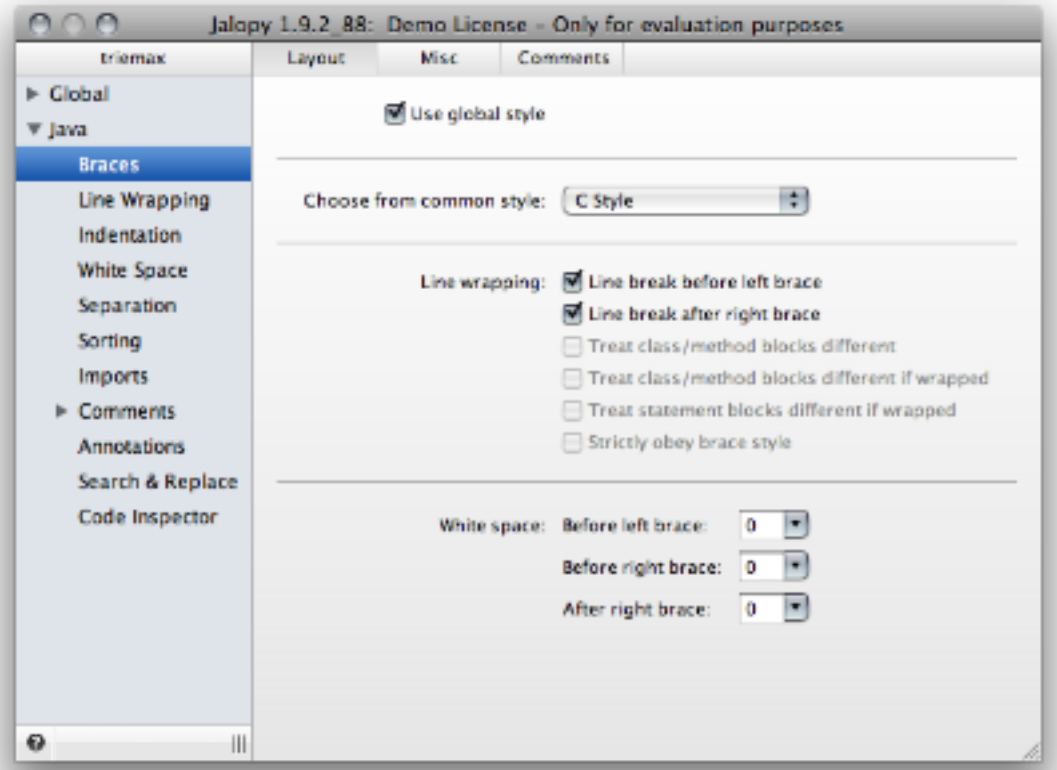
2.8.6 Braces

Controls the handling of curly braces (the Java block delimiters).

2.8.6.1 Layout

Lets you define how the enclosing block delimiters—open and closing curly brace—are printed. You can either choose from a predefined set of common styles or build one on your own. The brace style can either be configured individually for each block construct (details view) or one global style used (global view).

Figure 2.34. Braces Layout settings page (Global view)

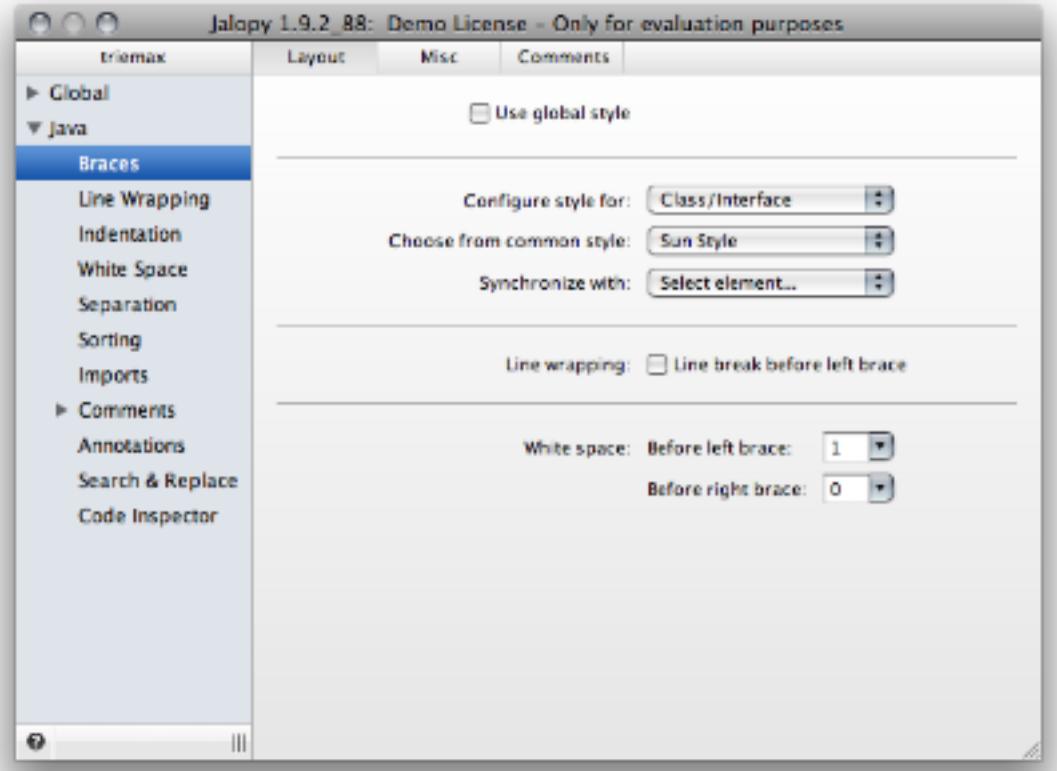


Use global style

Defines whether one brace style should be used for all block elements. When disabled, a combo box appears that lets you define the desired brace style for the different block elements individually.

Since 1.6

Figure 2.35. Braces Layout settings page (Details view)



Configure style for

Lets you choose the block construct whose brace style should be configured. Select an item to display the available brace style options for the chosen block construct. This option is only available when the global brace style check box is disabled.

Since 1.6

Choose from common style

Lets you choose a pre-configured brace style. Choosing a style will adjust all individual brace style options accordingly. The available styles are:

- C style.

  Activates the C brace style. This style is sometimes called "Allman style" or "BSD style" named for Eric Allman, a Berkeley hacker who wrote a lot of the BSD utilities in it. The style puts the brace associated with a control statement on the next line, indented to the same level as the control statement. Statements within the braces are indented to the next level.

Example 2.37. C style

```
private void routine()
{
    if (!done)
    {
        doSomething();
    }
    else
    {
        System.out.println("Finished");
    }
}
```

Advantages of this style are that the indented code is clearly set apart from the containing statement by lines that are almost completely whitespace, improving readability and the ending brace lines up in the same column as the beginning brace, making it easy to find the matching brace. Additionally, the blocking style delineates the actual block of code associated from the control statement itself. Commenting out the control statement, removing the control statement entirely, refactoring, or removing of the block of code is less apt to introduce syntax errors because of dangling or missing brackets. Proponents of this style often cite its use by ANSI and in other standards as justification for its adoption. The motivation of this style is to promote code readability through visually separating blocks from their control statements, deeming screen real estate a secondary concern.

- Sun Java style.

  Activates the Sun brace style, a variant of the so called "K&R style" named after Kernighan & Ritchie, because the examples in their famous "The C Programming Language" [Kernighan88] reference are formatted this way. It keeps the first opening brace on the same line as the control statement, indents the statements within the braces, and puts the closing brace on the same indentation level as the control statement (on a line of its own). Most of the standard source code for the Java API is written in this style.

Example 2.38. Sun style

```
private void routine() {
    if (!done) {
        doSomething();
    } else {
        System.out.println("Finished");
    }
}
```

Advantages of this style are that the beginning brace does not require an extra line by itself, and the ending brace lines up with the statement it conceptually belongs to. The big drawback here is that for block statements wrapped expressions require extra indentation because otherwise the block body is difficult to gather which violates symmetry.

- GNU style.

  Activates the GNU brace style. A brace style used throughout GNU EMACS and the Free Software Foundation code. Braces are put on a line by themselves. The braces are indented by 2 spaces, and the contained code is indented by a further 2 spaces.

Example 2.39. GNU style

```
private void routine()
  {
    if (!done)
      {
        doSomething();
      }
    else
      {
        System.out.println("Finished");
      }
  }
```

Synchronize with

Lets you synchronize the brace style of the currently selected block construct with the brace style of another one. Select an item in the list to have the brace style options updated accordingly. This option is only available when the global brace style check box is disabled.

Since 1.6

Line Wrapping

Lets you manually adjust the line wrapping behavior for braces. If one of the common brace styles does not satisfy your needs, you can manually adjust the appearance here.

Line break before left brace

Controls the line break behavior before the left curly brace. When enabled, a line break gets printed before the left curly brace.

Example 2.40. No line break before left brace

```
if(true){
    doSomething();
}
```

Example 2.41. Line break before left brace

```
if(true)
{
    doSomething();
}
```

Line break after right brace

Controls the line break behavior after the left curly brace. When enabled, a line break gets printed after the left curly brace whenever possible.

Example 2.42. No line break after right brace

```
try{
    doSomething();
}catch (Exception ex){
}
```

Example 2.43. Line break after right brace

```
try{
    doSomething();
}
catch (Exception ex){
}
```

Treat class/method blocks different

It is common in the Java developer community to have the opening brace at the end of the line of the keyword for all types of blocks (Sun brace style). One may find the C++ convention of treating class/interface and method/constructor blocks different from other blocks useful (K&R style). With this option you can achieve exactly that: if enabled, class/interface and method/constructor blocks are then always printed in C brace style (line break before left brace).

Example 2.44. Sun brace style

```
class VolkswagenBeetle extends AbstractAutomobile {
    public void tootHorn() {
        if (isNull) {
            throwConstraintViolated();
        } else {
            updateValue();
        }
    }
}
```

Example 2.45. Sun brace style, but class/method block treat different

```
class VolkswagenBeetle extends AbstractAutomobile
{
    public void tootHorn()
    {
        if (isNull) {
            throwConstraintViolated();
        } else {
            updateValue();
        }
    }
}
```

This option is only available when the global brace style check box is enabled.

Treat class/method blocks different if wrapped

Enabling this option forces a line break before the left brace for class/interface or method/constructor blocks (C style), if either the parameter list spawns several lines and a `throws` clause follows, or one of the possible clauses (`extends`, `implements`, `throws`) was wrapped. This option is only available when the global brace style check box is enabled.

Treat statement blocks different if wrapped

Using the Sun brace style can make it hard to identify the start of the block body. The recommended workaround is to increase indentation for the expression statements during line wrap (see "Block continuation indentation"). But it may be easier to change the style of the block brace for such cases. Enabling this option does exactly that: It forces a line break before the opening brace for block statements (if/for/while etc.), if the statement expression could not be printed in just one line.

Since 1.7

Example 2.46. Wrapped block statement (Sun brace style)

```
if (((x > 0) && (y > 0)) || ((x < 0) && (y < 0))
    && ((x > y) || (x < -y))) {
    reverseCourse();
}
```

Example 2.47. Wrapped block statement - left curly brace treated different

```
if (((x > 0) && (y > 0)) || ((x < 0) && (y < 0))
    && ((x > y) || (x < -y)))
{
    reverseCourse();
}
```

This option is only available when the global brace style check box is enabled or the style for *Statements* is currently configured.

Strictly obey brace style

When "Prefer wrap after assignments" and Section 2.8.6.1.1, "Line break before left brace" are enabled, a line break may be printed before the left curly brace of array initializers if the initializer does not fit into one line. But one might prefer to leave the curly brace in the same line as the variable declaration to obey the general brace style. With this option you can decide what you favor most: consistent brace style or consistent wrapping after assignments. The option is only available, if line breaks should be printed before left curly braces (see Section 2.8.6.1.1, "Line break before left brace") and either a global brace style (see Section 2.8.6.1, "Use global style") used or the brace style for arrays (see Section 2.8.6.1, "Configure style for") is configured.

Since 1.8

Example 2.48. Favor consistent brace style

```
private String[] data = {
        "aaaaaaaaaaaaaaaaaasssssssssssssssssssss",
        "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb",
        "ccccccccccccccccccccccccccccccccccccccc",
        "ddddddddddddddddddddddddddddddddd",
        "eeeeeeeeeeeeeeeeeeeeeeeeeee", "fffffffffffffffffffffffff"
    };
```

Example 2.49. Favor wrap after assignment

```
private String[] data =
    {
        "aaaaaaaaaaaaaaaaaasssssssssssssssssssss",
        "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb",
        "ccccccccccccccccccccccccccccccccccccccc",
        "ddddddddddddddddddddddddddddddddd",
        "eeeeeeeeeeeeeeeeeeeeeeeeeee", "fffffffffffffffffffffffff"
    };
```

White Space

Lets you adjust the indentation white space for curly braces individually.

Before left brace

Defines the amount of blank space that should be printed before the left curly brace.

Example 2.50. No white space before left brace

```
if(true){
    doSomething();
}
```

Example 2.51. One space before left brace

```
if(true) {
    doSomething();
}
```

Before right brace

Defines the amount of blank space that should be printed before the right curly brace.

Example 2.52. No white space before right brace

```
if(true){
    doSomething();
}else{
    quit()();
}
```

Example 2.53. One space before right brace

```
if(true){
    doSomething();
}else {
    quit()();
}
```

After right brace

Defines the amount of blank space that should be printed after the right curly brace.

Example 2.54. No white space after right brace

```
if(true){
    doSomething();
}else{
    quit()();
}
```

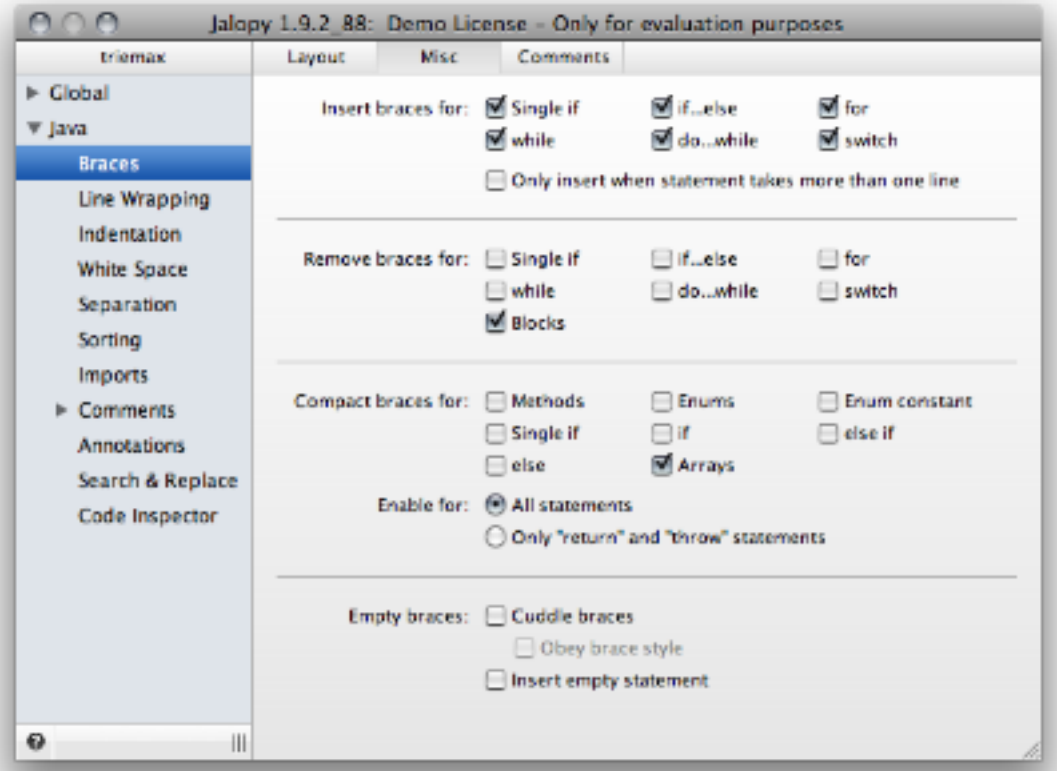
Example 2.55. One space after right brace

```
if(true){
    doSomething();
} else{
    quit()();
}
```

2.8.6.2 Misc

Controls miscellaneous brace options.

Figure 2.36. Braces Misc settings page



Insert braces for

Per definition braces are superfluous on single statements, but it is a common recommendation that braces should be always used in such cases. With this option, you can specify whether missing braces for single statements should be inserted for the control statements `if`, `for`, `while` and `do-while` and labeled statements inside `switch` blocks. Inserting and removing braces is mutually exclusive.

Single if

When enabled, braces are inserted around the body of single `if` statements when necessary.

Since 1.9.1

Example 2.56. Brace insertion for if statement

```
if (condition)
    break;
```

would become

```
if (condition) {
    break;
}
```

if...else

When enabled, braces are inserted around the body of `if...else` statements when necessary.

Example 2.57. Brace insertion for if-else statement

```
if (true)
    break;
else
    return;
```

would become

```
if (true) {
    break;
} else {
    return;
}
```

for

When enabled, braces are inserted around the body of `for` statements when necessary.

Example 2.58. Brace insertion of for statements

```
for (int i = 0; i < count; i++)
    System.out.println(i);
```

would become

```
for (int i = 0; i < count; i++) {
    System.out.println(i);
}
```

while

When enabled, braces are inserted around the body of `while` statements when necessary.

Example 2.59. Brace insertion for while statements

```
while (!isDone)
    doSomething();
```

would become

```
while (!isDone) {
    doSomething();
}
```

do...while

When enabled, braces are inserted around the body of `do...while` statements when necessary.

Example 2.60. Brace insertion for do...while statements

```
do
  something();
while (condition);
```

would become

```
do {
  something();
} while (condition);
```

switch

When enabled, braces are inserted around the body of labeled statements inside `switch` blocks when necessary. Braces are only inserted if the statement is not empty.

Since 1.4

Example 2.61. Brace insertion for labeled statements

```
switch (c) {
    case 'a':
    case 'b':
        System.out.println();
        break;
}
```

would become

```
switch (c) {
    case 'a':
    case 'b': {
        System.out.println();
        break;
    }
}
```

Only insert when statement takes more than one line

When enabled, brace insertion only happens when the block statement takes more than just one line to print.

Since 1.8

Example 2.62. Missing braces

```
if (arg == null)
    for (int i = 0; i < 10; i++)
        System.out.println("arg " + i);
```

Above you see an example with two block statements. Enabling brace insertion for if and for statements would yield:

Example 2.63. Inserted braces

```
if (arg == null) {
    for (int i = 0; i < 10; i++) {
        System.out.println("arg " + i);
    }
}
```

You see braces inserted for both block statements. But when you've enabled the multi-line option, you will get:

Example 2.64. Inserted braces limited to multi-line statements

```
if (arg == null) {
    for (int i = 0; i < 10; i++)
        System.out.println("arg " + i);
}
```

The statement of the if-block happens to be another block statement which is printed in two lines here, therefore the braces are inserted for the if-statement. The for-statement on the other hand does not have any braces inserted, because here the block can be printed in just one line.

Remove braces

It is permissible to remove braces in case they are superfluous. This not only applies to the control statements `if`, `for`, `while` and `do-while`, but also to every block in general

(remember a block is just a sequence of statements, local class declarations and local variable declaration statements within braces). Inserting and removing braces is mutually exclusive.

Single if

When enabled, braces around the body of single `if` statements are removed when possible.

Since 1.9.1

Example 2.65. Brace removal for single if statements

```
if (true) {
    break;
}
```

would become

```
if (true)
    break;
```

if..else

When enabled, braces around the body of `if...else` statements are removed when possible.

Example 2.66. Brace removal for if statements

```
if (true) {
    break;
} else {
    return;
}
```

would become

```
if (true)
    break;
else
    return;
```

for

When enabled, braces around the body of `for` statements are removed when possible.

Example 2.67. Brace removal of for statements

```
for (int i = 0; i < count; i++) {
    System.out.println(i);
}
```

would become

```
for (int i = 0; i < count; i++)
    System.out.println(i);
```

while

When enabled, braces around the body of `while` statements are removed when possible.

Example 2.68. Brace removal for while statements

```
while (!isDone) {
    doSomething();
}
```

would become

```
while (!isDone)
    doSomething();
```

do...while

When enabled, braces around the body of `do...while` statements are removed when possible.

Example 2.69. Brace removal for do...while statements

```
do {
  something();
} while (condition);
```

would become

```
do
  something();
while (condition);
```

switch

When enabled, braces are removed around the body of labeled statements inside switch blocks when possible.

Since 1.8

Example 2.70. Brace removal for switch

```
switch (c) {
    case 'a':
    case 'b': {
        System.out.println();
        break;
    }
}
```

would become

```
switch (c) {
    case 'a':
    case 'b':
        System.out.println();
        break;
}
```

Blocks

When enabled, arbitrary block braces are removed when possible.

Example 2.71. Brace removal for blocks

```
{
    System.out.println();
}
```

would become

```
System.out.println();
```

Compact braces

Allows you to print statement blocks in just one line when possible, i.e. when the block only contains one statement and does not exceed the maximal line length. Please note that

white space before a compacted block is not controlled by your general brace settings, but with the "Space before braces" option.

Methods

When enabled, method and constructor bodies will be printed in one line when possible.

Since 1.2

Example 2.72. Standard method declaration

```
public int getMagicNumber() {                                    |
    return 23;                                                   |
}                                                                |
```

Example 2.73. Compact method declaration

```
public int getMagicNumber() { return 23; }                       |
```

Single if

When enabled, single `if` statement bodies will be printed in one line when possible.

Since 1.2

Example 2.74. Standard if block

```
if (cond) {                                  |
    return getMagicNumber();                 |
}                                            |
```

Example 2.75. Compact if block

```
if (cond) { return getMagicNumber(); }       |
```

if

When enabled, statement bodies of the `if` the part of if/else statement will be printed in one line when possible.

Since 1.2

Example 2.76. Standard if/else block

```
if (cond) {                                  |
    return getMagicNumber();                 |
} else {                                     |
    return -1                                |
}
```

Example 2.77. Compact if of if/else block

```
if (cond) { return getMagicNumber(); }       |
else {                                       |
    return -1;                               |
}                                            |
```

Please note that when the `if` block has been compacted, the following `else if` or `else` statement always starts on a new line.

else if

When enabled, prints `else if` statement bodies of if/else statements in one line when possible.

Example 2.78. Standard else of if/else block

```
if (cond1) {                                                    |
    return getMagicNumber();                                    |
} else if (cond2) {                                             |
    throw new IllegalStateException();                          |
}                                                               |
```

Example 2.79. Compact else if block

```
if (cond1) {                                                    |
    return getMagicNumber();                                    |
} else if (cond2) { throw new IllegalStateException(); }        |
```

Please note that when the `else if` block has been compacted, the following `else` statement always starts on a new line.

Since 1.2

Example 2.80. Compact if block

```
if (cond1) {                                                    |
    return getMagicNumber();                                    |
} else if (cond2) { throw new IllegalStateException(); }        |
else {                                                          |
    return 0;                                                   |
}                                                               |
```

else

When enabled, prints `else` statement bodies in one line when possible.

Since 1.2

Example 2.81. Standard else block

```
if (cond1) {                                                    |
    return getMagicNumber();                                    |
} else if (cond2) {                                             |
    throw new IllegalStateException();                          |
} else {                                                        |
    return 0                                                    |
}                                                               |
```

Example 2.82. Compact else block

```
if (cond1) {                                                    |
    return getMagicNumber();                                    |
} else if (cond2) {                                             |
    throw new IllegalStateException();                          |
} else { return 0; }                                            |
```

Arrays

When enabled, array initializers will be printed in one line when possible.

Since 1.7

Example 2.83. Array initializer

```
String[] s = {                              |
        "First"                             |
    };                                      |
```

Example 2.84. Compact array initializer

```
String[] s = { "First" };                   |
```

Enums

When enabled, enum declaration bodies will be printed in one line when possible.

Since 1.4

Example 2.85. Enum declaration

```
public enum Mode {                                          |
    OPEN, CLOSE                                             |
}                                                           |
```

Example 2.86. Compact enum declaration

```
public enum Mode { OPEN, CLOSE }                            |
```

Enum constants

When enabled, enum constants will be printed in one line when possible. This option is only useful if you use *constant-specific* methods in your enums.

Since 1.4

Example 2.87. Enum constants with constant specific methods

```
public enum Operation {                                                 |
    PLUS {                                                              |
        double eval(double x, double y) {                               |
            return x + y;                                               |
        }                                                               |
    },                                                                  |
    MINUS {                                                             |
        double eval(double x, double y) {                               |
            return x - y; }                                             |
    },                                                                  |
    TIMES {                                                             |
        double eval(double x, double y) {                               |
            return x * y;                                               |
        }                                                               |
    },                                                                  |
                                                                        |
    // Do arithmetic op represented by this constant                    |
    abstract double eval(double x, double y);                           |
}                                                                       |
```

Example 2.88. Compact enum constants with constant specific methods

```
public enum Operation {                                                 |
    PLUS { double eval(double x, double y) { return x + y; } },         |
    MINUS { double eval(double x, double y) { return x - y; } },        |
    TIMES { double eval(double x, double y) { return x * y; } },        |
                                                                        |
    // Do arithmetic op represented by this constant                    |
    abstract double eval(double x, double y);                           |
}                                                                       |
```

Enable for

You can narrow the scope for the above mentioned options by selecting whether all statements should be printed on one line when possible or only `throw` and `return` statements.

Since 1.2

All statements

When enabled, all statements are compacted when possible.

Example 2.89. All statements on same line

```
if (true) { return result; }
else if (false) { System.out.println("unexpected condition"); }
else { throw new Error(); }
```

Only throw and return

When enabled, only throw and return statements are compacted when possible. All other statements are printed in normal block style.

Example 2.90. Only throw and return statements on same line

```
if (true) { return result; }
else if (false) {
    System.out.println("unexpected condition");
} else { throw new Error(); }
```

Empty braces

Controls how empty braces should be printed.

Cuddle braces

Prints both braces on one line right after the declaration or statement.

Example 2.91. Cuddled empty braces

```
class Foo {

    public void foo() { }
}
```

You can control the amount of white space that is printed before the left curly brace. See "Cuddled braces indent" for more information.

Obey brace style

Causes the left curly brace of the empty brace block to be positioned according to the current brace style settings. Depending on the style, the left brace is either printed directly after an element or will have a line break printed before. This option is only available when Section 2.8.6.2.4, "Cuddle braces" is enabled.

Since 1.7

Example 2.92. Cuddled braces, C brace style

```
class Foo
{

    public void foo() { }
}
```

Example 2.93. Cuddled braces, C brace style, obey brace style

```
class Foo
{

    public void foo()
    { }
}
```

Insert empty statement

Inserts an empty statement to make it obvious for the reader that the empty braces are intentional. Please note that this option does not apply for class/interface and method/constructor bodies, but is only used for control statements and blocks.

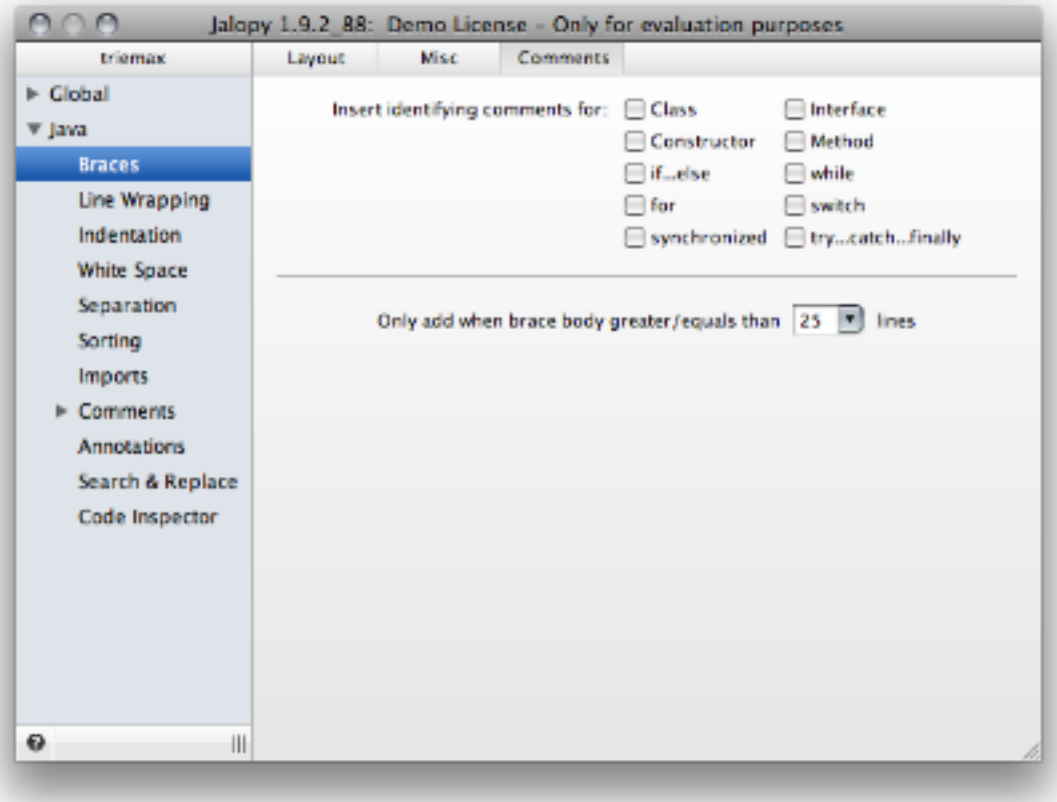
Example 2.94. Empty braces with empty statement

```
class Foo {

    public void foo() {
        ;
    }
}
```

2.8.6.3 Comments

Lets you control the insertion of trailing comments after a closing brace to assist matching corresponding braces. Such comments are called *identifying comments*.

Figure 2.37. Braces Comments settings page



Insert identifying comments for

Class

Lets you insert identifying comments for class declarations.

Since 1.3

Example 2.95. Identifying class comment

```
public class Foo {

    public String getName() {
        return ...;
    }

    ...
} // end class Foo
```

Interface

Lets you enable the insertion of identifying comments for interface declarations.

Since 1.3

Example 2.96. Identifying interface comment

```
public interface Fooable {

    public String getName();

    ...
} // end interface Fooable
```

Constructor

Lets you enable the insertion of identifying comments for constructor declarations.

Since 1.3

Example 2.97. Identifying constructor comment

```
public class Foo {

    public Foo() {
        super();
    } // end ctor Foo

    ...
}
```

Method

Lets you enable the insertion of identifying comments for method declarations.

Since 1.3

Example 2.98. Identifying method comment

```
public class Foo {

    public void getName() {
        return _name;
    } // end method getName

    ...
}
```

if...else

Lets you enable the insertion of identifying comments for `if/else` statement blocks.

Since 1.3

Example 2.99. Identifying if-else comment

```
if (true) {
    ...
} // end if


if (true) {
    ...
} else if (false) {
    ...
} // end if-else


if (true) {
    ...
} else if (false) {
    ...
} else {
    ...
} // end if-else
```

while

Lets you enable the insertion of identifying comments for `while` statement blocks.

Since 1.3

Example 2.100. Identifying while comment

```
while (true) {
    ...
} // end while
```

for

Lets you enable the insertion of identifying comments for `for` statement blocks.

Since 1.3

Example 2.101. Identifying for comment

```
for (int i = 0; object.length; i++) {
    ...
} // end for
```

switch

Lets you enable the insertion of identifying comments for `switch` statement blocks.

Since 1.8

Example 2.102. Identifying switch comment

```
switch (state) {
    ...
} // end switch
```

try...catch...finally

Lets you enable the insertion of identifying comments for `try/catch/finally` blocks.

Since 1.3

Example 2.103. Identifying try-catch comment

```
try {
    ...
} catch (Throwable ex) {
    ...
} // end try-catch


try {
    ...
} catch (IOException ex) {
    ...
} finally {
    ...
} // end try-catch-finally


try {
    ...
} finally {
    ...
} // end try-finally
```

synchronized

Lets you enable the insertion of identifying comments for `synchronized` statement blocks.

Since 1.3

Example 2.104. Identifying synchronized comment

```
synchronized (this) {
    ...
} // end synchronized
```

Only add when brace body greater/equal than n lines

Lets you specify the size of a brace body that is necessary in order to see identifying comments inserted. For example, you might want to require identifying comments only on brace bodies that have at least 30 lines.

Since 1.3

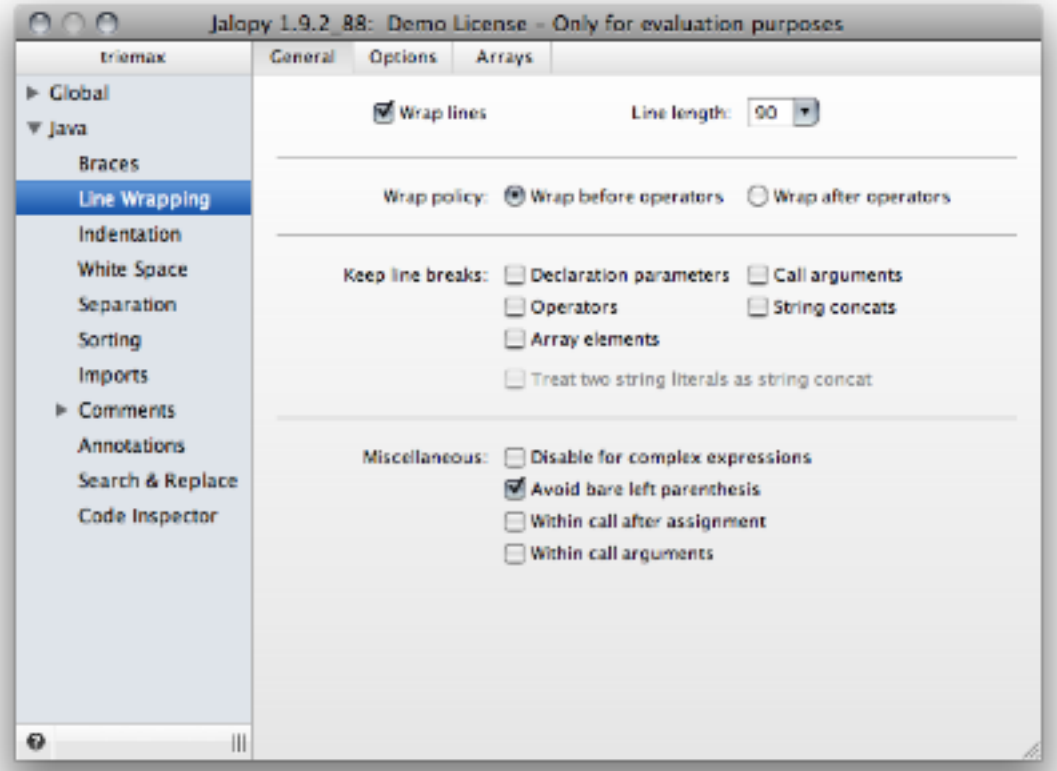
2.8.7 Line Wrapping

Controls when and how lines are wrapped.

2.8.7.1 General

Lets you control the general line wrapping options.

Figure 2.38. Wrapping settings page



Wrap lines

Enables or disables automatic line wrapping. When enabled, Jalopy tries to keep lines within the maximal line length and breaks statements across lines when necessary.

NOTE Disabling line wrapping does not mean that existing line breaks are kept, but rather that no effort is taken to keep lines in between the maximal line length upon reformatting

Line length

Lets you specify the maximal line length. Jalopy tries (more or less—depending on the used indentation scheme) to limit each line within the given length.

Policy

Lets you define the wrapping policy for operators. Line wrapping will often occur with statements that consist of several (possibly long) expressions. Here you specify whether line wrapping should occur before or after the expression operator.

Wrap before operators

When enabled, line breaks will be inserted before operators.

Example 2.105. Wrap before operators (Standard indented)

```
if ((condition1 && condition2)
    || (condition3 && condition4)
    || !(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Wrap after operators

When enabled, line breaks will be inserted after operators.

Example 2.106. Wrap after operators (Standard indented)

```
if ((condition1 && condition2) ||
    (condition3 && condition4) ||
    !(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Please note that wrapping for the comma and dot operator is currently always performed after the operators! If you happen to use Sun Brace styling, you might want to enable continuation indentation for blocks to let the statement body stand out. See "Block continuation indentation" for more information.

Keep line breaks

Lets you specify for which code elements line breaks should be kept.

Declaration parameters

When enabled, existing line breaks after the commas of declaration parameters are kept. Otherwise line wrapping is performed according to the current settings.

Since 1.6

Example 2.107. Nicely laid out method declaration

```
void test( String rName,
           boolean rPendandic ) {
    ...
}
```

After formatting this code could look like this (because everything fits in one line):

Example 2.108. Method call after formatting

```
void test( String rName, boolean rPendandic ) {
    ...
}
```

But with the "Keep line breaks" option enabled, it may look like this:

Example 2.109. Method declaration after formatting with kept line breaks (Endline indented)

```
void test( String rName,
           boolean rPendandic ) {
    ...
}
```

Alternatively, you might want to use the `//J:KEEP+` pragma comment to keep line breaks on a case by case basis.

Call arguments

When enabled, existing line breaks after the commas of call arguments are kept. Otherwise line wrapping is performed according to the current settings.

Since 1.6

Example 2.110. Nicely laid out method call

```
obj.method1( test,
             test2,
             test3 );
```

After formatting this code could look like this (because everything fits in one line):

Example 2.111. Method call after formatting

```
obj.method1( test, test2, test3 );
```

But with the "Keep line breaks" option enabled, it may look like this:

Example 2.112. Method call after formatting with kept line breaks (Endline indented)

```
obj.method1( test,
             test2,
             test3 );
```

Alternatively, you might want to use the `//J:KEEP+` pragma comment to keep line breaks on a case by case basis.

Operators

When enabled, existing line breaks before or after infix operators and the comma operator of method declaration parameters or method call arguments are kept. Otherwise line wrapping is performed according to the current settings.

Since 1.0

Example 2.113. Operators with forced line breaks

```
if ((condition1 && condition2)
    || (condition3 && condition4)
    || !(condition5 && condition6))
{
    ...
}
```

After formatting this code could look like this (because not everything fits in one line and not line breaks are forced):

Example 2.114. Operators after formatting (wrapping is done on-demand)

```
if ((condition1 && condition2) || (condition3 && condition4) ||
    !(condition5 && condition6))
{
    ...
}
```

But with the "Keep line breaks" option enabled, it may look like this:

Example 2.115. Operators after formatting with kept line breaks

```
if ((condition1 && condition2) ||
    (condition3 && condition4) ||
    !(condition5 && condition6))
{
    ...
}
```

Please note that it does not matter what wrapping policy for operators you choose. Jalopy will keep line breaks even if the operators move!

String concats

When enabled, existing line breaks before or after the plus operator of concatenated string literals are kept. Otherwise line wrapping is performed according to the current settings.

Since 1.0.1

Example 2.116. Nicely laid out string constant

```
query = "select a.prop_text, "
        + "       a.contest_title, "
        + "  from contest a "
        + "  where a.language_code = b.language_code "
        + "    and b.bob = c.bob "
        + "    and a.x = ? "
        + "  order by a.bob, "
        + "           c.language";
```

After formatting this code could look like this:

Example 2.117. String constant after formatting

```
query = "select a.prop_text, " + "       a.contest_title, " +
    "  from contest a "  + "  where a.language_code = b.language_code " +
    "    and b.bob = c.bob " + "    and a.x = ? " + "  order by a.bob, " +
    "           c.language";
```

But with the "Keep string concats" option enabled, it may look like this:

Example 2.118. String constant after formatting with "Keep line breaks" (Standard indented)

```
query = "select a.prop_text, " +
    "       a.contest_title, " +
    "  from contest a " +
    "  where a.language_code = b.language_code " +
    "    and b.bob = c.bob " +
    "    and a.x = ? " +
    "  order by a.bob, " +
    "           c.language";
```

Please note that it does not matter what wrapping policy for operators you choose. Jalopy will keep line breaks even if the operators move!

Array elements

When enabled, existing line breaks after the separator comma between individual array elements are kept. Otherwise line wrapping is performed according to the current settings if there can be no pragma comment found that indicates otherwise.

Since 1.5

Example 2.119. Array declarations

```
String[] foo = new String[]
    {
        "foo",
    };

String[] bar = new String[]
    {
        "bar",
        "car",
    };
```

Example 2.120. Array declarations after reformat

```
String[] foo = new String[] { "foo", };

String[] bar = new String[] { "bar", "car", };
```

Example 2.121. Array declarations after reformat with "Keep line breaks"

```
String[] foo = new String[]
    {
        "foo",
    };

String[] bar = new String[]
    {
        "bar",
        "car",
    };
```

Example 2.122. Array declarations with pragma comment to keep line breaks

```
String[] foo = new String[] { "foo", };

String[] bar = new String[]
    { //J:KEEP
        "bar",
        "car",
    };
```

Alternatively, you might want to use the `//J:KEEP+` pragma comment to keep line breaks on a case by case basis.

Strictly obey "Keep line breaks"

When using endline indentation, line breaks may not always be kept because doing so would break the endline indent contract and lead to inconsistent indentation behavior. Enabling this option will cause Jalopy to keep existing line breaks even in such cases. Please note that this option is only available when endline indentation is enabled and any of the "Keep line breaks" options selected!

Since 1.8

Example 2.123. Method call

```
String firstSymbol = eraseNonBreakingSpaceFromSymbol(
    symbol1, symbol2);
```

When the option is disabled, Jalopy won't keep the line break before the first call argument because it goes against the endline indentation rules.

Example 2.124. Method call - endline indented

```
String firstSymbol = eraseNonBreakingSpaceFromSymbol(symbol1,
                                                     symbol2);
```

But if you really favor keeping the line break, enabling the option would yield:

Example 2.125. Method call - line break kept

```
String firstSymbol = eraseNonBreakingSpaceFromSymbol(
        symbol1, symbol2);
```

Treat two string literals as string concatenation

This option lets you control what is considered a string concatenation. By default, Jalopy treats the plus (+) operator as a string concatenation when either one of the operands is a string literal. But you might want to narrow this behavior to only treat two string literals as a string concatenation. Please note that this option is only available when any of the "Keep line breaks" options is selected!

Since 1.8

Example 2.126. String concatenations

```
String query = "select a.prop_text, "
               + "       a.contest_title, "
               + "  from contest a "
               + "  where a.language_code = b.language_code ";

String name = "Walther"
              + getNick();
```

When this option is disabled, all operators in the example above are considered string concatenations and therefore line breaks are kept.

Example 2.127. String concatenations

```
String query = "select a.prop_text, "
               + "       a.contest_title, "
               + "  from contest a "
               + "  where a.language_code = b.language_code ";

String name = "Walther" + getNick();
```

When enabled, only those plus operators with two string literals on either side would retain any line breaks like in the example above.

Miscellaneous

Disable wrapping for complex expressions

For complex expressions a common technique to enhance readability is to break the expression into several sub-expressions that can be stored in temporary variables that are placed on different lines.

Example 2.128. Complex expression

```
if (conditionOne &&                                  |
    ("foo".equals(aStr) || "bar".equals(aStr))       |
  doSomething();                                     |
```

Example 2.129. Refactored expression

```
boolean conditionTwo = "foo".equals(aStr);          |
boolean conditionThree = "bar".equals(aStr);        |
                                                    |
if (conditionOne &&                                 |
    (conditionTwo || conditionThree))               |
  doSomething();                                    |
```

In order to determine occurrences of complex expressions, enabling this option will cause automatic line wrapping to be disabled when a complex expression gets printed. A warning message will be logged in the Printer category that informs you about the location of the expression.

Since 1.5

Example 2.130. Flagged complex expression

```
                                                    |
if (conditionOne && (conditionTwo || conditionThree))
  doSomething();                                    |
```

Avoid bare left parenthesis

With Endline indentation and "Wrap on-demand after left parenthesis" or "Keep line break for operators" enabled, line breaks after left parentheses may look ugly. If this option is enabled, such line breaks are avoided. This option was mainly introduced to fix some unwanted behavior of earlier releases without breaking compatibility. It is recommended to have it enabled.

Since 1.4

Example 2.131. Bare left parenthesis

```
this.customerNumber = new CustomerNumber(
                                        ServiceManager.createService());
```

Example 2.132. Avoided bare left parenthesis

```
this.customerNumber = new CustomerNumber(ServiceManager.createService());
```

Prefer within call after assignment

This option lets you define where a line wrap should preferably occur for call statements after assignments that don't fit into the maximal line length. The option is only meant to let you adjust behavior when "Prefer wrap after assignments" has been enabled. Please see explanation there.

Since 1.7

Example 2.133. Wrapping assignment expression

```
nuMBeans =
     NuvegaPropertiesHandler.getNNuvegaArrayProperty(
         NuvegaProperties.PROPERTIES_FILE_NUVEGA_BEAN_NAME);
```

Example 2.134. Prefer wrapping within call

```
nuMBeans = NuvegaPropertiesHandler.getNNuvegaArrayProperty(
         NuvegaProperties.PROPERTIES_FILE_NUVEGA_BEAN_NAME);
```

Prefer within call arguments

This option lets you define where a line wrap should preferably occur for call arguments that does not fit into the maximal line length. Normally a line gets wrapped before a call argument that don't fit into the maximal line length. Enabling this option will cause a line break inserted within the call argument for operator expressions when the operator expression is not immediately preceded with or followed by another operator expression.

Since 1.5

Example 2.135. Wrapping method call (default behavior)

```
failed(output, "a" + "b", null,                                    |
        "Failed to open prefs: " + e.getMessage());                |
                                                                   |
```

As you can see from the above example a line break is inserted before the third argument as it would exceed the maximal line length when printed in the same line.

Example 2.136. Prefer wrapping within call argument

```
failed(output, "a" + "b", null, "Failed to open prefs: "  |
       + e.getMessage());                                 |
                                                          |
```

When the option is enabled, the line break happens within along the operator of the third argument.

Example 2.137. Standard wrapping when two operator expressions

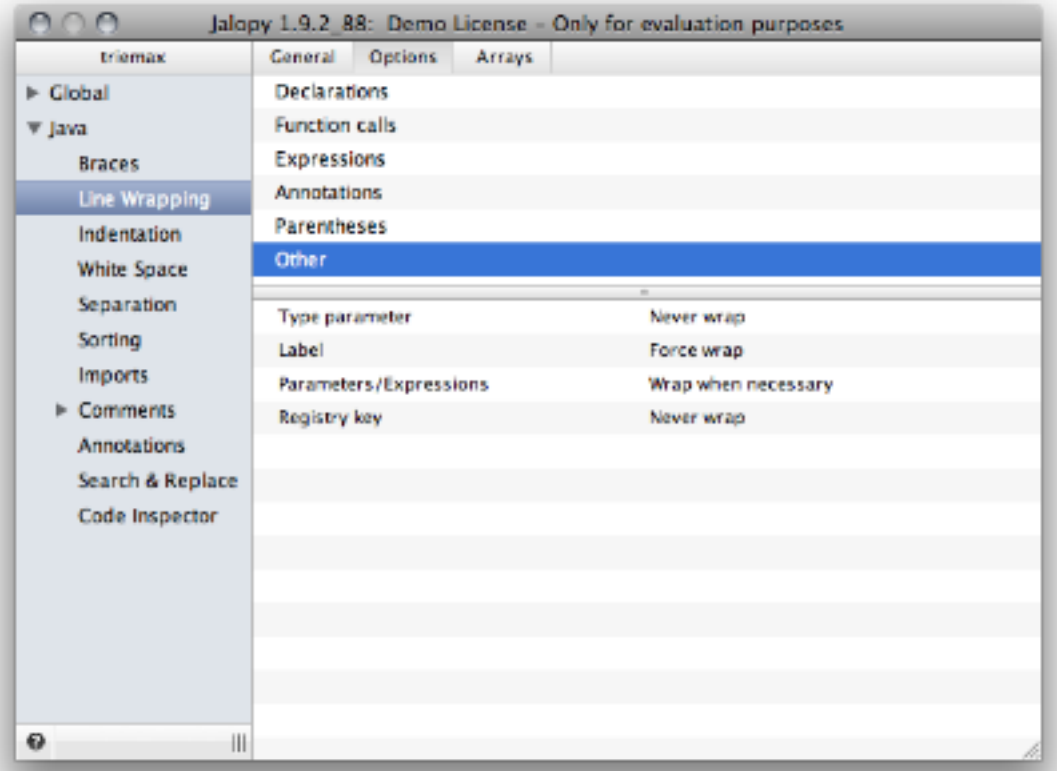
```
failed(output, "a" + "b",                                  |
       "Failed to open prefs: " + e.getMessage());         |
                                                           |
```

But if two operator expressions follow immediately, the standard behavior applies in order to enhance readability.

2.8.7.2 Options

Lets you configure the line wrapping behavior for individual elements.

Figure 2.39. Wrapping options



Jalopy supports different wrapping strategies that can be applied to individual elements as required. To configure wrapping for a specific element, locate the element in the tree and select the current strategy. A pop-up menu will open that lets you change the strategy. The preview instantly reflects any change so you can easily recognize what the impact of a different strategy would be. The current available strategies are:

Never wrap

Disables wrapping for a specific element altogether. Please note that disabling wrapping might lead to long lines that exceed the maximal line length and should therefore preferably only be used with certain elements to avoid line wrapping at specific positions.

Example 2.138. Wrapping disabled for import declaration

```
                                                                |
import com.mycompanyname.myprojectname.mypackagename.MyClassName;
                                                                |
```

Example 2.139. Wrapping enabled for import declaration

```
import com.mycompanyname.myprojectname.mypackagename     |
    .MyClassName;                                        |
```

Wrap when necessary

Only wraps when otherwise the maximal line length limit would be exceeded. One could refer to this policy as lower level or late line wrapping. This strategy favors the use of horizontal space and leads to compact code, but might make certain constructs difficult to read because there is sometimes no clear visual boundary between elements when used exclusively.

Example 2.140. Wrap only when necessary after assignment

```
String value = "xxxxxxxxxxxxxxxxxxxxxx" + "xxxxxxxxxxxxx"         |
    + "xxxxxxxxxxxxx";                                            |
```

Example 2.141. Wrap when exceed after assignment

```
String value =                                                    |
    "xxxxxxxxxxxxxxxxxxxxxxx" + "xxxxxxxxxxxxx" + "xxxxxxxxxxxxx"; |
```

Wrap when exceed

Wraps if an element would not fit completely into the current line. Think of it as higher level or early line wrapping. This is probably the most balanced strategy when available that combines very readable results with moderate vertical space requirements.

Example 2.142. Wrapped within extends clause when necessary

```
interface Testable extends Transferable, Recordable, |
    Playable, Immutable, Serializable {               |
}                                                     |
```

Example 2.143. Wrapped before extends keyword when exceed

```
interface Testable                                    |
    extends Transferable, Recordable, Playable,       |
         Immutable, Serializable {                    |
}                                                     |
```

Wrap all when first wrapped

Forces a line break for all related elements (like declaration parameters) if the first element has been wrapped. This leads to very uniform and readable code at the expense of higher vertical space requirements.

Example 2.144. Wrapped expression when necessary

```
while (                                                       |
    "picker".equals(xxxxxxxxxxxxxxxxxxxxxxxxx.getName()) |
        && m.isStatic() && m.isPublic() && m.isFinal()) { |
   ...                                                        |
}                                                             |
```

Example 2.145. Wrapped all operators as first operand wrapped

```
while (                                                       |
    "picker".equals(xxxxxxxxxxxxxxxxxxxxxxxxx.getName()) |
        && m.isStatic()                                       |
        && m.isPublic()                                       |
        && m.isFinal()) {                                     |
   ...                                                        |
}                                                             |
```

Wrap all when exceed

Similar to *Wrap all when first wrapped*, but forces a line break for all related elements (like declaration parameters), if the whole element would not fit into the current line.

Example 2.146. Wrapped expression when necessary

```
_userDatabase.addUser(                                        |
    "John", "Doo", encryptPassword("password", secretKey),  |
    "123 Nashville", "Surgrass");                             |
```

Example 2.147. Wrapped all operators as first operand wrapped

```
_userDatabase.addUser(                                          |
    "John",                                                     |
    "Doo",                                                      |
    encryptPassword("password", secretKey),                     |
    "123 Nashville",                                            |
    "Surgrass");                                                |
```

Force wrap

Always wraps a specific element or elements (like call arguments). This can be useful to ensure a consistent style e.g. for chained calls or parameter lists, but also to force line breaks at unusual positions that would normally not be considered during line wrapping.

Example 2.148. Method declaration

```
public static int foo() {
    ...
}
```

Example 2.149. Force line break before method name

```
public static int
foo() {
    ...
}
```

Please note that not all strategies apply to all elements, and therefore you might be presented with a different set of strategies for each element. Below you find all currently available elements listed along with some short examples of the output with different wrapping strategies.

Import declaration

For import declarations you can choose whether they should be wrapped along the dots if otherwise the maximal line length would be exceeded or not at all.

Since 1.4

Example 2.150. Very long import declaration

```
                                                               |
import com.mycompanyname.myprojectname.mypackagename.MyClassName;
                                                               |
```

Example 2.151. Wrapped import declaration

```
import com.mycompanyname.myprojectname.mypackagename    |
    .MyClassName;                                       |
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Before declaration keyword

Lets you choose to force a line break before `class`, `interface` `enum` and `@interface` keywords or disable line wrapping completely.

Since 1.4

Example 2.152. Class declaration

```
public class FooBar {
    ...
}
```

Example 2.153. Wrapped class declaration

```
public
class FooBar {
    ...
}
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

After class keyword

Lets you force a line break after the `class` keyword or disable line wrapping altogether.

Since 1.3

Example 2.154. Class declaration

```
public class FooBar {
    ...
}
```

Example 2.155. Wrapped class declaration

```
public class
FooBar {
    ...
}
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Before extends keyword

Lets you either force a line break before the extends keyword of class/interface/enum declarations, only print a line break if the extends clause would not completely fit into one line, or print a line break only if the extends clause would otherwise exceed the maximal line length. Only wrapping when absolutely necessary will require the least vertical space and yield the most compact code, but readability might be affected.

Example 2.156. Wrapping necessary

```
interface InterfaceWithAHugeAndSillyName        |
    extends Amicable, Adorable {                |
    ...                                         |
}                                               |
```

Enforcing a line break before the keyword if the complete clause would not fit into one line, provides a good balance between readability and space requirements.

Example 2.157. Extends clause does not fit into one line

```
                                                |
interface Testable extends Transferable, Recordable,
    Playable {                                  |
    ...                                         |
}                                               |
```

Example 2.158. Line break because extends clause would not fit into one line

```
interface Testable                                      |
    extends Transferable, Recordable, Playable {        |
    ...                                                 |
}                                                       |
```

Always enforcing a line break before the keyword might be a viable strategy to achieve the most consistent behavior at the expense of slightly higher vertical space requirements.

Example 2.159. No line wrapping necessary

```
interface Enyoyable extends Amicable, Adorable {  |
    ...                                            |
}                                                  |
```

Example 2.160. Wrapping forced

```
interface Enyoyable                                |
    extends Amicable, Adorable {                   |
    ...                                            |
}                                                  |
```

You can control the space printed before the keyword via the indentation settings. See "Extends indent size" for more information. For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

After extends types

Lets you configure the wrapping behavior for the type names of extended classes.

Example 2.161. Class/interface extends types

```
public interface Channel extends Puttable, Takable {
    ...
}
```

Example 2.162. Wrapped class/interface extends types (Endline indented)

```
public interface Channel extends Puttable,
                                 Takable {
    ...
}
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Before implements keyword

Forces a line break before the implements keyword of a class declaration.

Example 2.163. implements keyword

```
public class SynchronizedBoolean implements Comparable, Cloneable {
    ...
}
```

Example 2.164. Wrapped implements keyword (Standard indented)

```
public class SynchronizedBoolean
    implements Comparable, Cloneable {
    ...
}
```

You can control the space printed before the keyword via the indentation settings. See "Implements indent size" for more information. For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

After implements types

Forces a line wrap after each type name of the implemented classes.

Example 2.165. Class implements types

```
public class SynchronizedBoolean implements Comparable, Cloneable {
    ...
}
```

Example 2.166. Wrapped class implements types (Endline indented)

```
public class SynchronizedBoolean implements Comparable,
                                             Cloneable {
    ...
}
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Before throws keyword

Forces a line break before the throws keyword of a method/constructor declaration.

Example 2.167. throws keyword

```
private File getDestinationFile(File dest, String packageName,
                                String filename) throws IOException {
    ...
}
```

Example 2.168. Wrapped throws keyword (Endline indented)

```
private File getDestinationFile(File dest, String packageName,
                                String filename)
                            throws IOException {
    ...
}
```

You can control the space printed before the keyword via the indentation settings. See "Throws indent size" for more information. For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

After throws keyword

Forces a line break after the `throws` keyword.

Since 1.3

Example 2.169. Throws signature

```
public void foo()
    throws FooException,
        BarException {
    ...
}
```

Example 2.170. Throws signature (wrapped)

```
public void foo()
    throws
        FooException,
        BarException {
    ...
}
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

After throws types

Forces a line wrap after each type name of the throws clause of a method/constructor declaration.

Example 2.171. throws types

```
private File getDestinationFile(File dest, String packageName,
                                String filename)
                         throws IOException, FooException {
    ...
}
```

Example 2.172. Wrapped throws types (Standard indented)

```
private static final File getDestinationFile(File dest, String packageName,
                                             String filename)
    throws IOException,
        FooException {
    ...
}
```

Example 2.173. Wrapped throws types (Endline indented)

```
private File getDestinationFile(File dest, String packageName,
                                String filename)
                         throws IOException,
                                FooException {
    ...
}
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

After enum constant

When enabled, forces a line break after each enum constant of an enum declaration.

Since 1.2

Example 2.174. Enum constants

```
public enum Season {
    WINTER, SPRING, SUMMER, FALL
}
```

Example 2.175. Forced line break after enum constants

```
public enum Season {
    WINTER,
    SPRING,
    SUMMER,
    FALL
}
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Before field name

Forces a line wrap before the name of an instance field declaration. Please note that this option does not apply for multi-variables.

Since 1.2

Example 2.176. Field declaration

```
private int _count = 0
```

Example 2.177. Field declaration with line break between before the name

```
private int
_count = 0
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

After multi variable type

When enabled, a line break will be printed after the type identifier of the declaration.

Since 1.0

Example 2.178. Standard multi-variable

```
BigInteger q = null, p = null, g = null;
```

Example 2.179. Force wrap after type of multi-variables (Standard indented)

```
BigInteger
    q = null, p = null, g = null;
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

After multi variable declarator

When enabled, each declaration of a multi-variable declaration gets always printed on a new line. Otherwise, wrapping only occurs when necessary.

Since 1.0

Example 2.180. Standard multi-variable

```
BigInteger q = null, p = null, g = null;
```

Example 2.181. Force wrap after each declarator of multi-variables (Endline indented)

```
BigInteger q = null,
           p = null,
           g = null;
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Before method name

Forces a line wrap before the name of a method or constructor declaration.

Since 1.0.3

Example 2.182. Method declaration

```
public static int foo()
{
    ...
}
```

Example 2.183. Method declaration with line break between return type and name

```
public static int
foo()
{
    ...
}
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Parameters

Forces a line break after each parameter of a method or constructor declaration.

Example 2.184. Method declaration parameters

```
public static File create(File file, File directory, int backupLevel)
                  throws IOException {
    ...
}
```

Example 2.185. Wrapped method declaration parameters (Endline indented)

```
public static File create(File file,
                          File directory,
                          int backupLevel)
                  throws IOException {
    ...
}
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Chained method call

Lets you either force a line break before each call chain, preferably print a line break before a call chain if the whole call cannot be printed in just one line, only print a line break before a call chain when absolutely necessary to avoid breaking the line length limit, or disable line wrapping before call chains altoghether.

TIP You can (and probably should) have the individual call chains aligned. See below for instructions how to achieve such a style

Example 2.186. Always wrap chained method call

```
buf.append("xxxxxxxxxxxxxx")
    .append("xxxxx")
    .insert(0, "xxxxxxxxxxxxx")
    .append("xxx")
    .insert(0, "xxx");
```

Stricly enforcing a line break before each call chain is the simplest strategy, and leads to very uniform und easy to read code, but has the highest vertical space requirements.

Example 2.187. Wrap chained method call when exceed

```
buf.append("xxxxxxxxxxxxxx").append("xxxxx")
    .insert(0, "xxxxxxxxxxxxx").append("xxx").insert(0, "xxx");
```

Preferably wrapping before the call chain if the whole chain cannot be printed in one line, is the most sensible approach if you want a more compact, yet readable result.

Example 2.188. Wrap chained method call when necessary

```
buf.append("xxxxxxxxxxxxxx").append("xxxxx").insert(0,
    "xxxxxxxxxxxxxx").append("xxx").insert(0, "xxx");
```

If you want the most compact

Example 2.189. Never wrap chained method call

```
                                                                        |
message.format(ERROR_SOURCE_ADDRESS).param(m_session.getAimName()).send();
                                                                        |
buf.append("xxxxxxxxxxxxxx").append("xxxxx").insert(0,                  |
    "xxxxxxxxxxxxxx").append("AAA").insert(0, "xxx");                   |
```

Disabling wrapping before call chains altoghether, only makes a difference for calls without arguments. Otherwise, wrapping may still happen within the argument list, as you can see in the above example.

You can control the indentation for chained method calls with either "Indent dotted expressions" or "Align chained method calls". For enhanced readability it's probably best to have chained calls aligned.

Example 2.190. Aligned chained method call

```
buf.append("xxxxxxxxxxxxxx")
   .append("xxxxx")
   .insert(0, "xxxxxxxxxxxxx")
   .append("xxx")
   .insert(0, "xxx");
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Nested chained method call

Forces a line wrap after each method call chain of nested method calls, i.e. method calls used as call arguments.

Since 1.6

Example 2.191. Chained method call

```
message.format(ERROR_SOURCE_ADDRESS).param (m_session.getAimName()).send();
```

Example 2.192. Wrapped chained method call (aligned)

```
message.format(ERROR_SOURCE_ADDRESS)
       .param (m_session.getAimName())
       .send();
```

Please note that you can control the alignment for chained method calls with the Section 2.8.8.2.3, "Nested chained method calls" option. For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Call arguments

Forces a line wrap after each argument of a method call.

Example 2.193. Method call

```
doSomething();
_userDatabase.addUser("Name", encryptPassword("password", _secretKey),
                      "123 fake address");
doSomethingElse();
```

Example 2.194. Wrapped method call (Endline indented)

```
doSomething();
_userDatabase.addUser("Name",
                      encryptPassword("password",
                                      _secretKey),
                      "123 fake address");
doSomethingElse();
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Nested call arguments

Forces a line wrap after each argument of a method call if at least one argument is a method call itself. This option can prove especially useful if one prefers to nest method calls as arguments rather than adding local variables just to hold those arguments.

Example 2.195. Wrapped nested method call (Endline indented)

```
doSomething();
_userDatabase.addUser("Name",
                      encryptPassword("password", _secretKey),
                      "123 fake address");
doSomethingElse();
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Marker annotation

Lets you control the wrapping behavior of marker annotations. Marker annotations are annotations with no elements, e.g. `@Preliminary`. There are four strategies you can choose from:

Wrap when necessary	Only issue a line break after a marker annotation if otherwise the maximal line length limit would be exceeded
Wrap when exceed	Issue a line break after a marker annotation if the declaration does not fit within the maximal line length. Requires all marker annotations to appear before all other modifiers
Wrap last	Issue a line break after the *last* marker annotation. Requires all marker annotations to appear before all other modifiers
Wrap always	Issue a line break after *each* marker annotation. Requires all marker annotations to appear before all other modifiers

NOTE You can ensure that marker annotations appear before other Java modifiers via the modifier sorting settings, see Section 2.8.11.2, "Modifiers" for more information

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Since 1.1

Example 2.196. Wrap when necessary

```
@Preliminary public class TimeTravel {
    ...
}
```

Example 2.197. Wrap after last

```
@Preliminary
public class TimeTravel {
    ...
}
```

Example 2.198. No line break after marker annotation when modifier(s) before

```
public @Preliminary class TimeTravel {
    ...
}
```

Parameter marker annotation

Lets you control the wrapping behavior of marker annotations that appear within declaration parameters. There are two strategies you can choose from:

Wrap never	Never issue a line break after a marker annotation
Wrap when necessary	Only issue a line break after a marker annotation if otherwise the maximal line length limit would be exceeded

The second strategy only makes sense when using one of the endline indentation strategies. Otherwise line wrapping will preferably happen after the left parenthesis when necessary.

Since 1.9.3

Example 2.199. Wrap never

```
public Result find(@Name String rName) {
    ...
}
```

Example 2.200. Wrap when necessary

```
                                    |
public Result find(@Name            |
                   String rName) {  |
    ...                             |
}                                   |
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

After left parenthesis

Lets you control the wrapping behavior of the left parenthesis of normal and single-member annotations. You can avoid a line break altogether, only print a line break if the annotation would otherwise exceed the maximal line length, or print a line break whenever an annotation takes more than just one line.

Never wrapping is the best choice when using Section 2.8.8.1.1, “Strict Endline”, but with the other indentation strategies it depends on your preference and the annotations you use whether a line break is desirable. If you don’t want to disable wrapping for annotations altogether, it is recommended to allow a line break after the left parenthesis.

Since 1.9.2

Example 2.201. Never wrap, endline indent

```
@WebService(name = "com.company.FooDemoService",              |
            targetNamespace = "com.foo.demo")                 |
public interface Foo {                                        |
    @WebResult(targetNamespace = "com.company.jaxws.space.order")
    public Receipt placeOrder(Order order);                   |
}                                                             |
```

Example 2.202. Never wrap, standard indent

```
@WebService(name = "com.company.FooDemoService",              |
    targetNamespace = "com.foo.demo")                         |
public interface Foo {                                        |
    @WebResult(targetNamespace = "com.company.jaxws.space.order")
    public Receipt placeOrder(Order order);                   |
}                                                             |
```

Wrapping after the left parenthesis often provides more horizontal space and therefore can keep your annotations within the maximal line limit. When using anything other than Section 2.8.8.1.1, "Strict Endline" you probably want to at least enable this strategy if the line length limit is important.

Example 2.203. Wrap when necessary

```
@WebService(name = "com.company.FooDemoService",              |
            targetNamespace = "com.foo.demo")                 |
public interface Foo {                                        |
    @WebResult(                                               |
        targetNamespace = "com.company.jaxws.space.order")    |
    public Receipt placeOrder(Order order);                   |
}                                                             |
```

Wrap when exceed is the most aggressive strategy that will preferably issue a line break after the left parentheses whenever the annotation will take more than one line. This works best when using "Standard indentation", but might be desirable otherwise as well. But please note that because of backward compatibility constraints "Wrap before right parenthesis" must be enabled then.

Example 2.204. Wrap when exceed, standard indent

```
@WebService(                                                  |
    name = "com.company.FooDemoService",                      |
    targetNamespace = "com.foo.demo")                         |
public interface Foo {                                        |
    @WebResult(                                               |
        targetNamespace = "com.company.jaxws.space.order")    |
    public Receipt placeOrder(Order order);                   |
}                                                             |
```

Example 2.205. Wrap when necessary, mixed endline

```
@WebService(                                                  |
            name = "com.company.FooDemoService",              |
            targetNamespace = "com.foo.demo")                 |
public interface Foo {                                        |
    @WebResult(                                               |
        targetNamespace = "com.company.jaxws.space.order")    |
    public Receipt placeOrder(Order order);                   |
}                                                             |
```

Annotation members

Lets you control the wrapping behavior of the member expressions of normal annotations. You can disable wrapping altogether, only wrap after a member element when really neces-

sary, or force line wrapping after each element. Never wrapping can easily lead to long lines which exceeded the maximal line length, but this might be acceptable.

Since 1.5

Example 2.206. No line break after members

```
                                                                        |
@WebService(name="FooDemoService", targetNamespace="com.foo.Namespace")
public class BitTwiddle {                                               |
}                                                                       |
```

Please note that for nested annotations, wrapping might still happen depending on your parentheses preferences. If you want to disable wrapping for annotations altogether, you need to set the parentheses options to "Never wrap" as well.

Example 2.207. No line break after members

```
@Author(                                                                |
    @Name(first = "Joe", last = "Hacker", location = "Redmond") |
)                                                                       |
public class BitTwiddle { ... }                                         |
```

If the line length limit it important, you should at least enable "Wrap when necessary". This ensures that a line break will occur after an annotation member whenever the next annotation would exceed the maximal line length.

Example 2.208. Wrap when necessary

```
@WebService(name = "FooDemoService",                                    |
    targetNamespace = "com.foo.demo.Namespace")                         |
                                                                        |
public class BitTwiddle { ... }                                         |
```

Taking to the extreme, you can enforce a line break after each annotation member. This takes more vertical space, but leads to very unified and readable code. Probably best to allow wrapping after the left parentheses when not using Section 2.8.8.1.1, “Strict Endline”.

Example 2.209. Force wrap

```
@WebService(                                                            |
    name = "FooService",                                                |
    namespace = "com.foo.Namespace")                                    |
public class BitTwiddle { ... }                                         |
```

Example 2.210. Force wrap

```
@Author(
    @Name(
        first = "Joe",
        last = "Hacker"
    )
)
public class BitTwiddle { }
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Before right parenthesis

Lets you control the wrapping behavior of the right parenthesis of normal and single-member annotations. You can avoid a line break altogether, only print a line break if the anno-

tation would otherwise exceed the maximal line length, or print a line break whenever an annotation takes more than just one line.

Never wrapping is a good choice from an aesthetical point of view and should suffice for most needs. Only if you are anal about the maximal line length limit, you really should select another strategy.

Since 1.9.2

Example 2.211. Never wrap, endline indent

```
@WebService(name = "com.company.FooDemoService",              |
            targetNamespace = "com.foo.demo")                 |
public interface Foo {                                        |
    @WebResult(targetNamespace = "com.company.jaxws.space.order")
    public Receipt placeOrder(Order order);                   |
}
```

Example 2.212. Never wrap, standard indent

```
@WebService(name = "com.company.FooDemoService",              |
    targetNamespace = "com.foo.demo")                         |
public interface Foo {                                        |
    @WebResult(targetNamespace = "com.company.jaxws.space.order")
    public Receipt placeOrder(Order order);                   |
}
```

Wrapping before the right parenthesis should be required only on very rare occasions, because preferably a line break should happen after the left parenthesis. But if you absolutely strive to keep lines within the maximal line length limit, you should enable this strategy.

Example 2.213. Wrap when necessary

```
@WebResult(targetNamespace = "com.company.project.as.jaxws.space.order"|
)                                                                       |
public interface Foo { }                                                |
```

In case you prefer to let annotation members stand out, you can enable "Wrap when exceed". This way, a line break will be printed before the right parenthesis whenever the annotation list takes more than just one line to print **and** a line break has been printed after the left parenthesis. You might want to enable the corresponding strategy for the left parenthesis option if you like such a style.

Example 2.214. Wrap when exceed, standard indent

```
@WebService(
    name = "com.company.FooDemoService",
    targetNamespace = "com.foo.demo"
)
```

Operators

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Chained index operator

Lets you specify whether wrapping along the dots of chained index operators should be disabled.

Since 1.0

Example 2.215. Wrapped index operator

```
String value = objects[i].names[j]
    .first[k];
```

Example 2.216. Index operator (wrapping disabled)

```
String value =
    objects[i].names[j].first[k];
```

Note how in the above example wrapping does not occur along the dots of the index operator, but right after the assignment! For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Ternary question

Forces a line wrap after the first operand.

Example 2.217. Wrapped ternary expression question mark (Endline indented)

```
String comma = spaceAfterComma
               ? COMMA_SPACE : COMMA;
```

Indentation for consecutive lines depends on the used indentation scheme. See Section 2.8.8.1.1, "Strategies" for more information. You may further want to use continuation indentation. For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Ternary colon

Forces a line wrap after the second operand.

Example 2.218. Wrapped ternary expression colon (Endline indented)

```
String comma = spaceAfterComma ? COMMA_SPACE
                               : COMMA;
```

If both switches are disabled, ternary expressions are printed in one line (if everything fits in one line, that is).

Example 2.219. Ternary expressions

```
String comma = spaceAfterComma ? COMMA_SPACE : COMMA;
```

If both switches are enabled, you can force a style like the following:

Example 2.220. Wrapped ternary expressions (Standard indented)

```
String comma = spaceAfterComma
    ? COMMA_SPACE
    : COMMA;
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Qualifiers

Lets you specify whether wrapping along the dots of qualifiers should be disabled.

Since 1.0

Example 2.221. Wrapped qualifier

```
com.company.project
.MethodName.methodCall();
```

Example 2.222. Qualifier (wrapping disabled)

```
com.company.project.MethodName
.methodCall();
```

Note how in the above example, wrapping does not occur along the dots of the qualifier, but before the method call! For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Dotted expression

Lets you specify whether wrapping along dotted expressions should be disabled. This option covers all dotted expressions not handled by the more specific options for chained method calls, index operators or qualifiers (see above). The option is enabled by default for compatibility reasons. If you're serious about the maximal line length limit, we recommend to disable the option.

Since 1.5

Example 2.223. Wrapped dotted expression

```
boolean test = ((com.foo.highfly.test.internal.Foo) container)     |
    .transportDebugFlag;                                           |
                                                                   |
```

Example 2.224. Dotted expression (wrapping disabled)

```
boolean test =                                                     |
    ((com.foo.highfly.test.internal.Foo) container).transportDebugFlag;
                                                                   |
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

After assignment

Lets you control the way wrapping takes action for assignments. If left disabled, line wrapping preferably occurs as part of the expression printing. Otherwise wrapping will be performed right after the assignment whenever the expression cannot be printed in just one line.

Example 2.225. Prefer wrap along the expression

```
this.interessentenNr = new InteressentenNr(
        Fachschluesselerzeugung.createService()
        .getNeuerFachschluessel(
            FachschluesselerzeugungService.FACHSCHLUESSEL_KZ_INTERESSENT
        )
    );
```

Example 2.226. Prefer wrap after assignment

```
this.interessentenNr =
    new InteressentenNr(
        Fachschluesselerzeugung.createService()
        .getNeuerFachschluessel(
            FachschluesselerzeugungService.FACHSCHLUESSEL_KZ_INTERESSENT
        )
    );
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

After return

Lets you control the wrapping behavior for return statements. When enabled, a line break is inserted after the return statement when the expression would exceed the maximal line length.

Since 1.2.1

Example 2.227. return statement

```
return ((getKey() == null) ? (other.getKey() == null)           |
                           : getKey().equals(other.getKey()));  |
```

Example 2.228. Prefer wrapping after return statement

```
return                                                          |
    ((getKey() == null) ? (other.getKey() == null)              |
                        : getKey().equals(other.getKey()));     |
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

After left parenthesis

Lets you control the wrapping behavior for parameter, statement and expression lists. When left disabled, the first line break will be preferably inserted behind the first parameter or expression and only occurs after the left parenthesis if the maximal line length would be otherwise exceeded.

Example 2.229. Wrap after left parenthesis (disabled)

```
appServerReferencesVector.add(new AppServerReference(
        "RemoteApplicationManager",
        poa.create_reference_with_id(
            "RemoteApplicationManager".getBytes(),
            RemoteApplicationManagerHelper.id())));
```

When enabled, the line break will always occur behind the left parenthesis when the list parameters, arguments or expressions cannot be printed in just one line.

Example 2.230. Wrap after left parenthesis (enabled)

```
appServerReferencesVector.add(
    new AppServerReference(
        "RemoteApplicationManager",
        poa.create_reference_with_id(
            "RemoteApplicationManager".getBytes(),
            RemoteApplicationManagerHelper.id())));
```

This option affects the output style of method/constructor declarations and calls, creator calls and `if-else`, `for`, `while` and `do-while` blocks. As per default, the wrapped lines will be indented using standard indentation, but you may want to apply another indentation scheme. See Section 2.8.8.1.1, “Strategies” for more information. For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Before right parenthesis

Prints a line break before the right parenthesis of parameter or expression lists when at least one parameter/expression was wrapped. The parenthesis will be intended according to the current indentation level. This switch affects the output style of method/constructor declarations and calls, creator calls and `if-else`, `for`, `while` and `do-while` blocks.

Example 2.231. Right parenthesis (disabled)

```
public void severalParameters(String one,
                              int two,
                              String three,
                              StringObject four,
                              AnotherObject five) {
}
```

Example 2.232. Right parenthesis (enabled)

```
public void severalParameters(String one,
                              int two,
                              String three,
                              StringObject four,
                              AnotherObject five
) {
}
```

Both switches combined, looks like the following example:

Example 2.233. Left and right parenthesis

```
appServerReferencesVector.add(
    new AppServerReference(
        "RemoteApplicationManager",
        poa.create_reference_with_id(
            "RemoteApplicationManager".getBytes(),
            RemoteApplicationManagerHelper.id()
        )
    )
);
```

For blocks the output may go like this:

Example 2.234. Left and right parenthesis (wrapped)

```
if (
    "pick".equals(m.getName()) && m.isStatic() && m.isPublic()
) {
    pickFound = true;
} else if (
    "pick".equals(m.getName()) && m.isStatic() && m.isPublic()
) {
    pickFound = true;
}
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Grouping parentheses

Lets you control the wrapping behavior for grouping parentheses. When enabled, line breaks are inserted after left and before right parentheses of grouped expressions to let the expression(s) stand out.

Example 2.235. Grouping parentheses (standard indented)

```
if (
    !((bankverbindung instanceof ObjectValue)
    || (bankverbindung instanceof PrimitiveValue))
) {
    throw new RuntimeException();
}
```

Example 2.236. Wrapped grouping parentheses (standard indented)

```
if (
    !(
        (bankverbindung instanceof ObjectValue)
        || (bankverbindung instanceof TkPrimitiveValue)
    )
) {
    throw new RuntimeException();
}
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Type parameter

When enabled, type parameters of parametrized (generic) types are wrapped, when necessary.

Since 1.4

Example 2.237. Parameterized type that lead to exceeded maximal line length

```
                                                                                |
private final Map<Short, String> example = new HashMap<Short, String>();
                                                                                |
```

Example 2.238. Wrapped parameterized type (endline indented)

```
                                                                                |
private final Map<Short, String> example = new HashMap<Short,                   |
                                                          String>();            |
                                                                                |
```

Please note that with Endline indentation enabled, wrapping only happens if both type parameter names are either not single-lettered or contain one ore more bounds.

Example 2.239. Exceptions when using endline indentation

```
                                                                |
public class Test<A, B extends Comparable<B> & Cloneable> {
}                                                               |
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Labels

Forces a line wrap after labels.

Example 2.240. Label

```
// advance to the first CLASS_DEF or INTERFACE_DEF
LOOP:   for (AST child = tree.getFirstChild();
             child != null;
             child = child.getNextSibling()) {
            switch (child.getType()) {
                case JavaTokenTypes.CLASS_DEF :
                case JavaTokenTypes.INTERFACE_DEF :
                    next = child;
                    break LOOP;

                default :
                    break;
            }
        }
```

Example 2.241. Wrapped label (Standard indented)

```
// advance to the first CLASS_DEF or INTERFACE_DEF
LOOP:
        for (AST child = tree.getFirstChild();
             child != null;
             child = child.getNextSibling()) {
            switch (child.getType()) {
                case JavaTokenTypes.CLASS_DEF :
                case JavaTokenTypes.INTERFACE_DEF :
                    next = child;
                    break LOOP;

                default :
                    break;
            }
        }
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

Registry Keys

Forces line wrapping after assignments of public constants that are defined in a class whose name ends with the "Registry" suffix.

Since 1.7

Example 2.242. Registry keys

```
class FooRegistry {
    public final static boolean ALLOW_FAKE_FOOS = "com.company.layer.Foo";
}
```

Example 2.243. Registry keys (wrapping forced)

```
class FooRegistry {
    public final static boolean ALLOW_FAKE_FOOS =
        "com.company.layer.Foo";
}
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

After parameters/expressions

When enabled, this switch will cause all parameters/expressions to be wrapped, if and only if the first parameter/expression of the list has been wrapped.

Example 2.244. Expression list (all wrapped)

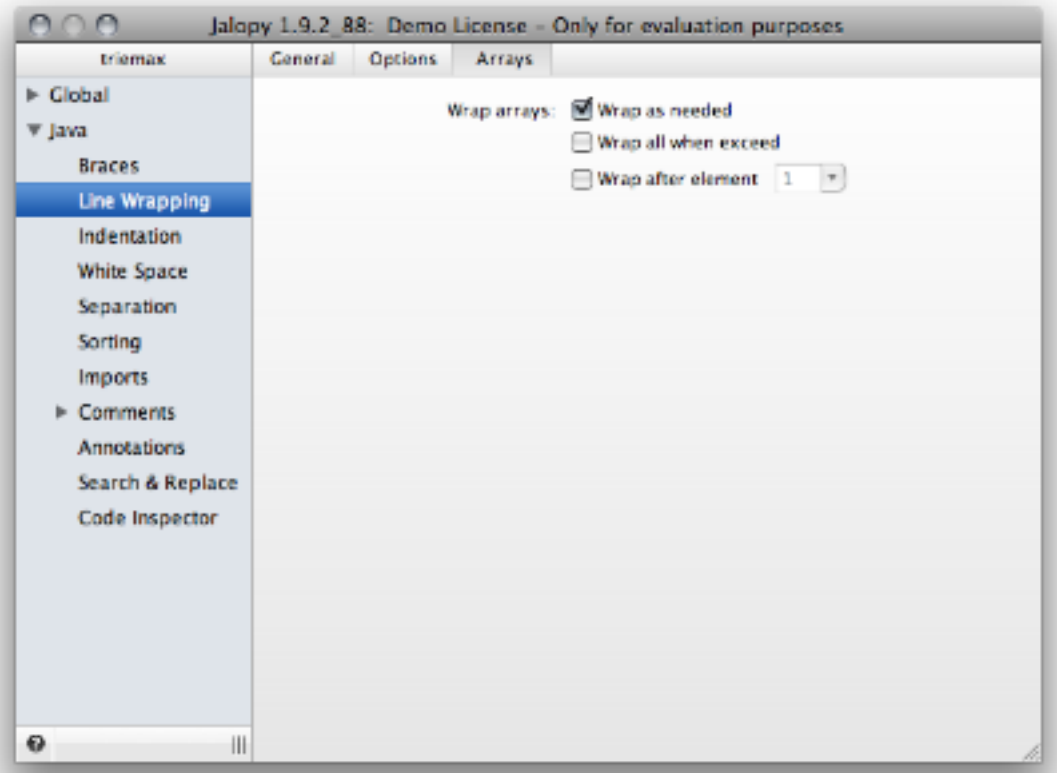
```
if (
    "pick".equals(m.getName()) &&
    m.isStatic() &&
    m.isPublic()
) {
    pickFound = true;
} else if (
    "pick".equals(m.getName()) &&
    m.isStatic() &&
    m.isPublic()
) {
    pickFound = true;
}
```

For general information about the available wrapping strategies, please refer to the wrapping strategies overview.

2.8.7.3 Arrays

Contains options to control the wrapping behavior for arrays.

Figure 2.40. Arrays settings page



Wrap as needed

Enabling this option means cause line breaks to be inserted, whenever an element would otherwise exceed the maximal line length limit.

Example 2.245. Wrap as needed

```
String[] s = new String[] {              |
    "first", "second", "third", "fourth",|
    "fifth", "sixth", "seventh",         |
    "eighth", "ninth", "tenth",          |
};                                       |
```

Wrap all when exceed

Forces a line break after every array element when all elements would not fit into the current line limit.

Since 1.7

Example 2.246. All elements fit into line

```
String[] s = new String[] {                                                  |
    "first", "second", "third", "fourth", "fifth", "sixth", "seventh",|
};                                                                           |
```

Example 2.247. Wrap all elements when line would get exceeded

```
String[] s = new String[] {                                                 |
    "first",                                                                |
    "second",                                                               |
    "third",                                                                |
    "fourth",                                                               |
    "fifth",                                                                |
    "sixth",                                                                |
    "seventh",                                                              |
    "eighth",                                                               |
};                                                                          |
```

Wrap after element

Forces a line break after every n-th element.

Example 2.248. Wrap after element 1

```
String[] s = new String[] {    |
    "first",                   |
    "second",                  |
    "third",                   |
    "fourth",                  |
    "fifth",                   |
    "sixth",                   |
    "seventh",                 |
    "eighth",                  |
    "ninth",                   |
    "tenth",                   |
};
```

Please note that no checking is done regarding the maximal line length limit which might easily lead to lines exceeding the maximal line length when you set *n* to something bigger than '1'.

Example 2.249. Wrap after element 3

```
String[] s = new String[] {    |
    "first", "second", "third",|
    "fourth", "fifth", "sixth",|
    "seventh", "eighth", "ninth",
    "tenth",                   |
}                              |
```

If neither option is enabled, the array elements will be printed in one line, right after the left curly brace, i.e. automatic line wrapping will be disabled.

Please note that you can further customize the wrapping behavior for arrays with the following options:

- Section 2.8.6.2.3, “Arrays”
- Section 2.8.7.1.2, “Array elements”

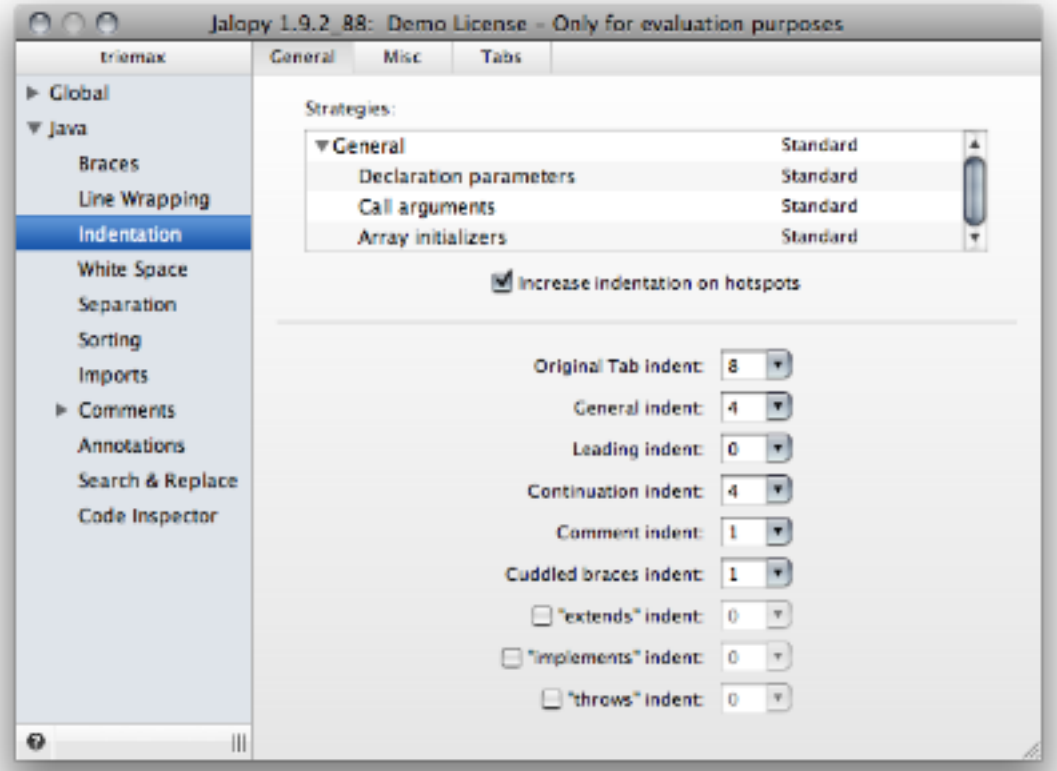
2.8.8 Indentation

Controls the indentation settings. Indentation is core to readability and describes the way white space is used to emphasize the logical structure of a program—logically subordinated elements are printed with increased indentation.

2.8.8.1 General

Lets you change the general indentation settings.

Figure 2.41. Indentation settings page



Strategies

Lets you choose the general strategy how lines should be indented. Jalopy supports several indentation strategies with different characteristics. You can even use different strategies for different elements, if you can't decide on one global policy.

Changing the general indentation strategy in the tree widget will adjust all subordinated elements that are configured to use the prior general indentation strategy as well.

Standard

With standard indentation, lines will be always indented according to the current indentation level. The indentation level changes as the block or parentheses level changes. Standard indentation gives you great consistency: Because indentation always uses the same sizes (multiples of the defined "Indent size"), source code is very uniformly layed out and the white space gaps tend to be small.

When standard indentation is enabled, line wrapping will always try to keep lines within the maximal line length.

Example 2.250. Method declaration

```
public void severalParameters(String one, int two,
    String three, StringObject four, AnotherObject five)
{
}
```

Example 2.251. Method call

```
vector.add(new AppServer(
        "RemoteApplicationManager",
        poa.create_reference_with_id(
            "RemoteApplicationManager".getBytes(),
            RemoteApplicationManager.id())));
```

Example 2.252. Assignment

```
doublette[PflegeController.GEBURTSDATUM] =
    resultSetRow[i].field[0].substring(0, 2) + "." +
    resultSetRow[i].field[0].substring(2, 4) + "." +
    resultSetRow[i].field[0].substring(4, 6);
```

Mixed Endline

Mixed endline indentation preferably lays out code relative to the most recent left parenthesis, assignment or curly brace offset (the *hotspots*). Whenever rigorously orienting on only the current hotspot would lead to a crossing of the maximal line length, the preceding hotspot is checked. This process is done recursively, therefore in rare cases, this strategy produces the exact same result as standard indentation. Mixed endline indentation uses bigger white space gaps than standard indentation as code tends to move towards the right edge—but this keeps related code more closely together. The downside is that indentation is not so uniformly distributed and more vertical space might be occupied.

When mixed endline indentation is enabled, line wrapping will always try to keep lines within the maximal line length.

Example 2.253. Method declaration

```
public void severalParameters(String one, int two,
                              String three,
                              StringObject four,
                              AnotherObject five)
{
}
```

Example 2.254. Method call

```
vector.add(new AppServer(
               "RemoteApplicationManager",
               poa.create_reference_with_id(
                   "RemoteApplicationManager"
                   .getBytes(),
                   RemoteApplicationManager
                   .id())));
```

Example 2.255. Assignment

```
doublette[PflegeController.GEBURTSDATUM] =
    resultSetRow[i].field[0].substring(0, 2) + "." +
    resultSetRow[i].field[0].substring(2, 4) + "." +
    resultSetRow[i].field[0].substring(4, 6);
```

Strict Endline

Strict endline indentation always lays out code relative to the most recent left parenthesis, assignment or curly brace offset (the *hotspots*). This way consecutive code sections are somewhat easier to recognize at the possible downside of consuming more vertical and/or horizontal space.

Strict endline indentation generally tries to keep lines within the maximal line length, but favors aligning over wrapping, and thus will often lead to code crossing the maximal line length.

NOTE It is recommend to avoid this strategy when possible, because depending on your wrapping settings it can produce quite scary results. It's mostly available for historic reasons in order to provide backwards compatibility and allow for a smooth transition phase

Example 2.256. Method declaration

```
public void severalParameters(String one, int two,
                              String three,
                              StringObject four,
                              AnotherObject five)
{
}
```

Example 2.257. Method call

```
vector.add(new AppServer("RemoteApplicationManager",
                         poa
                         .create_reference_with_id("RemoteApplicationManager"
                                                   .getBytes(),
                                                   RemoteApplicationManager
                                                   .id())));
```

Example 2.258. Assignment

```
doublette[PflegeController.GEBURTSDATUM] = resultSetRow[i]
                                           .field[0]
                                           .substring(0,
                                                      2) +
                                           "." +
                                           resultSetRow[i]
                                           .field[0]
                                           .substring(2,
                                                      4) +
                                           "." +
                                           resultSetRow[i]
                                           .field[0]
                                           .substring(4,
                                                      6);
```

Regarding array initializers, any of the endline indentation strategies will cause the initializer to be printed right after the assignment. But when enabled, standard indentation might cause the initializer to be printed on a line of its own (but only when the initializer takes more than one line to print).

Example 2.259. Array initializer (Endline indented)

```
String[] s = { "first" };   |
                            |
String[] s = {              |
                 "first",   |
                 "second"   |
             };             |
```

Example 2.260. Array initializer (Standard indented)

```
String[] s = { "first" };   |
                            |
String[] s =                |
    {                       |
        "first",            |
        "second"            |
    };                      |
```

If you want to enforce a line break before *all* array initializers, you need to disable the compact brace printing for array initializers. See Section 2.8.6.2.3, "Arrays" for more information.

Always increase indentation on hotspots

By default, Jalopy always increases indentation after certain code elements to emphasize scope and nesting level. Depending on your settings, hotspots might be left curly braces, left parentheses, operators and certain keywords like *return* and *assert*. Now, it can be that indentation is increased in a way that could be seen as superfluous, because already one level of indentation would be enough to indicate the basic logical structure of a code statement.

Example 2.261. Always increase indentation on hotspots

```
Object value = calculateValue( getFirstNumber(),
»   »   getSecondNumber(), getThirdNumber() );
```

As you can see from the above example, if the option is enabled, indentation will be increased on every hotspot (here the assignment and left parentheses), which for deeply nested code can easily take up quite some horizontal space. The second increase does not add significant information. If you prefer a more dense layout, disabling the option will cause indentation to be increased only when really necessary. The result often takes considerably less horizontal space without loosing significant information.

Example 2.262. Only increase indentation when absolutely necessary

```
Object value = calculateValue( getFirstNumber(),
»   getSecondNumber(), getThirdNumber() );
```

Here, the indentation is only increased once within the statement and thus upon wrapping the remaining call arguments are indented only one level.

Since 1.7

Sizes

Lets you set different indentation sizes.

Original Tab indent

For documents that contain real tabs, specifies the number of spaces per tab stop. Look in your IDE editor or formatting settings for the "Tab Size" or "Tab Width" option and set the Jalopy option to the value found there.

IMPORTANT Please be aware that it is essential to set the correct tab size. Otherwise some indentations or alignments may fail.

General indent

Specifies the number of spaces to use for general indentation (studies have found that 2 to 4 spaces for indentation is optimal).

Example 2.263. 2 space general indent

```
public class Preferences {
->private Preferences()
->{
->}

->public static void main(String[] argv) {
->->com.triemax.jalopy.swing.PreferencesDialog.main(argv);
->}
}
```

Example 2.264. 4 space general indent

```
public class Preferences {
--->private Preferences() {
--->}

--->public static void main(String[] argv) {
--->--->com.triemax.jalopy.swing.PreferencesDialog.main(argv);
--->}
}
```

Leading indent

Specifies the number of spaces to prepend before every line printed.

Example 2.265. 6 space leading indent

```
----->public class Preferences {
----->    private Preferences() {
----->    }

----->    public static void main(String[] argv) {
----->        com.triemax.jalopy.swing.PreferencesDialog.main(argv);
----->    }
----->}
```

Continuation indent

Specifies the number of spaces that should be inserted in front of continuation lines, i.e. the consecutive lines in case of a line wrap. Please note that this option only takes effect if continuation indentation is enabled. Refer to Section 2.8.8.2.2, "Continuation indent" for information on how to enable continuation indentation.

Example 2.266. 2 space continuation indent

```
if ((condition1 && condition2)
    ->|| (condition3 && condition4)
    ->|| !(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Example 2.267. 4 space continuation indent

```
if ((condition1 && condition2)
    --->|| (condition3 && condition4)
    --->|| !(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Comment indent

Specifies the number of spaces to insert between trailing comments and the preceding statement.

Example 2.268. 3 space trailing comment indent

```
new String[] {
    "Sunday",-->// Sunday
    "Monday",-->// Monday
    "Tuesday",-->// Tuesday
    "Wednesday",-->// Wednesday
    "Thursday",-->// Thursday
    "Friday",-->// Friday
    "Saturday"-->// Saturday
}
```

Cuddled braces indent

Specifies the number of spaces to print before the left curly brace of cuddled empty braces.

Example 2.269. 3 space cuddled braces indent

```
try {
    in.close();
} catch (IOException ignored)-->{}
```

See Section 2.8.6.2.4, "Cuddle braces" for more information about the empty braces handling.

Extends indent

When enabled, specifies the white space to print before the `extends` keyword in case it was printed on a new line.

Example 2.270. extends indentation with 6 spaces

```
public interface Channel
------>extends Puttable, Takable {
    ...
}
```

Implements indent

Specifies the white space to print before the `implements` keyword in case it was printed on a new line.

Example 2.271. implements indentation with 8 spaces

```
public class SynchronizedBoolean
------->implements Comparable, Cloneable {
    ...
}
```

Throws indent

Specifies the white space to print before the `throws` keyword in case it was printed on a new line.

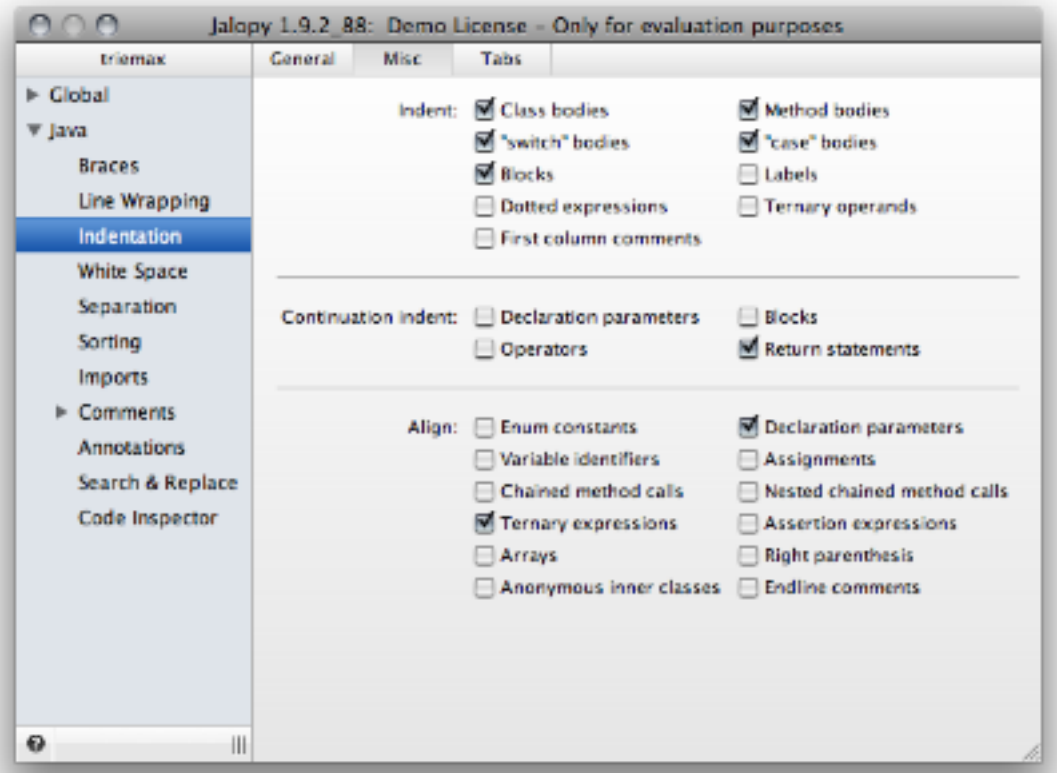
Example 2.272. throws indentation with 3 spaces

```
private static final File getDestinationFile(File dest, String packageName,
                                             String filename)
-->throws IOException, FooException {
    ...
}
```

2.8.8.2 Misc

Lets you control miscellaneous indentation settings.

Figure 2.42. Indentation Misc settings page



Indent

Indent class bodies

When enabled, indentation is increased for statements within class, interface, enum and annotation type declarations. Disabling this option might only make sense when using the GNU brace style.

Since 1.4

Example 2.273. Indented class body

```
public class Foo
    {
    --->public Foo()
    --->    {
    --->    }

    --->public boolean isFoo(Object other)
    --->    {
    --->        return other instanceof Foo;
    --->    }
    }
```

Example 2.274. Class body without increased indentation

```
public class Foo
    {
    public Foo()
        {
        }

    public boolean isFoo(Object other)
        {
            return other instanceof Foo;
        }
    }
```

Indent method bodies

When enabled, indentation is increased for statements within method and constructor bodies. Disabling this option might only make sense when using the GNU brace style.

Since 1.4

Example 2.275. Indented method body

```
public void toString()
    {
    --->return this.line + " " + this.column + " " + this.text;
    }
```

Example 2.276. Method body without increased indentation

```
public void toString()
    {
    return this.line + " " + this.column + " " + this.text;
    }
```

Indent "switch" bodies

The Sun Java code convention recommends a switch style where case statements are not indented relative to the switch statement as a whole. However, this option allows you to indent the case statements to make the entire switch statement stand out.

Example 2.277. Switch statement (unindented)

```
switch (prio) {
case Priority.ERROR_INT :
case Priority.FATAL_INT :
    color = Color.red;
    break;

case Priority.WARN_INT :
    color = Color.blue;
    break;

default:
    color = Color.black;
    break;
}
```

Example 2.278. Switch statement (indented)

```
switch (prio) {
--->case Priority.ERROR_INT :
--->case Priority.FATAL_INT :
--->    color = Color.red;
--->    break;

--->case Priority.WARN_INT :
--->    color = Color.blue;
--->    break;

--->default:
--->    color = Color.black;
--->    break;
}
```

Indent "case" bodies

When enabled, indentation is increased for statements within case statement bodies.

Since 1.4

Example 2.279. Indented case bodies

```
switch (prio) {
    case Priority.ERROR_INT :
    case Priority.FATAL_INT :
    --->color = Color.red;
    --->break;

    case Priority.WARN_INT :
    --->color = Color.blue;
    --->break;

    default:
    --->color = Color.black;
    --->break;
}
```

Example 2.280. Case statements without increased indentation

```
switch (prio) {
    case Priority.ERROR_INT :
    case Priority.FATAL_INT :
    color = Color.red;
    break;

    case Priority.WARN_INT :
    color = Color.blue;
    break;

    default:
    color = Color.black;
    break;
}
```

Indent block bodies

When enabled, indentation is increased for statements within blocks. Disabling this option might only make sense when using the GNU brace style.

Since 1.4

Example 2.281. Indented block

```
if (true)
    {
    --->doStuff();
    --->assertStuffDoneCorrectly();
    }
```

Example 2.282. Block without increased indentation

```
if (true)
    {
    doStuff();
    assertStuffDoneCorrectly();
    }
```

Indent labels

Specifies whether labels should be indented with the current indentation level.

Example 2.283. Unindented label

```
// advance to the first CLASS_DEF or INTERFACE_DEF
LOOP:
        for (AST child = tree.getFirstChild();
             child != null;
             child = child.getNextSibling()) {
            switch (child.getType()) {
                case JavaTokenTypes.CLASS_DEF :
                case JavaTokenTypes.INTERFACE_DEF :
                    next = child;
                    break LOOP;
                default :
                    break;
            }
        }
```

Example 2.284. Indented label

```
--->--->// advance to the first CLASS_DEF or INTERFACE_DEF
--->--->LOOP:
        for (AST child = tree.getFirstChild();
             child != null;
             child = child.getNextSibling()) {
            switch (child.getType()) {
                case JavaTokenTypes.CLASS_DEF :
                case JavaTokenTypes.INTERFACE_DEF :
                    next = child;
                    break LOOP;

                default :
                    break;
            }
        }
```

Indent dotted expressions

When enabled, indentation is increased for dotted expressions. This option is only present for historic reasons—to be able to address some unwanted behavior without breaking backwards compatibility. It is recommended to have this option always enabled.

Since 1.9

Example 2.285. Unindented dotted expression

```
((org.omg.CORBA_2_3.portable.OutputStream) s)
.write_abstract_interface(o);
```

Example 2.286. Indented dotted expression

```
((org.omg.CORBA_2_3.portable.OutputStream) s)
    .write_abstract_interface(o);
```

Indent ternary operands

When enabled, indentation is increased for the operands of the ternary operator. This option is only present for historic reasons - to be able to address some unwanted behavior without breaking backwards compatibility. It is recommended to have this option always enabled.

Since 1.8

Example 2.287. Indented ternary operands

```
text.setValidationSpec(config.getSpecification(),       |
                       config.getAda(data               |
                                     .hasTransmittal()  |
                                     ? data             |
                                       .getTransmitter()|
                                       .getElement()    |
                                     : data             |
                                       .getElement())); |
```

Example 2.288. Unindented ternary operands

```
text.setValidationSpec(config.getSpecification(),       |
                       config.getAda(data               |
                                     .hasTransmittal()  |
                                     ? data             |
                                     .getTransmitter()  |
                                     .getElement()      |
                                     : data             |
                                     .getElement()));   |
```

Indent first column comments

By default, all comments will be indented relative to their position in the code to avoid that comments break the logical structure of the program. But as for certain kind of comments it may be useful to put them at the first column, you can control here whether such comments should be printed without indentation.

Example 2.289. First column comment

```
    public static Printer create(AST node) {

/*
        if (node == null) {
            return new NullPrinter();
        }
*/
        return create(node.getType());
    }
```

Example 2.290. Indented comment

```
    public static Printer create(AST node) {

        /*
        if (node == null) {
            return new NullPrinter();
        }
        */
        return create(node.getType());
    }
```

Continuation indent

Lets you specify extra indentation for consecutive lines of certain elements.

Declaration parameters

With standard indentation enabled, this option causes an extra increase of the indentation level upon wrapping of method and constructor parameters. With endline indentation this option is meaningless.

Example 2.291. Wrapped method parameters

```
public void aMethod(int param 1, int param2, |
    int param3) {                            |
    int a = param1 + param2 + param3;        |
}                                            |
```

Example 2.292. Wrapped method parameters with continuation indentation

```
public void aMethod(int param 1, int param2, |
        int param3) {                        |
    int a = param1 + param2 + param3;        |
}                                            |
```

Blocks

The Sun brace style could make seeing the statement body difficult. To work around this problem, you may want to use continuation indentation in case you like this brace style. This setting applies for `if`, `for`, `while` and `do-while` blocks.

Example 2.293. Non-continuation indentation

```
if ((condition1 && condition2)
    || (condition3 && condition4)
    || !(condition5 && condition6)) { // BAD WRAPS
    doSomethingAboutIt();             // MAKE THIS LINE EASY TO MISS
}
```

Example 2.294. Continuation indentation

```
if ((condition1 && condition2)
        || (condition3 && condition4)
        || !(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Refer to Section 2.8.6.1, “Layout” for the available brace style options.

Operators

When enabled, indentation will be increased before an operand will be printed.

Example 2.295. Ternary expression (endline indented)

```
String comma = spaceAfterComma
               --->? COMMA_SPACE
               --->: COMMA;
```

return statements

Lets you increase the indentation level after return statements. This is only meaningful when a statement cannot be printed in one line.

Since 1.6

Example 2.296. return statement

```
return auswahlkriteriumComboBox.getUnselectedValueContent()   |
.equals(auswahlkriteriumComboBox.getSelectedItem());          |
```

Example 2.297. return statement with continuation indentation

```
return auswahlkriteriumComboBox.getUnselectedValueContent()   |
    .equals(auswahlkriteriumComboBox.getSelectedItem());      |
```

As you can see from the above examples, without continuation indentation it might happen that wrapped lines are printed at the same indentation level as the return statement. It really depends where the line wrapping takes place. The two following examples indent both no matter what setting (though different, of course).

Example 2.298. return statement

```
return auswahlkriteriumComboBox.getUnselectedValueContent().equals(     |
    auswahlkriteriumComboBox.getSelectedItem());                        |
```

Example 2.299. return statement with continuation indentation

```
return auswahlkriteriumComboBox.getUnselectedValueContent().equals(     |
        auswahlkriteriumComboBox.getSelectedItem());                    |
```

Align

Lets you control what elements should be visually aligned to each other.

Enum constants

Lets you align the parameter lists of enum constants. Please note that aligning only takes place if "Wrap after enum constants" has been enabled as well.

Since 1.8

Example 2.300. Enum constants

```
public enum Code {
    NOT_READY((byte) 0x09),
    ERROR((byte) 0x10),
    ILLEGAL((byte) 0x11);
}
```

Example 2.301. Aligned enum constants

```
public enum Code {
    NOT_READY((byte) 0x09),
    ERROR    ((byte) 0x10),
    ILLEGAL  ((byte) 0x11);
}
```

Declaration parameters

When enabled, aligns the parameters of method/constructor declarations. This only applies if all parameters will be wrapped; either because wrapping is forced or the max. line length is reached. To force aligning, you have to enable the wrapping for method parameters. See "Declaration parameters" for more information.

Example 2.302. Method declaration parameters

```
public static File create(final File file,
                          File directory,
                          int backupLevel) {
    ...
}
```

Example 2.303. Method declaration parameters (aligned)

```
public static File create(final File file,
                          File       directory,
                          int        backupLevel) {
    ...
}
```

Variable identifiers

When enabled, the identifiers of consecutive variable declarations are aligned. You can control what declarations are treated as consecutive with different chunk options. Refer to Section 2.8.10.2.1, "Chunks" for more information.

Example 2.304. Variable identifiers

```
String text = "text";
int a = -1;
History.Entry entry = new History.Entry(text);
```

Example 2.305. Variable identifiers (aligned)

```
String        text = "text";
int           a = -1;
History.Entry entry = new History.Entry(text);
```

Assignments

When enabled, consecutive assignment expressions and the assignment parts of consecutive variable declarations are aligned. You can control what statements are treated as consecutive with different chunk options. Refer to Section 2.8.10.2.1, "Chunks" for more information.

Example 2.306. Variable assignments (aligned)

```
String text         = "text";
int a               = -1;
History.Entry entry = new History.Entry(text);
```

If both variable alignment options are enabled, you can achieve a style like the following:

Example 2.307. Variable identifiers/assignments (both aligned)

```
String        text  = "text";
int           a     = -1;
History.Entry entry = new History.Entry(text);
```

Chained method calls

When disabled, indentation happens according to the current indentation level.

Example 2.308. Method call chain (standard indented)

```
Schluesselerzeugung.createService()
.getNeuerSchluesselService(
    SchluesselService.SCHLUESSEL_KZ_INTERESSENT);
```

Otherwise indentation is performed relative to the column offset of the first chain link.

Example 2.309. Method call chain (aligned)

```
Schluesselerzeugung.createService()
                   .getNeuerSchluesselService(
                       SchluesselerzeugungService.SCHLUESSEL_KZ_INTERESSENT);
```

Please note that you can enforce line breaks for chained method calls, see "Wrap chained method call".

Nested chained method calls

When disabled, indentation happens according to the current indentation level.

Since 1.6

Example 2.310. Method call chain (standard indented)

```
id = RefData.getSegmentId(RefData.getAccessor().getCutOff(fileInfo
                                                          .getCutoffId())
                          .getTheoreticalDate(),
                          fileInfo.getExternalSenderId(),
                          _fileType,
                          fileInfo.getCustomerFormat()
                          .getId());
```

Otherwise indentation is performed relative to the column offset of the first chain link.

Example 2.311. Method call chain (aligned)

```
id = RefData.getSegmentId(RefData.getAccessor()
                                 .getCutOff(fileInfo.getCutoffId())
                                 .getTheoreticalDate(),
                          fileInfo.getExternalSenderId(),
                          _fileType,
                          fileInfo.getCustomerFormat()
                                  .getId());
```

Please note that you can force line breaks for nested chained method calls, see "Wrap nested chained method call".

Ternary expressions

If disabled, ternary expressions are printed according to the current indentation policy. See Section 2.8.8.1.1, "Strategies" for more information.

Example 2.312. Ternary operators (standard indented)

```
alpha = (aLongBooleanExpression) ? beta    |
: gamma;                                   |
```

Example 2.313. Ternary operators (endline indented)

```
alpha = (aLongBooleanExpression) ? beta    |
        : gamma;                           |
```

When enabled, the second operator will always be aligned relative to the first one.

Example 2.314. Ternary expressions (aligned)

```
alpha = (aLongBooleanExpression) ? beta    |
                                 : gamma;  |
```

Note that this option only takes effect, if indeed a line break was inserted before the second expression. You can enforce such line breaks. See "Wrap ternary expression colon" for more information.

Assertion expressions

When enabled, the second expression of assertion statements is aligned.

Example 2.315. Assertion expression (standard indented)

```
assert ((nBits & ~ALL_BITS) != 0) : "Invalid modifier bits: " +
    (nBits & ~ALL_BITS);
```

Example 2.316. Assertion expression (endline indented)

```
assert ((nBits & ~ALL_BITS) != 0) : "Invalid modifier bits: " +
        (nBits & ~ALL_BITS);
```

Example 2.317. Assertion expression (aligned)

```
assert ((nBits & ~ALL_BITS) != 0) : "Invalid modifier bits: " +
                                    (nBits & ~ALL_BITS);
```

Arrays

Forces alignment of the curly braces of array initializers with the declaration. This only applies when using either the standard indentation policy or mixed endline indentation policy, because here by default the braces are indented according to the current indentation level and therefore do not align.

Since 1.0.3

Example 2.318. Array initialization (standard indented)

```
private static String [] sFields =
    {
        "LOCATION.ID",
        "TRACKING_EVENT.CREATE_TIME",
        "TRACKING_EVENT.EVENT_TIME",
        "TRACKING_EVENT.ORIGIN_TIME_ZONE_OFFSET",
        "TRACKING_EVENT.EVENT_TYPE",
    };
```

Example 2.319. Array initialization (standard indented, but aligned)

```
private static String [] sFields =
{
    "LOCATION.ID",
    "TRACKING_EVENT.CREATE_TIME",
    "TRACKING_EVENT.EVENT_TIME",
    "TRACKING_EVENT.ORIGIN_TIME_ZONE_OFFSET",
    "TRACKING_EVENT.EVENT_TYPE",
};
```

Right parenthesis

Forces alignment of wrapped right parentheses with the declaration or call. This only applies when using standard indentation policy, because here by default the parentheses are indented according to the current indentation level.

Since 1.2.1

Example 2.320. Method call (standard indented, 2 spaces indent)

```
myMethod(
  getSomeValue(
    param1
    ),
  param2
  );
```

Example 2.321. Method call (standard indented, but aligned)

```
myMethod(
  getSomeValue(
    param1
  ),
  param2
);
```

Example 2.322. Method declaration (standard indented)

```
public MyCustomStringTemplate createTemplate(
    Map three, int one, // xxx
    String  two // xxx
    )
{
    ...
}
```

Example 2.323. Method declaration (standard indented, but aligned)

```
public MyCustomStringTemplate createTemplate(
  Map three, int one, // xxx
  String  two // xxx
)
{
    ...
}
```

Anonymous inner classes

Lets you force alignment of anonymous inner brace blocks according to the current indentation level. Only applies when standard indentation is used.

Since 1.6

Example 2.324. Anonymous inner class (standard indented)

```
Action action = new AbstractAction("action") {
        public void actionPerformed (ActionEvent ev) {
        }
    };
```

Example 2.325. Anonymous inner class (standard indented but aligned)

```
Action action = new AbstractAction("action") {
    public void actionPerformed (ActionEvent ev) {
    }
};
```

Endline comments

Aligns endline comments that belong together. You can control how Jalopy determines what comments belong together with the Chunk settings.

Since 1.9

Example 2.326. Endline comments

```
if (m.tryMatch(h)) { // help match
    casHead(h, mn); // pop both h and m
} else { // lost match
    h.casNext(m, mn); // help unlink
}
```

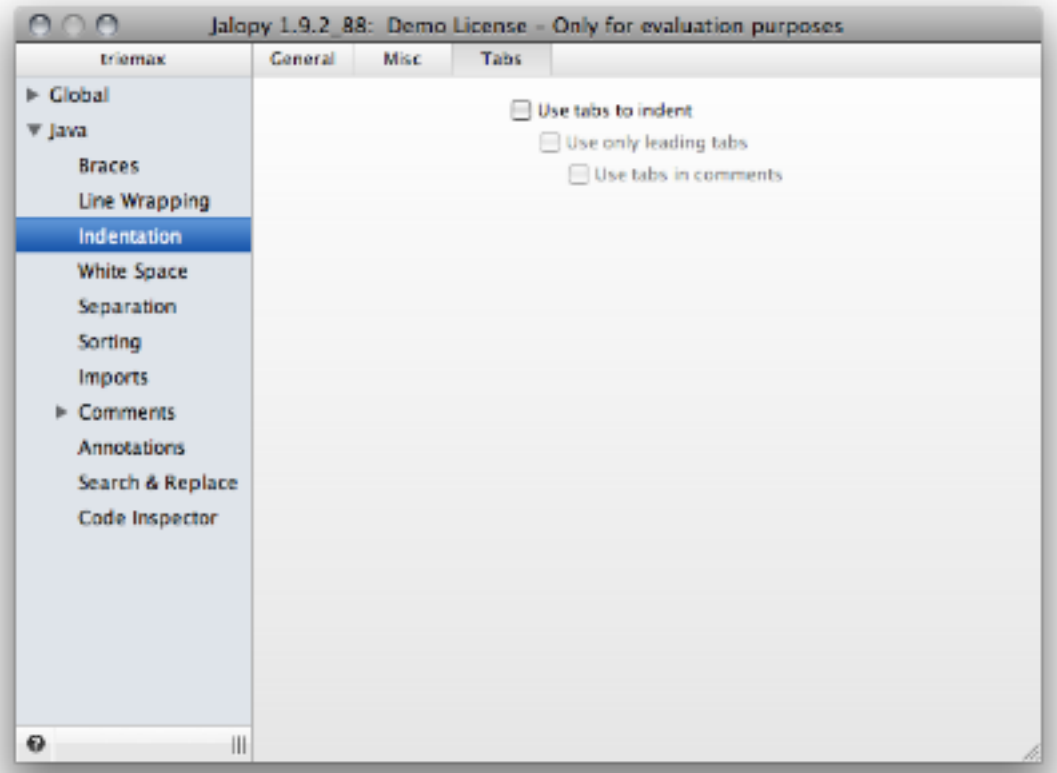
Example 2.327. Aligned endline comments

```
if (m.tryMatch(h)) {  // help match
    casHead(h, mn);   // pop both h and m
} else {              // lost match
    h.casNext(m, mn); // help unlink
}
```

2.8.8.3 Tabs

Lets you control the tab settings.

Figure 2.43. Indentation Tabs settings page



Use tabs to indent

Normally, Jalopy uses spaces to indent lines. If you prefer hard tabs, check this option. You can change the original tab indent on the general indentation settings page, see the Original Tab indent option.

Please note that it is very important that you specify the correct tab size for your original sources on the general indentation page.

Example 2.328. Use tabs for indentation

```
»   if ( true ) {
»   »   methodCall(param1,
»   »   »   param2
»   »   »   param3);
»   }
```

Use only leading tabs

When enabled, tabs are only used up to the current brace level, spaces are used afterwards.

Example 2.329. Leading indentation

```
»   if (test()) {
»   »   methodCall(param1,
»   »   ····param2
»   »   ····param3);
»   }
```

Use tabs in comments

When enabled, hard tabs are used for printing indentation in comments.

Since 1.2.1

Example 2.330. Comment that uses spaces for indentation

```
//·····System.out.println("DEBUG: line=" + _line);
```

Example 2.331. Comment that uses tabs for indentation

```
//» »···System.out.println("DEBUG: line=" + _line);
```

2.8.9 White Space

The white space settings page lets you configure how white space characters are used to separate individual syntax elements of source files. Making good use of spacing is considered good programming style and greatly enhances developer comprehension, therefore Jalopy provides a vast amount of flexibility to control white space behavior. Because of the sheer amount of elements, you can choose between two views that group the available options in different ways to provide a somewhat greater flexibility when adjusting the more than 150 individual options.

Choose view

Lets you choose between different views in which the white space options are presented to the user. The available choices are:

- Group by Java Token

  The token view groups the white space options by the Java separator and operator tokens like commas, parentheses or assignments. This is the default and preferred view, because ideally it takes little more than twenty adjustments to configure behavior for all available options.

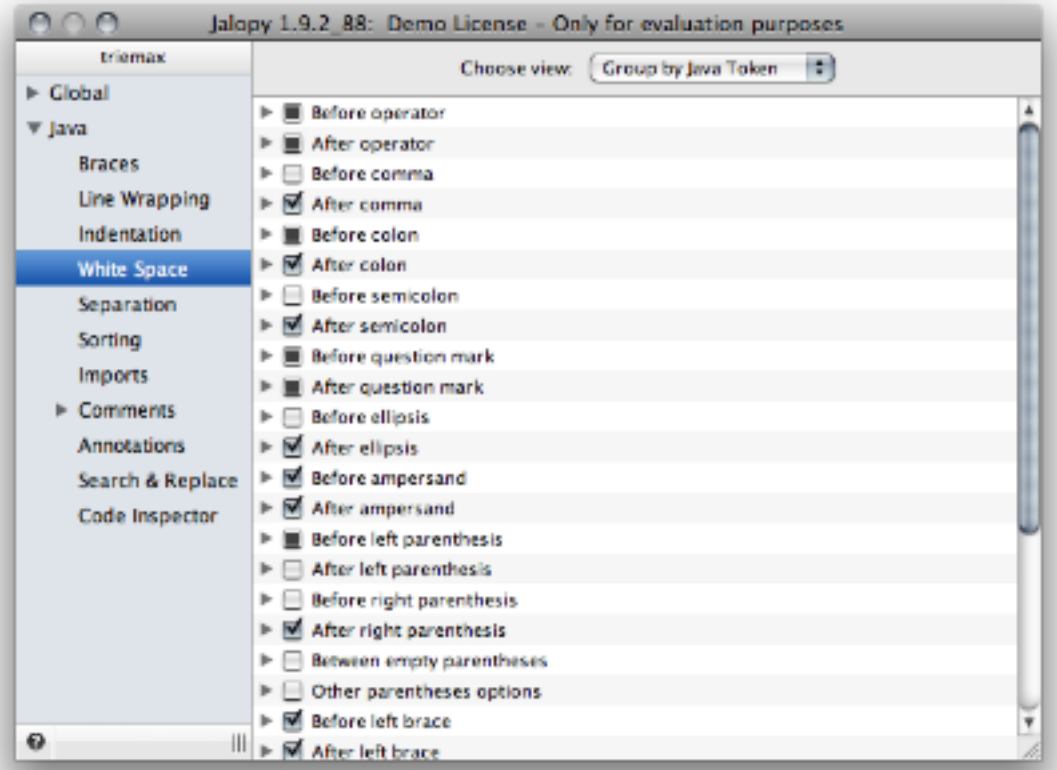
- Groupy by Java Element

The element view groups the white space options logically by the different available elements like declarations, control statements or expressions. This way it is easier to adjust options for just one element as all related options are presented together.

Since 1.6

2.8.9.1 Token view

Figure 2.44. White Space Token View



Before operator

Lets you specify what operators should have a blank space printed before.

Assignment operator

Controls whether a blank space will be printed before assignment operators. The assignment operators are: `= += -= *= \= %= &= |= ^= <<= >>= >>>=`

Example 2.332. Assignment operator

```
a=(b+c)*d;
a+=12;
```

Example 2.333. Assignment operator with space before

```
a =(b+c)*d;
a +=12;
```

Assignment operator in annotations

Controls whether a blank space will be printed before assignment operators in annotations.

Since 1.9

Example 2.334. Assignment operator

```
@Name(first="Joe",last="Hacker")
```

Example 2.335. Assignment operator with space before

```
@Name(first ="Joe",last ="Hacker")
```

Bitwise operator

Controls whether a blank space will be printed before bitwise operators. The bitwise operators are: `& | ^`

Example 2.336. Bitwise operator

```
return(getOperatingSystem()&PLAT_UNIX)!=0;
```

Example 2.337. Bitwise operator with space before

```
return(getOperatingSystem() &PLAT_UNIX)!=0;
```

Logical operator

Controls whether a blank space will be printed before logical operators. The logical operators are: `&& ||`

Example 2.338. Logical operator

```
if((LA(1)=='/')&&(LA(2)!='*'||(LA(2)=='*'&&LA(3)!='*'))) ...
```

Example 2.339. Logical operator with spaces around

```
if((LA(1)=='/') &&(LA(2)!='*' ||(LA(2)=='*' &&LA(3)!='*'))) ...
```

Math operator

Controls whether a blank space will be printed before mathematical operators. The mathematical operators are: `+ - / * %`

Example 2.340. Mathematical operator

```
a=(b+c)*d;
```

Example 2.341. Mathematical operator with space before

```
a=(b +c) *d;
```

Relational operator

Controls whether a blank space will be printed before relational operators. The relational operators are: `== != < > <= >=`

Example 2.342. Relational operator

```
if((LA(1)=='\n'||LA(1)=='\r')) ...
```

Example 2.343. Relational operator with space before

```
if((LA(1) =='\n'||LA(1) =='\r')) ...
```

Shift operator

Controls whether a blank space will be printed before shift operators. The shift operators are: `<< >> >>>`

Example 2.344. Shift operator

```
if(((1L<<i)&l)!=0) ...
```

Example 2.345. Shift operator with space before

```
if(((1L <<i)&l)!=0) ...
```

Postfix operator

Controls whether a blank space will be printed before postfix operators. The postfix operators are: ++ --

Since 1.6

Example 2.346. Postfix operator

```
int next = i++;
```

Example 2.347. Postfix operator with space before

```
int next = i ++;
```

String concat operator

Controls whether a blank space will be printed before the string concat operator.

Since 1.0.3

Example 2.348. String concat operator

```
a="a"+1;
b=1+"b";
c=1+2+3+"c";
d="d"+1+2+3;
e="e"+(1+2)+"e";
```

Example 2.349. String concat operator with space before

```
a="a" +1;
b=1 +"b";
c=1+2+3 +"c";
d="d" +1 +2 +3;
e="e" +(1+2) +"e";
```

After operator

Lets you specify what operators should have a blank space printed after.

Assignment operator

Controls whether a blank space will be printed after assignment operators. The assignment operators are: = += -= *= \= %= &= |= ^= <<= >>= >>>=

Example 2.350. Assignment operator

```
a=(b+c)*d;
a+=12;
```

Example 2.351. Assignment operator with space after

```
a= (b+c)*d;
a+= 12;
```

Assignment operator in annotations

Controls whether a blank space will be printed after assignment operators in annotations.

Since 1.9

Example 2.352. Assignment operator

```
@Name(first="Joe",last="Hacker")
```

Example 2.353. Assignment operator with space after

```
@Name(first= "Joe",last= "Hacker")
```

Bitwise operator

Controls whether a blank space will be printed after bitwise operators. The bitwise operators are: `& | ^`

Example 2.354. Bitwise operator

```
return(getOperatingSystem()&PLAT_UNIX)!=0;
```

Example 2.355. Bitwise operator with space after

```
return(getOperatingSystem()& PLAT_UNIX)!=0;
```

Logical operator

Controls whether a blank space will be printed after logical operators. The logical operators are: `&& ||`

Example 2.356. Logical operator

```
if((LA(1)=='/')&&(LA(2)!='*'||(LA(2)=='*'&&LA(3)!='*'))) ...
```

Example 2.357. Logical operator with spaces around

```
if((LA(1)=='/')&& (LA(2)!='*'|| (LA(2)=='*'&& LA(3)!='*'))) ...
```

Complement operator

Controls whether a blank space will be printed after complement operators. The logical operators are: `~ !`

Example 2.358. Complement operator

```
f(!x);
```

Example 2.359. Complement operator with space after

```
f(! x);
```

Mathematical operator

Controls whether a blank space will be printed after mathematical operators. The mathematical operators are: `+ - / * %`

Example 2.360. Mathematical operator

```
a=(b+c)*d;
```

Example 2.361. Mathematical operator with space after

```
a=(b+ c)* d;
```

Relational operator

Controls whether a blank space will be printed after relational operators. The relational operators are: `== != < > <= >=`

Example 2.362. Relational operator

```
if((LA(1)=='\n'||LA(1)=='\r')) ...
```

Example 2.363. Relational operator with space after

```
if((LA(1)== '\n'||LA(1)== '\r')) ...
```

Shift operators

Controls whether a blank space will be printed after shift operators. The shift operators are: << >> >>>

Example 2.364. Shift operator

```
if(((1L<<i)&l)!=0) ...
```

Example 2.365. Shift operator with space after

```
if(((1L<< i)&l)!=0) ...
```

Prefix operator

Controls whether a blank space will be printed after prefix operators. The prefix operators are: ++ --

Example 2.366. Prefix operator

```
int previous = --i;
```

Example 2.367. Prefix operator with space after

```
int previous = -- i;
```

Unary operator

Controls whether a blank space will be printed after unary operators. The unary operators are: - +

Since 1.6

Example 2.368. Unary operator

```
int x  = 3 * -4;
```

Example 2.369. Unary operator with space after

```
int x  = 3 * - 4;
```

String concat operator

Controls whether a blank space will be printed after the string concat operator.

Since 1.0.3

Example 2.370. String concat operator

```
a="a"+1;
b=1+"b";
c=1+2+3+"c";
d="d"+1+2+3;
e="e"+(1+2)+"e";
```

Example 2.371. String concat operator with space after

```
a="a"+ 1;
b=1+ "b";
c=1+2+3+ "c";
d="d"+ 1+ 2+ 3;
e="e"+ (1+2)+ "e";
```

Before comma

Lets you specify what commas should have a blank space printed before.

Annotation array

Controls whether a blank space will be printed before commas of annotation arrays.

Example 2.372. Annotation array

```
@Target({FIELD,METHOD,CONSTRUCTOR})
public class Foo { }
```

Example 2.373. Annotation array with space before comma

```
@Target({FIELD ,METHOD ,CONSTRUCTOR})
public class Foo { }
```

Annotation type member argument

Controls whether a blank space will be printed before commas of annotation member arguments.

Example 2.374. Annotation

```
@Point(x=23,y=-3)
public class Foo { }
```

Example 2.375. Annotation with space before comma

```
@Point(x=23 ,y=-3)
public class Foo { }
```

Enum constant

Controls whether a blank space will be printed before commas of enum constants.

Example 2.376. Enum constants

```
enum Color{GREEN,BLUE}
```

Example 2.377. Enum constants with space before comma

```
enum Color{GREEN ,BLUE}
```

Enum constant argument

Controls whether a blank space will be printed before commas of enum constants.

Example 2.378. Enum constant arguments

```
enum Color{GREEN(0,255.0),BLUE(0,0,255)}
```

Example 2.379. Enum constant arguments with space before comma

```
enum Color{GREEN(0 ,255 .0),BLUE(0 ,0 ,255)}
```

extends/implements

Controls whether a blank space will be printed before the commas of extends and/or implements types.

Extends type

Controls whether a blank space will be printed before the commas of extends types.

Example 2.380. Extends types

```
interface Fooable extends Doable,Readable {}
```

Example 2.381. Extends types with space before comma

```
interface Fooable extends Doable ,Readable {}
```

Extends type

Controls whether a blank space will be printed before the commas of implements types.

Example 2.382. Implements types

```
class Foo implements I0,I1,I2 {}
```

Example 2.383. Implements types with space before comma

```
class Foo implements I0 ,I1 ,I2 {}
```

Multiple declarations

Controls whether a blank space will be printed before the commas of multi-field and/or multi-variable declarations.

Field

Controls whether a blank space will be printed before the commas of multi-field declarations.

Example 2.384. Multi-field declaration

```
class Foo {
    int a=0,b=1,c=2,d=3;
}
```

Example 2.385. Multi-field declaration with space before commas

```
class Foo {
    int a=0 ,b=1 ,c=2 ,d=3;
}
```

Variable

Controls whether a blank space will be printed before the commas of multi-variable declarations.

Example 2.386. Multi-variable declaration

```
void foo() {
    int a=0,b=1,c=2,d=3;
}
```

Example 2.387. Multi-variable declaration with space before commas

```
void foo() {
    int a=0 ,b=1 ,c=2 ,d=3;
}
```

Declaration parameter

Controls whether a blank space will be printed before the commas of method and/or constructor declarations parameters.

Constructor

Example 2.388. Constructor declaration

```
Foo(int p1,int p2,int p3) {
}
```

Example 2.389. Constructor declaration with space before commas

```
Foo(int p1 ,int p2 ,int p3) {
}
```

Method

Example 2.390. Method declaration

```
void foo(int p1,int p2,int p3) {
}
```

Example 2.391. Method declaration with space before commas

```
void foo(int p1 ,int p2 ,int p3) {
}
```

Throws clauses

Controls whether a blank space will be printed before the commas of throws clauses of method and/or constructor declarations.

Constructor

Controls whether a blank space will be printed before the commas of throws clauses of constructor declarations.

Example 2.392. Constructor declaration throws clause

```
Foo() throws IOException,FooException {
}
```

Example 2.393. Constructor declaration throws clause with space before commas

```
Foo() throws IOException ,FooException {
}
```

Method

Controls whether a blank space will be printed before the commas of throws clauses of method declarations.

Example 2.394. Method declaration throws clause

```
void foo() throws IOException,FooException {
}
```

Example 2.395. Method declaration throws clause with space before commas

```
void foo() throws IOException ,FooException {
}
```

Call arguments

Controls whether a blank space will be printed before the commas of call arguments.

Constructor

Controls whether a blank space will be printed before the commas of constructor call arguments.

Example 2.396. Constructor call

```
Foo(int p1,int p2,int p3){
    super(p1,true);
}
```

Example 2.397. Constructor call space before commas

```
Foo(int p1,int p2,int p3){
    super(p1 ,true);
}
```

Method

Controls whether a blank space will be printed before the commas of method call arguments.

Example 2.398. Method call

```
test(x,y);
```

Example 2.399. Method call space before commas

```
test(x ,y);
```

Creator

Controls whether a blank space will be printed before the commas of creator call arguments.

Example 2.400. Creator call

```
Point point=new Point(x,y);
```

Example 2.401. Creator call space before commas

```
Point point=new Point(x ,y);
```

Array initializer

Controls whether a blank space will be printed before the commas of array initializers.

Example 2.402. Array initializer

```
int[] foo=new int[]{1,2,3};
```

Example 2.403. Array initializer with space before commas

```
int[] foo=new int[]{1 ,2 ,3};
```

for

Controls whether a blank space will be printed before the commas of for initializer and/or incrementor parts.

Initializer

Controls whether a blank space will be printed before the commas of for initializer parts.

Example 2.404. for initializer

```
for(int i=0,j=array.length;i<array.length;i++) {}
```

Example 2.405. for initializer with space before commas

```
for(int i=0 ,j=array.length;i<array.length;i++) {}
```

Incrementor

Controls whether a blank space will be printed before the commas of for incrementor parts.

Example 2.406. for incrementor

```
for(int i=0,j=array.length;i<array.length;i++,j--) {}
```

Example 2.407. for incrementor with space before commas

```
for(int i=0,j=array.length;i<array.length;i++ ,j--) {}
```

Parameterized types

Controls whether a blank space will be printed before the commas of parameterized types.

Type parameter

Controls whether a blank space will be printed before the commas of type parameters.

Example 2.408. Type parameter

```
class GenericType<S,T>{}
```

Example 2.409. Type parameter with space before commas

```
class GenericType<S ,T>{}
```

Type argument

Controls whether a blank space will be printed before the commas of type arguments.

Example 2.410. Type argument

```
caller.<String,Element>foo();
```

Example 2.411. Type argument with space before commas

```
caller.<String ,Element>foo();
```

After comma

Lets you specify what commas should have a blank space printed after.

Annotation array

Controls whether a blank space will be printed after commas of annotation arrays.

Example 2.412. Annotation array

```
@Target({FIELD,METHOD,CONSTRUCTOR})
public class Foo { }
```

Example 2.413. Annotation array with space after comma

```
@Target({FIELD, METHOD, CONSTRUCTOR})
public class Foo { }
```

Annotation type member argument

Controls whether a blank space will be printed after commas of annotation member arguments.

Example 2.414. Annotation

```
@Point(x=23,y=-3)
public class Foo { }
```

Example 2.415. Annotation with space after comma

```
@Point(x=23, y=-3)
public class Foo { }
```

Enum constant

Controls whether a blank space will be printed after commas of enum constants.

Example 2.416. Enum constants

```
enum Color{GREEN,BLUE}
```

Example 2.417. Enum constants with space after comma

```
enum Color{GREEN, BLUE}
```

Enum constant argument

Controls whether a blank space will be printed after commas of enum constants.

Example 2.418. Enum constant arguments

```
enum Color{GREEN(0,255.0),BLUE(0,0,255)}
```

Example 2.419. Enum constant arguments with space after comma

```
enum Color{GREEN(0, 255, 0),BLUE(0, 0, 255)}
```

extends/implements

Controls whether a blank space will be printed after the commas of extends and/or implements types.

Extends type

Controls whether a blank space will be printed after the commas of extends types.

Example 2.420. Extends types

```
interface Fooable extends Doable,Readable {}
```

Example 2.421. Extends types with space after comma

```
interface Fooable extends Doable, Readable {}
```

Implements type

Controls whether a blank space will be printed after the commas of implements types.

Example 2.422. Implements types

```
class Foo implements I0,I1,I2 {}
```

Example 2.423. Implements types with space after comma

```
class Foo implements I0, I1, I2 {}
```

IMultiple declarations

Controls whether a blank space will be printed after the commas of multi-field and/or multi-variable declarations.

Field

Controls whether a blank space will be printed after the commas of multi-field declarations.

Example 2.424. Multi-field declaration

```
class Foo {
    int a=0,b=1,c=2,d=3;
}
```

Example 2.425. Multi-field declaration with space after commas

```
class Foo {
    int a=0, b=1, c=2, d=3;
}
```

Variable

Controls whether a blank space will be printed after the commas of multi-variable declarations.

Example 2.426. Multi-variable declaration

```
void foo() {
    int a=0,b=1,c=2,d=3;
}
```

Example 2.427. Multi-variable declaration with space after commas

```
void foo() {
    int a=0, b=1, c=2, d=3;
}
```

Declaration parameter

Controls whether a blank space will be printed after the commas of method and/or constructor declarations parameters.

Constructor

Controls whether a blank space will be printed after the commas of constructor declarations parameters.

Example 2.428. Constructor declaration

```
Foo(int p1,int p2,int p3) {
}
```

Example 2.429. Constructor declaration with space after commas

```
Foo(int p1, int p2, int p3) {
}
```

Method

Example 2.430. Method declaration

```
void foo(int p1,int p2,int p3) {
}
```

Example 2.431. Method declaration with space after commas

```
void foo(int p1, int p2, int p3) {
}
```

Throws clauses

Controls whether a blank space will be printed after the commas of throws clauses of method and/or constructor declarations.

Constructor

Controls whether a blank space will be printed after the commas of throws clauses of constructor declarations.

Example 2.432. Constructor declaration throws clause

```
Foo() throws IOException,FooException {
}
```

Example 2.433. Constructor declaration throws clause with space after commas

```
Foo() throws IOException, FooException {
}
```

Method

Controls whether a blank space will be printed after the commas of throws clauses of method declarations.

Example 2.434. Method declaration throws clause

```
void foo() throws IOException,FooException {
}
```

Example 2.435. Method declaration throws clause with space after commas

```
void foo() throws IOException, FooException {
}
```

Call arguments

Controls whether a blank space will be printed after the commas of call arguments.

Constructor

Controls whether a blank space will be printed after the commas of constructor call arguments.

Example 2.436. Constructor call

```
Foo(int p1,int p2,int p3){
    super(p1,true);
}
```

Example 2.437. Constructor call space after commas

```
Foo(int p1,int p2,int p3){
    super(p1, true);
}
```

Method

Controls whether a blank space will be printed after the commas of method call arguments.

Example 2.438. Method call

```
test(x,y);
```

Example 2.439. Method call space after commas

```
test(x, y);
```

Creator

Controls whether a blank space will be printed after the commas of creator call arguments.

Example 2.440. Creator call

```
Point point=new Point(x,y);
```

Example 2.441. Creator call space after commas

```
Point point=new Point(x, y);
```

Array initializer

Controls whether a blank space will be printed after the commas of array initializers.

Example 2.442. Array initializer

```
int[] foo=new int[]{1,2,3};
```

Example 2.443. Array initializer with space after commas

```
int[] foo=new int[]{1, 2, 3};
```

for

Controls whether a blank space will be printed after the commas of for initializer and/or incrementor parts.

Initializer

Controls whether a blank space will be printed after the commas of for initializer parts.

Example 2.444. for initializer

```
for(int i=0,j=array.length;i<array.length;i++) {}
```

Example 2.445. for initializer with space after commas

```
for(int i=0, j=array.length;i<array.length;i++) {}
```

Incrementor

Controls whether a blank space will be printed after the commas of for incrementor parts.

Example 2.446. for incrementor

```
for(int i=0,j=array.length;i<array.length;i++,j--) {}
```

Example 2.447. for incrementor with space after commas

```
for(int i=0,j=array.length;i<array.length;i++, j--) {}
```

Parameterized types

Controls whether a blank space will be printed after the commas of parameterized types.

Type parameter

Controls whether a blank space will be printed after the commas of type parameters.

Example 2.448. Type parameter

```
class GenericType<S,T>{}
```

Example 2.449. Type parameter with space after commas

```
class GenericType<S, T>{}
```

Type argument

Controls whether a blank space will be printed after the commas of type arguments.

Example 2.450. Type argument

```
caller.<String,Element>foo();
```

Example 2.451. Type argument with space after commas

```
caller.<String, Element>foo();
```

Before colon

Lets you specify what colons should have a blank space printed before.

assert

Controls whether colons of assert statements should have a blank space printed before.

Example 2.452. assert statement

```
assert condition:reportError();
```

Example 2.453. assert statement with space before colon

```
assert condition :reportError();
```

case

Controls whether colons of case statements should have a blank space printed before.

Example 2.454. assert statement

```
switch (list[i]) {
case 't':
    break;
}
```

Example 2.455. assert statement with space before colon

```
switch (list[i]) {
case 't' :
    break;
}
```

Conditional

Controls whether colons of the conditional operator should have a blank space printed before.

Example 2.456. Conditional operator

```
String value=condition?TRUE:FALSE;
```

Example 2.457. Conditional operator with space before colon

```
String value=condition?TRUE :FALSE;
```

for

Controls whether colons of enhanced for statements should have a blank space printed before.

Example 2.458. Enhancement for statement

```
for (String s:names) { }
```

Example 2.459. Enhanced for statement with space before colon

```
for (String s :names) { }
```

Label

Controls whether colons of labeled statements should have a blank space printed before.

Example 2.460. Labeled statement

```
label: {
   ...
}
```

Example 2.461. Labeled statement with space before colon

```
label : {
    ...
}
```

After colon

Lets you specify what colons should have a blank space printed after.

assert

Controls whether colons of assert statements should have a blank space printed after.

Example 2.462. assert statement

```
assert condition:reportError();
```

Example 2.463. assert statement with space after colon

```
assert condition: reportError();
```

Conditional

Controls whether colons of the conditional operator should have a blank space printed after.

Example 2.464. Conditional operator

```
String value=condition?TRUE:FALSE;
```

Example 2.465. Conditional operator with space after colon

```
String value=condition?TRUE: FALSE;
```

for

Controls whether colons of enhanced for statements should have a blank space printed after.

Example 2.466. Enhancement for statement

```
for (String s:names) { }
```

Example 2.467. Enhanced for statement with space after colon

```
for (String s: names) { }
```

Label

Controls whether colons of labeled statements should have a blank space printed after.

Example 2.468. Labeled statement

```
label:for(;;){
   ...
}
```

Example 2.469. Labeled statement with space after colon

```
label: for(;;){
    ...
}
```

Please note that this option only applies when no line break is printed after the colon.

Before semicolon

Lets you specify what semicolons should have a blank space printed before.

for

Controls whether semicolons of for statements should have a blank space printed before.

Example 2.470. for statement

```
for(int i=0;i<array.length;i++) {}
```

Example 2.471. for statement with space before semicolon

```
for(int i=0 ;i<array.length ;i++) {}
```

After semicolon

Lets you specify what semicolons should have a blank space printed after.

for

Controls whether semicolons of for statements should have a blank space printed after.

Example 2.472. for statement

```
for(int i=0;i<array.length;i++) {}
```

Example 2.473. for statement with space after semicolon

```
for(int i=0; i<array.length; i++) {}
```

Before question mark

Lets you specify what question marks should have a blank space printed before.

Conditional operator

Controls whether question marks of the conditional operator should have a blank space printed before.

Example 2.474. Conditional operator

```
String value=condition?TRUE:FALSE;
```

Example 2.475. Conditional operator with space before question mark

```
String value=condition ?TRUE:FALSE;
```

Type parameter

Controls whether question marks of type parameters should have a blank space printed before. Please note that this option only applies if no white space after the left angle bracket is forced (See “Space after left bracket type parameter”).

Example 2.476. Type parameter

```
class QuestionMark<T extends Comparable< ? super Number>> {}
```

Example 2.477. Type parameter with space before question mark

```
class QuestionMark<T extends Comparable< ? super Number>> {}
```

Type argument

Controls whether question marks of type arguments should have a blank space printed before. Please note that this option only applies if no white space after the left angle bracket and/or commas is forced (See “Space after left bracket type argument”).

Example 2.478. Type argument

```
Map<X<?>,Y<? extends K,? super V>>t;
```

Example 2.479. Type argument with space before question mark

```
Map<X< ?>,Y< ? extends K, ? super V>>t;
```

After question mark

Lets you specify what question marks should have a blank space printed after.

Conditional operator

Controls whether question marks of the conditional operator should have a blank space printed after.

Example 2.480. Conditional operator

```
String value=condition?TRUE:FALSE;
```

Example 2.481. Conditional operator with space after question mark

```
String value=condition? TRUE:FALSE;
```

Type parameter

Controls whether question marks of type parameters should have a blank space printed after. Please note that this option only applies if no white space before the right angle bracket is forced (See "Space before right angle bracket type parameter").

Example 2.482. Type parameter

```
class X10<T extends Map.Entry<?,?>> {}
```

Example 2.483. Type parameter with space after question mark

```
class X10<T extends Map.Entry<? ,? >> {}
```

Type argument

Controls whether question marks of type arguments should have a blank space printed after. Please note that this option only applies if no white space before the right angle bracket and/or commas is forced (See "Space before right angle bracket type argument").

Example 2.484. Type argument

```
Map<X<?>,Y>t;
```

Example 2.485. Type argument with space after question mark

```
Map<X<? >,Y>t;
```

Before ellipsis

Lets you specify whether a blank space should be printed before the ellipsis.

Vararg

Controls whether a blank space will be printed before the ellipsis of a variable arity parameter (varag).

Since 1.2

Example 2.486. Vararg ellipsis

```
public void test(String[]...args) {
}
```

Example 2.487. Vararg ellipsis with space before

```
public void test(String[] ...args) {
}
```

After ellipsis

Lets you specify whether a blank space should be printed after the ellipsis.

Vararg

Controls whether a blank space will be printed after the ellipsis of a variable arity parameter (varag).

Since 1.6

Example 2.488. Vararg ellipsis

```
public void test(String[]...args) {
}
```

Example 2.489. Vararg ellipsis with space after

```
public void test(String[]... args) {
}
```

Before ampersand

Lets you specify whether a blank space should be printed before the ampersand.

Type parameter

Controls whether a blank space will be printed before the ampersand of type parameters.

Since 1.6

Example 2.490. Type parameter

```
class Foo<S,T extends Element&List> {
}
```

Example 2.491. Type parameter with space before ampersand

```
class Foo<S,T extends Element &List> {
}
```

After ampersand

Lets you specify whether a blank space should be printed after the ampersand.

Type parameter

Controls whether a blank space will be printed after the ampersand of type parameters.

Since 1.6

Example 2.492. Type parameter

```
class Foo<S,T extends Element&List> {
}
```

Example 2.493. Type parameter with space after ampersand

```
class Foo<S,T extends Element& List> {
}
```

Before left parenthesis

Lets you specify what left parentheses should have a blank space printed before.

Annotation argument list

Controls whether a blank space should be printed before the left parenthesis of annotation argument lists.

Example 2.494. Annotation

```
@Annot(x=23,y=-3)
class Foo {
}
```

Example 2.495. Annotation with space before argument list

```
@Annot (x=23,y=-3)
class Foo {
}
```

Annotation type member

Controls whether a blank space should be printed before the left parenthesis annotation type members.

Example 2.496. Annotation type member

```
@interface MyAnnotation {
    String value();
}
```

Example 2.497. Annotation type member with space left paren

```
@interface MyAnnotation {
    String value ();
}
```

Enum constant argument

Controls whether a blank space should be printed before the left parenthesis of enum constant argument lists.

Example 2.498. Enum constant

```
enum MyEnum {
    GREEN(0,255,0)
}
```

Example 2.499. Enum constant with space before left parenthesis

```
enum MyEnum {
    GREEN (0,255,0)
}
```

Declaration parameter

Controls whether a blank space will be printed before the left parenthesis of method and/or constructor parameter lists.

Constructor

Control whether a blank space will be printed before the left parenthesis of constructor parameter lists.

Example 2.500. Constructor declaration

```
Foo(int p1,int p2,int p3) {
}
```

Example 2.501. Constructor declaration with space before left parenthesis

```
Foo (int p1,int p2,int p3) {
}
```

Method

Control whether a blank space will be printed before the left parenthesis of method parameter lists.

Example 2.502. Method declaration

```
public void foo(int p1,int p2,int p3) {
}
```

Example 2.503. Method declaration with space before left parenthesis

```
public void foo (int p1,int p2,int p3) {
}
```

Statement expressions

Lets you control whether a blank space will be printed before the left parenthesis of statement expressions.

if

Lets you control whether a blank space will be printed before the left parenthesis of if expressions.

Example 2.504. if statement

```
if(condition) {
}
```

Example 2.505. if statement with space before left parenthesis

```
if (condition) {
}
```

for

Lets you control whether a blank space will be printed before the left parenthesis of for expressions.

Example 2.506. for statement

```
for(String s : names) {
}
```

Example 2.507. for statement with space before left parenthesis

```
for (String s : names) {
}
```

while

Lets you control whether a blank space will be printed before the left parenthesis of while expressions.

Example 2.508. while statement

```
while(condition) {
}
```

Example 2.509. while statement with space before left parenthesis

```
while (condition) {
}
```

switch

Lets you control whether a blank space will be printed before the left parenthesis of switch expressions.

Example 2.510. switch statement

```
switch(c) {
}
```

Example 2.511. switch statement with space before left parenthesis

```
switch (c) {
}
```

throw

Lets you control whether a blank space will be printed before the left parenthesis of throw expressions.

Example 2.512. throw statement

```
throw(new UnsupportOperationException());
```

Example 2.513. throw statement with space before left parenthesis

```
throw (new UnsupportOperationException());
```

synchronized

Lets you control whether a blank space will be printed before the left parenthesis of synchronized expressions.

Example 2.514. synchronized statement

```
synchronized(this) {
    performOperation();
}
```

Example 2.515. synchronized statement with space before left parenthesis

```
synchronized (this) {
    performOperation();
}
```

catch

Lets you control whether a blank space will be printed before the left parenthesis of catch expressions.

Example 2.516. catch statement

```
try {
    Integer.parseInt(value);
} catch(NumberFormatException ex) {
}
```

Example 2.517. catch statement with space before left parenthesis

```
try {
    Integer.parseInt(value);
} catch (NumberFormatException ex) {
}
```

return

Lets you control whether a blank space will be printed before the left parenthesis of return expressions.

Example 2.518. return statement

```
return(200 + (a * b));
```

Example 2.519. return statement with space before left parenthesis

```
return (200 + (a * b));
```

Call arguments

Controls whether a blank space will be printed before the left parenthesis of call arguments.

Constructor

Controls whether a blank space will be printed before the left parenthesis of constructor call arguments.

Example 2.520. Constructor call

```
Foo(int p1,int p2,int p3){
    super(p1,true);
}
```

Example 2.521. Constructor call with space before left parenthesis

```
Foo(int p1,int p2,int p3){
    super (p1,true);
}
```

Method

Controls whether a blank space will be printed before the left parenthesis of method call arguments.

Example 2.522. Method call

```
test(x,y);
```

Example 2.523. Method call space with space before left parenthesis

```
test (x,y);
```

Creator

Controls whether a blank space will be printed before the left parenthesis of creator call arguments.

Example 2.524. Creator call

```
Point point=new Point(x,y);
```

Example 2.525. Creator call with space before left parenthesis

```
Point point=new Point (x,y);
```

After left parenthesis

Lets you specify what left parentheses should have a blank space printed after.

Annotation argument list

Controls whether a blank space should be printed after the left parenthesis of annotation argument lists.

Example 2.526. Annotation

```
@Annot(x=23,y=-3)
class Foo {
}
```

Example 2.527. Annotation with space after left parenthesis

```
@Annot( x=23,y=-3)
class Foo {
}
```

Enum constant argument

Controls whether a blank space should be printed after the left parenthesis of enum constant argument lists.

Example 2.528. Enum constant

```
enum MyEnum {
    GREEN(0,255,0)
}
```

Example 2.529. Enum constant with space after left parenthesis

```
enum MyEnum {
    GREEN( 0,255,0)
}
```

Declaration parameter

Controls whether a blank space will be printed after the left parenthesis of method and/or constructor parameter lists.

Constructor

Control whether a blank space will be printed after the left parenthesis of constructor parameter lists.

Example 2.530. Constructor declaration

```
Foo(int p1,int p2,int p3) {
}
```

Example 2.531. Constructor declaration with space after left parenthesis

```
Foo( int p1,int p2,int p3) {
}
```

Method

Control whether a blank space will be printed after the left parenthesis of method parameter lists.

Example 2.532. Method declaration

```
public void foo(int p1,int p2,int p3) {
}
```

Example 2.533. Method declaration with space after left parenthesis

```
public void foo( int p1,int p2,int p3) {
}
```

Statement expressions

Lets you control whether a blank space will be printed before the left parenthesis of statement expressions.

if

Lets you control whether a blank space will be printed after the left parenthesis of if expressions.

Example 2.534. if statement

```
if(condition) {
}
```

Example 2.535. if statement with space after left parenthesis

```
if( condition) {
}
```

for

Lets you control whether a blank space will be printed after the left parenthesis of for expressions.

Example 2.536. for statement

```
for(String s : names) {
}
```

Example 2.537. for statement with space after left parenthesis

```
for( String s : names) {
}
```

while

Lets you control whether a blank space will be printed after the left parenthesis of while expressions.

Example 2.538. while statement

```
while(condition) {
}
```

Example 2.539. while statement with space after left parenthesis

```
while( condition) {
}
```

switch

Lets you control whether a blank space will be printed after the left parenthesis of switch expressions.

Example 2.540. switch statement

```
switch(c) {
}
```

Example 2.541. switch statement with space after left parenthesis

```
switch( c) {
}
```

throw

Lets you control whether a blank space will be printed after the left parenthesis of throw expressions.

Example 2.542. throw statement

```
throw(new UnsupportOperationException());
```

Example 2.543. throw statement with space after left parenthesis

```
throw( new UnsupportOperationException());
```

synchronized

Lets you control whether a blank space will be printed after the left parenthesis of synchronized expressions.

Example 2.544. synchronized statement

```
synchronized(this) {
    performOperation();
}
```

Example 2.545. synchronized statement with space after left parenthesis

```
synchronized( this) {
    performOperation();
}
```

catch

Lets you control whether a blank space will be printed after the left parenthesis of catch expressions.

Example 2.546. catch statement

```
try {
    Integer.parseInt(value);
} catch(NumberFormatException ex) {
}
```

Example 2.547. catch statement with space after left parenthesis

```
try {
    Integer.parseInt(value);
} catch( NumberFormatException ex) {
}
```

return

Lets you control whether a blank space will be printed after the left parenthesis of return expressions.

Example 2.548. return statement

```
return(200 + (a * b));
```

Example 2.549. return statement with space after left parenthesis

```
return( 200 + (a * b));
```

Call arguments

Controls whether a blank space will be printed after the left parenthesis of call arguments.

Constructor

Controls whether a blank space will be printed after the left parenthesis of constructor call arguments.

Example 2.550. Constructor call

```
Foo(int p1,int p2,int p3){
    super(p1,true);
}
```

Example 2.551. Constructor call with space after left parenthesis

```
Foo(int p1,int p2,int p3){
    super( p1,true);
}
```

Method

Controls whether a blank space will be printed after the left parenthesis of method call arguments.

Example 2.552. Method call

```
test(x,y);
```

Example 2.553. Method call space with space after left parenthesis

```
test( x,y);
```

Creator

Controls whether a blank space will be printed after the left parenthesis of creator call arguments.

Example 2.554. Creator call

```
Point point=new Point(x,y);
```

Example 2.555. Creator call with space after left parenthesis

```
Point point=new Point( x,y);
```

Parenthesized expression

Controls whether a blank space will be printed after the left parenthesis of parenthesized expressions..

Example 2.556. Expression

```
int r = (a * (b + c + d) * (e + f));
```

Example 2.557. Expression with space after left parenthesis

```
int r = ( a * ( b + c + d) * ( e + f));
```

Type cast

Controls whether a blank space will be printed after the left parenthesis of type casts.

Example 2.558. Type cast

```
LineManager m = (LineManager)a.getParent();
```

Example 2.559. Type cast with space after left parenthesis

```
LineManager m = ( LineManager)a.getParent();
```

Before right parenthesis

Lets you specify what right parentheses should have a blank space printed before.

Annotation argument list

Controls whether a blank space should be printed before the right parenthesis of annotation argument lists.

Example 2.560. Annotation

```
@Annot(x=23,y=-3)
class Foo {
}
```

Example 2.561. Annotation with space before right parenthesis

```
@Annot(x=23,y=-3 )
class Foo {
}
```

Enum constant argument

Controls whether a blank space should be printed before the right parenthesis of enum constant argument lists.

Example 2.562. Enum constant

```
enum MyEnum {
    GREEN(0,255,0)
}
```

Example 2.563. Enum constant with space before right parenthesis

```
enum MyEnum {
    GREEN(0,255,0 )
}
```

Declaration parameter

Controls whether a blank space will be printed after the left parenthesis of method and/or constructor parameter lists.

Constructor

Control whether a blank space will be printed before the right parenthesis of constructor parameter lists.

Example 2.564. Constructor declaration

```
Foo(int p1,int p2,int p3) {
}
```

Example 2.565. Constructor declaration with space before right parenthesis

```
Foo(int p1,int p2,int p3 ) {
}
```

Method

Control whether a blank space will be printed before the right parenthesis of method parameter lists.

Example 2.566. Method declaration

```
public void foo(int p1,int p2,int p3) {
}
```

Example 2.567. Method declaration with space before right parenthesis

```
public void foo(int p1,int p2,int p3 ) {
}
```

Statement expressions

Lets you control whether a blank space will be printed before the left parenthesis of statement expressions.

if

Lets you control whether a blank space will be printed before the right parenthesis of if expressions.

Example 2.568. if statement

```
if(condition) {
}
```

Example 2.569. if statement with space before right parenthesis

```
if(condition ) {
}
```

for

Lets you control whether a blank space will be printed before the right parenthesis of for expressions.

Example 2.570. for statement

```
for(String s : names) {
}
```

Example 2.571. for statement with space before right parenthesis

```
for(String s : names ) {
}
```

while

Lets you control whether a blank space will be printed before the right parenthesis of while expressions.

Example 2.572. while statement

```
while(condition) {
}
```

Example 2.573. while statement with space before right parenthesis

```
while(condition ) {
}
```

switch

Lets you control whether a blank space will be printed before the right parenthesis of switch expressions.

Example 2.574. switch statement

```
switch(c) {
}
```

Example 2.575. switch statement with space before right parenthesis

```
switch(c ) {
}
```

throw

Lets you control whether a blank space will be printed before the right parenthesis of throw expressions.

Example 2.576. throw statement

```
throw(new UnsupportOperationException());
```

Example 2.577. throw statement with space before right parenthesis

```
throw(new UnsupportOperationException() );
```

synchronized

Lets you control whether a blank space will be printed before the right parenthesis of synchronized expressions.

Example 2.578. synchronized statement

```
synchronized(this) {
    performOperation();
}
```

Example 2.579. synchronized statement with space before right parenthesis

```
synchronized(this ) {
    performOperation();
}
```

catch

Lets you control whether a blank space will be printed before the right parenthesis of catch expressions.

Example 2.580. catch statement

```
try {
    Integer.parseInt(value);
} catch(NumberFormatException ex) {
}
```

Example 2.581. catch statement with space before right parenthesis

```
try {
    Integer.parseInt(value);
} catch(NumberFormatException ex ) {
}
```

return

Lets you control whether a blank space will be printed before the right parenthesis of return expressions.

Example 2.582. return statement

```
return(200 + (a * b));
```

Example 2.583. return statement with space before right parenthesis

```
return(200 + (a * b) );
```

Call arguments

Controls whether a blank space will be printed before the right parenthesis of call arguments.

Constructor

Controls whether a blank space will be printed before the right parenthesis of constructor call arguments.

Example 2.584. Constructor call

```
Foo(int p1,int p2,int p3){
    super(p1,true);
}
```

Example 2.585. Constructor call with space before right parenthesis

```
Foo(int p1,int p2,int p3){
    super(p1,true );
}
```

Method

Controls whether a blank space will be printed before the right parenthesis of method call arguments.

Example 2.586. Method call

```
test(x,y);
```

Example 2.587. Method call space with space before right parenthesis

```
test(x,y );
```

Creator

Controls whether a blank space will be printed before the right parenthesis of creator call arguments.

Example 2.588. Creator call

```
Point point=new Point(x,y);
```

Example 2.589. Creator call with space before right parenthesis

```
Point point=new Point(x,y );
```

Parenthesized expression

Controls whether a blank space will be printed before the right parenthesis of parenthesized expressions..

Example 2.590. Expression

```
int r = (a * (b + c + d) * (e + f));
```

Example 2.591. Expression with space before right parenthesis

```
int r = (a * (b + c + d ) * (e + f ) );
```

Type cast

Controls whether a blank space will be printed before the right parenthesis of type casts.

Example 2.592. Type cast

```
LineManager m = (LineManager)a.getParent();
```

Example 2.593. Type cast with space before right parenthesis

```
LineManager m = (LineManager )a.getParent();
```

After right parenthesis

Lets you specify what right parentheses should have a blank space printed after.

Type cast

Controls whether a blank space will be printed after the right parenthesis of type casts.

Example 2.594. Type cast

```
LineManager m = (LineManager)a.getParent();
```

Example 2.595. Type cast with space after right parenthesis

```
LineManager m = (LineManager) a.getParent();
```

Between empty parentheses

Lets you specify what empty parentheses should have a blank space printed between.

Annotation type member

Controls whether a blank space should be printed between the empty parentheses of annotation type members.

Example 2.596. Annotation type member

```
@interface MyAnnotation {
    String value();
}
```

Example 2.597. Annotation type member with space between empty parentheses

```
@interface MyAnnotation {
    String value( );
}
```

Enum constant argument

Controls whether a blank space should be printed between the empty parentheses of enum constant argument lists.

Example 2.598. Enum constant

```
enum MyEnum {
    GREEN()
}
```

Example 2.599. Enum constant with space between empty parentheses

```
enum MyEnum {
    GREEN( )
}
```

Declaration parameter

Controls whether a blank space will be printed between the empty parentheses of method and/or constructor parameter lists.

Constructor

Control whether a blank space will be printed between the empty parentheses of constructor parameter lists.

Example 2.600. Constructor declaration

```
Foo() {
}
```

Example 2.601. Constructor declaration with space between empty parentheses

```
Foo( ) {
}
```

Method

Control whether a blank space will be printed before the empty parentheses of method parameter lists.

Example 2.602. Method declaration

```
public void foo() {
}
```

Example 2.603. Method declaration with space between empty parentheses

```
public void foo( ) {
}
```

Call arguments

Controls whether a blank space will be printed between the empty parentheses of call arguments.

Constructor

Controls whether a blank space will be printed between the empty parentheses of constructor call arguments.

Example 2.604. Constructor call

```
Foo(int p1,int p2,int p3){
    super();
}
```

Example 2.605. Constructor call with space between empty parentheses

```
Foo(int p1,int p2,int p3){
    super( );
}
```

Method

Controls whether a blank space will be printed between the empty parentheses of method call arguments.

Example 2.606. Method call

```
test();
```

Example 2.607. Method call space with space between empty parenthesis

```
test( );
```

Creator

Controls whether a blank space will be printed between the empty parentheses of creator call arguments.

Example 2.608. Creator call

```
Point point=new Point();
```

Example 2.609. Creator call with space between empty parentheses

```
Point point=new Point( );
```

Other parentheses

Lets you control some general parentheses behavior.

Same direction parentheses

When enabled, no white space will be printed before or after parentheses with the same direction.

Naturally, this option is only meaningful if any of the space after left parenthesis/space before right parenthesis options have been enabled.

Since 1.0.1

Example 2.610. Parentheses with same direction

```
if ( ( LA( 1 ) == '/' ) && ( LA( 2 ) != '*' ) )
    ...
```

Example 2.611. Parentheses with same direction (compacted)

```
if (( LA ( 1 ) == '/' ) && ( LA( 2 ) != '*' ))
    ...
```

Before left brace

Controls whether a blank space should be printed before the left curly brace.

Compact declaration

Controls whether a blank space should be printed before the left curly brace of compacted declaration blocks.

Example 2.612. Compact method declaration

```
void foo(){int i = 1;}
```

Example 2.613. Compact method declaration with space before left curly brace

```
void foo() {int i = 1;}
```

Array initializer

Controls whether a blank space should be printed before the left curly brace of array initializers that fit into one line.

Example 2.614. Array initializer

```
String[] first=new String[]{"1", "2"};
```

Example 2.615. Array initializer with space before left curly brace

```
String[] first=new String[] {"1", "2"};
```

After left brace

Controls whether a blank space should be printed after left curly braces.

Annotation array

Controls whether a blank space should be printed after the left curly brace of annotation arrays.

Example 2.616. Annotation array

```
@Target({FIELD, METHOD, CONSTRUCTOR})
class FOO {
}
```

Example 2.617. Annotation array with space after left curly brace

```
@Target({ FIELD, METHOD, CONSTRUCTOR})
class FOO {
}
```

Compact declaration

Controls whether a blank space should be printed after the left curly brace of compacted declaration blocks.

Example 2.618. Compact method declaration

```
void foo(){int i = 1;}
```

Example 2.619. Compact method declaration with space after left curly brace

```
void foo(){ int i = 1;}
```

Array initializer

Controls whether a blank space should be printed after the left curly brace of array initializers.

Example 2.620. Array initializer

```
String[] first=new String[]{"1", "2"};
```

Example 2.621. Array initializer with space after left curly brace

```
String[] first=new String[]{ "1", "2"};
```

Before right brace

Controls whether a blank space should be printed after left curly braces.

Annotation array

Controls whether a blank space should be printed before the right curly brace of annotation arrays.

Example 2.622. Annotation array

```
@Target({FIELD, METHOD, CONSTRUCTOR})
class FOO {
}
```

Example 2.623. Annotation array with space before right curly brace

```
@Target({FIELD, METHOD, CONSTRUCTOR })
class FOO {
}
```

Compact declaration

Controls whether a blank space should be printed before the right curly brace of compacted declaration blocks.

Example 2.624. Compact method declaration

```
void foo(){int i = 1;}
```

Example 2.625. Compact method declaration with space before right curly brace

```
void foo(){int i = 1; }
```

Array initializer

Controls whether a blank space should be printed before the right curly brace of array initializers.

Example 2.626. Array initializer

```
String[] first=new String[]{"1", "2"};
```

Example 2.627. Array initializer with space before right curly brace

```
String[] first=new String[]{"1", "2" };
```

Between empty braces

Controls whether a blank space should be printed between empty braces.

Compact declaration

Controls whether a blank space should be printed between empty curly braces of compacted declaration blocks.

Example 2.628. Compact method declaration

```
void foo(){}
```

Example 2.629. Compact method declaration with space between empty curly braces

```
void foo(){ }
```

Array initializer

Controls whether a blank space should be printed between empty curly braces of array initializers.

Example 2.630. Array initializer

```
String[] first=new String[]{};
```

Example 2.631. Array initializer with space between empty braces

```
String[] first=new String[]{ };
```

Before left bracket

Controls whether a blank space should be printed before left brackets.

Array declaration

Controls whether a blank space should be printed before the left bracket of array declaration statements.

Example 2.632. Array declaration statement

```
String[] first={};
```

Example 2.633. Array declaration statement with space before left bracket

```
String [] first={};
```

Array creator

Controls whether a blank space should be printed before the left bracket of array creation statements.

Example 2.634. Array creator statement

```
String[] third=new String[3];
```

Example 2.635. Array creator statement with space before left bracket

```
String[] third=new String [3];
```

Array accessor

Controls whether a blank space should be printed before the left bracket of array access statements.

Example 2.636. Array accessor

```
value=third[3];
```

Example 2.637. Array accessor with space before left bracket

```
value=third [3];
```

After left bracket

Controls whether a blank space should be printed after left brackets.

Array creator

Controls whether a blank space should be printed after the left bracket of array creation statements.

Example 2.638. Array creator statement

```
String[] third=new String[3];
```

Example 2.639. Array creator statement with space after left bracket

```
String[] third=new String[ 3];
```

Array accessor

Controls whether a blank space should be printed after the left bracket of array access statements.

Example 2.640. Array accessor

```
value=third[3];
```

Example 2.641. Array accessor with space after left bracket

```
value=third[ 3];
```

Before right bracket

Controls whether a blank space should be printed before right brackets.

Array creator

Controls whether a blank space should be printed before the right bracket of array creation statements.

Example 2.642. Array creator

```
String[] third=new String[3];
```

Example 2.643. Array creator with space before right bracket

```
String[] third=new String[3 ];
```

Array accessor

Controls whether a blank space should be printed before the right bracket of array access statements.

Example 2.644. Array accessor

```
value=third[3];
```

Example 2.645. Array accessor with space before right bracket

```
value=third[3 ];
```

Between empty brackets

Controls whether a blank space should be printed between empty brackets.

Array declaration

Controls whether a blank space should be printed between empty brackets of array declaration statements.

Example 2.646. Array declaration statement

```
String[] first={};
```

Example 2.647. Array declaration statement with space between empty bracket

```
String[ ] first={};
```

Array creator

Controls whether a blank space should be printed between empty brackets of array creator statements.

Example 2.648. Array creator statement

```
String[] first=new String[]{};
```

Example 2.649. Array creator statement with space between empty bracket

```
String[] first=new String[ ]{};
```

Before left angle bracket

Controls whether a blank space should be printed before left angle brackets of parameterized types.

Type parameter

Controls whether a blank space should be printed before left angle brackets of type parameters.

Example 2.650. Type parameter

```
class AngleBracket<S,T extends Element> {}
```

Example 2.651. Type parameter with space before left angle bracket

```
class AngleBracket <S,T extends Element> {}
```

Type argument

Controls whether a blank space should be printed before left angle brackets of type arguments.

Example 2.652. Type argument

```
caller.<String,Element>foo();
```

Example 2.653. Type argument with space before left angle bracket

```
caller. <String,Element>foo();
```

After left angle bracket

Controls whether a blank space should be printed after left angle brackets of parameterized types.

Type parameter

Controls whether a blank space should be printed after left angle brackets of type parameters.

Example 2.654. Type parameter

```
class AngleBracket<S,T extends Element> {}
```

Example 2.655. Type parameter with space after left angle bracket

```
class AngleBracket< S,T extends Element> {}
```

Type argument

Controls whether a blank space should be printed after left angle brackets of type arguments.

Example 2.656. Type argument

```
caller.<String,Element>foo();
```

Example 2.657. Type argument with space after left angle bracket

```
caller.< String,Element>foo();
```

Before right angle bracket

Controls whether a blank space should be printed before right angle brackets of parameterized types.

Type parameter

Controls whether a blank space should be printed before right angle brackets of type parameters.

Example 2.658. Type parameter

```
class AngleBracket<S,T extends Element> {}
```

Example 2.659. Type parameter with space before right angle bracket

```
class AngleBracket<S,T extends Element > {}
```

Type argument

Controls whether a blank space should be printed before right angle brackets of type arguments.

Example 2.660. Type argument

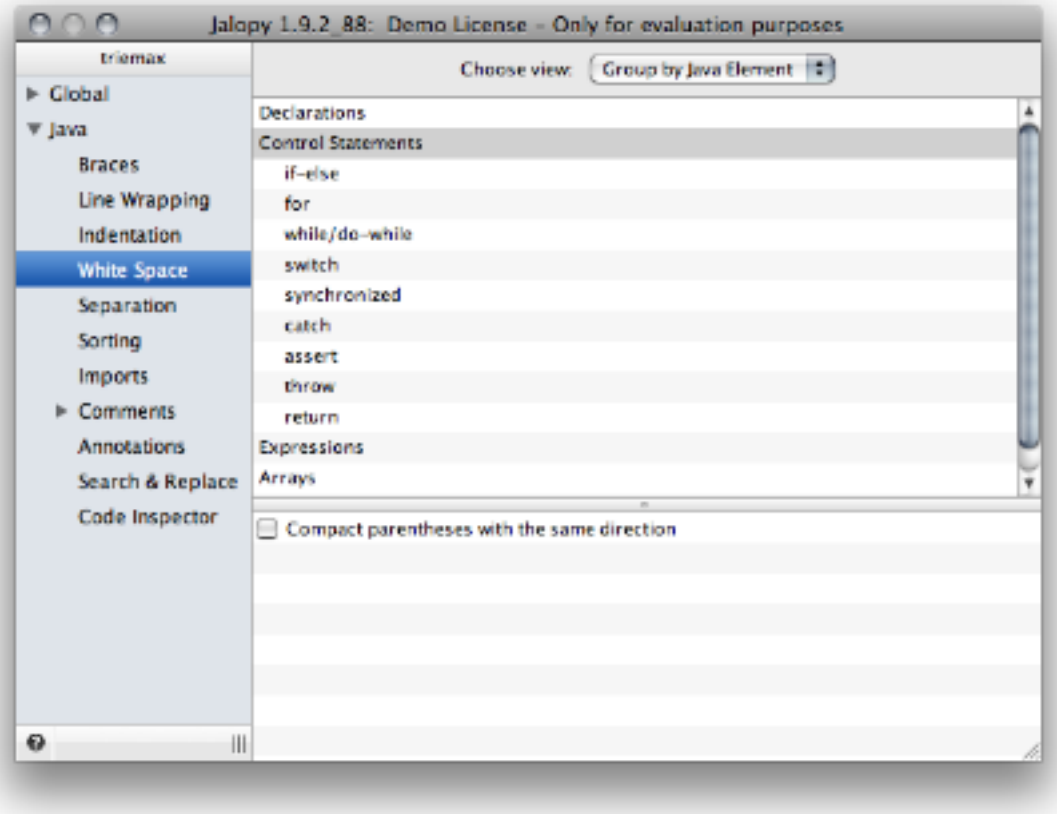
```
caller.<String,Element>foo();
```

Example 2.661. Type argument with space before right angle bracket

```
caller.<String,Element >foo();
```

2.8.9.2 Element view

Figure 2.45. White Space Element View



Declarations

Lets you configure the white space behavior for declarations.

Classes

Lets you configure the white space behavior for class declarations.

Before comma in implements clause

Please refer to the explanation for “Space before comma implements type”.

After comma in implements clause

Please refer to the explanation for “Space after comma implements type”.

Interfaces
Lets you configure the white space behavior for interface declarations.

Before comma in extends clause
Please refer to the explanation for “Space before comma extends type”.

After comma in extends clause
Please refer to the explanation for “Space after comma extends type”.

Enums
Lets you configure the white space behavior for enum declarations.

Before comma between constants
Please refer to the explanation for “Space before comma enum constant”.

After comma between constants
Please refer to the explanation for Section 2.8.9.1.4, “Enum constant”.

Before left parenthesis in constant argument list
Please refer to the explanation for “Space before left parenthesis enum constant argument”.

After left parenthesis in constant argument list
Please refer to the explanation for “Space after left parenthesis enum constant argument”.

Before comma in constant argument list
Please refer to the explanation for “Space before comma enum constant argument”.

After comma in constant argument list
Please refer to the explanation for Section 2.8.9.1.4, “Enum constant argument”.

Before right parenthesis in constant argument list
Please refer to the explanation for “Space before right parenthesis enum constant argument”.

Between empty parentheses in constant argument list
Please refer to the explanation for “Space between empty parentheses enum constant argument”.

Annotations
Lets you configure the white space behavior for annotation type declarations.

Before left parenthesis of type members
Please refer to the explanation for “Space before left parenthesis annotation type member”.

Between empty parentheses of type members
Please refer to the explanation for “Space between empty parentheses annotation type member”.

Before left parenthesis of member list
Please refer to the explanation for “Space before left parenthesis annotation argument list”.

After left parenthesis of member list
Please refer to the explanation for “Space after left parenthesis annotation argument list”.

Before assignment operator
Please refer to the explanation for “Space before assignment operator in annotations”.

After assignment operator
Please refer to the explanation for “Space after assignment operator in annotations”.

Before comma in member list
Please refer to the explanation for “Space before comma annotation type member argument”.

After comma in member list
Please refer to the explanation for “Space after comma annotation type member argument”.

Before right parenthesis of member list
Please refer to the explanation for “Space before right parenthesis annotation argument list”.

After left curly brace of annotation array
Please refer to the explanation for “Space after left curly brace annnotation array”.

Before comma in annotation array
Please refer to the explanation for “Space before comma annotation array”.

After comma in annotation array
Please refer to the explanation for “Space after comma annotation array”.

Before right curly brace of annotation array
Please refer to the explanation for “Space before right curly brace annotation array”.

Fields
Lets you configure the white space behavior for field declarations.

Before comma in multi-field
Please refer to the explanation for Section 2.8.9.1.3, “Field”.

After comma in multi-field
Please refer to the explanation for Space after comma multi-field.

Constructors
Lets you configure the white space behavior for constructor declarations.

Before left parenthesis of parameter list

Please refer to the explanation for Section 2.8.9.1.15, “Constructor”.

After left parenthesis of parameter list

Please refer to the explanation for “Space after left parenthesis constructor declaration”.

Before comma in parameter list

Please refer to the explanation for “Space before comma constructor declaration parameter”.

After comma in parameter list

Please refer to the explanation for Section 2.8.9.1.4, “Constructor”.

Before right parenthesis of parameter list

Please refer to the explanation for “Space before right parenthesis constructor declaration parameter”.

Between empty parentheses of parameter list

Please refer to the explanation for “Space between empty parentheses constructor declaration”.

Before comma in throws clause

Please refer to the explanation for “Space before comma constructor throws type”.

After comma in throws clause

Please refer to the explanation for “Space after comma constructor declaration throws type”.

Methods

Lets you configure the white space behavior for method declarations.

Before left parenthesis of parameter list

Please refer to the explanation for “Space before left parenthesis method declaration”.

After left parenthesis of parameter list

Please refer to the explanation for “Space after left parenthesis method declaration”.

Before comma in parameter list

Please refer to the explanation for “Space before comma method declaration parameter”.

After comma in parameter list

Please refer to the explanation for “Space after comma method declaration parameter”.

Before ellipsis in parameter list

Please refer to the explanation for Section 2.8.9.1.11, “Vararg”.

After ellipsis in parameter list

Please refer to the explanation for Section 2.8.9.1.12, “Vararg”.

Before right parenthesis of parameter list

Please refer to the explanation for “Space before right parenthesis method declaration parameter”.

Between empty parentheses of parameter list

Please refer to the explanation for “Space between empty parentheses enum constant argument”.

Before comma in throws clause

Please refer to the explanation for “Space before comma method throws type”.

After comma in throws clause

Please refer to the explanation for “Space after comma method declaration throws type”.

Local variables

Lets you configure the white space behavior for local variable declarations.

Before comma in multi-variable

Please refer to the explanation for Space before comma multi-var.

After comma in multi-variable

Please refer to the explanation for Space after comma multi-variable.

Labels

Before colon

Please refer to the explanation for Section 2.8.9.1.5, “Label”.

After colon

Please refer to the explanation for “Space after label colon”.

Control Statements

Lets you configure the white space behavior for control statements.

if

Lets you configure the white space behavior for `if` statements.

Before left parenthesis of expression list

Please refer to the explanation for “Space before left parenthesis if”.

After left parenthesis of expression list

Please refer to the explanation for Section 2.8.9.1.16, “if”.

Before right parenthesis of expression list

Please refer to the explanation for “Space before right parenthesis if”.

for

Lets you configure the white space behavior for `for` statements.

Before left parenthesis of expression list

Please refer to the explanation for “Space before left parenthesis for”.

After left parenthesis of expression list.

Please refer to the explanation for “Space after left parenthesis for”.

Before comma in initialization

Please refer to the explanation for “Space before comma for initializer”.

After comma in initialization

Please refer to the explanation for “Space after comma for initializer”.

Before comma in increment

Please refer to the explanation for “Space before comma for incrementor”.

After comma in increment

Please refer to the explanation for “Space after comma for incrementor”.

Before semicolon

Please refer to the explanation for “Space before semi colon for”.

After semicolon

Please refer to the explanation for “Space after semi colon for”.

Before colon

Please refer to the explanation for “Space before enhanced for colon”.

After colon

Please refer to the explanation for “Space after enhanced for colon”.

Before right parenthesis of expression list

Please refer to the explanation for “Space before right parenthesis for”.

while/do-while

Lets you configure the white space behavior for `while` and `do/while` statements.

Before left parenthesis of expression list

Please refer to the explanation for “Space before left parenthesis while”.

After left parenthesis of expression list

Please refer to the explanation for “Space after left parenthesis while”.

Before right parenthesis of expression list

Please refer to the explanation for “Space before right parenthesis while”.

switch

Lets you configure the white space behavior for `switch` statements.

Before left parenthesis of expression list

Please refer to the explanation for Section 2.8.9.1.15, “switch”.

After left parenthesis of expression list

Please refer to the explanation for “Space after left parenthesis switch”.

Before right parenthesis of expression list

Please refer to the explanation for “Space before right parenthesis switch”.

Before colon

Please refer to the explanation for “Space before case colon”.

synchronized

Lets you configure the white space behavior for `synchronized` statements.

Before left parenthesis of expression list

Please refer to the explanation for “Space before left parenthesis synchronized”.

After left parenthesis of expression list

Please refer to the explanation for “Space after left parenthesis synchronized”.

Before right parenthesis of expression list

Please refer to the explanation for “Space before right parenthesis synchronized”.

catch

Lets you configure the white space behavior for `catch` statements.

Before left parenthesis of expression list

Please refer to the explanation for Section 2.8.9.1.15, “catch”.

After left parenthesis of expression list

Please refer to the explanation for Section 2.8.9.1.16, “catch”.

Before right parenthesis of expression list

Please refer to the explanation for Section 2.8.9.1.17, “catch”.

assert

Lets you configure the white space behavior for `assert` statements.

Before colon

Please refer to the explanation for “Space before assertion colon”.

After colon

Please refer to the explanation for “Space after assertion colon”.

throw

Lets you configure the white space behavior for `throw` statements.

Before left parenthesis of expression

Please refer to the explanation for “Space before left parenthesis throw”.

After left parenthesis of expression

Please refer to the explanation for “Space after left parenthesis throw”.

Before right parenthesis of expression

Please refer to the explanation for “Space before right parenthesis throw”.

return

Lets you configure the white space behavior for `return` statements.

Before left parenthesis of expression

Please refer to the explanation for “Space before left parenthesis return”.

After left parenthesis of expression

Please refer to the explanation for “Space after left parenthesis return”.

Before right parenthesis of expression

Please refer to the explanation for Section 2.8.9.1.17, “return”.

Expressions

Lets you configure the white space behavior for expressions.

Constructor call

Lets you configure the white space behavior for constructor calls.

Before left parenthesis of argument list

Please refer to the explanation for “Space before left parenthesis constructor call”.

After left parenthesis of argument list

Please refer to the explanation for “Space after left parenthesis constructor call”.

Before comma in argument list

Please refer to the explanation for “Space before comma constructor call argument”.

After comma in argument list

Please refer to the explanation for “Space after comma constructor call argument”.

Before right parenthesis of argument list

Please refer to the explanation for “Space before right parenthesis constructor call”.

Between empty parentheses of argument list

Please refer to the explanation for “Space between empty parentheses constructor call”.

Creator call

Before left parenthesis of argument list
Please refer to the explanation for “Space before left parenthesis creator call”.

After left parenthesis of argument list
Please refer to the explanation for “Space after left parenthesis creator call”.

Before comma in argument list
Please refer to the explanation for “Space before comma creator call argument”.

After comma in argument list
Please refer to the explanation for “Space after comma creator call argument”.

Before right parenthesis of argument list
Please refer to the explanation for Section 2.8.9.1.17, “Creator”.

Between empty parentheses of argument list
Please refer to the explanation for “Space between empty parentheses creator call”.

Method call

Before left parenthesis of argument list
Please refer to the explanation for “Space before left parenthesis method call”.

After left parenthesis of argument list
Please refer to the explanation for “Space after left parenthesis method call”.

Before comma in argument list
Please refer to the explanation for “Space before comma method call argument”.

After comma in argument list
Please refer to the explanation for “Space after comma method call argument”.

Before right parenthesis of argument list
Please refer to the explanation for “Space before right parenthesis method call”.

Between empty parentheses of argument list
Please refer to the explanation for “Space between empty parentheses method call”.

Operators

Before assignment operator
Please refer to the explanation for “Space before assignment operator”.

After assignment operator
Please refer to the explanation for Section 2.8.9.1.2, “Assignment operator”.

Before assignment operator in annotations

Please refer to the explanation for “Space before assignment operator in annotations”.

After assignment operator in annotations

Please refer to the explanation for “Space after assignment operator in annotations”.

Before bitwise operator

Please refer to the explanation for “Space before bitwise operator”.

After bitwise operator

Please refer to the explanation for “Space after bitwise operator”.

Before logical operator

Please refer to the explanation for “Space before logical operator”.

After logical operator

Please refer to the explanation for “Space after logical operator”.

Before mathematical operator

Please refer to the explanation for “Space before mathematical operator”.

After mathematical operator

Please refer to the explanation for “Space after mathematical operator”.

Before string concat operator

Please refer to the explanation for “Space before concat operator”.

After string concat operator

Please refer to the explanation for “Space after concat operator”.

Before relational operator

Please refer to the explanation for Section 2.8.9.1.1, “Relational operator”.

After relational operator

Please refer to the explanation for “Space after relational operator”.

Before shift operator

Please refer to the explanation for “Space before shift operator”.

After shift operator

Please refer to the explanation for “Space after shift operator”.

Before conditional question operator

Please refer to the explanation for “Space before question mark conditional operator”.

After conditional question operator

Please refer to the explanation for “Space after question mark conditional operator”.

Before conditional colon operator
Please refer to the explanation for Section 2.8.9.1.5, “Conditional”.

After conditional colon operator
Please refer to the explanation for “Space after conditional operator colon”.

Parenthesized expression
Lets you control the white space behavior for parenthesized expressions.

After left parenthesis
Please refer to the explanation for Section 2.8.9.1.16, “Parenthesized expression”.

Before right parenthesis
Please refer to the explanation for Section 2.8.9.1.17, “Parenthesized expression”.

Type cast
Lets you control the white space behavior for type casts.

After left parenthesis
Please refer to the explanation for “Space after left parenthesis type cast”.

Before right parenthesis
Please refer to the explanation for “Space before right parenthesis type cast”.

After right parenthesis
Please refer to the explanation for “Space after right parenthesis type cast”.

Arrays
Lets you control the white space behavior for arrays.

Declaration
Lets you control the white space behavior for array declarations.

Before left bracket
Please refer to the explanation for “Space before left bracket array declaration”.

Between empty brackets
Please refer to the explanation for “Space between empty brackets array declaration”.

Allocation
Lets you control the white space behavior for array allocations.

Before left bracket
Please refer to the explanation for “Space before left bracket creator”.

After left bracket
Please refer to the explanation for Section 2.8.9.1.26, “Array creator”.

Before right bracket
Please refer to the explanation for “Space before right bracket array creator”.

Between empty brackets
Please refer to the explanation for “Space between empty brackets array creator”.

Initializer
Lets you control the white space behavior for array initializers.

Before left brace
Please refer to the explanation for “Space before left curly brace array initializer”.

After left brace
Please refer to the explanation for “Space after left curly brace array initializer”.

Before comma
Please refer to the explanation for “Space before comma array initializer”.

After comma
Please refer to the explanation for “Space after comma array initializer”.

Before right brace
Please refer to the explanation for “Space before right curly brace array initializer”.

Between empty braces
Please refer to the explanation for “Space between empty curly braces array initializer”.

Accessor
Lets you control the white space behavior for array accesssors.

Before left bracket
Please refer to the explanation for Section 2.8.9.1.25, “Array accessor”.

After left bracket
Please refer to the explanation for Section 2.8.9.1.26, “Array accessor”.

Before right bracket
Please refer to the explanation for “Space before right bracket array accessor”.

Parameterized types
Lets you control the white space behavior for parameterized (generic) types.

Type parameter
Lets you configure the white space behavior for type parameters.

Before left angle bracket
Please refer to the explanation for “Space before left angle bracket type parameter”.

After left angle bracket
Please refer to the explanation for “Space after left bracket type parameter”.

Before comma in brackets
Please refer to the explanation for “Space before comma type parameter”.

After comma in brackets
Please refer to the explanation for “Space after comma type parameter”.

Before ampersand in brackets
Please refer to the explanation for “Space before ampersand type parameter”.

After ampersand in brackets
Please refer to the explanation for “Space after ampersand type parameter”.

Before question mark in brackets
Please refer to the explanation for “Space before question mark type parameter”.

After question mark in brackets
Please refer to the explanation for “Space after question mark type parameter”.

Before right angle bracket
Please refer to the explanation for “Space before right angle bracket type parameter”.

Type argument
Lets you configure the white space behavior for type arguments.

Before left angle bracket
Please refer to the explanation for “Space before left angle bracket type argument”.

After left angle bracket
Please refer to the explanation for “Space before left angle bracket type argument”.

Before comma in brackets
Please refer to the explanation for “Space before comma type argument”.

After comma in brackets
Please refer to the explanation for “Space after comma type argument”.

Before question mark in brackets
Please refer to the explanation for “Space before question mark type argument”.

After question mark in brackets
Please refer to the explanation for “Space after question mark type argument”.

Before right angle bracket
Please refer to the explanation for “Space before right angle bracket type argument”.

2.8.10 Separation

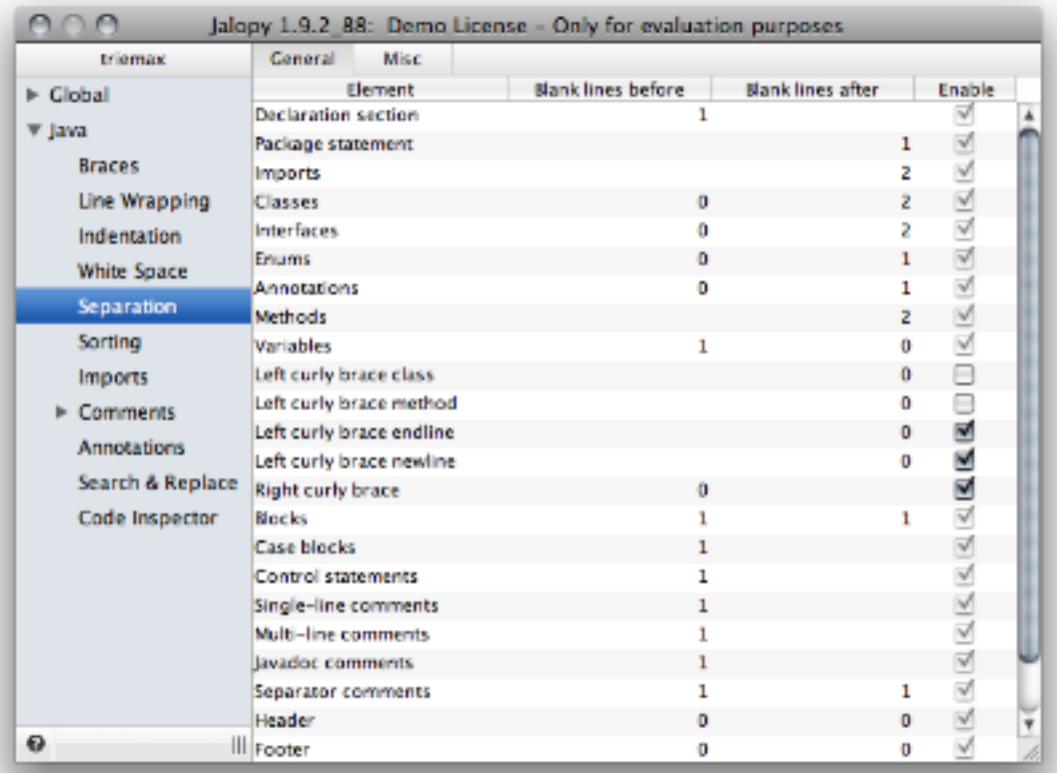
Lets you control the insertion of blank lines to separate statements or declarations with different functions or meanings. Just as related statements should be grouped together, unrelated statements should be separated from each other. In English, the start of a new paragraph is identified with indentation or a blank line. This greatly aids reading ease and helps to improve comprehension and reading speed. When coding, you should strive for a similar goal and always give the reader hints as to how the program is organized.

With Jalopy you are able to enforce consistent blank lines behavior and divide groups of related statements into paragraphs, separate routines from one another and highlight comments. You can give Jalopy complete control over the blank lines handling, or use a more relaxed style and keep existing blank lines up to a given number.

2.8.10.1 General

Lets you specify the number of blank lines that should appear before, after and between different Java source elements.

Figure 2.46. Blank Lines settings page



Declaration section

Lets you control how many blank lines should be printed before a new declaration section. A declaration section means an arbitary amount of similar declarations, like e.g. instance initializers, or methods or enum declarations. This option is only meaningful when code sorting is enabled for declarations. Refer to Section 2.8.11.1, “Declarations” for more information about the code sorting feature.

Since 1.4

Example 2.662. 2 blank lines before code sections

```
private Foo aFoo;
¶
¶
Constructor(Foo rFoo) {
}
¶
¶
public static void method(Foo rFoo) {
}
```

Package statement

Lets you control how many blank lines should be printed after the package statement.

Example 2.663. 3 blank lines after package statement

```
package com.triemax.jalopy.printer;
¶
¶
¶
import antlr.collections.AST;

import com.triemax.jalopy.parser.JavaAST;
import com.triemax.jalopy.parser.JavaTokenTypes;

...
```

Imports

Lets you control how many blank lines should be printed after the last import statement.

Example 2.664. 4 blank lines after last import statement

```
package com.triemax.jalopy.printer;

import antlr.collections.AST;

import com.triemax.jalopy.parser.JavaAST;
import com.triemax.jalopy.parser.JavaTokenTypes;
¶
¶
¶
¶
public class Printer {
}
```

Classes

Lets you control how many blank lines should be printed before the first top level class declaration of a compilation unit and between two class declarations.

Example 2.665. 1 blank line before class declaration

```
¶
class Foo {
}
```

The blank lines before setting is only meaningful if the class declaration is the first top level declaration of a compilation unit and not preceded by a package or import statement.

Example 2.666. 2 blank lines between two class declarations

```
class One {
}
¶
¶
class Two {
}
```

Interfaces

Lets you control how many blank lines should be printed before the first top level interface declaration of a compilation unit and between two interface declarations.

Example 2.667. 2 blank lines before first interface declarations

```
¶
¶
interface Fooable {
}
```

The blank lines before setting is only meaningful if the interface declaration is the first top level declaration of a compilation unit and not preceded by a package or import statement.

Example 2.668. 3 blank lines between two interface declarations

```
interface One {
}
¶
¶
¶
interface Two {
}
```

Enums

Lets you control how many blank lines should be printed before the first top level enum declaration of a compilation unit and between two enum declarations. The blank lines before setting is only meaningful if the enum declaration is the first top level declaration of a compilation unit and not preceded by a package or import statement.

Since 1.1

Example 2.669. 3 blank lines between two enum declarations

```
public enum Season {
    WINTER, SPRING, SUMMER, FALL
}
¶
¶
¶
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Annotations

Lets you control how many blank lines should be printed before the first top level annotation declaration of a compilation unit and between two annotation declarations. The blank lines before setting is only meaningful if the annotation declaration is the first top level declaration of a compilation unit and not preceded by a package or import statement.

Since 1.1

Example 2.670. 2 blank lines between two enum declarations

```
public @interface Name {
    String first();
    String last();
}
¶
¶
public @interface Endorsers {
    String[] value();
}
```

Methods

Lets you control how many blank lines should be printed between two method/constructor declarations.

Example 2.671. 3 blank lines between two method declarations

```
public static Printer getInstance() {
    return INSTANCE;
}
¶
¶
¶
public void print(AST node, ASTWriter out)
           throws IOException {
}
```

Variables

Lets you control how many blank lines should be printed before and after variable declarations.

Example 2.672. 1 blank line before variable declarations

```
System.out.println();
¶
int a = 1;
int b = 2;
```

Example 2.673. 2 blank lines after variable declarations

```
int a = 1;
int b = 2;
¶
¶
System.out.println();
```

Left curly brace class

Forces the given number of blank lines after left curly braces of class, interface and enum declarations. You need to explicitly enable the option to take effect. When enabled, this option takes highest precedence and overrides all other blank line settings.

Please note that you need to set this option only if you want different left curly blank lines behavior for class level blocks. Otherwise, the corresponding options for left curly brace endline/newline would be enough to enforce consistent behavior.

Since 1.9.1

Example 2.674. Blank lines before Javadoc=1

```
class Demo {
¶
    /** Demo method */
    public void foo() {
    }
}
```

Example 2.675. Blank lines before Javadoc=1, Blank lines after left curly braces=0

```
class Demo {
    /** Demo method */
    public void foo() {
    }
}
```

Left curly brace method

Forces the given number of blank lines after left curly braces of method and constructor declarations. You need to explicitly enable the option to take effect. When enabled, this option takes highest precedence and overrides all other blank line settings.

Please note that you need to set this option only if you want different left curly blank lines behavior for method level blocks. Otherwise, the corresponding options for left curly brace endline/newline would be enough to force consistent behavior.

Since 1.9.1

Example 2.676. Keep blank lines=1

```
private final void loadVersion() {
¶
    check();
}
```

Example 2.677. Keep blank lines=1, Blank lines after left curly braces methods=0

```
private final void loadVersion() {
    check();
}
```

Left curly brace endline

Forces the given number of blank lines after left curly braces that are printed at the end of the line beginning the compound statement (a.k.a. Sun brace style). You need to explicitly enable the option to take effect. When enabled, this option overrides all other blank line settings other than Left curly brace class and Left curly brace method.

Example 2.678. Blank lines before blocks=1

```
public void foo() {
¶
    if (condition()) {
¶
        if (anotherCondition()) {
            doSomething();
        }
    }
}
```

Example 2.679. Blank lines before blocks=1, Blank lines after left curly braces=0

```
public void foo() {
    if (condition()) {
        if (anotherCondition()) {
            doSomething();
        }
    }
}
```

Left curly brace endline newline

Forces the given number of blank lines after left curly braces that are printed at the beginning of lines (a.k.a. C brace style). When enabled, this option overrides all other blank line settings.

Since 1.7

Example 2.680. Blank lines before blocks=1

```
public void foo()
{
¶
    if (condition())
    {
¶
        if (anotherCondition())
        {
            doSomething();
        }
    }
}
```

Example 2.681. Blank lines before blocks=1, Blank lines after left curly braces=0

```
public void foo()
{
    if (condition())
    {
        if (anotherCondition())
        {
            doSomething();
        }
    }
}
```

Right curly brace

Forces the given number of blank lines before closing curly braces, no matter what other blank lines settings dictate.

Example 2.682. Blank lines before blocks=1

```
public void foo() {
    if (condititon()) {
        if (anotherCondition()) {
            doSomething();
¶
        }
¶
    }
¶
}
```

Blocks

Lets you control how many blank lines should be printed before and after statement blocks (if-else , for, while, do-while, switch, try-catch-finally, synchronized). Note that the "Blank Lines After" setting also applies for anonymous inner classes.

Example 2.683. 2 blank lines before and after blocks

```
AST type = null;
¶
¶
switch (next.getType()) {
    case JavaTokenTypes.LPAREN :
        type = PrinterUtils.advanceToFirstNonParen(next);
        break;
    default :
        type = next;
        break;
}
¶
¶
AST ident = type.getFirstChild();
```

Case blocks

Lets you control how many blank lines should be printed before each case block of a switch expression.

Example 2.684. 3 blank lines before case blocks

```
switch (next.getType()) {
¶
¶
¶
    case JavaTokenTypes.LPAREN :
        type = PrinterUtils.advanceToFirstNonParen(next);
        break;
¶
¶
¶
    default :
        type = next;
        break;
}
```

Control statements

Lets you control how many blank lines should be printed before the statements `return`, `break` and `continue`.

Example 2.685. 2 blank lines before case control statements

```
switch (next.getType()) {
    case JavaTokenTypes.LPAREN :
        type = PrinterUtils.advanceToFirstNonParen(next);
¶
¶
        break;

    default :
        type = next;
¶
¶
        break;
}
```

Note that this setting does not apply when a control statement appears directly after the `case` or `default` keyword or when the statement is the single member of a statement block without curly braces.

Example 2.686. Setting takes no effect before case control statements

```
switch (next.getType()) {
    case JavaTokenTypes.LPAREN :
        break;

    default :
        continue;
}
```

Example 2.687. Setting takes no effect for single statements in blocks

```
if (isClean())
    return;
```

Single-line comments

Lets you control how many blank lines should be printed before single-line comments.

Example 2.688. 1 blank line before single-line comment

```
System.out.println("ERROR");
¶
// XXX use log4j logger
ex.printStackTrace();
```

Multi-line comments

Lets you control how many blank lines should be printed before multi-line comments.

Example 2.689. 2 blank lines before multi-line comment

```
System.out.println("ERROR");
¶
¶
/* XXX use log4j logger */
ex.printStackTrace();
```

Javadoc comments

Lets you control how many blank lines should be printed before Javadoc comments.

Separator comments

Lets you control how many blank lines should be printed before and after separator comments.

Since 1.7

Example 2.690. 2 blank lines before/after separator comment

```
protected Foo instance;
¶
¶
//~ Constructors -----------------------------------------------------------
¶
¶
public Demo ( ) {}
public Demo (  Foo anotherFoo ) {}
```

Header

Lets you control how many blank lines should be printed before and after headers.

Example 2.691. No blank lines after header

```
// Copyright 1998-2000, Foo, Inc. All Rights Reserved.
// Confidential and Proprietary Information of Foo, Inc.
// Protected by or for use under one or more of the following patents:
// U.S. Patent Nos. X,XXX,XXX. Other Patents Pending.
package com.foobar;

...
```

Example 2.692. 2 blank lines after header

```
// Copyright 1998-2000, Foo, Inc. All Rights Reserved.
// Confidential and Proprietary Information of Foo, Inc.
// Protected by or for use under one or more of the following patents:
// U.S. Patent Nos. X,XXX,XXX. Other Patents Pending.
¶
¶
package com.foobar;

...
```

Footer

Lets you control how many blank lines should be printed before and after footers.

SQLJ clauses

Lets you control how many blank lines should be printed before and after SQLJ clauses.

Example 2.693. 2 blank lines before/after SQLJ clause

```
Integer salesRepID   = new Integer(358);
String  salesRepName = "Jouni Seppanen";
Date    dateSold     = new Date(97,11,6);
¶
¶
#sql { INSERT INTO SALES VALUES( :itemID,:itemName,:dateSold,:totalCost,
      :salesRepID,:salesRepName) };
¶
¶
SalesRecs sales;
```

Assignment section

Lets you control how many blank lines are printed after a section with several (at least two) consecutive assignment expressions.

Since 1.4

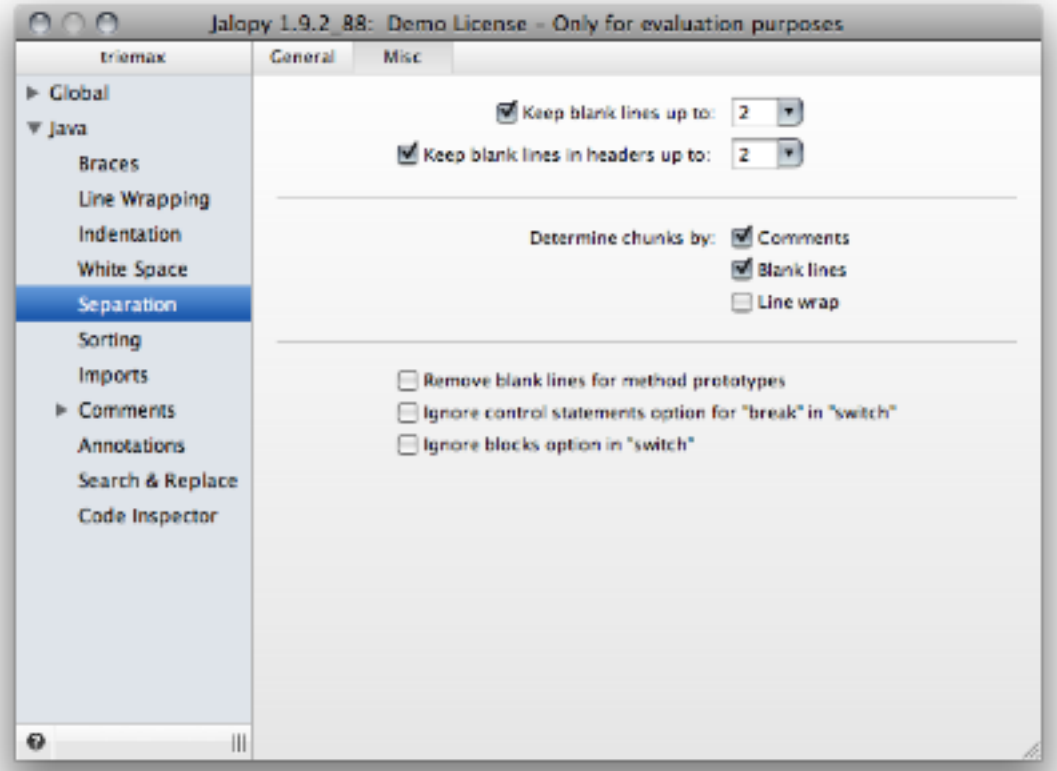
Example 2.694. 1 blank line after assignment section

```
c.fill = GridBagConstraints.HORIZONTAL;
c.insets = new Insets(0, 0, 0, 0);
¶
middlePanel.add(buttonsPanel, c);
```

2.8.10.2 Misc

Lets you control miscellaneous separation settings.

Figure 2.47. Blank Lines Misc settings page



Keep blank lines up to

When enabled, retains up to the given number of blank lines found in the original source. Note that Jalopy still takes your other blank lines settings into account. If you disable this option, all original blank lines will be ignored!

Example 2.695. Source code with blank lines to separate declaration sections

```
aMVString = new MultiValueString("abc");
¶
System.out.println("MV = "+aMVString);
¶
System.out.println("MV0 = "+aMVString.extract(0));
System.out.println("MV1 = "+aMVString.extract(1));
System.out.println("MV2 = "+aMVString.extract(2));
System.out.println("");
```

If this feature is left disabled, Jalopy will print the individual lines according to the current blank lines settings but won't try to keep any blank lines.

Keep blank lines in headers up to

When enabled, retains up to the given number of blank lines found in the original source file between header (and footer) comments. This option is only signficant when you enable the header/footer feature and specify a multi-header template, without enabling the override mode.

Since 1.7

Example 2.696. Source code with blank lines to separate headers

```
/*
 * Foo.java
 *
 * $Header: //depot/foo/src/main/com/foo/Foo.java#13 $
 */
¶
/*
 * Copyright (c) 2002 FooBar, Inc. All rights reserved.
 *
 * This software is the confidential and proprietary information
 * of FooBar, Inc. ("Confidential Information"). You shall not
 * disclose such Confidential Information and shall use it only
 * in accordance with the terms of the license agreement you
 * entered into with FooBar, Inc.
 */
package com.foobar.lemon;
```

Chunks

Lets you define what makes a chunk: a section of associated statements. With "Variable identifiers" and/or "Align assignments" enabled, Jalopy determines what statements can be aligned using these options. With “Align endline comments” enabled, Jalopy determines what comments should be aligned together.

By comments

When enabled, a statement with a comment before is recognized as the start of a new chunk.

Example 2.697. Aligned assignments/identifiers

```
String        text  = "text";
int           a     = -1;
// create a new entry
History.Entry entry = new History.Entry(text);
```

Example 2.698. Aligned assignments/identifiers, chunks by comments

```
String text  = "text";
int    a     = -1;
// create a new entry
History.Entry entry = new History.Entry(text);
```

By blank lines

When enabled, a statement which has one or more blank lines before is recognized as the start of a new chunk.

Example 2.699. Aligned assignments/identifiers

```
String        text  = "text";
int           a     = -1;
¶
History.Entry entry = new History.Entry(text);
```

Example 2.700. Aligned assignments/identifiers, chunks by blank lines

```
String text  = "text";
int    a     = -1;
¶
History.Entry entry = new History.Entry(text);
```

By line wrap

When enabled, a statement that takes more than just one line to print is recognized as the start of a new chunk. With standard indentation, it is recommended to have this option enabled, because otherwise wrapped expressions might obscure aligned assignments. Please note that this option does not affect endline comments!

Since 1.3

Example 2.701. Aligned assignments

```
int labelR_x      = Math_min(iconR.x, textR.x);
int labelR_with   = Math_max(iconR.x + iconR.width, textRrr.x) -
    labelR_x;
int labelR_width  = Math_max(iconR.x + iconR.width, textRrr.x) -
    labelR_x;
int labelR_y      = Math_min(iconR.y, textR.y);
int labelR_height = Math_max(iconR.y + iconR.height,
        textR.y + textR.height);
int labelR_x      = Math_min(iconR.x, textR.x);
int lab           = Math_min(iconR.x, textR.x);
```

Example 2.702. Aligned assignments, chunks by line wrap, prefer wrap after assign

```
int labelR_x = Math_min(iconR.x, textR.x);
int labelR_with =
    Math_max(iconR.x + iconR.width, textRrr.x) - labelR_x;
int labelR_width =
    Math_max(iconR.x + iconR.width, textRrr.x) - labelR_x;
int labelR_y = Math_min(iconR.y, textR.y);
int labelR_height =
    Math_max(iconR.y + iconR.height, textR.y + textR.height);
int labelR_x = Math_min(iconR.x, textR.x);
int lab      = Math_min(iconR.x, textR.x);
```

Remove blank lines for method prototypes

When enabled, blank lines around abstract method declarations are removed. This includes all methods in interfaces (which are implicitly abstract). And all methods explicitely declared abstract in classes. If left disabled, blank lines will be printed according to the blank lines settings for methods (see “Blank lines after methods”).

Since 1.2

Example 2.703. Method prototypes

```
interface foo {

    public void method1();

    public void method2();

    public void method3();
}
```

Example 2.704. Method prototypes without blank lines

```
interface foo {
    public void method1();
    public void method2();
    public void method3();
}
```

Ignore control statements option for break in switch

When enabled, the "Blank Lines before control statements" option is ignored for `break` statements within `switch` blocks.

Since 1.3

Example 2.705. 1 blank lines before control statements

```
switch (number) {
    case 1:
        System.out.println();

        break;

    default:
        System.out.println();

        break;
}
```

Example 2.706. 1 blank lines before control statements, but option ignored

```
switch (number) {
    case 1:
        System.out.println();
        break;

    default:
        System.out.println();
        break;
}
```

Ignore blocks option in switch

When enabled, the "Blank lines for blocks" option is ignored within `switch` blocks.

Since 1.3

Example 2.707. 1 blank lines before blocks

```
switch (number) {
    case 1:

        if (DEBUG)
            System.out.println("FIRST NUMBER");

        break;

    default:

        while (true)
            perform();

        break;
}
```

Example 2.708. 1 blank lines before blocks, but option ignored

```
switch (number) {
    case 1:
        if (DEBUG)
            System.out.println("FIRST NUMBER");

        break;

    default:
        while (true)
            perform();

        break;
}
```

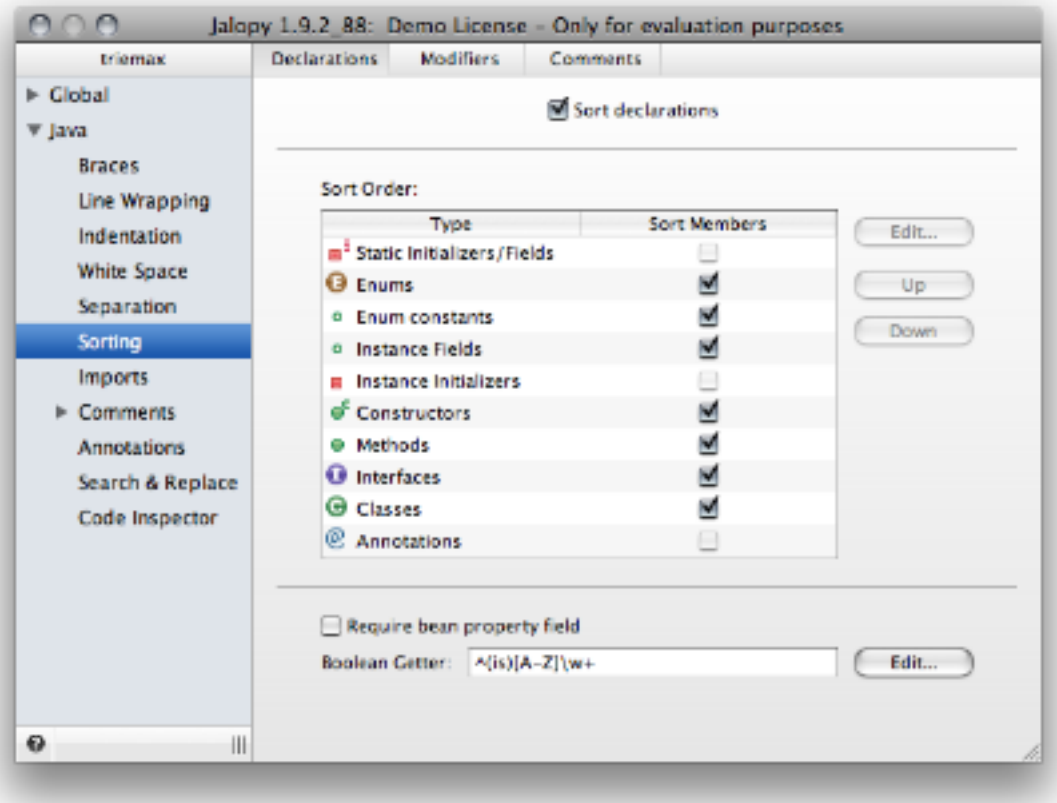
2.8.11 Sorting

Lets you control the code sorting. Code sorting lets you arrange elements in a specific order to ease navigation and improve comprehension.

2.8.11.1 Declarations

Lets you control the order of the main declaration elements of Java compilation units: classes, interfaces, enums, annotations, fields, initializers, constructors and methods.

Figure 2.48. Sorting Declarations settings page



Sort declarations

Enables or disables the sorting of declarations. When disabled, all declarations appear in their original order. Otherwise they are grouped and sorted according to the defined order.

Sort Order

You can specify the order in which static fields/initializers, instance fields, instance initializers, constructor, method, enum, annotation, class and interface declarations and enum constants should appear in source files by selecting an element type and moving it up or down the list with the *Up* and *Down* buttons. By default, with sorting enabled, the different declaration elements are grouped together, but within one element type the contained declarations are still placed in their original order. For example all methods will be grouped together, but otherwise the methods appear in their original order. Enable any of the check boxes, if you want to have all members of one type sorted, too.

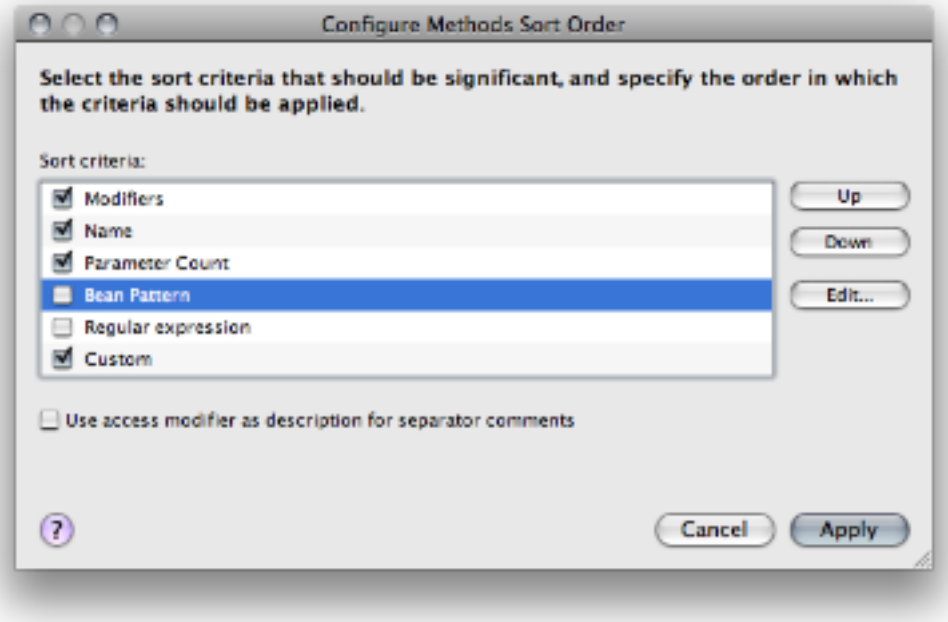
Use the *Edit...* button to define the order in which the individual members should appear. A dialog opens that lets you specify the sorting criteria, e.g. modifier or name. You can select what criteria should be used and in what order. Jalopy checks all significant sorting conditions in the specified order until the sorting position of a declaration member could be determined.

For example, if you want your methods to be sorted according to their access modifier and names, enable both criteria and disable the others. For two methods Jalopy will first check whether the modifiers are equal. If they differ, the sorting order is already obvious and no further criteria applied. But if they are equal, it will check their names and sort the two methods lexicographically.

For modifiers, you can further refine what modifiers should be significant for sorting and in what order they are tested. Though accessible from different dialogs, the settings for modifier sorting are global and used across all declaration types to achieve a consistent style.

Methods

Figure 2.49. Configure Sorting Order of Methods



For methods, you can specify sorting by modifiers, name, parameter count, Java Bean pattern, regular expression pattern or custom ordering. Enable the check box for each criteria that should be significant and use the *Up* and *Down* buttons to specify the order in which the criteria should be applied. To further refine the way sorting is applied for modifiers,

select the “Modifiers” entry and press the *Edit...* button to define what modifiers should be significant and adjust their order.

Example 2.709. Method

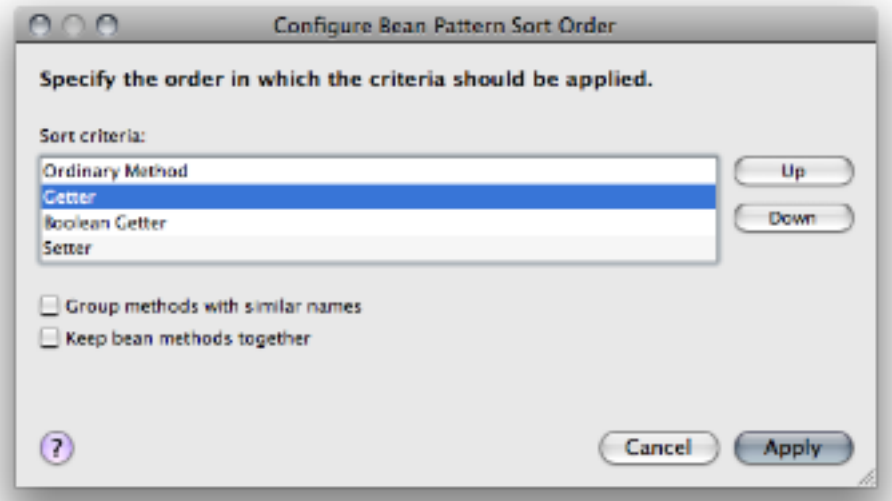
```
public final void setName(String first, String second) {
    ...
}
```

In the above example, the access modifier would be `“public final”`, the name `“setName”` and parameter count `'2';`.

Bean Pattern

Bean pattern refers to the JavaBeans specification naming convention that requires accessor and mutator methods to begin with either the *set*, *get* or *is* prefixes.

Figure 2.50. Configure Sorting Order of JavaBeans Methods



You can control the order in which JavaBeans methods and ordinary methods appear by selecting the entry and press the *Edit...* button. Adjust the order of the different elements in the list component of the new dialog and press *Apply*. The *Ordinary method* entry refers to all non-Bean methods.

Group methods with similar names

Normally bean pattern sorting means that all methods of one type (getters, boolean getters, setters, ordinary methods) are grouped together according to the specified order. Enabling this option causes all methods with similar names to be grouped together, i.e. bean methods and ordinary methods may be mixed, but all bean methods for one property stay together.

Similar methods are determined by stripping the bean prefix and comparing the resulting method names. Similar methods are grouped together according to the specified order (getters, boolean getters, setters, ordinary methods).

Since 1.5

Example 2.710. Bean pattern sorting

```
public void aaaaaa() {}

public void bbbbbb() {}

public void cccccc() {}

public Object getAaaaaa() {}

public Object getBbbbbb() {}

public Object getCccccc() {}

public boolean isAaaaaa() {}

public boolean isBbbbbb() {}

public boolean isCccccc() {}

public void setAaaaaa() {}

public void setBbbbbb() {}

public void setCccccc() {}
```

Example 2.711. Bean pattern sorting, group similar

```
public void aaaaaa() {}

public Object getAaaaaa() {}

public boolean isAaaaaa() {}

public void setAaaaaa() {}

public void bbbbbb() {}

public Object getBbbbbb() {}

public boolean isBbbbbb() {}

public void setBbbbbb() {}

public void cccccc() {}

public Object getCccccc() {}

public boolean isCccccc() {}

public void setCccccc() {}
```

Keep bean methods together

Grouping similar methods will let you group all bean methods for one property together, but the bean methods are otherwise still mixed with ordinary methods. If you instead prefer to have *all* bean methods grouped together, you can enable this option to build one large block with all bean methods.

Since 1.8

Example 2.712. Bean pattern sorting, group similar

```
public void aaaaaa() {}

public Object getAaaaaa() {}

public boolean isAaaaaa() {}

public void setAaaaaa() {}

public void bbbbbb() {}

public Object getBbbbbb() {}

public boolean isBbbbbb() {}

public void setBbbbbb() {}

public void cccccc() {}

public Object getCccccc() {}

public boolean isCccccc() {}

public void setCccccc() {}
```

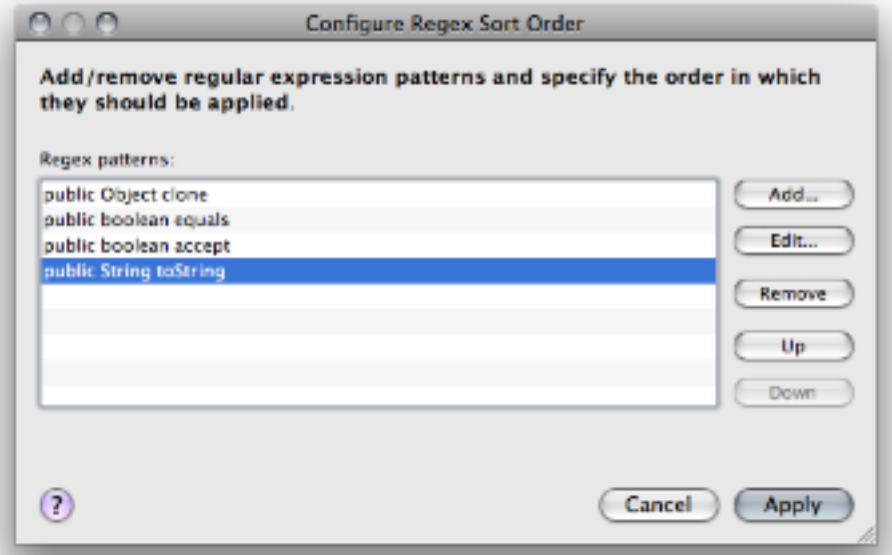
Example 2.713. Bean pattern sorting, group similar, bean methods kept together

```
public Object getAaaaaa() {}

public boolean isAaaaaa() {}

public void setAaaaaa() {}

public Object getBbbbbb() {}

public boolean isBbbbbb() {}

public void setBbbbbb() {}

public Object getCccccc() {}

public boolean isCccccc() {}

public void setCccccc() {}

public void aaaaaa() {}

public void bbbbbb() {}

public void cccccc() {}
```

Regular expression

Lets you define arbitrary regular expressions to match method signatures to specify absolute positions for specific methods.

Figure 2.51. Configure Regular Expression Sorting



Matching is performed against a simplified signature: only modifiers, return type and method name are used.

Example 2.714. Method declaration

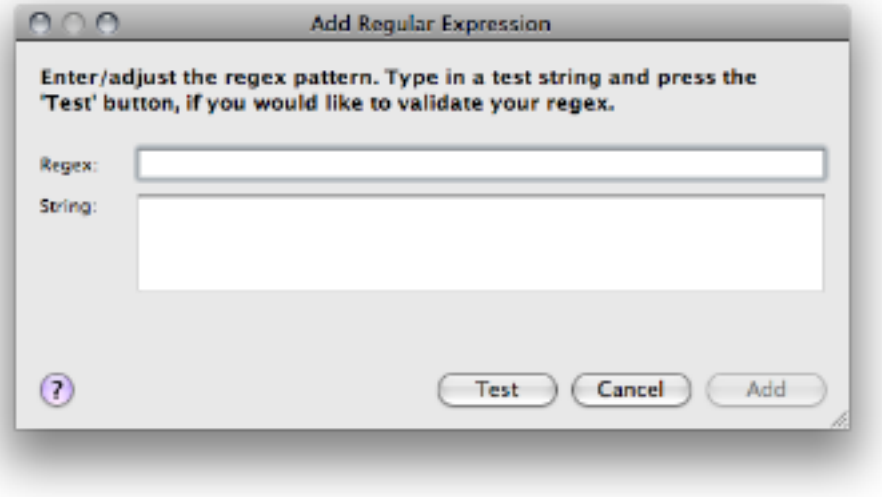
```
public boolean equals(Object rOther) {
    ...
}
```

The above method declaration would yield the following signature: *public boolean equals*

Add...

To add a new regular expression, press the *Add...* button.

Figure 2.52. Add Regular Expression



Enter the regular expression into the *Regex* text field and press the *Apply...* button to apply the addition. If you want to test the regular expression before you submit it, enter a test string in the *String* field and press the *Test* button to perform pattern matching.

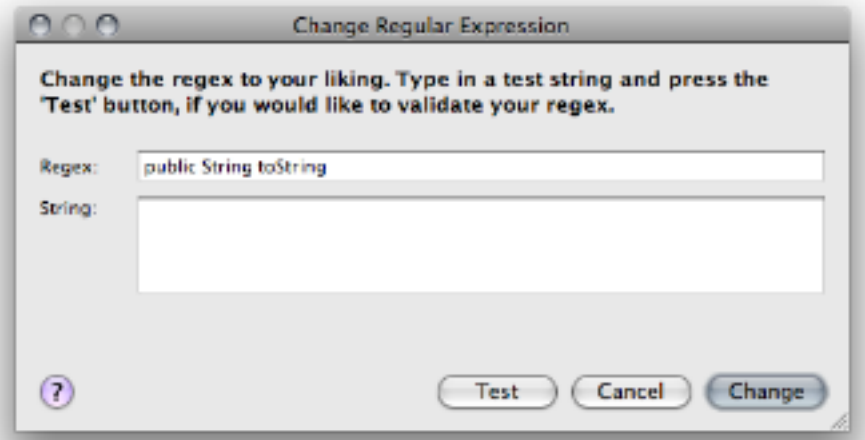
Remove

To remove an existing regular expression, select the expression you want to remove and press the *Remove* button.

Change...

To change an existing regular expression, select the expression you want to change and press the *Change...* button.

Figure 2.53. Change Regular Expression



Adjust the regular expression in the *Regex* text field and press the *Apply...* button to apply the change. If you want to test the regular expression before you submit it, enter a test string in the *String* field and press the *Test* button to perform pattern matching.

Custom Sort Order

As it might not always be sufficient to rely on method signature information alone, developers can take total control over method ordering using special Javadoc tags.

IMPORTANT Javadoc formatting, see Section 2.8.14.1.1, "Format comments", must be enabled for this feature to work.

In order to have methods grouped by purpose, check the *Custom* entry in the upper list of the dialog, move it to the top and utilize two custom Javadoc tags in your method commentary. In your method declaration comments, you need to add the Javadoc standalone tag `@jalopy.group` followed by a logical group name. This name can be freely chosen and defines the group a method belongs to.

Example 2.715. Sorting method declarations with @jalopy.group

```
/**
 * Returns the value of the Foo property.
 *
 * @jalopy.group  Accessors
 */
public int getFoo () {
    ...
}
```

Then you specify the order of methods with the `@jalopy.group-order` (you can use `@jalopy.group_order` to circumvent a bug in the Sun 1.4.2 Javadoc implementation)

in the class or interface Javadoc comment. Simply place all group names defined with `@jalopy.group` tags in the desired order here and all methods will be sorted accordingly.

There is no special requirement on how the logical group names should be written, but it is good practice to separate them by commas.

Example 2.716. Class declaration with @jalopy.group-order

```
 /**
  * I want methods ordered by value provided in the @jalopy.group tag of
  * each method in this order. If the method doesn't have a @jalopy.group tag,
  * fall back on project defaults.
  *
  * @jalopy.group-order  Constructors,Queries,Accessors
  */
class Foo {
    ...
}
```

This works recursively for all methods of a compilation unit, i.e. for inner classes Jalopy first checks the inner class declaration comment and if no `@jalopy.group-order` tag can be found, it recursively searches all parent class/interface declarations of the unit. A group name not only defines the sorting order, but is used for separator comments also (Refer to Section 2.8.11.3, "Comments" for more information on this feature).

Please note that all methods that have no custom group information associated are placed below the ones with grouping tags. Within each group the methods are sorted according to the normal criteria (access modifier, name, parameter count).

Since 1.1

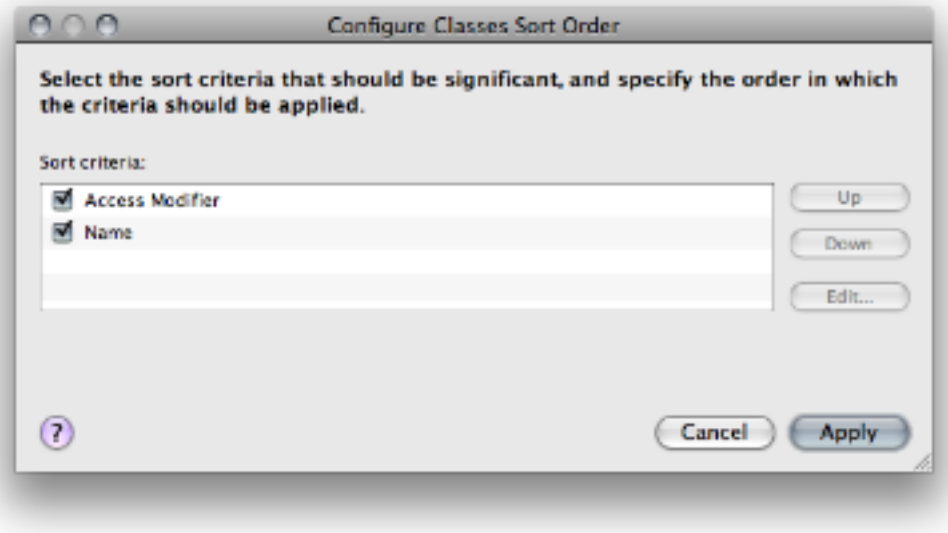
Example 2.717. Custom separator comments

```
//~ Queries ----------------------------------------------------------

/**
  * This method gets object by primary key
  *
  * @param  inConn  db conn
  * @param  inPK    the primary key
  *
  * @jalopy.group  Queries
  */
public static MyClass getByPK (Connection inConn, Long inPK) {
    ...
}

...

//~ Accessors --------------------------------------------------------

/**
  * This method returns the Name property.
  *
  * @jalopy.group  Accessors
  */
public String getName (Connection inConn, String inLoginName) {
    ...
}
```

Classes, Interfaces, Enums

Class, interface, enum and annotation declarations can be sorted according to access modifier and/or name.

Figure 2.54. Configure Sorting Order of Classes, Interfaces and Enums



To further refine the way sorting is applied for access modifiers, select the *Access Modifier* entry, press the *Edit...* button, configure what modifiers should be significant and adjust the order in which testing should be applied.

Example 2.718. Classes/Interfaces

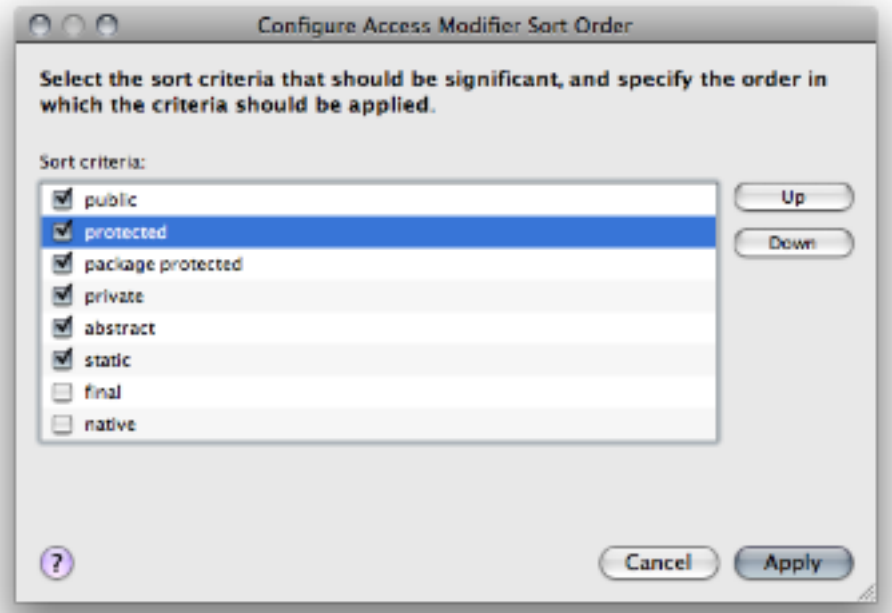
```
protected abstract class AbstractPage {
    ...
}
```

In the above example, the modifiers would be `"protected abstract"` and the name `"AbstractPage"`.

Access Modifier

Lets you define what modifiers should be significant when sorting by modifier and in which order the declarations should be sorted. The position of the *Modifiers* entry in the parent dialog defines when the modifiers are compared to determine the order of two declarations. E.g. if you want to sort by modifier first, *Modifiers* must be the topmost entry in the parent dialog.

Figure 2.55. Configure Sorting Order of Modifiers



Select the check box of each access modifier that should be used to determine the order of declarations and use the *Up* and *Down* buttons to define the order in which the declarations should be sorted.

For example, if you want to place all static methods together above the other ones, you would check the "static" modifier and move it to the top of the list.

Example 2.719. Sort by static modifier first, then access modifier

```
class Foo {

    public static void x() {}

    static void y() {}

    public void c() {}

    public void d() {}

    protected void b() {}

    private void a() {}
}
```

But if you only want to have the methods sorted by access modifier, just check the four access modifiers and specify the order in which the declarations are to be sorted, e.g. private, package protected, protected and public.

Example 2.720. Sort by access modifier

```
class Foo {

    private void a() {}

    static void y() {}

    protected void b() {}

    public void c() {}

    public void d() {}

    public static void x() {}
}
```

Require bean property field

By default all methods following the JavaBeans naming conventions are recognized as JavaBeans methods. In order to limit JavaBeans detection to methods that actually contain a corresponding property field, enable this option.

Bean property field detection is somewhat fuzzy as no attempt is made to match method names exactly in order to support field prefixes. Given the method name "getImportanceValue", Jalopy would match all fields that end with "importanceValue" as a corresponding property field (case is ignored). But a field name "importanceValue_" would not be matched.

Since 1.9

Example 2.721. Method declarations

```
private boolean importanceValue; // property field

public void setImportanceValue(boolean value) {}
public boolean isImportanceValue() {}
public void setTestValue(String value) {}
```

When the option is disabled, all methods in the above example would be recognized as JavaBeans methods and handled accordingly. But if the option is enabled, only the first two methods would be treated as JavaBeans methods, because only they contain a matching property field.

This option might affect the sorting of methods when the Bean Pattern criteria is enabled, please refer to Section 2.8.11.1.1, "Bean Pattern". It also impacts what Javadoc template might be chosen when generating Javadoc comments, see Section 2.8.14.5, "Templates".

Boolean Getter

Lets you configure the regular expression that is used to determine what method declarations are recognized as *Boolean Getters*. According to the JavaBeans naming conventions, only method declarations starting with the "`is`" prefix are Boolean Getters, but it might make sense to lift this restriction. In certain cases it is more reasonable to name methods in a way that better describes their purpose, but still treat them as Boolean Getters, like e.g. `canDelete()` or `shouldDelete()`.

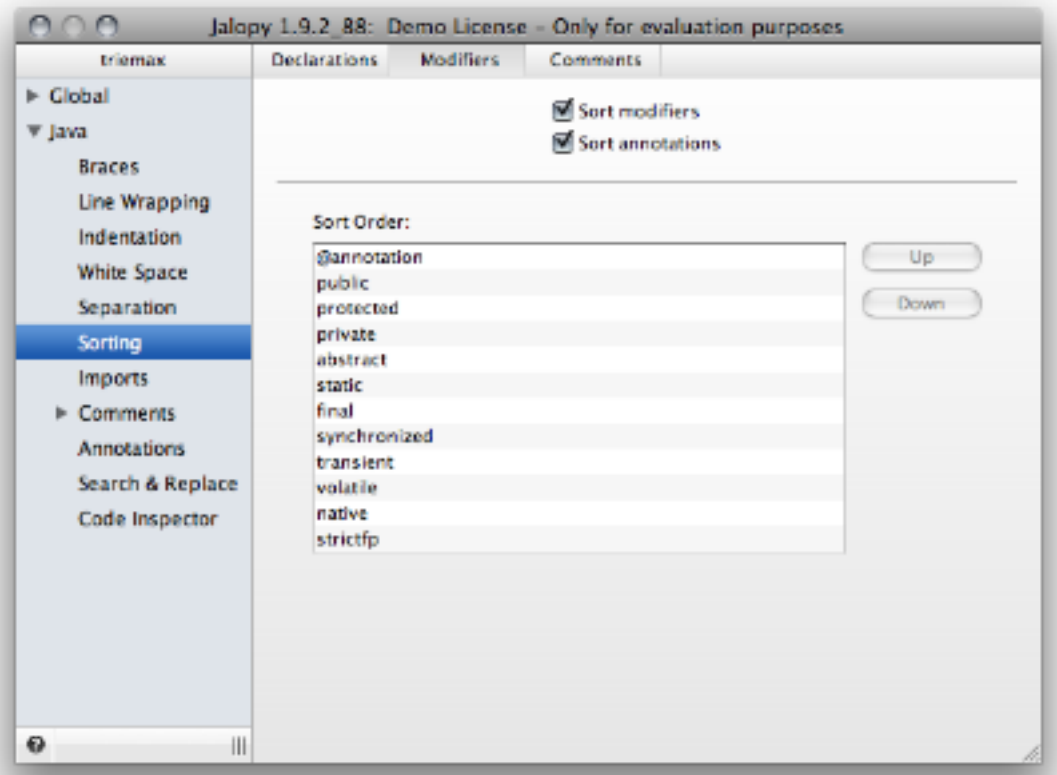
IMPORTANT The prefix must be enclosed with matching parentheses! Always use something like `^(is|should|can)[A-Z]\w+` rather than `^is|should|can[A-Z]\w+`.

Since 1.1

2.8.11.2 Modifiers

Controls the sorting of declaration modifiers.

Figure 2.56. Sorting Modifiers settings page



Sort modifiers

Enables or disables the sorting of modifiers. When disabled, the modifiers appear in their original order. Otherwise they are sorted according to the specified order (see below).

Sort annotations

Controls whether annotations should be sorted lexicographically.

Since 1.9

Example 2.722. Unsorted annotations

```
@Remote(Whatever.class)
@Interceptors(Test.class)
@Stateful
class Foo {}
```

Example 2.723. Sorted annotations

```
@Interceptors(Test.class)
@Remote(Whatever.class)
@Stateful
class Foo {}
```

Sort Order

Lets you specify the order in which the individual modifiers should appear. Select an entry in the list and use the *Up* and *Down* buttons to move it to the desired location. The list contains the different available Java modifiers as of J2SE 6.0. The `@annotation` entry represents annotations.

Example 2.724. Marker annotation placed before public modifier

```
@Preliminary public class TimeTravel {
    ...
}
```

Example 2.725. Marker annotation placed after public modifier

```
public @Preliminary class TimeTravel {
    ...
}
```

Please note that normal annotations and single-member annotations are always printed before all other modifiers!

Example 2.726. Normal annotation

```
@RequestForEnhancement(
    id       = 2868724,
    synopsis = "Provide time-travel functionality",
    engineer = "Mr. Peabody",
    date     = "4/1/2004"
)
public static void travelThroughTime(Date destination) {
    ...
}
```

Example 2.727. Single-member annotation

```
@Copyright("2002 Yoyodyne Propulsion Systems, Inc., All rights reserved.")
public class OscillationOverthruster {
    ...
}
```

2.8.11.3 Comments

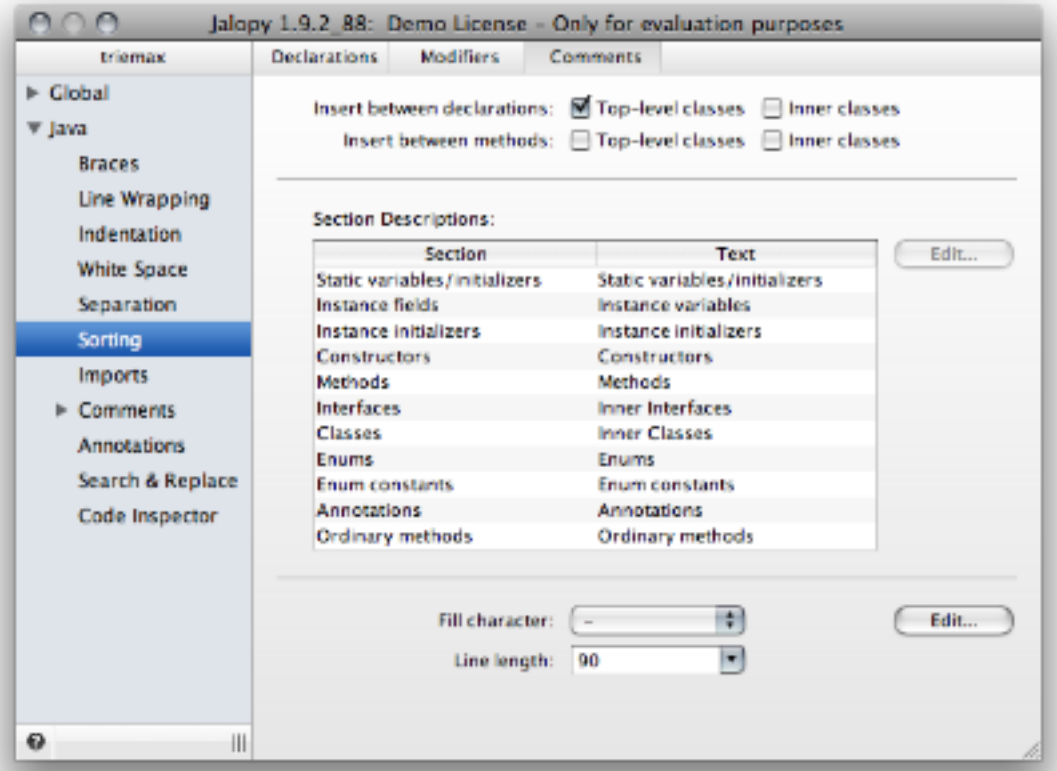
Lets you control the behavior of the separator comments. When the sorting of declarations is enabled, separator comments may be inserted before every element section to make it easier to identify the different parts of a source file.

A separator comment usually starts with a leading `//~` followed by the specified description text of a section and a certain number of fill characters to take up the rest of the space. But the style of the comments is fully configurable as well.

Example 2.728. Separator comment

```
//~ Methods ------------------------------------------------------------------
```

Figure 2.57. Comments Separator settings page



Insert

Controls when separator comments should be inserted.

Between sections

Enables the insertion of separator comments between the different code sections of a compilation unit. You can control the appearance of the comments as described in Section 2.8.11.3.2, “Separator Comment Descriptions” and Section 2.8.11.3.3, “Separator Comment Style”.

Example 2.729. Separator comments

```
public class Foo {

    //~ Static fields/initializers -----------------------------------

    static final String LABELED_BY_PROPERTY = "labeledBy";

    //~ Instance fields ----------------------------------------------

    private Icon defaultIcon = null;

    //~ Constructors -------------------------------------------------

    public Foo(String text, Icon icon) {
        ...
    }

    public Foo(String text) {
        ...
    }

    //~ Methods ------------------------------------------------------

    public Icon getDisabledIcon() {
        ...
    }

    public Icon getIcon() {
        ...
    }

    //~ Inner Classes ------------------------------------------------

    protected class FooContainer {
      ...
    }
}
```

IMPORTANT The option requires the "Sort declarations" option to be enabled in order to take effect

Between sections of inner classes

The insertion of separator comments for inner classes/interfaces may lead to confusion, therefore you can control it here separately.

Example 2.730. Separator comments

```
public class Foo {

    //~ Static fields/initializers ---------------------------------

    static final String LABELED_BY_PROPERTY = "labeledBy";

    //~ Instance fields --------------------------------------------

    private Icon defaultIcon = null;

    //~ Constructors -----------------------------------------------

    public Foo(String text, Icon icon) {
        ...
    }

    public Foo(String text) {
        ...
    }

    //~ Methods ----------------------------------------------------

    public Icon getDisabledIcon() {
        ...
    }

    public Icon getIcon() {
        ...
    }

    //~ Inner Classes ----------------------------------------------

    protected class FooContainer {

        public Component getParent() {
            ...
        }

        //~ Methods ------------------------------------------------

        public int getComponentCount() {
            ...
        }
    }
}
```

IMPORTANT The option requires the "Sort declarations" option to be enabled in order to take effect

Between methods

When enabled, separator comments are inserted between method declarations.

Since 1.3

Example 2.731. Method comment separator

```
/**
 * Returns the value of the disabledIcon property if it's been set
 *
 * @return The value of the disabledIcon property.
 */
public Icon getDisabledIcon() {
    ...
}

//~ ----------------------------------------------------------------

/**
 * Return the keycode that indicates a mnemonic key.
 *
 * @return int value for the mnemonic key
 */
public int getDisplayedMnemonic() {
    ...
}
```

IMPORTANT The option requires the "Sort declarations" option to be enabled in order to take effect

Between methods of inner classes

When enabled, separator comments are inserted between method declarations of inner classes.

Since 1.7

IMPORTANT "Sort declarations" must be enabled in order to take effect

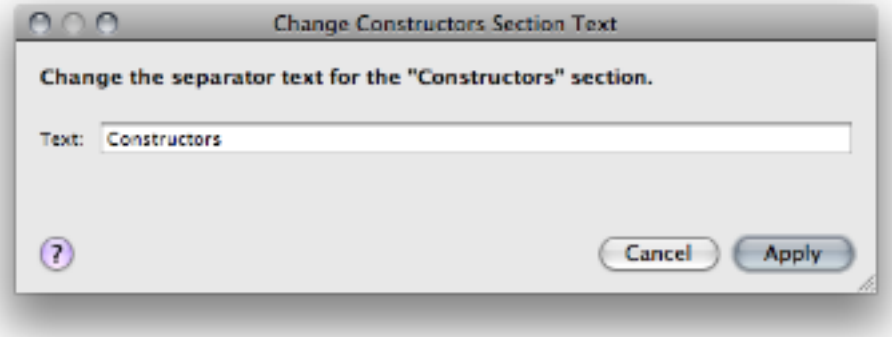
Example 2.732. Method comment separator of inner classes

```
public class Foo {

    ...

    static class Item {

        /**
         * Returns the value of the disabledIcon property if it's been
         * set
         *
         * @return The value of the disabledIcon property.
         */
        public Icon getDisabledIcon() {
            ...
        }

        //~ ////////////////////////////////////////////////////////////

        /**
         * Return the keycode that indicates a mnemonic key.
         *
         * @return int value for the mnemonic key
         */
        public int getDisplayedMnemonic() {
            ...
        }
    }
}
```

Descriptions

Lets you define the description texts for the individual code sections. Select a row in the list and press the *Change...* button to invoke a dialog that lets you specify the text for a specific section. The dialog may be invoked directly by double-clicking on the list.

Figure 2.58. Configure Section Description Text



If you want to disable the insertion for specific sections, you can achieve this means by removing the corresponding description text. Please note that the “Ordinary methods” text is only used when custom grouping is enabled, and specifies the section name for those methods that have no custom grouping info associated (see Section 2.8.11.1.1, “Custom Sort Order”).

Style

Lets you control the style of separator comments.

Fill character

Lets you define the fill character that should be used in comments.

Example 2.733. Fill character styles

```
//~ Methods

//~ Methods ..............................................................

//~ Methods --------------------------------------------------------------

//~ Methods ==============================================================

//~ Methods **************************************************************

//~ Methods //////////////////////////////////////////////////////////////
```

Edit...

Press the *Edit...* button to adjust the templates that will be used for separator comments. The templates may contain one or several single-line or multi-line comments. You can configure two different templates: one for the comments between the different declaration sections, and one for comments between method declarations.

Example 2.734. Default separator template

```
//~ ${description} ${fill.character}*
```

As you can see in the example above, certain variables may be used that are substituted during formatting to include the code section description or stretch the comment to completely fill a line.

Table 2.6. Separator template variables

Variable	Description
${description}	Lets you include the description text of the current code section (Refer to Section 2.8.11.3.2, "Separator Comment Descriptions" for information on how to adjust the descriptions).
${fill.character}	Lets you include the fill character as defined by the corresponding combo box. Please note that you have to place an asterix (*) after the variable if you want to have a comment line stretched to the full line length.

Please note that Jalopy needs a way to differ between user comments and separator comments, because separator comments must be removed on each run in order to ensure correct locations and behavior. Jalopy recognizes all single-line comments starting with `//~` as separator comments. If you would rather use a multi-line comment instead or don't like the default identifier, you need to make sure that the comments get removed. This means can be achieved by adding a unique identifier into the template and configure a custom removal pattern for it (see Section 2.8.13.3, "Comment Removal").

Example 2.735. Multi-line comment separator template

```
/**${fill.character}*
 *    ~#~       ${description}:
 *${fill.character}*//
```

In the example above, ~#~ would be configured as the removal pattern.

Since 1.4

Line length

Lets you define the maximal line length for separator comments. The specified fill character might be used to increase the length of a separator comment to span exactly until the specified line length.

Since 1.2.1

2.8.12 Imports

Controls the handling of import declarations. With Java, import declarations are used to make types available within a compilation unit. There are two types of import declarations:

Single-type imports import a single named type.

Example 2.736. Single-type imports

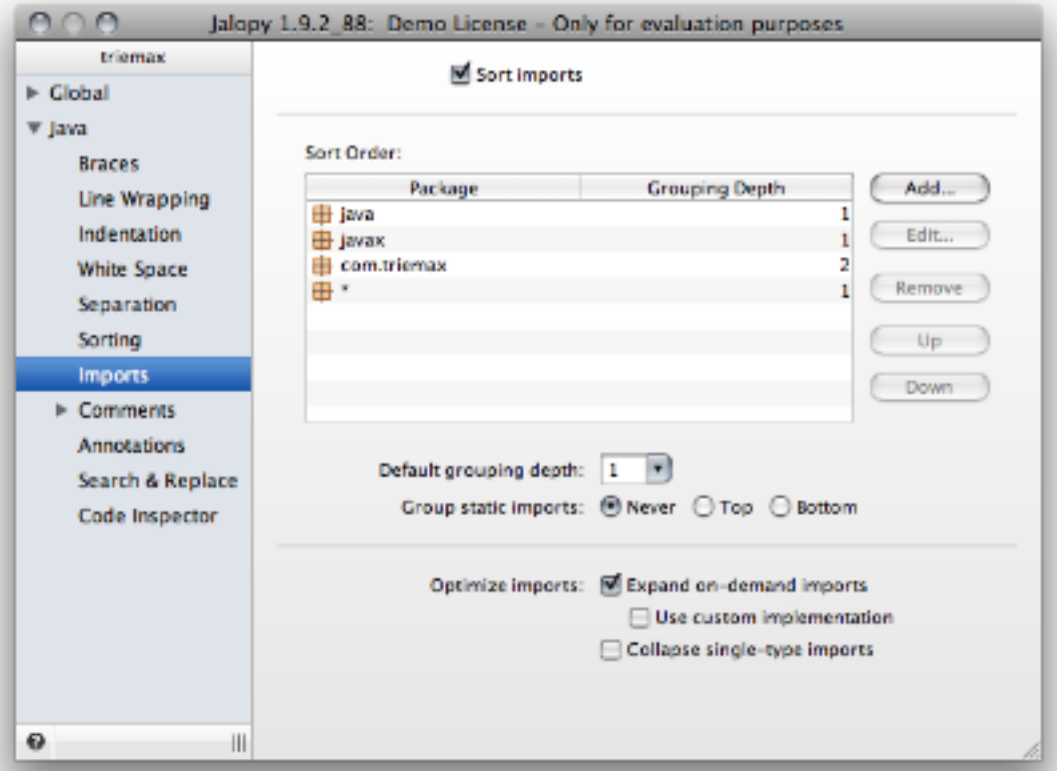
```
import java.util.ArrayList;
import java.util.List;
```

On-demand imports import all accessible types declared in the type or package.

Example 2.737. On-demand imports

```
import java.util.*;
import java.util.regex.*
```

Figure 2.59. Imports settings page



2.8.12.1 General

Lets you control the general imports settings.

Sort imports

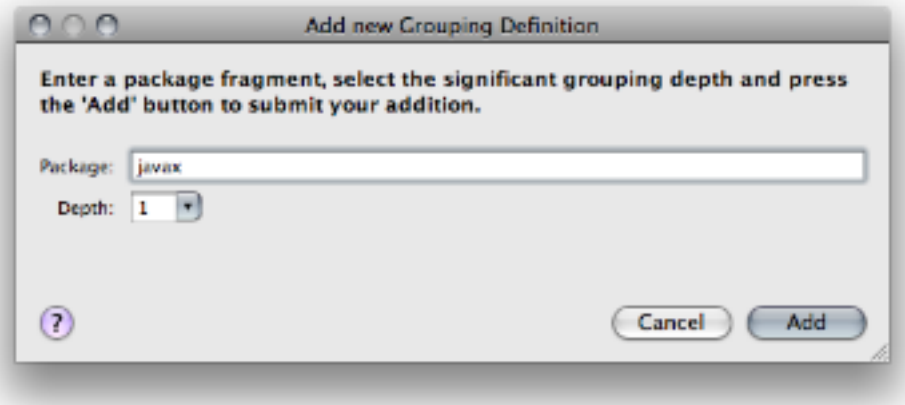
Enables or disables the sorting of import declarations. Enabling this option will sort all declarations. When disabled, the declarations are printed in their original order.

Sort Order

Lets you control the sorting order and grouping behavior of import declarations. When sorting is enabled, import declarations will be sorted according to the positions of the package names as specified in the list component. To specify the order in which the declarations should appear, you can use the *Up* and *Down* buttons.

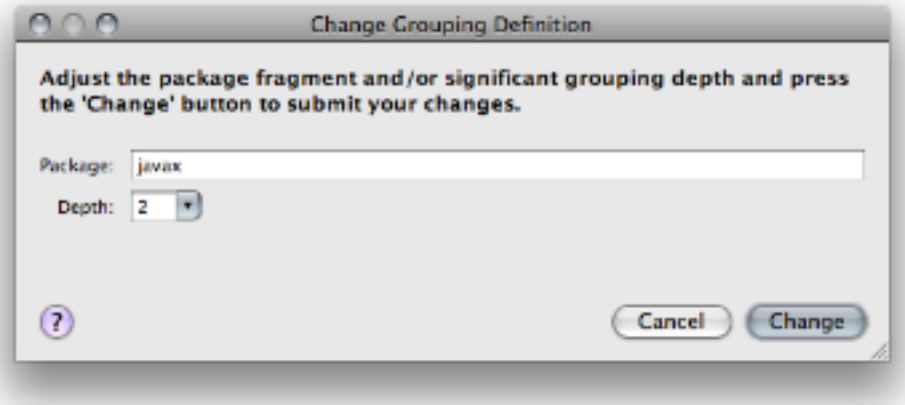
You can add/remove package names (e.g. javax, javax.swing or com.foo.sarah) to and from the list via the *Add...* and *Remove* buttons. A dialog appears that lets you add a new grouping definition. The star character (*) represents all undefined packages and cannot be removed.

Figure 2.60. Add new Grouping Definition



If you want to change an existing grouping definition, you can do so by selecting the definition you want to change in the list, and pressing the *Change...* button. A dialog appears that lets you change an existing grouping definition. Alternatively, you can invoke the dialog by just double-clicking with the mouse on the grouping definition you want to change.

Figure 2.61. Change existing Grouping Definition



Note that the * that represents all undefined packages, cannot be changed (you can only adjust the grouping depth).

Grouping

In addition to sorting, associated declarations can be grouped together to reduce complexity by packing related information into a common unit. Grouping means that associated declarations are separated by one blank line. Grouping only happens if sorting is enabled.

Via the grouping depth you can control how many parts of a package name should be considered when determine whether two import declarations are to be grouped together. Grouping only happens when all relevant parts are equal. So via the grouping depth you can effectively specify how many package name parts are relevant.

Default grouping depth

This switch lets you define the default grouping depth that should be used when no grouping depth was defined for a specific package name (see below). To disable grouping at all, set the default grouping depth to '0'.

Example 2.738. Grouping depth == 1

```
import java.awt.Color;
import java.awt.Component;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.util.List;
```

Only the first part of the package name will be used to determine grouping.

Example 2.739. Grouping depth == 2

```
import java.awt.Color;
import java.awt.Component;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.util.ArrayList;
import java.util.List;
```

The first two parts of the package name will be used to determine grouping.

Example 2.740. Grouping depth == 3

```
import java.awt.Color;
import java.awt.Component;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.util.ArrayList;
import java.util.List;
```

The first three parts of the package name will be used to determine grouping.

Group static imports

To better differ between standard import declarations and the static imports introduced in J2SE 5.0, you can control whether static imports should be grouped separately. You can select whether static import declarations should be printed together with the standard import declarations ("Never") or placed above ("Top") or below ("Bottom") all standard import declarations.

Since 1.4

Example 2.741. Mixed static/standard import declarations

```
import java.awt.BorderLayout;

import javax.swing.SwingUtilities;
import static javax.swing.WindowConstants;

import z.Foo;
```

Example 2.742. Static import declarations placed above

```
import static javax.swing.WindowConstants;

import java.awt.BorderLayout;

import javax.swing.SwingUtilities;

import z.Foo;
```

Example 2.743. Static import declarations placed below

```
import java.awt.BorderLayout;

import javax.swing.SwingUtilities;

import z.Foo;

import static javax.swing.WindowConstants;
```

2.8.12.2 Optimization

Lets you optimize the import declarations by either expanding or collapsing them, obsolete or unused imports are removed.

NOTE When using either one of the Ant, Console or Maven Plug-ins, you have to explicitly configure the class path for this feature to work. Please refer to the documentation of the individual Plug-ins to learn how one can accomplish this (see Part II, "Plug-ins")

Expand on-demand imports

When enabled, tries to expand all on-demand import declarations. Expanding means to resolve all on-demand imports (sometimes called wildcard imports) and replace them with single-type imports (sometimes called explicit imports) of the types that are actually used in the source file.

TIP Sorting (see Section 2.8.12.1, "Sort imports") should be enabled when different IDEs are used, as otherwise implementation details may yield to differences regarding import placement.

Single-type imports have several advantages and should be preferred over on-demand imports.

- They avoid any class path conflicts that could break your code when a class is added to a package you import
- They make dependencies explicit, so that anyone who has to read your code later knows what you meant to import and what you didn't mean to import
- They can make some compilation faster, because the compiler doesn't have to search the whole package to identify dependencies, though this is usually not a huge deal with modern compilers

Example 2.744. On-demand import declaration

```
import java.util.*;
```

could become

Example 2.745. Single-type import declarations

```
import java.util.ArrayList;
import java.util.List;
```

In the examples above, the on-demand import declaration has been expanded into two single-type import declarations that reference the needed types for this package.

Use custom implementation

The IDE Plug-ins usually leverage the build-in import optimization facility, but sometimes these may contain bugs that prevent their usage. Here, you can enable a custom implementation instead. It's the same implementation that is used by the non-IDE Plug-ins.

NOTE Please note that the custom import optimization implementation only supports expanding on-demand imports

Since 1.7

Collapse single-type imports

When enabled, tries to collapse all single-type declarations. Collapsing means to remove all single-type imports of a given package and replace them with one on-demand import declaration.

Example 2.746. Single-type import declarations

```
import java.awt.event.MouseEvent;
import javax.swing.JButton;
import javax.swing.JTable;
import javax.swing.JTextField;
```

could become

Example 2.747. On-demand import declarations

```
import java.awt.event.*;
import javax.swing.*;
```

Please note that there might be collisions that prevent collapsing when two types have the same name.

Example 2.748. Single-type import declarations

```
import java.awt.Color;
import java.util.List;
```

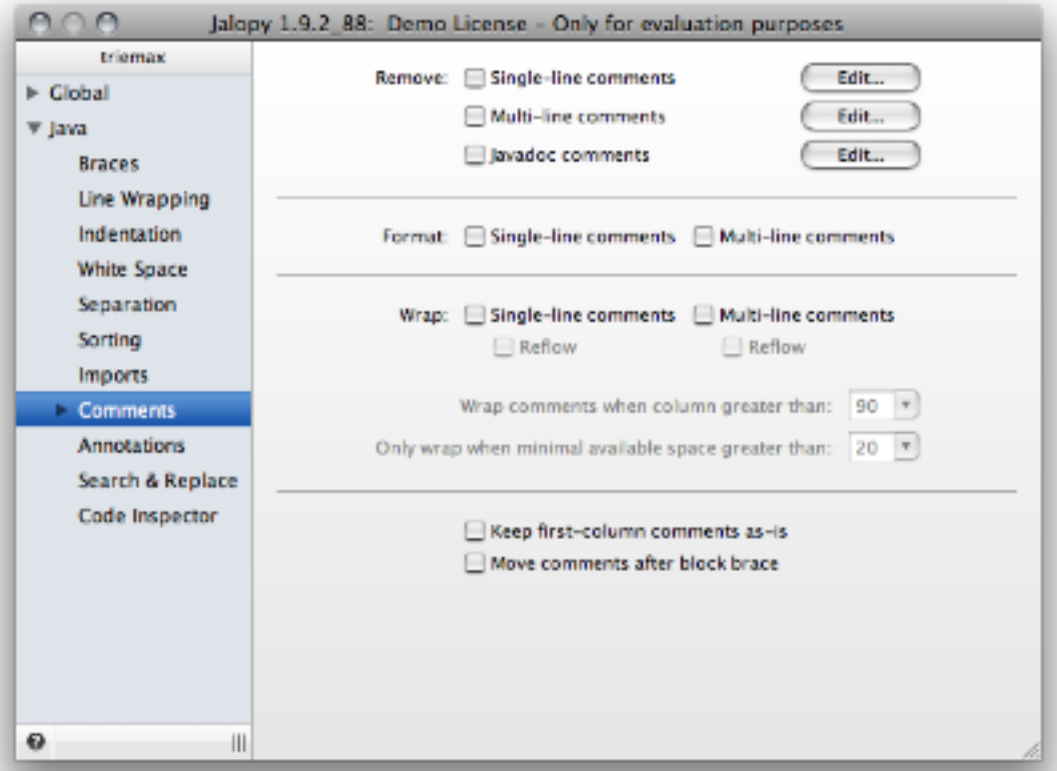
In the example above, collapsing both packages is not possible because this would lead to invalid code, as both `java.awt` and `java.util` contain a type named `List`. In such cases only one package will be collapsed (if no further conflicts are detected).

NOTE The NetBeans Plug-in currently does not support import collapsing.

2.8.13 Comments

Controls how Jalopy handles comments in source files.

Figure 2.62. Comments settings page



2.8.13.1 Comment types

Describes what comments Jalopy recognizes and how they are treated. As far as Jalopy is concerned, there are five types of comments:

Single-line comments

An *end-of-line comment*: all text from the ASCII characters `//` to the end of the line

Example 2.749. Single-line comment

```
// [PENDING] this should be part of the ErrorManager
```

Single-line comments are normally printed *as-is*, but can be formatted as well (see Section 2.8.13.4, “Format”).

Multi-line comments

A *traditional comment*: all text from the ASCII characters `/*` to the ASCII characters `*/`

Example 2.750. Multi-line comment

```
/*
public int getSubregionStartOffset(int line, int subregion)
{
    ChunkCache.LineInfo[] lineInfos = cache.getLineInfosForPhysicalLine(line);
    return buffer.getLineStartOffset(lineInfos[subregion].physicalLine)
        + lineInfos[subregion].offset;
}
*/
```

Multi-line comments are normally printed *as-is*, but can be formatted as well (see Section 2.8.13.4, "Format").

Javadoc comments

A *documentation comment*: actually a special kind of multi-line comment as defined by the Sun Javadoc specification; all text from the ASCII characters `/**` to the ASCII characters `*/`.

Example 2.751. Javadoc comment

```
/**
 * A scroll listener will be notified when the text area is scrolled, either
 * horizontally or vertically.
 */
```

Javadoc comments are normally printed *as-is*, but can be formatted according to the code convention settings (see Section 2.8.14.1, "Format").

Separator comments

A Jalopy-specific *separator comment*: actually a special kind of single-line comment; all text from the ASCII characters `//~` to the end of the line.

Example 2.752. Separator comment

```
//~ Inner classes ------------------------------------
```

Separator comments are always removed during parsing and may be re-inserted during emitting according to the code convention settings. Refer to Section 2.8.11.3, "Comments" for more information about separator comments.

Pragma comments //J[directive]

A Jalopy-specific *control comment*: actually a special kind of single-line comment; all text from the ASCII characters `//J` to the end of the line. Pragma comments are always printed *as-is*.

Currently, Jalopy supports the following pragma comments:

`//J-` and `//J+`

Lets you selectively disable formatting for certain code sections. `//J-` tells Jalopy to disable formatting until `//J+` will enable it again. All code between (and including) the two comments will be printed *as-is*.

Example 2.753. Pragma comments

```
//J-
     if (operator.equals("EQ"))  return (left == right);
else if (operator.equals("NE"))  return (left != right);
else if (operator.equals("LT"))  return (left <  right);
else if (operator.equals("GT"))  return (left >  right);
else if (operator.equals("LE"))  return (left <= right);
else if (operator.equals("GE"))  return (left >= right);
else
{
    throw new IllegalArgumentException("Unknown int if operator: " +
                                       operator);
}
//J+
```

IMPORTANT The two comments must always be used in conjunction and they must always appear on a line by themselves. Never place them after code elements!

`//JDOC-`

When placed in front of a Javadoc comment, disables the Javadoc generation feature no matter what the code convention dictates.

Example 2.754. //JDOC- comment

```
/**
 * DOCUMENT ME!
 */
public class Test {

    //JDOC-
    public void test(String input) {

        ...
    }
}
```

`//J:KEEP+`

Instructs Jalopy to keep existing line breaks within array initializers and parameter or argument lists. The comment must be placed either directly after the left curly brace of array initializers, or the left parenthesis of parameter or argument lists, or the first element of the initializer or list.

Since 1.9.3

Example 2.755. Keep line breaks within call argument list

```
FormLayout layout = new FormLayout( //J:KEEP+
    "fill:d:grow(1.0), 15px",
    "7px, d, 7px, fill:d:grow(1.0)");
```

Example 2.756. Keep line breaks within array initializer

```
String[] options =
    { //J:KEEP+
        "-jar", aLauncher.getAbsolutePath(),
        "-application", "org.foo.application.Main",
        "-debug",
        "-consolelog"
    };
```

Please refer to the documentation of the wrapping options in Section 2.8.7.1.2, “Keep line breaks”, for information how to configure the general wrapping behavior.

2.8.13.2 Comment association

Jalopy associates comments using an assignment heuristics considering empty lines and/or column offsets. If you plan to switch between several code conventions on a regular basis, e.g. to toggle between personal preference and company guidelines, you should make sure

to use similar separation settings in order to avoid any association differences. It is good practice to always craft a code convention that reflects the personal style from a master code convention (usually the company code convention) and only adjust those settings that run counter to personal taste. Please refer to the Profiles section for more information.

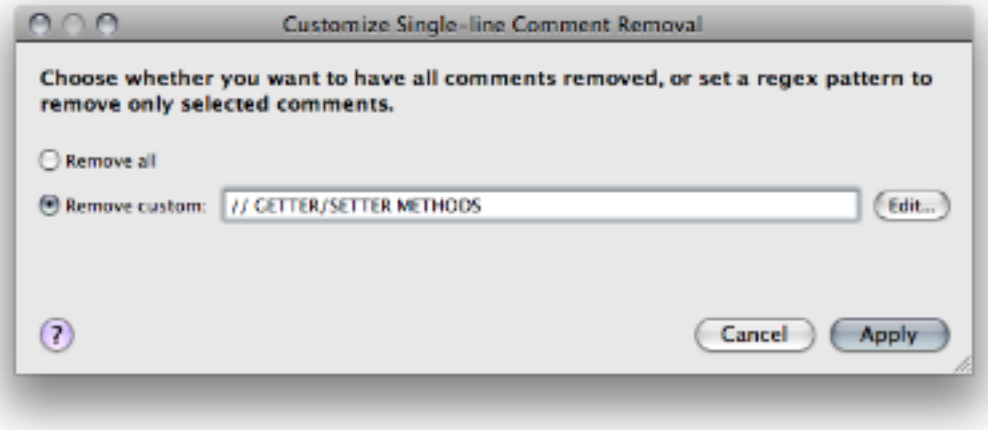
2.8.13.3 Remove

Controls whether and what types of comments should be removed during formatting.

Single-line comments

When enabled, removes all single-line comments found in a source file that matches certain criteria. To customize what single-line comments should be removed, you can use the *Customize...* button to specify the desired behavior (since 1.1).

Figure 2.63. Configure single-line comment removal



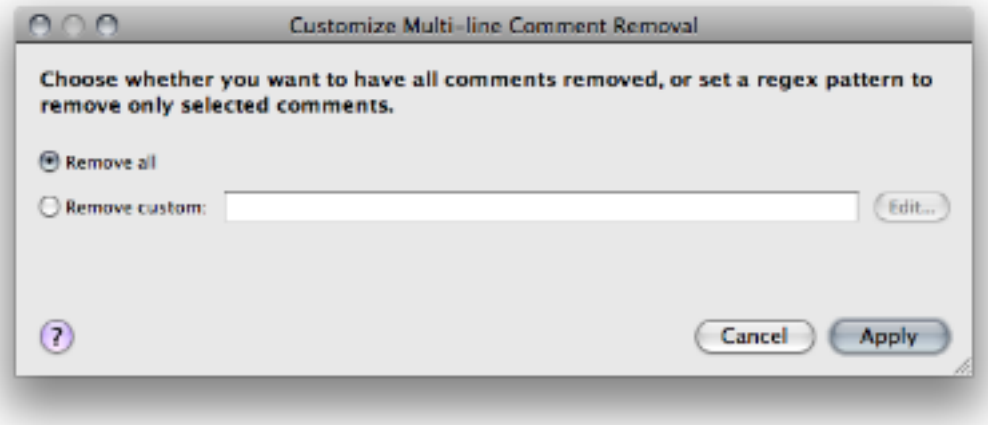
You can choose whether all single-comments should be removed or specify a regular expression (regex) to match only certain comments. Note that the regex defines a pattern that is contained in a comment - it must not match exactly.

You can either enter the regex directly into the provided text field or craft one with the help of a little tool that lets you interactively test the validity of your regex. You can invoke the regex helper via the *Change...* button. Note that the *Remove custom* radio box must be selected in order to be able to change the regex. The specified regular expression is only matched against the contents of comments, not any surrounding code elements! The regex helper is explained in detail below, see Section 2.8.13.3.1, “Regular expression tester” for more information.

Multi-line comments

When enabled, removes all multi-line comments (sometimes called block comments) found in a source file that matches certain criteria. To customize what multi-line comments should be removed, you can use the *Customize...* button to specify the desired behavior (since 1.1).

Figure 2.64. Configure multi-line comment removal



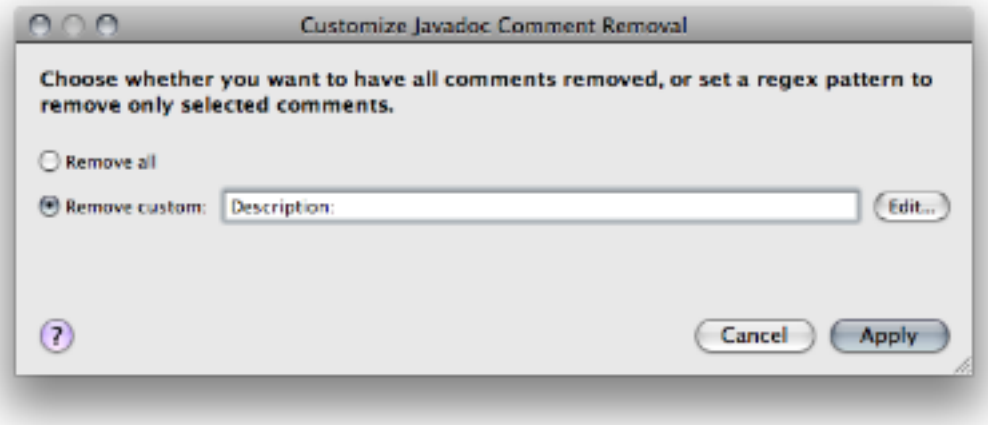
You can choose whether all multi-comments should be removed or specify a regular expression (regex) to match only certain comments. Note that the regex defines a pattern that is contained in a comment, it must not match exactly.

You can either enter the regex directly into the provided text field or craft one with the help of a little tool that lets you interactively test the validity of your regex. You can invoke the regex dialog via the *Change...* button. Note that the *Remove custom* radio box must be selected in order to be able to change the regex. The specified regular expression is only matched against the contents of comments, not any surrounding code elements! The regex tester is explained in detail below, see Section 2.8.13.3.1, "Regular expression tester" for more information.

Javadoc comments

When enabled, removes all Javadoc comments found in a source file that matches certain criteria. This may prove useful in conjunction with the Javadoc auto-generation capabilities to build Javadoc from scratch. To customize what Javadoc comments should be removed, you can use the *Customize...* button to specify the desired behavior (since 1.1).

Figure 2.65. Configure Javadoc comment removal



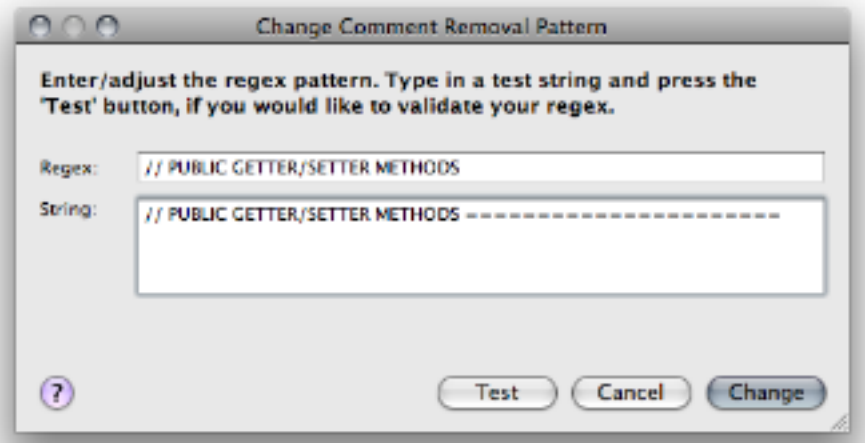
You can choose whether all single-comments should be removed or specify a regular expression (regex) to match only certain comments. Note that the regex defines a pattern that is contained in a comment, it must not match exactly.

You can either enter the regex directly into the provided text field or craft one with the help of a little tool that lets you interactively test the validity of your regex. You can invoke the regex dialog via the *Change...* button. Note that the *Remove custom* radio box must be selected in order to be able to change the regex. The specified regular expression is only matched against the contents of comments, not any surrounding code elements! The regex tester is explained in detail below, see Section 2.8.13.3.1, "Regular expression tester" for more information.

Regular expression tester

The regular expression tester lets you interactively craft a valid regular expression that contains a certain test string.

Figure 2.66. Regular expression tester



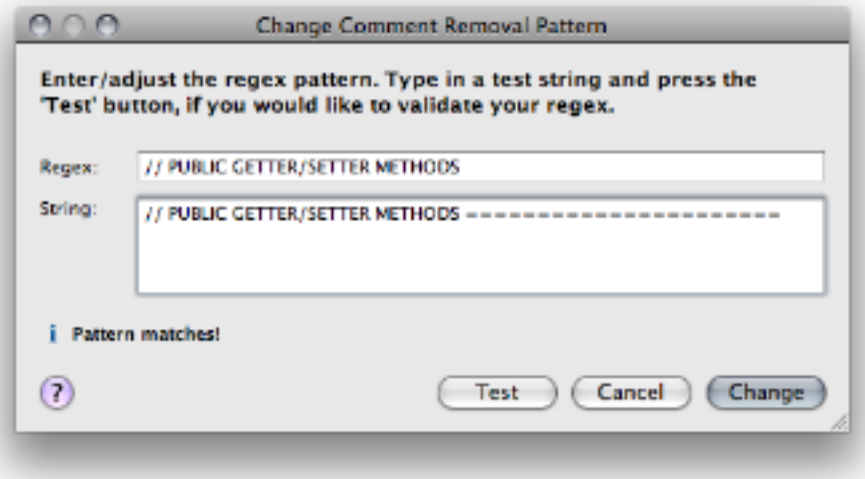
Regex

The *Regex* text field is where you have to insert the regular expression. This text field initially contains the current pattern for the comment type that is under construction. Jalopy uses Java's build-in regular expression engine which is roughly equivalent with Perl 5 regular expressions. The syntax is explained here: http://java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html. For a more precise description of the behavior of regular expression constructs consult *Mastering Regular Expressions* [Friedl97].

String

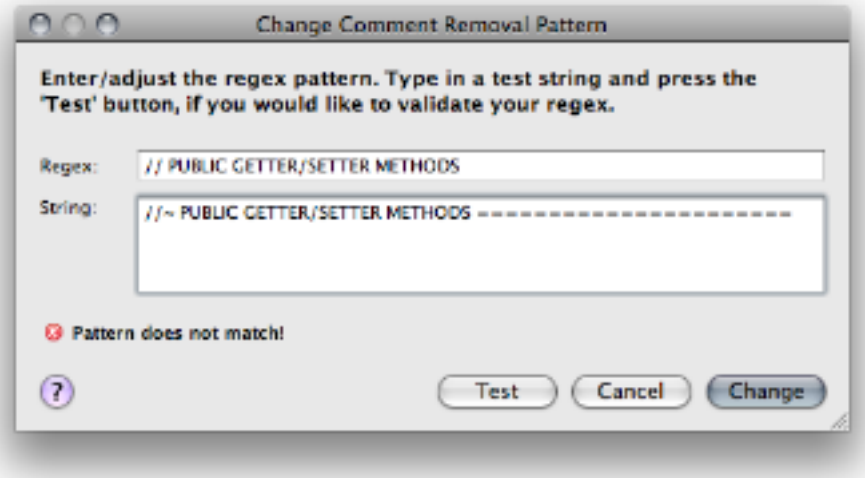
The *String* text field is where you have to enter a string that should be matched by the specified regular expression. This text field is initially empty. Once you have edited the two text fields you may want to use the *Test* button to perform a pattern matching test in order to make sure that the specified pattern matches as desired. If testing is successful, a green-colored message appears, to indicate that fact.

Figure 2.67. Successful regex test



Otherwise a red-colored message is displayed, and you may want to change your pattern and/or test string and restart the procedure.

Figure 2.68. Failed regex test



When you are done editing the regular expression, you can press the *Apply* button to take over (note that you are not required to perform any testing, the regex is accepted even when invalid!). You can always use the *Cancel* button to cancel editing at any time. The dialog will be closed and no changes applied.

2.8.13.4 Format

Controls the reformatting of comments.

Single-line comments

Enables the reformatting of single-line comments. Only affects the space between leading delimiter and comment text as shown in the examples below.

Since 1.0.3

Example 2.757. Single-line comment

```
//Sometimes people run
//the comments against the delimiters
```

Example 2.758. Single-line comment (reformatted)

```
// Sometimes people run
// the comments against the delimiters
```

Multi-line comments

Enables the reformatting of multi-line comments. Only affects the leading asterixes of consecutive comment lines as shown in the examples below.

Example 2.759. Multi-line comment

```
/* Multi-line
* comment.
* end.
*/
```

Example 2.760. Multi-line comment (reformatted)

```
/* Multi-line
 * comment.
 * end.
 */
```

Please note that as of Jalopy 1.6 you can disable formatting of individual comments using the special `/*-` delimiter.

Example 2.761. Multi-line comment that keeps its style

```
/*- Comment that
* should NOT
* be formatted
*/
```

2.8.13.5 Wrap

Controls the wrapping behavior of comments.

Single-line comments

When enabled, Jalopy tries to ensure that single-line comments do not exceed the maximal line length.

Since 1.0.3

Example 2.762. A long single-line comment

```
                            |
// this is a long comment so I would like it to wrap
                            |
```

Example 2.763. A long single-line comment that was wrapped

```
                            |
// this is a long comment so|
// I would like it to wrap  |
                            |
```

Reflow

By default, wrapping happens on a line-by-line basis, i.e. every line that would exceed the maximal length will be split and the resulting chunks will be each printed on a line of its

own. To put it in another way: all existing line breaks are kept. This is no problem with a single comment, but when there are multiple comments in row, wrapping may leave annoying artifacts like in the example below.

Since 1.0.3

Example 2.764. Wrapped single-line comments

```
// when comments are reflowed (both single and multi-line), empty  |
// comment lines are kept to allow some                             |
// sort of control how things are broke up                          |
//                                                                  |
// otherwise it might be very dangerous to use this feature.        |
// The choice is yours                                              |
```

A better strategy might be to ignore existing line breaks and have the comments reflowed.

Example 2.765. Reflowed single-line comments

```
// when comments are reflowed (both single and multi-line), empty  |
// comment lines are kept to allow some sort of control how things |
// are broke up                                                     |
//                                                                  |
// otherwise it might be very dangerous to use this feature. The    |
// choice is yours                                                  |
```

Please note that existing blank lines are always kept!

Multi-line comments

When enabled, ensures that multi-line comments do not exceed the maximal line length.

Since 1.0.3

Example 2.766. A long multi-line comment

```
                            |
/* A multi-line comment that spans multiple lines but exceeds the max.
 * line length              |
 */                         |
                            |
```

Example 2.767. A long multi-line comment that was wrapped

```
                            |
/* A multi-line comment that|
 * spans multiple lines but |
 * exceeds the max.         |
 * line length              |
 */                         |
                            |
```

Please note that as of Jalopy 1.6 you can disable wrapping of individual comments using the special `/*-` delimiter.

Example 2.768. Multi-line comment that keeps its style

```
/*- Comment that
* should NOT
* be wrapped
*/
```

Reflow

Works similar to single-line comments (see Reflow single-line comments).

Since 1.0.3

Example 2.769. A long multi-line comment that was reflowed

```
                            |
/* A multi-line comment that|
 * spans multiple lines but |
 * exceeds the max. line    |
 * length                   |
 */                          |
                            |
```

Compare this to the result of Example 2.767, "A long multi-line comment that was wrapped" and see how the last two lines differ.

Wrap comments when line length greater than

Lets you define the maximal column width that comments are allowed to use. Jalopy keeps the comments within this range. This option is only available with either "Wrap single-line comments" or "Wrap multi-line comments" enabled. Please note that this setting only covers non-Javadoc comments. Javadoc comments are controlled independently, see "Javadoc line length".

Since 1.6

Only wrap when space greater than

Lets you define the minimal amount of horizontal space that is required to let wrapping occur. This is the space between the current line offset and the maximal line length as defined above (see "Comment Line Length"). If the difference between these two boundaries is greater than the specified lower bound, wrapping occurs. In contrast, if the difference between current offset and maximal line length is smaller or equal to the specified lower bound, no wrapping will occur.

Please note that this option is only available with either "Wrap single-line comments" or "Wrap multi-line comments" enabled.

Since 1.0.3

Example 2.770. Insufficient space, wrapping impossible

```
                            |                                 |    |
System.out.println("Hello");|// quite a long endline comment that
                            |// appears after the statement     |
                            |                                 |    |
                            O                                 M    S
```

In the above example the space between the current column offset [O] and the maximal line length [M] is smaller than the specified minimal width (space between [O] and [S]), therefore wrapping is not possible.

Example 2.771. Sufficient space, wrapping possible

```
                            |                          |    |
System.out.println("Hello");|// quite a long endline   |    |
                            |// comment that appears after |
                            |// the statement          |    |
                            |                          |    |
                            O                          S    M
```

In the above example the space between the current column offset [O] and the maximal line length [M] is greater than the specified minimal width (space between [O] and [S]), therefore wrapping is performed.

2.8.13.6 Misc

Lets you control miscellaneous comment options.

Keep first column comments as-is

When enabled, first column comments are never formatted and/or wrapped. As the name implies, first column comments are those comments that start at column one. They are typically used for commenting out blocks of code during development—something you might not want to be changed by a formatter.

Since 1.6

Example 2.772. First column comment

```
// System.out.println("appendingRemainingName: " + name.toString());
// Exception e = new Exception();
// e.printStackTrace();
```

Example 2.773. Wrapped first column comment

```
// System.out.println("appendingRemainingName: " + name.toString()); Exception
// e = new Exception(); e.printStackTrace();
```

Move comments after brace block

When enabled, single comments that appear in the first line after the left curly brace of a statement block and only cover one line, are moved right after the brace. This way you can achieve a more dense layout in case you want to save vertical space.

Since 1.7

Example 2.774. Comments after left curly braces

```
void test()
{
    /**
     * @todo evaluate whether this is still necessary
     */
    if (condition1)
    {
        // i should do something
        doSomething();
    }
    else if (condition2)
    {
        // [PENDING] this should be part of whatever, ask Jeff
        // what to do
        takeAction();
    }
}
```

When the option is enabled, the example above would be printed as:

Example 2.775. Comments after left curly braces

```
void test()
{
    /**
     * @todo evaluate whether this is still necessary
     */
    if (condition1)
    { // i should do something
        doSomething();
    }
    else if (condition2)
    {
        // [PENDING] this should be part of whatever, ask Jeff
        // what to do
        takeAction();
    }
}
```

Please note how only the single comment within the if statement is affected. The comment before the if statement and the consecutive comments after the else statement have been left untouched.

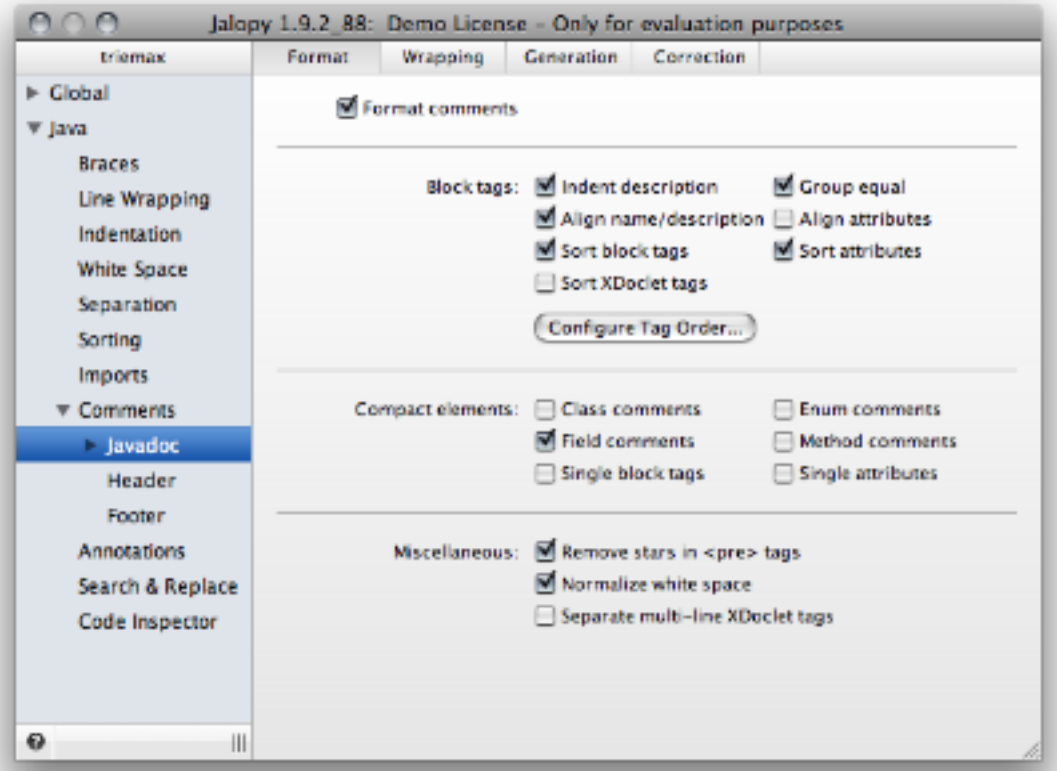
2.8.14 Javadoc

Lets you control all Javadoc-related options. Javadoc is a tool that parses the declarations and documentation comments in a set of Java source files and produces a corresponding set of HTML pages describing the public API or implementation documentation. Jalopy includes functionality to generate and maintain such comments automatically.

2.8.14.1 Format

Lets you control the Javadoc formatting options.

Figure 2.69. Javadoc settings page



Format comments

Enables or disables the formatting of Javadoc comments. When enabled, Javadoc comments are always formatted according to the options specified below.

NOTE The formatting style of leading and closing comment delimiter and leading asterisk characters are determined by analyzing the templates defined for the different declaration elements. Please refer to Section 2.8.14.5, "Templates" for more information about Javadoc templates

Block tags

Lets you control the handling of Javadoc block tags.

Indent description

When enabled, the description of a Javadoc block tag is indented/aligned beyond the tag name or tag parameter upon line wrapping. Otherwise successive description lines will start at the same column offset with the tag.

Since 1.6

Example 2.776. Indented tag descriptions

```
/**
 * This is overridden to return false if the current <code>Icon</code>'s
 * <code>Image</code> is not equal to the passed in <code>Image</code>
 * <code>img</code>.
 *
 * @see  java.awt.image.ImageObserver
 * @see  java.awt.Component#imageUpdate(java.awt.Image, int, int, int,
 *       int, int)
 *
 * @param  img  the <code>Image</code> to be compared
 * @param  infoflags  flags used to repaint the button when the image is
 *                    updated and which determine how much is to be
 *                    painted
 * @param  x  the x coordinate
 */
```

Example 2.777. Unindented tag descriptions

```
/**
 * This is overridden to return false if the current <code>Icon</code>'s
 * <code>Image</code> is not equal to the passed in <code>Image</code>
 * <code>img</code>.
 *
 * @see  java.awt.image.ImageObserver
 * @see  java.awt.Component#imageUpdate(java.awt.Image, int, int, int,
 * int, int)
 *
 * @param  img  the <code>Image</code> to be compared
 * @param  infoflags  flags used to repaint the button when the image is
 * updated and which determine how much is to be painted
 * @param  x  the x coordinate
 */
```

Group equal

When enabled, sections with equal tags are separated by a single blank line.

Since 1.0

Example 2.778. Javadoc without grouped tags

```
/**
 * Returns the next list element that starts with a prefix.
 *
 * @param      prefix      the string to test for a match
 * @param      startIndex  the index for starting the search
 * @param      bias        the search direction, either Position.Bias.
 *                         Forward or Position.Bias.Backward.
 * @return     the index of the next list element that starts with the
 *             prefix; otherwise -1
 * @exception  IllegalArgumentException  if prefix is null or startIndex is
 *                                       out of bounds
 * @since      1.4
 */
```

Example 2.779. Javadoc with grouped tags

```
/**
 * Returns the next list element that starts with a prefix.
 *
 * @param      prefix      the string to test for a match
 * @param      startIndex  the index for starting the search
 * @param      bias        the search direction, either Position.Bias.
 *                         Forward or Position.Bias.Backward.
 *
 * @return     the index of the next list element that starts with the
 *             prefix; otherwise -1
 *
 * @exception  IllegalArgumentException  if prefix is null or startIndex is
 *                                       out of bounds
 *
 * @since      1.4
 */
```

Align name/description

When enabled, the names and/or descriptions of tags are aligned in a table like manner to enhance readability. Otherwise each tag description is indented on its own. You need to have the indentation of descriptions enabled to see descriptions aligned.

Since 1.0

Example 2.780. Javadoc without aligned tags

```
/**
 * Returns the next list element that starts with a prefix.
 *
 * @param  prefix  the string to test for a match
 * @param  startIndex  the index for starting the search
 * @param  bias  the search direction, either Position.Bias.Forward or
 *               Position. Bias.Backward.
 * @return  the index of the next list element that starts with the prefix;
 *          otherwise -1
 * @exception  IllegalArgumentException  if prefix is null or startIndex is
 *                                       out of bounds
 * @since  1.4
 */
```

Example 2.781. Javadoc with aligned tags

```
/**
 * Returns the next list element that starts with a prefix.
 *
 * @param      prefix      the string to test for a match
 * @param      startIndex  the index for starting the search
 * @param      bias        the search direction, either Position.Bias.
 *                         Forward or Position.Bias.Backward.
 * @return     the index of the next list element that starts with the
 *             prefix; otherwise -1
 * @exception  IllegalArgumentException  if prefix is null or startIndex is
 *                                       out of bounds
 * @since      1.4
 */
```

Align attributes

When enabled, all attributes of tags with multiple attributes are indented to the level of the first `'.'` character in the name of the tag. Otherwise all attributes are indented by two spaces.

Since 1.0.1

Example 2.782. Javadoc with standard indented tag attributes

```
/**
 * @ejb.resource-ref
 *   res-auth     = "Container"
 *   res-ref-name = "${kirus.resource.database.name}"
 *   res-type     = "javax.sql.DataSource"
 */
```

Example 2.783. Javadoc with aligned tag attributes

```
/**
 * @ejb.resource-ref
 *      res-auth     = "Container"
 *      res-ref-name = "${kirus.resource.database.name}"
 *      res-type     = "javax.sql.DataSource"
 */
```

Sort block tags

When enabled, block tags are sorted. By default, the order is as recommended by the Javadoc creators (see http://java.sun.com/j2se/javadoc/writingdoccomments/ for more information), but you can configure the order manually yourself (see below). When disabled, block tags are printed in their original order.

Since 1.0

Sort attributes

When enabled, if a block tag contains multiple attributes, these are sorted by name. Otherwise the attributes are printed in their original order.

Since 1.0

Example 2.784. Unsorted XDoclet tag attributes

```
/**
 * Sets the model that represents the contents or “value” of the list and
 * clears the list selection after notifying <code>
 * PropertyChangeListeners</code>.<p> This is a JavaBeans bound property.
 *
 * @param      model  the <code>ListModel</code> that provides the list of
 *                    items for display
 *
 * @beaninfo
 *         bound: true
 *     attribute: visualUpdate true
 *   description: The object that contains the data to be drawn
 */
```

Example 2.785. Sorted XDoclet tags

```
/**
 * Sets the model that represents the contents or “value” of the list and
 * clears the list selection after notifying <code>
 * PropertyChangeListeners</code>.<p> This is a JavaBeans bound property.
 *
 * @param      model  the <code>ListModel</code> that provides the list of
 *                    items for display
 *
 * @beaninfo
 *     attribute: visualUpdate true
 *         bound: true
 *   description: The object that contains the data to be drawn
 */
```

Sort XDoclet tags

When enabled, XDoclet block tags are sorted by name. Otherwise the order is not changed. Please note that this option requires “Sort block tags” to be enabled.

Since 1.9.1

Example 2.786. Unsorted XDoclet tags

```
/**
 * @ejb.home
 *   extends       = "javax.ejb.EJBHome"
 *   local-extends = "javax.ejb.EJBLocalHome"
 * @ejb.interface
 *   extends       = "javax.ejb.EJBObject"
 *   local-extends = "javax.ejb.EJBLocalObject"
 *
 * @weblogic.transaction-descriptor
 *   trans-timeout-seconds = "122"
 * @weblogic.transaction-isolation
 *   TRANSACTION_READ_COMMITTED
 *
 * @ejb.transaction
 *   type = "RequiresNew"
 *
 * @weblogic.enable-call-by-reference
 *   True
 *
 * @ejb.resource-ref
 *   res-ref-name = "jdbc/foo-pool"
 *   res-type     = "javax.sql.DataSource"
 *   res-auth     = "Container"
 */
```

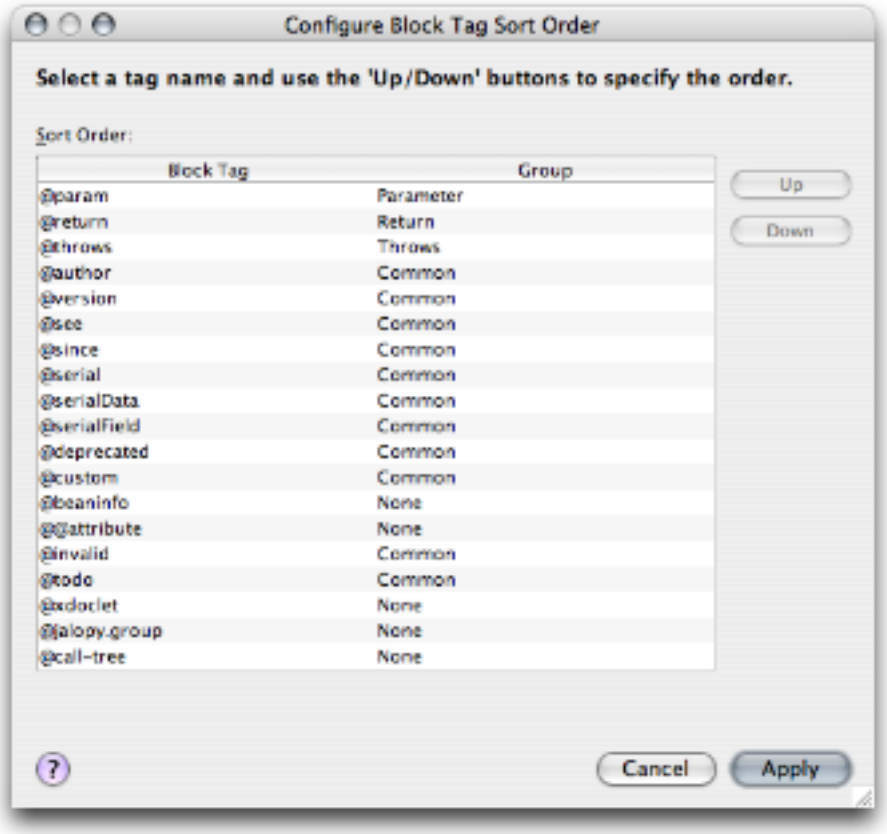
Example 2.787. Sorted XDoclet tags

```
/**
 * @ejb.home
 *   extends       = "javax.ejb.EJBHome"
 *   local-extends = "javax.ejb.EJBLocalHome"
 * @ejb.interface
 *   extends       = "javax.ejb.EJBObject"
 *   local-extends = "javax.ejb.EJBLocalObject"
 * @ejb.resource-ref
 *   res-ref-name = "jdbc/foo-pool"
 *   res-type     = "javax.sql.DataSource"
 *   res-auth     = "Container"
 * @ejb.transaction
 *   type = "RequiresNew"
 *
 * @weblogic.enable-call-by-reference
 *   True
 * @weblogic.transaction-descriptor
 *   trans-timeout-seconds = "122"
 * @weblogic.transaction-isolation
 *   TRANSACTION_READ_COMMITTED
 */
```

Configure Tag Order...

Lets you configure the order of block tags when tag sorting is enabled. Pressing the button will display a configuration dialog that lets you specify both the order of block tags and a grouping section to define what tags should be separated.

Figure 2.70. Configure Javadoc tag order



Select an entry in the list and use the *Up* and *Down* buttons to configure the sort order. To adjust grouping, specify the grouping section each tag should be part of. “None” means

that tags are always separated. Otherwise two consecutive tags are only separated if their grouping section is different.

Since 1.8

Compact elements

Javadoc start and end delimiters are usually printed on its own line. When the comment body does not contain much text, emitting everything in one line might be an easy way to save vertical space. Same with tags and attributes.

Class comments

Lets you specify how Javadoc comments of class, interface and annotation type declarations that fit into one line should be printed.

Since 1.5

Example 2.788. Class Javadoc comment

```
/**
 * A simple container for Foo data.
 */
public class Foo {
}
```

When enabled, Javadoc comments will be printed in a single line, when possible.

Example 2.789. Class Javadoc comment (shortened)

```
/** A simple container for Foo data. */
public class Foo {
}
```

Enum comments

Lets you specify how Javadoc comments of enum declarations that fit into one line should be printed.

Since 1.7

Example 2.790. Enum Javadoc comment

```
/**
 * The foo enumeration.
 */
enum Foo {
    ...
}
```

When enabled, Javadoc comments will be printed in a single line, when possible.

Example 2.791. Enum Javadoc comment (shortened)

```
/** The foo enumeration. */
enum Foo {
    ...
}
```

Field comments

Lets you specify how Javadoc comments of fields or enum constants, that fit into one line should be printed.

Example 2.792. Field Javadoc comment

```
/**
 * What history policy should be used?
 */
private History.Policy _historyPolicy = History.Policy.DISABLED;
```

When enabled, Javadoc comments for fields or enum constants will be printed in a single line, when possible.

Example 2.793. Field Javadoc comment (shortened)

```
/** What history policy should be used? */
private History.Policy _historyPolicy = History.Policy.DISABLED;
```

Method comments

When enabled, Javadoc comments for methods are printed in a single line when possible. This depends not only on the length of the description section, but also on your correction settings. If the auto-correction feature for the tag section is enabled, the comment is only printed in a single line, if the method does contain a void return type and no parameters (because otherwise Javadoc tags are inserted).

Since 1.3

Example 2.794. Methods

```
/**
 * Transfer all entries from src to dest tables
 */
private void transfer(Entry[] src, Entry[] dest) {
    ...
}
```

When enabled, Javadoc comments for methods and constructors will be printed in a single line, when possible.

Example 2.795. Method Javadoc comment (shortened)

```
/** Transfer all entries from src to dest tables */
private void transfer(Entry[] src, Entry[] dest) {
    ...
}
```

Single block tags

Lets you choose whether comments that only consists of a single block tag should be printed in one line when possible. Please note that enabling this option only affects those comments for which compacting has been enabled. E.g. if you want to compact fields with single block tags, you need to enable the "compact field comments" option.

Since 1.8

Example 2.796. Single block tag

```
/**
 * @see com.foo.MyClass
 */
class Demo{
}
```

Example 2.797. Single block tag compacted

```
/** @see com.foo.MyClass */
class Demo{
}
```

Single attributes

Lets you choose whether single attributes should be printed in just one line after the tag name when possible. Otherwise, a line break is printed after the tag name.

Since 1.9.1

Example 2.798. Single XDoclet attribute

```
/**
 * @weblogic.transaction-descriptor
 *   trans-timeout-seconds = "122"
 */
class Demo{
}
```

Example 2.799. Single XDoclet attribute compacted

```
/**
 * @weblogic.transaction-descriptor  trans-timeout-seconds = "122"
 */
class Demo{
}
```

Remove stars in <pre> tags

Lets you remove leading stars in pre-formatted sections. It is often tedious to manually maintain leading stars in front of code snippets enclosed with <pre> tags. They are ignored by Javadoc anyway. With this option you can control whether Jalopy should remove any leading stars in pre-formatted sections or have them printed.

Since 1.6

Example 2.800. Javadoc comment with preformatted section

```
/**
 * <p>As with <code>InputMap</code> if you create a cycle, eg:
 * <pre>
 *   ActionMap am = new ActionMap();
 *   ActionMap bm = new ActionMap():
 *   am.setParent(bm);
 *   bm.setParent(am);
 * </pre>
 * some of the methods will cause a StackOverflowError to be thrown.
 */
```

Example 2.801. Javadoc comment without leading stars in preformatted section

```
/**
 * <p>As with <code>InputMap</code> if you create a cycle, eg:
 * <pre>
     ActionMap am = new ActionMap();
     ActionMap bm = new ActionMap():
     am.setParent(bm);
     bm.setParent(am);
 * </pre>
 * some of the methods will cause a StackOverflowError to be thrown.
 */
```

Normalize white space

When enabled, all white space gaps are reduced to a single blank space (normalized). Otherwise Jalopy will left white space gaps after sentences alone.

Since 1.8

Example 2.802. White space gaps

```
/**
 * This is the first sentence.  This is the second sentence.  This is the
 * third sentence.  This is the forth sentence.  This is the fifth
 * sentence.  Thisssssssssssssssss
 */
```

Example 2.803. Normalized white space

```
/**
 * This is the first sentence. This is the second sentence. This is the
 * third sentence. This is the forth sentence. This is the fifth sentence.
 * Thisssssssssssssssss
 */
```

Separate multi-line XDoclet tags

When enabled, a blank line is printed before and after XDoclet tags that require more than one line.

Since 1.9.1

Example 2.804. Javadoc XDoclet tags

```
/**
 * @ejb.home
 *   extends       = "javax.ejb.EJBHome"
 *   local-extends = "javax.ejb.EJBLocalHome"
 * @ejb.bean
 *   name         = "AboActionManager"
 *   type         = "Stateless"
 *   display-name = "FooManagerBean"
 *   description  = "FooManager EJB"
 *   view-type    = "all"
 *   jndi-name    = "FooMgr"
 * @ejb.interface extends = "javax.ejb.EJBObject"
 */
```

Example 2.805. Separated Javadoc XDoclet tags

```
/**
 * @ejb.home
 *   extends       = "javax.ejb.EJBHome"
 *   local-extends = "javax.ejb.EJBLocalHome"
 *
 * @ejb.bean
 *   name         = "AboActionManager"
 *   type         = "Stateless"
 *   display-name = "FooManagerBean"
 *   description  = "FooManager EJB"
 *   view-type    = "all"
 *   jndi-name    = "FooMgr"
 *
 * @ejb.interface extends = "javax.ejb.EJBObject"
 */
```

2.8.14.2 Line Wrapping

Wrapping

Lets you control the wrapping options for Javadoc comments.

Line length

Lets you define the maximal column width that Javadoc comments are allowed to use. Jalopy tries to keep the comments within this range upon reformatting.

Since 1.0

Disable wrapping for in-line tags

Lets you disable automatic line wrapping for in-line tags. Please note that this means that in-line tags will always print in just one line. If the tag would exceed the maximal line length, a line break is inserted before the tag. But please be aware that the maximal line length could still be exceeded when the tag does not fit in a whole line!

Since 1.5

Example 2.806. Wrapped Javadoc in-line tag

```
/**                                                                 |
 * This is overridden to return false if the {@link java.awt.Icon |
 * Icon's} Image is not equal to the passed in Image.             |
 */                                                                |
```

Example 2.807. Javadoc in-line tag (wrapping disabled)

```
/**                                                                 |
 * This is overridden to return false if the                       |
 * {@link java.awt.Icon Icon's} Image is not equal to the passed  |
 * in Image.                                                       |
 */                                                                |
```

Misc

Lets you control miscellaneous Javadoc settings.

Inner spacing

Lets you define the amount of white space that gets printed between block tags and their description text.

Since 1.0

Example 2.808. One space inner spacing

```
/**
 * Returns the next list element that starts with a prefix.
 *
 * @param     prefix     the string to test for a match
 * @param     startIndex the index for starting the search
 * @param     bias       the search direction, either Position.Bias. Forward
 *                       or Position. Bias.Backward.
 *
 * @return    the index of the next list element that starts with the prefix;
 *            otherwise -1
 *
 * @exception IllegalArgumentException if prefix is null or startIndex is out
 *            of bounds
 *           ^
                         ^
 * @since     1.4
 */
```

Example 2.809. Two spaces inner spacing

```
/**
 * Returns the next list element that starts with a prefix.
 *
 * @param      prefix      the string to test for a match
 * @param      startIndex  the index for starting the search
 * @param      bias        the search direction, either Position.Bias.
 *                         Forward or Position.Bias.Backward.
 * @return     the index of the next list element that starts with the
 *             prefix; otherwise -1
 * @exception  IllegalArgumentException  if prefix is null or startIndex is
 *             out of bounds
 *           ^^
                         ^^
 * @since      1.4
 */
```

Indent HTML tags

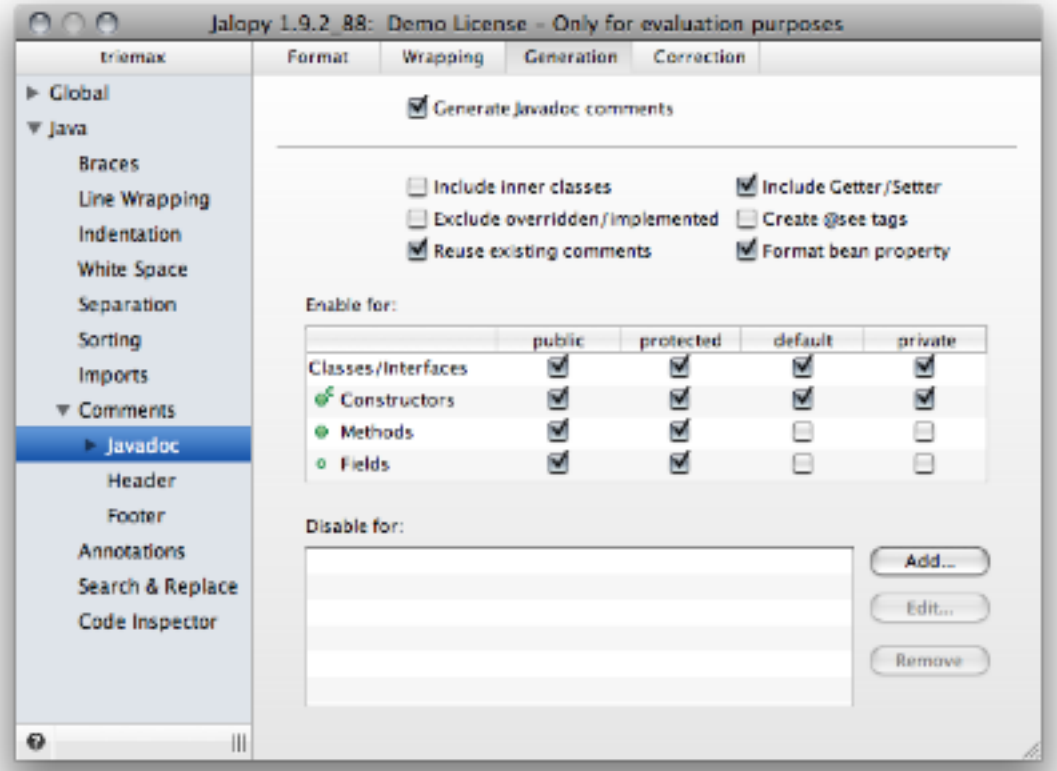
Enables the indentation of most HTML block tags (like lists, tables and the like). Please note that the HTML contents have to be well-formed for this feature to work! By default, Jalopy will inform you about invalid HTML when this feature was enabled. Another choice might be to enable the "Check HTML tags" feature to automatically ensure well-formed HTML (among other things, see below).

Since 1.0

2.8.14.3 Generation

Controls the general handling of Javadoc comments.

Figure 2.71. Javadoc settings page



Please refer to Section 2.8.14.5, “Templates” for information about how to define templates that are used for Javadoc comment generation.

Generate Javadoc comments

Enables or disables the comment auto-generation as a whole.

Since 1.2.1

Include inner classes

Enables comment auto-generation for inner classes, too. Auto-generation does not apply to anonymous inner classes.

Include Getter/Setter

Controls whether the auto-generation should be enabled for methods that follow the JavaBeans naming convention (Getter/Setter). Please note that you can control what methods should be recognized as Boolean Getters via a regular expression. Refer to “Boolean Getter Regex” for more information.

Since 1.3

Exclude Overridden/Implemented

Controls whether Javadoc comments should be generated for methods that are implementing/overriding others.

Example 2.810. Class hierarchy

```
public interface Foo {
    /**
     * Does foo.
     */
    public void doFoo();
}

public abstract class BaseFoo implements Foo {
    public void doFoo() {
        /* ... */
    }
}

public abstract class FooImpl extends BaseFoo {
    @Override public void doFoo() {
        /* ... */
    }
}
```

With this option enabled, Jalopy would not generate Javadoc for `doFoo()` in `BaseFoo` or `FooImpl`, because they implement or override another method.

Since 1.8

Generate @see tags

Controls whether Javadoc generation for methods that are implementing or overriding others, creates @see tags to point to the referenced method. When disabled, Javadoc generation uses the descriptions given in the different templates.

Example 2.811. Generated Javadocs

```
package com.foo;


public interface Foo {
    /**
     * Does foo.
     */
    public void doFoo();
}

public abstract class BaseFoo implements Foo {

    /**
     * DOCME!
     */
    public void doFoo() {
        ...
    }

    /**
     * DOCME!
     */
    public void doBar() {
        ...
    }
}

public abstract class FooImpl extends BaseFoo {

    /**
     * DOCME!
     */
    @Override public void doFoo() {
        ...
    }
}
```

But when this option has been enabled, Jalopy automatically creates @see tags that references the overridden or implemented method in order to point to the documentation available there.

Example 2.812. Generated Javadocs

```
package com.foo;


public interface Foo {
    /**
     * Does foo.
     */
    public void doFoo();
}

public abstract class BaseFoo implements Foo {

    /**
     * @see com.foo.Foo#doFoo()
     */
    public void doFoo() {
        ...
    }

    /**
     * DOCME!
     */
    public void doBar() {
        ...
    }
}

public abstract class FooImpl extends BaseFoo {

    /**
     * @see com.foo.Foo$doFoo()
     */
    @Override public void doFoo() {
        ...
    }
}
```

Since 1.8

Reuse existing comments

When enabled, the textual contents of all existing comments that appear before a certain node are used as the description section (instead of the one defined in the template). Otherwise the generated Javadoc comment is simply inserted below any already existing comments.

Since 1.5

Example 2.813. Method without Javadoc comment

```
/*
 * HTML is the top level element
 */
public static Node parseDocument(Lexer lexer) {}
```

When the option is left disabled, the above example would become

Example 2.814. Generated comment

```
/*
 * HTML is the top level element
 */
/**
 * DOCME!
 *
 * @param   lexer  DOCME!
 *
 * @return  DOCME!
 */
public static Node parseDocument(Lexer lexer) {}
```

But when the option is enabled, the result would be

Example 2.815. Generated comment that uses existing comment

```
/**
 * HTML is the top level element
 *
 * @param   lexer  DOCME!
 *
 * @return  DOCME!
 */
public static Node parseDocument(Lexer lexer) {}
```

Format bean property

Lets you control whether the value of the "property.name" local environment variable should be split into several chunks or just have its prefix stripped upon interpolation.

When enabled, the property name is determined from the method name by stripping the bean prefix and taking any upper case letter followed by a lower case letter and put a space in front of it. E.g. “getImportanceValue” would result in "Importance Value", “getABSValue” in "ABS Value" and “isValid” in “Valid”. The case of all chunks is adjusted according to the rules as sketched below.

> "8.8 Capitalization of inferred names.
>
> When we use design patterns to infer a property or event name, we need to decide what rules to follow for capitalizing the inferred name. If we extract the name from the middle of a normal mixedCase style Java name then the name will, by default, begin with a capital letter. Java programmers are accustomed to having normal identifiers start with lower case letters. Vigorous reviewer input has convinced us that we should follow this same conventional rule for property and event names.
>
> Thus when we extract a property or event name from the middle of an existing Java name, we normally convert the first character to lower case. However to support the occasional use of all upper-case names, we check if the first two characters of the name are both upper case and if so leave it alone. So for example,
>
> “FooBah” becomes “fooBah”
> “Z” becomes “z”
> “URL” becomes “URL”
>
> [taken from the JavaBeans Spec]

When disabled, just the bean prefix is stripped and the case of the first letter adjusted according to the rules laid out in the JavaBean spec.

Since 1.3

Example 2.816. Javadoc template for Setter

```
/**
 * Sets the value of the $property.name$ property.
 *
 * @param name $property.name$ property value.
 */
public void setImportanceValue(String value) {
    ...
}
```

Example 2.817. Generated Javadoc comment with formatted JavaBeans property name

```
/**
 * Sets the value of the Importance Value property.
 *
 * @param  name  importance value property value.
 */
public void setImportanceValue(String value) {
    ...
}
```

Example 2.818. Generated Javadoc comment with unformatted JavaBeans property name

```
/**
 * Sets the value of the ImportanceValue property.
 *
 * @param  name  importanceValue property value.
 */
public void setImportanceValue(String value) {
    ...
}
```

Enable for

The table component lets you selectively enable the auto-generation of missing Javadoc comments for specific code elements and access levels. Please note that you can selectively disable Javadoc generation in source code files by using a special pragma comment. Refer to the Pragma comments section for more information.

Disable for

Lets you disable Javadoc generation within classes that extend a certain class or implement a certain interface. This option is useful if you generally want to document inner classes, but obmit documentation for certain inner classes like e.g. Swing listeners. Use the *Add...* button to define the type names of `extends` or `implements` clauses that should disable Javadoc generation when found.

Please note that Javadoc generation will only be disabled for the methods, fields etc. defined within a class - the class declaration itself is not affected. Please note further that comparison is done using exact string matching: you need to specify the type name exactly as it appears in the source file. `ActionListener` and `java.awt.event.ActionListener` are treated as two different type names. If you mix qualified and simple type names in your source, you need to define both type names here.

Since 1.6

Example 2.819. Java source file with missing Javadoc

```
/**
 * A sample class.
 */
public class Foo {

    class MyAction implements ActionListener {

        public void actionPerformed(ActionEvent ev) {
            ...
        }
    }
}
```

Javadoc generation without exclusions could look like:

Example 2.820. Javadoc generation without exclusion

```
/**
 * A sample class.
 */
public class Foo {

    /**
     * DOCME!
     *
     * @author   Joe Tiger
     * @version  $version$
     */
    class MyAction implements ActionListener {

        /**
         * DOCME!
         *
         * @param  ev  DOCME!
         */
        public void actionPerformed(ActionEvent ev) {
            ...
        }
    }
}
```

But with Javadoc generation disabled for `ActionListener`, the result could look like:

Example 2.821. Javadoc generation with exclusion

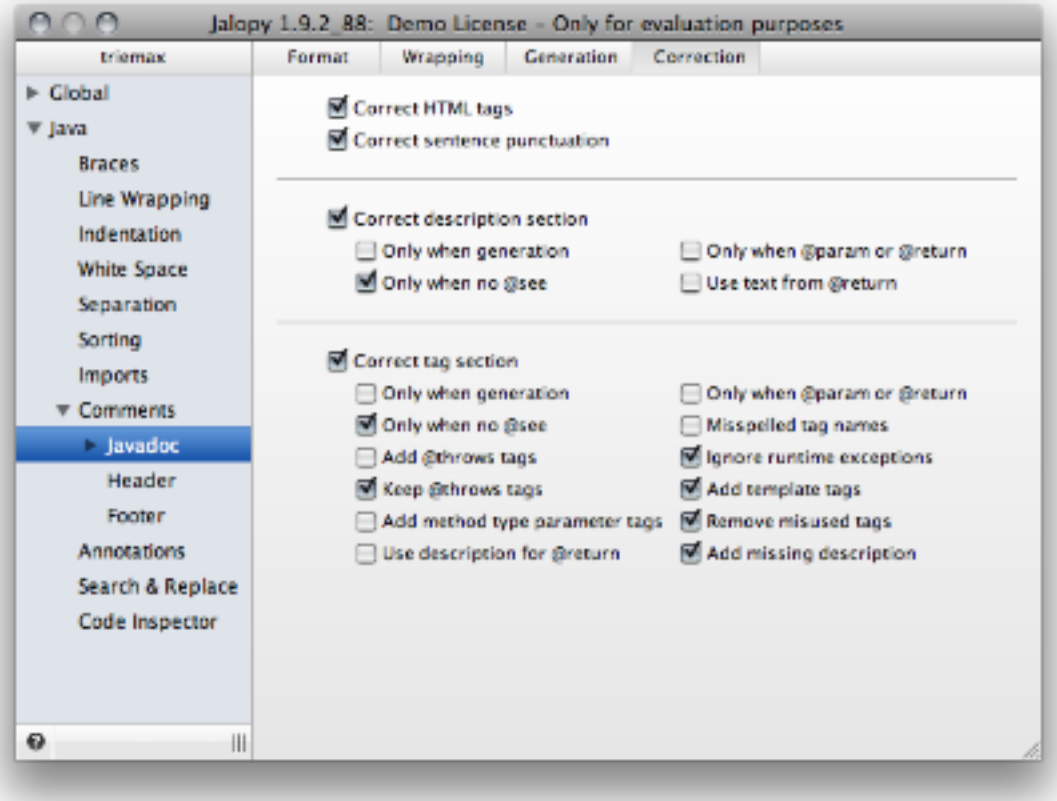
```
/**
 * A sample class.
 */
public class Foo {

    /**
     * DOCME!
     *
     * @author   Joe Tiger
     * @version  $version$
     */
    class MyAction implements ActionListener {

        public void actionPerformed(ActionEvent ev) {
            ...
        }
    }
}
```

Please note how Javadoc is added for the inner class declaration but omitted for the contained method!

2.8.14.4 Correction

Lets you control the Javadoc comment checking and auto-correction features.

Figure 2.72. Javadoc Correction settings page



Please note that unlike with the Open Source version, auto-correction may be enabled independently from Javadoc formatting. Still, it is recommended to enable the formatting of comments too, because otherwise you may encounter slight style differences when descriptions or tags are inserted/removed.

Correct HTML

This feature lets you enforce valid HTML. When enabled, Jalopy ensures that the comments only contain valid HTML 4.01 markup. Missing tags like optional end tags will be inserted to ensure well-formed contents.

Example 2.822. Javadoc comment with missing paragraph tags

```
/**
 * Indicates the kinds of program element to which an annotation type
 * is applicable.  If a Target meta-annotation is not present on an
 * annotation type declaration, the declared type may be used on any
 * program element.  If such a meta-annotation is present, the compiler
 * will enforce the specified usage restriction.
 *
 * For example, this meta-annotation indicates that the declared type is
 * itself a meta-annotation type.  It can only be used on annotation type
 * declarations.
 */
```

Example 2.823. Javadoc comment with inserted paragraph tags

```
/**
 * Indicates the kinds of program element to which an annotation type
 * is applicable.  If a Target meta-annotation is not present on an
 * annotation type declaration, the declared type may be used on any
 * program element.  If such a meta-annotation is present, the compiler
 * will enforce the specified usage restriction.
 *
 * <p>For example, this meta-annotation indicates that the declared type
 * is itself a meta-annotation type.  It can only be used on annotation
 * type declarations.</p>
 */
```

Example 2.824. Javadoc comment with missing tags

```
/**
 * Returns a short description of this throwable.
 * If this <code>Throwable</code> object was created with a non-null detail
 * message string, then the result is the concatenation of three strings:
 * <ul>
 * <li> The name of the actual class of this object
 * <li>": " (a colon and a space)
 * <li>The result of the {@link #getMessage} method for this object
 * </ul>
 */
```

Example 2.825. Javadoc comment with inserted tags

```
/**
 * Returns a short description of this throwable. If this <code>
 * Throwable</code> object was created with a non-null detail message
 * string, then the result is the concatenation of three strings:
 *
 * <ul>
 *   <li>The name of the actual class of this object</li>
 *   <li>": " (a colon and a space)</li>
 *   <li>The result of the {@link #getMessage} method for this object</li>
 * </ul>
 */
```

Since 1.0

Correct sentence punctuation

When enabled, ensures that the first sentence of the description section ends with punctuation. If no punctuation is present, a dot character will be added at the end of the first sentence. The first sentence is determined by either a blank line between two text chunks or by a HTML block tag. If no obvious sentence break could be found, the dot is added at the end of the description section.

Since 1.6

Example 2.826. Javadoc comment with missing period after first sentence

```
/**
 * The method used for creating the tree
 * <p>
 * This method adds an anonymous TreeSelectionListener to
 * the returned JTree.  Upon receiving TreeSelectionEvents,
 * this listener calls refresh with the selected node as a
 * parameter.
 */
```

Example 2.827. Reformatted Javadoc comment with added period after first sentence

```
/**
 * The method used for creating the tree.

 * <p>This method adds an anonymous TreeSelectionListener to the
 * returned JTree. Upon receiving TreeSelectionEvents, this listener
 * calls refresh with the selected node as a parameter.
 */
```

Description Section

Provides option to control the behavior of the description section of a Javadoc comments. The description begins after the starting delimiter `/**` and continues until the tag section. The tag section starts with the first block tag, which is defined by the first `@` character that begins a line (ignoring leading asterisks, white space and comment separator). The main description cannot continue after the tag section begins.

```
/**
 * This sentence would hold the main description for this doc comment.
 * @see java.lang.Object
 */
```

Correct description section

When enabled, Jalopy inserts a missing description into existing Javadoc comments. Unlike specified otherwise (see "Use text from @return" below), the description is taken from the template for the code element that contains the Javadoc comment. Refer to Section 2.8.14.5, "Templates" for information on how to customize the templates.

Since 1.2.1

Only when generation

Only inserts the missing description when Javadoc auto-generation has been enabled for the declaration element that contains the Javadoc comment.

Since 1.2.1

Example 2.828. Insert description when auto-generation enabled

```
/**
 * @jalopy.group accessor
 */
protected int getFoo () {
    ...
}
```

will only be formatted to

```
/**
 * DOCME!
 *
 * @jalopy.group accessor
 */
protected int getFoo () {
    ...
}
```

when Javadoc comment auto-generation is enabled for method declarations that have an access level of `protected`.

Only when @param or @return

Only inserts the missing description when a `@param` or `@return` block tag can be found in the Javadoc comment.

Since 1.2.1

Example 2.829. Insert description only when @param or @return tag present

With this option enabled, code like

```
/**
 * @jalopy.group accessor
 */
protected int getFoo () {
    ...
}
```

will be formatted as

```
/**
 * @jalopy.group  accessor
 */
protected int getFoo () {
    ...
}
```

because the Javadoc comment neither contains a `@param` nor a `@return` block tag. But

```
/**
 * @return        returns the foo property.
 * @jalopy.group  accessor
 */
protected int getFoo () {
    ...
}
```

will be formatted as

```
/**
 * DOCME!
 *
 * @return        returns the foo property.
 *
 * @jalopy.group  accessor
 */
protected int getFoo () {
    ...
}
```

because a `@return` tag can be found.

Only when no @see

Only inserts the missing description when no `@see` block tag can be found in the Javadoc comment. The default behavior is to disable the insertion of a missing description if the comment only consists of a single `@see` block tag or starts with an `{@inheritDoc}` in-line tag. In order to avoid adding information that is redundant, one may enable this switch when `@see` tags are used to point to related documentation.

Since 1.2.1

Example 2.830. Insert description only when no @see tag present

```
/**
 * @jalopy.group accessor
 */
protected int getFoo () {
    ...
}
```

would be formatted as

```
/**
 * DOCME!
 *
 * @jalopy.group  accessor
 */
protected int getFoo () {
    ...
}
```

because the Javadoc comment contains no `@see` tag.

But

```
/**
 * @jalopy.group  accessor
 * @see           #com.foo.OtherClass
 */
protected int getFoo () {
    ...
}
```

would be formatted as

```
/**
 * @jalopy.group  accessor
 * @see           #com.foo.OtherClass
 */
protected int getFoo () {
    ...
}
```

because a `@see` tag can be found.

Use text from @return

When enabled, the description text of the @return tag is used (when present) for a missing description. The first letter of the text will upper-cased.

Since 1.5

Example 2.831. Javadoc with missing description section

```
/**
 * @return  returns the result of the operation.
 */
public Object getResult() {
    ...
}
```

Example 2.832. Missing description generated from template

```
/**
 * TODO: DOCME!
 *
 * @return  returns the result of the operation.
 */
public Object getResult() {
    ...
}
```

Example 2.833. Missing description generated from @return tag

```
/**
 * Returns the result of the operation.
 *
 * @return  returns the result of the operation.
 */
public Object getResult() {
    ...
}
```

Tag Section

Provides options to control the behavior for the block tags of Javadoc comments. The description begins after the starting delimiter `/**` and continues until the tag section. The tag section starts with the first block tag, which is defined by the first `@` character that begins a line (ignoring leading asterisks, white space and comment separator). The main description cannot continue after the tag section begins. There can be any number of tags - some types of tags can be repeated while others cannot. In the following example, the `@see` tags starts the tag section:

```
/**
 * This sentence would hold the main description for this doc comment.
 * @see java.lang.Object
 */
```

Correct tag section

When enabled, missing Javadoc block tags will be inserted, obsolete tags can be removed. Spelling errors of block and in-line tags are corrected. The description of a tag is taken from the template for the code element that contains the Javadoc comment. Refer to Section 2.8.14.5, “Templates” for information on how to customize the templates.

Only when generation

Only corrects tags when Javadoc auto-generation has been enabled for the declaration element that contains the Javadoc comment.

Since 1.2.1

Example 2.834. Only correct tags when auto-generation enabled

```
/**
 * @jalopy.group accessor
 */
protected int getFoo () {
    ...
}
```

will only be formatted to

```
/**
 * @return       DOCME!
 *
 * @jalopy.group accessor
 */
protected int getFoo () {
    ...
}
```

when Javadoc comment auto-generation is enabled for method declarations that have an access level of `protected`.

Only when @param or @return

Only corrects tags when a `@param` or `@return` block tag can be found in the Javadoc comment.

Since 1.2.1

Example 2.835. Only correct tags when @param or @return tag present

With this option enabled, code like

```
/**
 * @jalopy.group  accessor
 */
protected int getFoo (int param) {
    ...
}
```

will be formatted as

```
/**
 * @jalopy.group  accessor
 */
protected int getFoo (int param) {
    ...
}
```

because the Javadoc comment neither contains a `@param` nor a `@return` block tag. But

```
/**
 * @param         param  a parameter
 * @jalopy.group  accessor
 */
protected int getFoo (int param) {
    ...
}
```

will be formatted as

```
/**
 * @param         param  a parameter
 *
 * @return        returns  the foo property.
 *
 * @jalopy.group  accessor
 */
protected int getFoo (int param) {
    ...
}
```

because a `@param` tag can be found.

Only when no @see

Disables the auto-correction when a `@see` tag is found in the comment. The default behavior is to disable auto-correction if the comment only consists of a single `@see` block tag or starts with an `{@inheritDoc}` in-line tag. In order to avoid adding information that is redundant, one may enable this switch when `@see` tags are used to point to related documentation.

Since 1.0.1

Misspelled tag names

When enabled, misspelled Javadoc tag names will be corrected when possible. When Jalopy encounters an invalid tag name, i.e. the name is not part of the list with valid tag names, it determines whether the tag name is vastly similar with one on the list. If so, Jalopy will pick the one from the list otherwise it reports an error. For information about the build-in list with valid Javadoc tag names refer to Section 2.8.14.6.1.1, "Block tags", Section 2.8.14.6.1.2, "In-line tags" and Appendix B, *Build-in XDoclet tags*.

Since 1.3

Add @throws tags

When enabled, performs an additional check for exceptions that are actually thrown from within a constructor or method body, but not documented and adds block tags. E.g. if a method only declares to throw an `IOException`, but actually throws a `FileNotFoundException`, and this `FileNotFoundException` has not been documented with a @throws tag, it will be added.

Example 2.836. Undocumented exception

```
/**
 * Description
 *
 * @param   rFile  input file.
 *
 * @throws  IOException if an I/O problem occurred.
 */
public void sample(File rFile) throws IOException {
    if (rFile.exists())
        throw new FileNotFoundException();

    ...
}
```

Example 2.837. Added exception

```
/**
 * Description
 *
 * @param   rFile  input file.
 *
 * @throws  IOException            if an I/O problem occurred.
 * @throws  FileNotFoundException  DOCUMENT ME!
 */
public void sample(File rFile) {
    if (rFile.exists())
        throw new FileNotFoundException();

    ...
}
```

Ignore runtime exceptions

When enabled, no tags will be added or removed for runtime exceptions and errors that are thrown from within a method or constructor body. Please note that enabling this option will cause present `@throws` tags that document runtime exceptions or errors to be removed! If you want to keep existing tags, please enable the "Keep @throws tags" option as well.

You have to explicitly enable type repository services for the Ant, Console and Maven Plug-ins to activate this feature! Please refer to the documentation of the individual Plug-ins to learn how one can accomplish this (see Part II, “Plug-ins”).

Since 1.0

Example 2.838. Generated missing throws clause

```
/**
 * Description
 *
 * @throws  IllegalArgumentException  DOCME!
 */
public void isNewline(int offset) {
    if (input <= 1) throw new IllegalArgumentException();
    ..
}
```

Example 2.839. No throws clause generated for runtime exception

```
/**
 * Description
 */
public void isNewline(int offset) {
    if (input <= 1) throw new IllegalArgumentException();
    ..
}
```

Keep @throws tags

When enabled, no existing @throws tags are removed from comments. This feature proves useful if you have comments with existing @throws tags for runtime exceptions that are not actually thrown from within a method body.

Since 1.0

Add template tags

When enabled, tags that are defined in the Javadoc template but missing in the Javadoc comment of the corresponding declaration node, are inserted. Missing tags are only inserted

when their declaration has its Javadoc generation option enabled for the current scope. This holds true if even when the Javadoc comment generation is disabled globally in order to allow fine grained control when and for what declarations missing tags should be inserted. Please refer to Enable Javadoc generation for for information on how to enable Javadoc generation for specific declarations. For information on how to customize the Javadoc templates, please refer to Section 2.8.14.5, "Templates".

Since 1.5

Example 2.840. Javadoc template

```
/**
 * TODO: DOCME!
 *
 * @author $user.name$
 */
```

With the above shown template, upon reformatting Jalopy ensures that all existing class level Javadoc comments contain the `@author` tag. Thus, the following comment

Example 2.841. Javadoc comment

```
/**
 * Encapsulate an attribute declaration.
 */
class AttributeDecl {
}
```

could become

Example 2.842. Javadoc comment after formatting

```
/**
 * Encapsulate an attribute declaration.
 *
 * @author  John Doo
 */
class AttributeDecl {
}
```

Note how the `$user.name` variable expression is interpolated during formatting! Environment variables are discussed in Section 2.4, "Environment". Missing tags are added at the end of the tag section in the order they are defined in the template. We recommend to enable Tag sorting in order to ensure a specific ordering.

Add method type parameter tags

When enabled, Jalopy enforces/corrects `@param` tags for generic method and constructor declarations.

Example 2.843. Generic method

```
/**
 * blah.
 *
 * @param  string  blah
 *
 * @return  blah
 */
<T, V extends T> V convert(String string);
```

will have tags inserted for the type parameters.

Since 1.6

Example 2.844. Generic method

```
/**
 * blah.
 *
 * @param  <T>  blah
 * @param  <V>  blah
 * @param  string     blah
 *
 * @return  blah
 */
<T, V extends T> V convert(String string);
```

Please note that existing `@param` tags documenting type parameters will be removed when this option is disabled!

Remove misused tags

When enabled, the validity of block tags will be checked. Not all tags can be used in all contexts. Tags that are invalid will be removed. If left disabled, Jalopy only prints warnings about misused tags.

Since 1.0

Use description for @return

When enabled, the text for a missing @return tag description is not taken from the template, but the first sentence of description section is taken (when present). The first letter of the description will be lower-cased.

Since 1.5

Example 2.845. Javadoc with missing @return tag description

```
/**
 * Returns the result of the calculation.
 */
public Object getResult() {
   ...
}
```

Example 2.846. Javadoc with @return tag description inserted from template

```
/**
 * Returns the result of the calculation.
 *
 * @return  DOCME!
 */
public Object getResult() {
   ...
}
```

Example 2.847. Javadoc with @return tag description taken from description section

```
/**
 * Returns the result of the calculation.
 *
 * @return returns the result of the calculation.
 */
public Object getResult() {
   ...
}
```

Add missing description

When enabled, missing descriptions of certain Javadoc block tags will be tagged with a marker. The tag marker is inserted for the following tags: `@author`, `@deprecated`, `@exception`, `@param`, `@return`, `@see`, `@serialData`, `@serialField`, `@since`, `@throws`, `@version`. The marker text is taken from the description section of corresponding template. XDoclet or custom tags will remain untouched.

Since 1.5

Example 2.848. Tags with missing description

```
/*
 * HTML is the top level element
 *
 * @param lexer
 *
 * @return
 */
public static Node parseDocument(Lexer lexer) {}
```

When the option is left disabled, the above example would become

Example 2.849. Tags with missing description

```
/**
 * HTML is the top level element
 *
 * @param   lexer
 *
 * @return
 */
public static Node parseDocument(Lexer lexer) {}
```

But when the option is enabled, the result would be

Example 2.850. Tagged missing descriptions

```
/**
 * HTML is the top level element
 *
 * @param   lexer  DOCME!
 *
 * @return  DOCME!
 */
public static Node parseDocument(Lexer lexer) {}
```

2.8.14.5 Templates

Lets you define templates to be inserted for the different declaration elements when Javadoc Generation (see Section 2.8.14.3, "Generation") has been enabled. Each element (Class, Interface, Field, Constructor and Method) has its own template. Depending on the element type, a template consists of up to five parts that together form a valid Javadoc comment. When Javadoc formatting (see Section 2.8.14.1, "Format") is enabled, the templates will be reformatted before they are inserted.

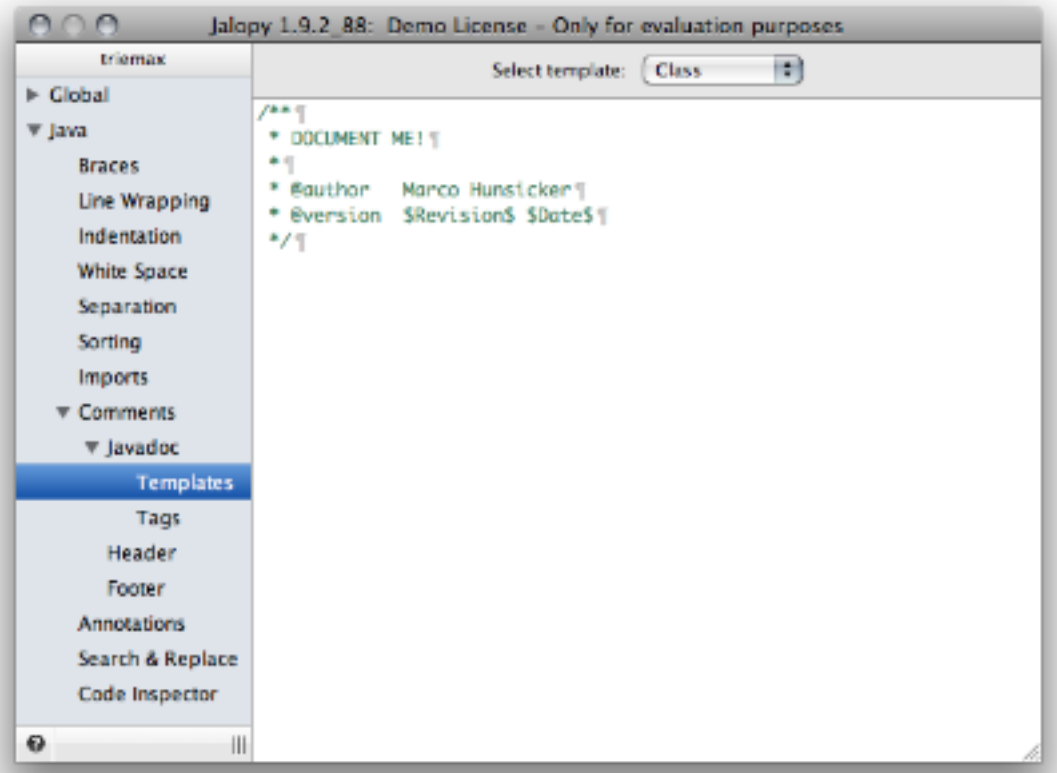
You can use variable expressions throughout your templates to insert various data automagically. See Section 2.4.3, "Local variables" for more information about the available variables.

IMPORTANT The templates also describe the formatting style for each element and are used to determine the description texts that are to be inserted for the Javadoc auto-correction feature (described in Section 2.8.14.4, “Correction”).

Class template

Lets you define the template for class and enum declarations (including inner classes).

Figure 2.73. Javadoc class template



Enter a valid Javadoc comment. The Preview window will update in real-time to reflect your changes.

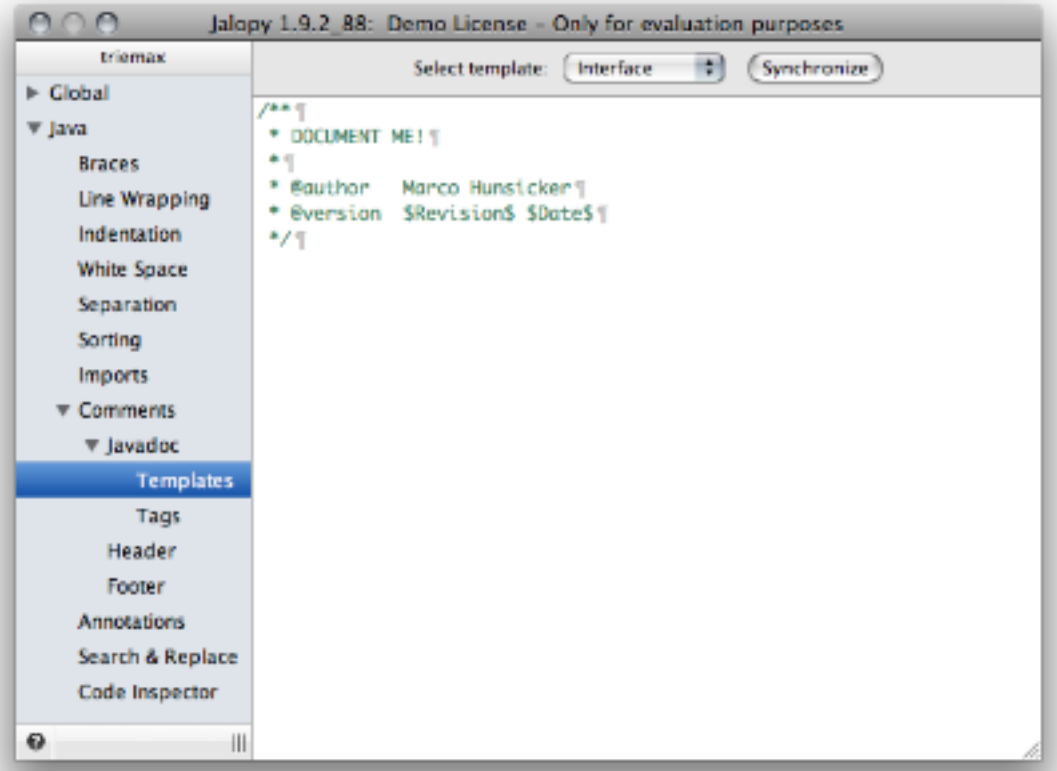
Example 2.851. Class declaration with generated Javadoc

```
/**
 * DOCME!
 *
 * @author $author$
 * @version $Revision: #22 $, $Date: 2007/08/15 $
 */
public class CompilationUnit {
}
```

Interface template

Lets you define the template for interface and annotation declarations.

Figure 2.74. Javadoc interface template



Enter a valid Javadoc comment. The Preview window will update in real-time to reflect your changes.

Example 2.852. Interface declaration with generated Javadoc

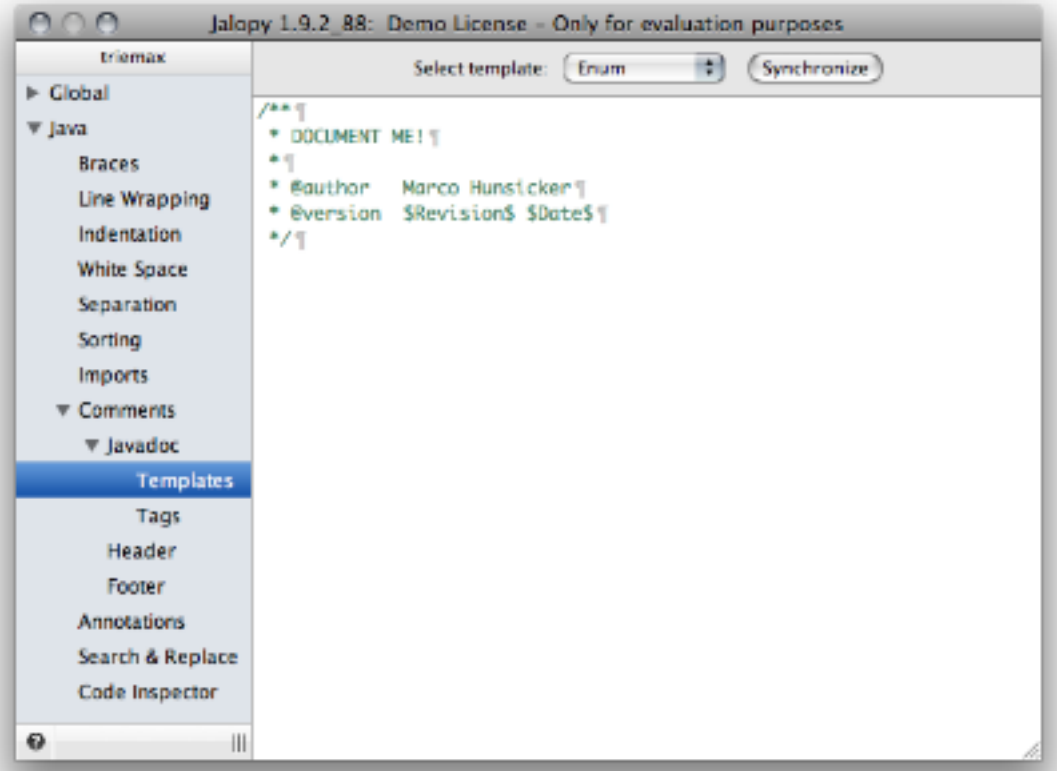
```
/**
 * DOCME!
 *
 * @author $author$
 * @version $Revision: #22 $, $Date: 2007/08/15 $
 */
public interface Saveable {
}
```

Enum template

Lets you define the template for enum declarations (includes inner enums).

Since 1.5

Figure 2.75. Javadoc enum template



Enter a valid Javadoc comment. The Preview window will update in real-time to reflect your changes.

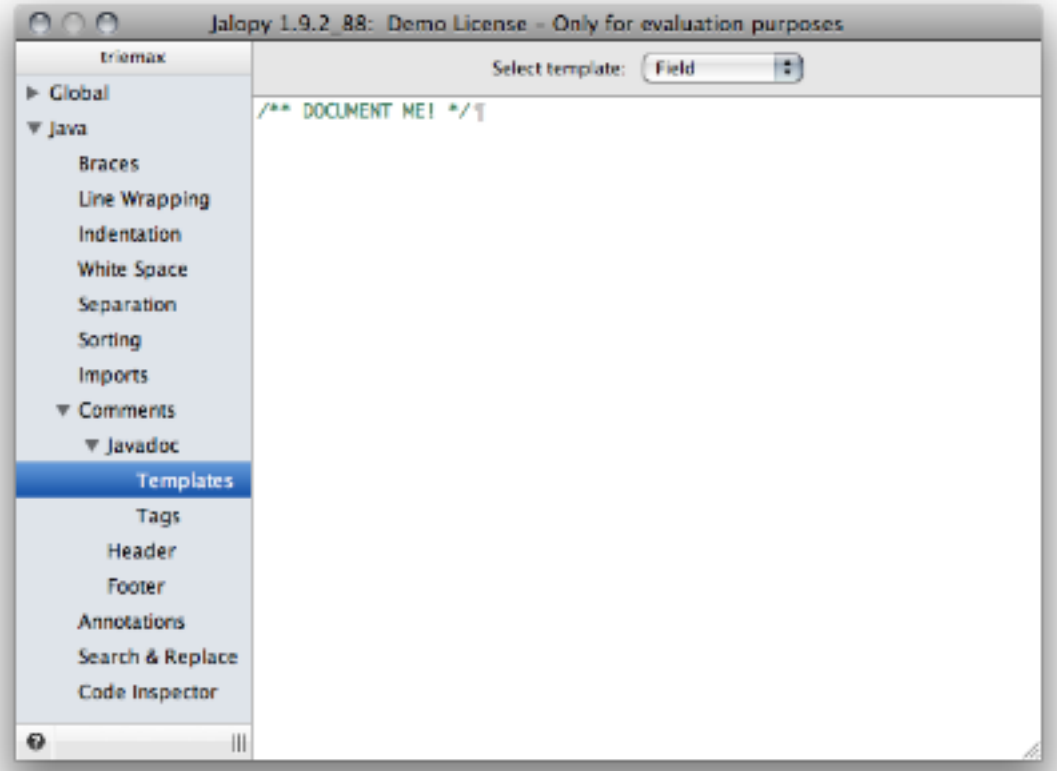
Example 2.853. Enum declaration with generated Javadoc

```
/**
 * DOCME!
 *
 * @author $author$
 * @version $Revision: #22 $, $Date: 2007/08/15 $
 */
public enum Week {
  ...
}
```

Field template

Lets you define the template for field declarations.

Figure 2.76. Javadoc field template



Enter a valid Javadoc comment. The Preview window will update in real-time to reflect your changes.

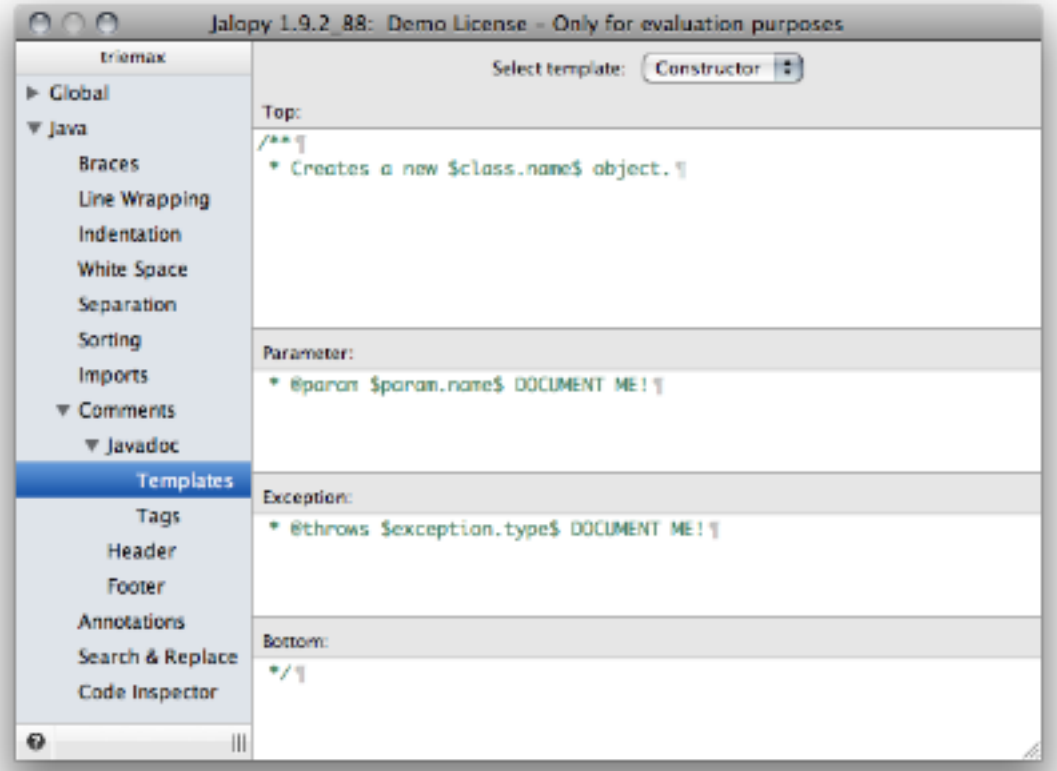
Example 2.854. Field declaration with generated Javadoc

```
/** DOCME! */
public String name;
```

Constructor template

Lets you define the template for constructor declarations.

Figure 2.77. Javadoc constructor template



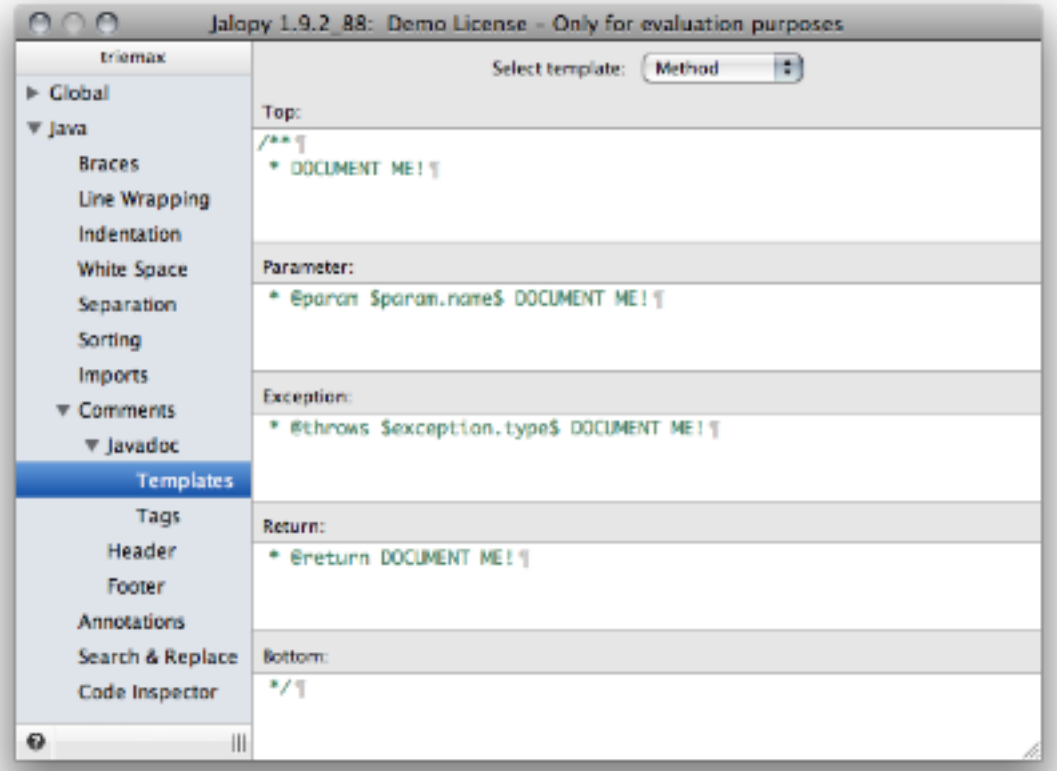
Example 2.855. Constructor declaration with generated Javadoc

```
/**
 * DOCME!
 *
 * @param source DOCME!
 */
public CompilationUnit(String source) {
}
```

Method template

Lets you define the template for method declarations.

Figure 2.78. Javadoc method template



The Preview window will update in real-time to reflect your changes.

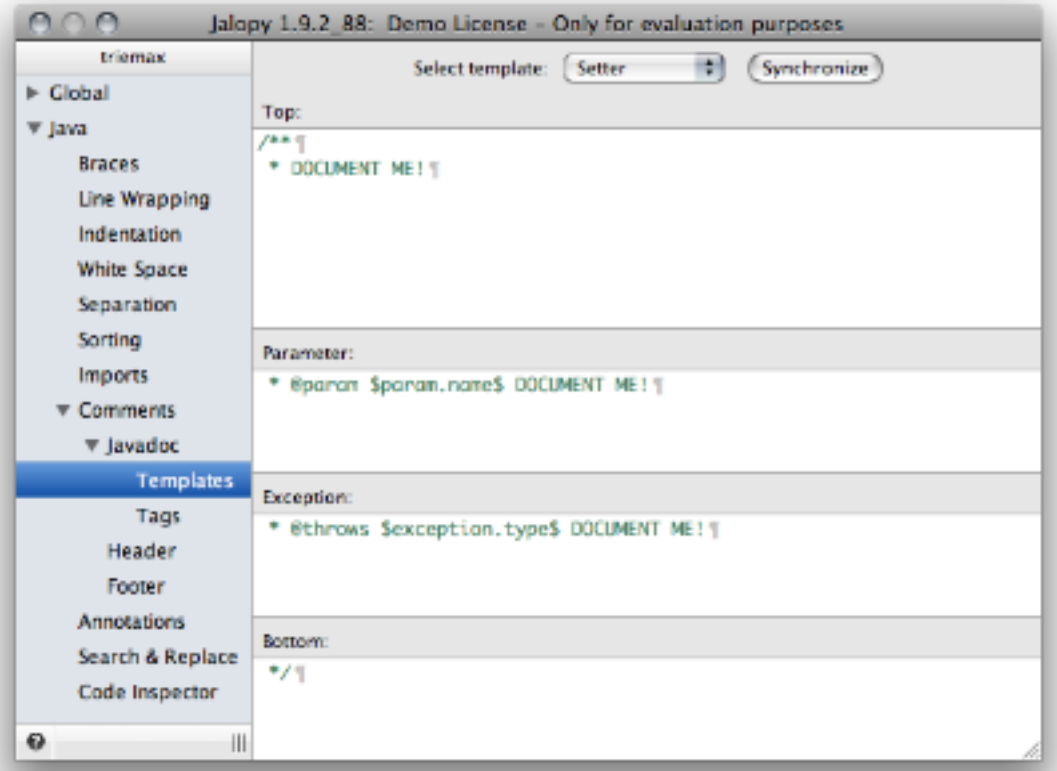
Example 2.856. Method declaration with generated Javadoc

```
/**
 * DOCME!
 *
 * @param source DOCME!
 *
 * @throws SyntaxException DOCME!
 */
public void compile(File source) throws SyntaxException {
}
```

Setter template

Lets you define the template for Setter methods (following the JavaBeans naming convention).

Figure 2.79. Javadoc Setter method template



The JavaBeans specification defines a standard way in which the properties for a JavaBean instance should be accessed. This same technique can also be applied to regular classes and interfaces to access their attributes. The Setter template is used for all methods that look like `setXxx()` (also called mutator methods).

Since 1.1

Example 2.857. Setter method declaration with generated Javadoc

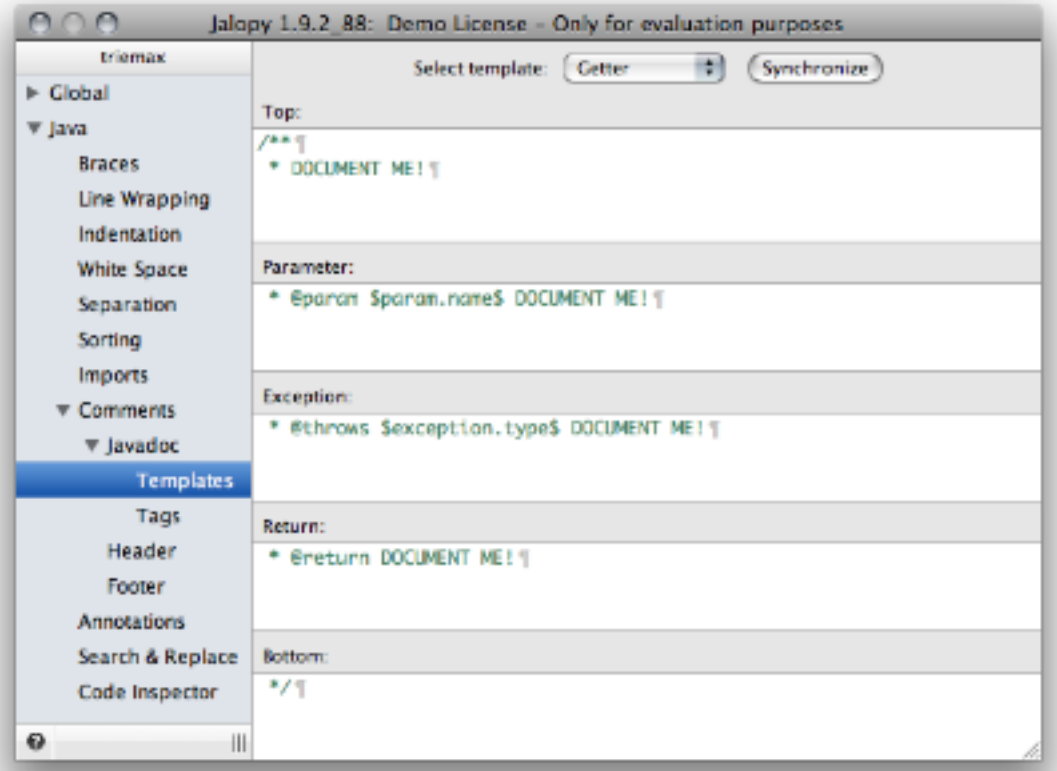
```
/**
 * Sets the value of the Importance Value property.
 *
 * @param name Importance Value property value.
 */
public void setImportanceValue(String value) {
}
```

You can use the *Synchronize* button to synchronize the template with the method declaration template.

Getter template

Lets you define the template for Getter methods (following the JavaBeans naming convention).

Figure 2.80. Javadoc Getter method template



The JavaBeans specification defines a standard way in which the properties for a JavaBean instance should be accessed. This same technique can also be applied to regular classes and interfaces to access their attributes.

The Getter template is used for all methods that look like `getXxx()` (also called accessor methods) and all methods matching the specified Boolean Getter pattern (refer to "Boolean Getter Regex" for further information).

Since 1.1

Example 2.858. Getter method declaration with generated Javadoc

```
/**
 * Returns the value of the Importance Value property.
 *
 * @return the Importance Value property.
 */
public void getImportanceValue() {
}
```

You can use the *Synchronize* button to synchronize the template with the method declaration template. Please note that you need to apply any changes made to the method declaration template first in order to see the changes propagated here.

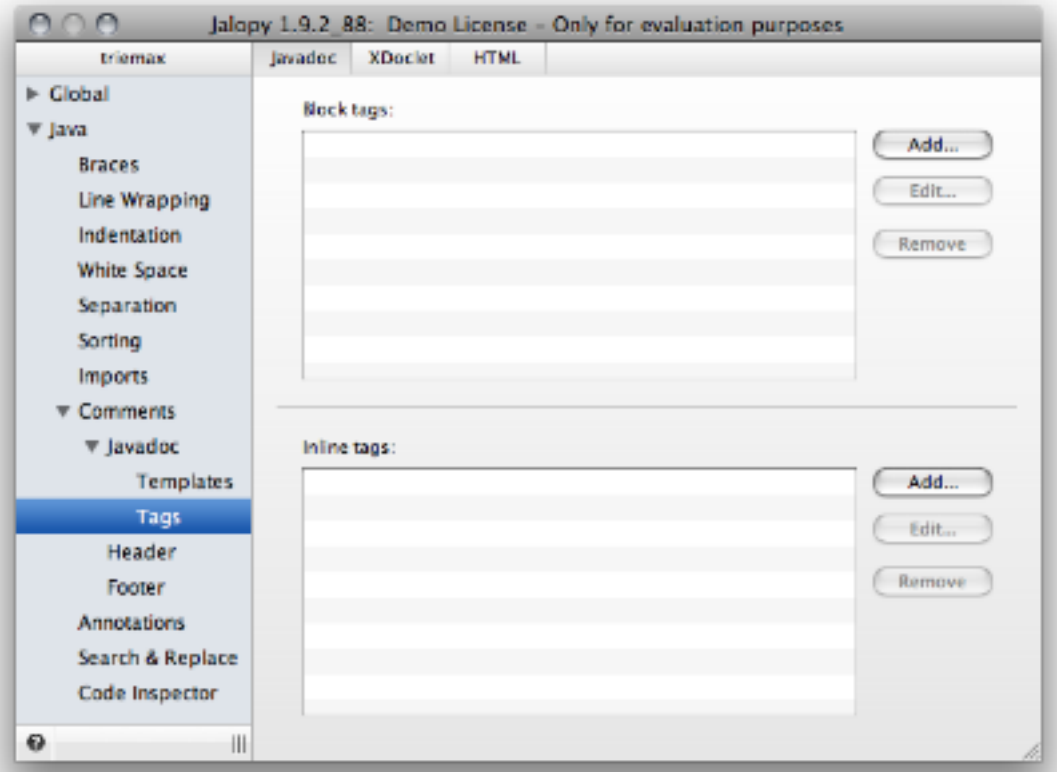
2.8.14.6 Tags

Lets you define custom tags that should be recognized by the Javadoc parser.

Javadoc

Lets you define custom Javadoc tags. You need to specify all non-standard tags that you use, i.e. all tags not defined in HTML 4.01, in order to see the Javadoc parser behave correctly. Otherwise errors are generated for every tag that is unknown to the system.

Figure 2.81. Define Custom Javadoc Tags



Refer to the tables below to learn about the tags that are supported by default.

Block tags

Lets you define custom Javadoc block tags. The table below shows the Javadoc block tags that are supported by default.

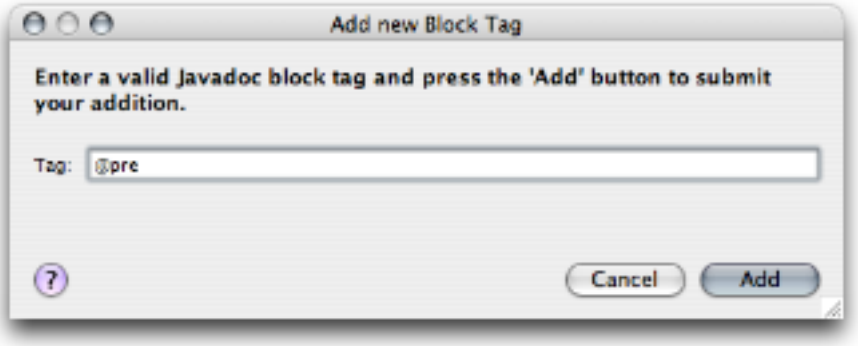
Table 2.7. Build-in Javadoc block tags

Name	Since
@author	1.0
@beaninfo	1.0
@deprecated	1.0
@exception	1.0
@jalopy.group	1.1
@jalopy.group-order	1.1
@jalopy.group_order	1.1
@param	1.0
@return	1.0
@see	1.0
@serial	1.0

Name	Since
@serialData	1.0
@serialField	1.0
@since	1.0
@throws	1.0
@todo	1.0
@version	1.0

Use the *Add...* and *Remove* buttons to add or remove items to and from the list.

Figure 2.82. Add new Block Tag



Valid block tags have the form `@[a-zA-Z]+`, e.g. `@pre`.

In-line tags

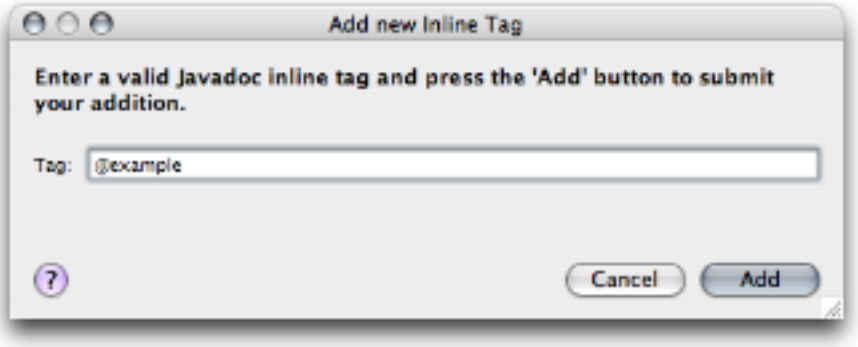
Lets you define custom Javadoc in-line tags. The table below shows the Javadoc in-line tags that are supported by default.

Table 2.8. Build-in Javadoc in-line tags

Name	Since
@code	1.3
@docRoot	1.0
@inheritDoc	1.0
@link	1.0
@linkPlain	1.0
@literal	1.3
@value	1.0

Use the *Add...* and *Remove* buttons to add or remove items to and from the list.

Figure 2.83. Add new In-line Tag

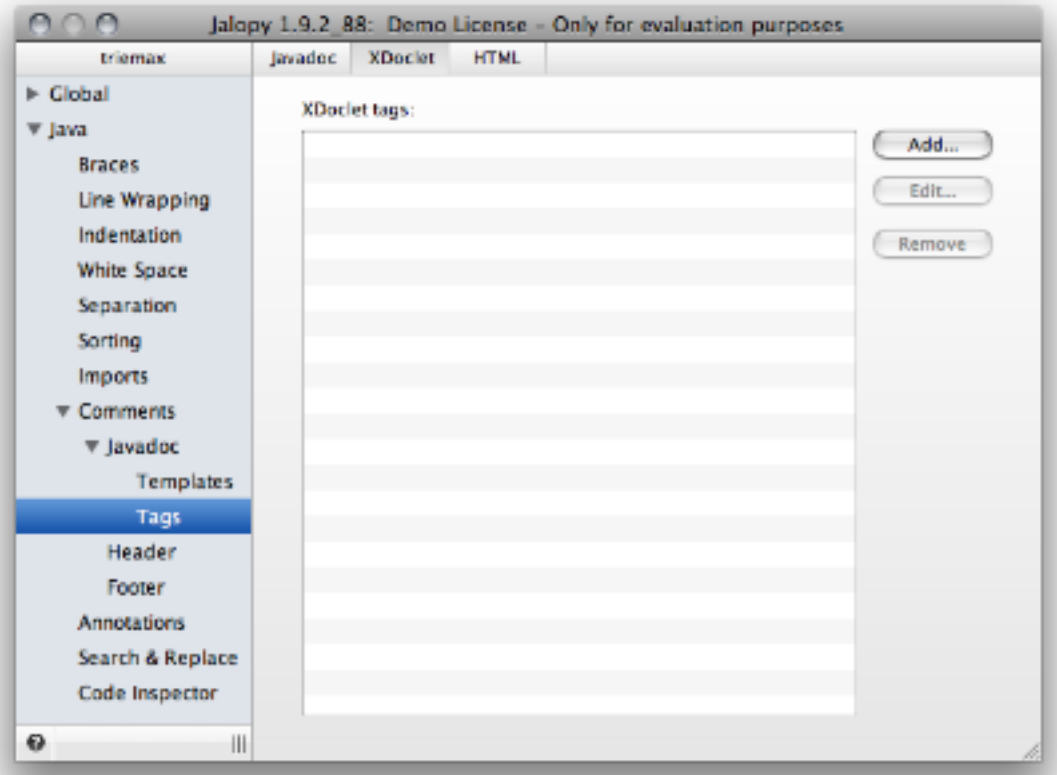


Valid in-line tags have the form `@[a-zA-Z]+`, e.g. `@root`.

XDoclet

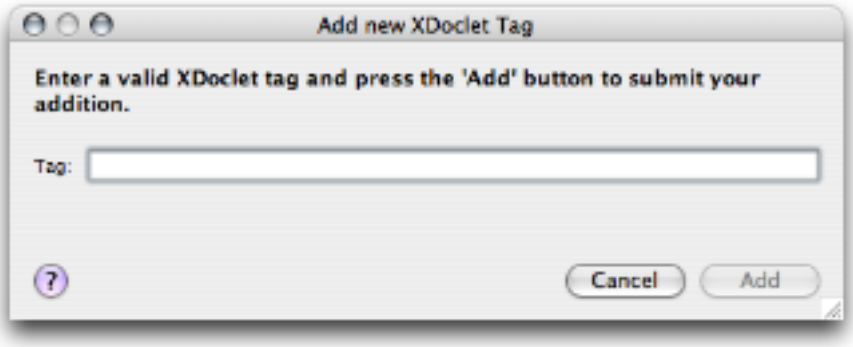
Lets you define custom XDoclet tags.

Figure 2.84. Define Custom XDoclet Tags



Refer to Appendix B, *Build-in XDoclet tags* for the tags that are supported by default. Use the *Add...* and *Remove* buttons to add or remove items to and from the list.

Figure 2.85. Add new XDoclet Tag



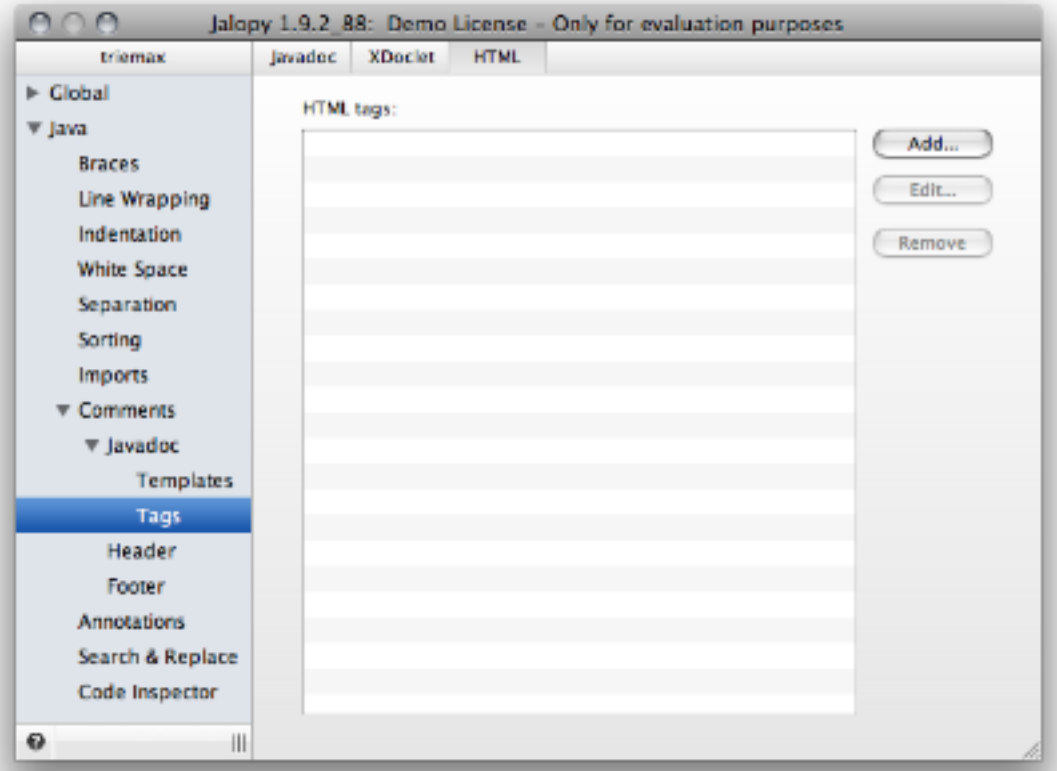
Valid XDoclet tags have the form `@[a-zA-Z.:_-]+`, e.g. `@jonas.session-timeout`.

Since 1.0

HTML

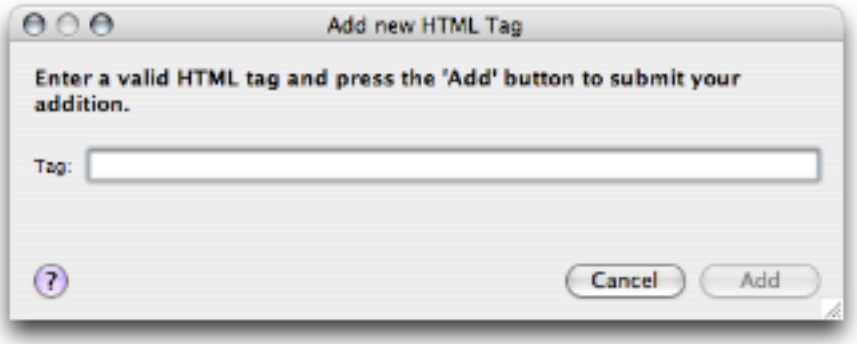
Lets you define custom HTML tags.

Figure 2.86. Define Custom HTML Tags



The standard supported tags are those of the HTML 4.01 standard. Use the *Add...* and *Remove* buttons to add or remove items to and from the list.

Figure 2.87. Add new HTML Tag



Since 1.0

2.8.15 Header

This section describes the available options to control the handling of headers. A header is a uniform comment that appears at the very top of a source file and usually displays the company's copyright notice.

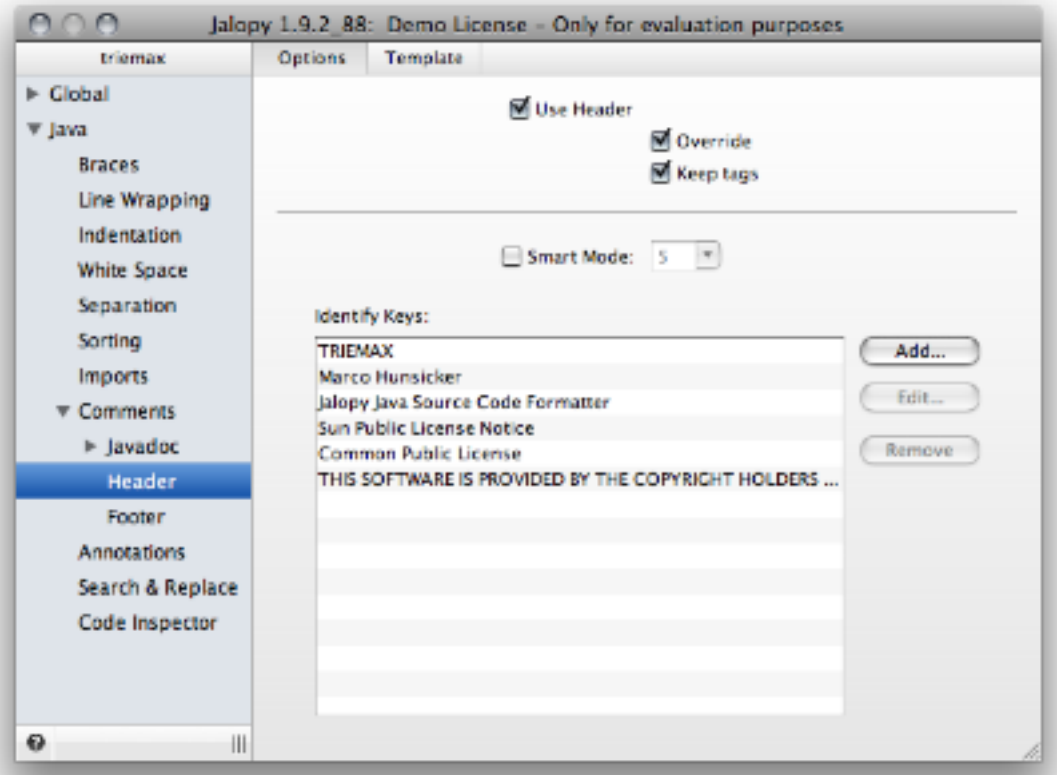
Example 2.859. Typical header before package statement

```
/*
 *                 Sun Public License Notice
 *
 * The contents of this file are subject to the Sun Public License
 * Version 1.0 (the "License"). You may not use this file except in
 * compliance with the License. A copy of the License is available at
 * http://www.sun.com/
 *
 * The Original Code is NetBeans. The Initial Developer of the Original
 * Code is Sun Microsystems, Inc. Portions Copyright 1997-2000 Sun
 * Microsystems, Inc. All Rights Reserved.
 */
package org.netbeans.editor;

...
```

2.8.15.1 Options

Lets you control the different header options.

Figure 2.88. Header Options settings page



Use Header

Enables or disables the header feature. When enabled and no header could be detected, the specified header template will be inserted. To avoid header duplication, you have to tell Jalopy how to detect existing headers. See Section 2.8.15.1.1, “Detection” below.

Override

If you enable this option, the header template will be re-inserted with every run. Any existing header(s) will be removed. Note that when you specify multiple keys to identify existing headers (see below), *all* recognized headers will be removed! This option is only available, if you've enabled the header feature.

Since 1.0

Keep tags

When enabled, Jalopy keeps expanded RCS-style tags in existing header comments. An expanded RCS tag looks like `$keyword: data $`.

It is good advise to replace headers upon every formatting run in order to enforce the company's copyright statement under all circumstances. This might cause problems though when the header contains RCS-style tags. Because the current keyword data is lost upon formatting, the SCM thinks the files are different even if the file did not change otherwise. So, after submitting the files show no differences (because the tags have been expanded again by the SCM). For example, if you have a file with the CVS `$Id$` tag, after checkout the header might look like this

Example 2.860. Header before formatting (keyword data present)

```
/* Copyright (c) 2001-2003, Foobar Systems Ltd.
 *
 * $Id: TestCheckin.java,v 1.2 2004/01/12 21:52:18 xf016997 Exp $
 */
```

But after formatting, the keyword data gets lost when the *Keep tags* option was disabled

Example 2.861. Header after formatting (keyword data deleted)

```
/* Copyright (c) 2001-2003, Foobar Systems Ltd.
 *
 * $Id$
 */
```

Enabling this option will allow you to keep the existing data and the file will look exactly like in Example 2.860, "Header before formatting (keyword data present)" after formatting. Please note that this feature works for nested tags, too. If you define RCS tags in your template that contain variable expressions, their values are still kept.

Since 1.0.3

Example 2.862. Header template with nested tags

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Fossi GmbH Source File: $file.name$
// Copyright (c) 2003-$date.year$ by Fossi GmbH
//
// $Created:    $date$ ($time.long$) by $user.name$ $
// Last Change: $date$ ($time.long$) by $user.name$
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

After the first formatting run, the header could look like this

Example 2.863. Header template after formatting

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Fossi GmbH Source File: Installer.java
// Copyright (c) 2003-2004 by Fossi GmbH
//
// $Created:    11.05.2004 (09:52:12) by eso $
// Last Change: 11.05.2004 (09:52:12) by eso
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

After the next formatting run, it might look like this

Example 2.864. Header template after further formatting

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Fossi GmbH Source File: Installer.java
// Copyright (c) 2003-2004 by Fossi GmbH
//
// $Created:    11.05.2004 (09:52:12) by eso $
// Last Change: 28.05.2004 (13:04:39) by harold
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

This option is only available, when the header feature has been enabled.

Detection

To avoid header duplication, Jalopy needs to know how existing headers can be recognized. Two methods are provided to allow great flexibility.

Smart Mode

Lets you specify the number of single-line comments at the top of a file (before the first language keyword, being either `package`, `import`, `class`, `interface`, `@interface` or `enum`) that should be recognized as a header.

Jalopy simply counts the number of single-line comments at the top of a source file and if this number is greater or equal to the specified #Smart Mode# count, *all* consecutive single-line comments at the top will be assumed to be part of a header.

Example 2.865. Single-line comment header

```
//============================================================
// file :       Byte.java
// project:     bsjt-rt
//
// last change: date:       $Date$
//              by:         $Author$
//              revision:   $Revision$
//------------------------------------------------------------
// copyright:   BSJT Software License (see class documentation)
//============================================================

package com.bsjt.foo;

import ...
```

In order to recognize the above single-line comments as a header, the #Smart Mode# count must be no less than '1', but it would be best to set it to '10'.

A number equal to '0' disables #Smart Mode#.

Identify Keys

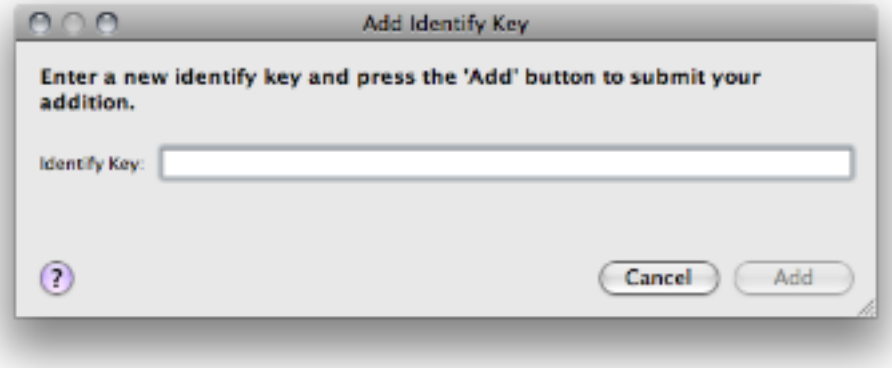
The second approach is to specify one or several unique keys that are part of your header. This technique only works with headers that are defined as multi-line comments. To add, remove or change identify keys, use the corresponding button beneath the keys list.

Specifying several keys makes it easy to switch between headers. Define both a key for the old header that is to be removed and for your new header that should be inserted. This way, you are sure that even new additions that happen to contain the old header (maybe checked out from some SCM) are treated correctly. A good key for the header in Example 2.859, "Typical header before package statement" above would be #Sun Public License Notice#.

Add...

Lets you add new identify keys. Pressing the button will invoke a new dialog where you can enter the identify key.

Figure 2.89. Add new Identify Key

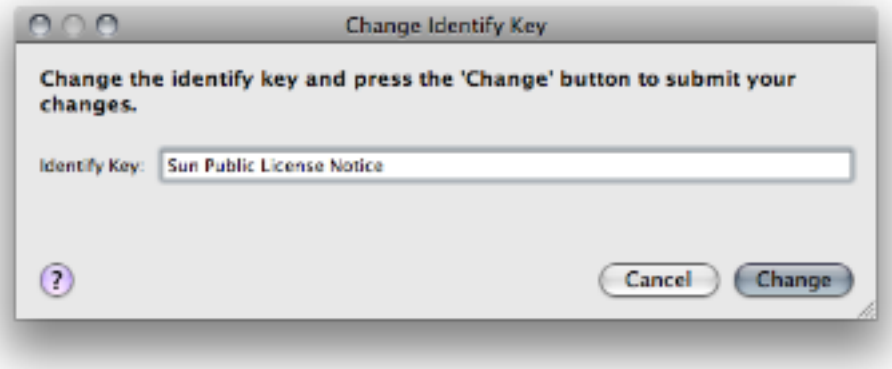


Enter the identify key in the text field and press the *Add* button to submit your addition. Press the *Cancel* button if you want to dismiss the action. Please note that the *Add* button is only available if the text field is not empty!

Change...

Lets you alter an already defined identify key. Pressing the button will invoke a new dialog where you can change the currently selected identify key. The button is only available if an item is currently selected in the keys list.

Figure 2.90. Change existing Identify Key



Change the identify key and press the *Change* button to submit your change. Press the *Cancel* button if you want to dismiss the action. Please note that the *Change* button is only available if the text field is not empty!

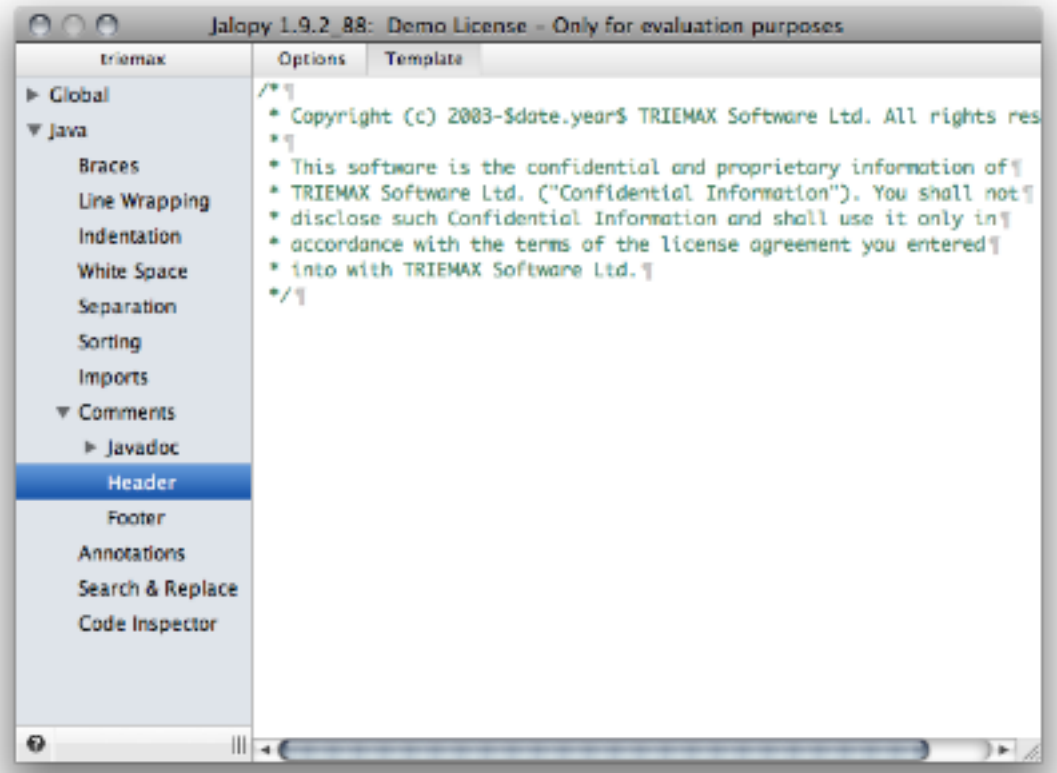
Remove

Lets you remove the currently selected key(s) from the list. The *Remove* button is only available if an item is currently selected in the keys list.

2.8.15.2 Template

Lets you specify the header template. Enter the desired text into the text area. You should use either one multi-line comment or several single-line comments. Any leading or trailing white space will be removed upon saving. Note that if you leave the template text empty, no header template will be inserted during printing, but existing headers may still be removed!

Figure 2.91. Header Template settings page



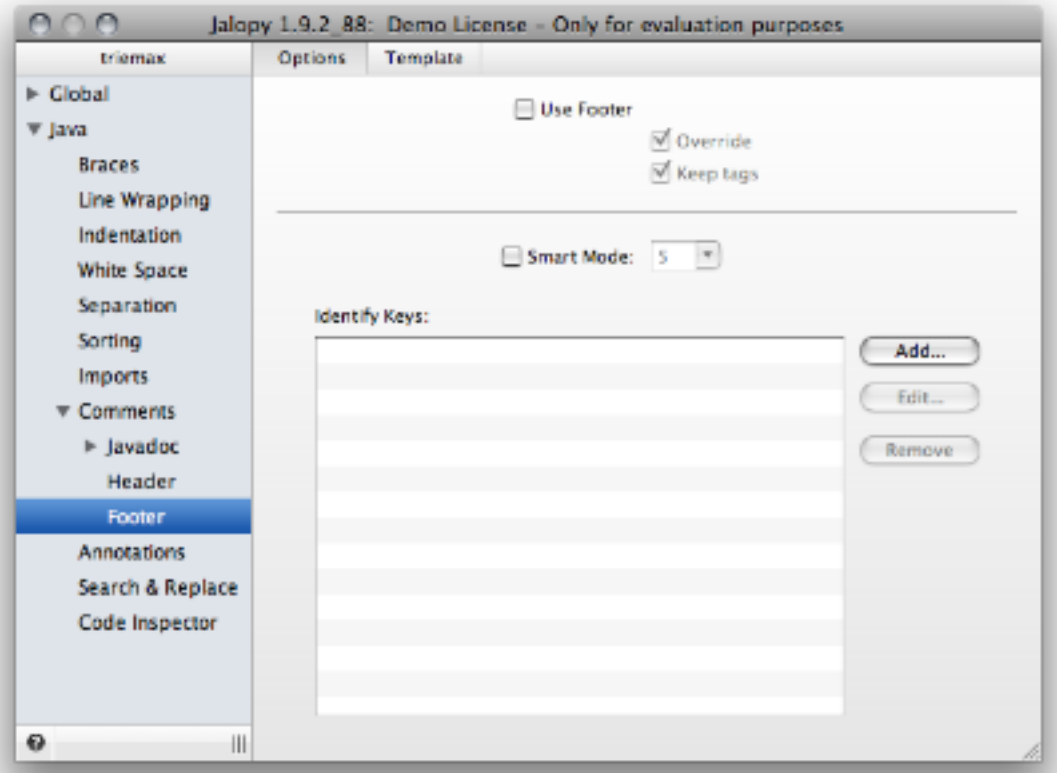
Since Jalopy 1.7, it's also possible to maintain multiple header comments in source files. Please note that if you prefer to separate the different comments with blank lines and don't want to enable the override feature, you need to tell Jalopy to keep blank lines between header comments with the Section 2.8.10.2, "Keep blank lines in headers up to" option. You can use variable expressions throughout the template text. See Section 2.4.4, "Usage" for more information about this feature.

2.8.16 Footer

Controls the printing of footers. A footer is a comment that appears at the very bottom of a source file and usually displays the change history or similar information. As the handling of

footers is analogous to headers, please refer to Section 2.8.15, “Header” for an explanation of the different options.

Figure 2.92. Footer settings page

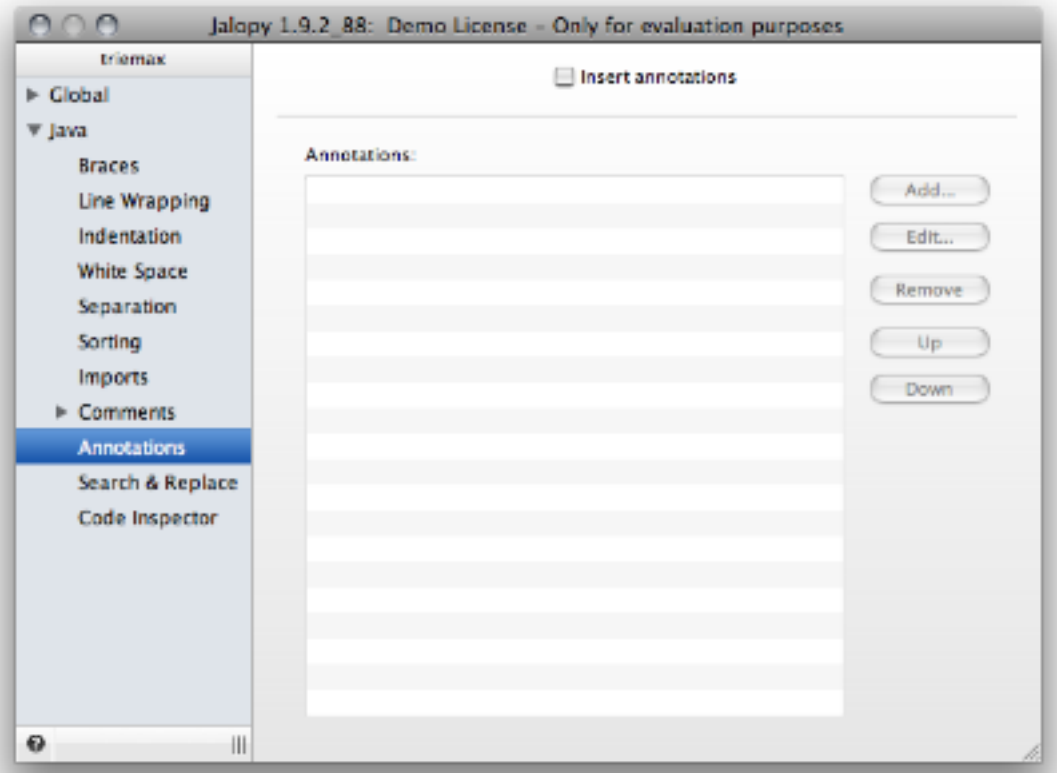


Note that Jalopy always prints one trailing empty line after the footer.

2.8.17 Annotations

Lets you configure annotations that should be added to top-level class, interface, enum and annotation type declarations.

Figure 2.93. Annotations settings page



Insert annotations

Enables or disables the automatic insertion of annotations. When enabled, all custom defined annotations (see below) that are not already present for a top-level declaration, will be inserted during formatting.

Since 1.8

2.8.17.1 Annotation patterns

Lets you define the annotation patterns that should be inserted. You can specify an arbitrary amount of patterns. The patterns will be inserted during formatting in the given order. The list component displays all patterns currently defined. Use the button bar on the right to add, remove or change patterns and define the order in which the patterns should be inserted.

Add...

Lets you add new annotation patterns. Pressing the button will invoke a new dialog where you can enter the pattern. Please note that the annotation must be fully qualified. During insertion, the package name will be stripped and the corresponding import declaration inserted.

Edit...

Lets you alter an already defined annotation pattern. This button is only available if an item is currently selected in the pattern list. Pressing the button will invoke a new dialog where you can change the Annotation pattern for the currently selected item in the pattern list.

Remove

Lets you remove an already defined annotation pattern. This button is only available if an item is currently selected in the pattern list.

Up

Lets you change the position of an already defined annotation pattern in the pattern list. This button is only available if an item is currently selected in the pattern list and this is not the topmost item.

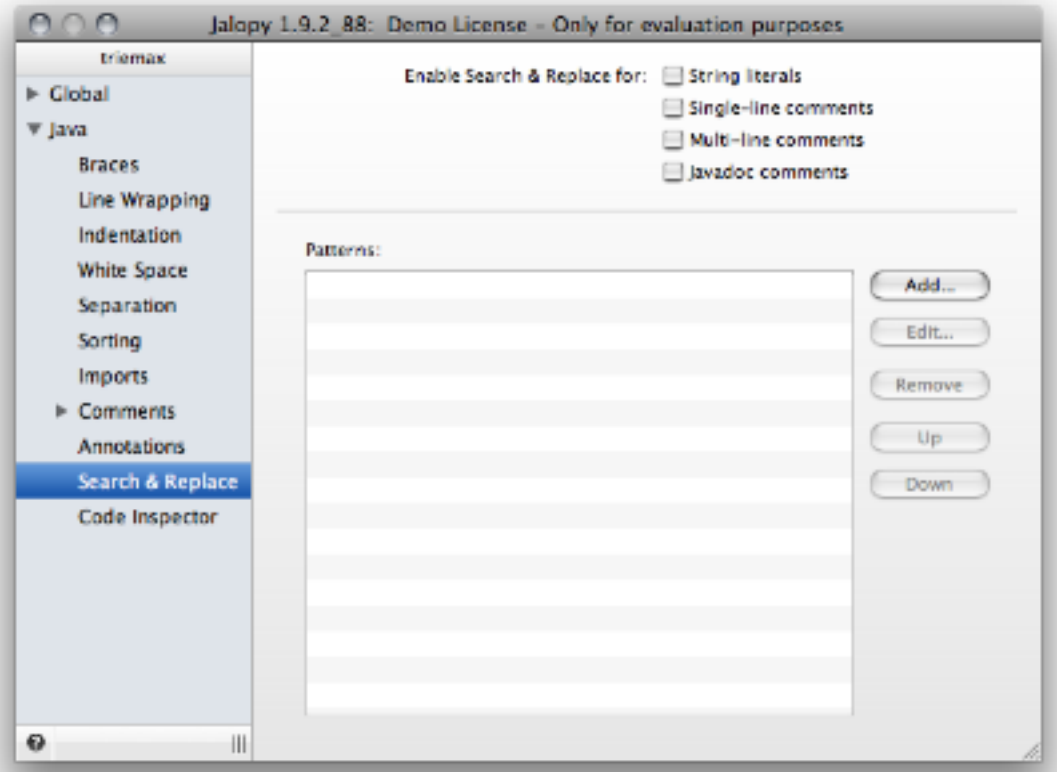
Down

Lets you change the position of an already defined annotation pattern in the pattern list. This button is only available if an item is currently selected in the pattern list and this is not the last item.

2.8.18 Search & Replace

Lets you perform string Search & Replace operations during formatting.

Figure 2.94. Search & Replace settings page



2.8.18.1 Scope

Lets you configure the elements for which Search & Replace should be performed.

String Literals

Enables Search & Replace for string literals. Please note that you first need to define at least one pattern in order to be able to enable Search & Replace! See Section 2.8.18.2, “Patterns” for information on adding patterns.

Since 1.7

Example 2.866. String literal

```
String literal = "String literals are enclosed in double quotes";
```

Single-line comments

Enables Search & Replace for single-line comments. Please note that you first need to define at least one pattern in order to be able to enable Search & Replace! See Section 2.8.18.2, “Patterns” for information on adding patterns.

Since 1.7

Example 2.867. Single-line comment

```
// Single-line comments are similar like in C++
```

Multi-line comments

Enables Search & Replace for multi-line comments. Please note that you first need to define at least one pattern in order to be able to enable Search & Replace! See Section 2.8.18.2, “Patterns” for information on adding patterns.

Since 1.7

Example 2.868. Multi-line comment

```
/* Multi-line comments are similar like in C/C++ */
```

Javadoc comments

Enables Search & Replace for Javadoc comments. Please note that you first need to define at least one pattern in order to be able to enable Search & Replace! See Section 2.8.18.2, “Patterns” for information on adding patterns.

Since 1.7

Example 2.869. Javadoc comment

```
/**
 * Javadoc comments are basically multi-line comments using
 * a special notation
 */
```

2.8.18.2 Patterns

Lets you define regular expression patterns to use for Search & Replace. You can define an arbitrary amount of Search & Replace patterns that are executed in the order defined when formatting a file. The list component displays all patterns currently defined. Use the

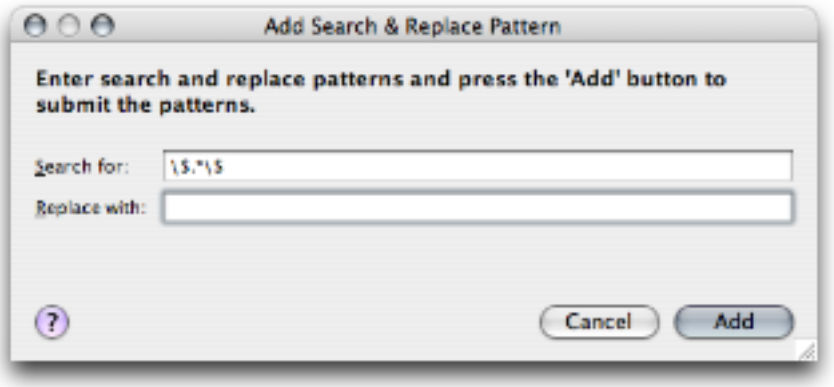
button bar on the right to add, remove or change patterns and define the order in which the patterns should be applied.

Jalopy uses Java's build-in regular expression engine which is roughly equivalent with Perl 5 regular expressions. The syntax is explained here: http://java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html. For a more precise description of the behavior of regular expression constructs consult *Mastering Regular Expressions* [Friedl97].

Add...

Lets you add new Search & Replace patterns. Pressing the button will invoke a new dialog where you can enter the Search & Replace patterns.

Figure 2.95. Add Search & Replace pattern



Enter the search pattern in the *Search for* text field and the replace pattern in the *Replace with* text field. The replace pattern may contain variable interpolations referring to the saved parenthesized groups of the search pattern. A variable interpolation is denoted by *$1*, *$2*, or *$3*, etc.

Suppose you have the search pattern *b\d+:* and you want to substitute the *b*'s for *a*'s and the colon for a dash in parts of your input matching the pattern. You can do this by changing the search pattern to *b(\d+):* and using the replace pattern *a$1-*. When a substitution is made, the *$1* means "Substitute whatever was matched by the first saved group of the matching pattern." An input of *b123:* after substitution would yield a result of *a123-*. For the given patterns

```
Tank b123: 85  Tank b256: 32  Tank b78: 22
```

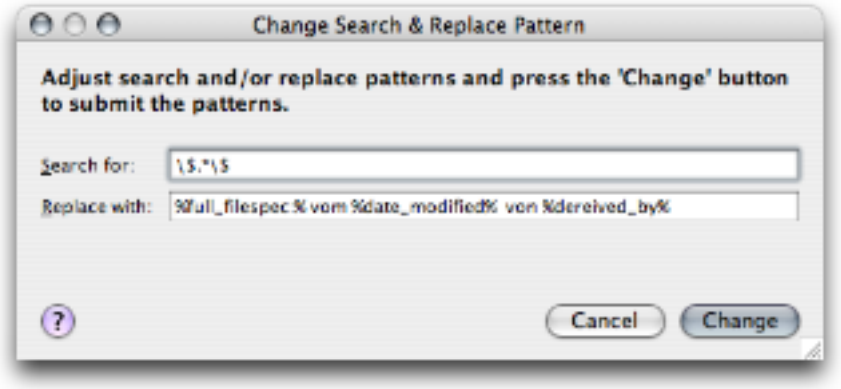
would become

```
Tank a123- 85  Tank a256- 32  Tank a78- 22
```

Edit...

Lets you alter an already defined Search & Replace pattern. This button is only available if an item is currently selected in the pattern list. Pressing the button will invoke a new dialog where you can change the Search & Replace patterns for the currently selected item in the pattern list.

Figure 2.96. Change Search & Replace pattern



Please refer to Section 2.8.18.2, “Add...” for an explanation of the available regular expression capabilities.

Remove

Lets you remove an already defined Search & Replace pattern. This button is only available if an item is currently selected in the pattern list.

Up

Lets you change the position of an already defined Search & Replace pattern in the pattern list. This button is only available if an item is currently selected in the pattern list and this is not the topmost item.

Down

Lets you change the position of an already defined Search & Replace pattern in the pattern list. This button is only available if an item is currently selected in the pattern list and this is not the last item.

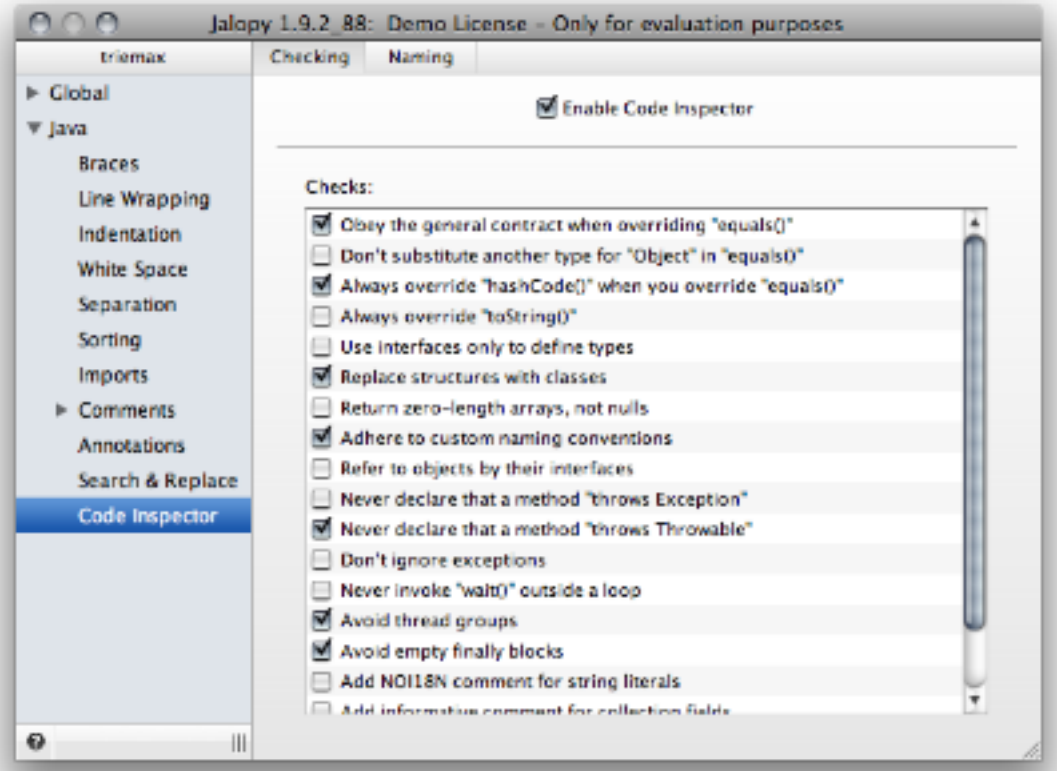
2.8.19 Code Inspector

Lets you configure the code inspector. The code inspector inspects source files for naming convention violations and possible code weaknesses.

2.8.19.1 Checking

Lets you control the general code inspector settings.

Figure 2.97. Code Inspector settings page



Enable

Lets you enable or disable the code inspector. You still need to enable at least one of the provided checks.

Checks

Lets you selectively choose what actions should be performed during inspection.

Obey the general contract when overriding equals

Checks whether the body of a `equals` method contains a `throw` statement that would violate the equals contract. Only applies if the method body does contain more than one statement; thus, if the method body consists of a single `throw` statement, we assume you know what you do and leave it alone. For more information see *Effective Java* [Bloch01], Item 7, pp. 32.

Don't substitute another type for Object

Don't substitute another type for `Object` in the equals declaration The `equals` method should not be overloaded.

Example 2.870. Overloaded equals - DO NOT USE!

```
public boolean equals(MyClass o) {
    ...
}
```

For a detailed discussion see *Effective Java* [Bloch01], Item 7, pp. 35.

Always override hashCode when you override equals

Failure to do so will result in a violation of the general contract for `Object#hashCode`, which will prevent the class from functioning properly in conjunction with all hash-based collections, including `HashMap` and `HashSet`. For a detailed discussion see *Effective Java* [Bloch01], Item 8, pp. 36.

Always override toString

It might be quite useful for diagnostic purposes to have objects generating interesting information with `toString`. This way you can easily use logging techniques to track programs execution.

Example 2.871. Provide useful toString() implemenation

```
System.out.println("Failed to connect: " + phoneNumber);
```

For a detailed discussion see *Effective Java* [Bloch01], Item 9, pp. 42.

Use interfaces only to define types

Interfaces should say something about what a client can do with instances of the class. It is inappropriate to define an interface for any other purpose. When enabled, Jalopy warns about uses of the so-called *constant interfaces* pattern, i.e. an interface that only consists of constants.

Example 2.872. Constant interface pattern - DO NOT USE!

```
public interface PysicalConstants {

    /** Boltzmann constant (J/K) */
    double BOLTZMANN_CONSTANT = 1.3806503e-23;

    /** Mass of the electron (kg) */
    double ELECTRON_MASS = 9.10938188e-31;
}
```

For a detailed discussion see *Effective Java* [Bloch01], Item 89, pp. 89.

Replace structures with classes

Degenerated classes consisting solely of data fields are loosely equivalent to C structures, but should not be used as they do not offer the benefits of encapsulation.

Example 2.873. Public degenerate class - DO NOT USE!

```
public class Point {
    public float x;
    public float y;
}
```

For a detailed discussion see *Effective Java* [Bloch01], Item 19, pp. 97.

Return zero-length arrays, not nulls

There is really no reason to ever return `null` from an array-valued method instead of returning a zero-length array. For a detailed discussion see *Effective Java* [Bloch01], Item 27, pp. 134.

Adhere to custom naming conventions

Following a naming convention aids readability and maintenance as confusion and irritation is avoided. For a detailed discussion see *Effective Java* [Bloch01], Item 38, pp. 165.

Refer to objects by their interfaces

When possible, always use the interface type for parameters, return values, variables and fields as your program will be much more flexible. When enabled, Jalopy will print warnings when the different Java collection implementations are used directly.

Example 2.874. // Good - uses interface as type

```
List subscribers = new ArrayList();
```

Example 2.875. // Bad - uses interface as type - DO NOT USE!

```
ArrayList subscribers = new ArrayList();
```

For a detailed discussion see *Effective Java* [Bloch01], Item 34, pp. 156.

Never declare that a method "throws Exception"

It is usually bad practise to declare that a method throws `Exception` because it obscures any other exception that may be thrown in the same context and denies any guidance to the programmer conceding the exceptions that the method is capable of throwing. For a detailed discussion see *Effective Java* [Bloch01], Item 44, pp. 181.

Never declare that a method "throws Throwable"

It is usually bad practise to declare that a method throws `Throwable` because it obscures any other exception that may be thrown in the same context and denies any guidance to the programmer conceding the exceptions that the method is capable of throwing. For a detailed discussion see *Effective Java* [Bloch01], Item 44, pp. 181.

Don't ignore exceptions

An empty catch block defeats the purpose of exceptions. At the very least, the catch block should contain a comment explaining why it is appropriate to ignore the exception.

Example 2.876. Empty catch block ignores exception - DO NOT USE!

```
try {
    ...
} catch (SomeException ex) {
}
```

For a detailed discussion see *Effective Java* [Bloch01], Item 47, pp. 187.

Never invoke wait outside a loop

Always use the `wait` loop idiom to invoke the `wait` method. Never invoke it outside of a loop as the loop serves to test the condition before and after waiting ensuring *liveness* and *safety* For a detailed discussion see *Effective Java* [Bloch01], Item 50, pp. 201.

Avoid thread groups

As thread groups are largely obsolete, don't use them. They don't provide much in the way of useful functionality, and much of the functionality they do provide is flawed. For a detailed discussion see *Effective Java* [Bloch01], Item 53, pp. 211.

Avoid empty finally blocks

Empty `finally` blocks are of no use and may indicate programmer errors.

Example 2.877. Empty `finally` block

```
Writer writer = new BufferedWriter(new FileWriter(file));

try {
    write.write(data);
} catch (IOException ex) {
    System.err.println("file could not be written: " + file);
} finally {
}
```

The programmer certainly wanted to close the `Writer` in the `finally` block to ensure that allocated system resources will be freed.

Add NOI18N comment for string literals

Enabling this check will cause warnings for all string literals without associated `/* NOI18N */` comment. Internationalizing Java applications is often done with nifty tools that use marker comments to indicate that a given string literal should not be considered for localization. Most tools (at least the ones I know of) use trailing single-line comments which may not be very robust for processing with a formatting tool such as Jalopy. In contrast the author uses a multi-line comment of the form `/* NOI18N */` that gets directly placed after a string literal and will therefore always stuck with it.

Example 2.878. $NON-NLS-1$ comment

```
FileDialog dialog = new FileDialog(this,
    ResourceBundle.getBundle(BUNDLE_NAME)
    .getString("BTN_SAVE_AS", FileDialog.SAVE); //$NON-NLS-1$
```

This trailing comment could be easily moved away from its string literal during formatting which would result in an unwanted notice on successive internationalization runs.

Example 2.879. $NON-NLS-1$ comment (moved)

```
FileDialog dialog =
    new FileDialog(this,
                   ResourceBundle.getBundle(BUNDLE_NAME).
                                  getString("BTN_SAVE_AS",
                   FileDialog.SAVE); //$NON-NLS-1$
```

Example 2.880. NOI18N comment

```
FileDialog dialog =
    new FileDialog(this,
                   ResourceBundle.getBundle(BUNDLE_NAME).
                                  getString("BTN_SAVE_AS" /* NOI18N */),
                   FileDialog.SAVE);
```

Add informative comment for collection fields

When not using strong-typed collections (a.k.a. Java Generics), it is best to document the object type of the items hold by a collection. When enabled, Jalopy checks for the existence of such comments and warns when they are missing.

Example 2.881. Collection comment

```
private List _favorableTypes = new ArrayList(20); // List of <String>
```

Warn about lines that exceed the maximal line length

When enabled, prints a warning for every line that was not printed in between the maximal line length.

Example 2.882. Line length limit

```
throw new IllegalArgumentException(                                |
    "condition must be one of WHEN_IN_FOCUSED_WINDOW or WHEN_FOCUSED");
                                                                   |
```

Suppress within //J- //J+ pragma comments

When enabled, no warnings are printed for code sections enclosed with pragma comments.

Since 1.8

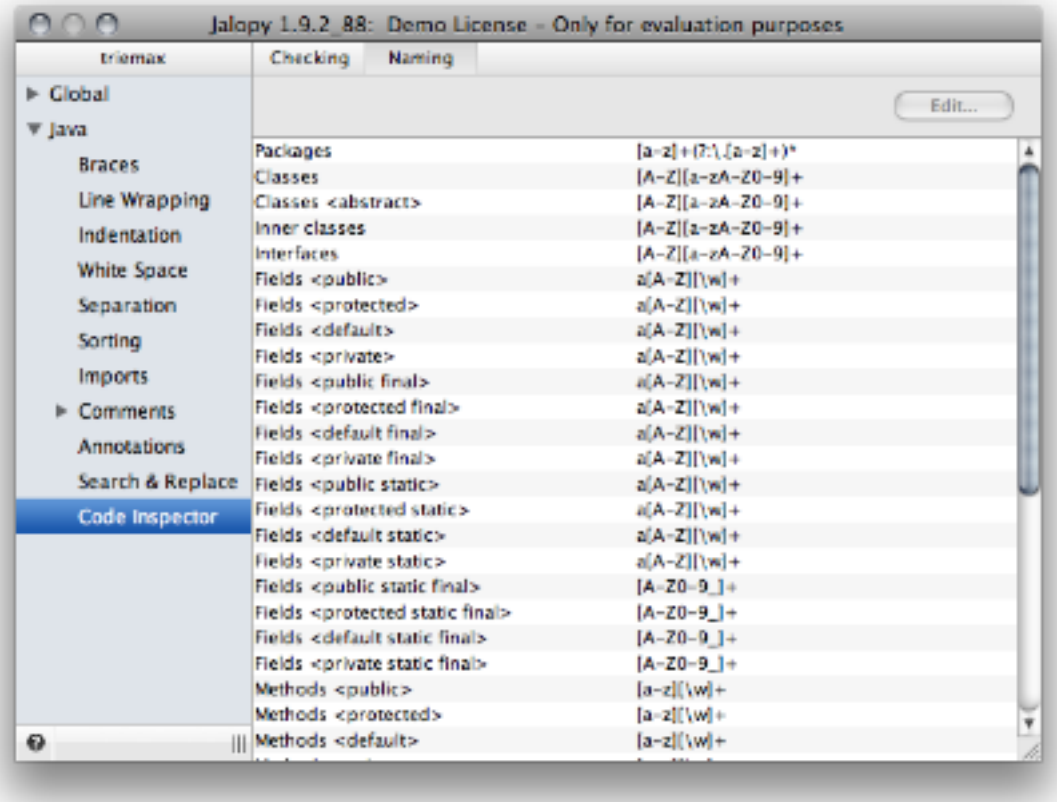
Example 2.883. Line length limit

```
//J-
throw new IllegalArgumentException(                                |
    "condition must be one of WHEN_IN_FOCUSED_WINDOW or WHEN_FOCUSED");
//J+                                                               |
```

2.8.19.2 Naming

Lets you specify the naming constraints for different Java source file elements. These constraints are naturally expressed with regular expressions. Note that you have to enable both the Code Inspector and the naming convention check in order to see naming checks performed. See Section 2.8.19.1.1, “Adhere to custom naming conventions” for more information.

Figure 2.98. Code Inspector Naming settings page



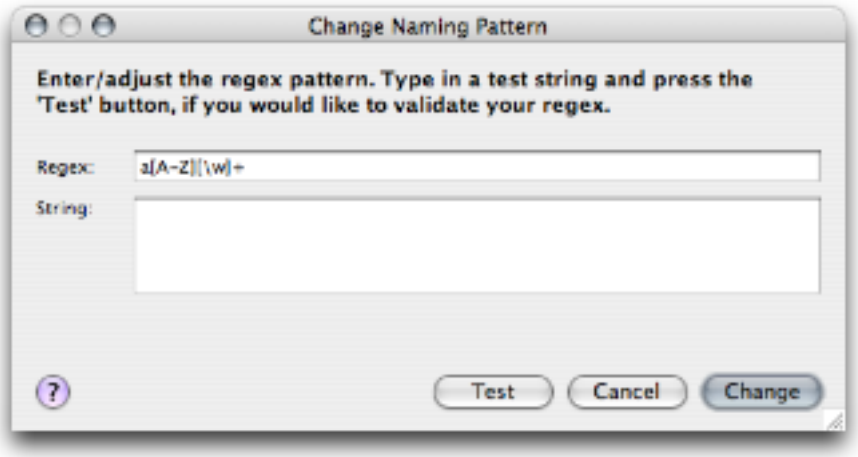
The list component displays all provided naming checks along with their current regular expression.

Selecting an item in the list and either pressing the *Edit...* button or double-clicking the item will open a dialog that lets you change the regular expression.

Change Naming Pattern

The Change Naming Pattern dialog lets you interactively craft a valid regular expression for a naming check.

Figure 2.99. Change Naming Pattern



Regex

The *Regex* text field is where you have to insert the regular expression. This text field initially contains the current pattern for the list item that is under construction.

Jalopy uses Java's build-in regular expression engine which is roughly equivalent with Perl 5 regular expressions. The syntax is explained here: http://java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html. For a more precise description of the behavior of regular expression constructs consult *Mastering Regular Expressions* [Friedl97]. The defined pattern must match exactly.

String

The *String* text field is where you have to enter a string that should be matched by the specified regular expression. This text field is initially empty.

Test

Once you have edited the two text fields you may want to use the *Test* button to perform a pattern matching test in order to make sure that the specified regex matches as desired. You will be informed about the match status and can decide whether you want to alter your pattern and/or test string and restart the procedure.

Figure 2.100. Successful regex test

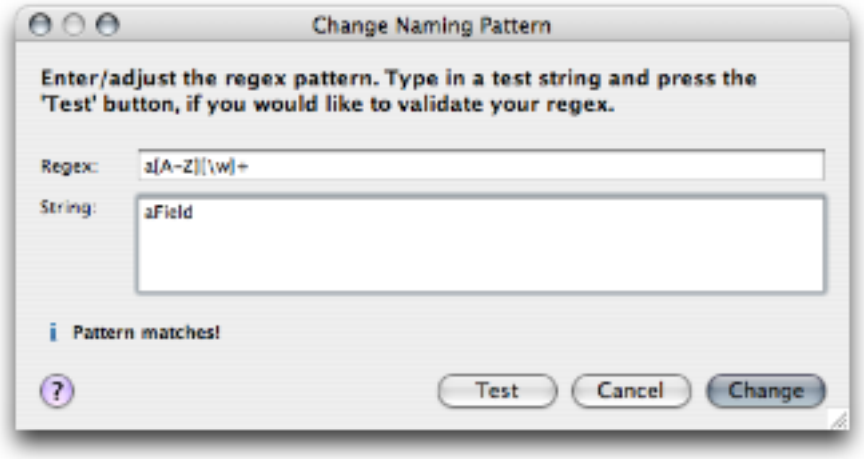
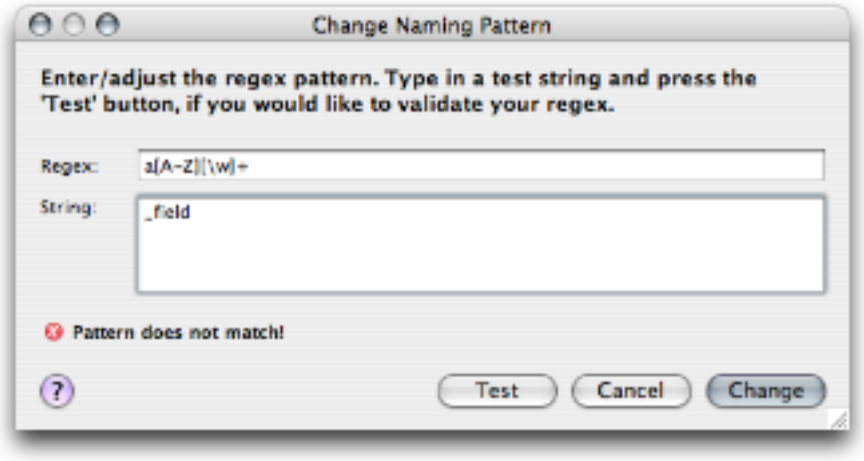


Figure 2.101. Failed regex test



Change

If you are finished editing the regular expression, you can press the *Change* button to take over.

Cancel

You can always use the *Cancel* button to cancel editing at any time. The dialog will be closed and no changes applied to the list.

Chapter 3. Usage

Usage depends on the distribution you received. Please refer to the individual Plug-in chapters in Part II, “Plug-ins” for details.

Part II. Plug-ins

This part of the manual covers the Plug-ins that ship with Jalopy. Plug-ins seamlessly integrate the extensive formatting capabilities of the Jalopy formatting engine into your favorite application. There is a wide range of Plug-ins available for IDEs, build tools and command line usage or scripting.

- Chapter 4, *Ant Task*
- Chapter 5, *Console Application*
- Chapter 6, *Eclipse Plug-in*
- Chapter 7, *IDEA Plug-in*
- Chapter 8, *JDeveloper Extension*
- Chapter 9, *jEdit Plug-in*
- Chapter 10, *Maven 1 Plug-in*
- Chapter 11, *Maven 2 Plug-in*
- Chapter 12, *NetBeans Module*

Chapter 4. Ant Task

Describes the installation and usage of the Jalopy Ant task. Its authors describe Ant [Link] as a “Java-based build tool. In theory, it is kind of like Make, but without Make’s wrinkles.

Why another build tool when there is already make, gnumake, nmake, jam, and others? Because all those tools have limitations that Ant’s original author couldn’t live with when developing software across multiple platforms. Make-like tools are inherently shell-based —they evaluate a set of dependencies, then execute commands not unlike what you would issue in a shell. This means that you can easily extend these tools by using or writing any program for the OS that you are working on. However, this also means that you limit yourself to the OS, or at least the OS type such as Unix, that you are working on. Makefiles are inherently evil as well. Anybody who has worked on them for any time has run into the dreaded tab problem. "Is my command not executing because I have a space in front of my tab???" asked the original author of Ant way too many times. Tools like Jam took care of this to a great degree, but still have yet another format to use and remember.

Ant is different. Instead of a model where it is extended with shell-based commands, Ant is extended using Java classes. Instead of writing shell commands, the configuration files are XML-based, calling out a target tree where various tasks get executed. Each task is run by an object that implements a particular Task interface. Granted, this removes some of the expressive power that is inherent by being able to construct a shell command such as 'find . -name foo -exec rm {}', but it gives you the ability to be cross platform—to work anywhere and everywhere. And hey, if you really need to execute a shell command, Ant has an <exec> task that allows different commands to be executed based on the OS that it is executing on.”

4.1 Installation

Explains the steps involved in getting the Ant task up and running.

4.1.1 System requirements

The Plug-in requires Ant 1.5 or later. See Section 1.1, “System requirements” for the basic requirements to run Jalopy.

4.1.2 Installation

The Plug-in comes as an executable Jar Archive (JAR) that contains a graphical setup wizard to let you easily install the software. Wizard installation is recommended and explained in detail in Section 1.3, “Wizard Installation”.

If you would rather install the Plug-in manually, you have to decompress and copy the appropriate files into the different application and/or settings folders. To decompress the contents of the installer JAR, you can often use the build-in support of your file manager (e.g. Nautilus) or any other software that can handle the ZIP compression format (e.g. 7Zip, WinZip or Stuffit Expander). If you don’t have access to one of the convenience tools, you might resort to the jar command-line application that ships with your Java distribution.

If you're upgrading from a prior version and want to keep your settings, first copy or rename the current Jalopy settings directory to match the version number of the new release. For instance, if your current settings directory is `C:\Documents and Settings\John Doo\.jalopy\1.9` and you're about to install Jalopy 1.9.3, either copy the directory contents or rename it to `C:\Documents and Settings\.jalopy\John Doo\1.9.3`. Wizard installation can perform this step automatically.

The Jalopy Ant task requires two JAR files, the Ant library `jalopy-ant-1.9.3.jar` and the core engine `jalopy-1.9.3.jar`. These must be added to the class path. You can do this in a number of ways:

- Add the two JAR files to the SCM repository and explicitly load the tasks using a class path you set up in the build file. This is often the best approach as there's no need for any work by the individual developers.
- Copy the two JAR files from the temporary directory into a private directory and add the directory contents to the path via Ant's `-lib` option. You can include this directory in the ANT_ARGS environment variable for automatic inclusion.
- Copy the two JAR files from the temporary directory into the `$HOME/.ant/lib` folder below your home directory. The library will be available on all projects which may lead to library version conflicts.
- Copy the two JAR files from the temporary directory into the `$ANT_HOME/lib` directory of your Ant installation. The library will be available to all users of a machine on all projects which may lead to library version conflicts.

Please note that you should make sure that no other Jalopy binaries are in the class path. Again you might need to check the global `$ANT_HOME/lib` directory of your Ant installation, the user specific `$HOME/.ant/lib` folder and any directories you include with the -lib option when running Ant, and remove any older `jalopy-*.jar` entries.

4.2 Configuration

Although Jalopy ships with sensible default settings (mimicking the Sun Java coding convention), you most likely want to configure the formatter to match your needs (adding copyright headers, tune Javadoc handling and the like). For such, Jalopy comes with a graphical configuration tool that lets you interactively customize the settings. See Chapter 2, *Configuration* for an in-depth discussion of the available options.

To display the configuration tool, you should use the matching wrapper script for your platform. The wrapper scripts are called `jalopy.xxx`. Open a shell and invoke the script with the `--configure` option:

```
% jalopy --configure
```

Or you can execute the JAR directly with

```
% java -jar jalopy-1.9.3.jar --configure
```

When you're done configuring the settings, you should export the code convention as described in Section 2.1.1.8, "Export code convention". The exported settings file is typically used as part of the Jalopy task configuration in the build script.

4.3 Usage

Before you can use the Jalopy Ant task in your build scripts, you have to define the task. This can be done in several ways, depending on the Ant and Jalopy versions you use.

The most conservative way to define the task that works with all versions, is to utilize the <taskdef> element in your build script and specify the class name of the Jalopy task. In order to achieve a self-contained build, you should not place the Jalopy libraries into your Ant `/lib` folder, but use the *classpath* attribute to point to the binaries:

Example 4.1. Task definition with specific class path

```
<taskdef name="jalopy"
         classname="com.triemax.JalopyTask"
         classpath="${deps}/jalopy-ant-1.9.3.jar" />
```

Please note that it's sufficient to reference the Jalopy Ant task library, if the core engine file sits in the same directory (which should be the norm).

If the Jalopy libraries have been added to the Ant class path (by placing them in the Ant`/lib` folder), you can simply bind the task as follows:

Example 4.2. Task definition with global search path

```
<taskdef name="jalopy" classname="com.triemax.JalopyTask" />
```

After the task has been defined, you can use it in the same manner like any other task:

Example 4.3. Task usage without namespace

```
<target name="format">
  <jalopy ...>
    ...
  </jalopy>
</target>
```

Since Jalopy 1.9.3, you can utilize the library feature available with Ant 1.6 or later. When the library has been added to the Ant class path, you can either bind the task globally for the whole script:

Example 4.4. Task definition via project namespace declaration

```
<project name="foo" xmlns:triemax="antlib:com.triemax" ...>
   ...
</project>
```

Or limited to a specific target:

Example 4.5. Task definition via target namespace declaration

```
<target name="jalopy" xmlns:triemax="antlib:com.triemax" />
  ...
<target/>
```

Declaring the namespace will automatically load the task and you can access it using the prefixed name defined in the declaration:

Example 4.6. Task usage with namespace

```
<target name="format">
  <triemax:jalopy ...>
    ...
  </triemax:jalopy>
</target>
```

But it is usually more sensible to leave the Ant class path alone and instead explicitly handle the class path in the build script to achieve a self-contained build process:

Example 4.7. Task definition with typedef

```
<target name="format">
  <typedef resource="com/triemax/antlib.xml" classpath="${deps}" />
  <jalopy ...>
    ...
  </jalopy>
</target>
```

Nested <fileset> elements can and should be used to specify the source files and/or directories:

Example 4.8. Target specification using fileset

```
<target name="format">
  <jalopy ...>
    <fileset dir="${dir.src.java}">
      <include name="**/*.java" />
    </fileset>
  </jalopy>
</target>
```

You can also set user environment variables for a run by using nested <variable> elements.

4.3.1 Parameters

The task itself can take several parameters to control the runtime behavior of Jalopy. The parameters are optional. When omitted, your current profile settings will be used. But it is recommended that at least a settings file is specified. The valid parameters are listed in the table below.

Table 4.1. Jalopy Ant task parameters

Attribute	Type	Description	Since	Required
backup	Boolean	Sets whether backup copies of all processed source files should be kept. When omitted, the corresponding code convention setting will be used (see Section 2.2.2.2, "Backup")	1.0	No
classpath	Path	The class path to setup the type repository with. If you want the import optimization or Insert Serial Version UID features to work, you have to specify the class path you use to compile your sources here. The referenced path must contain *all* types that are needed by your project. Specifying the Java runtime classes is optional; if they are omitted, the runtime classes used by Ant will be automatically added	1.9.3	No
classpathref	String	The class path to setup the type repository with, given as a reference to a path defined elsewhere. If you want the import optimization or Insert Serial Version UID features to work, you have to specify the class path reference you use to compile your sources here. The referenced path must contain *all* types that are needed by your project. Specifying the Java runtime classes is optional; if they are omitted, the runtime classes used by Ant will be automatically added	1.0	No
convention	String	Sets the location to the code convention settings file to use—given either relative to the project's base directory or as an absolute local path or internet address (refer to	1.0	No

Attribute	Type	Description	Since	Required
		Section 2.1.1.8, "Export code convention" for information how to export your settings). When omitted, and no profile is specified, the settings of the currently active profile will be used		
destdir	String	Sets the destination directory to create/copy all formatting output into. If the given directory does not exist, it will be created. When omitted, all input files will simply be overridden	1.0	No
encoding	String	Sets the encoding that controls how Jalopy interprets text files containing characters beyond the ASCII character set. Defaults to the platform default encoding	1.0	No
failonerror	Boolean	Sets whether a run should be held if errors occurred. Defaults to "true"	1.0	No
file	String	Specifies a single source file to format.	1.0	Yes, if no fileset is specified
fileformat	String	Sets the file format of the output files. The file format controls what end of line character is used. Either one of "UNIX", "DOS", "DEFAULT" or "AUTO" can be used (case insensitive). Defaults to "AUTO"	1.0	No
force	Boolean	Sets whether the formatting of files should be forced, even if a file is up-to-date. Defaults to "false"	1.0	No
fork	Boolean	Sets whether the processing should be performed in a separate VM. Defaults to "false"	1.0	No
history	String	Sets the history policy to use. Either one of "ADLER32", "CRC32" or "NONE" can be used (case insensitive). When omitted, the corresponding code convention setting will used (see Section 2.2.2.1, "History")	1.0.3	No
inputEncoding	String	Sets the character encoding that controls how Jalopy interprets input text files containing characters beyond the ASCII character set. Defaults to the platform default encoding. Please note that this setting always overrides *encoding*	1.6	No
javadoc	String	Indicates whether Javadoc related messages should be printed. Defaults to "true"	1.0	No
loglevel	String	Specifies the logging level for message output. Either one of "ERROR", "WARN", "INFO" or "DEBUG" can be used (case insensitive). When omitted, the current code convention settings will be used (see Section 2.6.1, "Categories")	1.0	No
log	String	Specifies the log file to use for logging output. The format of the logging output is determined by the extension of the given file. Valid extensions are ".log" for a custom plain text format, ".xml" for a plain XML format and ".html" for an hierarchical HTML report. When omitted, the current code convention setting will be used (see Section 2.6.2, "Logging")	1.0.3	No
outputEncoding	String	Sets the character encoding Jalopy uses to write files. Defaults to the platform default encoding. Please note that this setting always overrides *encoding*	1.6	No
profile	String	Sets the Jalopy profile that should be activated during the formatting run (refer to Section 2.1.1.1, "Main window" for more information about profiles). The currently active profile will be restored after formatting. Please note that the profile must exist!	1.5	No
repository	Boolean	Indicates whether the type repository should be used for type lookup. When disabled, this currently means that all dependent features despite the import optimization will	1.6	No

Attribute	Type	Description	Since	Required
		be disabled! Only meaningful if *classpathref* has been set. You may want to use this option if you commonly format a single file or only a sets of files in order to avoid the maintenance overhead of the type repository. Defaults to "true"		
test	Boolean	Sets whether formatting output should actually be written to disk. If set to "true" no output will be written to disk. The default is "false"	1.0	No
threads	Integer	Specifies the number of processing threads to use. Integer between `1 - 8`. When omitted, the current code convention setting will be used	1.0	No

4.3.2 Parameters specified as nested elements

Some parameters may be specified as nested elements.

<classpath>

The class path might be alternatively specified using the nested <classpath> element. It is recommended to use the same class path as with your compile target, to ensure that all project types are accessible.

Since 1.9.3

Example 4.9. Class path declaration using nested classpath element

```
<jalopy ...>
  <classpath>
    <pathelement name="${classpath}" />
  </classpath>
  ...
</jalopy>
```

<variable>

Used to specify a user environment variable that shall be available during a run. If a variable with the given name already exists, its value will be overridden during the run and restored afterwards.

Since 1.0

Table 4.2. Nested variable parameter

Attribute	Type	Description	Required
name	String	Specifies the name of the variable.	Yes
value	String	Specifies the value that should be assigned to the variable.	Yes

Example 4.10. Variable declaration

```
<jalopy ...>
  <variable name="author" value="John Doo" />
  ...
</jalopy>
```

4.4 Example

The following example demonstrates how you can make use of the Jalopy Ant task. Note that the `format` target depends on the `compile` target. This way we can make sure that the provided class path covers the complete type information.

Example 4.11. Example Ant build file

```
<?xml version="1.0" ?>
<project name="myProject" default="format" basedir=".">

  <property name="dir.compile" value="${basedir}/build/classes" />
  <property name="dir.lib" value="${basedir}/lib" />
  <property name="dir.src.java" value="${basedir}/src/main/java" />

  <!-- ================================================================== -->
  <!-- Defines the project class path -->
  <!-- ================================================================== -->
  <path id="project.classpath" >
    <!-- our compilation directory -->
    <pathelement location="${dir.compile}" />
    <!-- needed 3rd party libraries -->
    <fileset dir="${dir.lib}" >
      <include name="**/*.jar" />
    </fileset>
  </path>

  <!-- ================================================================== -->
  <!-- Compiles the project sources -->
  <!-- ================================================================== -->
  <target name="compile">
    <javac destdir="${dir.compile}" classpathref="project.classpath">
      <src path="${dir.src.java}" />
    </javac>
  </target>

  <!-- ================================================================== -->
  <!-- Formats the project source -->
  <!-- ================================================================== -->
  <target name="format" depends="compile">
    <!--
      Load the task using explicit class path. Please note that it's sufficient
      to reference the Jalopy Ant library JAR if the core engine JAR sits in
      the same directory (which should be the norm)
      -->
    <typedef resource="com/triemax/antlib.xml"
             classpath="${basedir}/../deps/jalopy-ant-1.9.3.jar" />

    <!--
      Invokes Jalopy as follows:

      - load the code convention from the given url
      - the import optimization feature will work (if enabled in the code
        convention), because a class path reference is given
      - all formatted files will have unix file format (\n)
      - override the convention to use alder32 checksums of files as history
        policy
      - override the convention to use loglevel 'info'
      - the task will use 4 worker threads
      - the user environment variable 'author' is set and the value
        'John Doo' assigned

      Since Jalopy 1.3 an include pattern is no longer necessary if you want
      to format all supported source files of a directory structure
      -->
```

```
    <jalopy convention="http://shared-server/cisco-omg.xml"
            classpathref="project.classpath"
            fileformat="unix"
            history="adler32"
            loglevel="info"
            threads="4">
      <variable name="author" value="John Doo" />
      <fileset dir="${dir.src.java}">
        <include name="**/*.java" />
      </fileset>
    </jalopy>
  </target>
</project>
```

Chapter 5. Console Application

Describes the installation and usage of the Console Plug-in. The Console Plug-in provides a powerful command-line interface for Jalopy.

5.1 Installation

Explains the steps involved to install the Console Plug-in.

5.1.1 System requirements

See Section 1.1, "System requirements" for the basic requirements to run Jalopy.

5.1.2 Installation

The Plug-in comes as an executable Jar Archive (JAR) that contains a graphical setup wizard to let you easily install the software. Wizard installation is recommended and explained in detail in Section 1.3, "Wizard Installation".

If you would rather install the Plug-in manually, you have to decompress and copy the appropriate files into the different application and/or settings folders. To decompress the contents of the installer JAR, you can often use the build-in support of your file manager (e.g. Nautilus) or any other software that can handle the ZIP compression format (e.g. 7Zip, WinZip or Stuffit Expander). If you don't have access to one of the convenience tools, you might resort to the jar command-line application that ships with your Java distribution.

If you're upgrading from a prior version and want to keep your settings, first copy or rename the current Jalopy settings directory to match the version number of the new release. For instance, if your current settings directory is `C:\Documents and Settings\John Doo\.jalopy\1.9` and you're about to install Jalopy 1.9.3, either copy the directory contents or rename it to `C:\Documents and Settings\.jalopy\John Doo\1.9.3`. Wizard installation can perform this step automatically.

Decompress the contents of the JAR file into a temporary directory. Afterwards create the actual installation directory, e.g. `C:\Program Files\Jalopy` or `/usr/local/java/jalopy` whatever. Create a new subfolder `/lib` and copy the file `jalopy-1.9.3.jar` from the temporary directory into the `/lib` folder. Copy the `/bin` folder from the temporary directory into the installation directory.

To invoke Jalopy, you can find wrapper scripts for the common platforms in the `/bin` folder. You may want to add this folder to your path. If your platform is not covered, you should make use of the `-jar` or `-cp` options of the Java application launcher (the *java* command), since this requires no class path manipulation (see Section 5.3.1, "Synopsis" below).

But if you don't want to use any of these options, you can add `jalopy-1.9.3.jar` to your class path as usual. For the Unix Bash shell, this means can be achieved using

```
% export CLASSPATH=${CLASSPATH}:<JALOPY_HOME>/lib/jalopy-1.9.3.jar
```

For Windows, use something like

```
% set CLASSPATH=%CLASSPATH%;<JALOPY_HOME>\lib\jalopy-1.9.3.jar
```

Refer to your system documentation on how to apply these changes more permanently.

5.2 Configuration

Although Jalopy ships with sensible default settings (mimicking the Sun Java coding convention), you most likely want to configure the formatter to match your needs (adding copyright headers, tune Javadoc handling and the like). For such, Jalopy comes with a graphical configuration tool that lets you interactively customize the settings. See Chapter 2, *Configuration* for an in-depth discussion of the available options.

To display the settings dialog you should use the provided wrapper script for your platform, called `jalopy.xxx` (available in the `/bin` folder of the distribution).

```
% jalopy --configure
```

Jalopy comes as an executable JAR file, you therefore can make use of the -jar option of the Java launcher:

```
% java -jar jalopy-1.9.3.jar --configure
```

Or you give the class path directly to the launcher

```
% java -cp jalopy-1.9.3.jar Jalopy --configure
```

Of course, you can externally configure the class path yourself by adding all `.jar` files as usual and then type

```
% java Jalopy --configure
```

on the console.

5.3 Usage

Presents the available command-line options along with some usage examples.

5.3.1 Synopsis

To start Jalopy from the command-line you may either use the provided launch script

```
% jalopy [-options] filespec...
```

Or use the Java launcher to execute the Jalopy binary directly

```
% java -jar jalopy-1.9.3.jar [-options] filespec...
```

Or use the Java launcher to call the main class

```
% java -cp jalopy-1.9.3.jar Jalopy [-options] filespec...
```

Or manually configure the class path and use the Java launcher to invoke the main class

```
% java Jalopy [-options] filespec...
```

Options

The command-line interface provides many options to control runtime behavior.

Table 5.1. Jalopy Console Plug-in command-line options

Option	Long Option	Arguments	Description	Since
	--classpath	<filepath>	Specifies the class path to use for type lookup. Entries are separated by semi colons. If you want to have either one of the Optimize imports, Insert Serial Version UID or Ignore runtime exceptions features working, you need to specify the class path used to compile your project here. The class path must contain *all* types that are needed by your project. Specifying the Java runtime classes is optional - if they are omitted, the runtime classes of the running VM will be automatically added. As a special convenience, specifying a directory is considered equivalent to specifying a list of all the files in the directory with the extension .jar or .JAR	1.1
	--configure		Invokes the graphical configuration dialog	1.0
-c	--convention	<filepath>	Specifies the absolute path to the exported code convention whose settings should be used for formatting, e.g. `"/work/quality/otng-jalopy.xml"`. Please note that specifying an exported code convention impacts your local profiles as Jalopy will import the code convention into the corresponding profile. If no profile with the name stored in the exported code contention exists, it will be created. Specifying a distinct profile to use via the -p,--profile option is therefore useless in this case. When omitted, the settings of the active profile will be used	1.0
-d	--dest	<filepath>	Sets the destination directory to create/copy all formatting output into. Expects a valid directory name. If the specified directory does not exist, it will be created. When omitted, all input files will be overridden	1.0
-e	--encoding	<string>	Specifies the encoding that controls how Jalopy interprets text files containing characters beyond the ASCII character set. Expects a Java supported character encoding name (like "US-ASCII", "ISO-8859-1" or "UTF-8"). Consult the release documentation for your Java implementation to see what encodings are supported. Please note that currently Jalopy does not support any "UTF-16" encoding. When omitted, the platform default encoding will be used	1.0
	--filespec	<filepath>	Specifies the absolute path to a file that defines the filespecs to use for formatting (see below). The filespec strings must be separated by line delimiters. Empty lines are ignored. Please note that you can still define filespecs directly on the command-line. When omitted, the file specs defined on the command-line will be used	1.7
-f	--format	<string>	Sets the file format of the output files. The file format controls what end-of-line character is used. Expects either one of "UNIX", "DOS", "MAC", "DEFAULT" or "AUTO" (case insensitive). When omitted, the corresponding code convention setting will be used	1.0
	--force		Sets whether the formatting of files should be forced, even if a file is up-to-date. When omitted, the corresponding code convention setting will be used	1.0
-h	--help		Displays a short help	1.0

Option	Long Option	Arguments	Description	Since
	--history	<string>	Sets the history policy to use. Either one of "ADLER32", "CRC32" or "NONE" can be used (case insensitive). When omitted, the corresponding code convention setting will used	1.0
	--input	<string>	Specifies the encoding that controls how Jalopy interprets input text files containing characters beyond the ASCII character set. Expects a Java supported character encoding name (like "US-ASCII", "ISO-8859-1" or "UTF-8"). Consult the release documentation for your Java implementation to see what encodings are supported. Please note that Jalopy does not yet support any "UTF-16" encoding. When omitted, the platform default encoding will be used	1.6
-l	--loglevel	<string>	Specifies the logging level for message output. Expects either one of "ERROR", "WARN", "INFO" or "DEBUG" (case insensitive). When omitted, the corresponding code convention settings will be used	1.0
	--look	<string>	Defines the Swing Look & Feel that should be used. Expects either the fully qualified name of a Swing Look & Feel that can be found on the class path. Or the abbreviation for some well known Look & Feels: Alloy, BlackStar, GreenDream, Liquid, Metal, Motif, Nimbus, PGS, Plastic, Plastic3d, PlasticXP, Synthetica, Windows (case-insensitive). Only meaningful in combination with the --configure option. When omitted, the default Look & Feel will be used (varies from platform to platform, but can be configured via the "swing.properties" preferences file)	1.0
	--nobackup		Indicates that no backup copies should be kept. When omitted, the corresponding code convention setting will be used	1.0
	--nofail		Indicates that processing should not stop when an error occurred. When omitted, processing terminates when an error occurs	1.0
	--norepository		Indicates that the type repository should not be used for type lookup. Please note that this currently means that all dependent features despite the import optimization will be disabled! Only meaningful when --classpath has been set. You may want to use this option if you commonly format a single file or only a small portion of files in order to avoid the maintenance overhead of the type repository. When omitted, the disk based type repository will be used	1.6
	--output	<string>	Specifies the character encoding that Jalopy uses to write text files. Expects a Java supported character encoding name (like "US-ASCII", "ISO-8859-1" or "UTF-8"). Consult the release documentation for your Java implementation to see what encodings are supported. Please note that currently Jalopy does not support any "UTF-16" encoding. When omitted, the platform default encoding will be used	1.6
-o	--override	<filepath> or <string>	Specifies local environment variable overrides. The value might either be a file path pointing to a properties file with key/value pairs. Or you may specify the key/value pair(s) directly using a `key=value` notation where the different pairs are separated by semicolons, e.g. `-o ";author=John Doo;project=FOZZY"`	1.6

Option	Long Option	Arguments	Description	Since
			Please note that when you want to specify several variables, the value string must be enclosed with quotes! Please refer to Section 2.4, "Environment" for more information about environment variables. When omitted, only the environment variables defined in the code convention will be used	
	--priority	<integer>	Sets the priority to use for worker threads. Expects an integer between 1-10 (inclusive). Bigger number means higher priority. Defaults to 5	1.9.2
--p	--profile	<string>	Sets the Jalopy profile that should be activated during the formatting run. Expects the name of an existing profile, e.g. "default" for the default profile. The currently active profile will be restored after formatting. When omitted, the currently active profile will be used if no code convention is specified	1.2.1
	--progress	<string>	Displays a progress bar during formatting. Runtime messages will be stored in the file "jalopy.log" in the current working directory	1.9.2
-q	--quiet		Suppresses noncritical messages. When omitted, the message settings of the code convention will be used	1.0
-r	--recursive		Recursively formats all files in the specified directories. When omitted, only the files in the specified directories will be formatted	1.0
	--test	<boolean>	Sets whether formatting output should actually be written to disk. If set to "true" no output will be written to disk. When omitted, all output will be written to disk	1.0
-t	--thread	<integer>	Specifies the number of processing threads to use. Expects an integer argument between 1-8 (inclusive). When omitted, the corresponding code convention setting will be used	1.0
	--track	<filepath>	Specifies the absolute path to a file where Jalopy will keep track of those files that would be actually formatted during a run. The file path strings will be separated by the platform line delimiter. Implies --test. When omitted, no track file will be written	1.4

Filespec

Filespecs define the source files and/or directories that should be formatted. You can specify as many filespecs as you want, where *filespec* describes either file paths, directories or filter expressions. If no filespec is given and no `--filespec` option specified, Jalopy starts listening on *STDIN*.

You can use any valid regular expression as a filter expression. Jalopy uses Java's build-in regular expression engine which is roughly equivalent with Perl 5 regular expressions. The syntax is explained here: http://java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html. For a more precise description of the behavior of regular expression constructs consult *Mastering Regular Expressions* [Friedl97].

5.4 Examples

Example 5.1. Sample command-line usage

```
% jalopy -r /dev/foo/src/java
```

Formats all source files found in directory `/dev/foo/src/java` and all subdirectories. The settings from the active profile are used.

Example 5.2. Sample command-line usage

```
% jalopy -d /test/foo -f DOS File1.java File1.java
```

Formats the two files `File1.java` and `File2.java` and writes the new files into directory `/test/foo`. Uses the settings from the active profile, but sets DOS as the file format used to write the files.

Example 5.3. Sample command-line usage

```
% jalopy -c /quality/foo.xml -r -d /test/foo ^A.*java
```

Formats all Java source files found in the current directory and all subfolders whose name start with a capital 'A' and writes the new files into directory `/test/foo`. The settings of the code convention `/quality/foo.xml` are used.

Example 5.4. Sample command-line usage

```
% type C:\Sources\Foo.java | jalopy > Foo.java
```

Formats the file `C:\Sources\Foo.java` read from STDIN and outputs its formatted contents to the file `Foo.java` in the current directory using the active profile.

Example 5.5. Sample command-line usage

```
% java -cp /usr/local/jalopy/jalopy-1.9.3.jar Jalopy         \
--convention=/usr/local/jalopy/jalopy.xml --norepository     \
--filespec=/tmp/3bD0W8.lst --track=/tmp/tXO7tE.lst
```

Formats all files specified in `/tmp/3bD0W8.lst` according to the given code convention, uses in-memory type lookup, and writes the paths of all files that were actually modified to the track file `/tmp/tXO7tE.lst.lst`. This is the typical invocation pattern when using Jalopy in a SCM pre-commit trigger to verify that all files have been formatted prior to check-in.

Chapter 6. Eclipse Plug-in

Describes the installation and usage of the Jalopy Eclipse Plug-in. Eclipse [Link] is an open platform for tool integration built by an open community of tool providers. Operating under an open source paradigm, with a common public license that provides royalty free source code and world wide redistribution rights, the Eclipse platform provides tool developers with ultimate flexibility and control over their software technology. Please note that the Plug-in also supports other Eclipse based products, like IBM Websphere Application Developer (WSAD), IBM Rational Application Developer (RAD), JBoss Developer Studio, CodeGear JBuilder, Genuitec MyEclipse etc.

6.1 Installation

Explains the steps involved to install the Eclipse Plug-in.

6.1.1 System requirements

The Plug-in requires Eclipse 3.0 or later. See Section 1.1, "System requirements" for the basic requirements to run Jalopy.

6.1.2 Setup

The Plug-in comes as an executable Jar Archive (JAR) that contains a graphical setup wizard to let you easily install the software. Wizard installation is recommended and explained in detail in Section 1.3, "Wizard Installation".

If you would rather install the Plug-in manually, you have to decompress and copy the appropriate files into the different application and/or settings folders. To decompress the contents of the installer JAR, you can often use the build-in support of your file manager (e.g. Nautilus) or any other software that can handle the ZIP compression format (e.g. 7Zip, WinZip or Stuffit Expander). If you don't have access to one of the convenience tools, you might resort to the jar command-line application that ships with your Java distribution.

If you're upgrading from a prior version and want to keep your settings, first copy or rename the current Jalopy settings directory to match the version number of the new release. For instance, if your current settings directory is `C:\Documents and Settings\John Doo\.jalopy\1.9` and you're about to install Jalopy 1.9.3, either copy the directory contents or rename it to `C:\Documents and Settings\.jalopy\John Doo\1.9.3`. Wizard installation can perform this step automatically.

Make sure Eclipse is not running and remove any present `com.triemax.jalopy_1.9.3` directory in your Eclipse plugin folder. This folder is usually located in the root directory of your Eclipse installation, e.g. `C:\Program Files\Eclipse\plugins\`.

Copy the Jalopy Plug-in folder `com_triemax.jalopy_1.9.3` from the temporary directory into the Eclipse plugin folder. Then place the two JAR files `jalopy-1.9.3.jar` and `jalopy-eclipse-1.9.3.jar` from the temporary directory into the Jalopy Plug-in folder.

If you are running Eclipse 2.1, as a final step delete the file `plugin.xml` from the Jalopy Plug-in folder and rename the file `plugin.xml-2.x.xml` to `plugin.xml`.

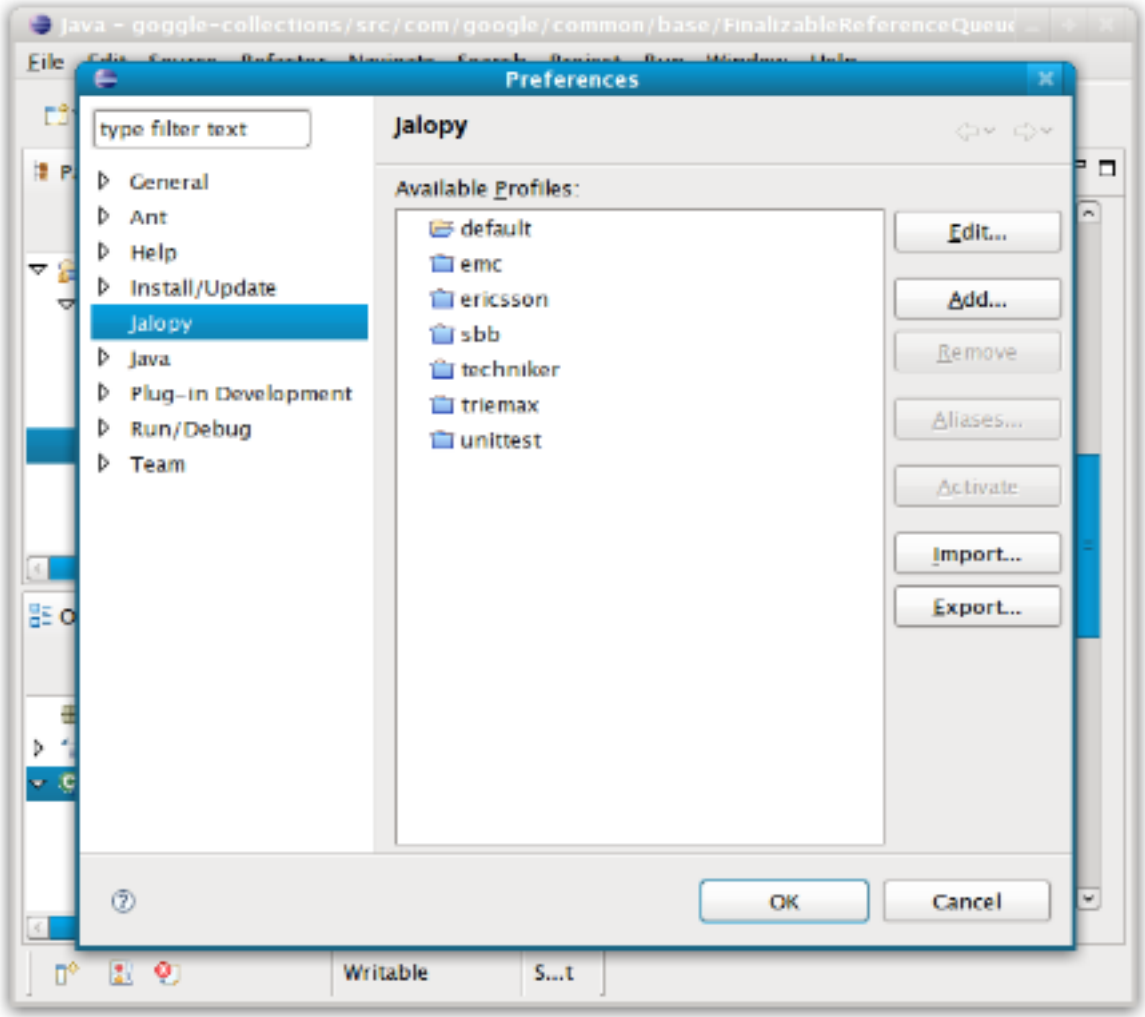
6.2 Integration

Describes how the Plug-in integrates into the Eclipse IDE.

6.2.1 Preferences

The Jalopy preferences are available through the Eclipse preferences dialog. In order to access the preferences, on Mac OS X you use Eclipse > Preferences... and select the Jalopy item on the left pane. On other platforms the dialog is available through Window > Preferences.... In order to quickly locate the item, you might want to type "Jalopy" in the filter field at the top of the left pane.

Figure 6.1. Main Jalopy Preferences page



The main preferences page lets you manage your Jalopy profiles. A profile stores the actual code convention that defines the formatting style, as well as user-specific data like file and dialog histories. You can add, remove, configure, activate and associate any number of profiles. For a detailed explanation of the available options, please refer to Section 2.1.1.1, "Main window".

NOTE Due to technical reasons it is currently not possible to configure profiles from within Eclipse when running on Mac OS X. When using Mac OS X, you need to invoke the Jalopy preferences dialog from outside Eclipse. Simply install the Console Plug-in and invoke the dialog as described in Section 5.2, "Configuration". Configure your code convention and afterwards export it to a file. From within Eclipse you can then import this configuration

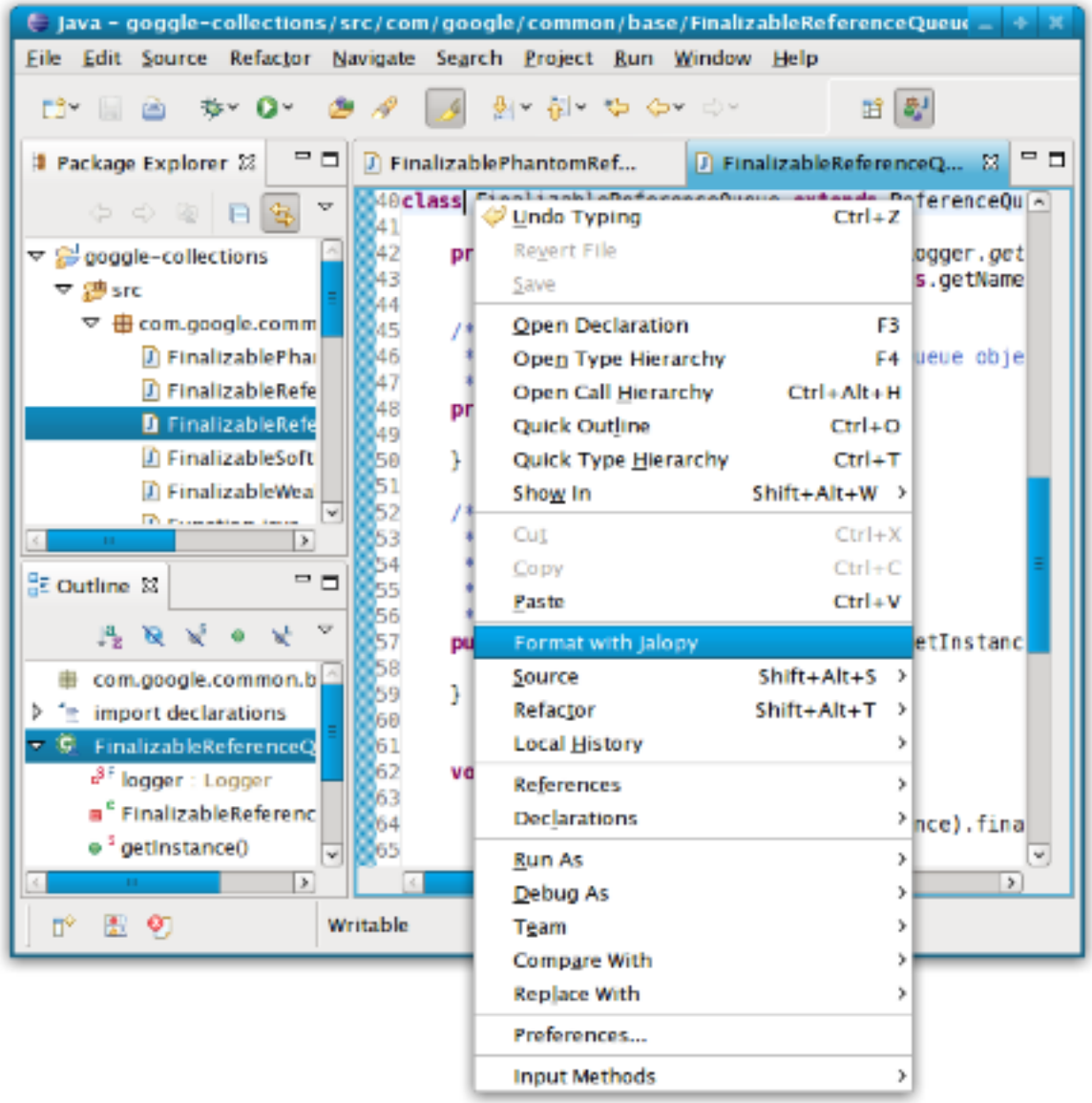
6.2.2 Java Editor pop-up menu

The software adds a new menu item to the pop-up menu of Java editors.

Format with Jalopy

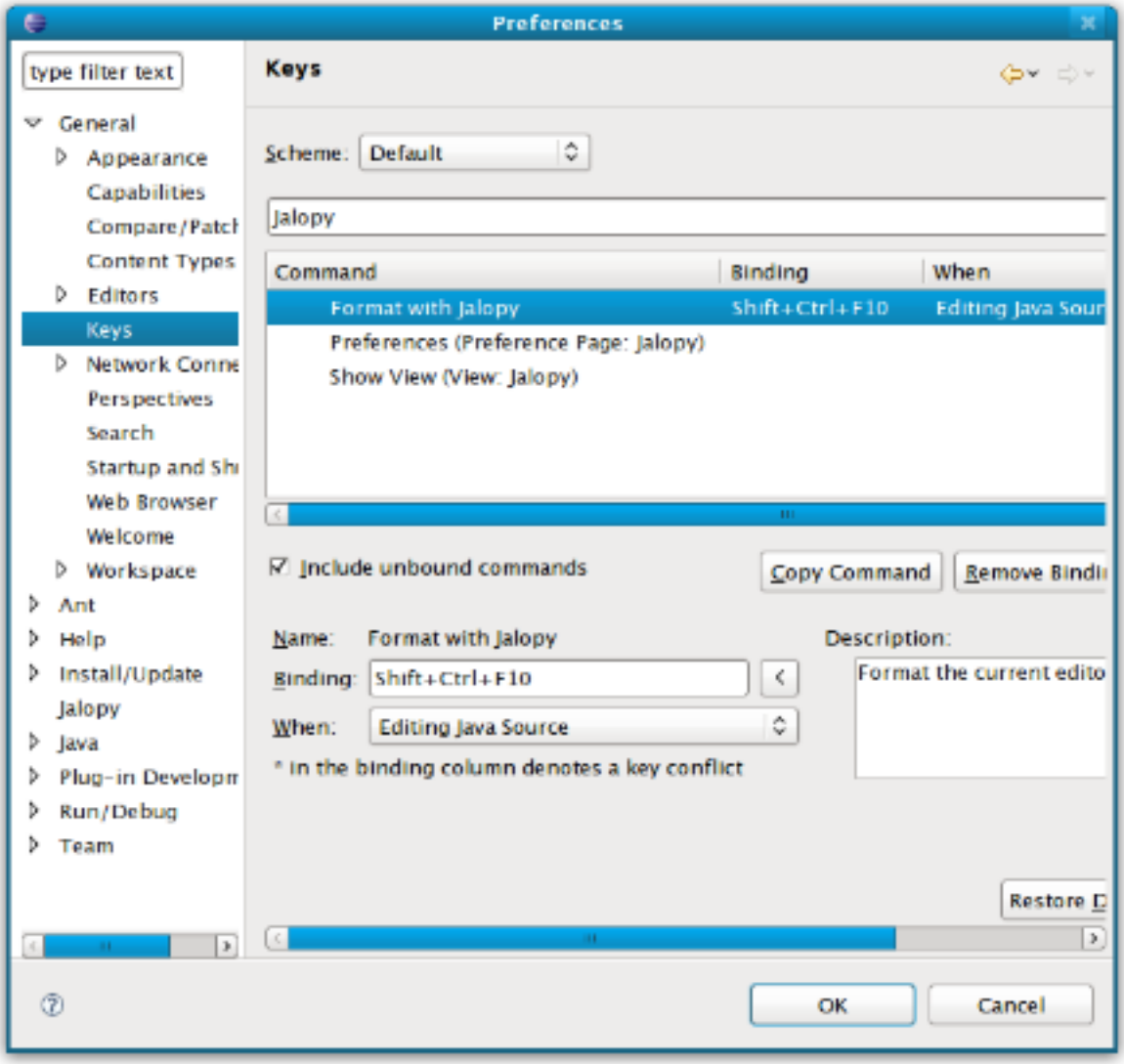
Formats the contents of the editor. When currently some text is selected in the editor, only the selected text will be formatted (*selective formatting*). This can be especially helpful when editing portions of very large files, as selective formatting can speed up processing considerably. But comes especially when you want to limit formatting to a specific file portion in order to avoid unnecessary differences when editing a file that has not (yet) been formatted according to the active code convention.

Figure 6.2. Editor pop-up menu



The default keyboard shortcut for this action is Ctrl-Shift-F10. To configure the shortcut, open the Eclipse Keys preference page via Window > Preferences > Workbench > Keys. Open the *Source* category and select the Format with Jalopy item.

Figure 6.3. Keyboard shortcut



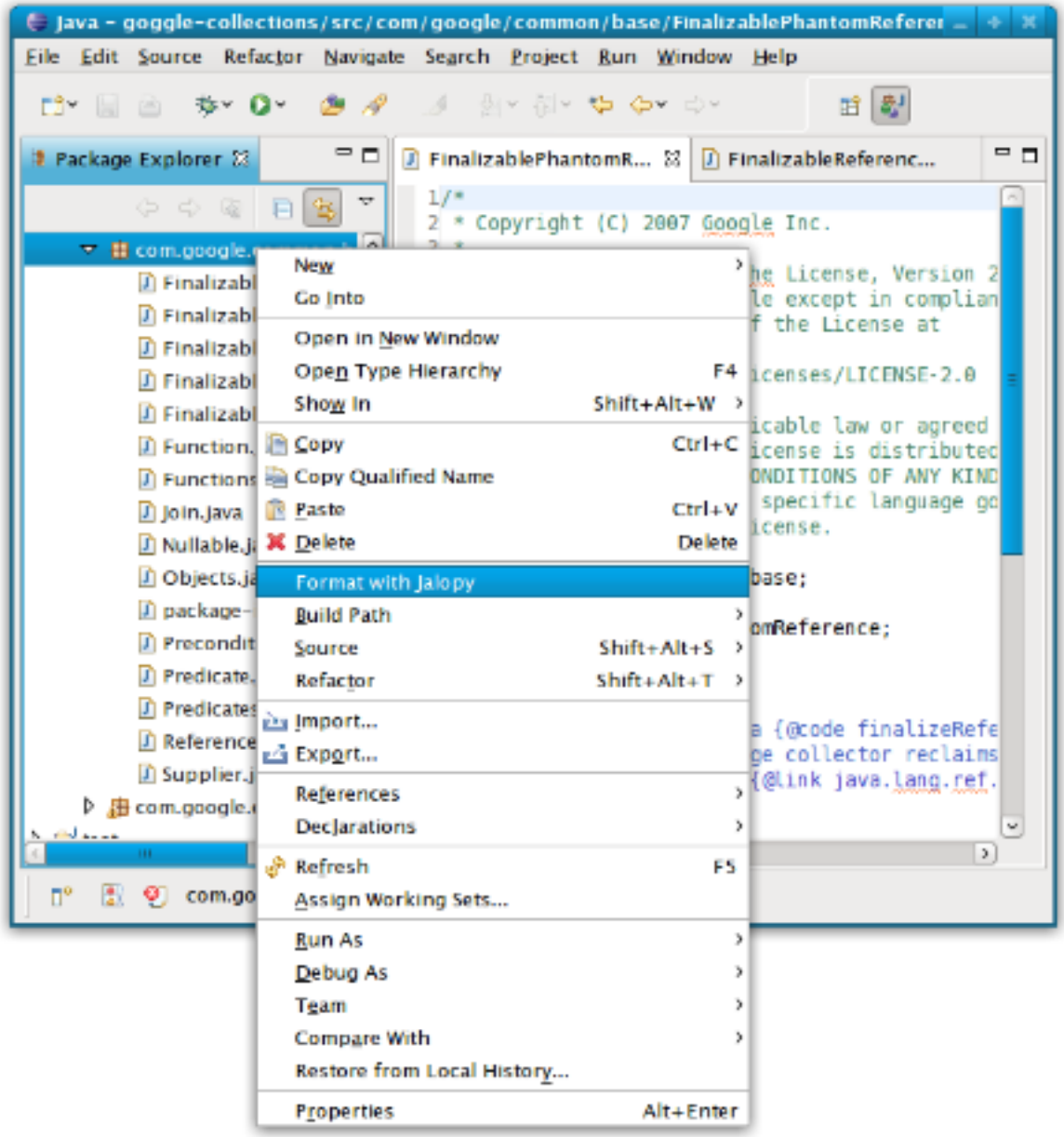
6.2.3 Project, Folder, File pop-up menus

The software adds a new menu item to the pop-up menu of projects, folders, packages and Java source files in the Navigator and Package Explorer view of the Java perspective.

Format with Jalopy

Formats the selected item(s). Depending on the object type (working set, project, folder, file) formats either all Java source files of the project, the contents of the selected folder(s), including subfolders, or the currently selected source file(s).

Figure 6.4. Project pop-up menu



6.3 Configuration

Although Jalopy ships with sensible default settings (mimicking the Sun Java coding convention), you most likely want to configure the formatter to match your needs (adding copyright headers, tune Javadoc handling and the like). For such, Jalopy comes with a graphical configuration tool that lets you interactively customize the settings. See Chapter 2, *Configuration* for an in-depth discussion of the available options. Please refer to Section 6.2.1, "Preferences" for information on how to display the configuration tool from within Eclipse.

Chapter 7. IDEA Plug-in

Describes the installation and usage of the Jalopy IntelliJ IDEA Plug-in. IntelliJ IDEA [Link] is an intelligent Java IDE intensely focused on developer productivity. It provides a robust combination of enhanced development tools, including: refactoring, J2EE support, Ant, JUnit, and CVS integration. Packaged with an intelligent Java editor, coding assistance and advanced code automation tools, IDEA enables Java programmers to boost their productivity while reducing routine time consuming tasks.

7.1 Installation

Explains the steps involved to install the IDEA Plug-in.

7.1.1 System requirements

This Plug-in requires IDEA 5.0 or higher. See Section 1.1, "System requirements" for the basic requirements to run Jalopy.

7.1.2 Setup

The Plug-in comes as an executable Jar Archive (JAR) that contains a graphical setup wizard to let you easily install the software. Wizard installation is recommended and explained in detail in Section 1.3, "Wizard Installation".

If you would rather install the Plug-in manually, you have to decompress and copy the appropriate files into the different application and/or settings folders. To decompress the contents of the installer JAR, you can often use the build-in support of your file manager (e.g. Nautilus) or any other software that can handle the ZIP compression format (e.g. 7Zip, WinZip or Stuffit Expander). If you don't have access to one of the convenience tools, you might resort to the jar command-line application that ships with your Java distribution.

If you're upgrading from a prior version and want to keep your settings, first copy or rename the current Jalopy settings directory to match the version number of the new release. For instance, if your current settings directory is `C:\Documents and Settings\John Doo\.jalopy\1.9` and you're about to install Jalopy 1.9.3, either copy the directory contents or rename it to `C:\Documents and Settings\.jalopy\John Doo\1.9.3`. Wizard installation can perform this step automatically.

Make sure IDEA is not running and remove any present Jalopy files in your IDEA Plugin folder. The IDEA Plugin folder is located in the root directory of your IDEA installation, e.g. `C:\Program Files\IDEA\plugins`. Check for a `jalopy/lib` subdirectory. If it exists, delete its contents. Otherwise create it.

Then copy the two JAR files `jalopy-1.9.3.jar` and `jalopy-idea-1.9.3.jar` into this `plugins/jalopy/lib` folder.

7.2 Integration

Describes how the Plug-in is integrated with the IntelliJ IDEA IDE.

7.2.1 Settings

The Jalopy preferences are available through the IDEA preferences dialog. In order to access the preferences, on Mac OS X you use IntelliJ IDEA > Preferences... and select the Jalopy item in the window. On other platforms the dialog is available through File > Settings....

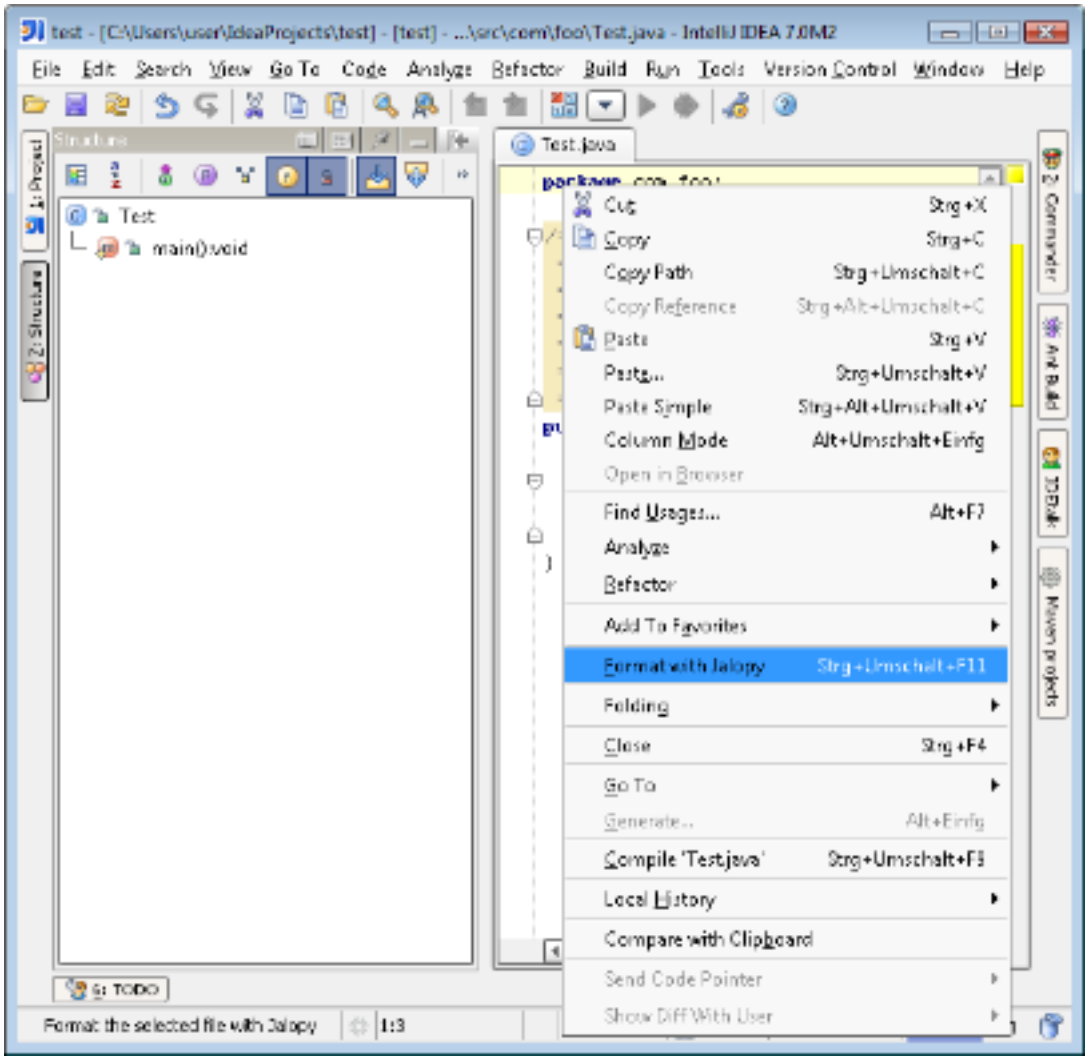
7.2.2 Code Editor Pop-up Menu

The software adds a new menu item to the pop-up menu of Java code editors.

Format with Jalopy

Formats the contents of the editor. When currently some text is selected in the editor, only the selected text will be formatted (*selective formatting*). This can be especially helpful when editing portions of very large files, as selective formatting can speed up processing considerably. But comes especially when you want to limit formatting to a specific file portion in order to avoid unnecessary differences when editing a file that has not (yet) been formatted according to the active code convention.

Figure 7.1. Code editor pop-up menu item



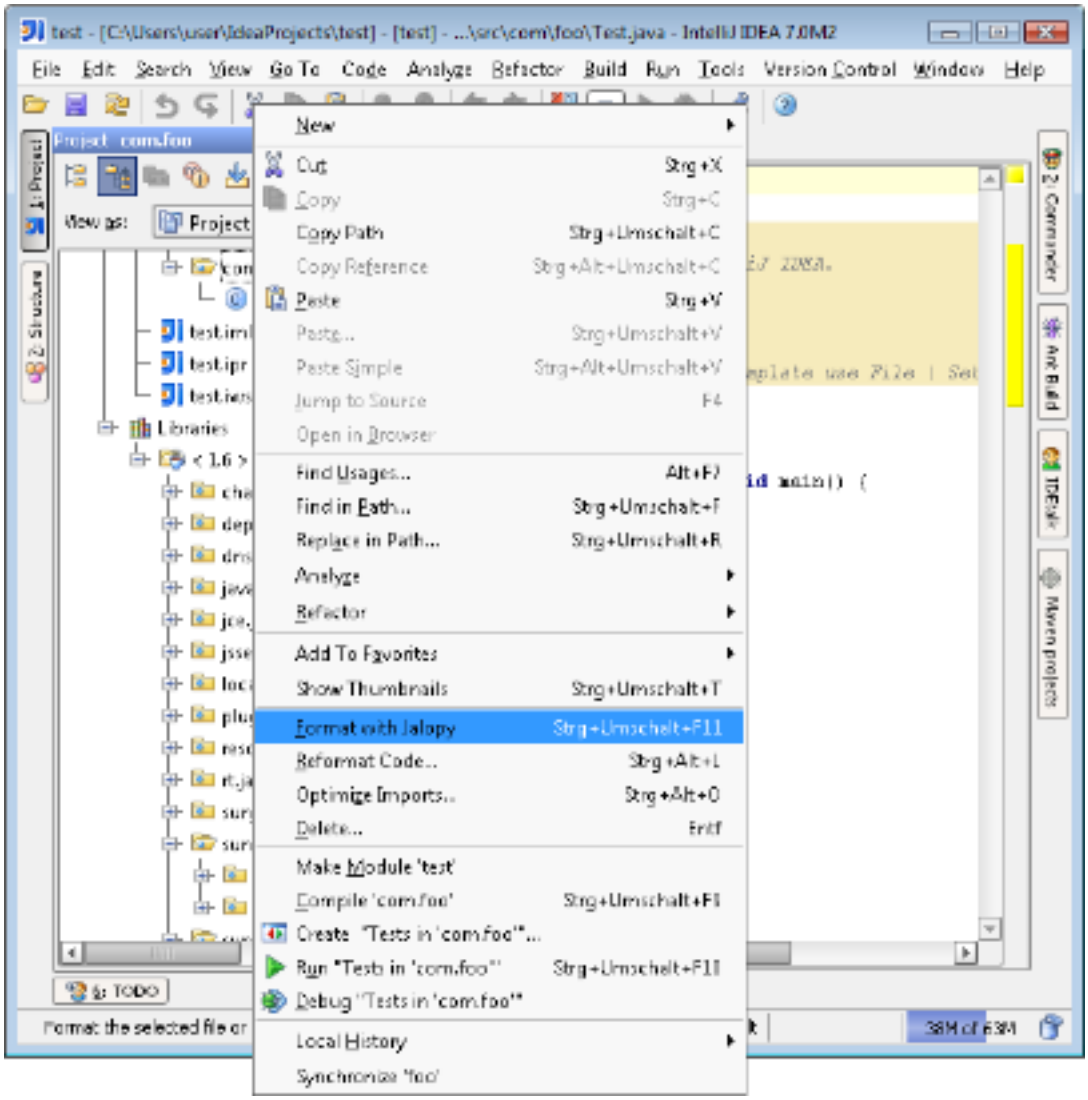
7.2.3 Tool Windows Popup Menu

The software adds a new menu item to the pop-up menu of certain Tool Windows. Currently it adds an item to the Project, Structure, Commander and Hierarchy Tool Window.

Format with Jalopy

Formats the selected files. Depending on the state either all Java source files of the project, the contents of the selected folder(s) (including subfolders) or the currently selected Java source file(s).

Figure 7.2. Tool window pop-up menu item

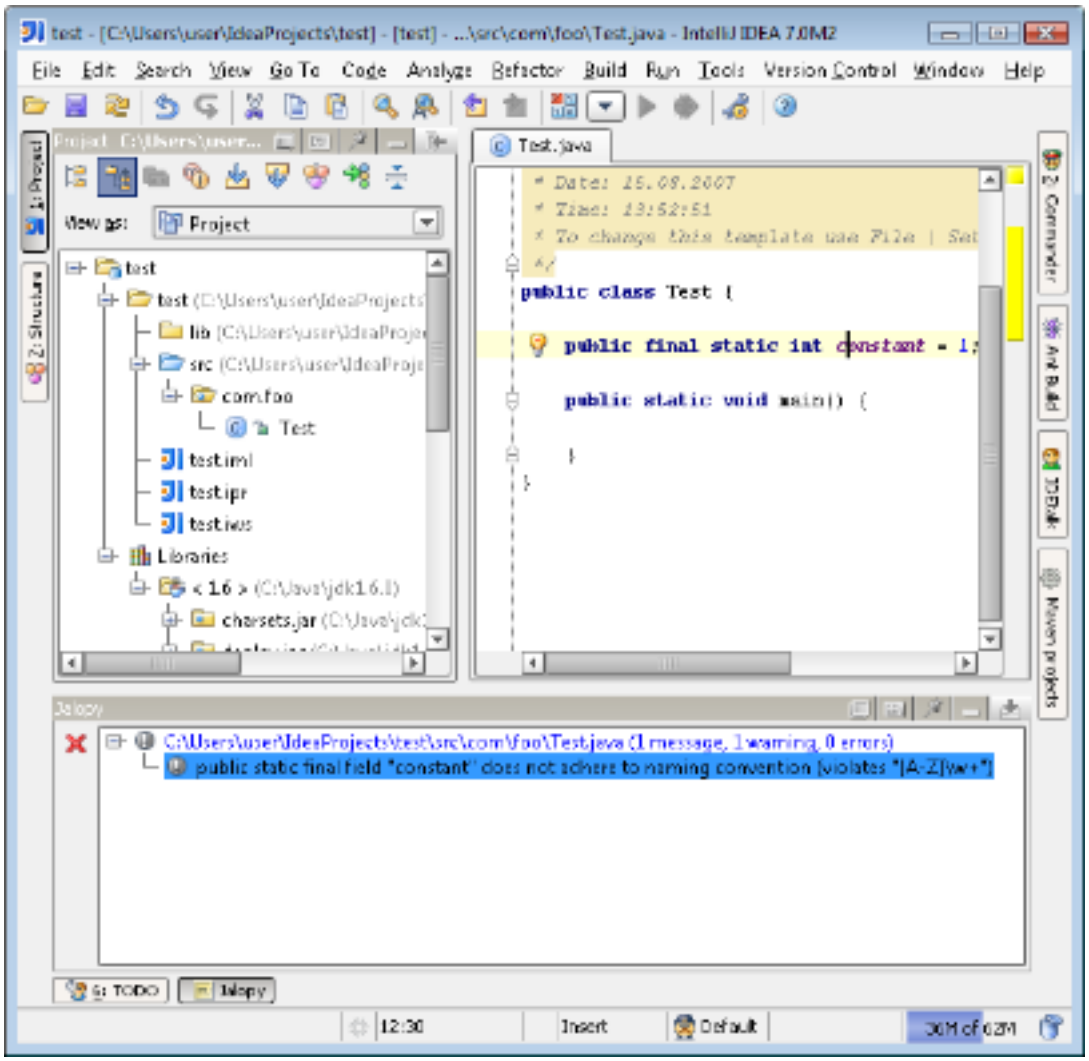


7.2.4 Tool window

Jalopy displays all runtime messages in its own tool window. Messages are shown in a tree control, with each branch containing the messages for a specific file, and individual messages displayed as leafs. File messages show the number of leaves and the warning and error count.

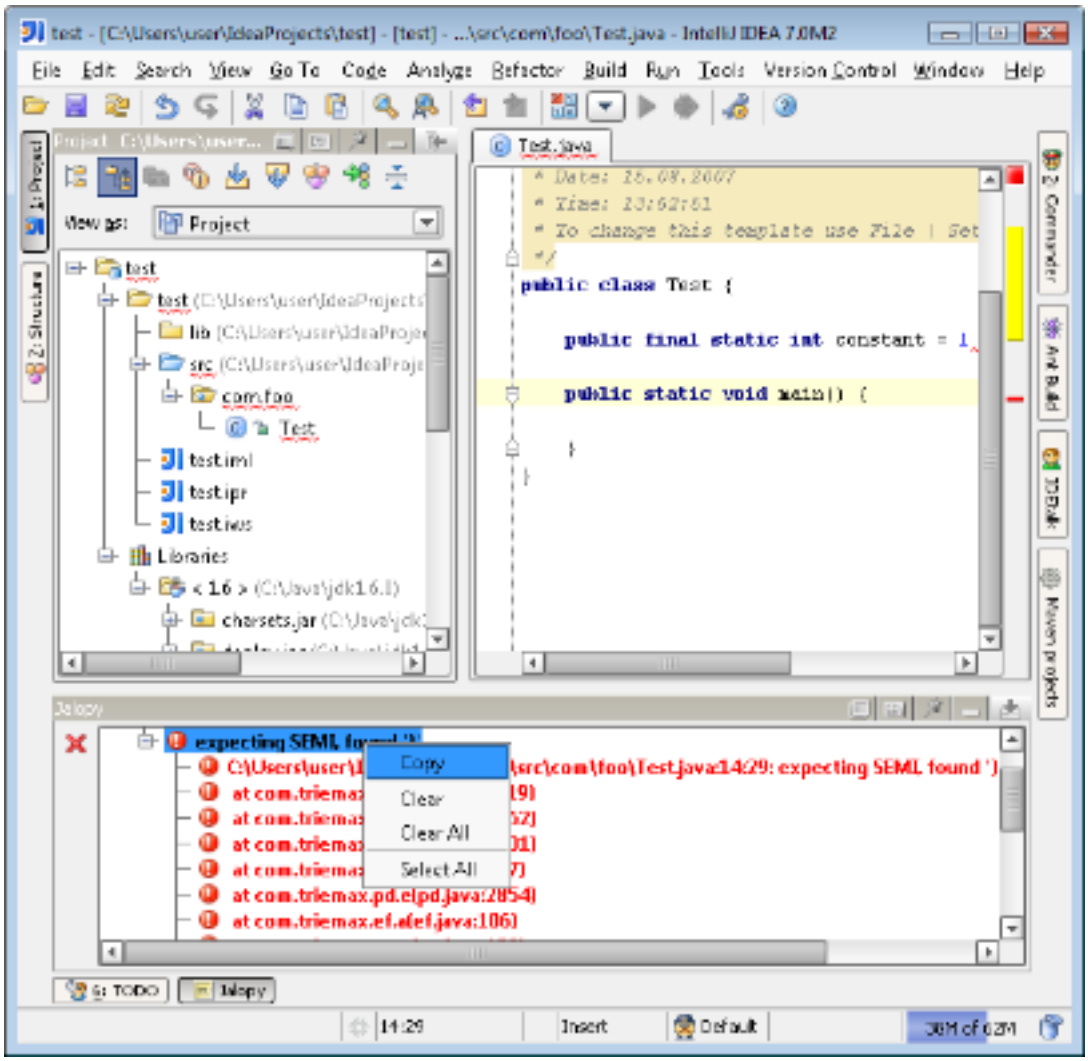
The message types are differentiated with icons and by color: Errors are red with an error icon, warnings are shown in blue and display a warning sign, informational messages are black and carry a file icon and debugging messages are black and prepended by a bug icon.

Figure 7.3. Jalopy Tool Window



Clicking on a file name will open that file, clicking on a message that contains location information will open the file containing the message and move the caret to the nominated location.

Figure 7.4. Jalopy Tool Window Context Menu



The window provides a context menu with some useful actions.

Copy	Copies the textual contents of the selected messages into the System clipboard. If a message contains children, the contents of all children are copied as well
Clear	Removes all selected messages
Clear All	Removes all messages currently being displayed in the window
Select All	Selects all messages currently being displayed in the window

7.3 Configuration

Although Jalopy ships with sensible default settings (mimicking the Sun Java coding convention), you most likely want to configure the formatter to match your needs (adding copyright headers, tune Javadoc handling and the like). For such, Jalopy comes with a graphical configuration tool that lets you interactively customize the settings. See Chapter 2, *Configuration* for an in-depth discussion of the available options.

Please refer to Section 7.2, "Integration" for information on how to display the configuration tool from within IDEA.

Chapter 8. JDeveloper Extension

Describes the installation and usage of the Jalopy JDeveloper Plug-in Extension. Oracle JDeveloper [Link] is an award-winning, comprehensive Java and Web services IDE. Optimized to run with Oracle Application Server and Oracle Database, JDeveloper is committed to open standards and platforms, supporting all major J2EE application servers and databases, and providing pure implementations for Struts, CVS, Ant and JUnit.

8.1 Installation

Explains the steps involved to install the JDeveloper Plug-in.

8.1.1 System requirements

The JDeveloper Plug-in requires JDeveloper 10g (9.0.5.1 - 10.1.2). See Section 1.1, "System requirements" for the basic requirements to run Jalopy.

8.1.2 Setup

The Plug-in comes as an executable Jar Archive (JAR) that contains a graphical setup wizard to let you easily install the software. Wizard installation is recommended and explained in detail in Section 1.3, "Wizard Installation".

If you would rather install the Plug-in manually, you have to decompress and copy the appropriate files into the different application and/or settings folders. To decompress the contents of the installer JAR, you can often use the build-in support of your file manager (e.g. Nautilus) or any other software that can handle the ZIP compression format (e.g. 7Zip, WinZip or Stuffit Expander). If you don't have access to one of the convenience tools, you might resort to the jar command-line application that ships with your Java distribution.

If you're upgrading from a prior version and want to keep your settings, first copy or rename the current Jalopy settings directory to match the version number of the new release. For instance, if your current settings directory is `C:\Documents and Settings\John Doo\.jalopy\1.9` and you're about to install Jalopy 1.9.3, either copy the directory contents or rename it to `C:\Documents and Settings\.jalopy\John Doo\1.9.3`. Wizard installation can perform this step automatically.

Make sure JDeveloper is not running and remove any prior Jalopy JAR files from your JDeveloper extension folder. The JDeveloper extension folder is located under the root directory of your JDeveloper installation, e.g. `C:\Program Files\JDeveloper\jdev\lib\ext`. Remove all JAR files starting with `jalopy-`. Now decompress the contents of the installer JAR file into a temporary directory and copy the two JAR files `jalopy-1.9.3.jar` and `jalopy-jdev-1.9.3.jar` from the temporary directory into the JDeveloper extension folder.

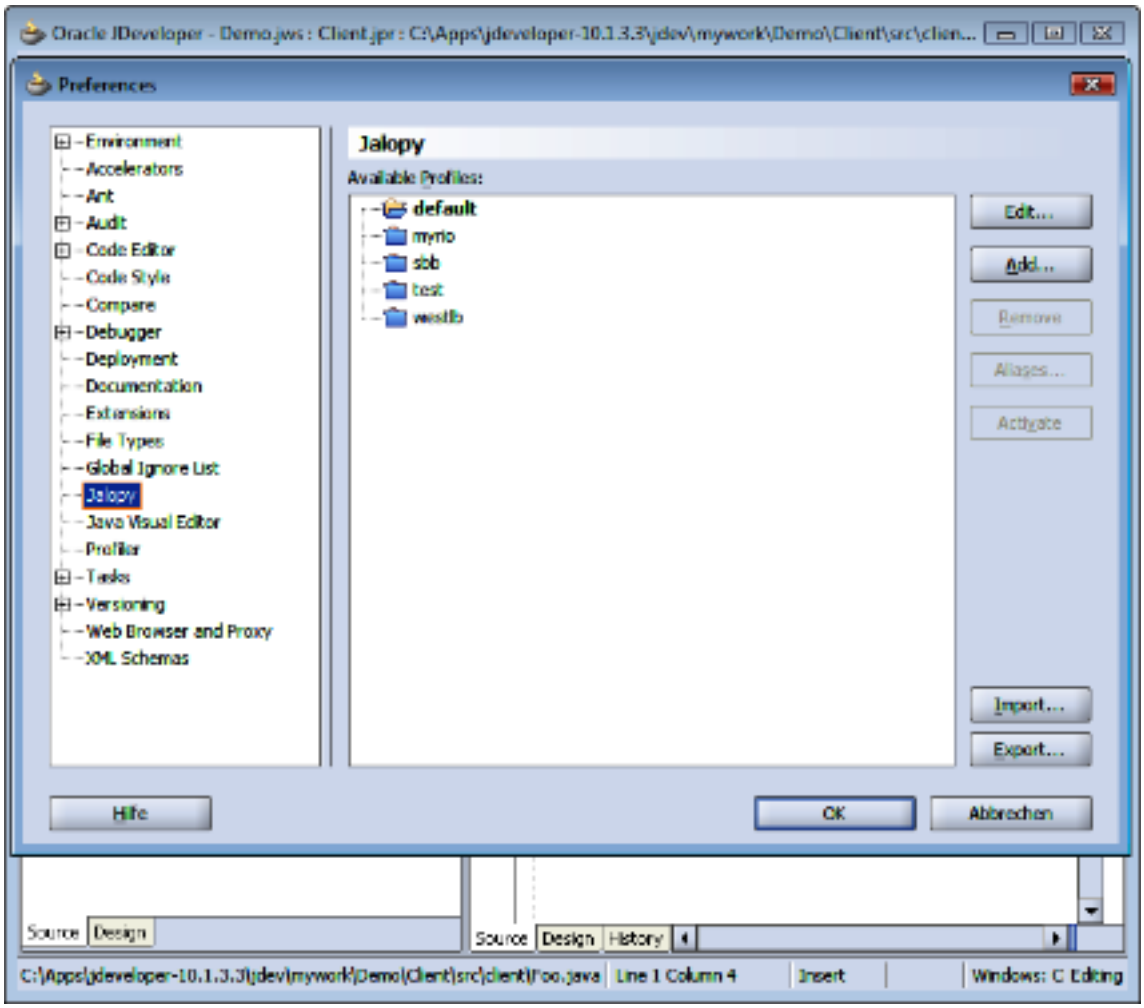
8.2 Integration

Describes how the Plug-in integrates into JDeveloper.

8.2.1 Preferences dialog

The Jalopy preferences are available through the JDeveloper preferences dialog. In order to access the preferences, on Mac OS X you use JDeveloper > Preferences... and select the Jalopy item on the left pane. On other platforms the dialog is available through Tools > Preferences....

Figure 8.1. Main Jalopy Preferences page



The main preferences page lets you manage your Jalopy profiles. A profile stores the actual code convention to define formatting output, as well as user-specific data like file and dialog histories. You can add, remove, activate, map and configure any number of profiles. For a detailed explanation of the available options, please refer to Section 2.1.1.1, “Main window”.

Please note that Jalopy does not store its preferences within the IDE configuration files, but uses its own provisions to store preferences in order to allow the resuse of Jalopy preferences between different IDEs.

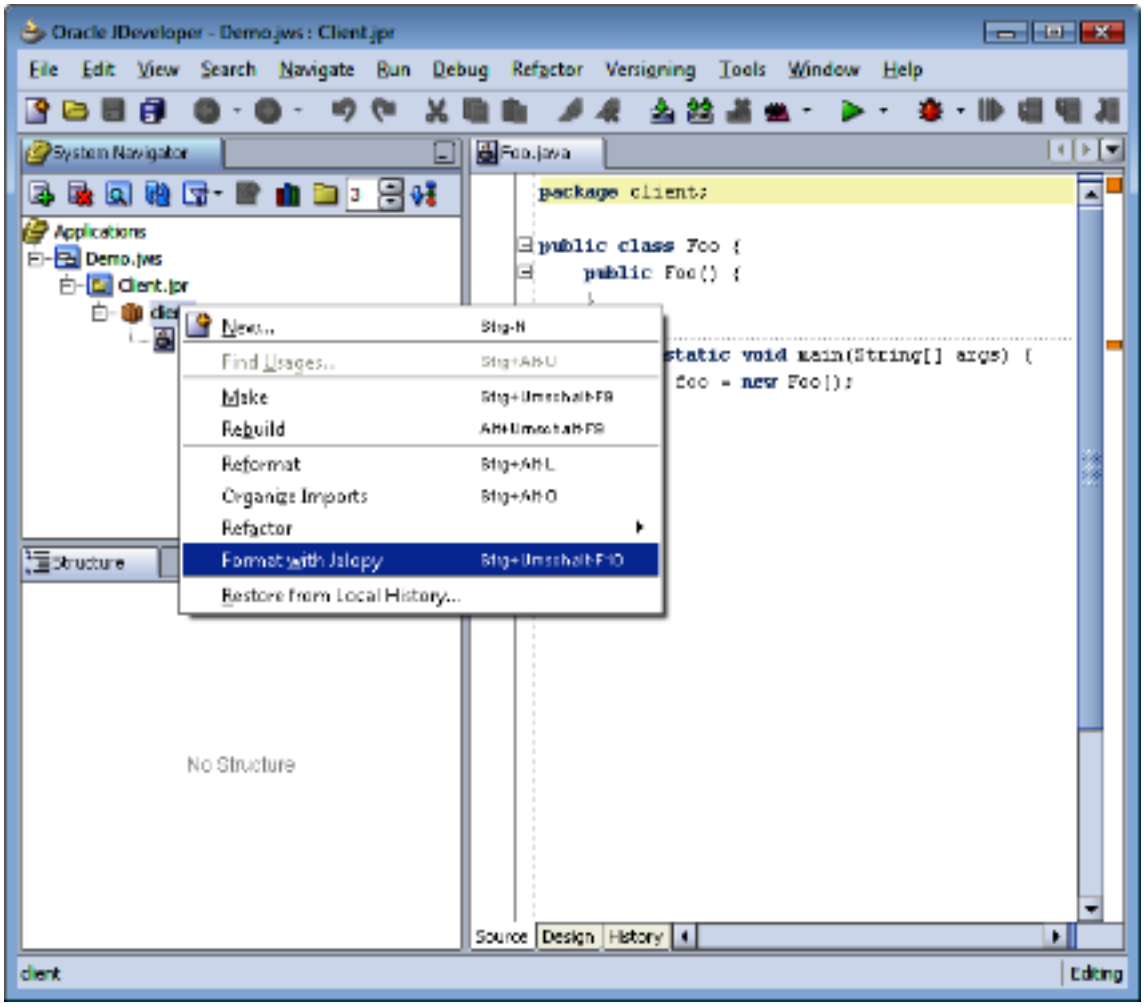
8.2.2 Navigator context menu

The software adds a new menu item into the context pop-up menu of the Navigator:

Format with Jalopy

By selecting the "Format with Jalopy" menu item, all Java sources of the selected node are formatted according to the current Jalopy preferences. The item appears in the pop-up menu when the right mouse button is clicked on a Java source node or any other parent node that may contain Java sources (such as Workspace, Project, Directory, EJB or BC4J nodes).

Figure 8.2. JDeveloper Navigator context menu



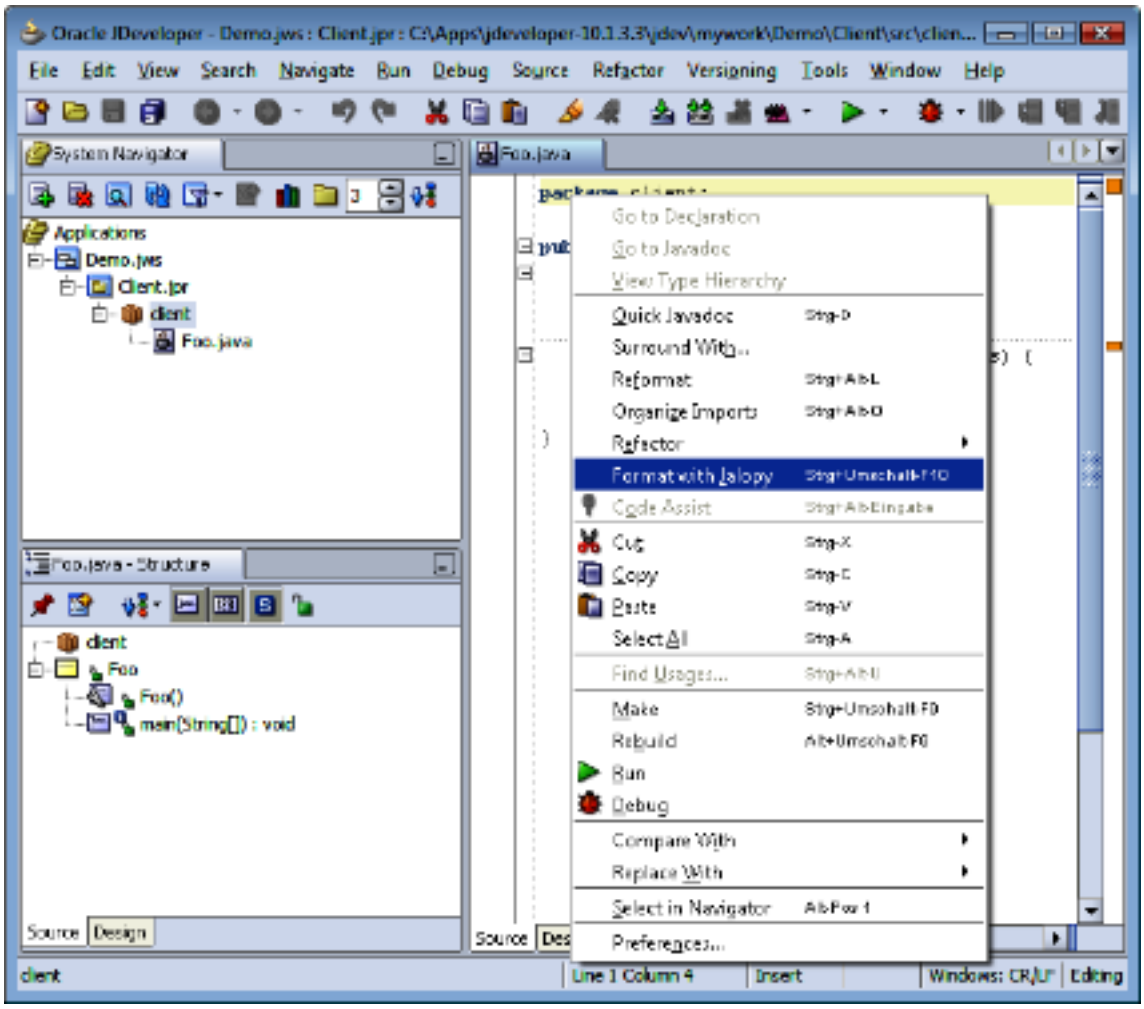
8.2.3 Editor context menu

The software adds a new menu item into the pop-up menu of Java code editors:

Format with Jalopy

Formats the contents of the editor. When currently some text is selected in the editor, only the selected text will be formatted (*selective formatting*). This can be especially helpful when editing portions of very large files, as selective formatting can speed up processing considerably. But comes especially when you want to limit formatting to a specific file portion in order to avoid unnecessary differences when editing a file that has not (yet) been formatted according to the active code convention.

Figure 8.3. JDeveloper Editor context menu

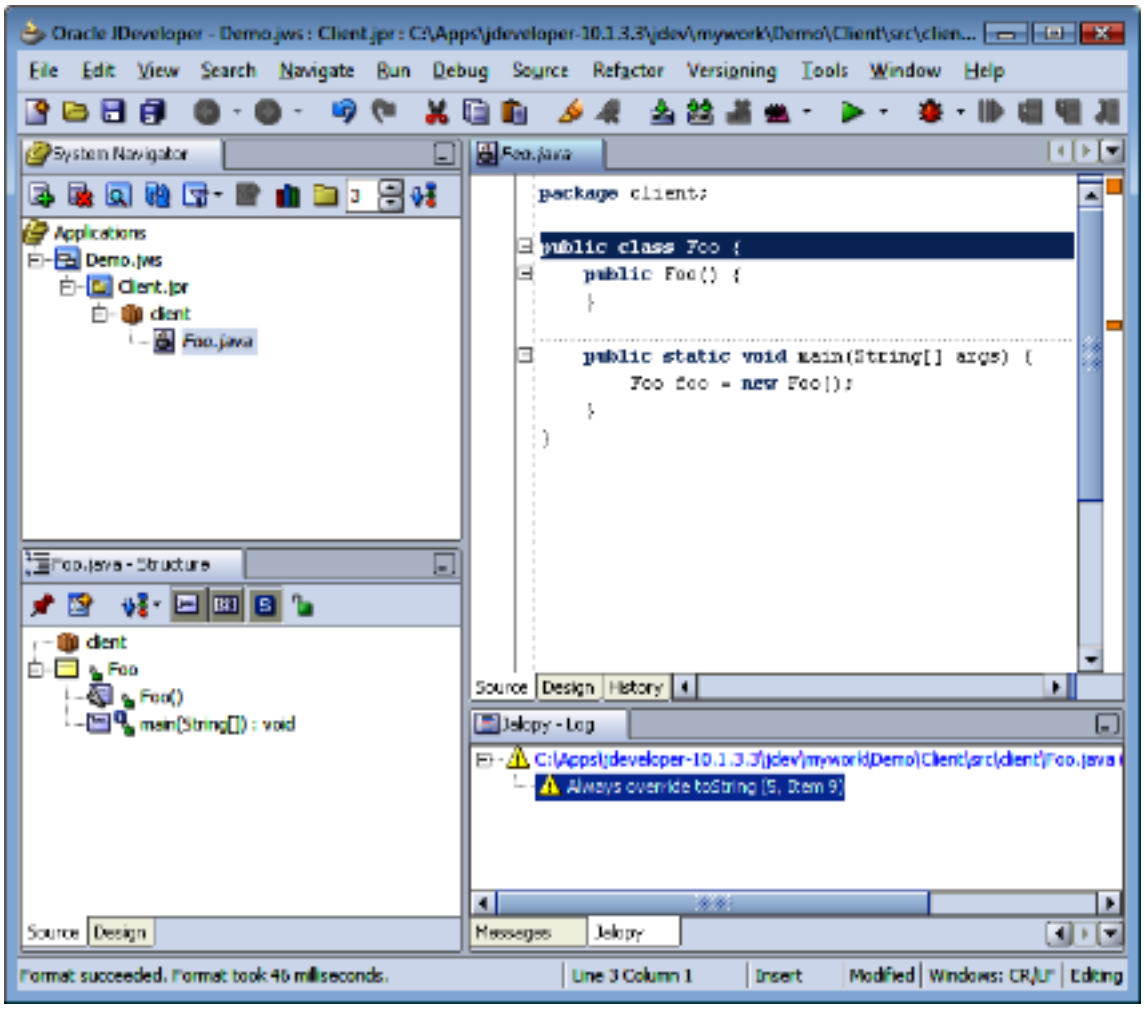


8.2.4 Log window

Jalopy displays all runtime messages in its own log window. Messages are shown in a tree control, with each branch containing the messages for a specific file, and individual messages displayed as leafs. File messages displays the number of leaves and the warning and error count.

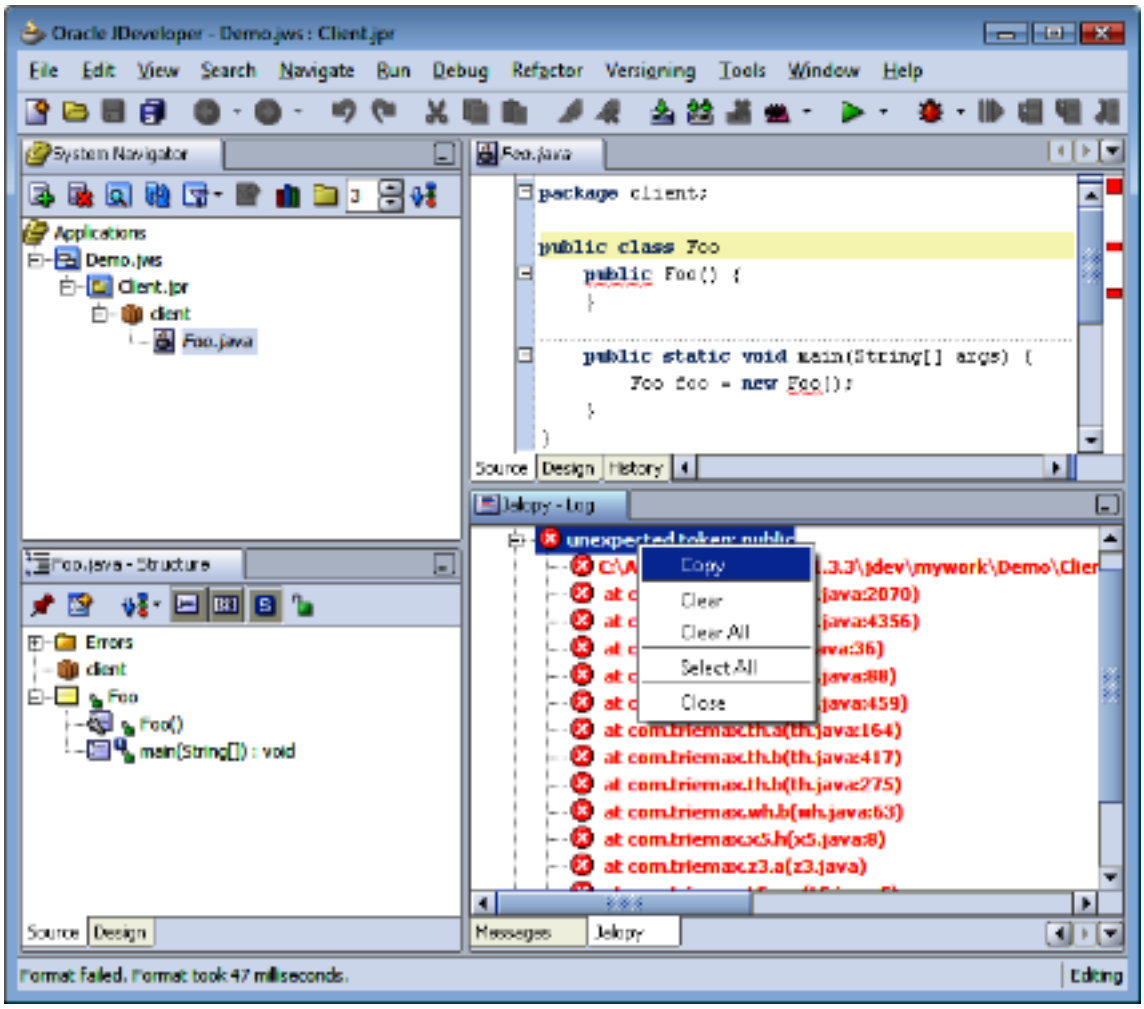
The message types are differentiated with icons and by color: Errors are red with an error icon, warnings are shown in blue and display a warning sign, informational messages are black and carry a file icon and debugging messages are black and prepended by a bug icon.

Figure 8.4. Jalopy Log Window



Clicking on a file name will open that file, clicking on a message that contains location information will open the file containing the message and move the caret to the nominated location.

Figure 8.5. Jalopy Log Window Context Menu



The window provides a context menu with some useful actions.

Copy	Copies the textual contents of the selected messages into the System clipboard. If a message contains children, the contents of all children are copied as well
Clear	Removes all selected messages
Clear All	Removes all messages currently being displayed in the window
Select All	Selects all messages currently being displayed in the window

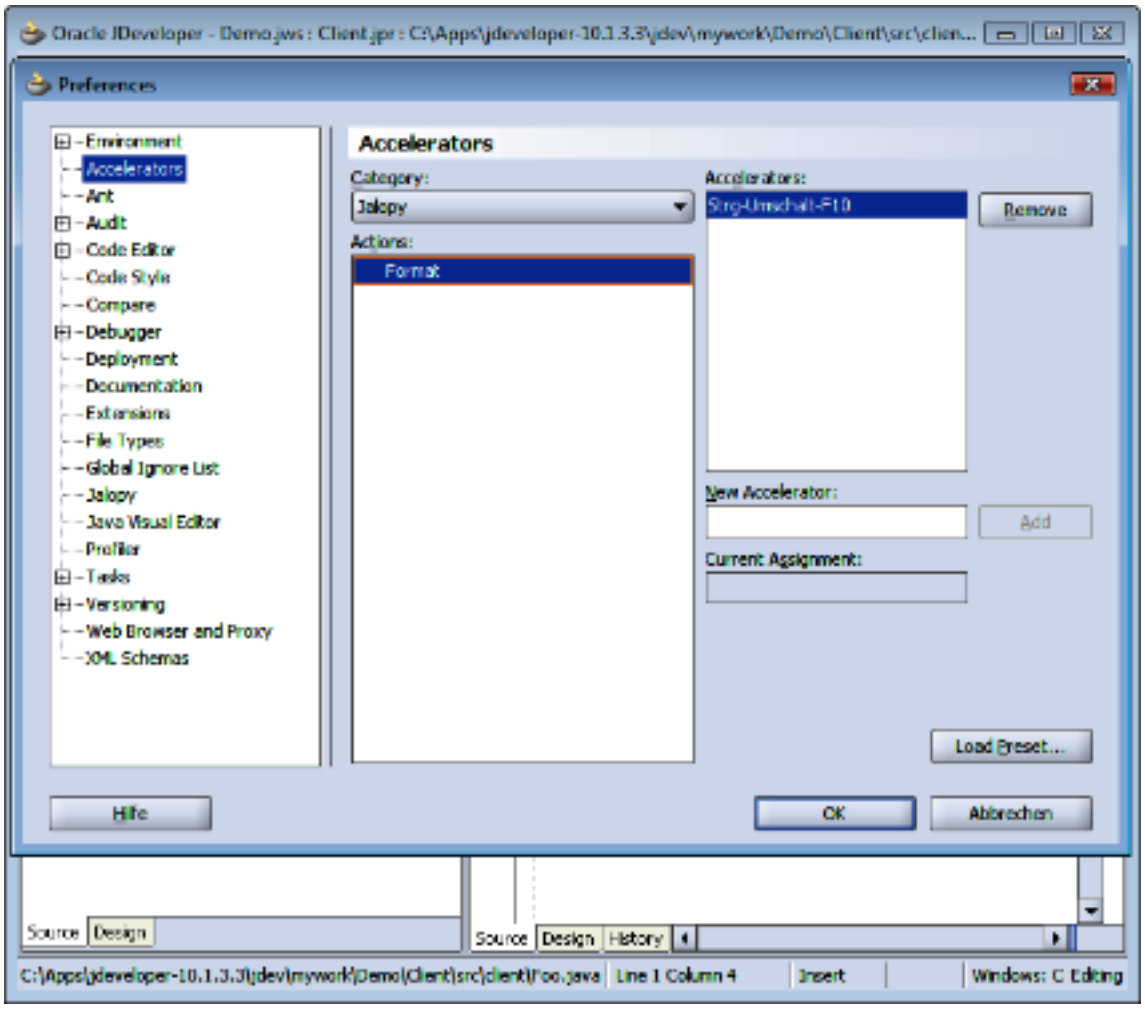
8.2.5 Keyboard Accelerator

The extension adds a new category to the JDeveloper accelerator preferences:

- Tools > Preferences... > Accelerators

  Select the "Jalopy" category and specify your preferred keyboard accelerator for the provided actions. The "Format" action is by default associated with `Strg+Shift+F10`.

Figure 8.6. JDeveloper accelerator preferences



Please note that the accelerators are global and the "Format" action considers the current application context, i.e. if the editor view has the focus, the accelerator triggers the formatting of the active editor. If the System Navigator contains the focus, the accelerator triggers the formatting of all selected nodes in the Navigator.

8.3 Configuration

Although Jalopy ships with sensible default settings (mimicking the Sun Java coding convention), you most likely want to configure the formatter to match your needs (adding copyright headers, tune Javadoc handling and the like). For such, Jalopy comes with a graphical configuration tool that lets you interactively customize the settings. See Chapter 2, *Configuration* for an in-depth discussion of the available options to configure formatting output. Please refer to Section 8.2.1, "Preferences dialog" for information on how to display the configuration tool from within JDeveloper.

Chapter 9. jEdit Plug-in

Describes the installation and usage of the Jalopy jEdit Plug-in. jEdit [Link] is a mature programmer's text editor written in Java that provides auto indent and syntax highlighting for more than 130 languages and is easily extensible with its Plug-in architecture.

9.1 Installation

Explains the steps involved to install the jEdit Plug-in.

9.1.1 System requirements

The Jalopy jEdit Plug-in requires jEdit 4.1 or later. See Section 1.1, "System requirements" for the basic requirements to run Jalopy.

9.1.2 Installation

The Plug-in comes as an executable Jar Archive (JAR) that contains a graphical setup wizard to let you easily install the software. Wizard installation is recommended and explained in detail in Section 1.3, "Wizard Installation".

If you would rather install the Plug-in manually, you have to decompress and copy the appropriate files into the different application and/or settings folders. To decompress the contents of the installer JAR, you can often use the build-in support of your file manager (e.g. Nautilus) or any other software that can handle the ZIP compression format (e.g. 7Zip, WinZip or Stuffit Expander). If you don't have access to one of the convenience tools, you might resort to the jar command-line application that ships with your Java distribution.

If you're upgrading from a prior version and want to keep your settings, first copy or rename the current Jalopy settings directory to match the version number of the new release. For instance, if your current settings directory is `C:\Documents and Settings\John Doo\.jalopy\1.9` and you're about to install Jalopy 1.9.3, either copy the directory contents or rename it to `C:\Documents and Settings\.jalopy\John Doo\1.9.3`. Wizard installation can perform this step automatically.

Make sure jEdit is not running and remove any prior Jalopy JAR files in your jEdit Plugin folder. The jEdit Plugin folder is located under the root directory of your jEdit installation, e.g. `C:\Program Files\jEdit\jars`. Remove all JAR files whose names start with `jalopy-`. Now decompress the contents of the installer JAR file into a temporary directory and copy the two JAR files `jalopy-1.9.3.jar` and `jalopy-jedit-1.9.3.jar` from the temporary directory into the jEdit Plugin folder.

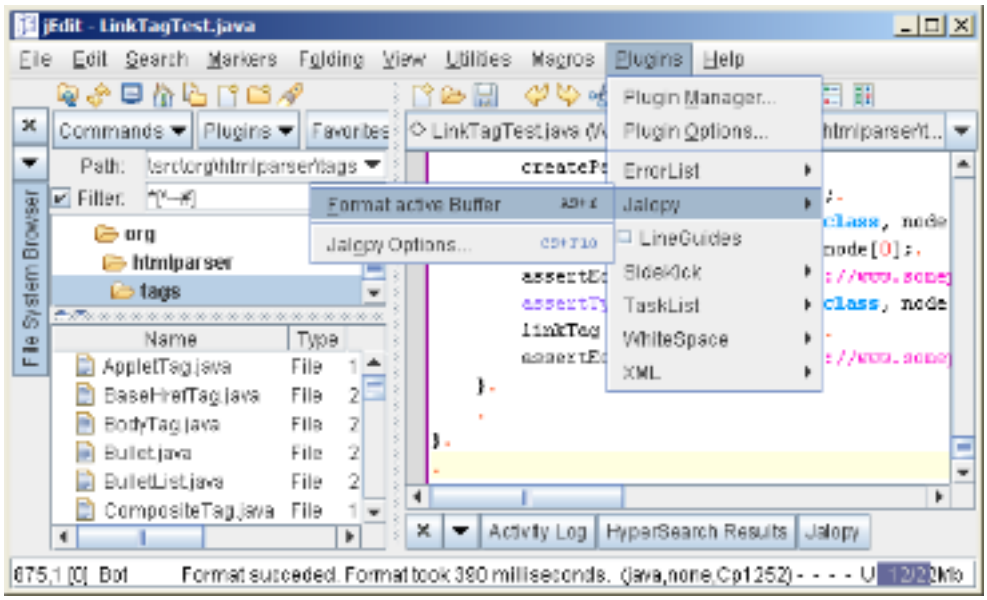
9.2 Integration

Describes how the Plug-in integrates into jEdit.

9.2.1 Menu bar

The software adds a new menu item group into the Plugins menu of the main view. Available are two new menu items:

Figure 9.1. jEdit menu bar items



- Plugins > Jalopy > Format active Buffer

  Formats the contents of the active text area. Note that this menu item reflects the state of the text area: it will only be enabled if the current edit mode is supported by Jalopy.

- Plugins > Jalopy > Jalopy Options....

  Displays the Jalopy options dialog. Use this item if you want to change your settings to control the layout of any formatted code.

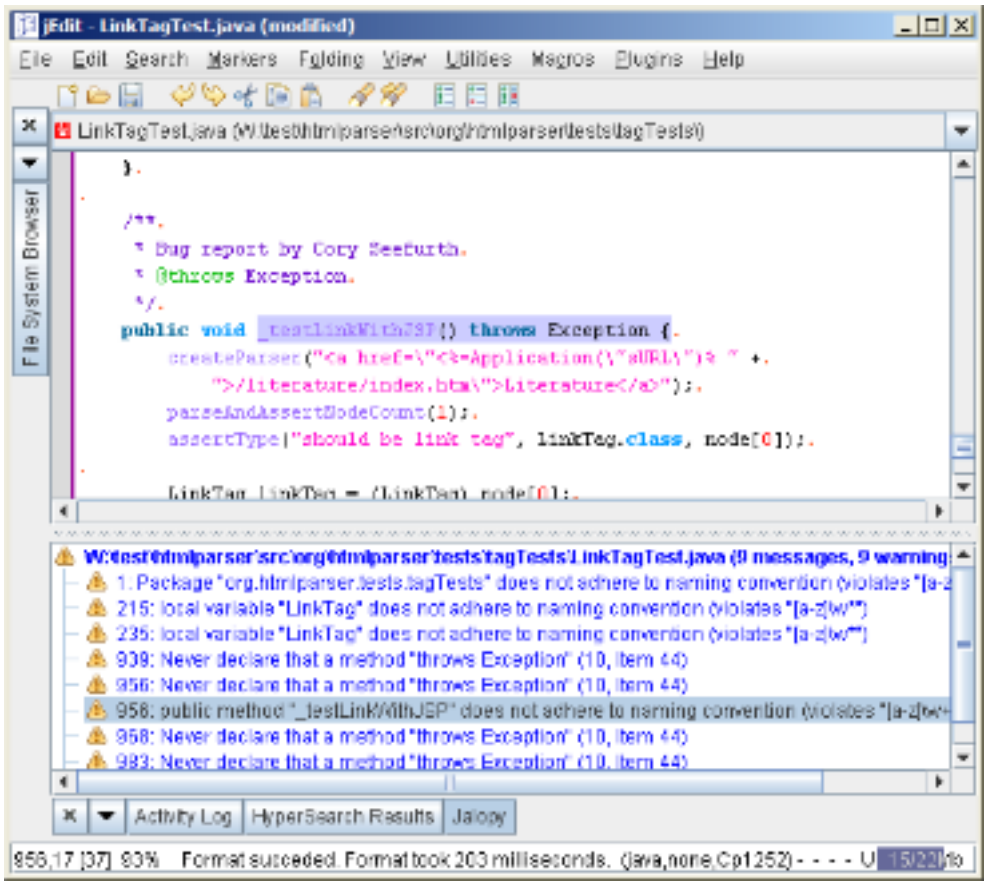
  Please note that all options are available under jEdit's Plugins > Plugin Options... dialog as well, but the Jalopy dialog provides the advantage of a live-preview that makes editing the options somewhat easier. You find the Jalopy settings system and options dialog described in detail in Chapter 2, *Configuration*.

9.2.2 Dockable window

Jalopy displays all runtime messages in its own dockable window that works similar like to the ErrorList Plug-in, but is not limited to just display errors and warnings. Messages are shown in a tree control, with each branch containing the messages for a specific file, and individual messages displayed as leafs. File messages display the number of leaves and the warning and error count.

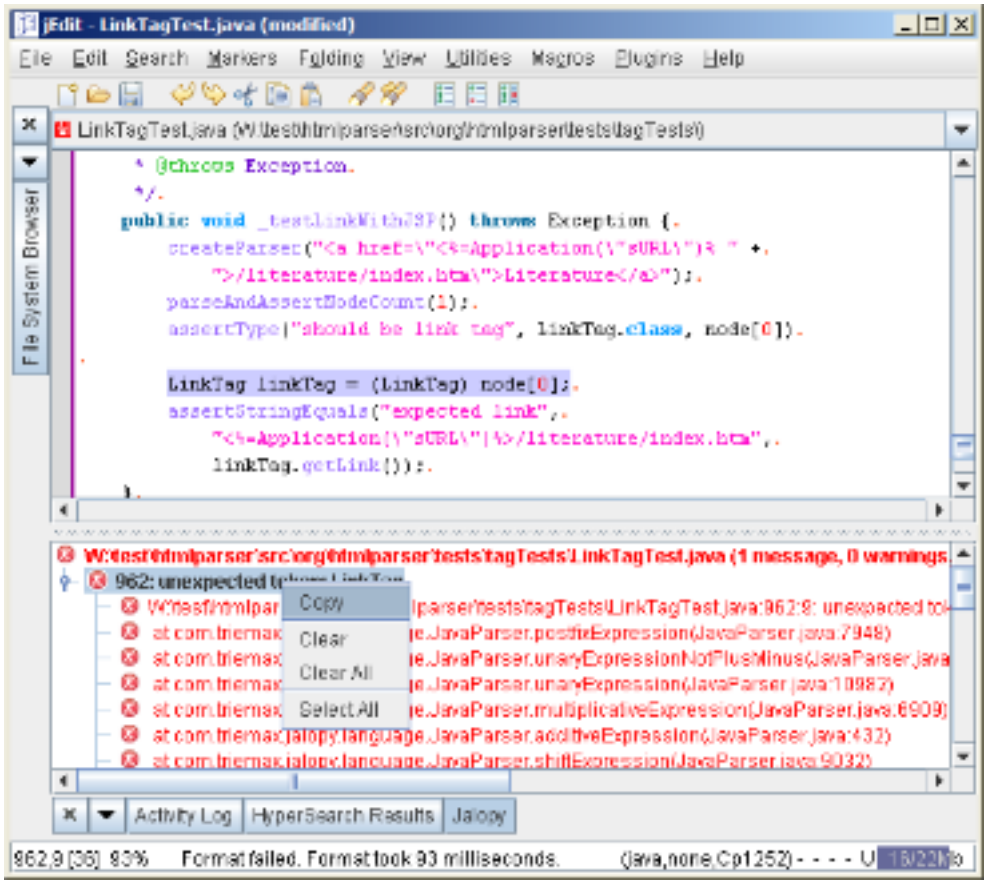
The message types are differentiated with icons and by color: Errors are red with an error icon, warnings are shown in blue and display a warning sign, informational messages are black and carry a file icon and debugging messages are black and prepended by a bug icon.

Figure 9.2. jEdit dockable window



Clicking on a file name will open that file, clicking on a message that contains location information will open the file containing the message and move the caret to the nominated location.

Figure 9.3. jEdit dockable window



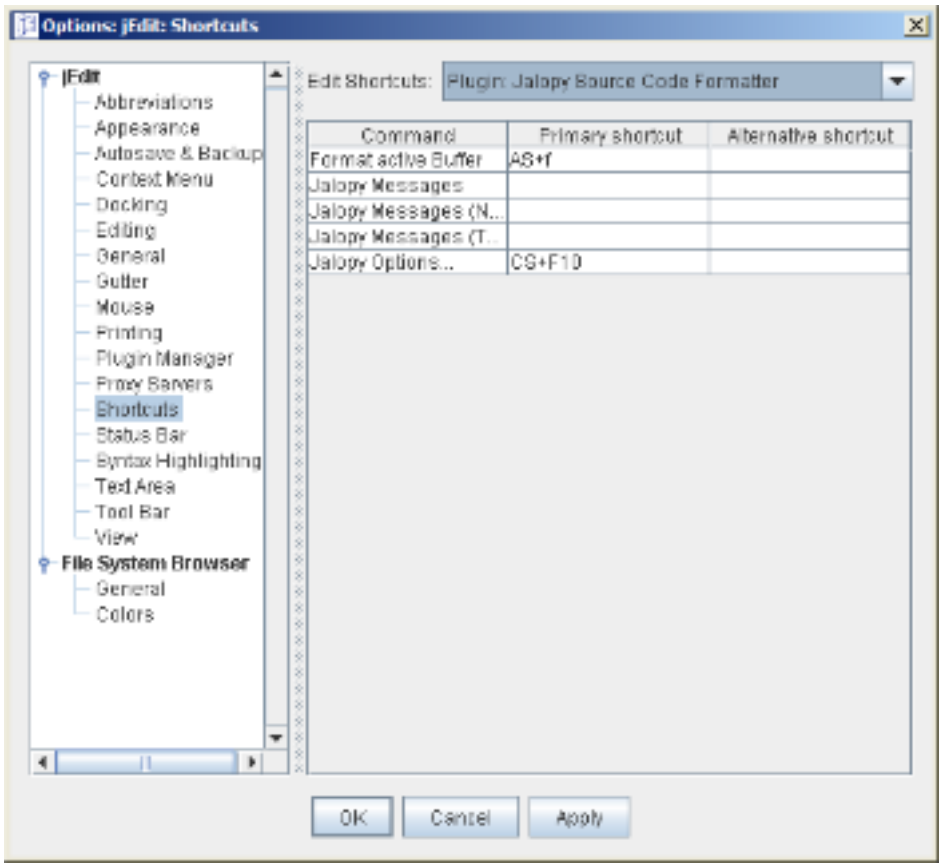
The window provides a context menu with some useful actions.

Copy	Copies the textual contents of the selected messages into the System clipboard. If a message contains children, the contents of all children are copied as well
Clear	Removes all selected messages
Clear All	Removes all messages currently being displayed in the window
Select All	Selects all messages currently being displayed in the window

9.2.3 Keyboard shortcuts

You can define keyboard shortcuts for the different menu and dockable window actions via the jEdit Shortcut options: Utilities > Global Options... > jEdit > Shortcuts. In the *Edit Shortcuts* combo box, select Plugin: Jalopy Source Code Formatter to display the available actions.

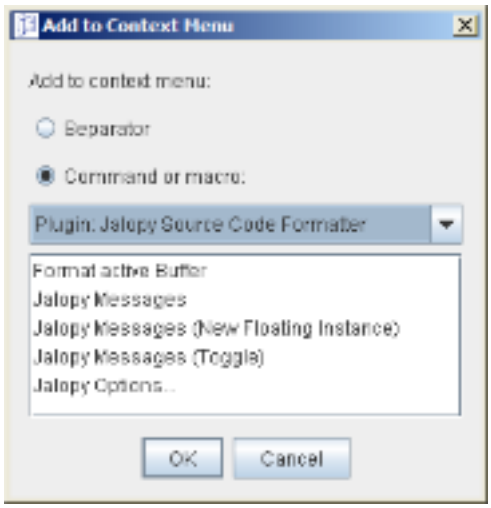
Figure 9.4. Define keyboard shortcuts



9.2.4 Context menu

You can add the different menu and dockable window actions to the context menu of the text area via the jEdit Context Menu options: Utilities > Global Options... > jEdit > Context Menu. Press the + button and, select the *Command or macro* option and choose Plugin: Jalopy Source Code Formatter to display the available actions.

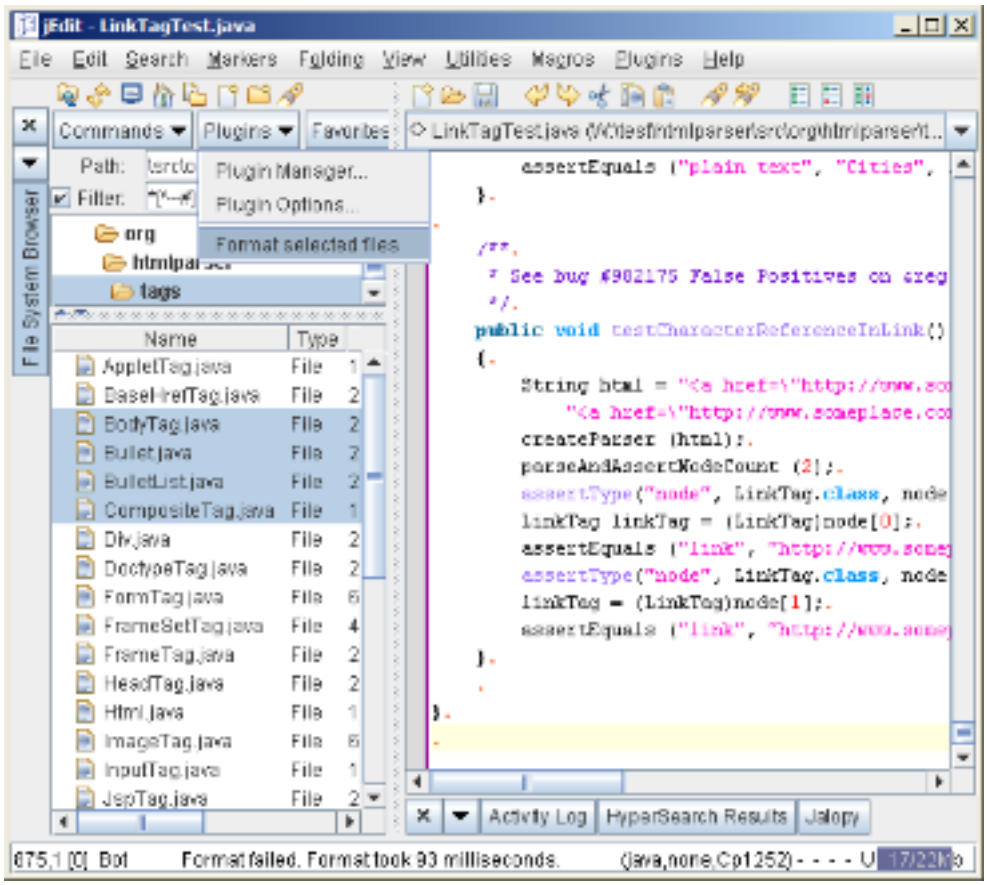
Figure 9.5. Add actions to context menu



9.2.5 File System Browser Plugins menu

Jalopy registers an action in the File System Browser Plugins menu to let you bulk format files and/or directories selected in the browser. Please note that if nothing is selected in the browser table component, the contents of the currently selected parent directory are formatted.

Figure 9.6. File System Browser Plugins menu



9.3 Configuration

Although Jalopy ships with sensible default settings (mimicking the Sun Java coding convention), you most likely want to configure the formatter to match your needs (adding copyright headers, tune Javadoc handling and the like). For such, Jalopy comes with a graphical configuration tool that lets you interactively customize the settings. See Chapter 2, *Configuration* for an in-depth discussion of the available options to configure formatting output. Please refer to Section 9.2, “Integration” for information on how to display the configuration tool from within jEdit.

Chapter 10. Maven 1 Plug-in

Describes the installation and usage of the Jalopy Maven 1 Plug-in. Maven [Link] is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

NOTE Maven 1 is in maintenance mode, i.e. development is restricted to support and bug fixes. You might be better of with the current release. Please refer to the main Maven site for further information

10.1 Installation

Explains the steps involved in getting the Maven 1 Plug-in up and running.

10.1.1 System requirements

The Plug-in requires Maven 1.0 - 1.1. See Section 1.1, "System requirements" for the basic requirements to run Jalopy. Please note that it won't work with later versions. A different Plug-in is available for more recent Maven versions (see Chapter 11, *Maven 2 Plug-in*).

10.1.2 Setup

The Plug-in comes as an executable Jar Archive (JAR) that contains a graphical setup wizard to let you easily install the software. Wizard installation is recommended and explained in detail in Section 1.3, "Wizard Installation".

If you would rather install the Plug-in manually, you have to decompress and copy the appropriate files into the different application and/or settings folders. To decompress the contents of the installer JAR, you can often use the build-in support of your file manager (e.g. Nautilus) or any other software that can handle the ZIP compression format (e.g. 7Zip, WinZip or Stuffit Expander). If you don't have access to one of the convenience tools, you might resort to the jar command-line application that ships with your Java distribution.

If you're upgrading from a prior version and want to keep your settings, first copy or rename the current Jalopy settings directory to match the version number of the new release. For instance, if your current settings directory is `C:\Documents and Settings\John Doo\.jalopy\1.9` and you're about to install Jalopy 1.9.3, either copy the directory contents or rename it to `C:\Documents and Settings\.jalopy\John Doo\1.9.3`. Wizard installation can perform this step automatically.

Remove any `jalopy-1.9.3.jar` files from the `/lib` and `/plugins` directories of your Maven installation, e.g from `/home/John Doo/apps/maven-1.0.2/lib/` and `/home/John Doo/apps/maven-1.0.2/plugins/`.

Copy the files `jalopy-1.9.3.jar` and `jalopy-ant-1.9.3.jar` from the temporary directory into the `/lib` folder of your Maven installation. If you don't have the Console Plug-in installed and want to be able to configure Jalopy from the command-line, copy the contents of the `/bin` folder from the temporary directory to the `/bin` folder of your Maven installation.

As a last step, copy the file `jalopy-maven-1.9.3.jar` from the temporary directory into the `/plugins` folder of your Maven installation.

10.2 Configuration

Although Jalopy ships with sensible default settings (mimicking the Sun Java coding convention), you most likely want to configure the formatter to match your needs (adding copyright headers, tune Javadoc handling and the like). For such, Jalopy comes with a graphical configuration tool that lets you interactively customize the settings. See Chapter 2, *Configuration* for an in-depth discussion of the available options.

To display the configuration tool, you should use the matching wrapper script for your platform. The wrapper scripts are called `jalopy.xxx` . Invoke the script with the `--configure` option.

```
% jalopy --configure
```

If you don't want to install the Console Plug-in, you can make use of the -jar option of the Java launcher, as Jalopy comes as an executable JAR file:

```
% java -jar <path_to>\jalopy-1.9.3.jar --configure
```

Or you give the class path directly to the launcher

```
% java -cp <path_to>\jalopy-1.9.3.jar Jalopy --configure
```

When you're done configuring the settings, you should export the code convention as described in Section 2.1.1.8, "Export code convention". The exported settings file is typically used as part of the Jalopy task configuration in the build script.

10.2.1 Properties

The Plug-in allows some optional properties to control how formatting is applied.

Table 10.1. Jalopy Maven Plug-in properties

Property	Type	Description	Since	Required
maven.jalopy.backup	Boolean	Sets whether backup copies of all processed source files should be kept. If omitted, the corresponding code convention setting will be used (see Section 2.2.2.2, "Backup").	1.5	No
maven.jalopy.convention	String	Sets the location to the code convention file to use - given either relative to the project's base directory or as an absolute local path or Internet address (refer to Section 2.1.1.7, "Import code convention" for information how to export your settings). If omitted, the current settings are used, if available. Otherwise the Jalopy build-in defaults will be used.	1.5	No
maven.jalopy.destdir	String	Sets the destination directory to create/copy all formatting output into. It can either be given as an absolute path, or relative to the working directory. If the directory does not exist, it will be created. If omitted, all input files will be overridden.	1.5	No
maven.jalopy.encoding	String	Sets the encoding that controls how Jalopy interprets text files containing characters be-	1.5	No

Property	Type	Description	Since	Required
		yond the ASCII character set. Defaults to the platform default encoding.		
maven.jalopy.failOnError	Boolean	Sets whether a run should be held if errors occurred. Defaults to "true".	1.5	No
maven.jalopy.fileFormat	String	Sets the file format of the output files. The file format controls what end of line character is used. Either one of "UNIX", "DOS", "DEFAULT" or "AUTO" can be used (case insensitive). Defaults to "AUTO".	1.5	No
maven.jalopy.filesetInclude	String	Comma- or space-separated list of patterns of source files that should be formatted. Defaults to "**/*.java".	1.5	No
maven.jalopy.filesetExclude	String	Comma- or space-separated list of patterns of source files that should be excluded from formatting; no files (except default excludes) are excluded when omitted. The default is to format all source files.	1.5	No
maven.jalopy.force	Boolean	Sets whether the formatting of files should be forced, even if a file is up-to-date. Defaults to "false".	1.5	No
maven.jalopy.fork	Boolean	Sets whether the processing should be performed in a separate VM. Defaults to "false".	1.5	No
maven.jalopy.history	String	Sets the history policy to use. Either one of "ADLER32", "CRC32" or "NONE" can be used (case insensitive). If omitted, the corresponding code convention setting will used (see Section 2.2.2.1, "History").	1.5	No
maven.jalopy.inputEncoding	String	Sets the encoding that controls how Jalopy interprets text files containing characters beyond the ASCII character set. Defaults to the platform default encoding. Please note that this setting always overrides *encoding*.	1.6	No
maven.jalopy.javadoc	String	Indicates whether Javadoc related messages should be printed. Defaults to "true".	1.5	No
maven.jalopy.logLevel	String	Specifies the logging level for message output. Either one of "ERROR", "WARN", "INFO" or "DEBUG" can be used (case insensitive). If omitted, the current code convention settings will be used (see Section 2.6.1, "Categories").	1.5	No
maven.jalopy.log	String	Specifies the log file to use for logging output. The format of the logging output is determined by the extension of the given file. Valid extensions are `".log"` for a custom plain text format, `".xml"` for a plain XML format and `".html"` for an hierarchical HTML report. If omitted, the current code convention setting will be used (see Section 2.6.2, "Logging").	1.5	No
maven.jalopy.outputEncoding	String	Sets the character encoding Jalopy uses to write files. Defaults to the platform default encoding. Please note that this setting always overrides *encoding*.	1.6	No
maven.jalopy.profile	String	Sets the Jalopy profile that should be activated during the formatting run (refer to Section 2.1.1.1, "Main window" for more information about profiles). The currently active profile will be restored after formatting. Please note that the profile must exist!	1.5	No

Property	Type	Description	Since	Required
maven.jalopy.repository	Boolean	Indicates whether the type repository should be used for type lookup. When disabled, this currently means that all dependent features despite the import optimization will be disabled! You may want to use this option if you commonly format a single file or only a small sets of files in order to avoid the maintenance overhead of the type repository. Defaults to "true".	1.6	No
maven.jalopy.src.filesetInclude	Boolean	For "src/java" directory. Comma- or space-separated list of patterns of source files that should be formatted. Defaults to "${maven.jalopy.filesetInclude}".	1.5	No
maven.jalopy.src.filesetExclude	Boolean	For "src/java" directory. Comma- or space-separated list of patterns of source files that should be excluded from formatting. Defaults to "{maven.jalopy.filesetExclude}".	1.5	No
maven.jalopy.test	Boolean	Sets whether formatting output should actually be written to disk. If set to "true" no output will be written to disk. The default is "false".	1.5	No
maven.jalopy.test.filesetInclude	Boolean	For "src/test" directory. Comma- or space-separated list of patterns of source files that should be formatted. Defaults to "${maven.jalopy.filesetInclude}".	1.5	No
maven.jalopy.test.filesetExclude	Boolean	For "src/test" directory. Comma- or space-separated list of patterns of source files that should be excluded from formatting. Defaults to "${maven.jalopy.filesetExclude}".	1.5	No
maven.jalopy.threads	Integer	Specifies the number of processing threads to use. Integer between `1` - `8`. Defaults to '1'.	1.5	No

10.3 Usage

The Jalopy Plug-in provides a standard goal to format your sources. For example, to format all source files from the current project, run:

```
% maven triemax-jalopy
```

You'll notice that all of the code is compiled before any formatting is applied. This is good practice in order to ensure valid input. If you want to bypass the compilation, you can use another goal:

```
% maven triemax-jalopy:format
```

10.3.1 Goals

Table 10.2. Jalopy Maven Jelly goals

Goal	Description
triemax-jalopy	Formats the source files according to coding convention. The source files will be compiled before formatting takes place.
triemax-jalopy:format	Formats the source files according to coding convention.
triemax-jalopy:taskdef	Defines the Jalopy task to Ant and Jelly.

Chapter 11. Maven 2 Plug-in

Describes the installation and usage of the Jalopy Maven 2 Plug-in. Maven [Link] is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

11.1 Installation

Explains the steps involved in getting the Maven 2 Plug-in up and running.

11.1.1 System requirements

The Plug-in requires Maven 2.0 or later. See Section 1.1, "System requirements" for the basic requirements to run Jalopy. Please note that a different Plug-in is available for older Maven releases (see Chapter 10, *Maven 1 Plug-in*).

11.1.2 Setup

The Plug-in comes as an executable Jar Archive (JAR) that contains a graphical setup wizard to let you easily install the software. Wizard installation is recommended and explained in detail in Section 1.3, "Wizard Installation".

If you would rather install the Plug-in manually, you have to decompress and copy the appropriate files into the different application and/or settings folders. To decompress the contents of the installer JAR, you can often use the build-in support of your file manager (e.g. Nautilus) or any other software that can handle the ZIP compression format (e.g. 7Zip, WinZip or Stuffit Expander). If you don't have access to one of the convenience tools, you might resort to the jar command-line application that ships with your Java distribution.

If you're upgrading from a prior version and want to keep your settings, first copy or rename the current Jalopy settings directory to match the version number of the new release. For instance, if your current settings directory is `C:\Documents and Settings\John Doo\.jalopy\1.9` and you're about to install Jalopy 1.9.3, either copy the directory contents or rename it to `C:\Documents and Settings\.jalopy\John Doo\1.9.3`. Wizard installation can perform this step automatically.

First copy the Jalopy Maven Plug-in folder `triemax` from the temporary directory into the Maven 2 repository directory. This might either be the local repository (e.g. `C:\Documents And Settings\John Doo\.m2\repository\` on a Windows XP system) or the repository directory of your internal repository in case you're using a shared repository for your organization.

Copy the file `jalopy-1.9.3.jar` below the `triemax/jalopy/1.9.3/` folder in your Maven repository, and `jalopy-maven-1.9.3.jar` from the temporary directory into the `triemax/jalopy-maven/1.9.3/` folder of your Maven 2 repository. After all steps have been performed, you should have a folder structure like the following:

```
..
  repository
    [...]
    triemax
      jalopy
        1.9.3-156
          jalopy-1.9.3-156.jar
          jalopy-1.9.3-156.pom
      jalopy-maven
        1.9.3-156
          jalopy-maven-1.9.3-156.jar
          jalopy-maven-1.9.3-156.pom
          maven-metadata-local.xml
        maven-metadata-central.xml
    [...]
```

11.2 Configuration

Although Jalopy ships with sensible default settings (mimicking the Sun Java coding convention), you most likely want to configure the formatter to match your needs (adding copyright headers, tune Javadoc handling and the like). For such, Jalopy comes with a graphical configuration tool that lets you interactively customize the settings. See Chapter 2, *Configuration* for an in-depth discussion of the available options.

To display the configuration tool, you can use the *configure* goal from within Maven. Either using the complete notation

```
% mvn triemax:jalopy-maven:configure
```

or just the shorthand

```
% mvn jalopy:configure
```

When you're done configuring the settings, you should export the code convention as described in Section 2.1.1.8, "Export code convention". The exported settings file is typically used as part of the Jalopy plugin configuration in the build script.

11.3 Usage

In order to integrate Jalopy into your build process, you need to edit your `pom.xml` under the *plugins* section:

```
<project>
  [...]
  <plugins>
    <plugin>
      <groupId>triemax</groupId>
      <artifactId>jalopy-maven</artifactId>
      <configuration>
        [...]
      </configuration>
      <executions>
        <execution>
          <phase>process-classes</phase>
          <goals>
            <goal>format</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    [...]
  </plugins>
</project>
```

This would execute Jalopy during the *process-classes* phase. Please note that you should always apply formatting after the *compile* phase has finished in order to ensure that the project is clean, but you're otherwise free to choose the phase that best suits your needs. For more information about the Maven build lifecycle, please refer to the "Maven Build Lifecycle Guide". It contains a reference of the available build phases. If you don't want to have the format goal execute during the build, you simply don't bind the goal to a specific phase:

```
<project>
  [...]
  <plugins>
    <plugin>
      <groupId>triemax</groupId>
      <artifactId>jalopy-maven</artifactId>
      <configuration>
        [...]
      </configuration>
    </plugin>
    [...]
  </plugins>
</project>
```

Formatting can be manually triggered from the command-line using either

```
% mvn triemax:jalopy-maven:format
```

or the shorthand

```
% mvn jalopy:format
```

Configuration

Naturally the Plug-in provides a few parameters to configure its behavior.

Table 11.1. Jalopy Maven Plug-in parameters

Property	Type	Description	Since
backup	Boolean	Sets whether backup copies of all processed source files should be kept. When omitted, the corresponding code convention setting will be used (see Section 2.2.2.2, "Backup")	1.7

Property	Type	Description	Since
classpathElements	List	Defines the class path to use for type lookup. By default, the project class path ${project.compileClasspathElements} is used	1.7
convention	String	Sets the location of the code convention file to use - given either relative to the project's base directory or as an absolute local path or Internet address (refer to Section 2.1.1.7, "Import code convention" for information how to export your settings). When omitted and not specified otherwise (see "profile" below), the settings of the current profile are used Since Jalopy 1.9.3, it's also possible to load conventions from the class path. This can be achieved with the following syntax: <convention>classpath:[path]<convention> where [path] denotes the relative path to the resource in the artifact, e.g. jalopy.xml if the settings file can be found at the root level, or config/jalopy/sonar.xml when located in a nested folder	1.7
destDir	String	Sets the destination directory to create/copy all formatting output into. If the given directory does not exist, it will be created. When omitted, all input files will simply be overridden	1.7
encoding	String	Sets the encoding that controls how Jalopy interprets text files containing characters beyond the ASCII character set. Defaults to the platform default encoding	1.7
environment	Map	Defines temporary environment variables overrides	1.7
excludes	List	A list of exclusion filters. Uses the standard Maven pattern syntax. Please note that Jalopy ignores all files it cannot format by default, so exclusions are only necessary if you want to omit formatting for certain files, like e.g. test data files etc.	1.7
failOnError	Boolean	Sets whether a run should be held if errors occurred. Defaults to "true"	1.7
fileFormat	String	Sets the file format of the output files. The file format controls what end of line character is used. Either one of "UNIX", "DOS", "DEFAULT" or "AUTO" can be used (case insensitive). Defaults to "AUTO"	1.7
force	Boolean	Sets whether the formatting of files should be forced, even if a file is up-to-date. Defaults to "false"	1.7
fork	Boolean	Sets whether the processing should be performed in a separate VM. Defaults to "false"	1.7
history	String	Sets the history policy to use. Either one of "ADLER32", "CRC32" or "NONE" can be used (case insensitive). If omitted, the corresponding code convention setting will used (see Section 2.2.2.1, "History")	1.7
includes	List	A list of inclusion filters for formatting. Uses the standard Maven pattern syntax. Please note that Jalopy ignores all files it cannot format by default, so inclusions are only necessary if you want to omit formatting for certain files, like e.g. test data files etc.	1.7
inputEncoding	String	Sets the encoding that controls how Jalopy interprets text files containing characters beyond the ASCII character set. Defaults to the platform default encoding. Please note that this setting always overrides *encoding*	1.7
javadoc	String	Indicates whether Javadoc related messages should be printed. Defaults to "true"	1.7
logLevel	String	Specifies the logging level for message output. Either one of "ERROR", "WARN", "INFO" or "DEBUG" can be used (case insensitive). When omitted, the current code convention settings will be used (see Section 2.6.1, "Categories")	1.7
logFile	String	Specifies the log file to use for logging output. The format of the logging output is determined by the extension of the given file. Valid extensions are `".log"` for a custom plain text format, `".xml"` for a plain XML format and `".html"` for a hierarchical	1.7

Property	Type	Description	Since
		HTML report. When omitted, the current code convention setting will be used (see Section 2.6.2, "Logging")	
outputEncoding	String	Sets the character encoding Jalopy uses to write files. Defaults to the platform default encoding. Please note that this setting always overrides *encoding*	1.7
profile	String	Sets the Jalopy profile that should be used during the formatting run (refer to Section 2.1.1.1, "Main window" for more information about profiles). The currently active profile will be restored after formatting. Please note that if no convention is specified, the profile must exist!	1.7
repository	Boolean	Indicates whether the disk-based type repository should be used for type lookup. You may want to disable the disk-based type repository if you commonly format a single file or only a small set of files in order to avoid the maintenance overhead of the type repository. Defaults to "true"	1.7
sources	List	The source directories containing the sources to be formatted. When omitted, uses the directories defined for the compiler (${project.compileSourceRoots}) instead	1.7
test	Boolean	Sets whether formatting output should actually be written to disk. If set to "true" no output will be written to disk. The default is "false"	1.7
threads	Integer	Specifies the number of processing threads to use. Integer between `1 - 8`. Defaults to '1';	1.7

To configure the Plug-in, you specify elements named after the available parameters where the contents of an element is the value to be assigned to the parameter.

```
<plugin>
  <groupId>triemax</groupId>
  <artifactId>jalopy-maven</artifactId>
  <configuration>
    <threads>4</threads>
    <profile>test</profile>
  </configuration>
  [...]
</plugin>
```

For parameters of type *List* you would use multiple element tags to add the different values to the list.

```
<plugin>
  <groupId>triemax</groupId>
  <artifactId>jalopy-maven</artifactId>
  <configuration>
    <includes>
      <include>com/foo/siri/**</include>
      <include>com/foo/lana/**</include>
    </includes>
    <excludes>
      <exclude>**/*.sqlj</exclude>
      <exclude>**/*Test/**</exclude>
    </excludes>
    [...]
  </configuration>
</plugin>
```

Configuring *Maps* works similar: elements are named after the keys and the element contents is the value to be assigned to the key.

```
<plugin>
  <groupId>triemax</groupId>
  <artifactId>jalopy-maven</artifactId>
  <configuration>
    <environment>
      <lead>John Doo</lead>
      <office>Alta Nova</office>
    </environment>
    [...]
  </configuration>
</plugin>
```

For a complete example, please refer to Section 11.4, "Example" below.

11.4 Example

Below you find a complete POM that performs formatting after each compile run finished successfully, disables logging of Javadoc related messages, only displays messages with warning severity or higher, activates the profile "test" during formatting and imports the code convention jalopy.xml from the build-config artifact. Formatting is applied to all Java source files of the project that are not located below the "testdata" folder.

Example 11.1. Maven POM

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>testing</groupId>
  <artifactId>test</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>triemax</groupId>
        <artifactId>jalopy-maven</artifactId>
        <configuration>
          <javadoc>false</javadoc>
          <logLevel>warn</logLevel>
          <convention>classpath:jalopy.xml</convention>
          <sources>
            <source>/work/foo/main/src/java</source>
            <source>/work/foo/test/src/java</source>
          </sources>
          <includes>
            <include>**/*.java</include>
          </includes>
          <excludes>
            <exclude>**/testdata/**</exclude>
          </excludes>
          <environment>
            <lead>John Doo</lead>
            <office>Alta Nova</office>
          </environment>
```

```
        </configuration>
        <dependencies>
          <!-- Import the artifact that provides the code convention -->
          <dependency>
            <groupId>com.mycompany</groupId>
            <artifactId>build-config</artifactId>
            <version>1.0.2</version>
          </dependency>
        </dependencies>
        <executions>
          <execution>
            <phase>process-classes</phase>
            <goals>
              <goal>format</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Chapter 12. NetBeans Module

Describes the installation and usage of the Jalopy NetBeans Plug-in module. NetBeans [Link] is the original full-featured, free and open source IDE for Java software developers to create cross-platform desktop, mobile and web applications based on industry standards utilizing the latest technologies.

12.1 Installation

Explains the steps involved to install the NetBeans Plug-in module.

12.1.1 System requirements

The Plug-in works with NetBeans releases 4.0 - 6.5 or the corresponding Sun ONE Studio or Java Studio Creator versions. See Section 1.1, “System requirements” for the basic requirements to run Jalopy.

12.1.2 Setup

The Plug-in comes as an executable Jar Archive (JAR) that contains a graphical setup wizard to let you easily install the software. Wizard installation is mandatory and explained in detail in Section 1.3, “Wizard Installation”.

12.2 Integration

Describes how the Plug-in integrates into NetBeans.

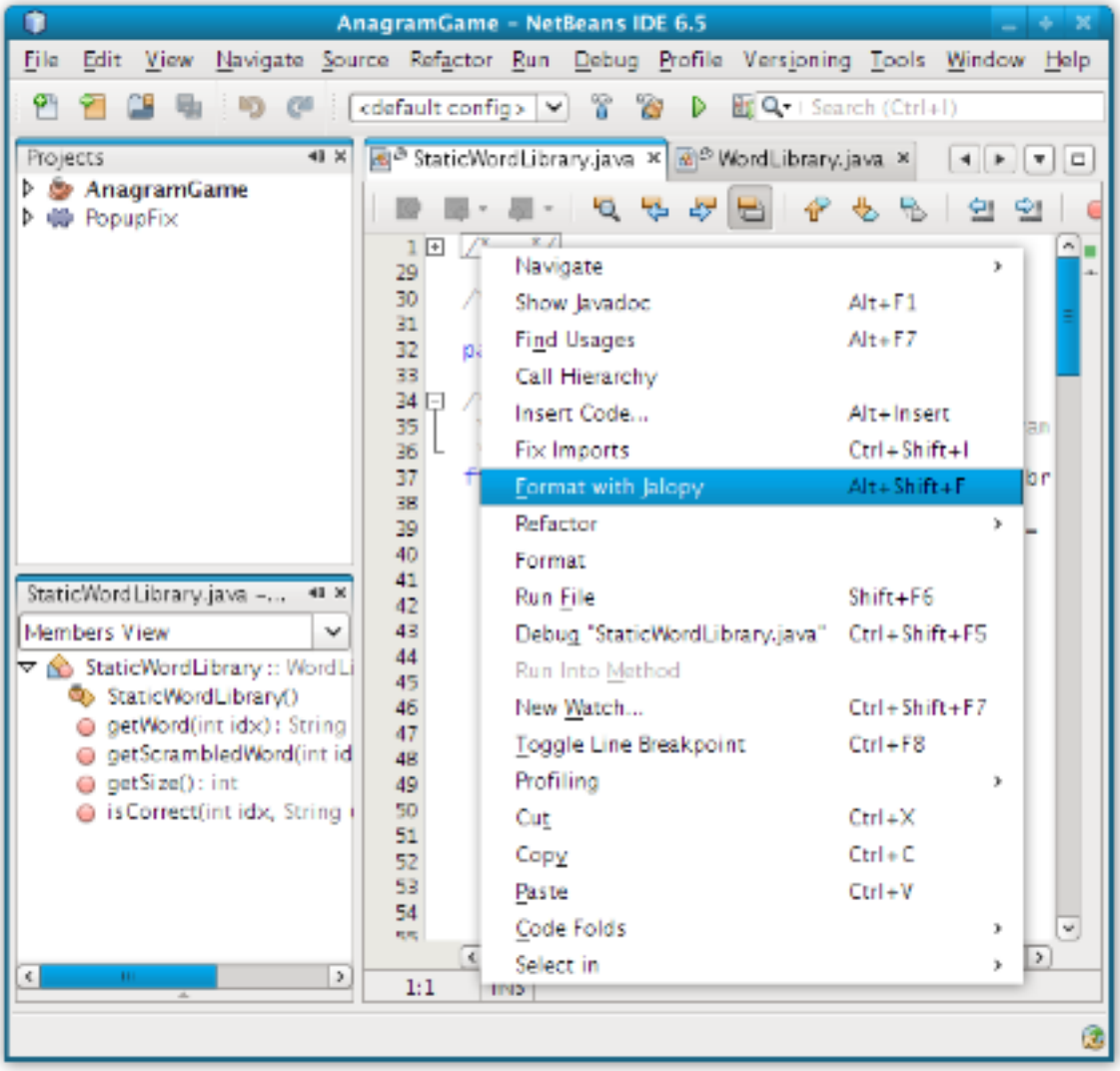
12.2.1 Editor pop-up menu

The Plug-in adds a new menu item into the context pop-up menu of Java code editors:

Format with Jalopy

Formats the contents of the editor. When currently some text is selected in the editor, only the selected text will be formatted (*selective formatting*). This can be especially helpful when editing portions of very large files, as selective formatting can speed up processing considerably. But comes especially when you want to limit formatting to a specific file portion in order to avoid unnecessary differences when editing a file that has not (yet) been formatted according to the active code convention.

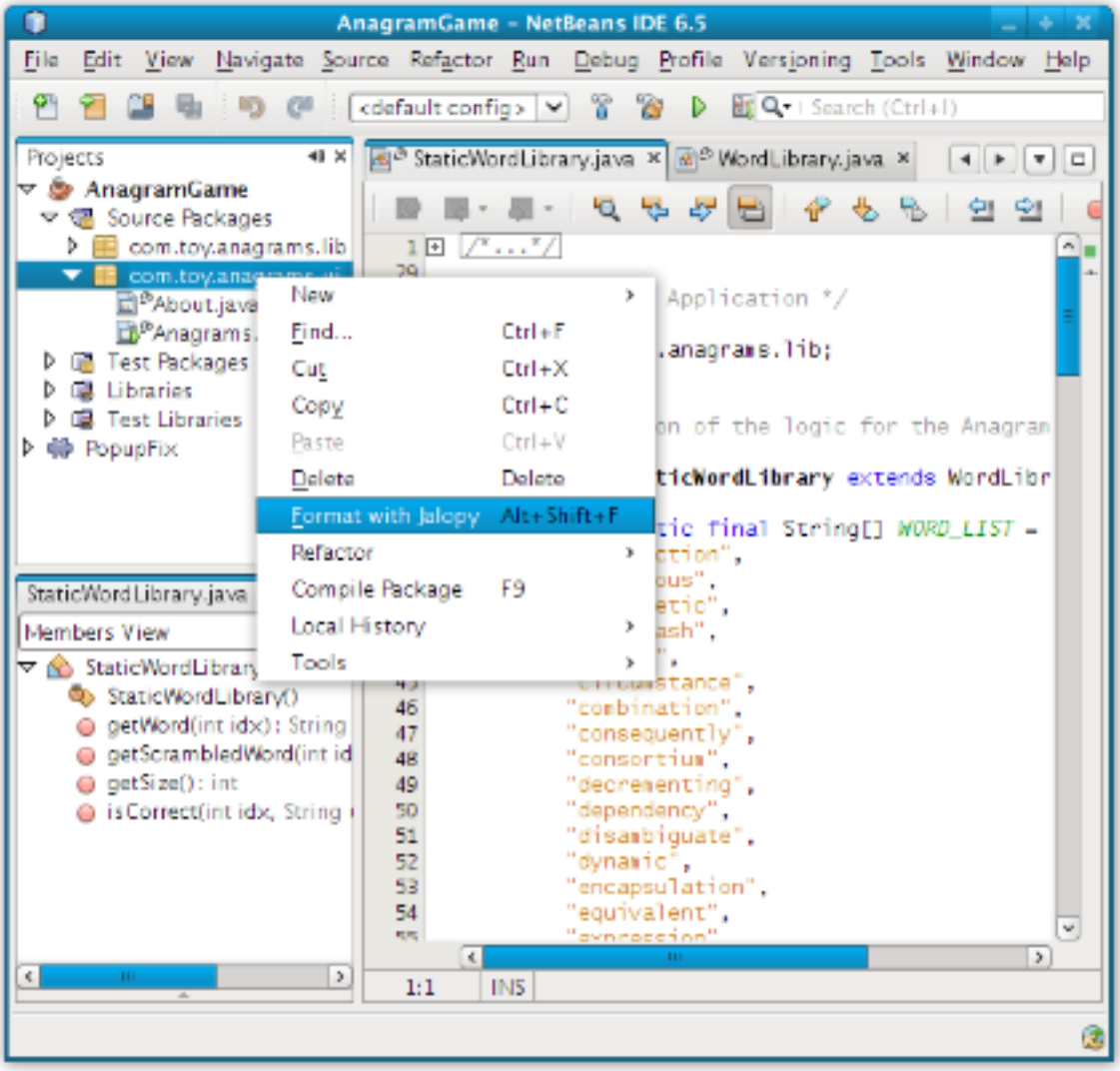
Figure 12.1. Jalopy editor pop-up menu item



12.2.2 Explorer pop-up menu

Formatting can also be triggered via the pop-up menu of several views, like the Projects or Files window. Note that the item only appears for folder nodes or Java source files.

Figure 12.2. Jalopy Format pop-up menu item



If it happens that a file has an open editor, this editor will be updated, not the actual file. You have to save the editor first, to see the physical file updated.

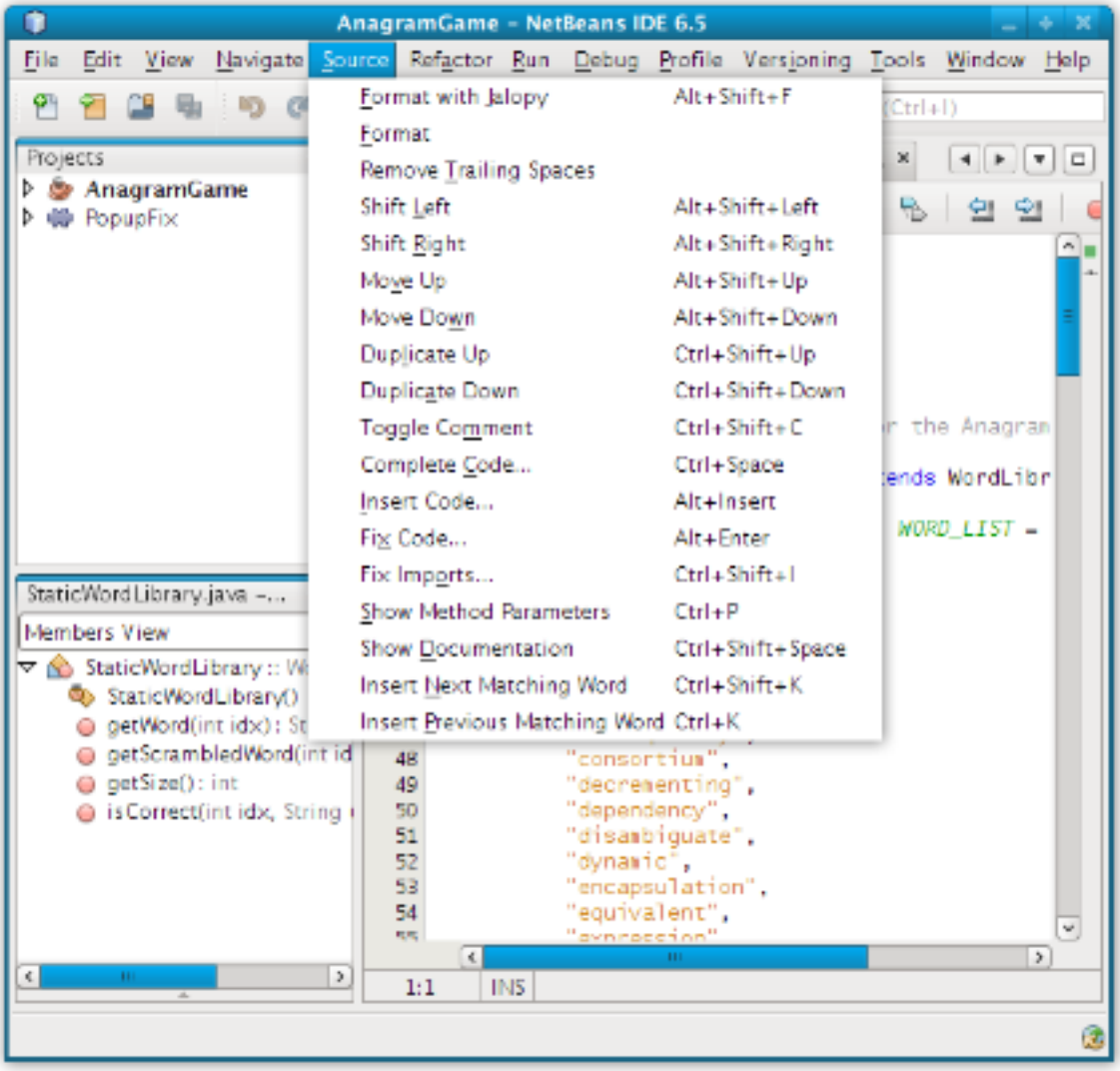
12.2.3 Workspace main menu

The module adds a new menu item into the main menu of the current workspace to seamlessly integrate with NetBeans:

Source > Format with Jalopy

Formats the currently selected node(s) or the currently active editor. Only available if there are indeed nodes selected which represents or contains Java source files or an editor is focused.

Figure 12.3. Jalopy Format menu bar item

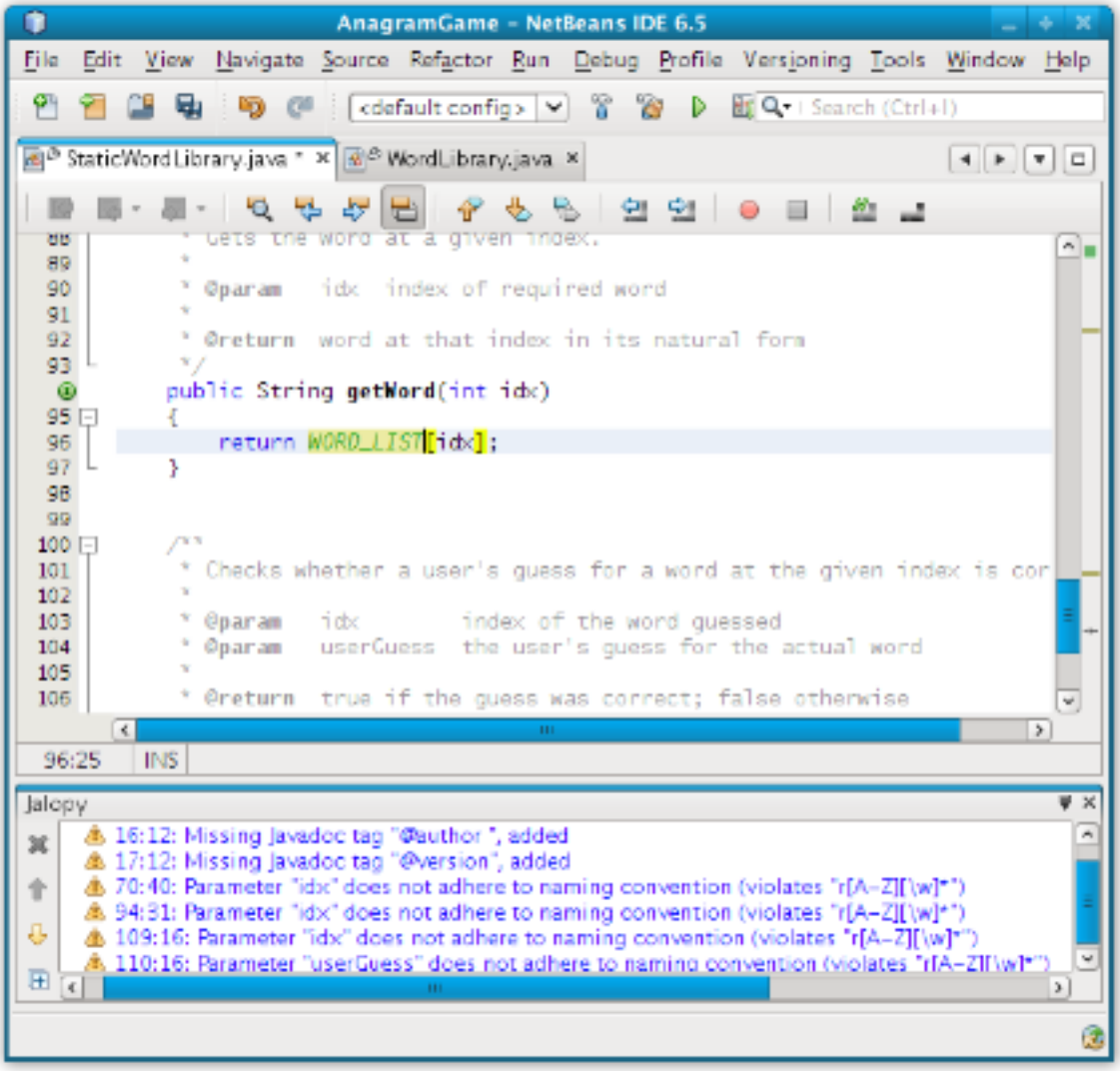


12.2.4 Message window

Jalopy displays all runtime messages in its own dockable window. Messages are shown in a tree control, with each branch containing the messages for a specific file, and individual messages displayed as leafs. File messages display the number of leaves and the warning and error count.

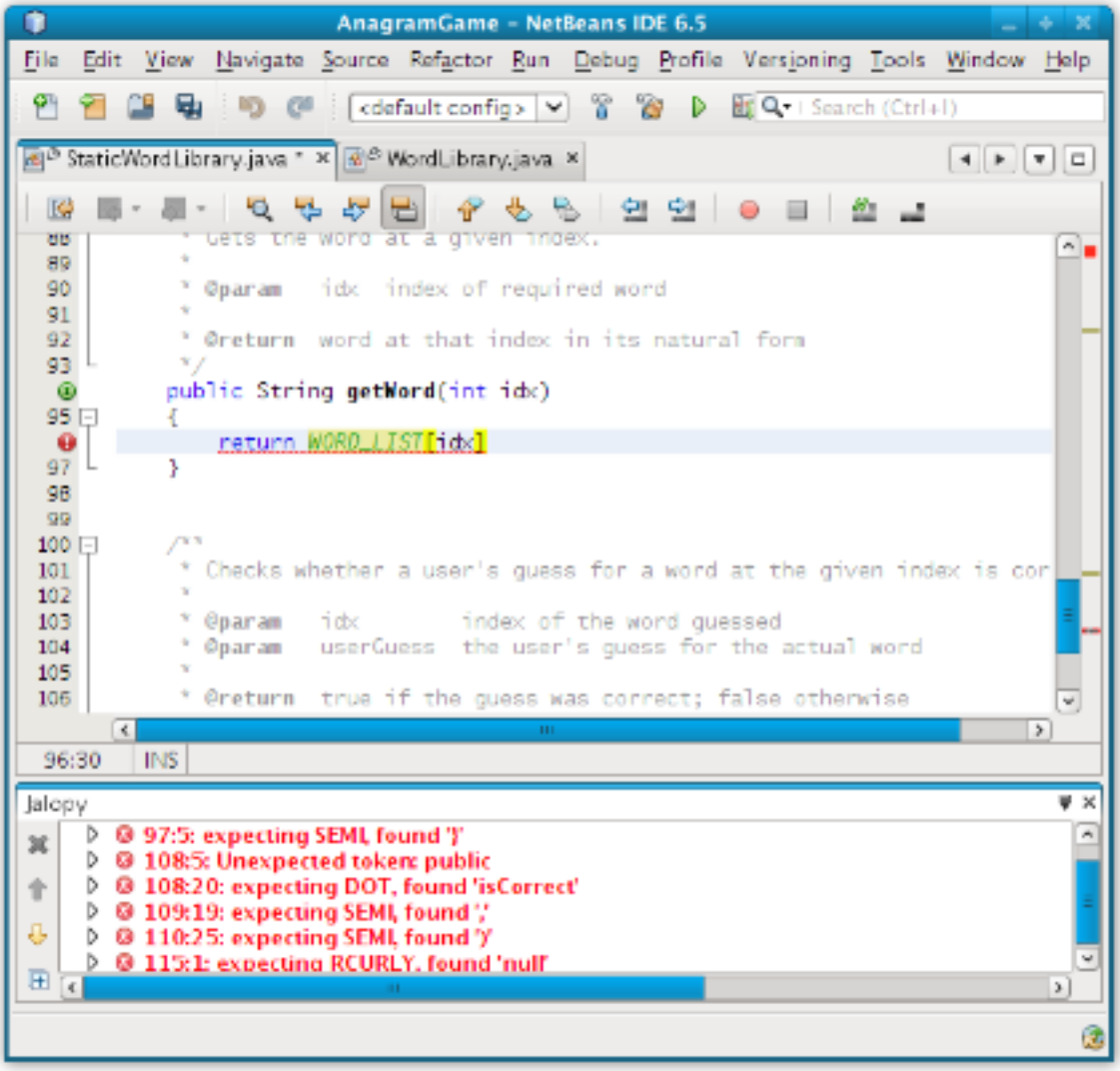
The message types are differentiated with icons and by color: Errors are red with an error icon, warnings are shown in blue and display a warning sign, informational messages are black and carry a file icon and debugging messages are black and prepended by a bug icon.

Figure 12.4. Jalopy dockable window



Clicking on a file name will open that file, clicking on a message that contains location information will open the file containing the message and move the caret to the nominated location.

Figure 12.5. Jalopy dockable window



The window provides a context menu with some useful actions.

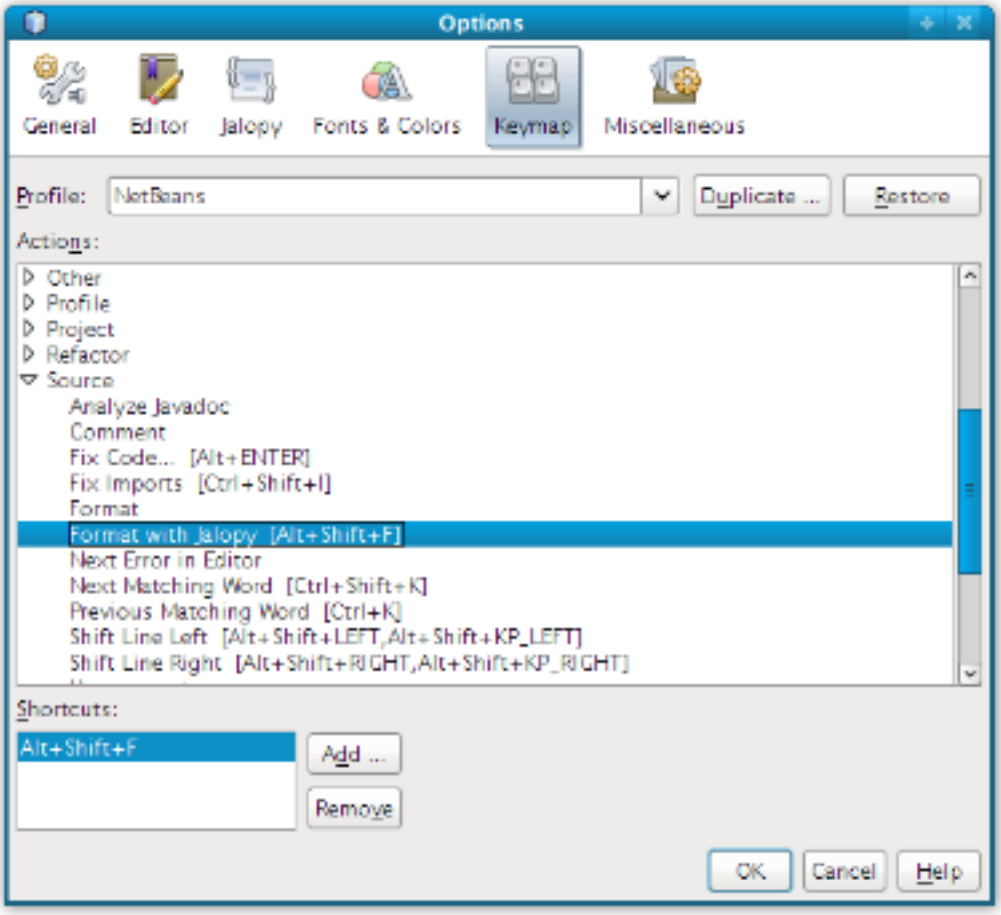
Copy	Copies the textual contents of the selected messages into the System clipboard. If a message contains children, the contents of all children are copied as well
Clear	Removes all selected messages
Clear All	Removes all messages currently being displayed in the window
Select All	Selects all messages currently being displayed in the window

12.2.5 Keyboard shortcuts

You can define keyboard shortcuts for the different Jalopy actions via the NetBeans Keymap dialog. Open the options dialog via Tools > Options (NetBeans > Preferences... on Mac OS X) and select the Keymap item.

Jalopy provides two actions. The “Format with Jalopy” action in the “Source” section and the “Jalopy Options” action to invoke the configuration dialog in the “Tools” section.

Figure 12.6. Keyboard shortcuts

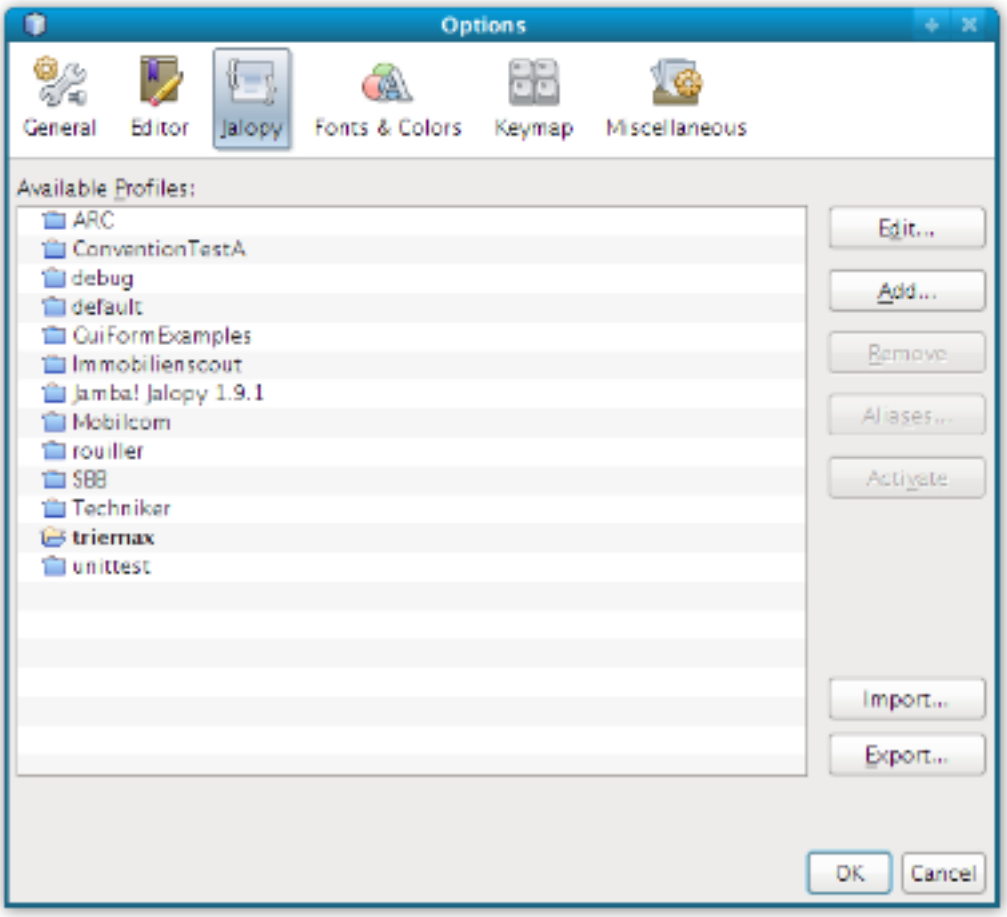


To configure a keyboard shortcut, select an action and either press the *Add...* button to add a keyboard shortcut. Or use the *Remove* button to remove an existing shortcut. For compatibility reasons, the default keyboard shortcut for the "Format with Jalopy" action is Strg+Shift+F10, but it is recommended to adjust the shortcut to something more accessible, like Alt+Shift+F.

12.2.6 Options dialog

The Jalopy options are available through the NetBeans options dialog. In order to display the options dialog, on Mac OS X you use NetBeans > Preferences... and select the "Jalopy" item in the top pane. On other platforms the dialog is available through Tools > Options. Please note that the options dialog is only available since NetBeans 5.0. With earlier versions, Jalopy adds a new menu item to the menu bar to display the Jalopy options.

Figure 12.7. Jalopy options dialog



The main options page lets you manage your Jalopy profiles. A profile stores the actual code convention to define formatting output, as well as user-specific data like file and dialog histories. You can add, remove, activate, map and configure any number of profiles. For a detailed explanation of the available options, please refer to Section 2.1.1.1, “Main window”.

12.3 Configuration

Although Jalopy ships with sensible default settings (mimicking the Sun Java coding convention), you most likely want to configure the formatter to match your needs (adding copyright headers, tune Javadoc handling and the like). For such, Jalopy comes with a graphical configuration tool that lets you interactively customize the settings. See Chapter 2, *Configuration* for an in-depth discussion of the available options. Please refer to Section 12.2, “Integration” for information on how to display the configuration dialog from within NetBeans.

Part III. Appendices

Appendix A. Library Dependencies

Jalopy depends on the following freely available libraries:

Table A.1. Library dependencies

Name:	ANTLR Parser Generator 2.7.2
Author:	jGuru.com (MageLang Institute), project lead by Dr. Terence Parr
License:	Custom (Public Domain)
Info:	Contains some minor patches to make it work with Jalopy
URL:	http://www.antlr.org/
Name:	Apache Harmony
Author:	Apache Software Foundation
License:	Apache License 2.0
Info:	Jalopy incorporates their java.util.Formatter implementation backported to Java 1.4
URL:	http://harmony.apache.org/
Name:	ASM 1.4.1
Author:	Object Web Consortium, project lead by Eric Bruneton
License:	BSD
Info:	Jalopy incorporates only the minimal required functionality to read and perform basic analysis of Java class files
URL:	http://asm.objectweb.org/
Name:	Jakarta Commons CLI 1.0
Author:	Apache Software Foundation
License:	Apache License 1.1
URL:	http://jakarta.apache.org/commons/
Name:	JGoodies Forms Framework 1.0.3
Author:	JGoodies Karsten Lentzsch
License:	BSD
Info:	Contains a small patch to allow customization of paragraph separator
URL:	http://www.jgoodies.com/
Name:	JDBM 0.20
Author:	Cees de Groot, Alex Boisvert
License:	BSD
Info:	Contains a small patch to not include the serializers in the database files upon serialization
URL:	http://jdbm.sourceforge.net/
Name:	JDOM XML API 1.0
Author:	JDOM Group, lead by Jason Hunter and Brett McLaughlin
License:	BSD/Apache style
URL:	http://www.jdom.org/
Name:	JSyntaxPane

Author: Ayman Al-Sairafi
License: Apache License 2.0
Info: Jalopy incorporates a patched version that uses ANTLR based lexers
URL: http://code.google.com/p/jsyntaxpane/

Name: log4j logging toolkit 1.2.8
Author: Apache Software Foundation
License: Apache License 1.1
URL: http://logging.apache.org/log4j/

Name: Maven Jalopy Plugin 1.3.1
Author: Apache Software Foundation
License: Apache License 2.0
Info: Jalopy incorporates the version of the Jalopy Plug-in that ships with Maven updated to use the commercial Jalopy formatting engine
URL: http://maven.apache.org/

Name: One-JAR 0.95
Author: Simon Tuffs
License: BSD
URL: http://one-jar.sourceforge.net/

Name: Progress 1.2
Author: Bernhard Picher
License: Creative Commons Attribution
Info: Backported to Java 1.4
URL: http://www.repher.at/

Name: SoftHashMap
Author: Dr. Heinz M. Kabutz
License: Unknown—used with express permission of the author
URL: http://www.javaspecialists.co.za/archive/Issue098.html

Name: TreeTable
Author: Sun Microsystems, Inc.
License: BSD
URL: http://java.sun.com/products/jfc/tsc/articles/bookmarks/

All libraries have been repackaged to avoid class path issues.

Appendix B. Build-in XDoclet tags

The Javadoc formatter recognizes the following tags as XDoclet-tags.

Table B.1. Build-in XDoclet tags

@actionscript.class	@actionscript.property
@ant.attribute	@ant.element
@ant.ignore	@ant.not-required
@ant.required	@axis.method
@axis.service	@bes.bean
@bes.cross-table	@bes.datasource
@bes.ejb-local-ref	@bes.ejb-ref
@bes.property	@bes.relation
@bes.resource-env-ref	@bes.resource-ref
@castor.class	@castor.field
@castor.field-sql	@castor.field-xml
@contrib.checkbox-group	@contrib.choose
@contrib.control-checkbox	@contrib.controlled-checkbox
@contrib.date-field	@contrib.dump-object
@contrib.form-conditional	@contrib.form-table
@contrib.inspector-button	@contrib.mask-edit
@contrib.multiple-property-selection	@contrib.numeric-field
@contrib.otherwise	@contrib.palette
@contrib.selector	@contrib.show-description
@contrib.show-engine	@contrib.show-properties
@contrib.show-specification	@contrib.show-template
@contrib.simple-table-column-component	@contrib.simple-table-column-form-component
@contrib.table	@contrib.table-columns
@contrib.table-form-pages	@contrib.table-form-rows
@contrib.table-pages	@contrib.table-rows
@contrib.table-values	@contrib.table-view
@contrib.timeout	@contrib.tree
@contrib.tree-data-view	@contrib.tree-node-view
@contrib.tree-table	@contrib.tree-table-data-view
@contrib.tree-table-node-view-delegator	@contrib.tree-view
@contrib.validating-text-field	@contrib.view-tabs
@contrib.when	@contrib.x-tile
@dao.call	@doc.param
@doc.tag	@easerver.ejb-ref
@easerver.resource-ref	@ejb.activation-config-property
@ejb.aggregate	@ejb.bean
@ejb.create-method	@ejb.dao
@ejb.data-object	@ejb.destination-ref
@ejb.ejb-external-ref	@ejb.ejb-ref
@ejb.ejb-service-ref	@ejb.env-entry
@ejb.facade	@ejb.facade-method
@ejb.finder	@ejb.home

@ejb.home-method
@ejb.interface-method
@ejb.permission
@ejb.persistence-field
@ejb.pk
@ejb.relation
@ejb.resource-env-ref
@ejb.security-identity
@ejb.security-roles
@ejb.transaction
@ejb.util
@ejb.value-object-field
@generama.property
@hibernate.any-column
@hibernate.bag
@hibernate.class
@hibernate.collection-composite-element
@hibernate.collection-id
@hibernate.collection-jcs-cache
@hibernate.collection-key-column
@hibernate.collection-one-to-many
@hibernate.comment
@hibernate.composite-element
@hibernate.composite-index
@hibernate.composite-map-key
@hibernate.discriminator-column
@hibernate.filter
@hibernate.filter-param
@hibernate.generator-param
@hibernate.idbag
@hibernate.index
@hibernate.index-many-to-any
@hibernate.jcs-cache
@hibernate.join-key
@hibernate.joined-subclass-key
@hibernate.key-column
@hibernate.key-property
@hibernate.list-index
@hibernate.many-to-any
@hibernate.many-to-many
@hibernate.map
@hibernate.map-key-many-to-many
@hibernate.meta
@hibernate.natural-id
@hibernate.one-to-one
@hibernate.primitive-array
@hibernate.property
@hibernate.query-list
@ejb.interface
@ejb.message-destination
@ejb.persistence
@ejb.persistent-field
@ejb.pk-field
@ejb.remote-facade
@ejb.resource-ref
@ejb.security-role-ref
@ejb.select
@ejb.transaction-method
@ejb.value-object
@foo:bar
@hibernate.any
@hibernate.array
@hibernate.cache
@hibernate.collection-cache
@hibernate.collection-element
@hibernate.collection-index
@hibernate.collection-key
@hibernate.collection-many-to-many
@hibernate.column
@hibernate.component
@hibernate.composite-id
@hibernate.composite-key
@hibernate.discriminator
@hibernate.element
@hibernate.filter-def
@hibernate.formula
@hibernate.id
@hibernate.import
@hibernate.index-column
@hibernate.index-many-to-many
@hibernate.join
@hibernate.joined-subclass
@hibernate.key
@hibernate.key-many-to-one
@hibernate.list
@hibernate.loader
@hibernate.many-to-any-column
@hibernate.many-to-one
@hibernate.map-key
@hibernate.mapping
@hibernate.meta-value
@hibernate.one-to-many
@hibernate.parent
@hibernate.properties
@hibernate.query
@hibernate.set

@hibernate.sql-delete
@hibernate.sql-delete-all
@hibernate.sql-insert
@hibernate.sql-query
@hibernate.sql-update
@hibernate.subclass
@hibernate.subselect
@hibernate.synchronize
@hibernate.timestamp
@hibernate.tuplizer
@hibernate.type
@hibernate.type-param
@hibernate.typedef
@hibernate.typedef-param
@hibernate.union-subclass
@hibernate.version
@hpas.bean
@hpas.ejb-ref
@hpas.pool
@javabean.attribute
@javabean.class
@javabean.icons
@javabean.method
@javabean.param
@javabean.parameter
@javabean.property
@jboss.audit
@jboss.audit-created-by
@jboss.audit-created-time
@jboss.audit-updated-by
@jboss.audit-updated-time
@jboss.cache-invalidation
@jboss.cache-invalidation-config
@jboss.cluster-config
@jboss.clustered
@jboss.cmp-field
@jboss.column-name
@jboss.container-configuration
@jboss.create-table
@jboss.declared-sql
@jboss.depends
@jboss.destination-jndi-name
@jboss.dvc-property
@jboss.ejb-local-ref
@jboss.ejb-ref-jndi
@jboss.entity-command
@jboss.entity-command-attribute
@jboss.finder-query
@jboss.jdbc-type
@jboss.method-attributes
@jboss.not-persisted-field
@jboss.persistence
@jboss.port-component
@jboss.query
@jboss.read-ahead
@jboss.read-only
@jboss.relation
@jboss.relation-mapping
@jboss.relation-read-ahead
@jboss.relation-table
@jboss.remove-table
@jboss.resource-adapter
@jboss.resource-env-ref
@jboss.resource-manager
@jboss.resource-ref
@jboss.security-proxy
@jboss.service
@jboss.sql-type
@jboss.subscriber
@jboss.target-relation
@jboss.unknown-pk
@jdo.class
@jdo.fetchgroup
@jdo.field
@jdo.persistence-capable
@jdo.query
@jmx.managed-attribute
@jmx.managed-constructor
@jmx.managed-operation
@jmx.managed-parameter
@jmx.mbean
@jmx.mlet-entry
@jmx.notification
@jonas.bean
@jonas.cmp-field-jdbc-mapping
@jonas.ejb-ref
@jonas.finder-method-jdbc-mapping
@jonas.is-modified-method-name
@jonas.jdbc-mapping
@jonas.max-cache-size
@jonas.message-driven-destination
@jonas.min-pool-size
@jonas.passivation-timeout
@jonas.resource
@jonas.resource-env
@jonas.session-timeout

@jonas.shared
@jrun.cluster-home
@jrun.commit-option
@jrun.ejb-ref
@jrun.jdbc-mappings
@jrun.message-driven-destination
@jrun.resource-env-ref
@jrun.timeout
@jsf.bean
@jsf.managed-property
@jsf.render-kit
@jsf.validator-attribute
@jsp.tag
@jsp.variable
@lido.future
@msg.bundle
@mvcsoft.col-name
@mvcsoft.exclude-from-optimistic-lock
@mvcsoft.high-low-key
@mvcsoft.lightweight
@mvcsoft.relation
@mvcsoft.unknown-key
@mvcsoft.wrap
@oc4j.field-persistence-manager
@oc4j.persistence
@orion.field-persistence-manager-property
@portlet.portlet
@portlet.portlet-init-param
@portlet.preferences-validator
@portlet.supports
@pramati.destination-mapping
@pramati.ejb-ref
@pramati.resource-env-ref
@pramati.server-session
@qtags.alias
@qtags.default
@qtags.ignore
@qtags.location
@qtags.required
@resin-ejb.cmp-field
@resin-ejb.entity-method
@resin-ejb.relation
@soap.service
@spring.constructor-arg
@spring.validator
@spring.validator-var
@struts.action
@struts.action-forward
@jrun.always-dirty
@jrun.cluster-object
@jrun.ejb-local-ref
@jrun.instance-pool
@jrun.jndi-name
@jrun.message-driven-subscription
@jrun.resource-ref
@jrun.tx-domain
@jsf.converter
@jsf.navigation
@jsf.validator
@jsp.attribute
@jsp.validator-init-param
@kodo.table
@mock.generate
@msg.message
@mvcsoft.entity
@mvcsoft.fault-group
@mvcsoft.jdbc-type
@mvcsoft.query
@mvcsoft.sql-type
@mvcsoft.uuid-key
@oc4j.bean
@oc4j.field-persistence-manager-property
@orion.bean
@orion.persistence
@portlet.portlet-info
@portlet.preference
@portlet.security-role-ref
@pramati.bean
@pramati.ejb-local-ref
@pramati.persistence
@pramati.resource-mapping
@pramati.thread-pool
@qtags.allowed-value
@qtags.deprecated
@qtags.list-token
@qtags.once
@qtags.verbatim
@resin-ejb.entity-bean
@resin-ejb.message-bean
@soap.method
@spring.bean
@spring.property
@spring.validator-args
@sql.table
@struts.action-exception
@struts.action-set-property

@struts.dynaform
@struts.dynaform-field
@struts.form
@struts.form-field
@struts.tiles
@struts.tiles-put
@struts.validator
@struts.validator-args
@struts.validator-var
@sunone.bean
@sunone.bean-cache
@sunone.bean-pool
@sunone.consistency
@sunone.fetched-with
@sunone.finder
@sunone.persistence-manager
@sunone.pool-manager
@sunone.relation
@tacos.ajax-direct-link
@tacos.ajax-field-observer
@tacos.ajax-form
@tacos.ajax-link-submit
@tacos.ajax-submit
@tacos.autocompleter
@tacos.date-picker
@tacos.dialog
@tacos.dirty-form-warning
@tacos.drop-target
@tacos.editor
@tacos.fisheye-list
@tacos.floating-pane
@tacos.inline-edit-box
@tacos.palette
@tacos.partial-for
@tacos.progress-bar
@tacos.refresh
@tacos.site-map
@tacos.tree
@tapestry.action-link
@tapestry.any
@tapestry.asset
@tapestry.bean
@tapestry.binding
@tapestry.block
@tapestry.body
@tapestry.button
@tapestry.card
@tapestry.checkbox
@tapestry.component
@tapestry.component-specification
@tapestry.conditional
@tapestry.context-asset
@tapestry.date-picker
@tapestry.delegator
@tapestry.describe
@tapestry.direct-link
@tapestry.do
@tapestry.else
@tapestry.exception-display
@tapestry.external-asset
@tapestry.external-link
@tapestry.field-label
@tapestry.for
@tapestry.foreach
@tapestry.form
@tapestry.frame
@tapestry.generic-link
@tapestry.go
@tapestry.hidden
@tapestry.if
@tapestry.image
@tapestry.image-submit
@tapestry.inherited-binding
@tapestry.inject
@tapestry.inject-meta
@tapestry.inject-object
@tapestry.inject-page
@tapestry.inject-script
@tapestry.inject-state
@tapestry.input
@tapestry.insert
@tapestry.insert-text
@tapestry.invoke-listener
@tapestry.link-submit
@tapestry.list-edit
@tapestry.listener-binding
@tapestry.message-binding
@tapestry.meta
@tapestry.on-event
@tapestry.option
@tapestry.page-link
@tapestry.page-specification
@tapestry.parameter
@tapestry.postfield
@tapestry.private-asset
@tapestry.property

@tapestry.property-selection
@tapestry.property-specification
@tapestry.radio
@tapestry.radio-group
@tapestry.render-block
@tapestry.render-body
@tapestry.request-display
@tapestry.reserved-parameter
@tapestry.rollover
@tapestry.script
@tapestry.select
@tapestry.selection-field
@tapestry.service-link
@tapestry.set
@tapestry.set-message-property
@tapestry.set-property
@tapestry.setvar
@tapestry.shell
@tapestry.static-binding
@tapestry.submit
@tapestry.text-area
@tapestry.text-field
@tapestry.timer
@tapestry.upload
@tapestry.valid-field
@web.ejb-local-ref
@web.ejb-ref
@web.env-entry
@web.filter
@web.filter-init-param
@web.filter-mapping
@web.interface-method
@web.listener
@web.resource-env-ref
@web.resource-ref
@web.security-role
@web.security-role-ref
@web.servlet
@web.servlet-init-param
@web.servlet-mapping
@weblogic.allow-concurrent-calls
@weblogic.allow-remove-during-transaction
@weblogic.automatic-key-generation
@weblogic.cache
@weblogic.cache-ref
@weblogic.clients-on-same-server
@weblogic.clustering
@weblogic.column-map
@weblogic.create-as-principal-name
@weblogic.data-source-name
@weblogic.dbms-column-type
@weblogic.delay-database-insert-until
@weblogic.dispatch-policy
@weblogic.ejb-local-reference-description
@weblogic.ejb-reference-description
@weblogic.enable-batch-operations
@weblogic.enable-bean-class-redeploy
@weblogic.enable-call-by-reference
@weblogic.enable-dynamic-queries
@weblogic.enable-tuned-updates
@weblogic.field-group
@weblogic.finder
@weblogic.idempotent-methods
@weblogic.iiop-security-descriptor
@weblogic.instance-lock-order
@weblogic.invalidation-target
@weblogic.lifecycle
@weblogic.lock-order
@weblogic.message-driven
@weblogic.order-database-operations
@weblogic.passivate-as-principal-name
@weblogic.persistence
@weblogic.pool
@weblogic.pool-name
@weblogic.relation
@weblogic.remove-as-principal-name
@weblogic.resource-description
@weblogic.resource-env-description
@weblogic.run-as-identity-principal
@weblogic.run-as-principal-name
@weblogic.select
@weblogic.target-column-map
@weblogic.transaction-descriptor
@weblogic.transaction-isolation
@weblogic.use-select-for-update
@weblogic.verify-rows
@websphere.bean
@websphere.bean-cache
@websphere.cmp
@websphere.finder-query
@websphere.local-transaction
@websphere.mapping
@websphere.mapping-constraint
@websphere.mdb
@websphere.resource-ref
@webwork.action

@webwork.command
@wsee.jaxrpc-mapping
@wsee.variable-mapping
@xdoclet.taghandler
@xwork.exception-mapping
@xwork.param
@wsee.handler
@wsee.port-component
@xdoclet.merge-file
@xwork.action
@xwork.interceptor-ref
@xwork.result

Appendix C. ANTLR Software License

ANTLR 1989-2003 Developed by jGuru.com (MageLang Institute), http://www.ANTLR.org and http://www.jGuru.com

We reserve no legal rights to the ANTLR—it is fully in the public domain. An individual or company may do whatever they wish with source code distributed with ANTLR or the code generated by ANTLR, including the incorporation of ANTLR, or its output, into commercial software.

We encourage users to develop software with ANTLR. However, we do ask that credit is given to us for developing ANTLR. By "credit", we mean that if you use ANTLR or incorporate any source code into one of your programs (commercial product, research project, or otherwise) that you acknowledge this fact somewhere in the documentation, research report, etc... If you like ANTLR and have developed a nice tool with the output, please mention that you developed it using ANTLR. In addition, we ask that the headers remain intact in our source code. As long as these guidelines are kept, we expect to continue enhancing this system and expect to make other tools available as they are completed.

Appendix D. Apache Software License 1.1

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)". Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear.
4. The names "Ant" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org.
5. Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software consists of voluntary contributions made by many individuals on behalf of the Apache Software Foundation. For more information on the Apache Software Foundation, please see http://www.apache.org.

Appendix E. Apache Software License 2.0

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License.

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare

Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License.

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution.

You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

a. You must give any other recipients of the Work or Derivative Works a copy of this License; and

b. You must cause any modified files to carry prominent notices stating that You changed the files; and

c. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

d. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions.

Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks.

This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty.

Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability.

In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability.

While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

Appendix F. ASM Software License

ASM: a very small and fast Java bytecode manipulation framework

Appendix G. Common Public License 1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS COMMON PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

"Contribution" means:

a. in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and

b. in the case of each subsequent Contributor:

   i. changes to the Program, and

   ii. additions to the Program;

where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents" mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

"Program" means the Contributions distributed in accordance with this Agreement.

"Recipient" means anyone who receives the Program under this Agreement, including all Contributors.

2. GRANT OF RIGHTS

a. Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form.

b. Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.

c. Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.

d. Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

a. it complies with the terms and conditions of this Agreement; and

b. its license agreement:

   i. effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;

   ii. effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;

   iii. states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and

   iv. states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

a. it must be made available under this Agreement; and

b. a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor")

hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

6. DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. GENERAL

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against a Contributor with respect to a patent applicable to software (including a cross-claim or counterclaim in a lawsuit), then any patent licenses granted by that Contributor to such Recipient under this Agreement shall terminate as of the date such litigation is filed. In addition, if Recipient institutes patent litigation against any entity (including a cross-claim or

counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable. However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. IBM is the initial Agreement Steward. IBM may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.

Appendix H. Creative Commons Attribution License

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS LIABILITY FOR DAMAGES RESULTING FROM ITS USE.

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS

1. Definitions

a. "**Collective Work**" means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with a number of other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.

b. "**Derivative Work**" means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License.

c. "**Licensor**" means the individual or entity that offers the Work under the terms of this License.

d. "**Original Author**" means the individual or entity who created the Work.

e. "**Work**" means the copyrightable work of authorship offered under the terms of this License.

f. "**You**" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;

b. to create and reproduce Derivative Works;

c. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;

d. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.

e. For the avoidance of doubt, where the work is a musical composition:

   i. **Performance Royalties Under Blanket Licenses.** Licensor waives the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work.

   ii. **Mechanical Rights and Statutory Royalties.** Licensor waives the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).

f. **Webcasting Rights and Statutory Royalties.** For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any credit as required by clause 4(b), as requested. If You create a Derivative Work, upon notice from

any Licensor You must, to the extent practicable, remove from the Derivative Work any credit as required by clause 4(b), as requested.

b. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work or any Derivative Works or Collective Works, You must keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or (ii) if the Original Author and/or Licensor designate another party or parties (e.g. a sponsor institute, publishing entity, journal) for attribution in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work (e.g. "French translation of the Work by Original Author" or "Screenplay based on original Work by Original Author"). Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit.

5. Representations, Warranties and Disclaimer UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Derivative Works or Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

b. Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, neither party will use the trademark “Creative Commons” or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time.

Creative Commons may be contacted at http://creativecommons.org/.

Appendix I. JDBM Software License

JDBM LICENSE v1.0

Redistribution and use of this software and associated documentation ("Software"), with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain copyright statements and notices. Redistributions must also contain a copy of this document.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The name "JDBM" must not be used to endorse or promote products derived from this Software without prior written permission of Cees de Groot.
4. Products derived from this Software may not be called "JDBM" nor may "JDBM" appear in their names without prior written permission of Cees de Groot.
5. Due credit should be given to the JDBM Project (http://jdbm.sourceforge.net/).

THIS SOFTWARE IS PROVIDED BY THE JDBM PROJECT AND CONTRIBUTORS "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL CEES DE GROOT OR ANY CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Appendix J. JDOM Software License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution.
- The name “JDOM” must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact <license AT jdom DOT org<.
- Products derived from this software may not be called “JDOM”, nor may “JDOM” appear in their name, without prior written permission from the JDOM Project Management <pm AT jdom DOT org>.

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgment equivalent to the following:

> “This product includes software developed by the JDOM Project (http://www.jdom.org/)”.

Alternatively, the acknowledgment may be graphical using the logos available at http://www.jdom.org/pdf-images/logos/.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE JDOM AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software consists of voluntary contributions made by many individuals on behalf of the JDOM Project and was originally created by Jason Hunter <jhunter AT jdom DOT org> and Brett McLaughlin <brett AT jdom DOT org>.

For more information on the JDOM Project, please see http://www.jdom.org/.

Appendix K. JGoodies Software License

Appendix L. One-JAR Software License

Appendix M. TreeTable Software License

Resources

Apache Software Foundation, Ant. http://ant.apache.org/

Eclipse Foundation, Eclipse. http://www.eclipse.org/

Oracle, Inc., JDeveloper. http://www.oracle.com/technology/products/jdev

jEdit community, jEdit. http://www.jedit.org

JetBrains s.r.o., IntelliJ IDEA. http://www.jetbrains.com/idea/

Apache Software Foundation, Maven. http://maven.apache.org/

Sun Microsystems, Inc., NetBeans. http://www.netbeans.org/

Bibliography

[Bloch01]
Joshua Bloch. *Effective Java* . Programming Language Guide. Addison-Wesley, 2001. ISBN: 0-201-31005-8.

[Friedl97]
Mastering Regular Expressions . O'Reilly, 1997. ISBN: 1-56592-257-3.

[Kernighan88]
Brian Kernighan. Dennis Ritchie. *The C Programming Language* . Prentice-Hall, 1988. ISBN: 0-13-110362-8.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X