Version

6.7

PREEMPTIVE SOLUTIONS

DASHO

# User's Guide




Manual Version 6.7-09272010
www.preemptive.com

TRADEMARKS

DashO, Overload-Induction, the PreEmptive Solutions logo, and the DashO logo are trademarks of PreEmptive Solutions, LLC

Java™ is a trademark of Sun Microsystems, Inc.

.NET™ is a trademark of Microsoft, Inc.

All other trademarks are property of their respective owners.

THIS PUBLICATION IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD CONTAIN TYPOGRAPHIC ERRORS AND/OR TECHNICAL INACCURACIES. UPDATES AND MODIFICATIONS MAY BE MADE TO THIS DOCUMENT AND/OR SUPPORTING SOFTWARE AT ANY TIME.

PreEmptive Solutions, LLC has intellectual property rights relating to technology embodied in this product. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents or pending patent applications in the U.S. and/or other countries.

This product is distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product may be reproduced in any form by any means without prior written authorization of PreEmptive Solutions, LLC.

# Contents

# Introduction

DashO is a Java obfuscator, compactor, optimizer, and watermarker. This section provides an overview of the benefits of using DashO.

## Why Obfuscate?

Java uses expressive file syntax for delivery of executable code. Being higher-level than binary machine code, class files contain identifiers metadata that makes source code recovery possible. Attackers can use a decompiler to reverse engineer code, exposing software licensing code, copy protection mechanisms, or proprietary business logic.

Obfuscation is a technique that provides seamless renaming of symbols in applications as well as other tricks to foil decompilers. Properly applied obfuscation increases the protection against decompilation by orders of magnitude, while leaving the application intact.

## Goal of Obfuscation

The goal of obfuscation is to create confusion. As the confusion builds, the ability to recover source from class files deteriorates. This says nothing about altering the executable logic - only representing it incomprehensibly.

An obfuscator works at the byte code level to confuse a human interpreter and break decompilers while preserving the executable logic. As a result, attempts to reverse-engineer the instructions fail or produces code that fails to compile.

## DashO Features

PreEmptive Solutions has been protecting and improving intermediate compiled software since 1996, beginning with its DashO tools for Java. Its products for both Java and .NET have enjoyed market-success due to their power, versatility, and patented features.

### Pruning

Starting with entry points into the application DashO determines the classes, methods, and fields that an application uses and creates a package of just those elements. This extends to third-party libraries allowing you to ship only the pieces that your application uses.

## Renaming

DashO uses Overload Induction™, a patented algorithm devised by PreEmptive Solutions. Overload Induction will rename as many methods as possible to the same name. The following example illustrates the technique.

First the original source code:

**Example**

```
private void calcPayroll(SpecialList employeeGroup) {
   while (employeeGroup.hasMore()) {
        employee = employeeGroup.getNext(true);
        employee.updateSalary();
        distributeCheck(employee);
    }
}
```

And now reverse-engineered source after Overload Induction:

**Example**

```
private void a(a b) {
    while (b.a()) {
        a = b.a(true);
        a.a();
        a(a);
    }
}
```

DashO also generates a name mapping file so that obfuscated names can be reapplied between successive releases. This allows patched files to integrate into the previously deployed systems.

## Control Flow Obfuscation

DashO works by destroying the code patterns that decompilers use to recreate source code. The end result is code that is semantically equivalent to the original but thwarts decompilers.

## String Encryption

DashO encrypts strings in all or part of your application, providing a barrier against attackers searching for specific strings in an application to locate logic for registration or serial numbers.

## Byte Code Optimization

Byte code optimizations can be performed on all or part of your application. DashO performs algebraic identity, strength reduction, and other peep-hole optimizations.

## Watermarking

DashO can add watermarks to obfuscated jar files that can be used to track unauthorized copies of software back to the source. Watermarking is used to unobtrusively embed data such as unique customer identification numbers or copyright information into an application without impacting its runtime behavior.

# Getting Started

## Launching the DashO User Interface

Start the DashO user interface by running **dashogui** or **dashogui.bat** in the DashO directory.

**Note**

In Windows, you can start the DashO user interface by clicking **Start > Programs > PreEmptive Solutions > DashO 6.5 > DashO 6.5**.

The user interface is described in Advanced User Interface and Quick Jar User Interface. After you have created a project using the interface you can run it from there or from the command line.

# Registering DashO

The first time you use the product you will be prompted with a registration dialog that will walk you through the process. Fill out the registration form using the serial number provided via email upon purchase confirmation or approval of evaluation. Required fields are highlighted until they have valid information entered.

If you use a proxy server to access the web you may have to enter that information now. In general, DashO will pick up the proxy information from the operating system and you can just click on **Register**.

After your registration is submitted DashO requests you to participate in the Customer Feedback Program. If you are evaluating DashO, then you are automatically enrolled.

You will receive an email confirming your installation.

**Note**

Click **Help > About DashO** to locate your serial number if you need to contact our Support Department. The PreEmptive Solutions Support Department can be reached via phone at (216) 732-5895 ext. 3, or via email at support@preemptive.com.

# DashO FAQ

Frequently Asked Questions regarding DashO may be viewed online at www.preemptive.com/products/dasho/FAQ.html.

# Selecting a Project Type

PreEmptive Solutions has designed DashO to meet the needs in varying situations. There are two principal modes for operating DashO.

1. Advanced (Entry Point) Mode User Interface is best for complex applications or fine-grained control, and where pruning is desired.
2. Quick Jar Mode User Interface is ideal for simple standalone applications with a main method, and where pruning is not required.

Following is a list of criteria to consider when deciding whether to use either the Quick Jar or Advanced mode.

| Criteria | Quick Jar Mode is appropriate if *all* of the following are met | Advanced Mode is appropriate if *any* of the following are met |
|---|---|---|
| **Application Components** | Application or library that consists of only jars. Limited use of reflection. | Application or library that contains jars *and* directories of class files. Uses reflection-based frameworks such as Spring or Hibernate. |
| **Granularity of Control** | Coarse – obfuscations can be turned on or off. | Fine – obfuscations can be turned on or off and particular classes/methods/fields can be excluded from a single obfuscation. |
| **Pruning** | All methods and fields should be retained. | Unused methods and fields should be removed. |
| **Packaging** | Obfuscated classes should retain their original packaging | All obfuscated classes should be placed in a single jar. |

In Quick Jar mode, DashO checks to see if the manifest of each of the input jars contains the `Main-Class` information. The class specified as the `Main-Class` in the manifest is added as an entry point. If none of the input jars have a main class in the manifest, then all classes within the input jars are added as entry points. The entry point or entry points are used by DashO to analyze what classes are required in the input and supporting jars.

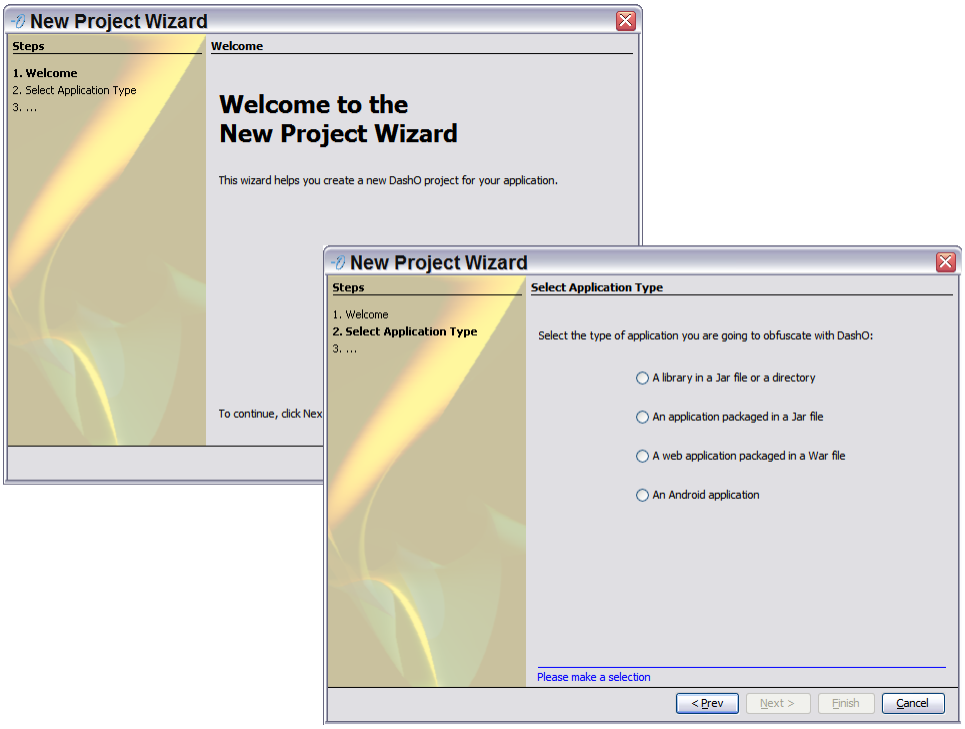
**Note**

In Quick Jar mode, DashO does not remove any classes from the input jars. The output jar has all the classes from all the input quick jars, and DashO may rename these classes. Non-class files from the input jars are automatically included in the output.

Both project types can be built from the graphical user interface or the command line. DashO also provides tasks for Apache Ant and a plug-in for use with the Eclipse IDE.

# New Project Wizard

The easiest way to create a DashO project is to use the New Project Wizard. The wizard examines your application and determines the settings to obfuscate your application. To start the wizard, go to **File > New Project > Wizard**.

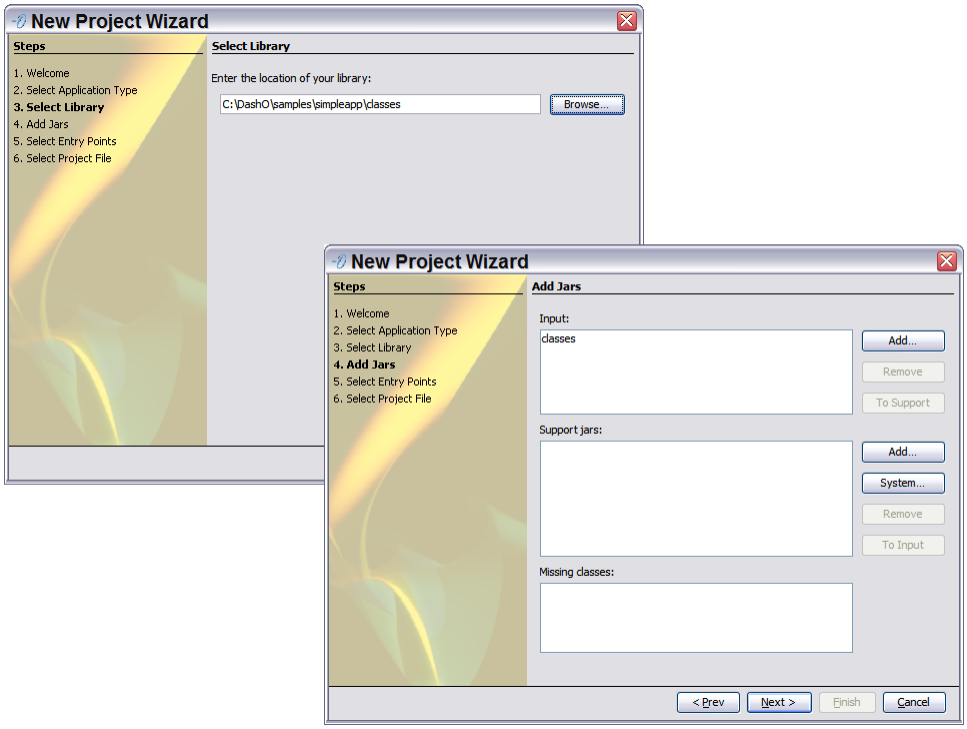



The first step in using the wizard is to characterize your application. Select the type that best describes your application.

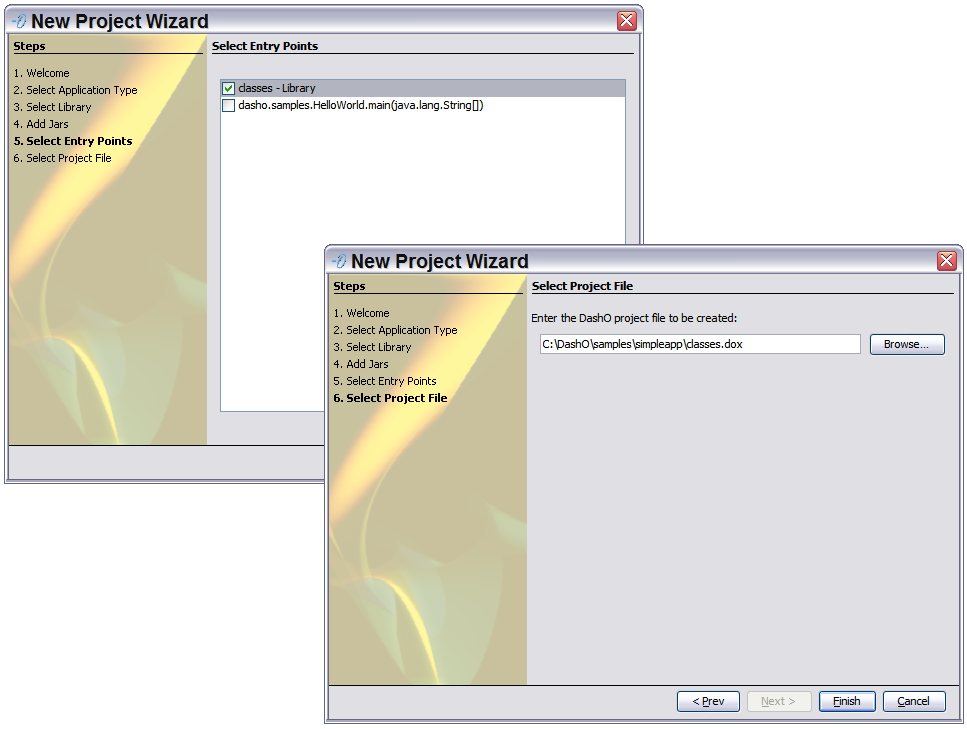
Based on your selection the wizard will ask you a series of questions that are specific to your application type. The following sections show you how to use the wizard for each type of application.

## Library Applications

When you select a library to be obfuscated the wizard will ask you for the location of the jar or directory that contains the library.




The wizard will examine the library and determine dependencies that will be needed at runtime or for obfuscation purposes. You can add additional jars as input to be obfuscated or as runtime support jars. The missing classes list shows classes that are referenced by your library.

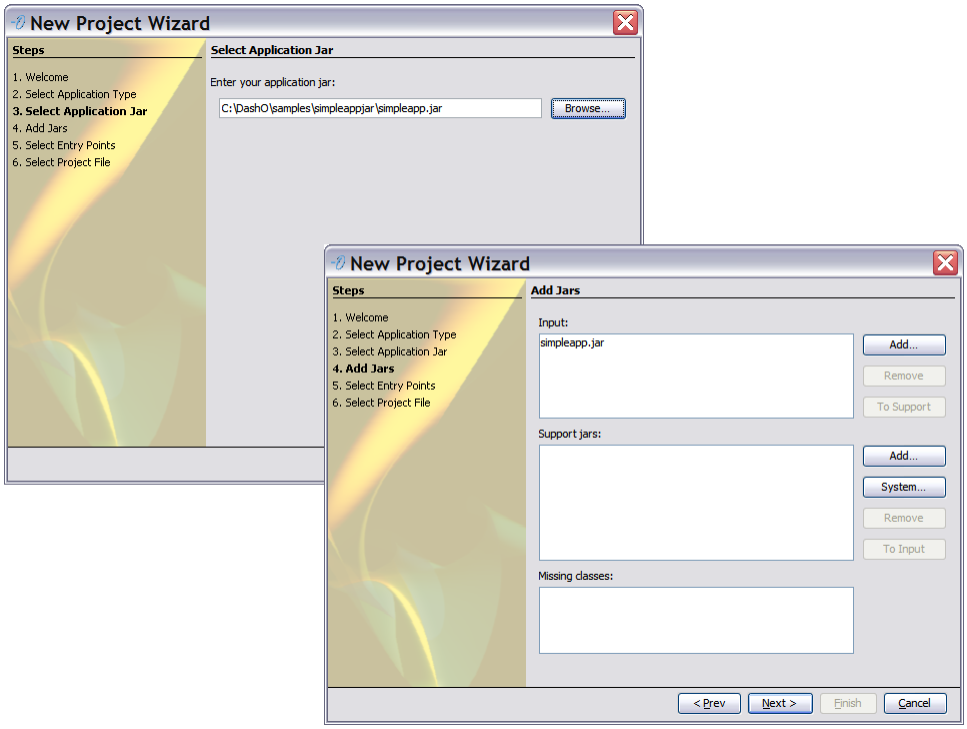



Next the wizard will ask about the entry points in the library. The wizard will show the entire library as an entry point along with any special classes or methods that are used as entry points.

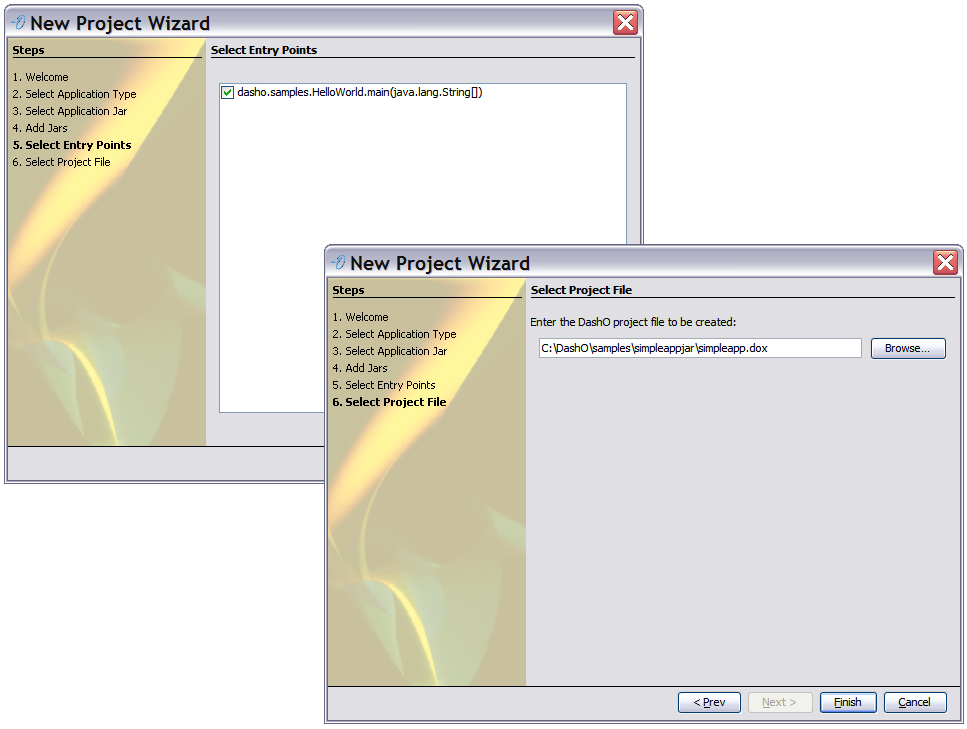
Finally, the wizard asks where you want to save the resulting project file. If you overwrite an existing project file the wizard will update the project file with your new selections. Other obfuscation and Runtime Intelligence settings are preserved.

## Applications in a Jar

When you select an application jar to be obfuscated the wizard will ask you for the location of the jar that contains the application.




The wizard will examine the application and determine dependencies that will be needed at runtime or for obfuscation purposes. You can add additional jars as input to be obfuscated or as runtime support jars. The missing classes list shows classes that are referenced by your application.

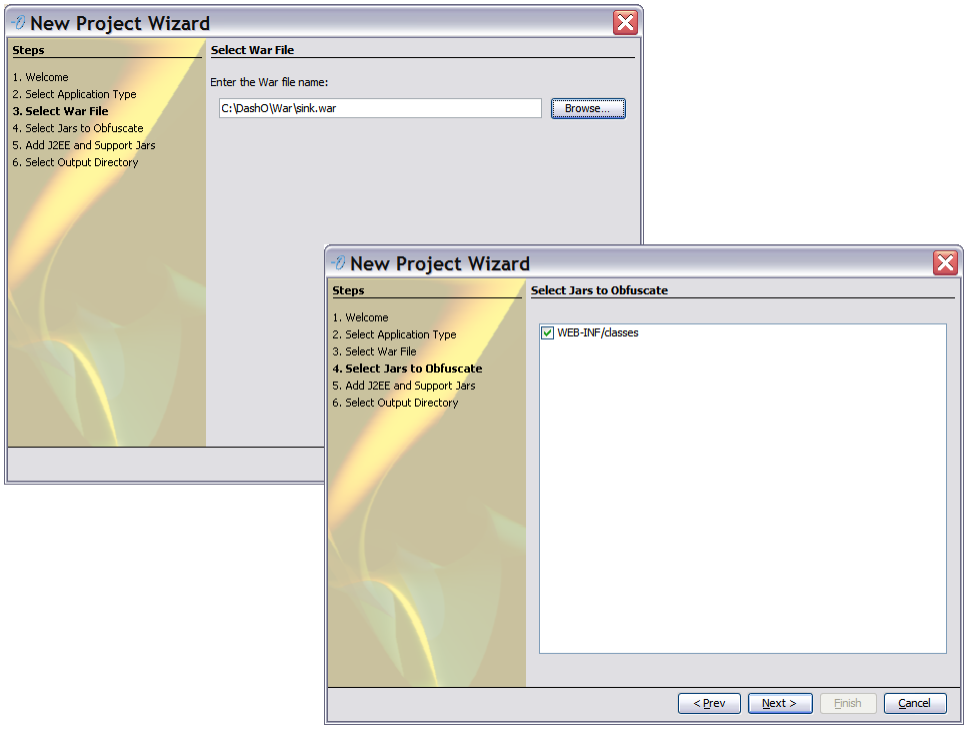



Next the wizard will ask about the entry points of the application. If the jar's manifest included a `Main-Class` attribute it will be listed as an entry point. In addition, the wizard will show the special classes or methods that could also be used as entry points. DashO uses these entry points to determine unused items that will be pruned from the obfuscated output. You can select as many entry points you wish to have DashO follow, but should always select at least one.

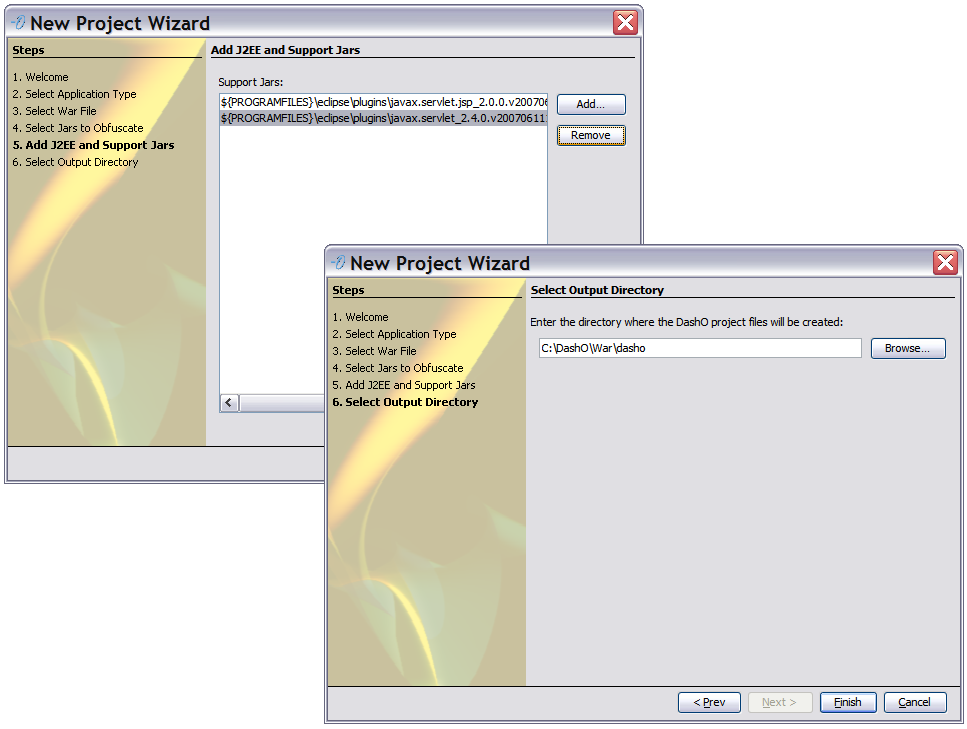
Finally, the wizard asks where you want to save the resulting project file. If you overwrite an existing project file the wizard will update the project file with your new selections. Other obfuscation and Runtime Intelligence settings are preserved.

## WAR Files

When you select a WAR file to be obfuscated the wizard will ask you for the location and name of the WAR to be obfuscated.




The wizard will examine the WAR file for classes and jars that are included in the WAR. These items include the special locations in `WEB-INF` that are used by the web container as well as jars that may be referenced by JNLP files. You can select which items in the WAR file that you wish to obfuscate.




In addition to the jar files that are stored in the WAR file, DashO needs the classes that are part of the Servlet and JSP APIs. The wizard will look for these jars in well known locations and add them to the list of support jars that will not are not obfuscated. If your application expects the web container to provide any other classes shared amongst web application, such as the logging service log4j, you need to add it to the list of jars.

Finally, the wizard asks for the directory where you want to save the wizard's output. The wizard will create several files in addition to a project file:

- `obfuscate.xml`: An Ant script that opens the WAR file, runs the DashO project file, and then re-assembles the WAR file.
- `obfuscate.properties`: A Java properties file read by obfuscate.xml. Use this file to change location defaults.

The obfuscate.xml file can be executed by running Ant:

**Example**

```
ant -f obfuscate.xml
```

or by calling it from another Ant file:

**Example**

```
<ant antfile="obfuscate.xml"/>
```

It performs three tasks:

- It un-WARs the WAR file into a directory. The default directory is `.unwar`.
- It runs the Wizard generated project files against the contents of the `.unwar` directory. Results are temporarily stored in the `.obfus` directory.
- It recreates the WAR with the obfuscated results into a new WAR file with `_dashoed` added to the original file name.

**Note**

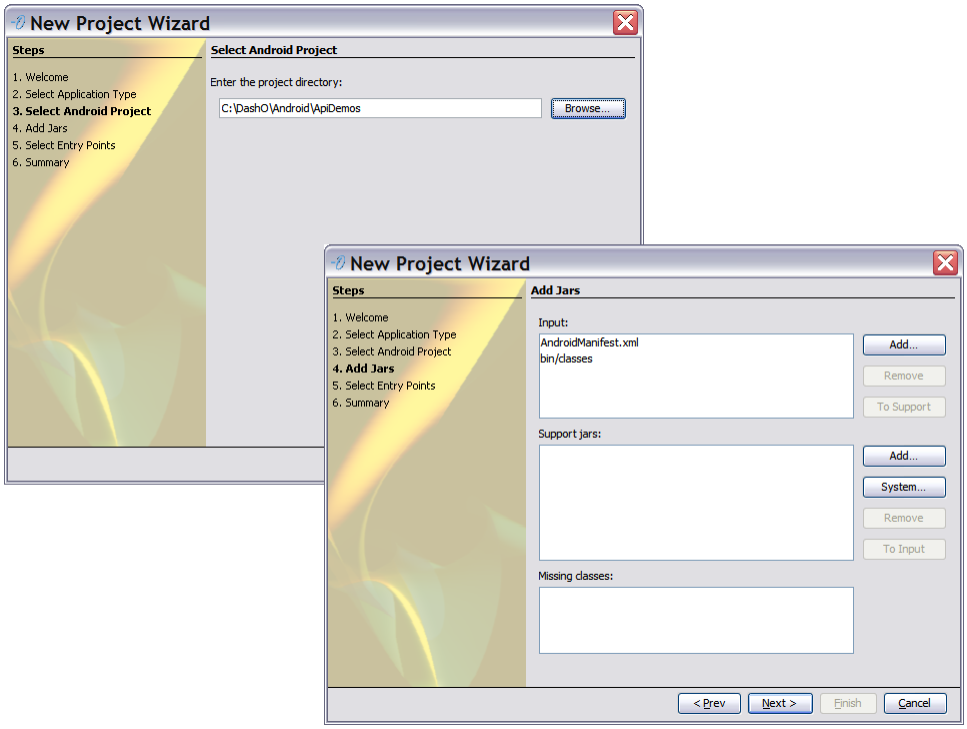
The default memory allocated to DashO processes is 192M. You can change this and other defaults used in WAR file processing by editing `obfuscate.properties`.

If you overwrite an existing project file the wizard will update the project file with your new selections. Other obfuscation and Runtime Intelligence settings are preserved.

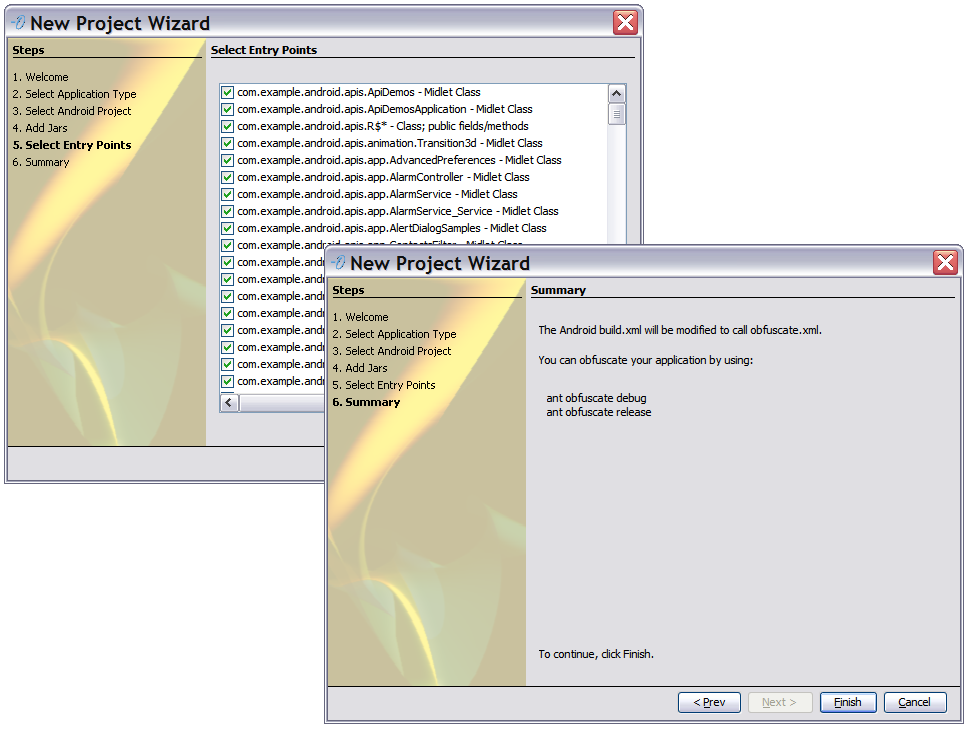
You will need to install DashO's Ant tasks to perform the obfuscation. See the DashO's Ant Task documentation for details.

## Android Applications

When you select an Android application to be obfuscated the wizard will ask you for the location of the Android project. The wizard considers a directory the location of an Android project if it contains an `Android.xml` and `build.xml` file. If you created your Android application using an IDE the `build.xml` may not exist. You can create the required build.xml using `activitycreator` that comes with the Android SDK. Running this in a directory that already contains source for an Android application will not overwrite any of your files.




The wizard will examine the application and determine dependencies that will be needed at runtime or for obfuscation purposes. You can add additional jars as input to be obfuscated or as runtime support jars. The missing classes list shows classes that are referenced by your application.




The wizard analyzes the `AndroidManifest.xml`, the resources, and the compiled classes to determine the entry point of the application. For Android applications you should use all the entry points suggested by the wizard.

The wizard will create an obfuscate.xml file that will be called from a modified version of the `build.xml` created by `activitycreator`. To obfuscate the application you simply need to execute the obfuscate target before the target that you use for building the application:

- `ant obfuscate debug`
- `ant obfuscate release`
- `ant obfuscate install`
- `ant obfuscate reinstall`

You will need to install DashO's Ant tasks to perform the obfuscation. See the DashO's Ant task documentation for details.

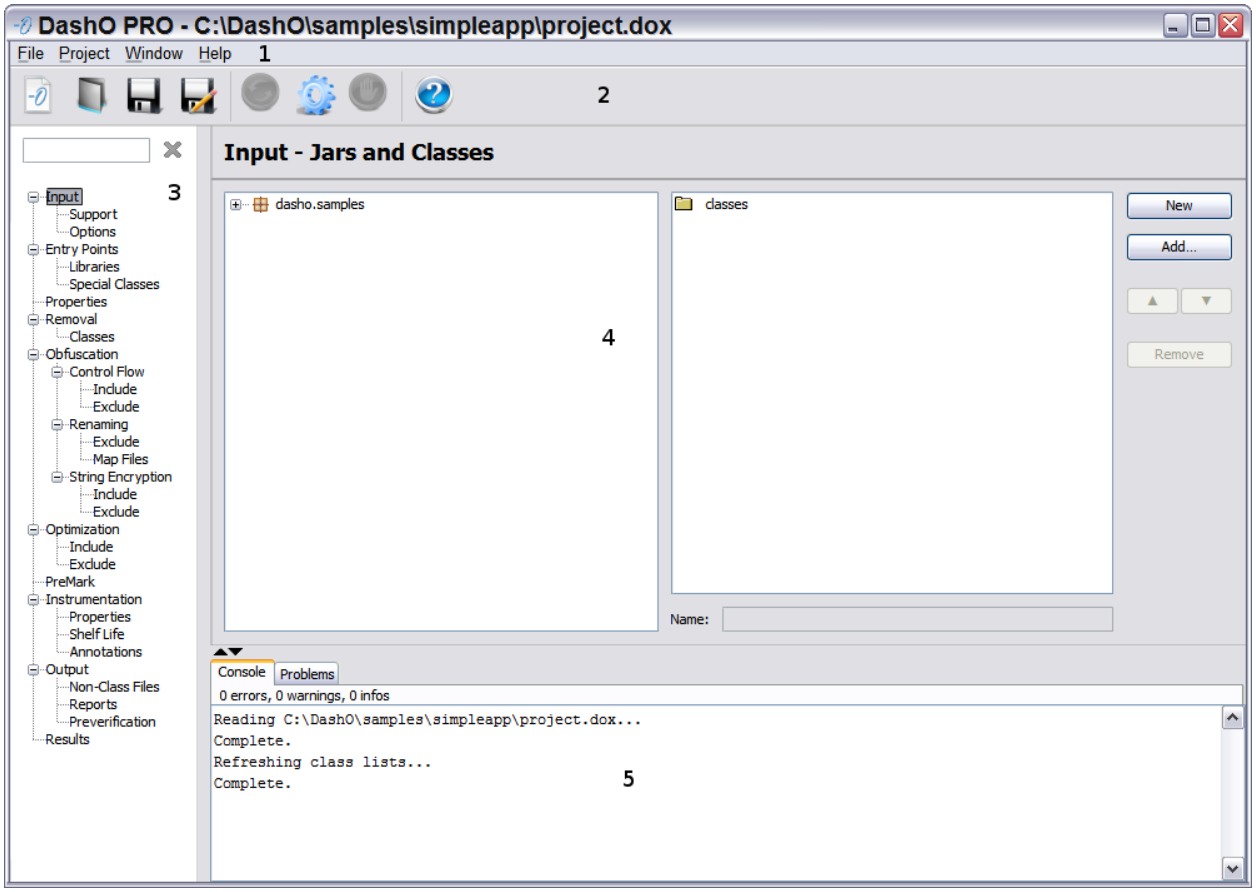
## Building the Project

After the wizard has created the project file it is loaded into DashO's user interface. From the menu, select **Project > Build Project** to produce the obfuscated results.

# User Interface Reference

## The Main User Interface Window

When DashO is launched, the user interface displays on the desktop. The default view, shown below, is the Advanced Mode user interface **Input** panel.




The DashO user interface consists of five activity zones:

| | | |
|---|---|---|
| **1** | **Menu Bar** | At the top, the familiar Menu Bar. |
| **2** | **Toolbar** | Below the Menu Bar is the Toolbar containing icons of frequently accessed actions. |
| **3** | **Navigation Tree** | Below the Toolbar and to the left of the Work Area is the Navigation Tree which organizes the specification and command activities for the Project. |
| **4** | **Work Area** | The Work Area consumes the most real estate in the main window. As the name suggests, this is where work activity occurs. |
| **5** | **Console** | At the bottom, a scrollable console pane is provided for viewing output. |

From the DashO user interface you can select to use one of the user interfaces:

- A jar mode user interface to create and edit DashO projects using Quick Jar entry points.
- An advanced user interface for the DashO projects with the traditional entry points.

## Menu Bar

| Menu Item | Sub Menu Item | Description |
|---|---|---|
| **File** | **New Project** | Create a new Advanced or Quick Jar project, or select the Wizard to have DashO create a project for you. |
| | **Open Project** | Open an existing project. |
| | **Recent Projects** | Open, or see a list of, recently accessed projects. |
| | **Save Project** | Save a new or modified project. |
| | **Save Project As** | Save an existing or modified project with a different name and/or in a different location. |
| | **Exit** | Exit and close DashO. |
| **Project** | **Build Project** | Obfuscate a project. |
| | **Cancel** | Only available when building or refreshing a project. Cancel enables you to cancel a build or a reload. |
| | **Reload Class List** | Available when new classes are added to a project. |
| | **Convert** | Convert a Quick Jar project into an Advanced project. |
| | **View Project** | Only available when a project is saved after modification or creation. Enables you to view an existing project as a text file. |
| | **View Report File** | Only available when a report is saved after modification or creation. Enables you to view an existing report. |
| | **View Renaming Report File** | Only available when a renaming report file is saved after modification or creation. Enables you to view an existing renaming report file. |
| **Window** | **Decode Stack Trace** | Recover the stack trace from an obfuscated program. See Decoding Stack Traces. |
| | **Shelf Life Token** | Create a Shelf Life token that can be saved to a file. See Generate Shelf Life Token. |
| | **User Preferences** | Select general and DashO Engine options. See User Preferences. |
| **Help** | **Help** | Instant access to DashO assistance. |
| | **Register Product** | Only available during the registration process. Enables users to register their version of DashO. |
| | **Check for Updates** | Check to ensure you have the most recent version of DashO. |
| | **Customer Feedback Options** | Participate in DashO's anonymous customer feedback program. |
| | **About DashO** | Provides information about the installed copy of DashO. |

## Toolbar

| Icon | Icon Name | Function |
|---|---|---|
|  | **New Project** | Click to create a new project. |
|  | **Open Project** | Click to open an existing project. |
|  | **Save Project** | Click to save a new or modified project |
|  | **Project Save As** | Click to save an existing or modified project with a different name and/or in a different location. |
|  | **Reload Class List** | Click to refresh the class list when new sources are added to it. |
|  | **Build Project** | Click to obfuscate a project. |
|  | **Cancel Build** | Click to cancel an in-progress project build |
|  | **Help** | Click to access to this User Guide. |

## Work Area

The content of the Work Area is dependent upon the item selected in the Navigation Tree.

**Note**

The Work Area contains toggle buttons. The ▼ enables the user to *increase* the size of the Work Area by *collapsing* the Console. The Console can be expanded by clicking the ▲.

## Console

The console area contains two tabs:

- **Console** – Displays progress of project creation and build and provides a count of errors, warnings, and informational message.
- **Prolbems** – Lists any informational, warning, and error messages encountered during obfuscation.

# Advanced Mode User Interface

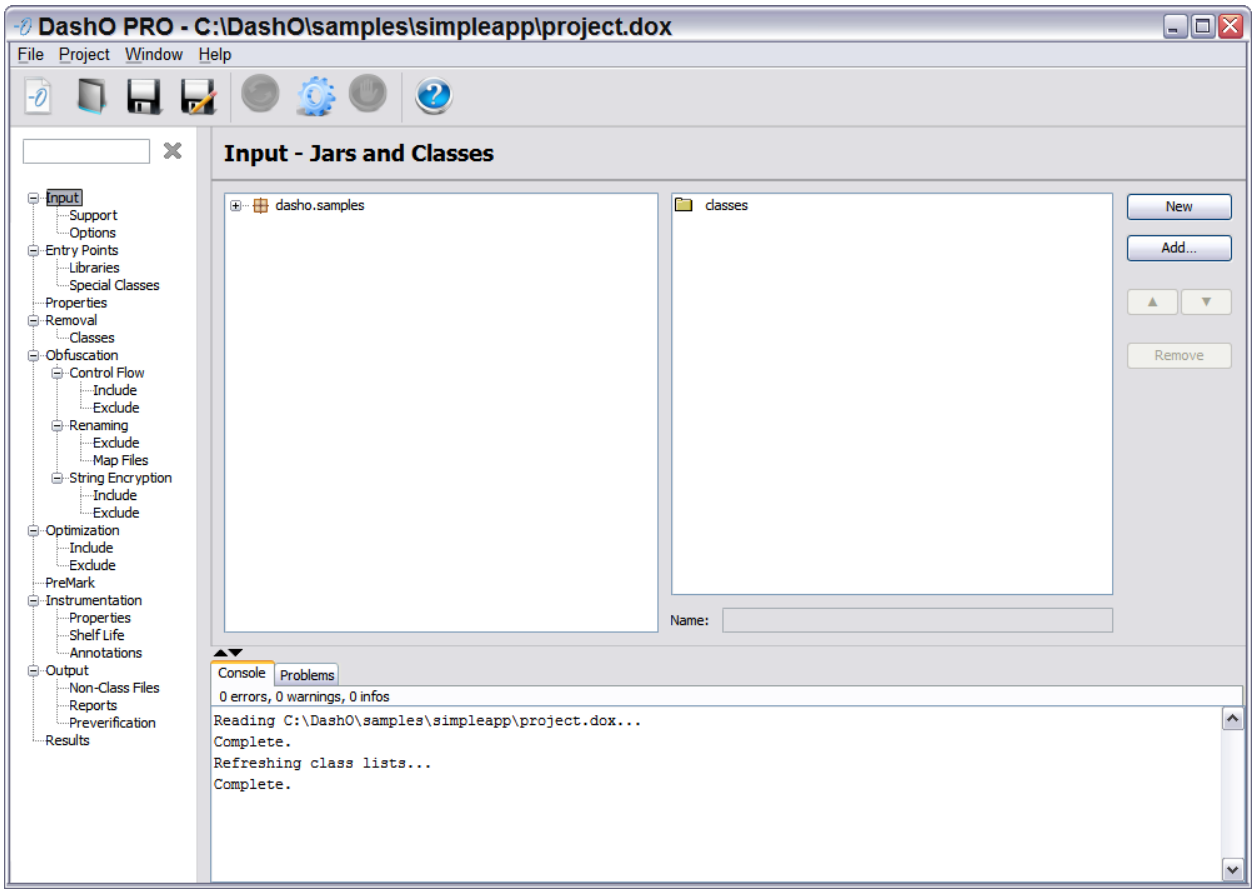
In this section, we describe how to use DashO's interface for advanced mode projects. You can use the interface to create new projects or edit existing ones. The resulting project can be saved and used later by the command line interface, Ant, or you can obfuscate within the interface and view the results.

## Input Section

The *Input Section* is used to configure the input to the project. This includes the location of jars and directories of classes that will be processed and entry points into these classes that are used to analyze the dynamic flow of the application.

### Input – Jars and Classes

The *Input Section* starts with the locations for the classes to be processed. DashO can handle directories or classes, zip files, and jar files in the classpath. Entries may be added by selecting them from the file system using the Add button. You can also create an entry by using the **New** button and editing its name. After adding or removing items from the input use the refresh class list item in the toolbar or from the menu.

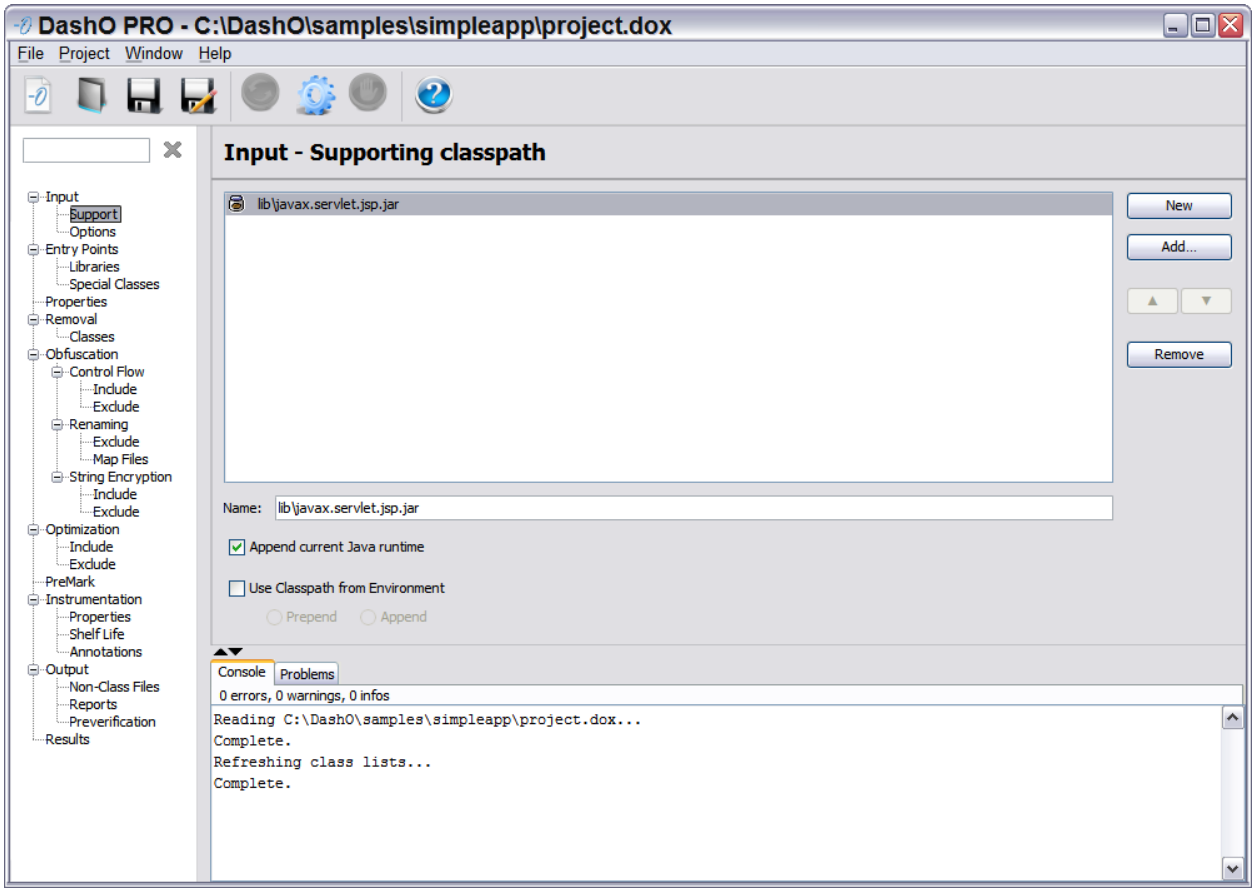



See the <inputpath> Section for more information regarding the creation of input entries.

## *Input – Supporting Classpath*

DashO needs access to classes in the Java runtime and to classes in third party jars. The classes referenced here are needed for DashO's analysis but are not processed. Entries may be added by selecting them from the file system using the **Add** button. You can also create an entry by using the **New** button and editing its name.

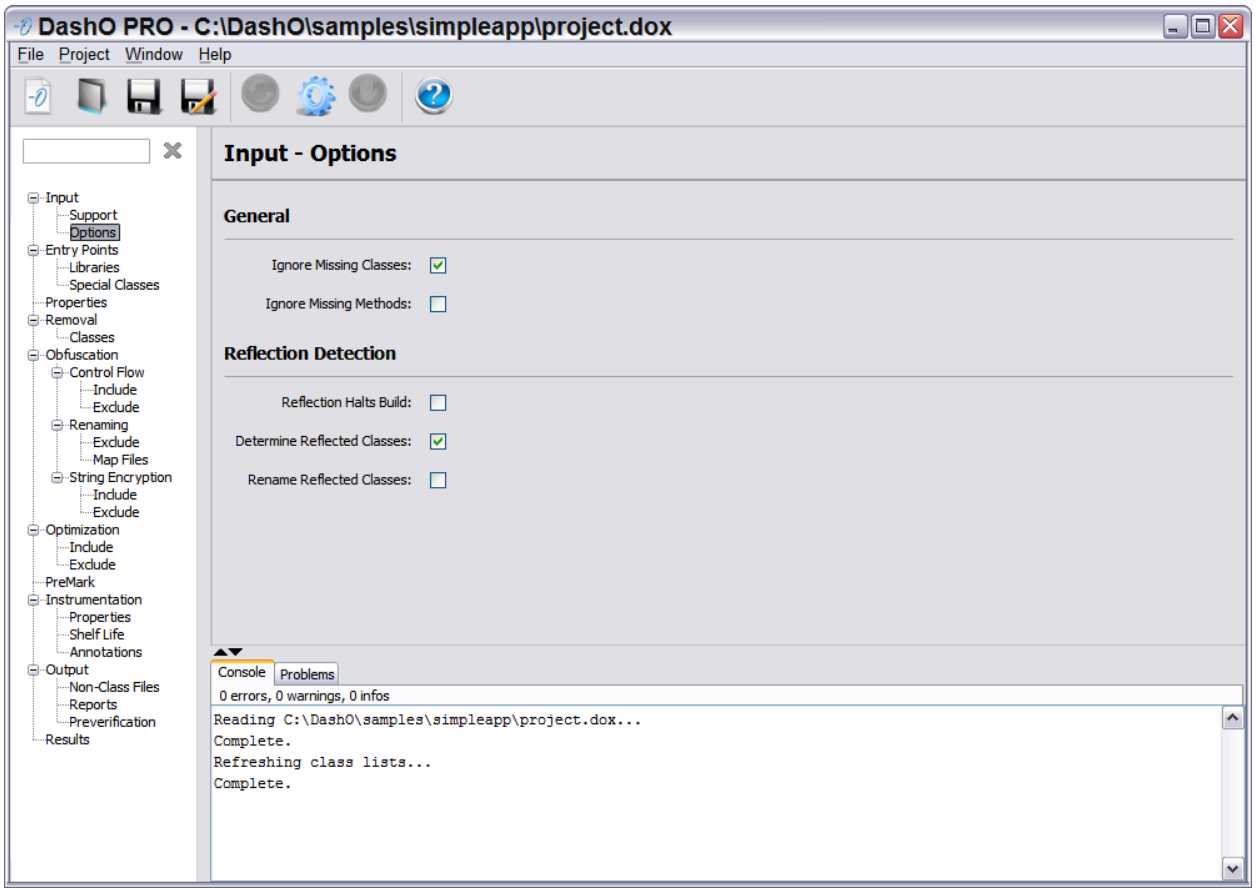
By default the location of the Java runtime used by DashO is added to the path. Projects that use J2ME or the Android API should *not* append the runtime jar to the classpath. These projects require the runtime jar for these particular environments: e.g., `midpapi10.jar` or `Android.jar`. You may also `append` or `prepend` the environmental classpath to the provided entries.




See the <classpath> Section for more information regarding the creation of new classpath entries.

## Input Options

This panel controls some basic options that DashO uses while analyzing the input classes.




### *Ignore Missing Classes*

DashO will attempt to analyze all classes that the application attempts to call. You can instruct DashO to ignore these classes by selecting this option. Note that DashO cannot skip classes and interfaces that the application extends or implements.

### *Ignore Missing Methods*

DashO will attempt to locate concrete implementations of methods as part of its analysis. Turning this option on lets DashO proceeded even if it cannot locate the desired method. Use this option with caution.

### *Reflection Halts Build*

DashO's analysis makes note of reflection usage in the application so that the targets of reflection can be identified. Turn this option on when you are determining what parts of the application use reflection.

### *Determine Reflected Classes*

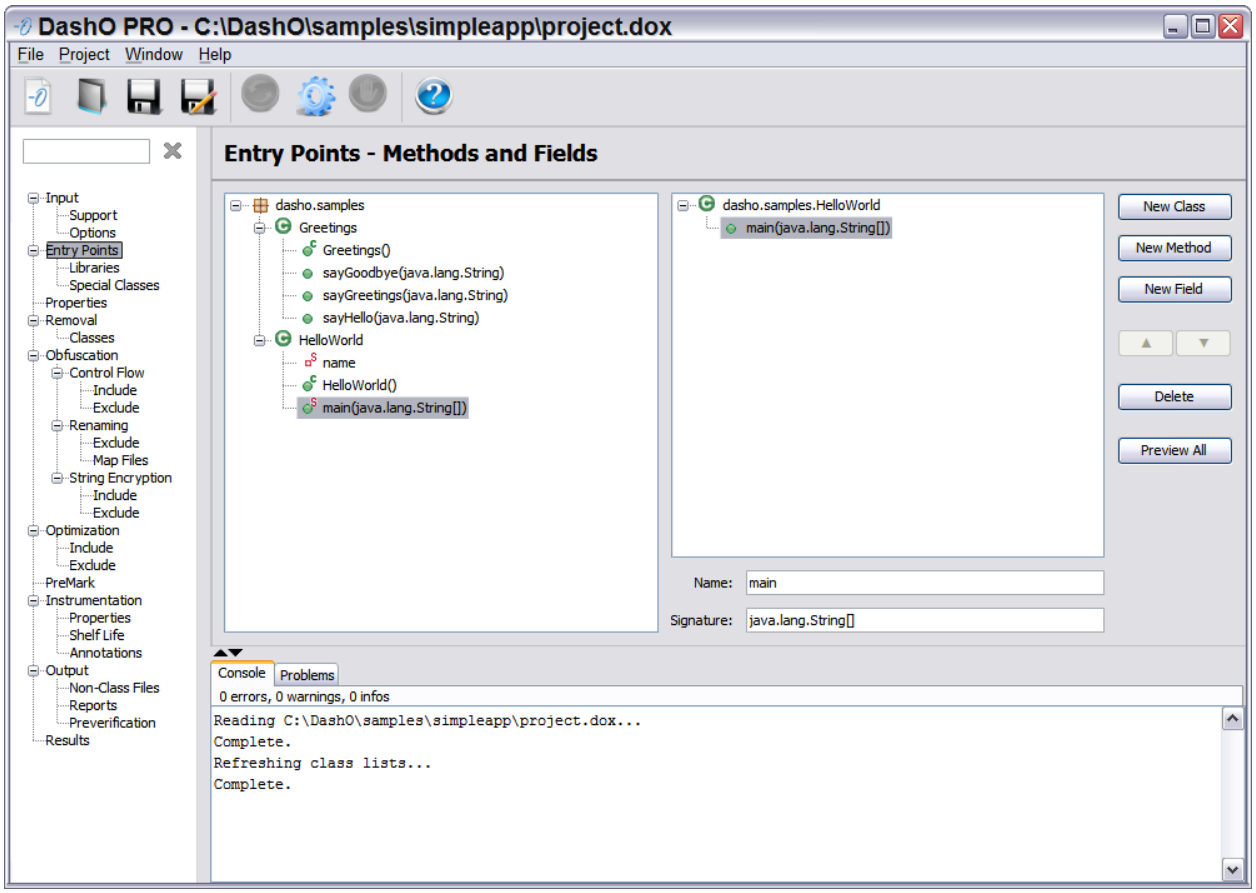
DashO can determine some targets of reflection and automatically add make sure that these classes appear in the output. Note that this processing can increase the build time.

### *Rename Reflected Classes*

By default targets of reflection are not renamed. Use this option to allow these classes to be renamed.

## Entry Points – Methods and Fields

Fields and methods are used to indicate entry points into the application. DashO's analysis begins at these locations and is used to traverse the call graph of the application. This allows DashO to prune unused classes and members. Methods and fields that are used as entry points are non-renameable by default. The class and/or member can be made renameable by right-clicking on the item to bring up its properties, and checking **Rename** item.




See the Using the Graphical Rules Editing Interface section to compose rules that define method and field based entry points.

### *Conditional Including*

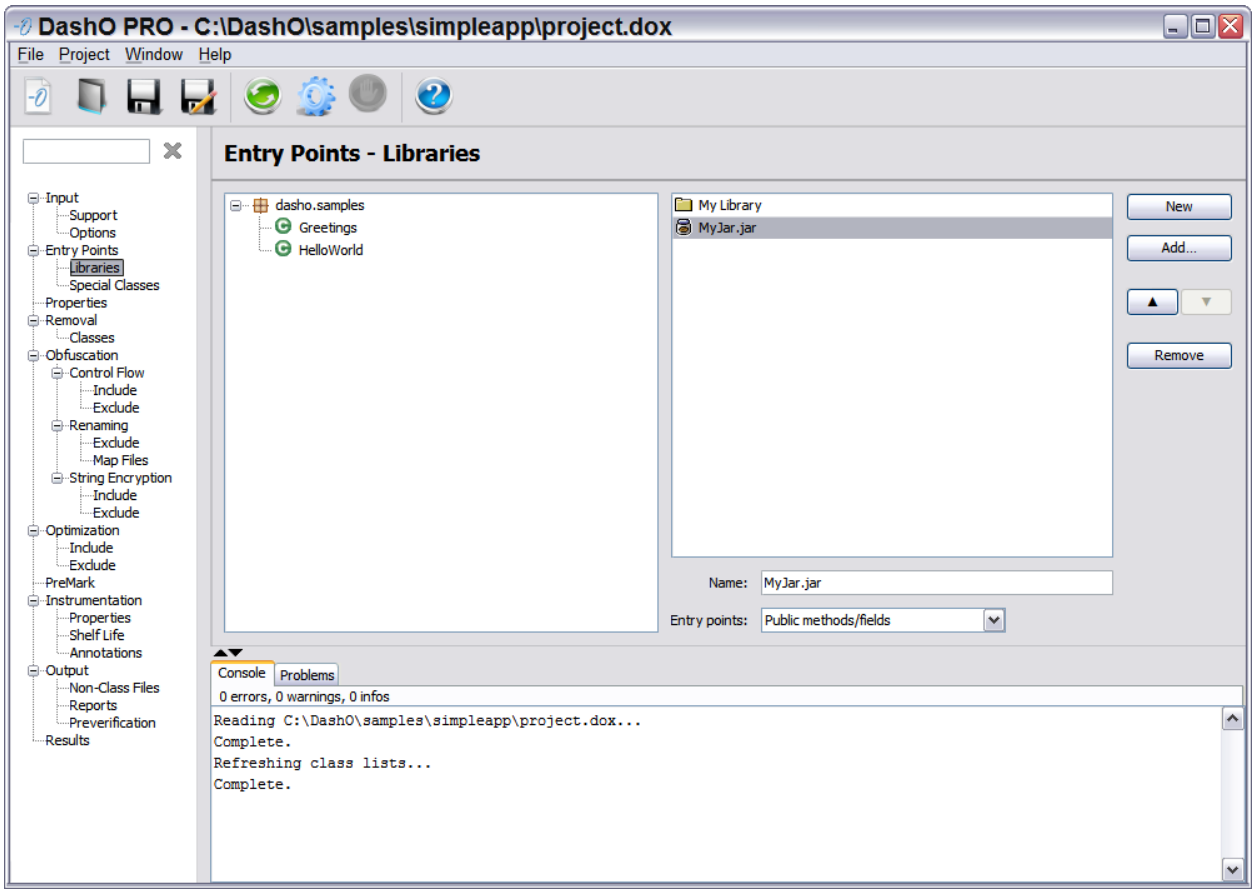
It is sometimes necessary to manually include class files into the project. If the `Class.forName()` construct is used anywhere in the project, DashO will be unable to determine all possible classes that might be needed. In this case, any classes that will be referenced in the `forName()` construct must be manually included as entry points. These classes should not be renameable. See Advanced Topics for more details on `forname` detection.

**Note**

If no entry points are defined DashO will see if it can find entry points in the Manifests of input jars. If none are found the it defaults to *library mode* where all public and protected classes and members are used as entry points.

## Entry Points - Libraries

Jars or directories of classes can be used as a library entry point. DashO uses all public members of the classes as non-renameable entry points. Optionally, protected members can be added as non-renameable entry points.




Libraries may be added by selecting them from the file system using the **Add** button. You can also create a library entry by using the **New** button and editing its name. The names of library entries can contain property references.

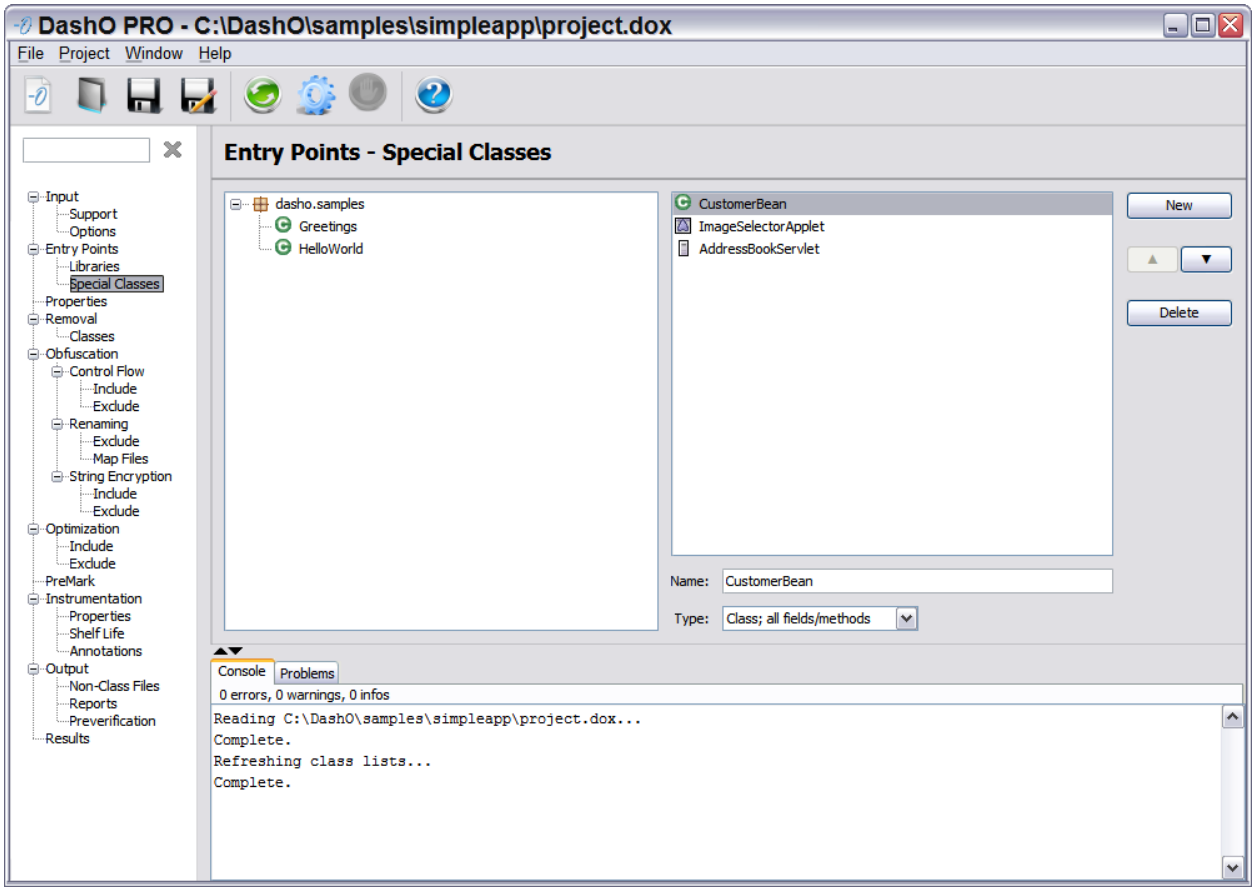
See the section <library> Entry Point for details concerning library entry points.

**Note**

Jars or folders added as libraries do *not* need to be added to the input list. Libraries are combined with the input list to determine the classes to be processed. When adding or removing library entries you can use the refresh option from the toolbar or menu to update the list of input classes.

## Entry Points - Special Classes

A special class entry point allows the specification of a class that contains the implementation of interfaces or extensions of a class that define an entry point into an application. These entry points are typically defined by frameworks for such things as J2ME or Applets. The names of these classes can be specified as an exact match, a pattern, or a by a regular expressions.




By default special classes are non-renameable. The class, and in most cases its members, can be made renameable by right-clicking on the item to edit its properties and check the **Rename Class** or **Rename Members** item.

### *Applets*

For DashO, an applet is a class that directly or indirectly extends `java.applet.Applet`. The applet's class can be made renameable, but the methods defined by `java.applet.Applet` are not renameable. See the <applet> section for details.

### *Servlets*

For DashO a servlet is a class that directly or indirectly implements `javax.servlet.Servlet`. The servlet's class can be made renameable, but the methods defined by `java.applet.Applet` are not renameable. See the <servlet> section for details.

### *Enterprise JavaBeans - EJBs*

Enterprise JavaBeans are server-side components written in Java that can be used to write distributed object-oriented enterprise applications. For DashO's purposes an EJB is any class that extends the interfaces defined in the `javax.ejb` package including the bean's home and key classes. See the <ejb> section for details.

### *Midlet and iAppli*

A Midlet is a Java class that runs on embedded devices using Java ME, CLDC, or MIDP. The Midlet class should extend `javax.microedition.midlet.Midlet` directly or indirectly. iAppli classes are similar but use the NTT DoCoMo's iAppli framework and extend `com.nttdocomo.ui.IApplication` either directly or indirectly. The midlet's and iappli's classes can be made renameable, but the methods defined by `java.applet.Applet` are not renameable. See the <midlet> and <iappli> section for details.

### *Class public fields/methods*

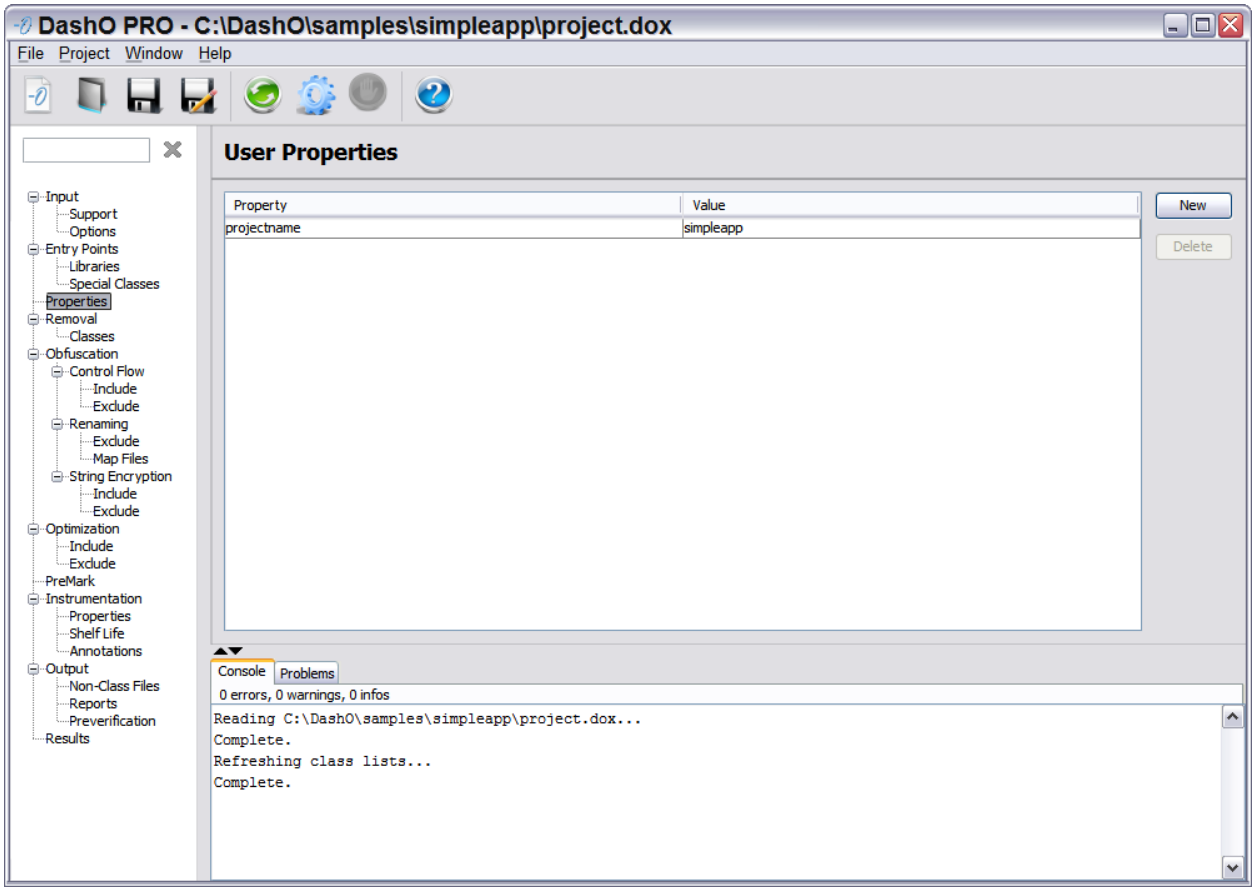
DashO uses all public fields and methods in the classes as entry points. The class and its public members will not be renamed. See the <publics> section for details.

### *Class all fields/methods*

DashO uses all fields and methods in the classes as entry points. The class and all its members will not be renamed. Specifying classes in this manner performs an unconditional include of the class. See the <unconditional> Entry Point section for details.

## *Options - User Properties*

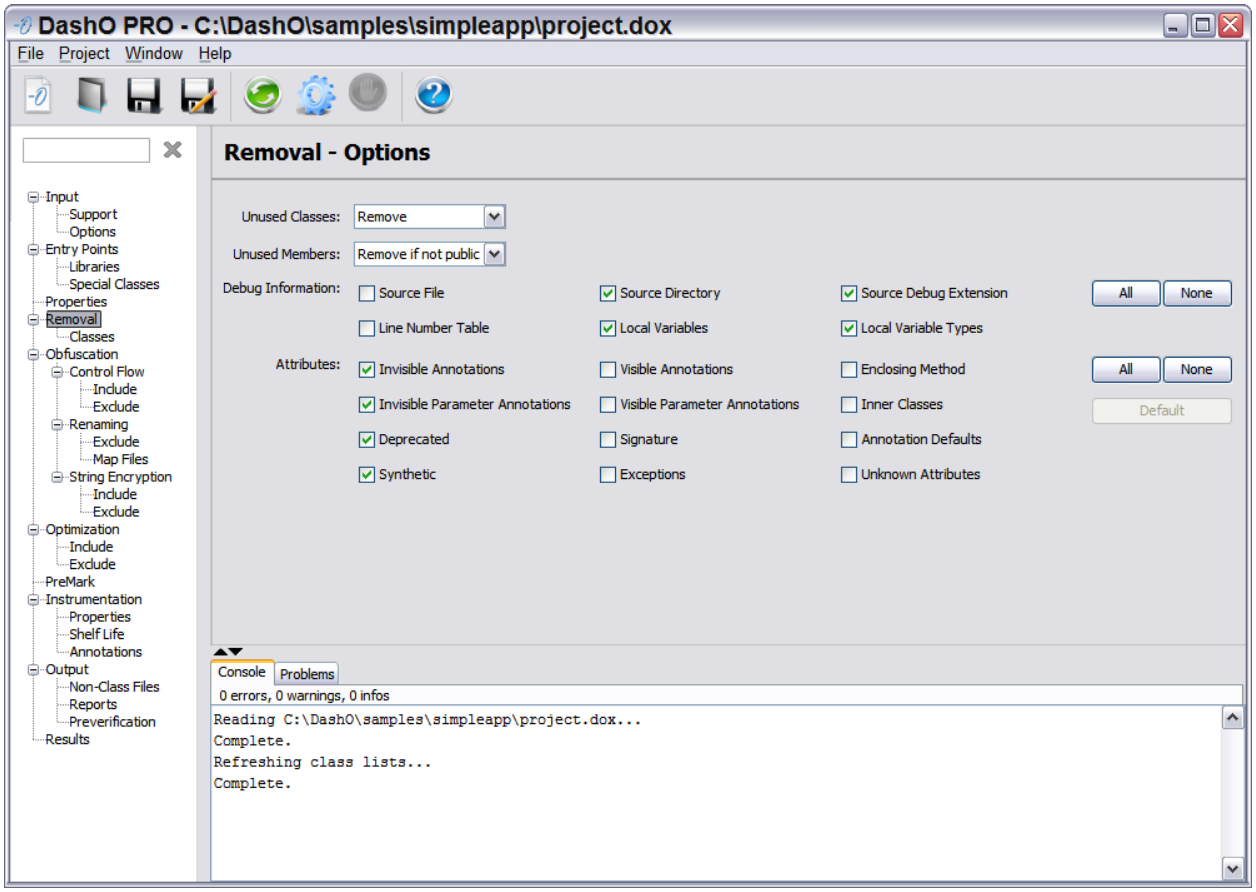
The *User Properties* panel lets you create and assign values to properties that can be referenced in the project. This can allow you to create a project that acts as a template. Properties may be defined in the terms of other properties, manipulate the value of other properties, or provide default values. The value of a property may be specified using one or more property references including to references to environment variables. These property references can include default values, indirection, or substitution syntax. See Property References for details. Recursive property definitions are not allowed.




See the <propertylist> section for information about using properties in your project.

# Removal – Options

The *Removal Options* panel control which what happens to unused classes and members in the input and the removal of metadata.




## *Unused Classes*

This controls the handling of unused classes. Options are to remove all unused class, only those that are not public, or to perform no removal at all. See the section on <removal> for details.

## *Unused Members*

This controls the handling of unused methods and fields. Options are to remove all unused members, only those that are not public, or to perform no removal at all. See the section on <removal> for details.

## *Debug Information*

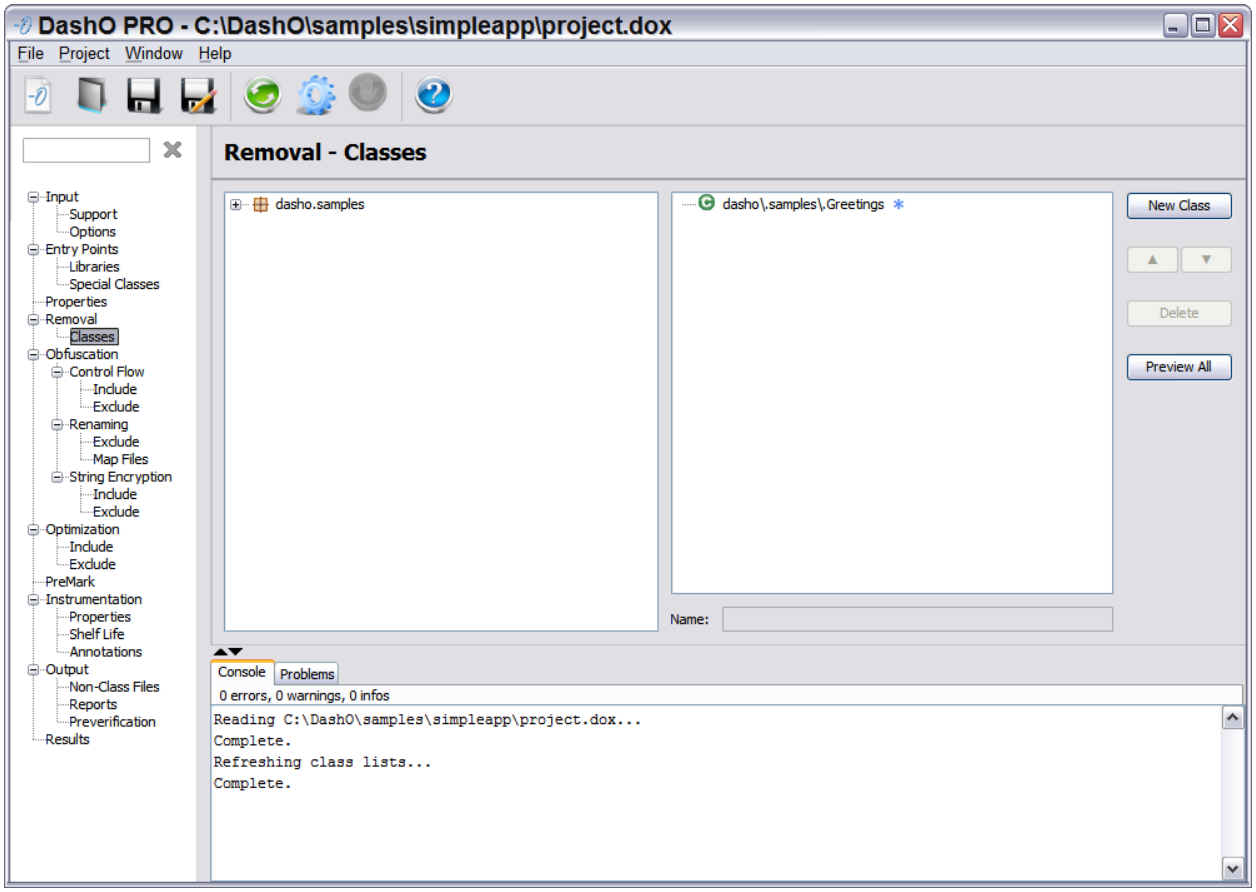
This controls the removal of debugging information inserted by the compiler. Most information is for use by debuggers, but the most useful to retain are line numbers and the source file. To generate a stack trace with line number these two should be retained. See <debug> Section for details.

## *Attributes*

The Java compiler generates additional meta data for classes and their members and stores that information in attributes in the class files. Some of this information is required when for the compiler when you compile against a library or by applications using reflection. You can use these settings to selectively remove information that your application does not require at runtime to reduce the size of your class files. See <attributes> Section for details.

## Removal – Classes

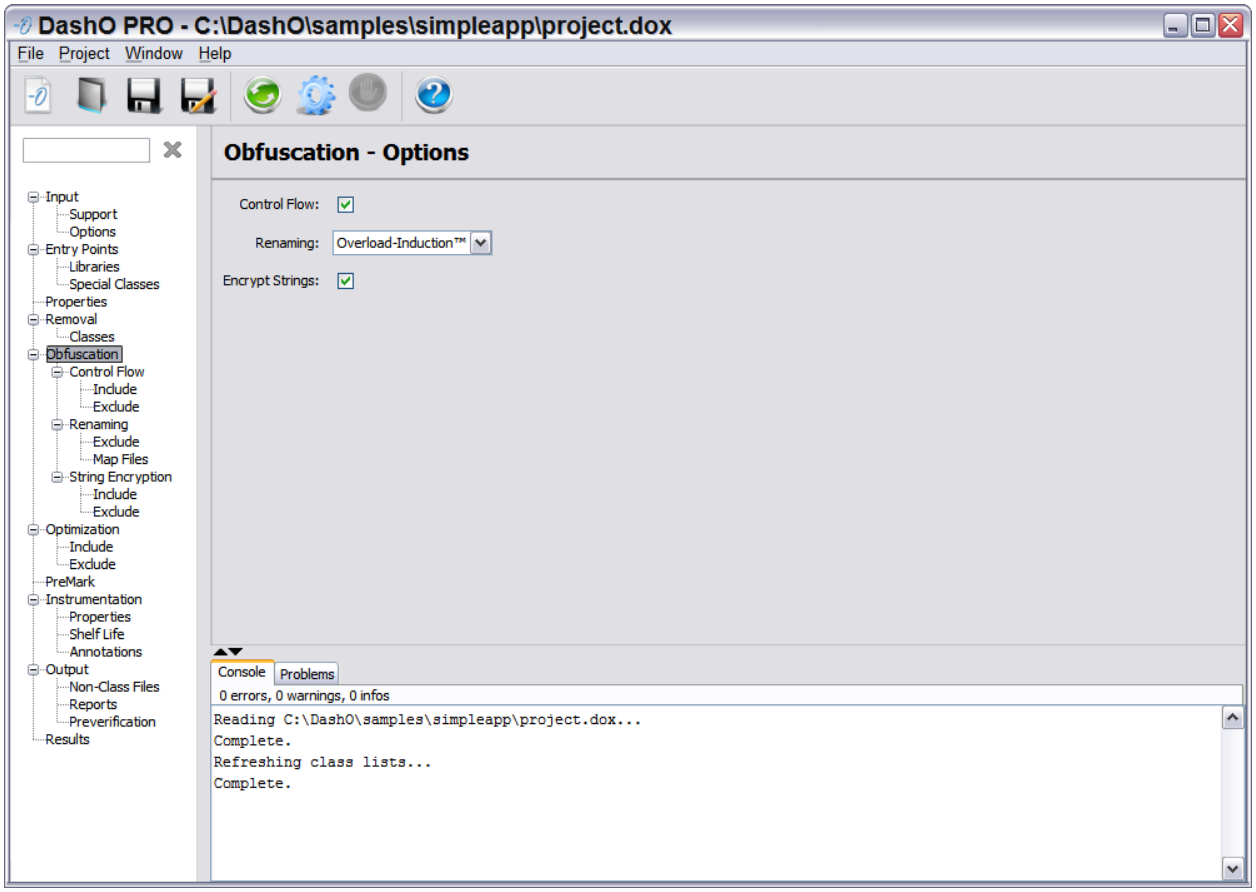
If your inputs contain classes that you do not want to appear in the resulting output, such as unit tests or samples, you can have DashO remove them. Classes matched by these rules will not appear in DashO's output. If any other input classes reference then they will treated as if they were support classes.




You can create rules that exclude individual or groups of classes or even entire packages using regular expressions. See Graphical Rules Editing Interface for details.

# Obfuscation - Options

The *Obfuscation Options* panel controls the basic obfuscation setting for your project. Other panels under obfuscation allow you to change the specifics of each action and applying the obfuscation technique to all or part of your application.




## *Control Flow*

Enables or disables control flow obfuscation globally. You can control the portions of the application to which control flow is applied by using include and exclude rules. If you do not specify any rules then all methods will have control flow applied.

## *Renaming*

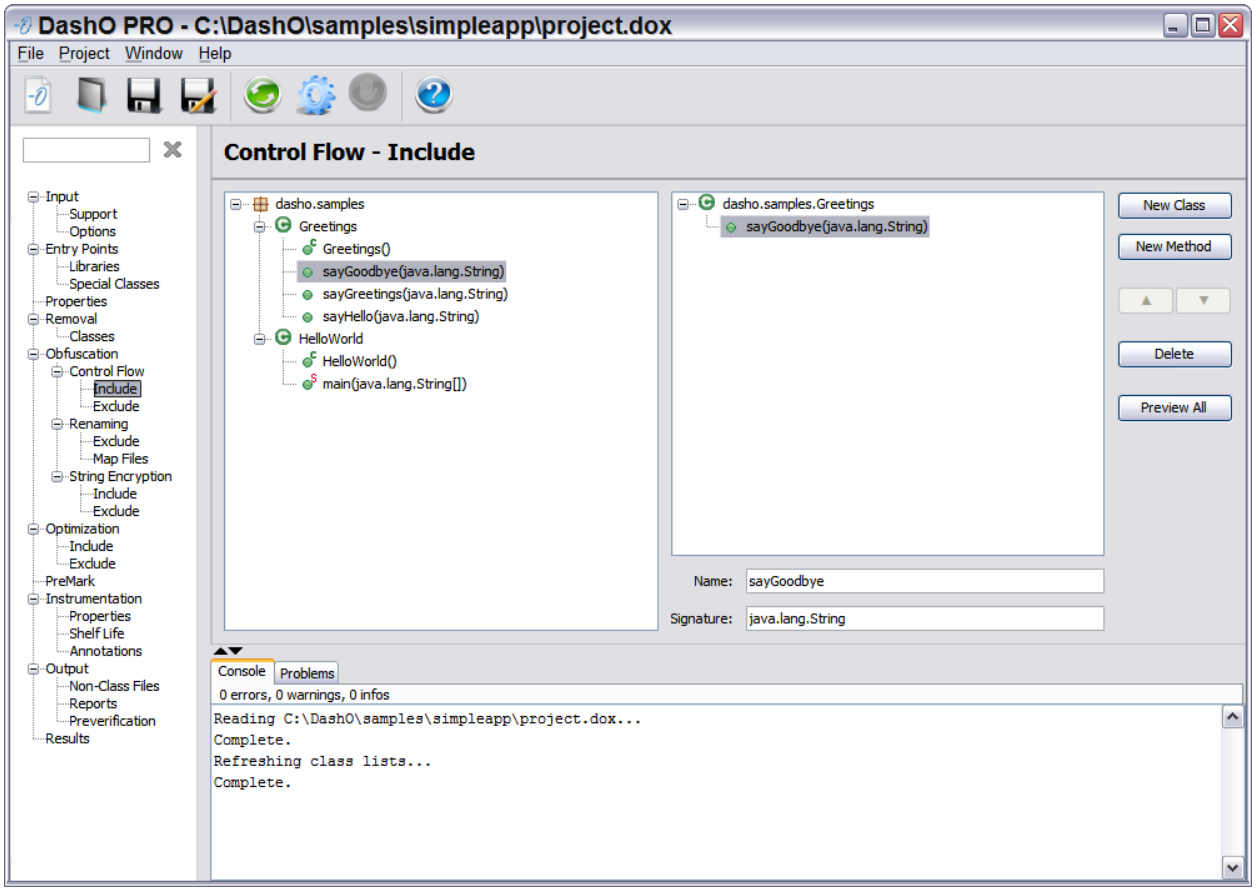
Enables or disables renaming of classes, methods, and fields globally. You can control the portions of the application to which renaming is applied by using exclude rules as well as controlling the renaming of packages, classes, and methods. Overload Induction™ renames method based on method signatures to produce many methods with the same name. Simple renaming renames the methods so that there is no overloading.

## *Encrypt Strings*

Enables or disables string encryption obfuscation globally. You can control the portions of the application to which string encryption is applied by using include and exclude rules. If you do not specify any rules then all methods will have their strings encrypted.

## Control Flow – Include and Exclude

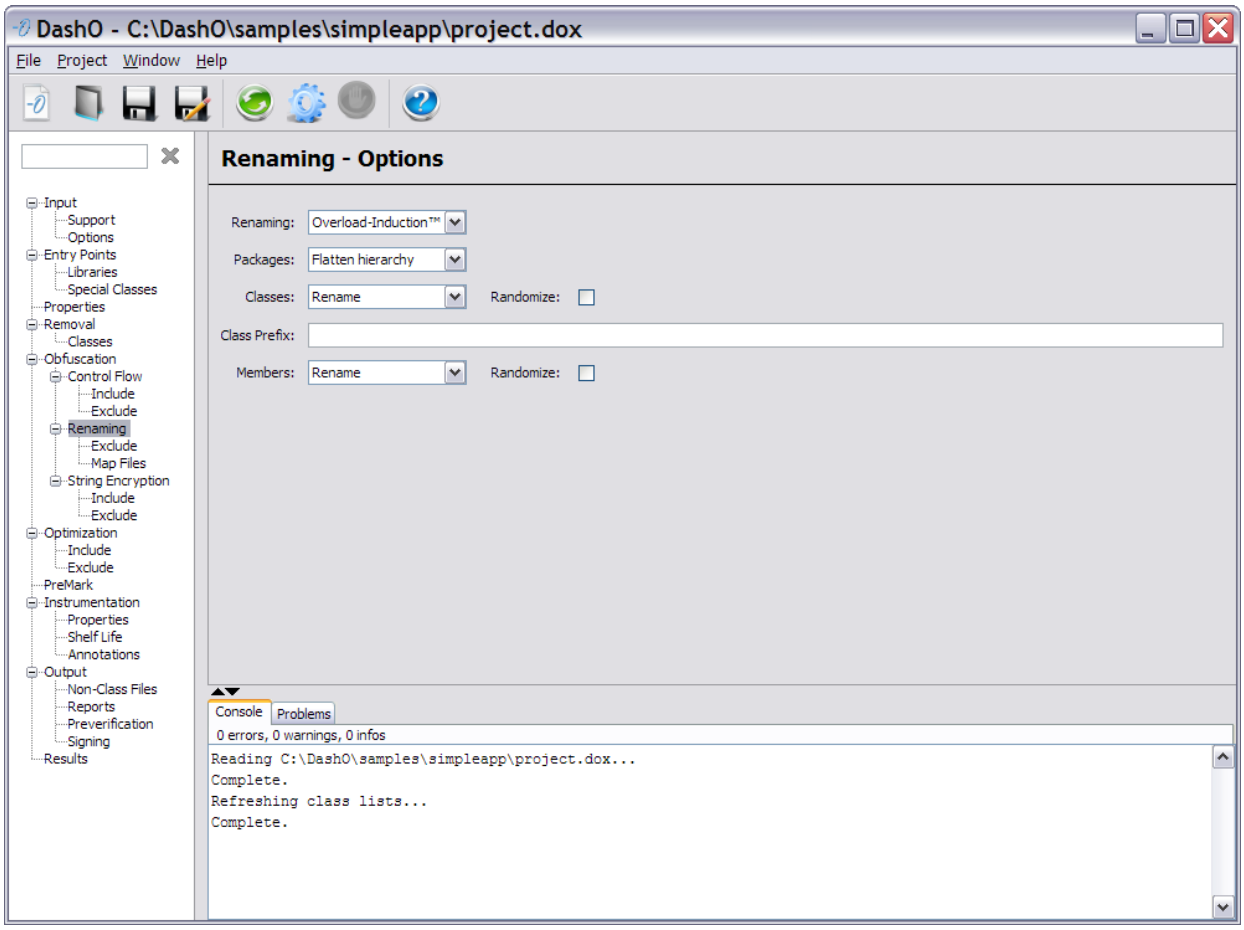
The *Control Flow Include and Exclude* panels lets you compose rules that determine which parts of the application will have control flow obfuscation applied to methods. Methods, classes, or entire packages may be selected. Items should be excluded if you are concerned about possible performance issues.




See Graphical Rules Editing Interface for details.

## Renaming - Options

When renaming has been enabled the *Renaming Options* panel gives you additional control over the renaming of items in your application.




### *Renaming*

Enables or disables renaming of classes, methods, and fields globally. You can control the portions of the application to which renaming is applied by using exclude rules as well as controlling the renaming of packages, classes, and methods. Overload Induction™ renames method based on method signatures to produce many methods with the same name. Simple renaming renames the methods so that there is no overloading.

### *Packages*

When classes are renamed you can specify if the package hierarchy should be flattened[1] or if the package naming hierarchy is retained.

### *Classes*

You can elect to rename classes or to keep their original names. This provides for a very course level of control – you can use exclusions to preserve the names of individual classes, whole packages, or

[1] This option puts all renamed classes into the default package.

classes that meet certain criteria. When randomize is selected new class names are assigned in a random fashion from the list of shortest available identifiers.

### *Class Prefix*

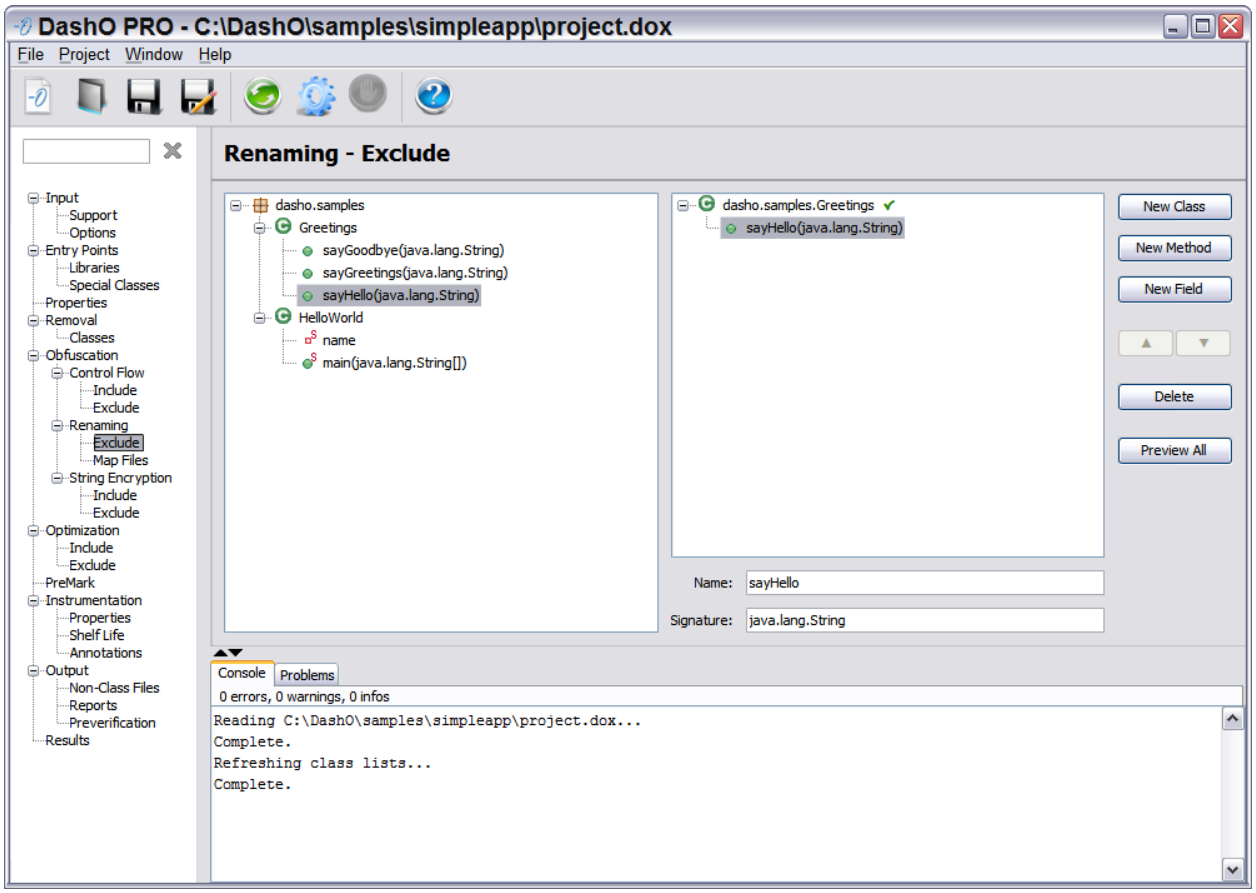
When a class is renamed you can add an optional prefix to the new name. You can use periods in the prefix to place the renamed classes into a different package.

### *Members*

You can elect to rename all methods and fields or to retain the names of public members. This provides for a very course level of control – you can use exclusions to preserve the names of particular methods based on their names, arguments and other criteria. When randomize is selected new method and field names are assigned in a random fashion from the list of shortest available identifiers.

## *Renaming – Excludes*

The *Renaming Excludes* panel lets you compose rules that exclude classes and/or their methods and fields from renaming. Individual methods, fields, classes, or entire packages may be excluded.

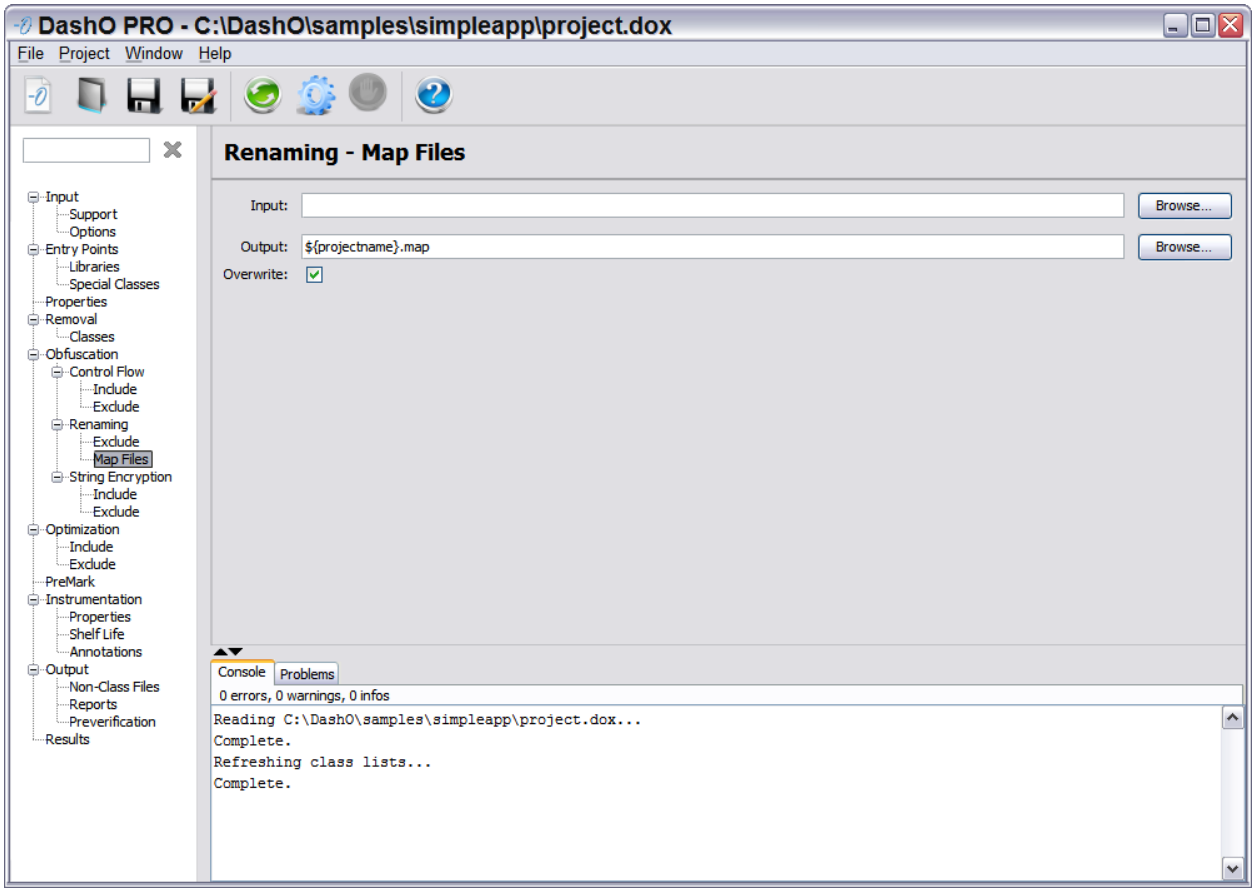



When a class rule is defined it can be used to exclude the class itself from renaming or only the members matched by its method and field rules. To change this setting, right-click on the rule to edit it's properties and change the **Selects Class** setting.

See Graphical Rules Editing Interface for details.

## Renaming – Map Files

The *Renaming Map Files* panel is used to instruct DashO to read or write the renaming information for the project. This information is used to perform incremental renaming or to decode stack traces from an obfuscated application.




### Map Input File

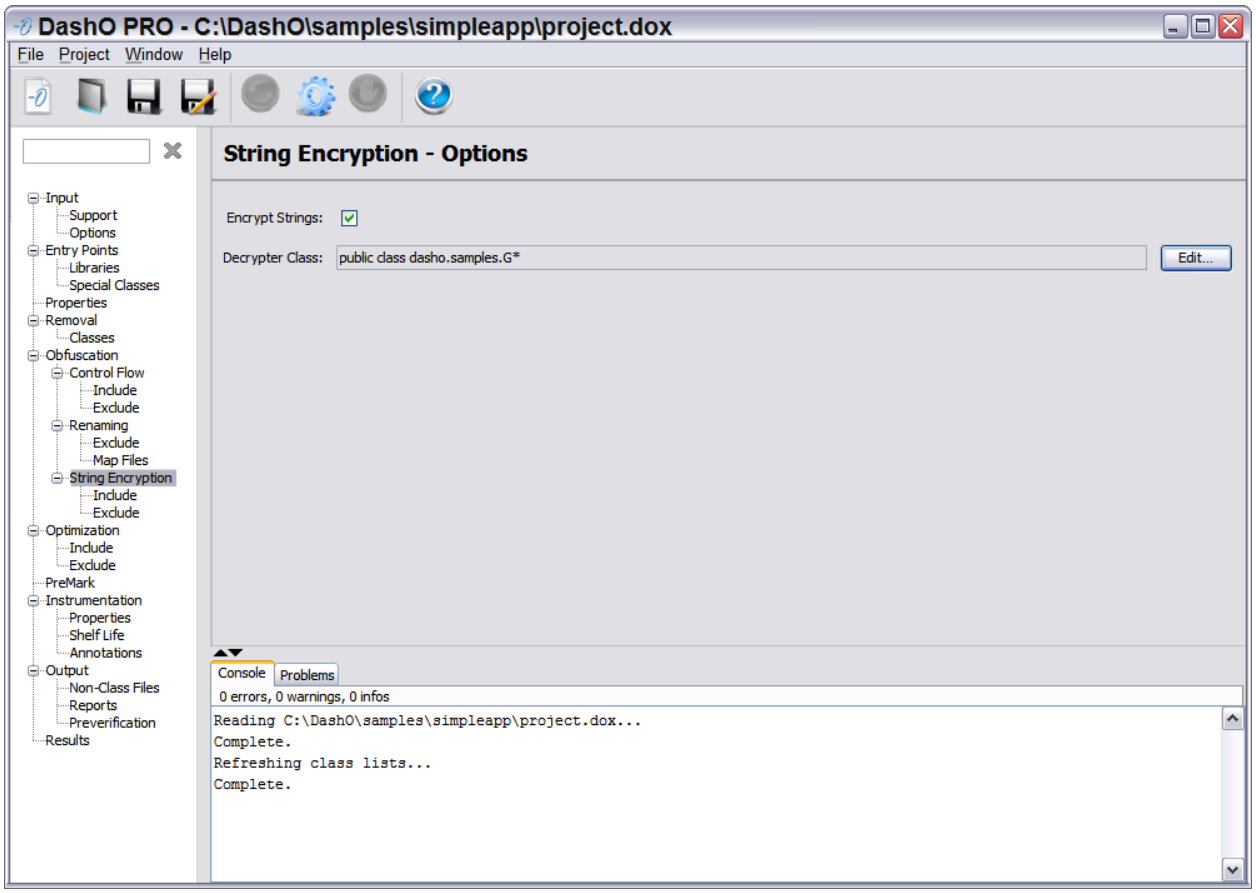
The map input file specified is a file created by a previous DashO run. Using this file, DashO uses the names used in the previous run. The map report file will note the changes detected and the renamer's reaction to those changes.

### Map Output File

The information created in this file can be used for the map input file in a future DashO run. It is also used to decode a stack trace from your obfuscated application. Since accidental loss of this file could destroy your chances of incrementally updating your application in the future, DashO does not automatically overwrite this file. Selecting the **Overwrite** option allows DashO to overwrite an existing file.

## String Encryption – Options

The *String Encryption – Options* panel controls the encryption of strings and allows you to control the location where the decryption method is placed.




### *Encrypt Strings*

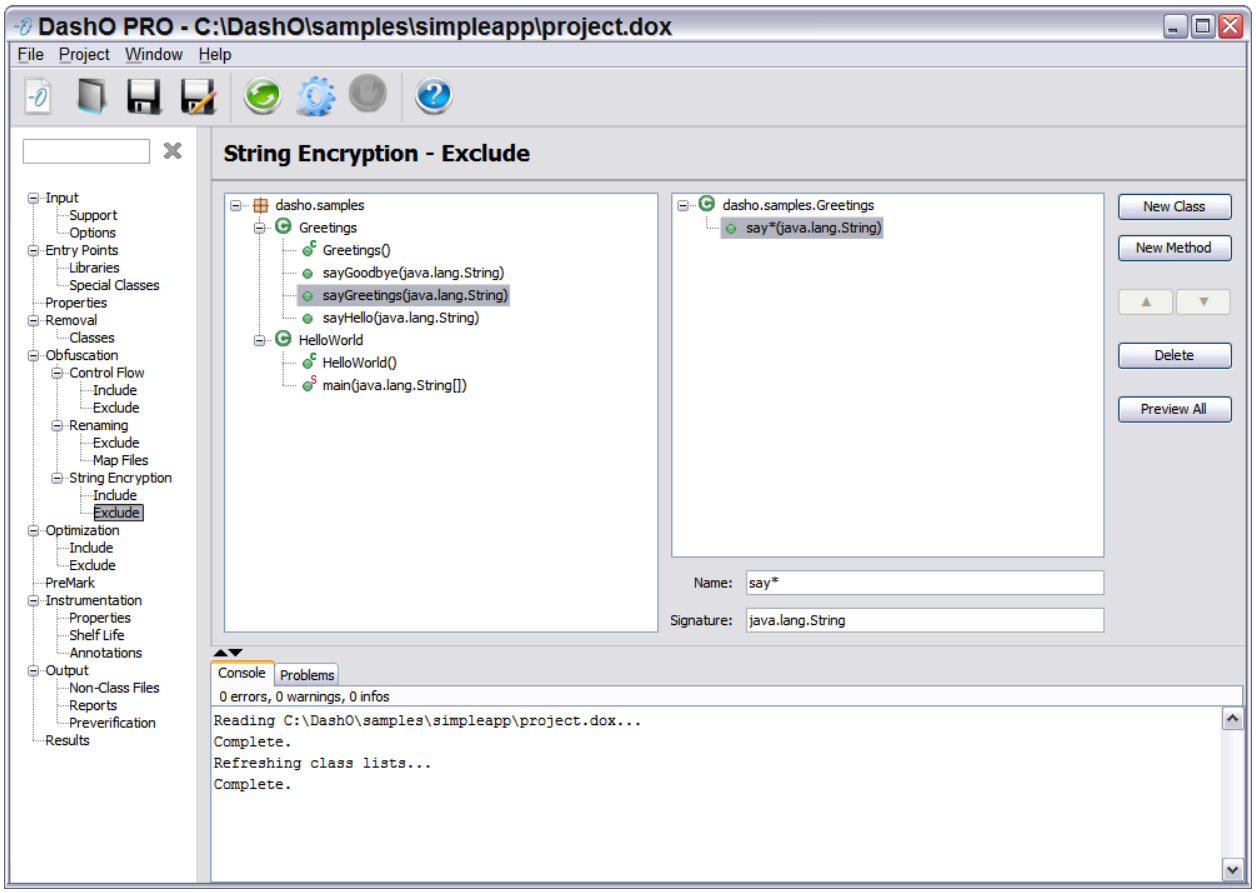
Enables or disables string encryption obfuscation globally. You can control the portions of the application to which string encryption is applied by using include and exclude rules. If you do not specify any rules then all methods will have their strings encrypted.

### *Decrypter Class*

This setting lets you control the exact class where the decrypter or a set of criteria that limits where it can be placed. If you do not specify any value DashO will choose a class from the public classes in the input. To change the selection criteria click the **Edit** button to bring up a properties dialog.

## String Encryption – Includes and Excludes

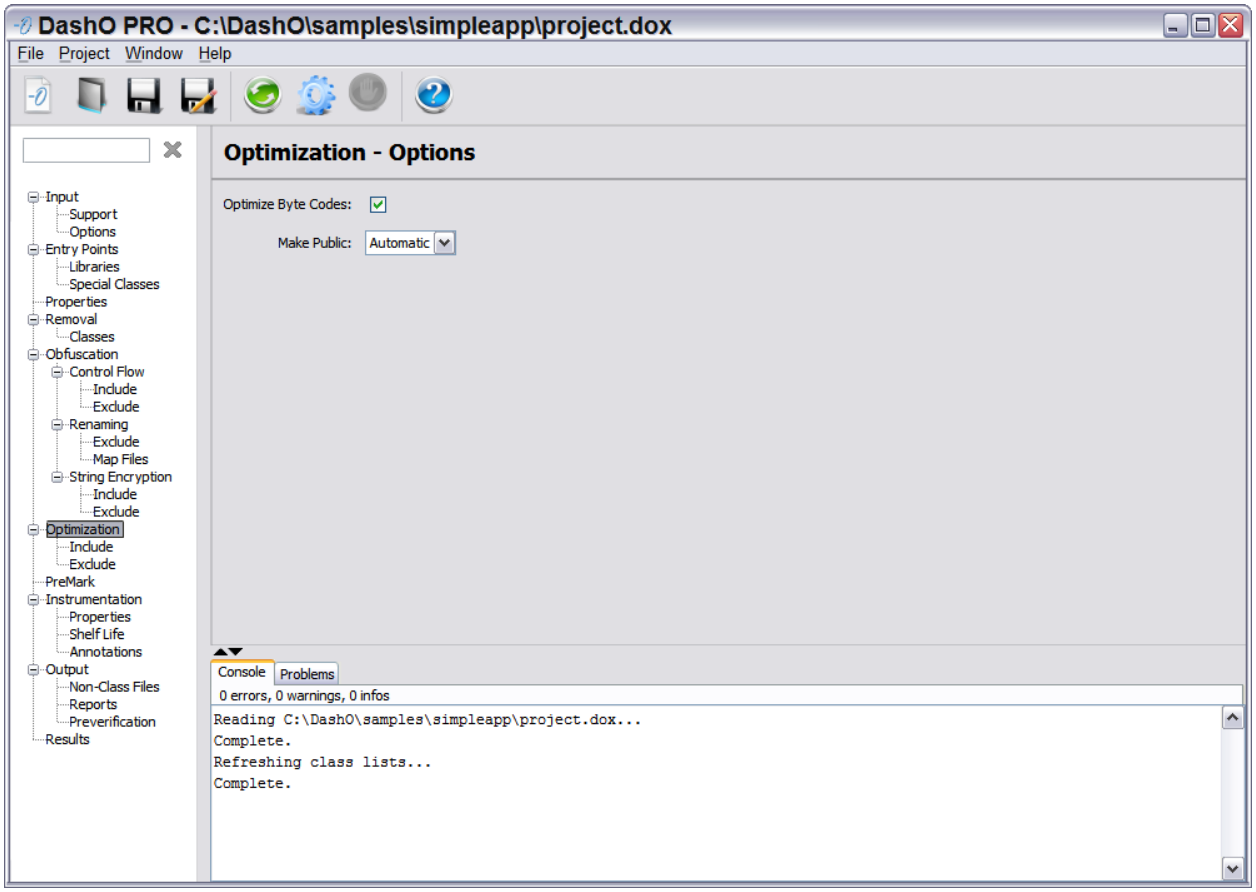
The *String Encryption Include and Exclude* panels let you compose rules that determine which parts of the application will have strings encrypted. Methods, classes, or entire packages can be selected. Since string encryption adds a size and runtime performance cost, you can selectively include parts of you application where sensitive string information is located or exclude sections where performance may be impacted by the runtime decryption.




See Graphical Rules Editing Interface for details.

# Optimization – Options

The *Optimization - Options* panel controls the optimization settings for your project.




## *Optimize Byte Codes*

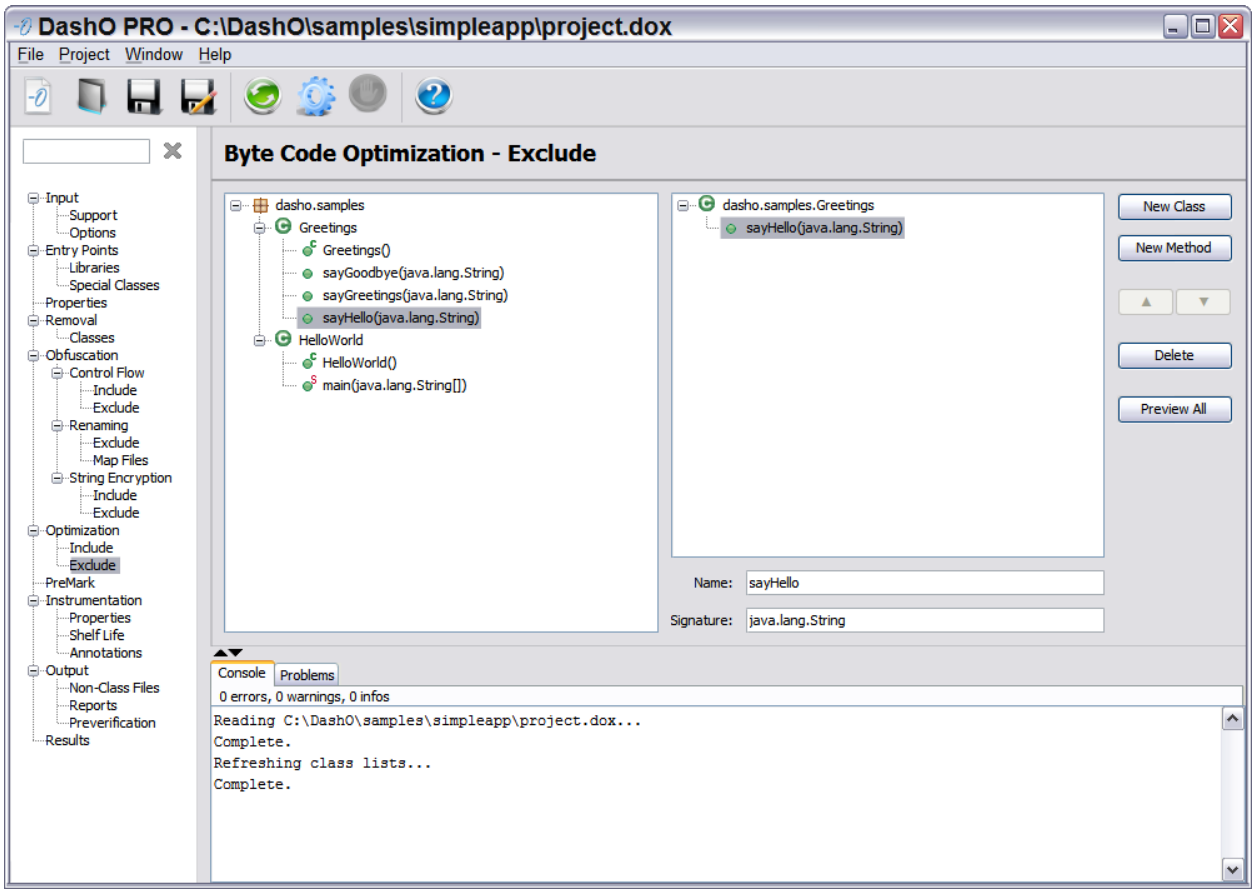
Enables or disables byte code optimization globally. You can control the portions of the application to which byte code optimization is applied by using include and exclude rules.

## *Make Public*

This controls the modification of access control to public. Options are to force or prohibit the conversion to public access or to let DashO decide. The default value is to let DashO decide. See the section on makepublic and nomakepublic global options for details.

## Byte Code Optimization – Include and Exclude

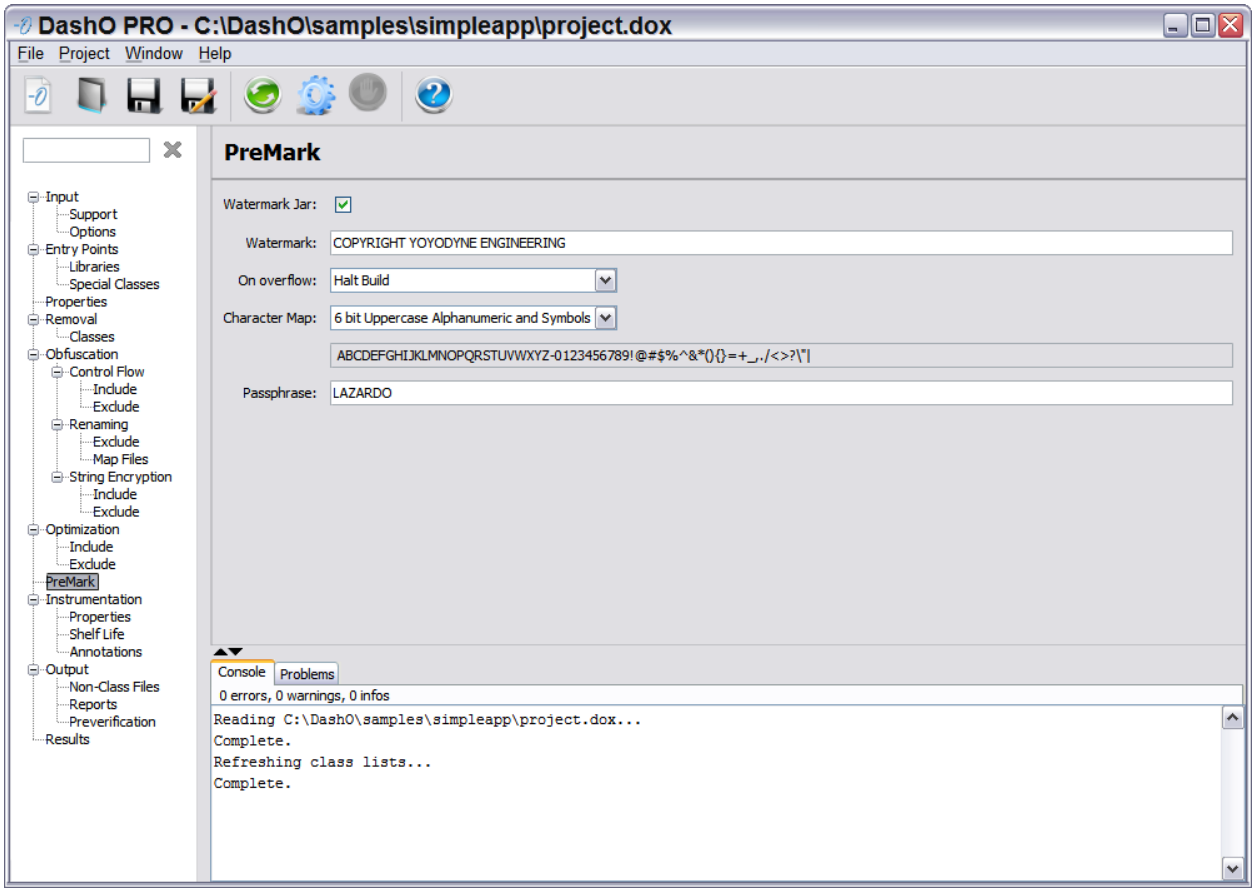
The *Byte Code Optimization Include and Exclude* panels let you compose rules that determine which parts of the application will be optimized. Methods, classes, or entire packages can be selected.




See Graphical Rules Editing Interface for details.

# PreMark

The *PreMark* panel is used to add a watermark to jars produced by DashO. Watermarks can only be applied to jars and this feature will be disabled when DashO's output is to a directory. If multiple jars are created the same watermark is added to all jars.




## *Watermark Jar*

Enables and disables the watermarking feature.

## *Watermark*

This is the watermark string that will be applied to the jar. The characters that can be used in the watermark are determined by the character map setting.

## *On overflow*

If the watermark string is too long to be applied the jar, DashO can either truncate the string and proceed, or halt the build.

## *Character map*

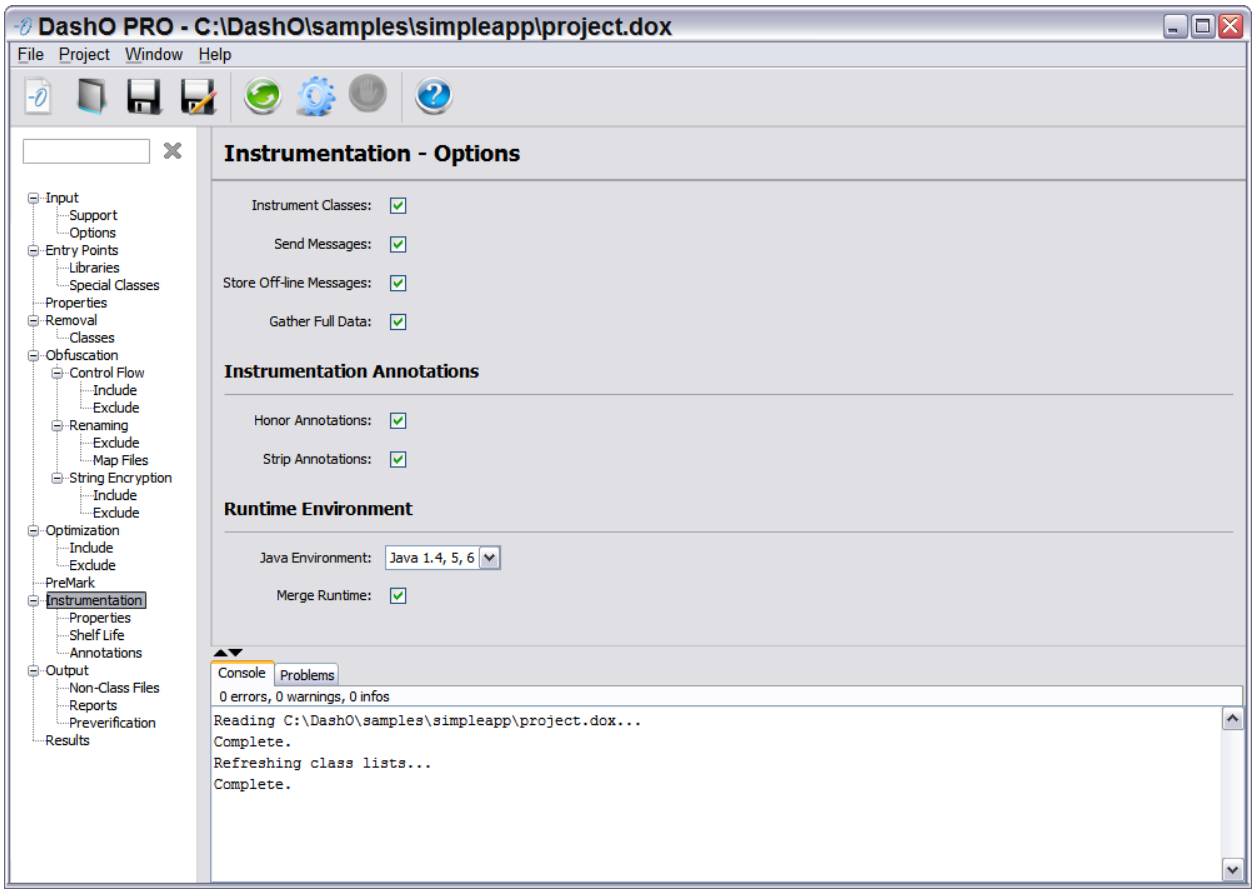
The character map is used to encode the watermark string into a minimal set of bits.

## *Passphrase*

The optional passphrase is used to encrypt the watermark before it is applied to the jar.

# Instrumentation – Options

The *Instrumentation – Options* panel is where you control the setting for instrumenting your application for Runtime Intelligence Services. You enable the instrumenting of classes for Runtime Intelligence Services, define annotation processing, and select the runtime Java environment for the application.




## *Instrument Classes*

Enables or disable the feature. Instrumentation is used to add Runtime Intelligence message and Shelf Life expiration to the application.

## *Send Messages*

Should messages be sent to the Runtime Intelligence server when the application is *on-line* or should messages be stored for later transmission.

## *Store Off-line Messages*

Should messages that cannot be sent to the Runtime Intelligence server, either because the application is off-line or Send Messages is disabled, be saved for later transmission.

## *Gather Full Data*

Some Runtime Intelligence actions, such as the performance probe and the system profile, can return either full or partial data. If your do not require detailed information such as the machines manufacture and model, you can opt to return partial data. This can reduce the time required to generate the message as well as the transmission and/or storage requirements.

### *Honor Annotations*

Should PreEmptive instrumentation annotations references in the code be honored or ignored. Annotations in the code are merged with virtual annotations to determine the instrumentation that will take place.

### *Strip Annotations*

Should PreEmptive instrumentation annotations references be removed from the input classes. If the annotations are not stripped you may have to ship the annotations jar with your application.

### *Java Environment*

This selects the runtime environment of your application, and determines which Runtime Intelligence implementation jar will be used with your application.

### *Merge Runtime*

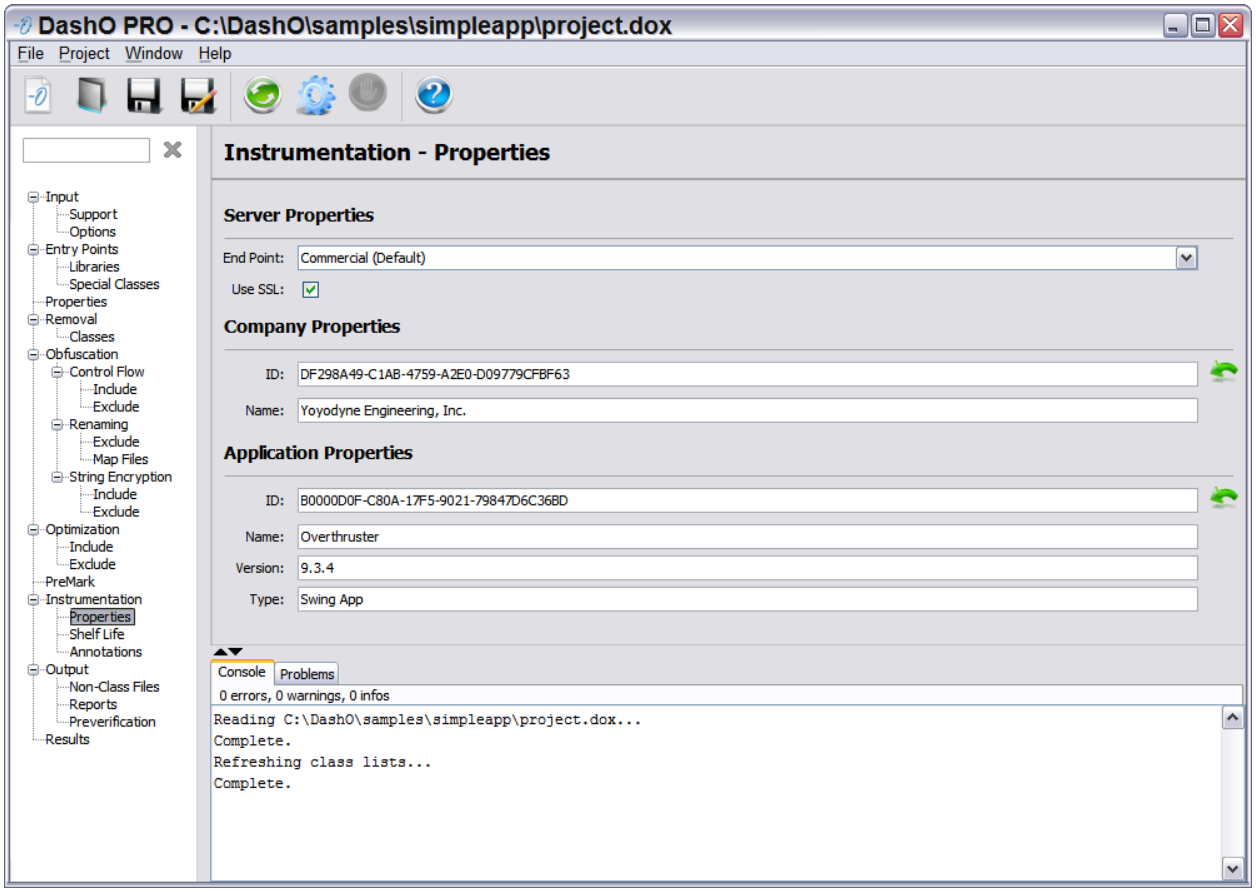
Should the jar that implements the Runtime Intelligence classes be merged with the application or left as a separate jar. When merged with the application DashO will first try to merge it with one of the input jars. If no jars are available, or the classes in the jar have been excluded or pruned to the point where the jar is empty, it will select the first directory. If you do not merge the runtime jar then you will need to ship it separately with would application.

**Note**

If **Send Messages** and **Store Off-line Message** are off then message generation is disabled. These global values can be overridden by real or virtual annotations. Values for both options can be either fixed boolean values or from dynamic sources.

# Instrumentation – Properties

The *Instrumentation – Properties* panel is where the unique identifiers for the application and the application owner are entered. Entering this information enables Runtime Intelligence Services to identify the application and the rightful owner so that the correct information is available to you in the Runtime Intelligence Portal.




## *End Point*

This is the location of the Runtime Intelligence server. You can either choose from the list of endpoints provided by PreEmptive or enter a custom endpoint if you have a self-hosted server. The end point is like a URL but does not include the protocol.

## *Use SSL*

Should HTTP or HTTPS protocol be used when sending data to the endpoint.

## *Company ID*

This is the unique ID assigned to your company by PreEmptive Solutions when you request the Runtime Intelligence activation code. Clicking the green arrow next to ID will automatically populate the company fields with the Runtime Intelligence information you entered in User Preferences.

## *Company Name*

This is the name of your company.

### *Application ID*

Click the green arrow to auto populate this field with a unique ID for your application instrumented for Runtime Intelligence. Clicking the green arrow next to ID will automatically populate the ID field with a random identifier that you can use for your application.

### *Application Name*

This is the name of your application that will be sent to the Runtime Intelligence server and will be used for identification purposes.

### *Version*

This is the version number of your application that will be sent to the Runtime Intelligence server and will be used for identification purposes.

### *Type*

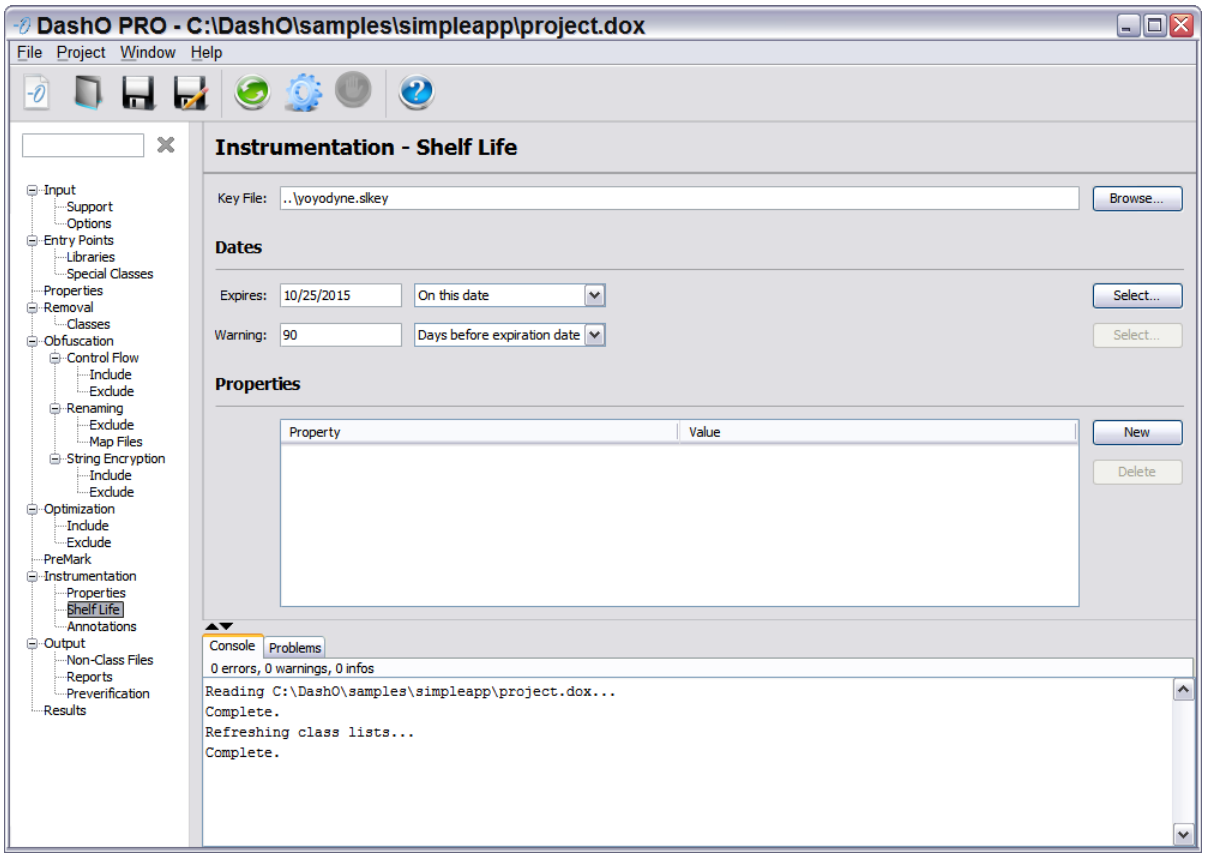
This identifies the application type that will be sent to the Runtime Intelligence server and will be used for identification purposes.

**Note**

All the data on this panel is optional. It can be combined or overridden by annotations in the code or by virtual annotations in DashO. The values on this panel will be used only if they are not superseded by annotations.

# Instrumentation – Shelf Life

The *Instrumentation – Shelf Life* panel is where you configure the addition of an expiration check to your application. DashO uses this information to create an expiration token which is placed inside your code to enforce the expiration policy. The check is performed where an `ExpiryCheck` annotation appears in the code or with a virtual annotation. Expiration tokens can also be read in from external files or resources using the `ExpiryTokenSource` annotation in which case you can leave the entries on this panel blank.




## *Key File*

Enter the location of the Shelf Life key file you received from PreEmptive Solutions. This file authorizes you to add expiration checks to your application.

## *Expiration Date*

Your application can be configured to expire on an explicit date or a certain number of days after a dynamically determined start date. When an explicit date type is selected you can use the Select button to pop-up a calendar to select the date. Dates are always in `MM/DD/YYYY` format regardless of the local convention.

## *Warning Date*

Your application can be configured to issue expiration warnings starting on either an explicit date or a certain number of days before it is due to expire. When an explicit date type is selected you can use the Select button to pop-up a calendar to select the date. Dates are always in `MM/DD/YYYY` format regardless of the local convention.

### *Properties*

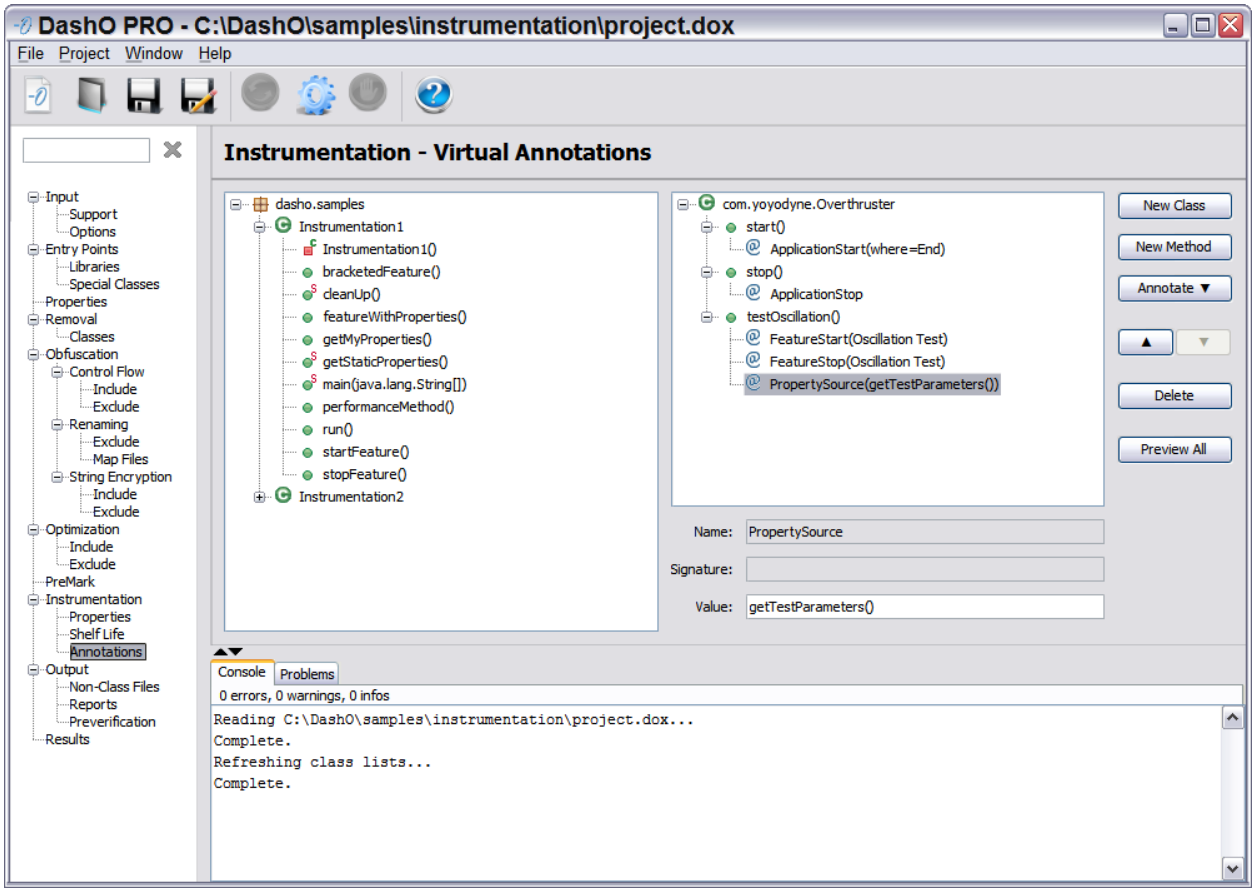
You can add arbitrary properties to the expiration token that can be retrieved by your application. To use this feature you need to supply a user defined action to the `ExpiryCheck` – this action method is passed the expiration token where you will be able to retrieve these properties. Note that both the property name and values can contain DashO property references.

**Note**

The information supplied on this panel can be overridden or supplemented by annotations in your code or by DashO's virtual annotations.

# Instrumentation – Annotations

DashO uses annotations to perform instrumentation. Annotations are the instructions for identifying what is to be instrumented, such as classes or methods, and how to instrument them. These annotations augment or override annotations present in the class files. The Virtual Annotations screen behaves similarly to the Graphical Rules Editing Interface.




## *Name*

This is the name of the class, method, or annotation that is clicked on or highlighted.

## *Signature*

This is a list of types that match the types in the method's parameter list.

## *Value*

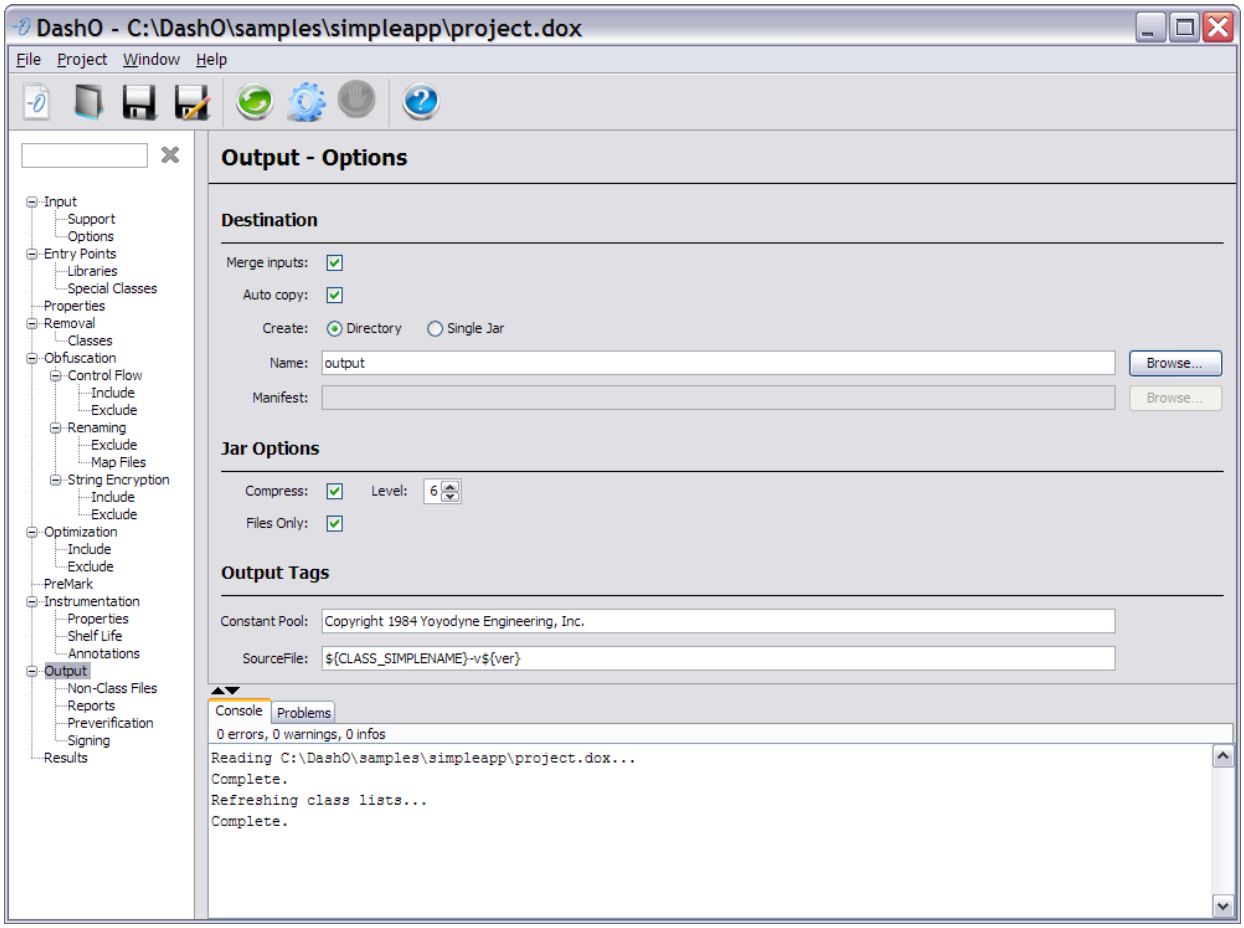
This is an annotation specific value. Annotations such as `FeatureStart` and `FeatureStop` use this as the name of the feature. Not all annotations use a *value*.

## *Annotate Button*

This allows you to add annotations to be applied to the method or class. You can also add the annotations by right-clicking rules or items in the class list.

# Output – Options

The *Output - Options* panel controls where DashO will place the results of the build and what form those results will take.




## Merge inputs

DashO can combine the obfuscated results into a single directory or jar or keep the original packaging of the input classes.

## Auto copy

When you keep the original packaging of the input classes non-class files in input jars input are automatically copied to their respective output. Non-class files that appear in input directories are *never* copied. When merging inputs you can turn this feature on or off.

## Create Directory/Single Jar

When you are merging DashO can place the results of the build into either a directory or into a single jar. Turning off merging requires that the output be a directory. See the <output> Section for instruction regarding writing the output to a directory or a jar file.

## Name and Manifest

The **Name** field specifies the name of the output directory or jar. When merging is off only a directory can be used. DashO will use this as the root of the output and will attempt to recreate the hierarchy of the original input jars and directories.

If you have DashO create a single merged jar for you DashO can add a manifest file to the jar. The manifest can either be in the form of a text file or DashO can extract the manifest from a jar file.

## Jar Options

- **Compress**: Not only store data but also compress it.
- **Files Only**: Store only file entries, not directory entries.
- **Level**: Level at which file compression should be performed. Valid values range from 0 (no compression/fastest) to 9. The default value is 6.

## Constant Pool Tag

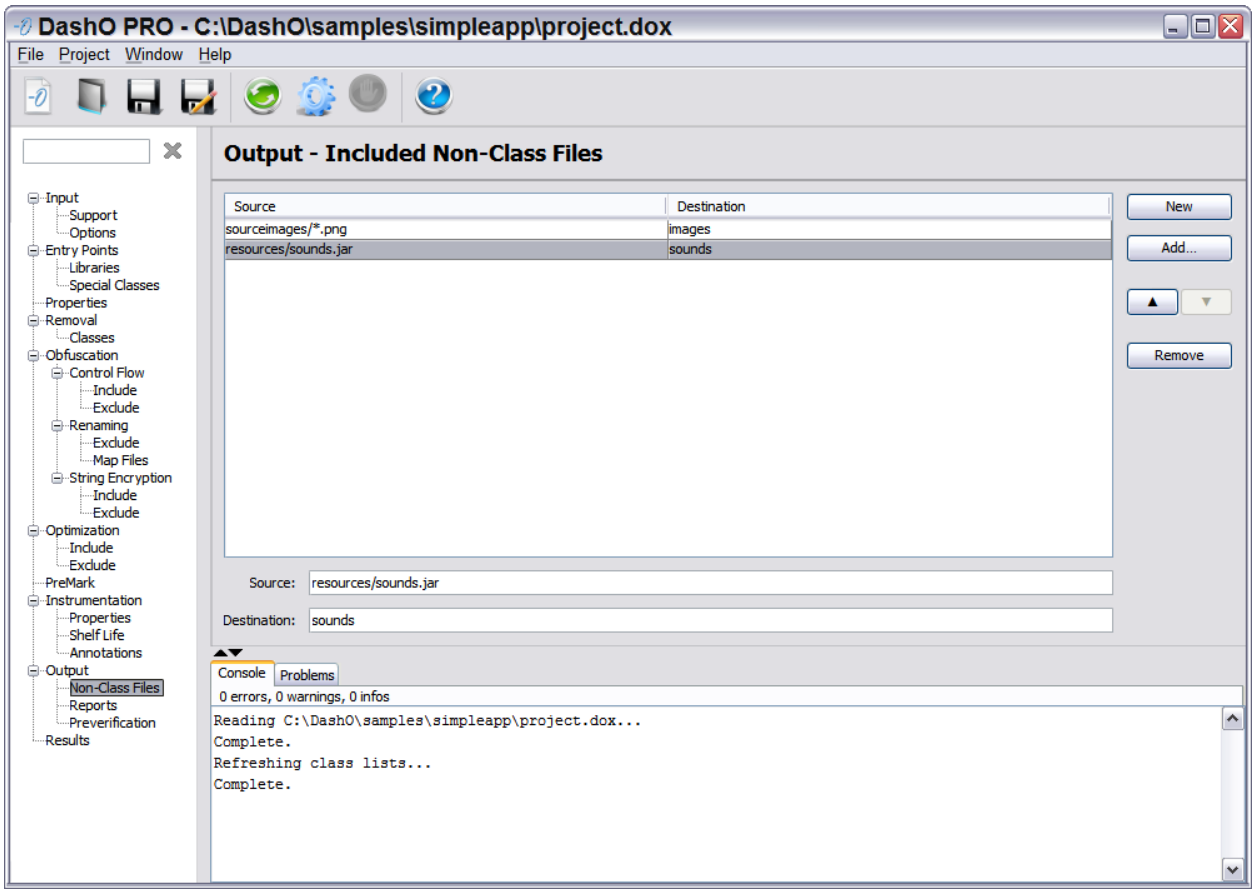
The optional constant pool tag text is inserted into every class in the resulting output. See <constpooltag> for details.

## SourceFile Tag

The SourceFile attribute of every resulting output class is set to the given value. See <sourcefile> for details.

## Output – Included Non-Class Files

The *Output – Included Non-Class Files* panel lets you specify the source for non-class contents, such as images, property, or configuration files that need to be in the application.




Directories, individual files, or jar files may be added to list by selecting an existing file using **Add** button. You can also create an entry by using the **New** button and editing its name. For directories and jars all non-class files are copied into DashO's output. Directory entries can contain wildcard patterns using the ***** character to select particular file types.

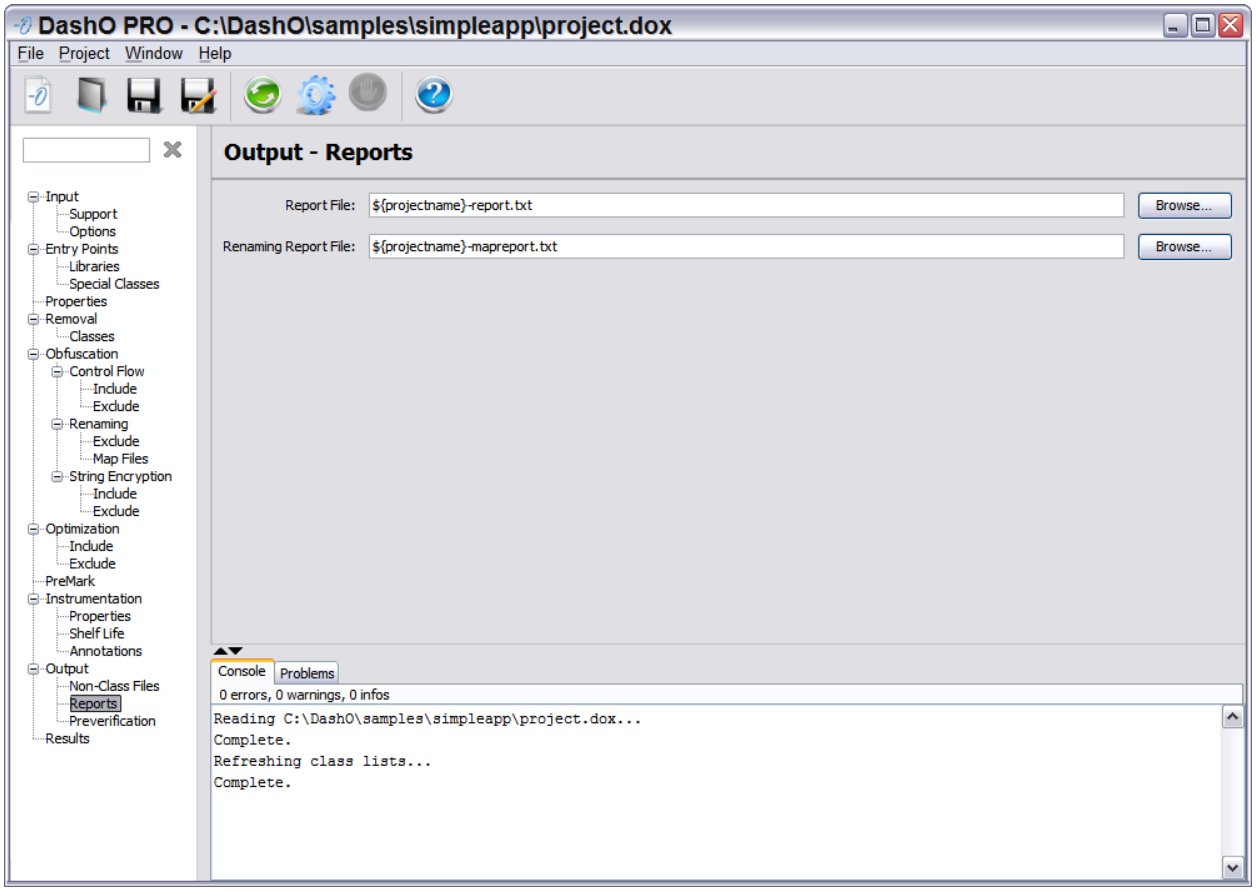
If the non-class files need to be placed into a location in the output that does not correspond to their input locations you can specify a destination that is relative to DashO's output location. See the <includenonclassfiles> section for details.

**Note**

If you are merging inputs and your input jars files contain the non-class files you can either turn on **Auto Copy** or reference them here. If you are *not* merging inputs DashO will copy over all non-class files in your input jars automatically. Non-class files that appear in input directories are *never* copied.

## Output – Reports

The *Output - Reports* panel configures the generation of reports that detail the results of the build.




### *Report file*

Specifies the name and location for a report outlining the class and member removal and renaming performed by DashO. A summary is given detailing the total methods/fields/constant pool entries, as well as the final number and percentage of reduction after DashO execution. It also contains information about dynamically loaded classes, including reflection and `Class.forName()` calls.

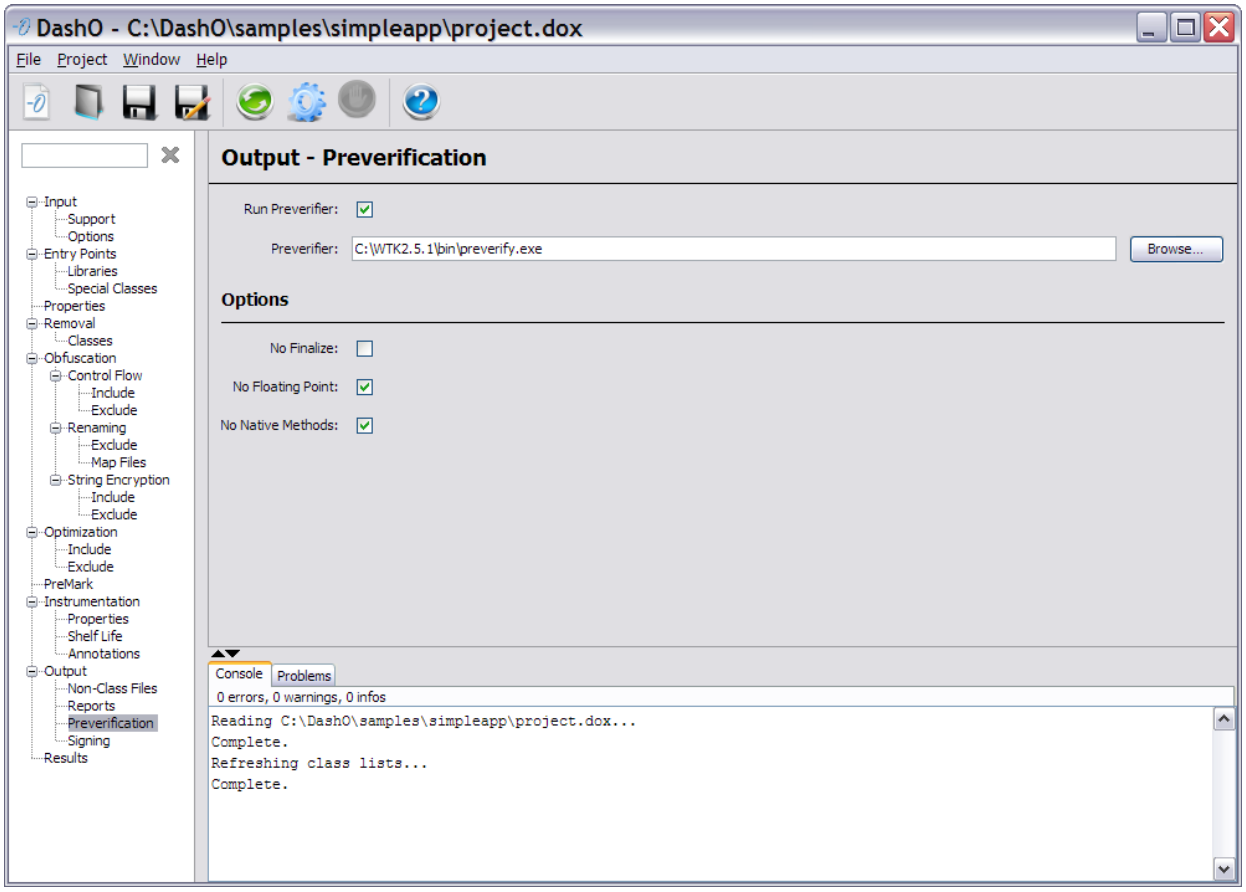
### *Renaming Report File*

This specifies the name and location for a report listing old and new names for renamed classes as well as their renamed members.

## Output - Preverification

You can choose to run the preverifier on your CLDC application after DashO is finished processing your class files. You can enable or disable preverification by checking the **Run Preverifier** checkbox.

By default DashO will try to find the preverifier application `preverify` on the system path. If you need to run a particular version of the preverifier you can explicitly specify which one to run.




### *No Finalize*

Pass `-nofinalize` to the preverifier: no finalizers are allowed in the input.

### *No Floating Point*

Pass `-nofp` to the preverifier: no floating point operations allowed in the input.

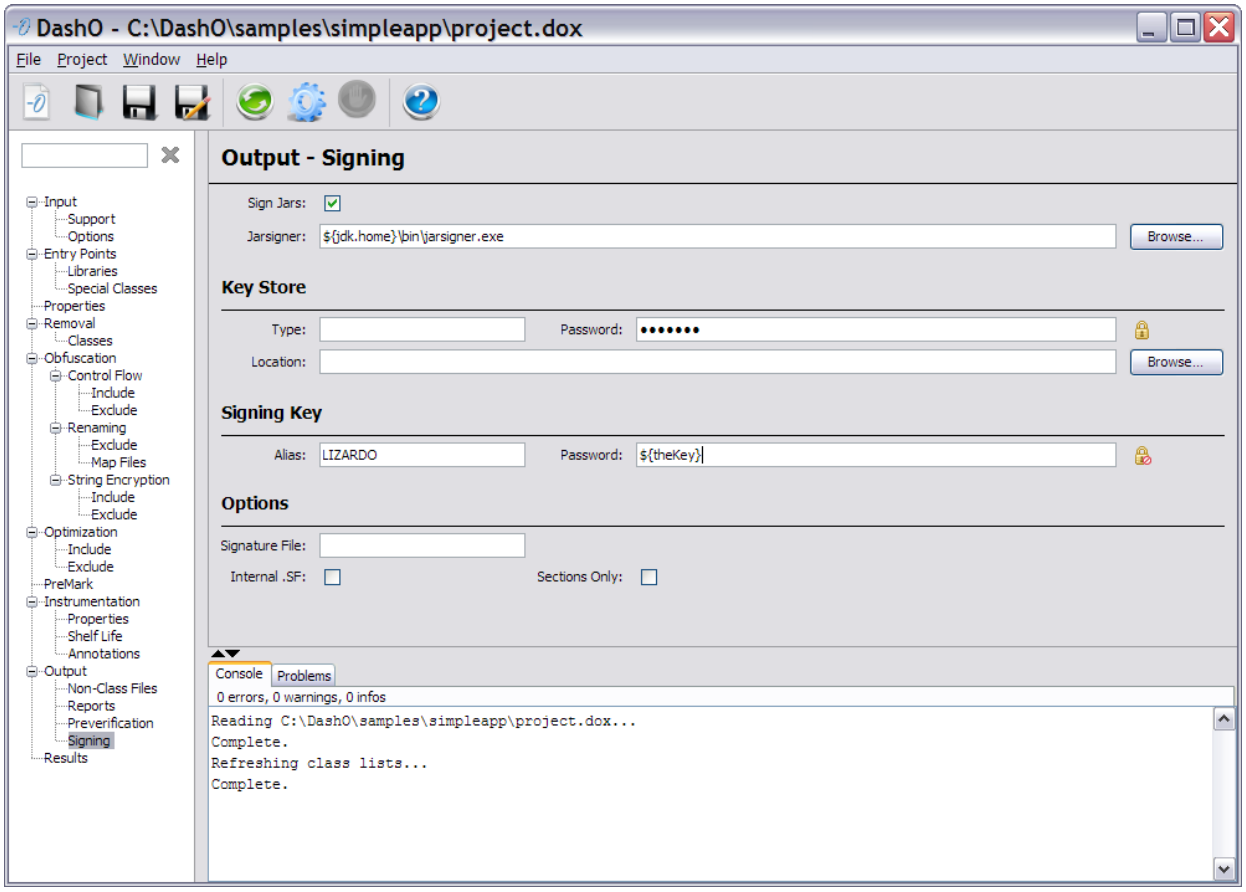
### *No Native Methods*

Pass `-nonative` to the preverifier: no native methods allowed.

## Output – Signing

You can have DashO sign the output jars that it produces. You can enable or disable signing by checking the **Sign Jars** checkbox.

By default DashO will try to find the signing application `jarsigner` on the system path. If you need to run a particular version of the jar signer you can explicitly specify which one to run.




### *Key Store*

This information defines the key store that contains the private key used for signing. Only the **Password** is required. The **Type** defaults to type specified in the global `keystore.type` security property and the Location defaults to the `.keystore` file in your home directory. Passwords that do not contain property references are stored in an encrypted form in the project file.

### *Signing Key*

This information specifies the private key that is used to perform the signing. Only the **Alias** value is required. The **Password** defaults to the password specified for the key store. This password is also store encrypted in the project file if it does not contain any property references.

### *Options*

These values corresponds to the `-sigFile`, `-internalsf`, and `-sectionsonly` options of `jarsigner`. Please see jarsigner - JAR Signing and Verification Tool for details on their use.

## Building

There are two ways to build in the user interface. You can click the **Build Project** button on the Toolbar or select **File > Build** in the Menu.

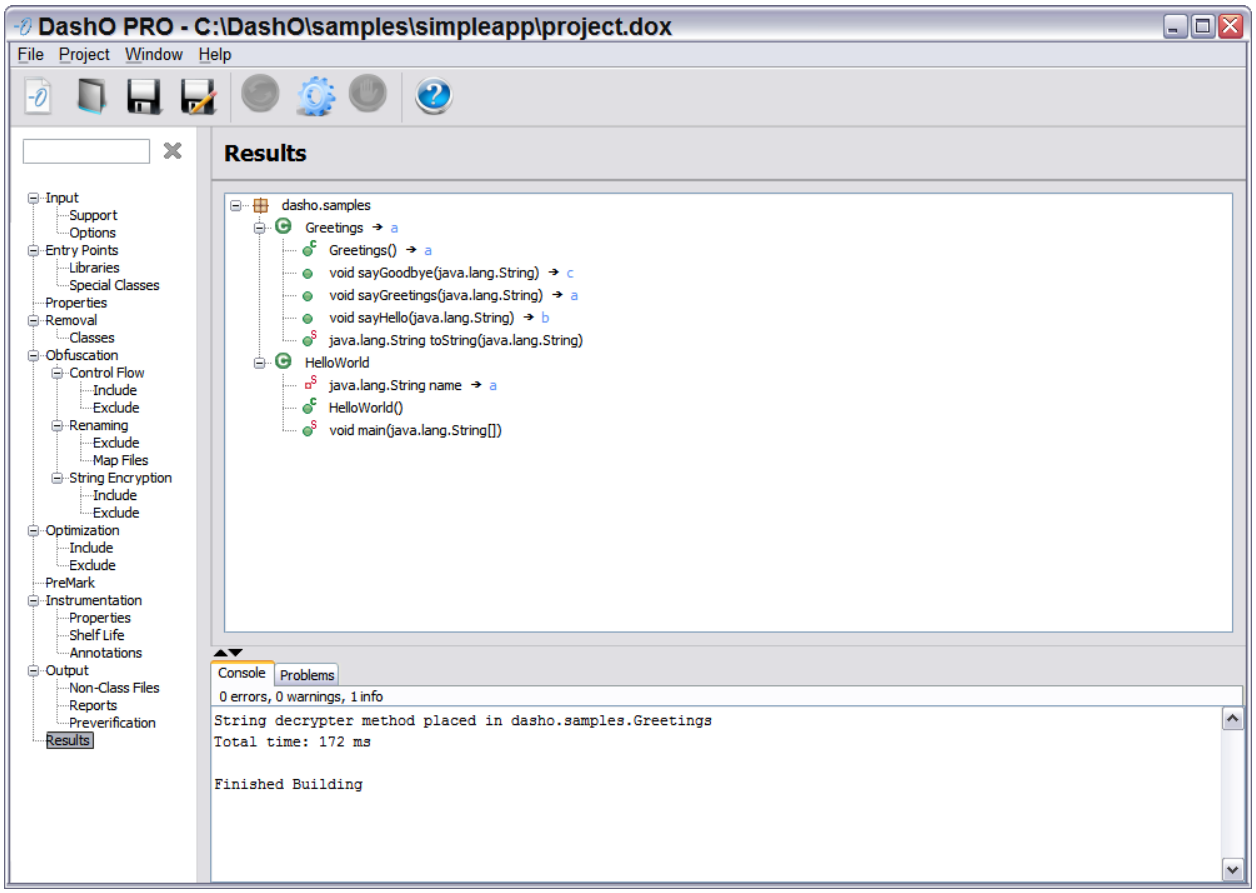
During and after the build, you may view DashO's output in the console area.

The build performs method/field removal, renaming, control flow, string encryption, optimization, and writing and packaging of the final classes. This may take up to several minutes depending on the number of classes DashO need to process.

When the build has completed DashO will show the results panel in the work area screen automatically. If the build encountered an error the console display will switch to the Problems tab.

## Output – DashO Results

The *Output - DashO Results* panel shows the class hierarchy of the input classes of the project and the results of renaming.



# Using the Graphical Rules Editor

Many of the panels in DashO's user interface are rule editors, primarily for including and excluding elements in your application in an obfuscation transformation. A rule editor is divided into two lists – a class list on the left hand side that shows the classes and members of the input and a rule list on the right. The rules specify what parts of the input are affected by the operation and in some cases the actions to be taken. The rules editor is used to set rules for the following operations:

- Renaming exclude rules
- Control Flow Obfuscation include and exclude rules
- String Encryption include and exclude rules
- Optimization include and rules

Other parts of DashO, such as entry points, use an interface very similar to the rule editor.

## Creating Rules

There are several ways to create rules in the interface:

- **Right-click items in the class list** – you can click on items in the class list to bring up a contextual menu. From there you can build a rule that will match the item that you have selected. If you hold down the shift key when you create the rule the rule will be made a regular expression. If you create a rule for a method or a field, DashO will add the new rule to a pre-existing class rule or create one if needed.
- **Drag and drop items from the class list** – you can drag an item from the class list and drop it on the rules. If you drag and drop either a method or a field, DashO will add the new rule to a pre-existing class rule or create one if needed.
- **Using the buttons** – You can click the *new* buttons to the right of the rules list to create a new entry. A new rule will be created with a dummy name that you can edit.

## Editing Rules

The basic parts of a rule can be modified directly in the editor. The name of any item and the signature or methods can be changed by using the text field immediately below the rules list. Specialized editors may also provide for direct editing of their values.

To access all the settings for a rule right-click on the rule and select the Properties item on the contextual menu. In the properties box you will find the settings for values such as:

- **Modifiers** – the Java modifiers, or their negation, that are required for this rule to match an item. See the description of the Modifiers attribute for values you can use here.
- **Name** – the name of the item that the rule affects. This can be a constant value, a pattern, or a regular expression.
- **Signature** – the signature for methods.
- **Type** – determine how the name and/or signature are to be interpreted. See Patterns and Regular Expressions for details.
- **Select class** – For rules that affect the class itself as well as its members, this setting determines if the rule applies to the class, or if the class is just a container for nested field or method rules.
- **Renaming controls** – Entry points are non-renameable by default. Some types of entry points can be made renameable and these controls determine if the class and/or its members can be renamed.

- **Values for annotations** – Virtual annotations can contain many specialized values. Some contain only a generic *value* – use the tool tip display to determine its use. Annotations that perform an action will also have a *where* value. This determines the location in the method where the action will take place.

## Previewing rules

You can use the preview function to determine what will be affected by the rules. You can elect to preview a single rule or all rules. Right-click a rule to bring up the contextual menu and select **Preview Rule** or **Preview All**. The items in the class list that will be affected by the rule will be displayed in bold. You can use the contextual menu in either list to clear the highlighting of rules.

Section on interaction between include and exclude

## Patterns and Regular Expressions

A simple rule selects a particular item, such as a class, using the name of the item literally. A rule can also select items by using patterns or by using regular expressions. See Patterns and Regular Expressions.

You can check your regular expression for correctness by right-clicking on the rule and selecting the Check Expression item. If your expression has an error a red X will be displayed next to it. Move the pointer over the rule and the tool tip will display the location of the error and its description.

**Note**

Regular expressions apply to the rule as a whole. If a class name is specified as a regular expression, all member names will be treated as regular expression. Patterns do not have this restriction.

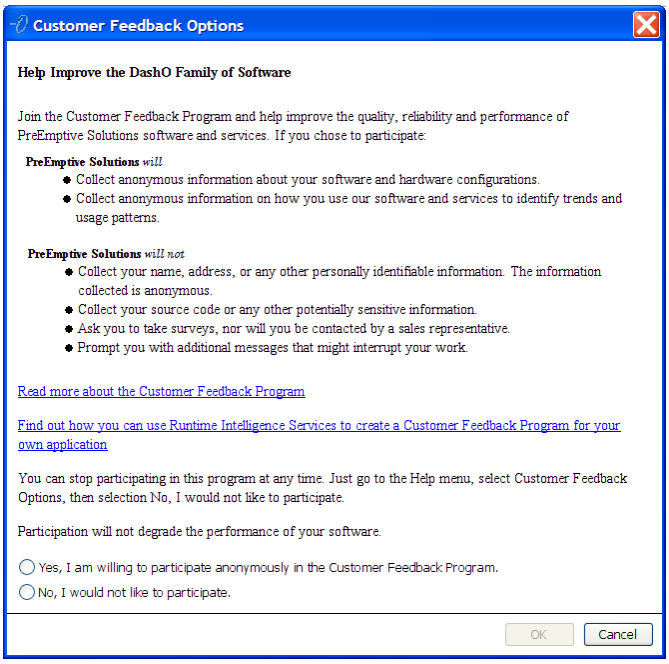
## Combining Include and Exclude Rules

DashO can use a combination of inclusion and exclusion to determine what parts of your application to obfuscate. When an obfuscation transformation allows for the definition of both includes and exclude it is important to remember how the two are combined:

- If no include rules are defined all items are included by default.
- If no exclude rules are defined no items are excluded by default.
- Includes are determined first, then excludes. An item must be included by at least one rule and not excluded by any rule to have a transformation apply to it.

# Customer Feedback Options

DashO provides an anonymous usage reporting system that users can opt-in to. If you opt in to this program, only anonymous high level usage data will be gathered by PreEmptive Solutions with the sole intent of improving DashO. You may change your options at any time from the **Help > Customer Feedback Options** menu.

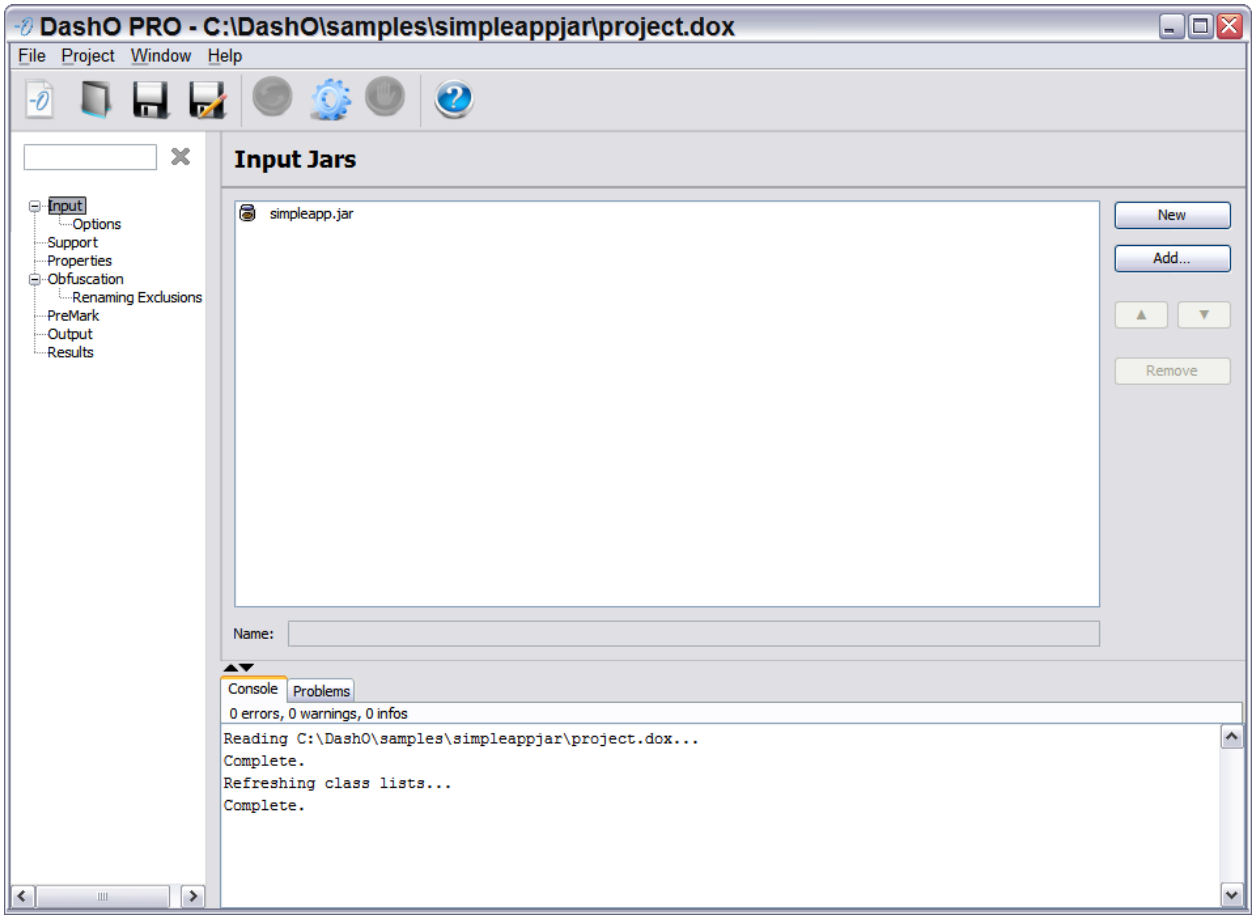


# Quick Jar User Interface

In this section, we explain how to use DashO's interface for quick jar projects. You can use the interface to create new quick jar projects or edit existing ones.

## Input Jars

The *Input Jars* panel is used to specify the jars that are to be processed. DashO examines these jars for manifests that contain Main-Class entries. These will be used as the entry points into the application. If there are no Main-Class entries, then the jars are processed as libraries. Any non-class files in the input jars are copied to the output jar.



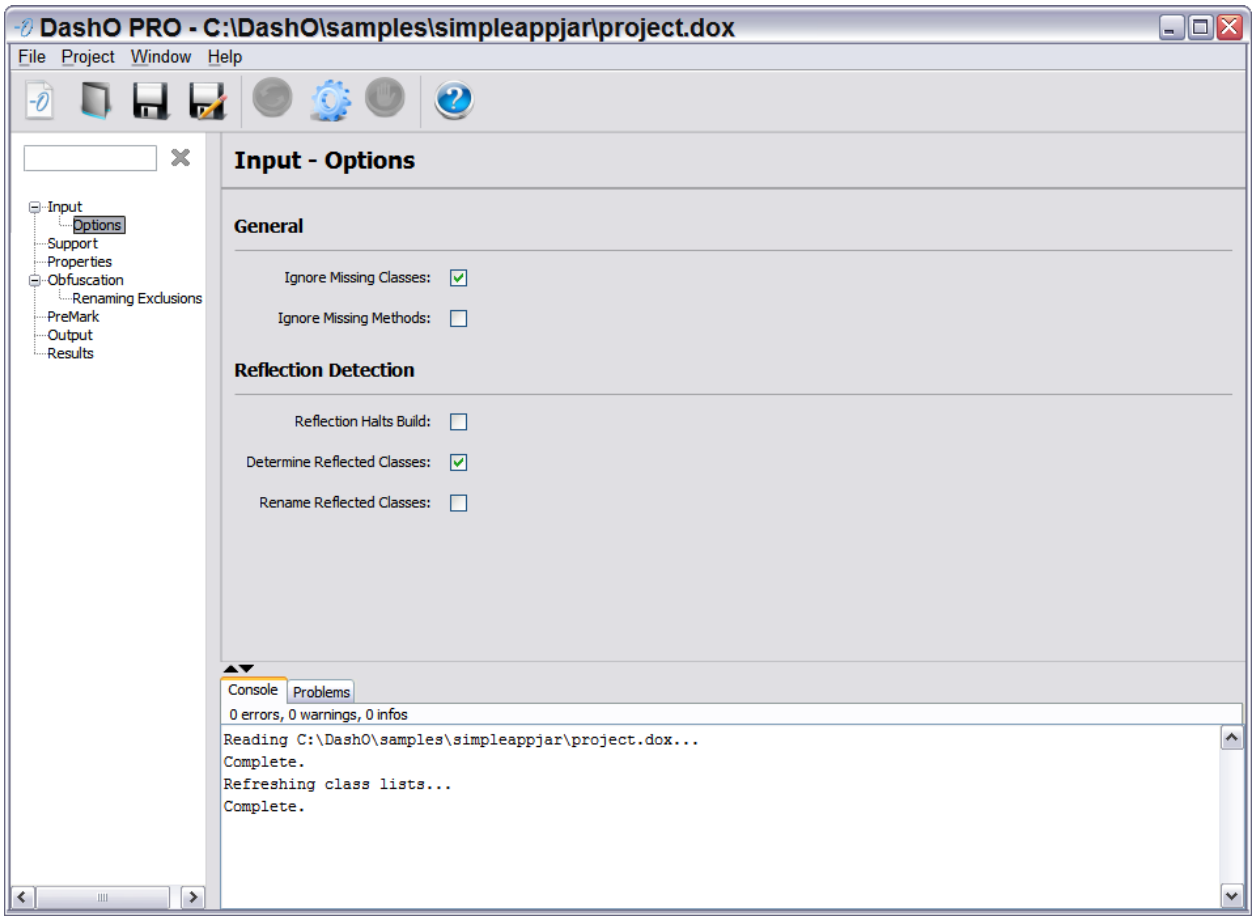

Click the **Add** button to bring up a browse dialog that allows you to navigate your file system and select one or more a jars. You can also add jars by clicking **New** and typing the name of the jar in the *Name:* field.

**Note**

Input Jar names support properties.

# Input - Options

The *Input - Options* panel controls the actions that are applied to the input jars.




## *Ignore Missing Classes*

DashO will attempt to analyze all classes that the application attempts to call. You can instruct DashO to ignore these classes by selecting this option. Note that DashO cannot skip classes and interfaces that the application extends or implements.

## *Ignore Missing Methods*

DashO will attempt to locate concrete implementations of methods as part of its analysis. Turning this method on lets DashO proceeded even if it cannot locate the desired method. Use this option with caution.

## *Reflection Halts Build*

DashO's analysis makes note of reflection usage in the application so that the targets of reflection can be addressed. Turn this option on when you are determining what parts of the application use reflection.

## *Determine Reflected Classes*

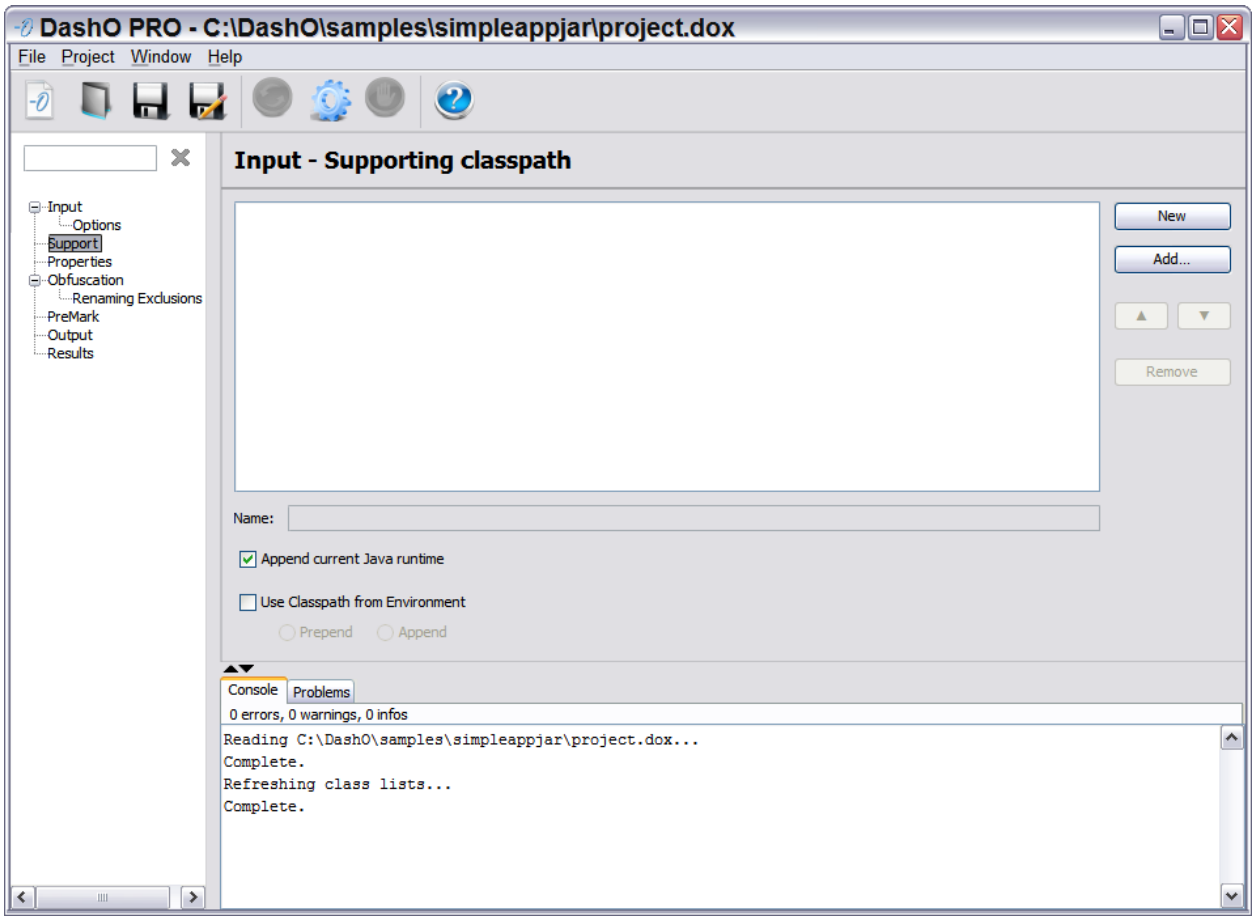
DashO can determine simple targets of reflection and automatically add these classes to the list of included classes. Note that this behavior can increase build time.

### *Rename Reflected Classes*

By default targets of reflection are not renamed. Use this option to allow these classes to be renamed.

## Supporting Classpath

The supporting classpath has the list third party jars and class files used by your application that you do *not* want to obfuscate or include in the final jar. These are important to DashO since your classes can extend from the third party libraries and the renaming system needs to see those classes to determine the methods that are safe to rename.



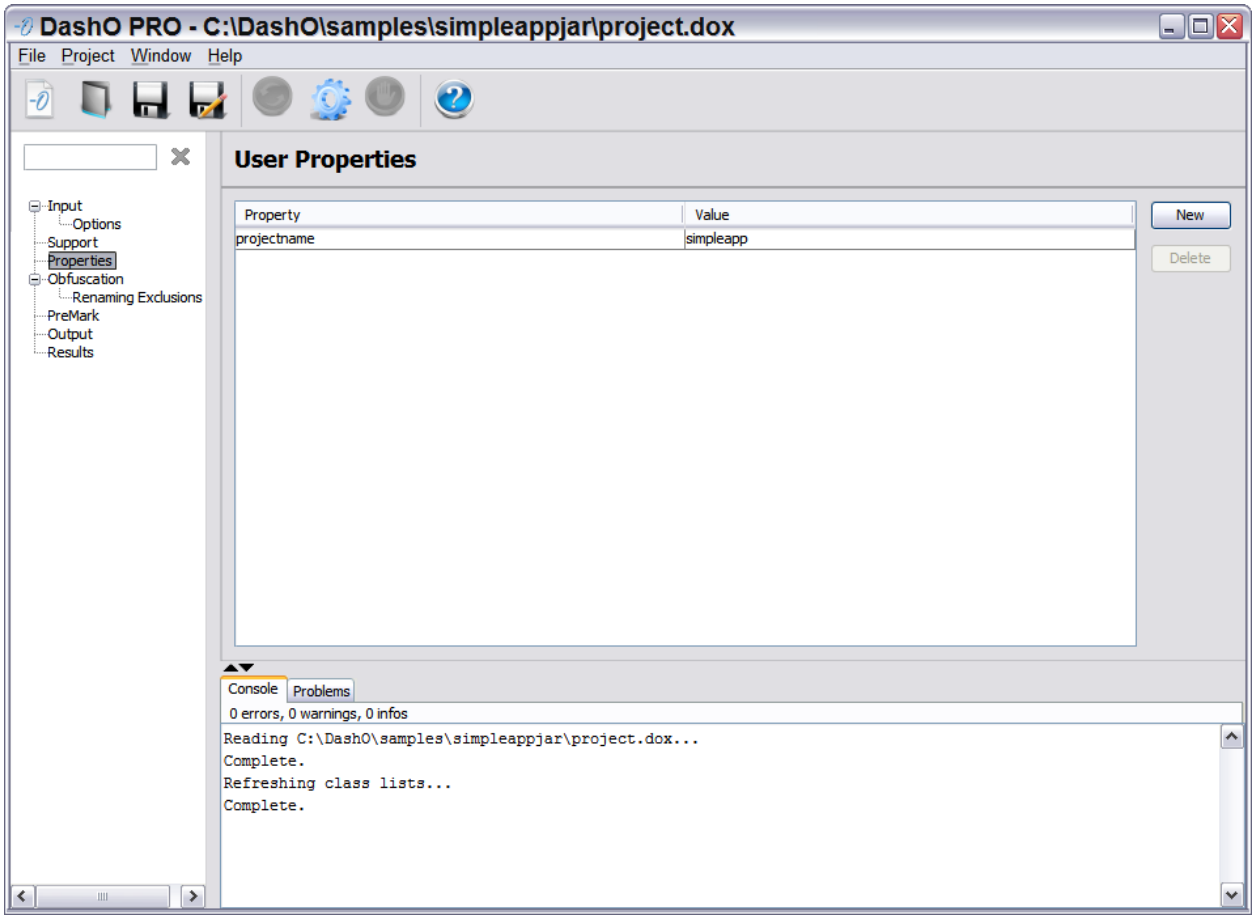

To add a supporting jar, click the **Add** button and select the required jars. You can also add jars by clicking **New** and typing the name of the jar in the **Name***:* field.

**Note**

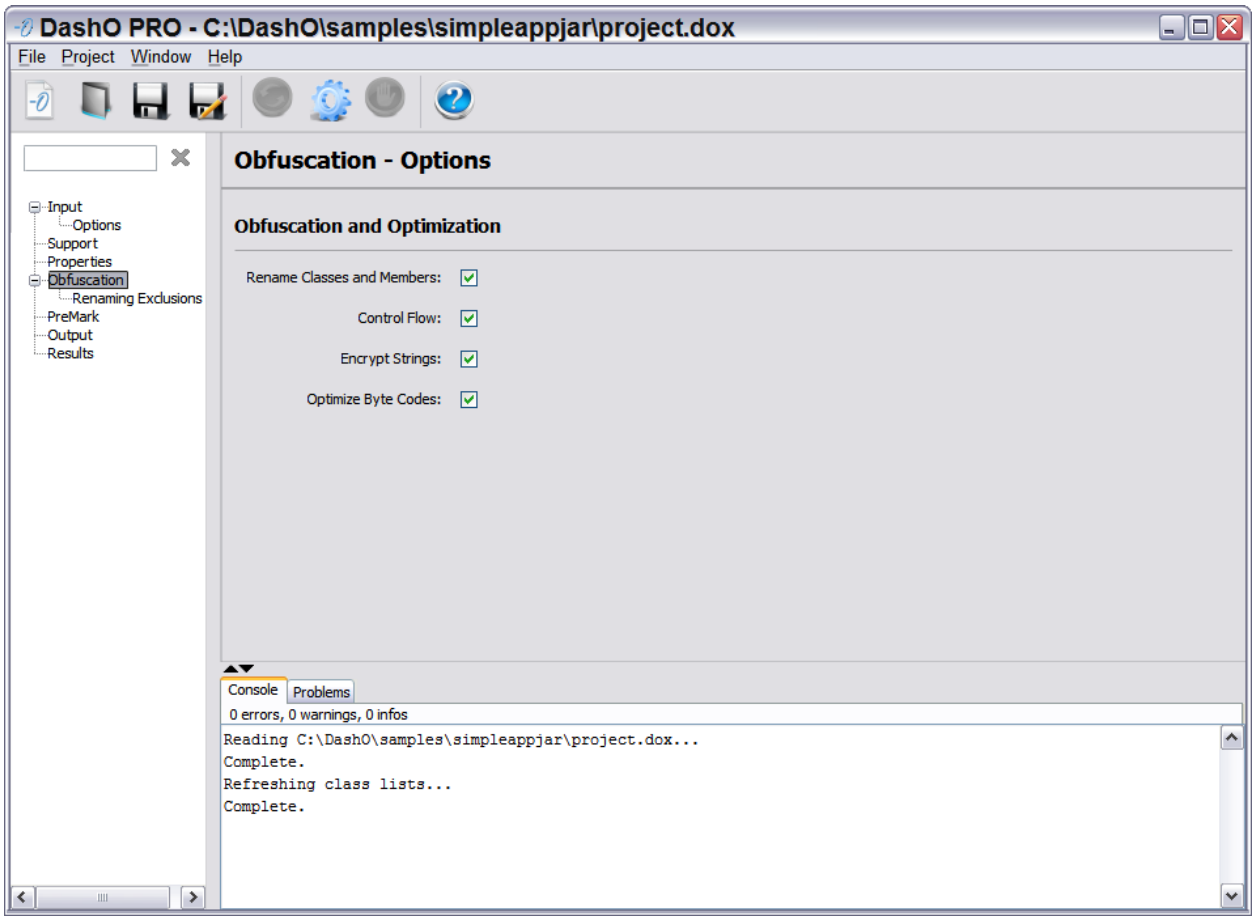
Supporting Classpath names support properties.

# User Properties

The *User Properties* lets you create and assign values to properties that can be referenced in the project. This can allow you to create a project that acts as a template. See the <propertylist> section for information about using properties in your project.



# Obfuscation – Options




## *Rename Classes and Method*

Enables or disables the renaming of class and methods in the project. If the input contain manifests that have `Main-Class` entries those classes and their `main()` method will not be renamed. If the jars are being processed as a library, only non-public items will be renamed.

## *Control Flow*

Enables or disables control flow obfuscation globally.

## *Encrypt Strings*

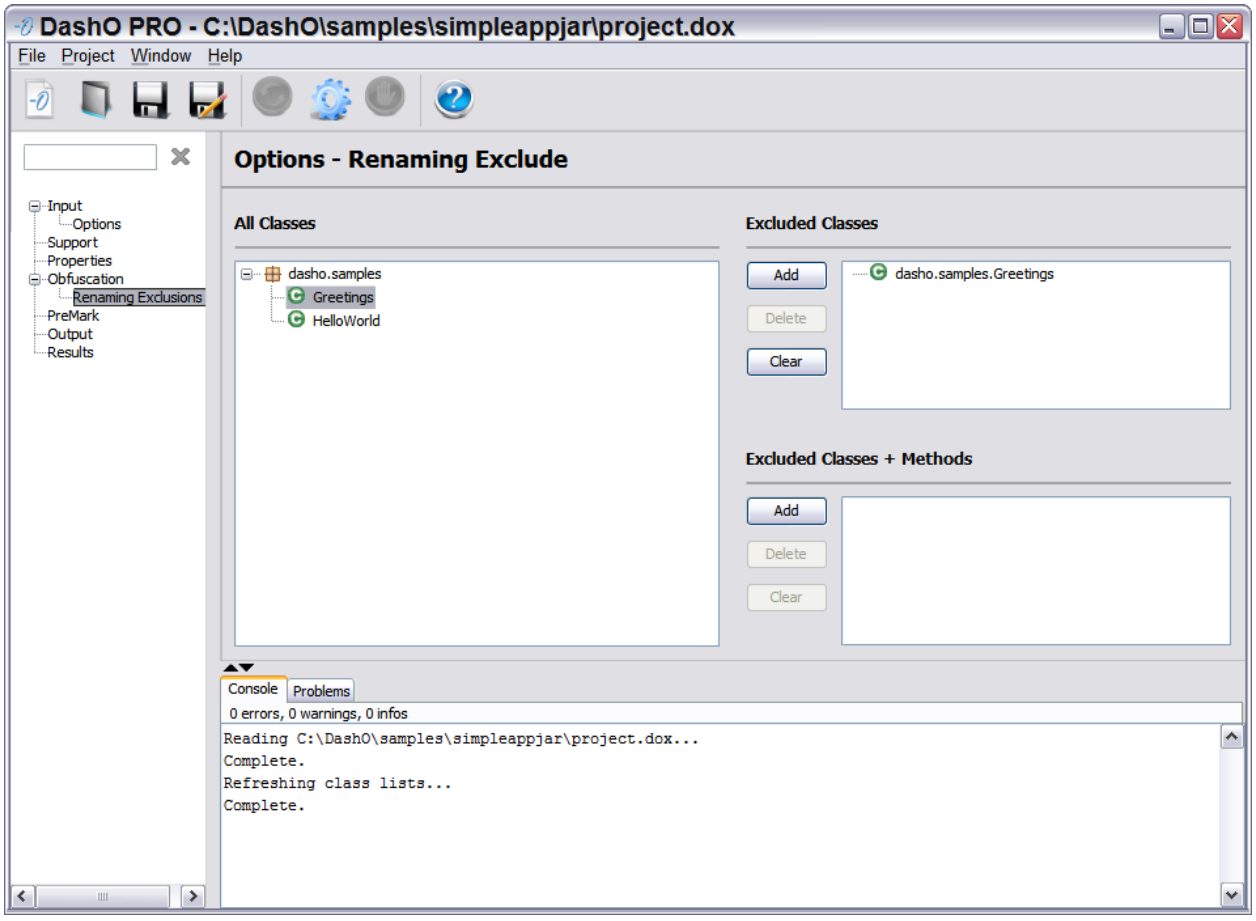
Enables or disable string encryption obfuscation globally.

## *Optimize Byte Codes*

Enables or disables byte code optimization globally.

# Options – Renaming Excludes

The *Options - Renaming Excludes* panel lets you specify classes and/or their methods that are excluded from renaming.




## *Excluded Class*

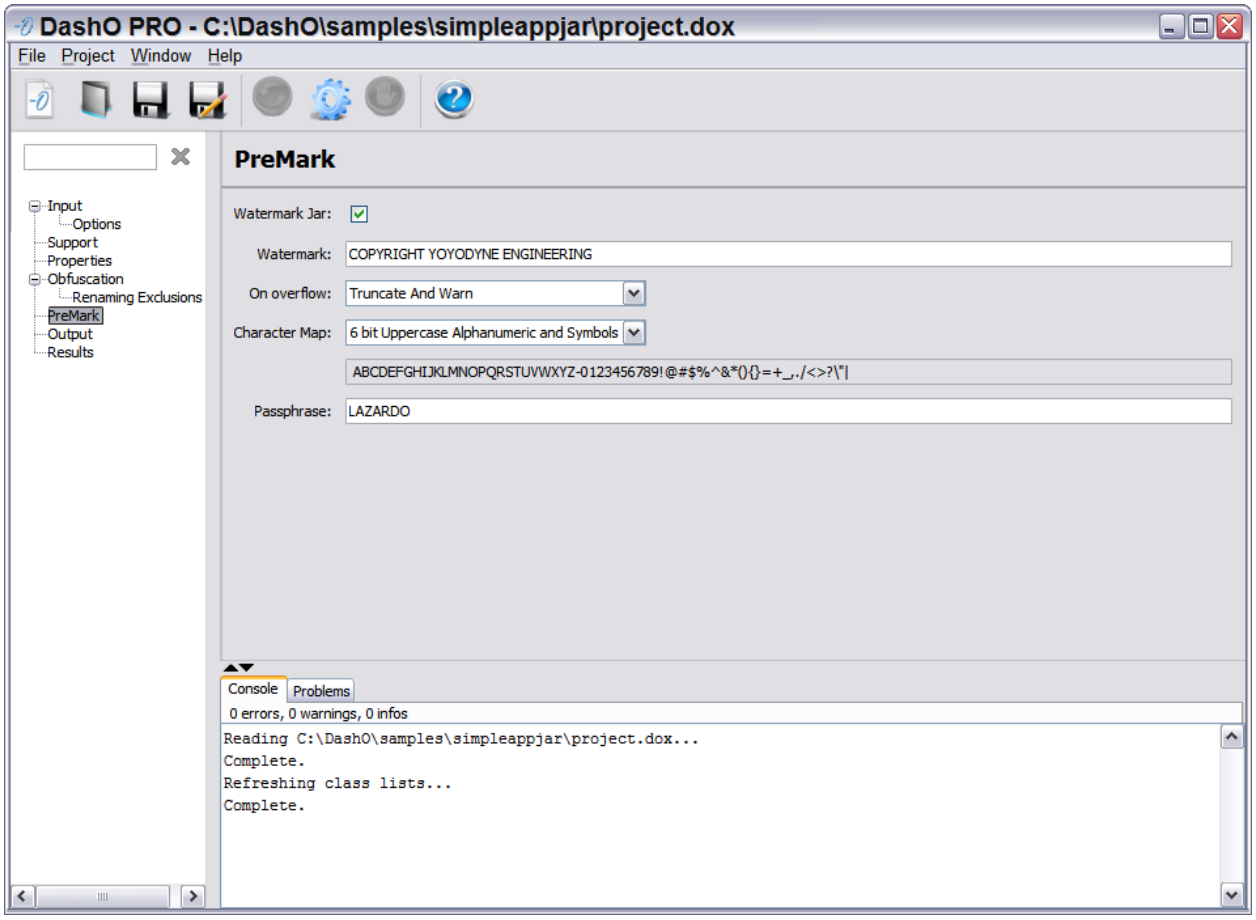
Adding a class or package to the Excluded Classes list instructs DashO that the class should not be renamed. Methods in the class may be renamed. See <classes> Entry Point for details.

## *Excluded Classes + Methods*

Adding a class or package to the Excluded Classes list instructs DashO that the class and its methods should not be renamed. See the <unconditional> section for details.

## PreMark

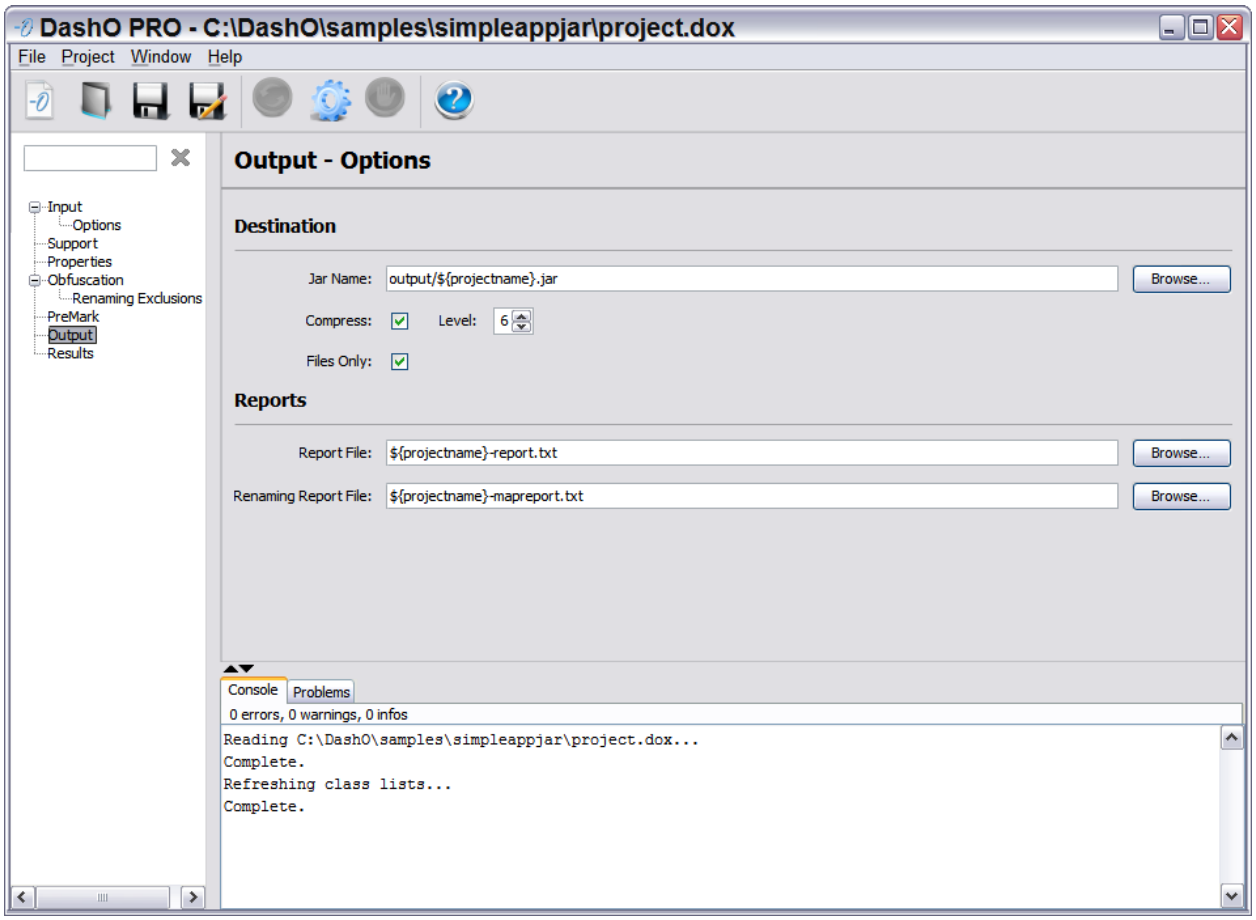
The *PreMark* panel is used to add a watermark to the obfuscated jar produced by DashO. Watermarks can only be applied to jars and this feature will be disabled when DashO's output is to a directory.




See the PreMark section of the Advanced Interface for details.

## Output - Options

The **Destination** is where you specify an output jar file. DashO writes all the obfuscated classes to the output jar. If you know the name and path of the output jar file you want to use, you can enter it directly in the text box. Alternatively, you can browse your file system for the intended file location using the **Browse** button.




**Note**

Only one output jar is generated regardless of the number of input jars specified.

### *Report file*

This specifies the name and location for a report outlining the method/field removal and renaming performed by DashO. A summary is given detailing the total methods/fields/constant pool entries, as well as the final number and percentage of reduction after DashO execution. It also contains information about dynamically loaded classes, including reflection and `forName` calls.

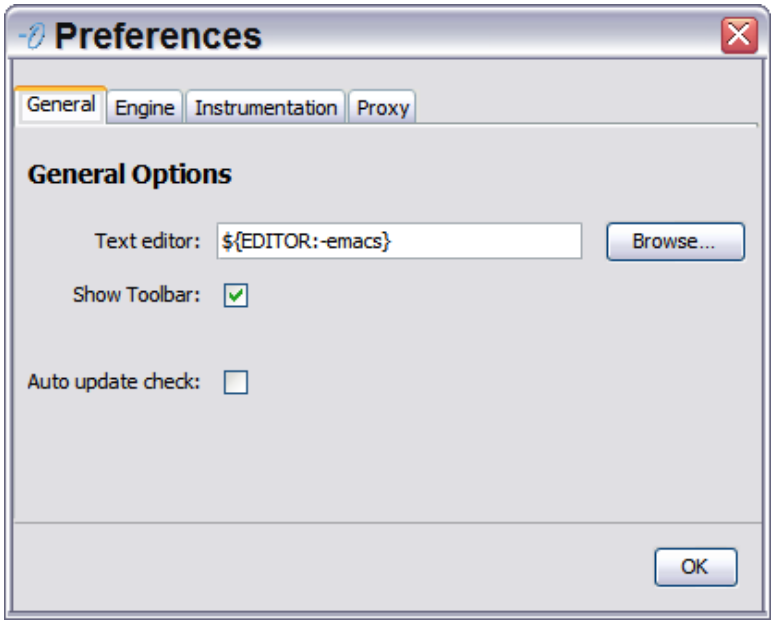
### *Renaming Report File*

This specifies the name and location for a report listing old and new names for renamed classes as well as their renamed members.

# User Preferences

The *User Preferences* dialog lets you configure some aspects of DashO's user interface and pass options to the obfuscation engine. These values are not saved in project files and apply to any projects that DashO has loaded.

## General Options



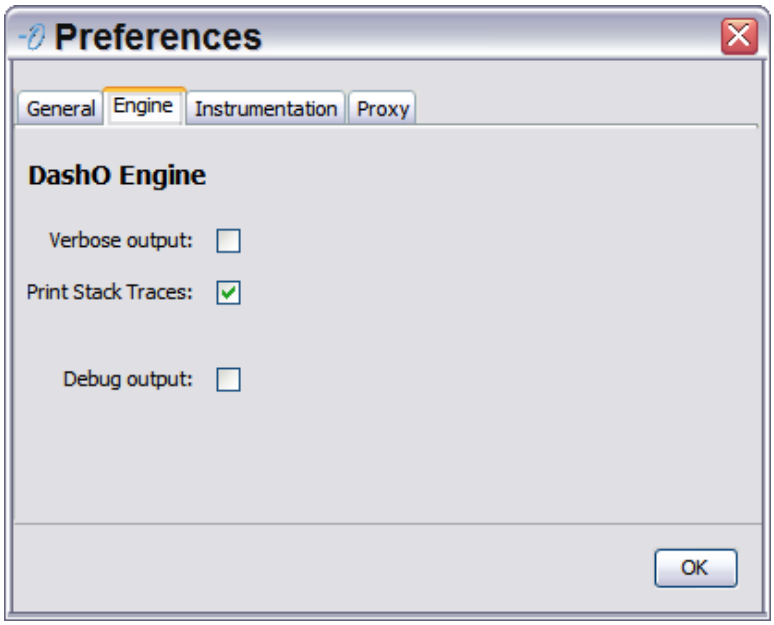

### *Text editor*

This is the name of the application that is used to view the project file and reports generated by DashO. Note that you can use property references when setting the text editor name.

### *Auto update check*

This enables an update check at start-up that sees if an updated version of DashO is available.

## DashO Engine Options




### *Verbose output*

This is the same as passing `--verbose` to DashO's command line version. See DashO Command Line for details. Please note that enabling verbose output can increase the build time.

### *Print Stack Traces*

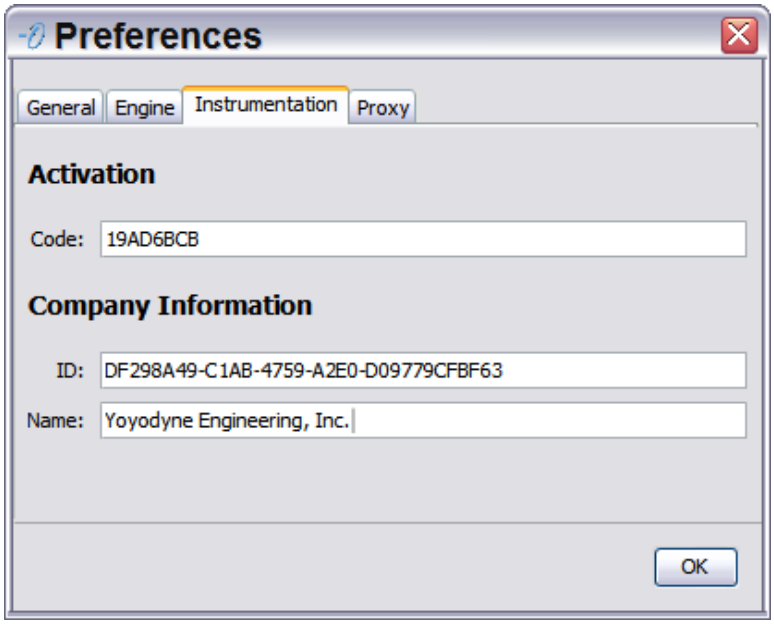
This is the same as passing `--printStackTraces` to DashO's command line version. See DashO Command Line for details.

### *Debug output*

This requests DashO to produce debugging output. In general this option should remain off unless instructed by PreEmptive support staff.

## Runtime Intelligence Options

Application analytics support is disabled in DashO until you sign up for the Runtime Intelligence Service. After enrolling with the service and registering your copy of DashO, you will receive an activation code and company ID from PreEmptive Solutions that will enable this feature.




To activate Runtime Intelligence:

- Click **Help > User Preferences** to open the Preferences Dialog Box. Click on the Runtime Intelligence tab.
- Paste in your **activation code**.
- Enter your **Company ID** and the **Name** of your company in the appropriate fields. The ID and the Name are the default values for your projects. These can be copied into the project by using the arrow in the **Instrumentation – Properties** page.
- Click **OK**.

For more information about obtaining an activation code, see How Do I Get an Activation Code.

# Decoding Stack Traces

One potential drawback of obfuscation is that troubleshooting obfuscated applications is difficult due to name-mangling. DashO addresses this issue by providing an integrated tool that allows you to use your output mapping files to recover the original symbols from obfuscated stack traces.

For example, if you have an obfuscated application that you have shipped and you receive a stack trace from one of your customers, that stack trace might look something like this:

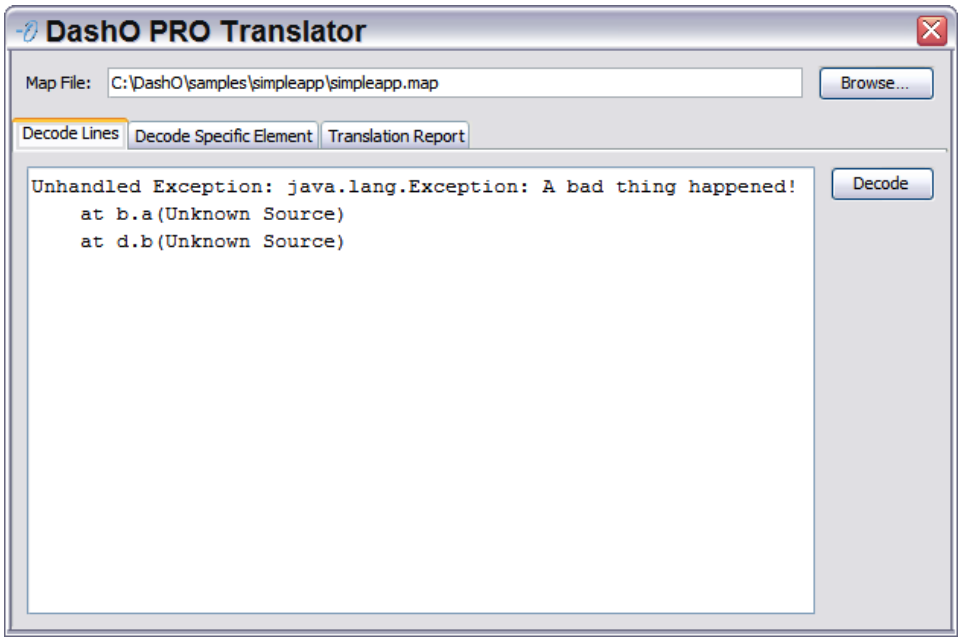
**Example**

```
Unhandled Exception: java.lang.Exception: A bad thing happened!
    at b.a(Unknown Source)
    at d.b(Unknown Source)
```

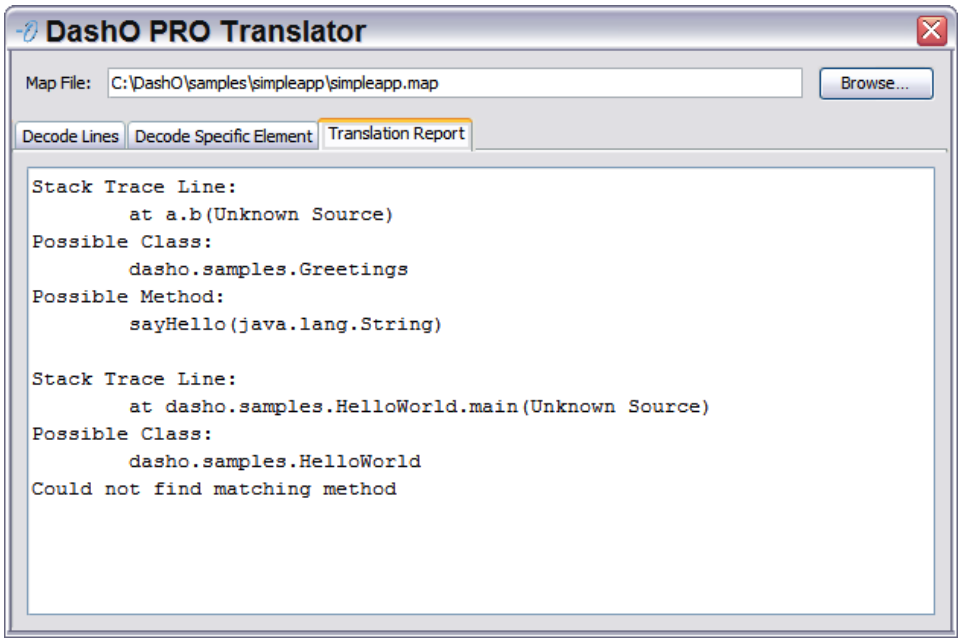
**FIX** Keep in mind that using the `leavedebugginginfo` global option allows your stack traces to keep their line numbers. This tool and that option can greatly improve your ability to debug obfuscated programs.

You can use your mapping report file to manually recover the original names, but this is a tedious and time consuming process.

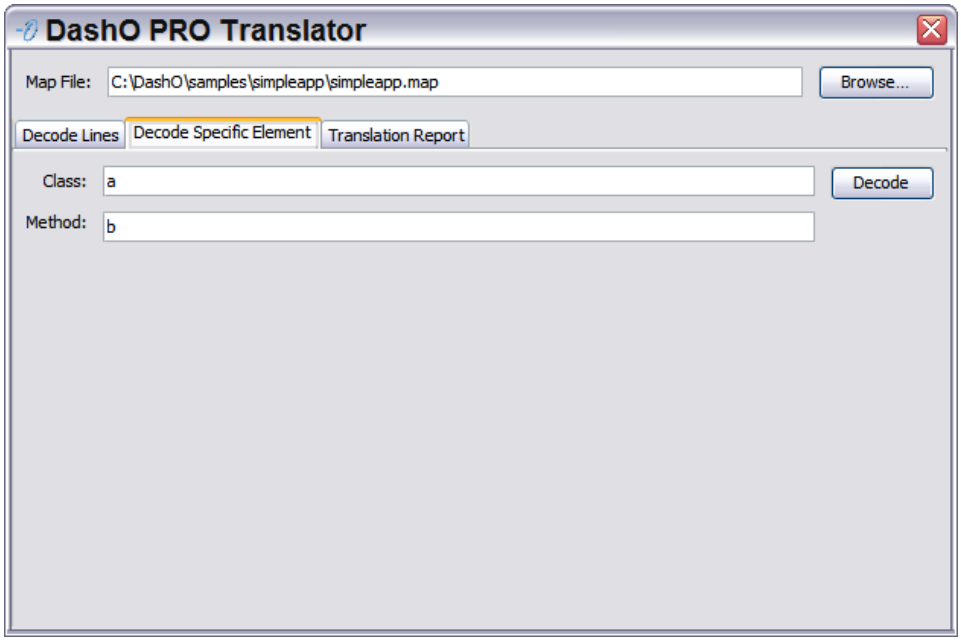
The stack trace translation tool automates this by letting you provide a map output file, paste the stack trace into a window, and press the **Translate** button. The translated stack trace is shown on the **Translation Report** tab.



Some methods in the obfuscated stack trace might be ambiguous: when using Overload Induction there may be more than one matching un-obfuscated method. In these cases all possibilities are displayed.

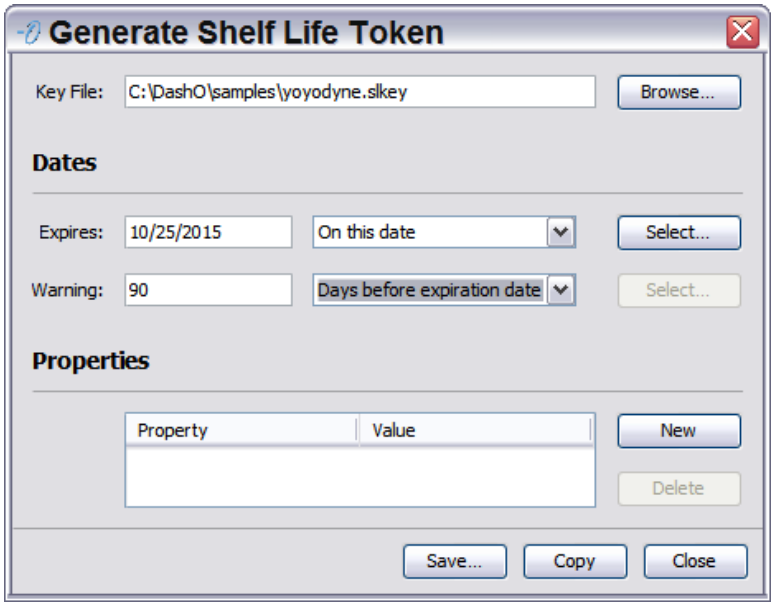



If you just want to look up a specific class or method by name, click the **Decode Specific Element** tab. You will see a screen that will allow you to type in the obfuscated names of the specific items you want to translate.



# Generating Shelf Life Tokens

You can use DashO's user interface to generate Shelf Life tokens that are read in by your application at runtime. The information need to create the tokens is similar to having DashO inject the tokens directly into your application. See Instrumentation – Shelf Life.




## *Save*

The **Save** button lets you save the configured token to a file.

## *Copy*

The **Copy** button copies the token as text to the clipboard so that you can paste it into source code, resources, or property files.

# Using the Command Line Interface

This section describes using DashO as a command line program. The command line interface is designed to allow you to:

- Obfuscate from the command line without requiring you to create a configuration file.
- Override or supplement options in an existing configuration file using command line options.
- Add a watermark to a jar.

## DashO Command Line

Command line options must begin with the '-' character.

**Example**

```
dashocmd [options] [projectfile]
```

The following is a summary of the command line options.

| Options | Description |
|---|---|
| `projectfile` | DashO project file |
| `-h, --help` | Display command line help |
| `-e, --printStackTraces` | Print stack traces for exceptions |
| `-v, --verbose` | Print verbose messages |
| `-q, --quiet` | Print minimal amount of messages |
| `-f, --force` | Force execution |

The `projectfile` is a configuration file that is required for every run of DashO unless Quick Jar mode is specified. Notice you do not enter entry point methods on the command line. This information must be found in the configuration file.

The `-h, --help` option displays command line help on demand.

The `-e, --printStackTraces` option will print stack traces for exceptions.

The `-v, --verbose` option induces DashO to provide printed verbose messages about its progress during execution.

The `-q, --quiet` option tells DashO to run completely and print a minimal amount of messages. This is suitable for inclusion into application build sequences. This option overrides verbose mode.

The `-f, --force` option forces execution even if DashO finds `Class.forName()` methods (discussed in detail in Advanced Topics). The use of the force global option is preferred over the command line use of this option.

## Building Projects from the Command Line

DashO can execute a project file from the command line. To do this, use:

**Example**

```
dashocmd [options] projectfile
```

The project file can be either an advanced mode or quick jar mode project.

# Watermarking PreMark Tool

You can use the PreMark tool to add a watermark or to read the watermark. It is a command line tool to watermark a jar without needing to start DashO. Using this tool you can PreMark any jar file even if it has not been obfuscated by DashO.

To run the command line PreMark tool, use the following command:

**Example**

```
premark [options] inputfile
```

The command line options must begin with the '-' character. The following is a summary of those options.

| Traditional Options | Description |
|---|---|
| `-h, --help` | Display command line help |
| `-e, --printStackTraces` | Print stack traces for exceptions |
| `-v, --verbose` | Print verbose messages |
| `-q, --quiet` | Print minimal amount of messages |
| `--version` | Show version and exit |
| `-r, --read` | Read watermark |
| `-m, --mark <watermark>` | Add watermark |
| `-o, --output <file>` | Output file |
| `-p, --passphrase <passphrase>` | Passphrase to encrypt/decrypt watermark string |
| `-t, --truncate` | Truncate watermark if too big (default: fail) |
| `-c, --charmap <charmap>` | Character map name (6bit-a\|6bit-b\|7bit-a\|4bit-a\|utf8) |

The `-h, --help` option displays command line help on demand.

The `-e, --printStackTraces` option will print stack traces for exceptions. .

The `-v, --verbose` option causes the PreMark tool to provide printed verbose messages about its progress during execution.

The `-q, --quiet` option tells DashO to run completely and print a minimal amount of messages. This is suitable for inclusion into application build sequences. This option overrides verbose mode.

The `--version` option causes the PreMark tool to provide the version of the application and then to exit that application.

The `-r, --read` option reads the watermark string from the specified input file.

The `-m, --mark <watermark>` option watermarks the given input jar with the specified watermark string.

The `-o, --output <file>` option allows you to specify the path to the watermarked output jar.

The `-p, --passphrase <passphrase>` option sets the passphrase. The PreMark tool uses this passphrase to encrypt or decrypt the watermark string.

The `-t, --truncate` option truncates the watermark string if it is too long. If this option is not specified, the default is to halt without watermarking the file.

The `-c` option tells the PreMark tool which character map encoding should be used to embed the watermark string in the given input jar.

**Note**

The value of `charmap` can be `6bit-a, 6bit-b, 7bit-a, 4bit-a`, or `utf8.`

# Advanced Topics

This section describes different scenarios and issues encountered when obfuscating Java applications and libraries.

## Overload-Induction Method Renaming

DashO implements patented technology for method renaming called Overload-Induction™. Whereas most renaming systems simply assign one new name per old-name (*i.e.* `getX()` will become `a()`, `getY()` will become `b()`), Overload-Induction induces method overloading maximally. The underlying idea being that the algorithm attempts to rename as many methods as possible to exactly the same name.

The original source code before obfuscation:

**Example**

```
private void calcPayroll(SpecialList employeeGroup) {
    while (employeeGroup.hasMore()) {
        employee = employeeGroup.getNext(true);
        employee.updateSalary();
        distributeCheck(employee);
    }
}
```

And the reverse-engineered source after Overload Induction:

**Example**

```
private void a(a b) {
    while (b.a()) {
        a = b.a(true);
        a.a();
        a(a);
    }
}
```

One of the things you probably noticed about the example is that the obfuscated code is more compact. A positive side effect of renaming is size reduction. For example, if you have a name that is 20 characters long, renaming it to `a()` saves a lot of space (specifically 19 characters). This also saves space by conserving string heap entries. Renaming everything to `a` means that `a` is stored only once, and each method or field renamed to **a** can point to it. Overload Induction enhances this effect because the shortest identifiers are continually reused.

# Dynamic Class Loading

The `forName()` method of `java.lang.Class` is the way to load classes dynamically at runtime. It is impossible for DashO to determine what classes are dynamically loaded in all cases. Consider the following code:

**Example**

```
public Object getNewClass() {
    String newClassName = getUserInputString();
    try {
        Object newClass = Class.forName(newClassName).newInstance();
        return newClass;
    }
    catch(Exception e) {
        // handle
    }
}
```

This code loads a class by name and dynamically instantiates it. In addition, the name comes from a string input by the user. There is no way for DashO to predict which class names the user will enter. The solution is to exclude the names of all potentially loadable classes (method and field renaming can still be performed). This is where manual configuration is required.

**Note**

Incorrect specification of dynamically loaded classes can cause obfuscated applications to fail at runtime.

## Predictable Dynamic-Loading

The simplest case is when you know your application well enough to know exactly what classes could be loaded via dynamic-loading. If the dynamically loaded classes share a base class or common interface:

**Example**

```
String s = getShapeName();
Shape myShape = (Shape)Class.forName(s).newInstance();
myShape.draw();
```

In this example, DashO's can detect this pattern automatically and include all `Shape` classes. If another type of creation pattern is used the classes would need to be added individually in the entrypoints section of the DashO configuration file:

**Example**

```
<entrypoints>
    <classes name="Triangle"/>
    <classes name="Rectangle"/>
</entrypoints>
```

In this case DashO will be able to remove unused methods from the `Shape` hierarchy.

## Unpredictable Dynamic-Loading

In cases where the dynamically loaded classes would not know at the time the application is obfuscated, for example, a user interface building application could allow users of the application to include their own or third-party components, the existing classes must be added as unconditional entry points::

**Example**

```
<entrypoints>
    <unconditional name="Triangle"/>
    <unconditional name="Rectangle"/>
</entrypoints>
```

This has several ramifications:

- Regardless of removal options, no methods or fields will be removed from an unconditionally included class.
- Regardless of renaming options, neither the class nor its members will be renamed.
- All methods within the class will be treated as entry point methods.

These rules enforce the idea that your interface to as-yet-unknown classes will remain intact.

## Reflection Report

DashO has several facilities to allow you to specify how or what is dynamically loaded. The `fornamedetection` option in DashO handles most or all dynamically loaded class instances.

**Example**

```
<global>
    <option>fornamedetection</option>
</global>
```

DashO reports all places it finds usage of `forName().` This is provided as part of the report file and as output after dependency analysis. Note that the `fornamedetection` option will not give a wrong answer but it may give no answer at all. Manual configuration is required in those instances where DashO reports "unable to determine" dynamically loaded class.

**Example**

```
NOTE:
Reflection use public void com.yoyodyne.Application.getInterface() -
     java.lang.Class.newInstance() -
     [BaseInterface Possible: InterfaceImplementor]
Reflection use public boolean com.yoyodyne.Test.connect() -
     Class.forName()
Reflection use public float com.yoyodyne.Test.calculate() -
     Class.forName() - [com.yoyodyne.Linker]
```

Since DashO is unable to determine what class is dynamically loaded in the method `connect()` manual configuration becomes necessary. The class that is dynamically loaded in this method must be included using the `<classes>` tag under the <entrypoints> section.

If DashO finds reflection usage and you do not specify the force global option, DashO will not create any output classes or jars.

# Serialization

If your application is brand new, meaning there are no existing serialized objects within your application, then serialization may not be an issue for you. Classes that were serializable before DashO obfuscated them will still be serialized afterwards.

If you have persistent objects already in existence, then you need to identify which classes they were created from before running DashO. Method/field removal and renaming will make reloading these objects impossible. The simple solution is to unconditionally include the classes. List all your to-be-serialized objects there.

DashO automatically keeps fields with the name `serialVersionUID` intact (no removal or renaming) to facilitate compatibility between versions. In addition, if the `readObject()`, `writeObject()`, `writeReplace()`, or `readResolve()` methods of the serializable framework are used DashO will automatically treat them as entry points.

# Runtime Intelligence

This section documents the development process when using Runtime Intelligence. It describes the different options available using the custom and extended attributes and provides examples to illustrate.

## Overview

Runtime Intelligence is a technology and a service that gives application authors and users insight into how their applications are being used. DashO can be used to Runtime Intelligence enabled Java applications and components.

DashO can instrument an application such that a message is sent when the application starts and stops or when a designated feature is being used. The Runtime Intelligence Service aggregates this lifecycle data from the application and exposes it through the Runtime Intelligence Portal, available to Runtime Intelligence Services subscribers. To use this functionality, you must first activate Runtime Intelligence from within DashO and be a Runtime Intelligence Services subscriber.

DashO adds Runtime Intelligence support to the application based on guidance provided from custom annotations. When run on a properly annotated Java application, DashO processes the annotations and instruments the application accordingly. The resulting output application will be ready to send Runtime Intelligence data to the service.

### Message Types

Runtime Intelligence defines several message types:

- Application and Session Start
- Application and Session Stop
- Feature
- Performance Probe
- System Profile

Application and Session Start and Stop messages, the application lifecycle messages, are sent when an application starts running and when it shuts down. The information contained in these messages tracks application behavior and usage patterns. Extended usage and environment information is obtained by using the Feature, Performance Probe, or System Profile messages.

The data from these messages drive the Runtime Intelligence Portal's dashboards. To have your application send these messages, you must:

- Be a Runtime Intelligence Services subscriber. This gives you access to the dashboards and data in the portal.
- Activate Runtime Intelligence from within DashO.
- Annotate your application with Application Start and Stop.
- Run your application through DashO with instrumentation turned on.

# Activating Runtime Intelligence

Application analytics support is disabled in DashO until you sign up for the Runtime Intelligence Service. To sign up for the service and to obtain the proper credentials to access the Runtime Intelligence Portal, do the following:

1. Register your copy of DashO.
2. Copy your serial number from the splash screen and then email your serial number to support@preemptive.com, requesting an activation code.
3. Once your request for Runtime Intelligence Services is processed you will receive an email containing your activation code, company key, a link to the Runtime Intelligence web portal, your portal login ID, and your portal login password.
4. To activate Runtime Intelligence, open the User Preferences dialog.
5. Enter, or copy and paste, in the activation code that was sent to you in the email.
6. Click **OK**.
7. Click the web portal link which was included in the email to login.

For more information about obtaining an activation code, see How Do I Get an Activation Code.

# Custom Annotations

All Runtime Intelligence custom annotations are defined in `dasho-annotations.jar`, which is located in the lib folder where you installed DashO. To add Runtime Intelligence custom annotations to an application, add a reference to this jar that must be available at compile time. While injecting Runtime Intelligence code, DashO removes references to these annotations; therefore, the jar is not required at application runtime and does not need to be distributed with the application.

In addition to using the custom annotations, all Runtime Intelligence annotations may be specified as virtual annotations using the DashO User Interface. If you use virtual annotations you do not have to modify the application source code. For a programmer's reference, see the javadocs.

## Feature Usage Tracking

DashO provides support for feature usage tracking via the feature annotations. The developer may add a feature annotation to any method which maps to the start, stop, or entirety of a feature. When DashO encounters a feature annotation during its processing it adds code to the method to send a Runtime Intelligence message.

### Feature Name

In order to make sense of feature-level analytics, features must be identified by a name. The name is a string value which defines the name of the feature in question. This name need not follow any particular convention; but it should be descriptive and unique, except in cases where the feature in question is one half of a start-stop pair in which case, the feature names must match.

## Feature Event Types

DashO has three annotations for denoting the event type.

- `FeatureTick` – a feature has been executed.
- `FeatureStart` – a feature has been started.
- `FeatureStop` – a previously started feature has ended.

`FeatureStart` and `FeatureStop` are used to compute execution time for a feature in addition to tallying how many times it has been used. `FeatureTick` is used to only tally usage.

**Example**

```
@FeatureStart("Find")
private void beginFind() {
    // ...
}
```

If a method's logic fully encompasses a feature, you may place a start and stop annotation on the method. DashO sends the start message when the method begins and the stop message when the method completes.

**Example**

```
@FeatureStart("Find")
@FeatureStop("Find")
private void doFind() {
    // ...
}
```

# Gathering Performance Information

Runtime Intelligence code can be used to gather and send performance related information while the application is executing. To add support for this to an application, place a `PerformanceProbe` annotation on a method or methods in the application. When DashO encounters the attribute during its processing, it adds code to obtain performance information and send a message to the Runtime Intelligence Service.

Performance data collected includes:

- CPU Utilization
- Memory available
- Memory used by current process

**Example**

```
@PerformanceProbe
public void doSomething() {
    // ...
}
```

The collected performance data is available in the *Data Extract* report on the Runtime Intelligence Portal. It can also be downloaded from the *File Feeds* section.

# Gathering Environment Information

Runtime Intelligence code can be used to gather and send information about the system the application is running on. To add support for this to an application, place a `SystemProfile` attribute on a method in the application. When DashO encounters the attribute during its processing, it adds code to gather the system profile and send a message to the Runtime Intelligence Service. Typically this data only needs to be collected once during an application run.

Below is a high level description of the kind of system data that is gathered:

- **Processors** – Number of processors, clock speeds, manufacturer, and processor ID
- **Logical Disks** – Number of logical disks, volume name, size, free space, file system
- **Memory** - Speed, capacity
- **Network Adapters** - IP address, MAC address
- **Domain**[2] - Domain name and role
- **Display** - Name, refresh rate, vertical and horizontal resolution
- **Video** - Name, memory size, color depth
- **Terminal Services**[3] - Connections allowed
- **Sound** – Name, manufacturer
- **Modem** – Model, device type

[2] Windows only.
[3] Windows only.

**Example**

```
@SystemProfile
public void initialize() {
    // ...
}
```

The collected data is available in the *Data Extract* report on the Runtime Intelligence Portal. It can also be downloaded from the *File Feeds* section.

# Sending User Defined Data

Most Runtime Intelligence message types allow user defined data in the form of key-value pairs to be gathered and sent along with the message. To send this information, specify a `PropertySource` on the method.

DashO uses the `PropertySource` to generate code that gathers the key-value pairs at runtime. The `PropertySource` is a source for a Properties instance, either a field or method. See Specifying Sources for more information.

**Example**

```
@FeatureTick("Click")
@PropertySource("getProperties")
private void buttonClick(JComponent sender) {
    // ...
}

// Creates and populates custom properties
private Properties getProperties() {
    Properties props = new Properties();
    props.setProperty("key1", "val1");
    props.setProperty("key2", "val2");
    props.put("numeric", new Integer(934));
    return props;
}
```

Properties sent by the application are available in the Data Extract report on the Runtime Intelligence Portal. It can also be downloaded from the *File Feeds* section.

# Download Message Data

The Runtime Intelligence Portal provides the capability to securely download raw message data originating from instrumented applications. The data is available in CSV files compatible with MS Excel or OpenOffice Calc.

To obtain raw message data, access the Runtime Intelligence Portal at http://runtimeintelligence.com. Enter the **User Name** and **Password** provided by PreEmptive Solutions and then navigate to *File Feeds*, located under *Data Extracts*.

# Tamper Checking and Response

DashO can instrument applications to detect if they have been tampered with and optionally send a message to the Runtime Intelligence Service. When the service receives the message it notifies the application owner via email. Tamper checking requires that your application be signed either by DashO or by another process following instrumentation by DashO. The tamper checking and response are implemented using Runtime Intelligences Custom Annotations that can either be placed in your source code or added via Virtual Annotations.

To use this feature you must first activate Runtime Intelligence from within DashO.

## Tamper Checking

To detect tampering place a `TamperCheck` on one or more methods in your application. DashO adds code that performs a runtime check that verifies the code has been signed by a particular certificate. If the check fails you can respond to it in one or more ways. You can choose one or all of the following at the time the check is performed:

- **Send a Runtime Intelligence tamper message.**
  A tamper message will be sent to the Runtime Intelligence server. The default is to not send a message. If your application is using analytics and contains an `ApplicationStart` you need no further configuration. If you are only using `TamperCheck`s then you need to supply the company and application IDs using other annotations or provide them on the Instrumentation Properties panel.
- **Call a method or set a field.**
  You can have the tamper state passed back to your application by invoking a method that takes a single `boolean` or by setting a `boolean` field. When a tamper check fails the boolean value is `true`. If the check passes then `false` is used. Your application can act on this information immediately or store it for later interaction with a TamperResponse annotation.
- **Perform a response.**
  There are several immediate responses that can be taken: `exit` – exit the application with a randomly non-zero return code; `hang` – cause the current thread to hang; `error` – throw a randomly selected error; `exception` – throw a randomly selected unchecked exception. If the value is left blank then the default response of `none` is taken. The randomization of return codes and `Throwable`s is performed at time the check is injected not at run time. Errors and exceptions are thrown with an empty stack trace to conceal their origin.

When you select more than one of these actions they are performed in the order listed above. If you do not request any of these or none are valid the tamper check will be skipped and DashO will produce a warning message.

An application can contain any number of `TamperChecks` with various configurations. Using more than one check or mixing the responses will hamper attackers.

**Examples**

```
private static boolean tamperFlag;

@TamperCheck(sendMessage=true, action="@tamperFlag")
public static void main(final String[] args){

}

@TamperCheck(response="hang")
private int computeResult(){

}
```

### Interaction with Signing

The tamper check is performed by verifying at runtime that the code has been signed by a particular certificate. If DashO is used to sign the resulting jars that no further configuration is required. If the jars are signed by another process after using DashO to add tamper checking you need to tell DashO about the signing information using the `SignerInfo` annotation. This allows DashO to retrieve the key information required to perform the runtime tamper checking. The `SignerInfo` lets you specify information similar to what is found on the Output Signing panel.

**Examples**

```
@SignerInfo(storepass="${master.psw}", storetype="JKS", alias="ProdKey")
@TamperCheck(sendMessage=true, action="@tamperFlag")
public static void main(final String[] args){

}
```

When you use the user interface to enter a password for `storepass` value and it does not contain property references DashO will store the password in an encrypted form.

## Tamper Response

Separating the detection and response makes it more difficult for attackers. Having multiple and different responses scattered through-out the application increases the difficulty. Making those responses non-deterministic can make the process maddening. DashO lets you configure your response to a tampered application as simple or as complex as you desire.

The `TamperResponse` annotation adds code that interacts with a `TamperCheck` to separate the detection and response code. You can add one or more TamperResponses to your application.

The `TamperResponse` coordinates with the `TamperCheck` via a `boolean` value. A value set using the `TamperCheck`'s action is retrieved with the `TamperResponse`'s source. If the retrieved value is `true` then the response is executed.

Like the `TamperCheck` the `TamperResponse` can send a Runtime Intelligence message and/or perform a response. In addition the response action can be made conditional based on a probability factor ranging from `0.0` (never) to `1.0` (always) – the default is `1.0`.

```
private static boolean tamperFlag;

@TamperCheck(action="@tamperFlag")
public static void main(final String[] args){

}

@TamperResponse(source="@tamperFlag", sendMessage=true)
private void init() {

}

@TamperResponse(source="@tamperFlag", response="exit", probability=.05)
private int computeResult(){

}

@TamperResponse(source="@tamperFlag", response="error", probability=.1)
private FileInputStream readInput(){

}
```

When you are requesting the sending of Runtime Intelligence message with `TamperResponse`s you may need to provide some additional configuration information. If your application is using analytics and contains an `ApplicationStart` you need no further configuration. If you are using `TamperResponse`s that send messages then you need to supply the company and application IDs using other annotations or provide them on the Instrumentation Properties panel.

# Shelf Life

Shelf Life is an application inventory management function that allows you to add expiration and notification logic to your application. This logic enforces an expiration policy by exiting the application and/or sending a Runtime Intelligence message. For example, a beta application can be made to expire on a particular date. You can schedule an application's expiration for a specific date or a number of days from a starting date and optionally specify a warning period prior to expiration. The expiration information may be placed within your application or can be read from an encrypted external token file. The latter allows you to extend the expiration of the application by issuing a new token file rather than rebuilding your application. Expiration checks can be added to one or more locations in your application.

## Activation Key

To start using Shelf Life you must obtain a Shelf Life Activation Key from PreEmptive Solutions. This key is used to generate the tokens that contain the expiration information. PreEmptive will issue you a data file containing the key that generates the tokens and identifies your application on the Runtime Intelligence Portal. This key is read by DashO when your code is instrumented and can either be specified in the user interface or via a Shelf Life annotation.

## Shelf Life Tokens

A Shelf Life Token is an encrypted set of data containing application and expiration information. It can be inserted into your application or stored outside of the application. You can use the DashO user interface or an Ant task to create an externally stored token.

The expiration and warning information for the token is entered via the user interface or via Shelf Life annotations. The annotations can either be added to your source or added with DashO's virtual annotations. Expiration and warning dates can be specified in two different ways:

**Absolute Dates** – A fixed date for the expiration date or the beginning of the warning period can be specified.

**Relative Dates** – The expiration period is the number of days from a start date. The warning period is the number of days prior to the expiration date.

You can combine absolute and relative dates - e.g. expire on 1/1/2021 and warn 30 days before expiration. Expiration information is required to create the token, but warning information is optional.

## Expiration Check

The `ExpiryCheck` annotation is used to define the location in your application where an expiration check will take place. The `ExpiryCheck` can be added to your source or added with DashO's virtual annotations. If you added the annotation to your source you will need to compile with `dasho-annotations.jar` which is located in the `lib` folder where you installed DashO. By default, DashO removes references to these annotations; therefore, the jar is not required at application runtime and does not need to be distributed with your application.

If the expiration information is set on the Instrumentation – Shelf Life screen, at the minimum a key file and expiration date, only a single annotation is required to add the expiration check:

**Example**

```
@ExpiryCheck
public static void main(final String[] args){
    if(args.length == 0){
        System.out.println("Hello no name");
    }else{
        System.out.println("Hello " + args[0]);
    }
}
```

This adds an expiration check to the application at the start of `main()`. You can also specify all the information as annotations:

**Example**

```
@ExpiryKeyFile("yoyodyne.slkey")
@ExpiryDate("01/01/2021")
@WarningPeriod("30")
@ExpiryCheck
public static void main(final String[] args){
    // ...
}
```

The values for the annotations, dates and periods, are all strings. This allows you to use DashO's properties or environment values to parameterize them:

**Example**

```
@ExpiryKeyFile("${key_dir}/yoyodyne.slkey")
@ExpiryDate("01/01/${exp_year}")
@WarningPeriod("${warn_period}")
@ExpiryCheck
public static void main(final String[] args){
    // ...
}
```

## Relative Expiration Date

Expiration can be specified as a number of days from a dynamic start date. The start date could be something like the install date or the date on which the application is first run. The start date is provided at runtime by your application:

**Example**

```
@StartDateSource("getInstallDate()")
@ExpiryPeriod("90")
@ExpiryCheck
public static void main(final String[] args){
    // ...
}

private static Date getInstallDate(){
    return new Date(Preferences.userRoot().node("MyApp").
                    getInt("installDate", 0));
}
```

You can use static or instance methods or fields as a source for the start date. See Specifying Sources for details.

## Externally Stored Tokens

In the previous example DashO has embedded the Shelf Life token into your application. The token can also be stored externally as a file or resource and read in at run-time:

**Example**

```
@ExpiryTokenSource("getToken()")
@ExpiryCheck
public static void main(final String[] args){
    // ...
}

private static Reader getToken(){
    return new InputStreamReader(HelloWorld.class.getClassLoader().
                                 getResourceAsStream("expiry.dat"));
}
```

The source for the token is a static or instance method that returns a `java.io.Reader` that provides the token data. See Specifying Sources for details.

## Expiration Action

When `ExpiryCheck` is executed, the default action is to print a message to `System.out` and to exit with a non-zero return code:

**Example**

```
This application expired on January 1, 2010
```

If the application is in the warning period a message is printed to `System.out` and execution continues:

**Example**

This application will expire on January 1, 2011

For a more sophisticated application a custom application action can be specified:

**Example**

```
@ExpiryCheck(action="check()")
public static void main(final String[] args){
    // ...
}

private static void check(Token token) {
    if(token.isExpired()){
        JOptionPane.showMessageDialog(null,
            "The application expired on " + token.getExpirationDate(),
            "Expired",
            JOptionPane.ERROR_MESSAGE);
        System.exit(1);
    }
    if(token.isInWarning()){
        JOptionPane.showMessageDialog(null,
            "The application will expire in " +
                token.getDaysTillExpiration() + " days",
            "Expiration Warning",
            JOptionPane.WARNING_MESSAGE);
    }
}
```

The action is passed the Shelf Life token which is then used to determine the action to be taken.

## Shelf Life Runtime Intelligence Messages

Shelf Life can send Runtime Intelligence messages so you can track applications that have expired or are about to expire. The `ExpiryCheck` has a `sendMessage` property:

**Example**

```
@ExpiryCheck(sendMessage=true)
public static void main(final String[] args){
    // ...
}
```

If your application already contains an `ApplicationStart` you do not need to add any additional annotations. If you are going to use Runtime Intelligence for tracking expiration you must add annotations that will identify your application:

**Example**

```
@ExpiryCheck(sendMessage=true)
@CompanyId("DF29A894-C1AB-5947-E0A2-0D9779CFFB63")
@ApplicationId("F0000FDA-9500-1B92-9564-A9DA3D8C3CF0")
public static void main(final String[] args){
    // ...
}
```

The `ExpiryCheck` will then automatically handle the `ApplicationStart` and `ApplicationStop` to send your expiration messages to the Runtime Intelligence Portal.

**Note**

You can also add any of the following annotations with the `ExpiryCheck` to identify your application: `Company`; `CompanyId`; `CompanyName`; `Application`; `ApplicationId`; `ApplicationName`; `ApplicationType`; `ApplicationVersion`; `ApplicationVersionSource`; `ApplicationInstanceIdSource`; `UseSsl`.

# Exception Reporting

DashO can instrument applications to report on unhandled exceptions thrown by an application and optionally send a message to the Runtime Intelligence Service. Additionally the application can be instrumented to report on exceptions that are caught, uncaught, or thrown at the method level. The exception reporting is implemented using Runtime Intelligences Custom Annotations that can either be placed in your source code or added via Virtual Annotations.

To use this feature you must first activate Runtime Intelligence from within DashO.

## Application and Thread-level Reporting

Unhandled exceptions can be intercepted and reported at either the application level or on a per-thread basis. The unhandled exceptions can be sent directly to the Runtime Intelligence Service for non-GUI applications without user interaction. For GUI applications you can select to have a dialog box presented to your application's user so they may choose to sent the report or not. They will also be able to enter information about the activities they were performing prior to the exception as well as some contact information. This information is optional, but if entered is available on the Runtime Intelligence Portal along with the exception information.

Application and thread-level reporting is added with the `AddUncaughtExceptionHandler` annotation. Properties of the Annotation determine if the handler is installed as the default handler or only for the current thread and whether a dialog is displayed to the user.

**Examples**

```
@AddUncaughtExceptionHandler(showDialog=true)
public static void main(final String[] args){
    // ...
}

new Thread() {
    @AddUncaughtExceptionHandler(thread=true)
    public void run() {
        // ...
    }
}.start();
```

Allowing the user to interact with a dialog gives them an opportunity to override the global opt-in setting. If the user chooses to send the report it will override an opt-out from other Runtime Intelligence messages. Your application will still require the configuration information that will allow it to be identified to the Runtime Intelligence Server. If the dialog is requested in a non-GUI application the report will only be sent if the user has opted-in to sending messages.

If you choose to send the report without user interaction the report will only be sent if the user has opted-in to sending Runtime Intelligence messages.

**Note**

These features are available as an API in the `ExceptionHandler` class.

# Method-level Reporting

If you require a more fine grained approach to the reporting of exceptions you can use Annotations to track exceptions at the method level. DashO provides three Annotations that are used to add method level exception reporting: `ReportCaughtExceptions`; `ReportThrownExceptions`; `ReportUncaughtExceptions`. All three annotations support the following behaviors:

- **Send a Runtime Intelligence fault message.**
  A fault message will be sent to the Runtime Intelligence server. This is the `sendMessage` property. The default is to send a message. To send the message your application must contains an `ApplicationStart` and the user must opt-in to the sending of messages.
- **Call a method or set a field.**
  You can have the exception passed back to your application by invoking a method that takes a `Throwable` or by setting a `Throwable` field. This is the `action` property.

If you use both behaviors the sending of the message is performed before the action. The following example show how the reporting of exceptions can be added at the class level so that it applied to all methods in the class. In the example the messages are sent to the Runtime Intelligence server as well as logged locally using Log4J.

**Examples**

```
import org.apache.log4j.Logger

@ReportCaughtExceptions(action="@onCatch()")
@ReportThrownExceptions(action="@onThrow()")
class MyClass {
    private final static Logger log = Logger.getLogger(MyClass.class)

    public void execute(){
        // ...
    }

    private static void onCatch(Throwable t){
        log.info("MyClass caught " + t.getClass().getName(), t);
    }

    private static void onThrow(Throwable t){
        log.warn("MyClass threw " + t.getClass().getName(), t);
    }

}
```

In addition to these previously described properties the `ReportUncaughtExceptions` annotation allows you to ignore the unhandled exception. Methods that have a numeric return will return zero when an unhandled exception is ignored. Methods that return objects or arrays will return `null`.

In the following example calling the `div(x, 0)` could cause an `ArithmeticException` to be thrown and printed to `System.err` but the method would return zero.

**Example**

```
@ReportUncaughtExceptions(sendMessage=false, action="onErr()",
                          ignore=true)
int div(int num, int denom){
    return num / denom;
}

void onErr(Throwable t){
    t.printStackTrace();
}
```

# Project File Reference

This section documents DashO's XML project file. It contains detailed descriptions of each option, making it useful as a reference, even if you are using the user interface to generate a file for you.

DashO project files may have any name or extension, but the preferred extension is `.dox`. Project files contain information about how a given application is to be obfuscated. The project file is an XML document conforming to `dasho.xsd` distributed with DashO.

## <dasho>

The `<dasho>` tag is the outermost tag of the `.dox` file.

### Version Attribute

The file version is a required attribute. It specifies the earliest version of DashO that is capable of reading the project file. For example, you should be able to use a `version="5.2"` project with version 6.5 of DashO without having to edit the project.

**Example**

```
<dasho version="6.7.1">
```

**Note**

DashO may create project files with versions different from the application version. The file version represents the minimum version of DashO that is able to use the project file.

## <propertylist> Section

The optional property list section allows for the definition and assignment of variables called properties. These may be used in the project file or to define the values of other properties.

**Example**

```
<propertylist>
    <property name="projectname" value="myproject"/>
    <property name="projectdir" value="c:\myprojects"/>
</propertylist>
```

There is a built-in external property called `configdir`, which reflects the directory in which the project file resides. For a new project that has not been saved, `configdir` reflects DashO's installation directory.

Properties are useful for creating project files that act as templates for multiple projects, for different versions of the same project, and for simple portability across different build environments.

## Property References

A property is referenced with the following syntax:

**Example**

```
${property_name}
```

Property references are case sensitive, therefore, `${MyProjectDir}` references a different property than does `${myprojectdir}`. If you reference a property that has not been defined, its literal value is used. Properties may be defined in the terms of other properties. The value of a property may be specified using one or more property references including to references to environment variables. These property references can include default values, indirection, or substitution syntax. Recursive variable definition is not allowed.

DashO provides many flexible ways to reference properties:

| | |
|---|---|
| **${prop}** | Simple replacement. If the value for prop is undefined or is blank, then no replacement takes place and `${prop}` is left unchanged. |
| **${prop:-default}** | Replacement with default value. If prop is defined and not blank, use its value. Otherwise, use default as the value. |
| **${prop:+value}** | Replace when defined. If prop is defined and not blank, then value is used. Otherwise a blank string is substituted when prop is defined. |
| **${prop:?message}** | Generate error if not set. If prop is defined and not blank then its value is used. Otherwise an error with the text of message is generated and the build ends. |
| **${prop/pattern/replace}** | Replacement after pattern substitution. Replaces the first occurrence of the regular expression *pattern* with the replacement text *replace*. If replace is blank, then the matching text is deleted. |
| **${prop//pattern/replace}** | Replacement after pattern substitution. Replaces all occurrences of the regular expression *pattern* with the replacement text *replace*. If replace is blank, then the matching text is deleted. |
| **${prop/#pattern/replace}** | Replacement after pattern substitution. Replaces the leading regular expression *pattern* with the replacement text *replace*. If replace is blank then the matching text is deleted. |
| **${prop/%pattern/replace}** | Replacement after pattern substitution. Replaces the trailing regular expression *pattern* with the replacement text *replace*. If replace is blank then the matching text is deleted. |
| **${prop#pattern}** | Replacement after pattern deletion. Deletes the leading regular expression *pattern*. |
| **${prop%pattern}** | Replacement after pattern deletion. Delete the trailing regular expression *pattern*. |
| **${!prop}** | Indirect replacement. If prop is defined and not blank, then its value is used as a property name. The value of this property is then used as the replacement value. You can use indirect placement followed by any of the previously described references. |

**Note**

You can use `${prop:-}` to substitute an empty string when prop is undefined.

## Dynamic Properties

In some places of the project file you can use dynamic properties whose values contain information about the class or method that is being processed.

- `CLASS_NAME` - the full name of the current class, including its package name.
- `CLASS_SIMPLENAME` - the simple name of the class, i.e. the class name without its package name.
- `CLASS_PACKAGE` - the package name of the class, including a trailing period. This will be an empty string for classes in the default package: use `${CLASS_PACKAGE:-}` to ensure that the property will be expanded.
- `METHOD_NAME` - the name of the current method. For constructors this is the same as `CLASS_SIMPLENAME`.
- `PROP_NAME` - if the method is a setter or getter the name of the related property. For constructors this is the same as `CLASS_SIMPLENAME`.

## Timestamp property

DashO provides the tstamp property to allow the insertion of information about the current date and/or time. The tstamp property can be used in two different ways:

- `${tstamp}` insert the date information using the default format for the locale.
- `${tstamp[`*`pattern`*`]}` inserts the date information using a format specification. The *`pattern`* is the same as used by Java's `SimpleDateFormat` class.

## Property Precedence

You can reference properties defined in your project file, values from Java's system properties, or from the environment. To resolve the value for the property DashO consults the sources in the following order:

- Java system properties
- Environment properties
- Project file properties

In this way you can override properties defined in the project file using the Java command line's `-D` option or via Ant.

Properties may be used with the following tags:

- `<entrypoints>`/`<applet>`'s `name` attribute
- `<entrypoints>`/`<ejb>`'s `name` attribute
- `<entrypoints>`/`<iappli>`'s `name` attribute
- `<entrypoints>`/`<library>`/`<jar>` and `<entrypoints>`/`<library>`/`<dir>`'s `path` attribute
- `<entrypoints>`/`<midlet>`'s `name` attribute
- `<entrypoints>`/`<publics>`'s `name` attribute
- `<entrypoints>`/`<quickjar>`'s `path` attribute
- `<entrypoints>/<servlet>`'s `name` attribute
- `<entrypoints>`/`<unconditional>`'s `name` attribute
- `<global>`/`<exclude>`'s `classname` attribute
- `<includenonclassfiles>`/`<copy>`'s `source` and `relativedest` attributes
- `<inputpath>`/`<pathelement>` and `<classpath>`/`<pathelement>`'s `location` attribute
- `<mapping>`/`<mapinput>`'s `suffix` attribute

- <mapping>/<mapinput>'s path attribute
- <mapping>/<mapoutput>'s path attribute
- <mapping>/<mapreport>'s path attribute
- <output>/<dir>'s path attribute
- <output>/<jar>'s path and manifest attributes
- <output>/<constpooltag>'s value
- <output>/<sourcefile>'s value
- <premark>/<passphrase>'s value
- <premark>/<watermark>'s value
- <preverifier>'s value
- <rename>/<class-options>'s prefix attribute
- <report>'s path attribute
- <expiry>'s period, warningperiod, date, and warningdate attributes
- <expiry>/<property> name and value attributes

# <global> Section

The optional global section is for defining options that apply across the entire run. This section contains the global options and the global excludes.

**Note**

Global options are not case sensitive.

## Fornamedetection Global Option

The `fornamedetection` option turns on DashO's built-in ability to search for dynamically included classes. This adds significant processing time to the run. It is best to run your application with this on initially and then add these classes as entry points to your file.

In some cases, it is not possible for DashO to determine which classes are dynamically loaded. A program could request the name of the class to be loaded as user-input or to be specified in an external file. However, most inclusions are not that vague and DashO can safely determine what they are. The report file reports a confidence level associated with a given class inclusion discovery. A HIGH confidence level is almost assuredly correct. In other words, DashO detected something such as:

**Example**

```
Class a = Class.forName("java.awt.Rectangle");
```

A POSSIBLE confidence level is an educated guess. DashO has detected code similar to:

**Example**

```
String s = getUnknownString();
Class a = Class.forName(s);
Rectangle r = (Rectangle)a.newInstance();
```

In this case, DashO cannot detect which exact class is loaded. However, it does "know" that the loaded class will be cast to a Rectangle. Therefore, DashO finds all subclasses of Rectangle and includes them with a possible confidence level.

Using the `fornamedetection` option and running in force mode instructs DashO to automatically include what it finds. Only do this if you are confident in DashO's determination.

Running in force mode does not stop even if it cannot determine dynamically loaded classes for any given `forName()` call.

**Example**

```
<global>
    <option>fornamedetection</option>
</global>
```

## Ignorenotfoundclasses Global Option

The `ignorenotfoundclasses` option allows DashO to process an application even if it encounters references to classes that are not present in the classpath. DashO cannot ignore all missing classes: if it cannot find a super class or super interface it cannot continue.

This option should only be used as a means to allow DashO to finish so that information from the run can be gathered. Without access to all classes, DashO cannot safely determine all needed dependencies.

**Example**

```
<global>
    <option>ignorenotfoundclasses</option>
</global>
```

## Ignorebrokencalls Global Option

The `ignorebrokencalls` option allows DashO to process an application even if it encounters references to methods in classes that it cannot find. This can be caused by errors in your classpath or by jars that are out of date. Although DashO will be able to process the classes, you will want to check the classpath and jars to make sure they contain the classes you expect.

**Example**

```
<global>
    <option>ignorebrokencalls</option>
</global>
```

## Force Option

When DashO detects the use of reflection in classes, it makes note of the location and target of the reflective code and continues its analysis. At the end of the analysis, it prints a reflection report and halts the build process. Once you have dealt with all of the reflection issues in your project, the force option can be added to let DashO complete the build. The force option can be specified in the project file or passed to DashO via the command line `--force` option: the former is preferred.

**Example**

```
<global>
    <option>force</option>
</global>
```

## Makepublic and Nomakepublic Global Options

By default, DashO changes the access modification of all classes, non-private methods, and non-private fields to public before writing them to disk. This has several ramifications:

- This solves the problem of classes that change package membership and contain default methods.
- In general, this is not dangerous since DashO does not induce method calls into your program. After all, the compiler enforced the access restrictions at compile time.
- Feasibly, dynamic linking of public methods should be faster than that of more restricted access levels. Primarily because public has no restrictions there is no need for the runtime to verify equivalent package or class membership.

To *stop* this behavior, use the `nomakepublic` option; to *force* this behavior, use the `makepublic` option; to use the *default* behavior, that is to let DashO decide what to do, do not include either option.

**Note**

Using `nomakepublic` may cause access errors with protected/default methods. That is, a class that was in a given package may now be in a new renamed package. However, it may still access non-public classes from the original package causing an access exception.

DashO's default behavior generally avoids this problem and has been shown to be safe for most applications.

**Example**

```
<global>
    <option>nomakepublic</option>
</global>
```

## Renameforname Global Option

The `renameforname` option allows dynamically loaded classes to be renamed. In the case where DashO cannot unambiguously determine the string used to load a class, that class should be listed in the <entrypoints> section. These ambiguous cases correspond to what the `fornamedetection` option would report as a *possible* confidence level.

**Note**

Using assertions in your code with Java's assert keyword makes a class self-reflective. DashO will make that class non-renameable unless you use the `renameforname` option.

**Example**

```
<global>
    <option>renameforname</option>
</global>
```

## Global <exclude>

The exclude option allows you to specify classes or methods that appear as part of the input classes but should not be included in the final output of DashO. The classes matching the regular expression of a global exclude will not be processed or included in the final output.

For example, you could exclude tests or samples present in third party jars:

**Example**

```
<global>
    <exclude classname="com\.thirdparty\.tests\..*"/>
    <exclude classname="com\.thirdparty\.sample\..*"/>
</global>
```

The names of excluded classes are always specified as regular expressions.

# <inputpath> Section

The `<inputpath>` section contains the location of the classes that DashO will process. DashO can handle directories, zip files, and jar files.

**Example**

```
<inputpath>
    <pathelement location="c:\test\app.jar"/>
    <pathelement location="c:\test\classes"/>
</inputpath>
```

**Note**

`c:\test\classes` is not to be any part of a package designation. It is to be the directory where the packages are stored.

# <classpath> Section

The <classpath> section contains the location of classes that DashO may need in analyzing the input classes. These classes are typically third-party packages or jars that are part of the Java Runtime Environment. DashO can handle directories, zip files, and jar files in the classpath.

**Example**

```
<classpath>
    <pathelement location="c:\test\app.jar"/>
    <pathelement location="c:\test\classes"/>
</classpath>
```

**Note**

c:\test\classes is not to be any part of a package designation. It is to be the directory where the packages are stored.

## Systemclasspath attribute

Optionally you can append or prepend the system classpath[4] in addition to the directories and jars specified in the <classpath> section. The following option appends the system classpath to the list of the directories and jars specified in the <classpath> section.

**Example**

```
<classpath systemclasspath="append">
    <pathelement location="c:\test\classes"/>
    …
</classpath>
```

## Appendrtjar attribute

By default DashO adds the runtime jar of the version of Java it is currently using. If you need to use a different version you can control this behavior.

**Example**

```
<classpath appendrtjar="false">
    <pathelement location="c:\Java\jre1.5.0_12\lib\rt.jar"/>
</classpath>
```

**Note**

Projects that use J2ME or the Android API must use this option. These projects require the runtime jar for these particular environments: e.g. midpapi10.jar or Android.jar.

[4] The system classpath is composed of the values in the sun.boot.class.path property and the CLASSPATH environment variable where available.

# <entrypoints> Section

Entry points are starting points for the dependency analysis that DashO performs in order to determine which classes, methods, and fields are used by your application. In other words, these are the entry points for your application.

Entry points are analyzed by DashO to determine which classes, methods, and fields are required for classes to function. For example, all methods called by your entry points, and subsequent methods called by those methods, are required by DashO. That is, if you tell DashO a specific main method is required, then all the methods that main method calls are required as well.

## <quickjar> Entry Point

The Quick Jar entry point may be used to obfuscate a program or an API library encapsulated in a JAR file.

In order for this option to work for an application, the manifest of the JAR file must contain a line of the form `Main-Class: classname`. Here, `classname` identifies the class having the `public static void main(String[] args)` method that serves as your application's starting point.

DashO uses this information in the manifest to do the static dependency analysis. If the manifest does not have `Main-Class` information, DashO processes the jar as a library. DashO automatically uses all the public methods in the jar as entry points.

Assume the input jar has a manifest file with a `Main-Class` entry:

**Example**

```
Main-Class: test.MyApplication
```

In this case, the main method of the class `test.MyApplication` will be considered as entry point.

- You may specify multiple Quick Jar entry points; however, you cannot mix Quick Jar entry points with any other kind of entry points.
- The pruning or removal feature of DashO is shut-off in quick jar mode and the values put in the <removal> section are ignored.
- If you specify multiple Quick Jars as entry points, then DashO writes the obfuscated classes to only a single output jar or directory.
- All the non-class files in the input jar are automatically included in the output.
- Control Flow, Optimization, and String Encryption includes and excludes are *ignored*.

**Example**

```
<entrypoints>
    <quickjar path="c:\myapp.jar"/>
</classpath>
```

## <method> and <field> Entry Points

Entry point methods indicate where your application starts. If you were looking at a code listing and someone asked what that code did, where would you look first? You would probably check if the code contained a main method. That main is the entry point method for that application. In general, you use the main of a given class:

**Example**

```
<entrypoints>
    <classes name="test.MyApplication">
        <method name="main" signature="java.lang.String[]"/>
    </classes>
</entrypoints>
```

When specifying a signature for methods use fully qualified class names: as in the above example, use `java.lang.String[]` rather than `String[].` Multiple parameters should be separated by commas without spaces. You can also specify constructors as entry points using the special `<init>` notation as the method name. Remember to use `<` and `>` since the project file is in XML.

Names of classes, members and method signatures can be specified as literals, patterns, or regular expressions. See Names: Literals, Patterns, and Regular Expressions for details.

### Rename Attribute

By default entry points are not renameable since they are referenced by some outside mechanism. In some cases entry points can be renamed. For example, DashO will update the `Main-Class` attribute of a jars manifest with the renamed class. In this case the class can be renamed, but the main method cannot. The `<classes>`, `<method>` and `<field>` tag all have a `rename` attribute to control the renameability of the item.

**Example**

```
<entrypoints>
    <classes name="test.OtherApplication" rename="true">
        <method name="main" signature="java.lang.String[]"/>
    </classes>
</entrypoints>
```

## <publics> Entry Point

Like the `<classes>` entry point, the `name` attribute can be a literal, a pattern, or a regular expression. At times, you may have a class that is an interface to your application. It may have many methods that are entry points into your application. If you wish to specify all the public methods of a given class as entry points, you can specify each individually or use the `<publics>` tag.

**Example**

```
<entrypoints>
    <publics name="test.MyApplet"/>
</entrypoints>
```

The classes and members specified by the `<publics>` can also be made renameable using the `rename-class` and `rename-members` attributes: both attributes are optional and default to `false`.

**Example**

```
<entrypoints>
    <publics name="com.yoyodyne.**Bean" rename-class="true"
      rename-members="false"/>
</entrypoints>
```

## <library> Entry Point

For API libraries, you can specify all public and protected methods of all classes in a directory or jar by using the library option.

**Example**

```
<entrypoints>
    <library public="off">
        <dir path="myAPIDirectory"/>
    </library>
</entrypoints>
```

**Note**

If you add a directory or jar as a library it does not need to be added to the <classpath>: DashO does this automatically.

```
<entrypoints>
    <library public="off">
        <jar path="myAPI.jar"/>
    </library>
</entrypoints>
```

The value of public can be `on` or `off`. If omitted, `on` is assumed.

All public methods of all classes found through a recursive descent of all directories below `myAPIDirectory` will be used as Triggers. The `myAPIDirectory` should not be part of a package designation. It is to be the directory where the packages live: it would be the same directory you would put on the classpath. Using a jar is straightforward - all classes found in the jar are used.

If you would like only public methods to be used as entry points, you may use the `library` tag with `public` set to `on`.

## <classes> Entry Point

It is also possible to include classes without making assumptions about which methods are to be included. Manually specifying classes here is vital for some applications to package correctly. If you use the `Class.forName()` construct in your application, DashO cannot determine all possible classes that are needed. In this case, DashO informs you of the locations of the `forName(s)` and exits. You must then manually enter those classes here and re-run DashO with the force option. Any classes specified here are open to method and/or field removal.

**Example**

```
<entrypoints>
    <classes name="com.yoyodyne.Application"/>
</entrypoints>
```

This would cause DashO to load `com.yoyodyne.Application` but not use any methods or fields as entry points. If that class overrides system methods or other user-created methods, then its methods are included too. Classes entered this way are renameable if the global `renameforname` option is turned on.

## <unconditional> Entry Point

At times, you need a class to be included, even if it is not explicitly referenced by other classes. This is done through the unconditional entry points. In this case all members of the class are used as entry points and will appear in the output. Like other entry points, the name may be a literal, a pattern or a regular expression. By default the class and members are not renameable, but they can be made renameable by using the `rename-class` and `rename-members` attributes.

## Entry Points for Special Applications

DashO has several distinct types of Java application configurations built-in. For convenience, DashO defines special syntax to specify the entry points for some of these applications. Names can be specified as literals, patterns or as regular expressions: See Names: Literals, Patterns, and Regular Expressions. By default the class and members are not renameable. All types use the `rename-class` attribute and some support rename-members.

### <applets>

An applet's `init()` and `paint()` methods, among others, are automatically included. Any methods that an applet overrides automatically become entry points. However, you do need to specify the full format designation for what applet class is the entry point class.

**Example**

```
<entrypoints>
    <applet name="test.MyApplet"/>
</entrypoints>
```

The `<applets>` tag uses the `rename-class` attribute, but not the rename-members: the entry point members are defined by the interface and are not renameable. Other methods in the class are renameable.

### <servlets>

You may specify entry points for servlets as:

**Example**

```
<entrypoints>
    <servlet name="test.MyServlet"/>
</entrypoints>
```

The `<servlets>` tag uses the `rename-class` attribute, but not the rename-members: the entry point members are defined by the base classes and are not renameable. Other methods in the class are renameable.

### <ejb> Enterprise Java Beans

Enterprise JavaBeans have their own notation to designate classes related to a given EJB:

**Example**

```
<entrypoints>
    <ejb name="MyEntityBean"/>
    <ejb name="MyRemoteInterface"/>
    <ejb name="MyHomeInterface"/>
    <ejb name="MySessionBean"/>
    <ejb name="MyPrimaryKey"/>
</entrypoints>
```

The `<ejb>` tag uses both the `rename-class` and rename-members attributes.

**Note**

Alternatively, you can use the `<publics>` notation with the EJBs to specify their entry points.

## <midlet> and <iappli>

DashO provides explicit support for applications written to the Mobile Interconnected Device Profile (MIDP) specification (midlets). Your midlet class can be a subclass of `javax.microedition.midlet.Midlet` at any level of descendancy.

**Example**

```
<entrypoints>
    <midlet name="test.MyMidlet"/>
</entrypoints>
```

Similarly, DashO explicitly supports applications written for NTT DoCoMo's iAppli framework. Your iAppli class can be a subclass of `com.nttdocomo.ui.IApplication` at any level of descendancy.

**Example**

```
<entrypoints>
    <iappli name="test.MyIappli"/>
</entrypoints>
```

Both tags use the `rename-class` attribute, but not the rename-members: the entry point members are defined by the base classes and are not renameable. Other methods in the class are renameable.

### *Configuration for Midlet Projects*

Do not include the Java Runtime Jar. See the appendrtjar attribute section. Instead setup an environment variable or property that points to your installation of the Wireless Toolkit from Sun.

**Note**

If you have an environment variable called WTK_HOME setup you can use it directly or you can create a DashO property to use its setting.

**Example**

```
<propertylist>
    <!-- To force the environment variable to be set -->
    <property name="wtk.home" value="${WTK_HOME:?WTK_HOME not defined}"/>
    <!-- To use a default if not set -->
    <property name="wtk.home" value="${WTK_HOME:-C:\WTK2.5.1}"/>
</propertylist>
```

Add the `cldcapi` and `midpapi` jars to the classpath. These jars are part of the Wireless Toolkit from Sun.

**Example**

```
<classpath>
    <jar path="${wtk.home}/lib/cldcapi10.jar"/>
    <jar path="${wtk.home}/lib/midpapi10.jar"/>
</classpath>
```

Set up preverification for the project.

**Example**

```
<preverifier run="true">
    ${wtk.home}/bin/preverify.exe
</preverifier>
```

## <report> Section

If a report file is specified, DashO creates a report indicating all methods and fields removed. It also summarizes the total numbers for the entire project including total method, field, and constant pool entry removals.

**Example**

```
<report path="c:\output\dasho-report.txt"/>
```

Since there is no removal in Quick Jar mode, there is also no report file produced. Warnings and errors go to the console.

A snippet from a report looks like:

**Example**

```
Removal Option : Remove all unused

Dependency Report for Entry Points:
GifWiz.Editor.main(java.lang.String[])
--------------------------------------------------------------------------

gifwiz.ConsoleMessage
==========================================================================
  Removable Method display()
  Removable Method outline(int)
  Removable Field n
  Removable Field z1

gifWiz.Arc
==========================================================================
  Removable Method round(double)
  Removable Method update_scp()
  Removable Method update_scp(int)
  Removable Method corners()
  Removable Field ccw
  Removable Field cw
  Removable Field dalpha21
  Removable Field dalpha10
```

Each Removable method was determined by DashO to be unneeded during the execution of the program.

DashO also outputs summary results for the run:

**Example**

```
Statistics      In     Out  Change
==========================================================================
Classes        612     596   -2.6%
Methods       8975    7095  -20.9%
Fields        4953    2792  -43.6%
Constants   103306   90756  -21.9%

Processing Time: 4:46.977 min
```

This DashO run was able to remove almost 21% of all methods. However, this does not mean the application size was reduced by 21%. The percentage of methods removed may be only 1% of the application size.

## <output> Section

This option indicates whether you want DashO to write the output to a directory or a jar file. The format of the output is dependent upon your renaming options. It also controls whether the results are merged into a single output or retain the same packaging as the input. If you specify no renaming, then the directory/package structure that currently exists will be recreated in the specified directory so be sure your destination is not the same as your source! If you rename, notions of packages can be removed and all classes will be put in the directory specified.

**Example**

```
<output merge="true">
    <dir path="c:\output\"/>
</output>
```

Optionally you can specify a manifest with the jar output. DashO will copy the manifest file to the output jar. If you specify a jar file for the manifest, it will be used as the source of the manifest.

**Example**

```
<output merge="true">
    <jar path="c:\output\dashoed.jar" manifest="c:\misc\Manifest.mf"/>
</output>
```

**Note**

Both `path` and `manifest` attributes support properties.

## Jar Attributes

When DashO creates one or more jars, either by using the <jar> tag or when merge=false, you can specify attributes that customize the jar creation:

- `compress="boolean"` – Determines if the entries in the jars be compressed. Defaults to `true`.
- `level="0-9"` – The compression level for jar entries. Defaults to `6`. Higher values give higher compression.
- `filesonly="boolean"` – Determines if the jars contain only file entries or both field and directory entries. Defaults to `true.`

**Example**

```
<output merge="false" compress="true" level="4" filesonly="false">
    <dir path="c:\output\"/>
</output>
```

This sample would produce jars with a moderate level of compression that contained entries for both the files and their directory structure.

## Merge Attribute

DashO can combine the obfuscated results into a single directory or jar or keep the original packaging of the input classes. This behavior is controlled using the `<output>` tag's `merge` attribute. The values for the `merge` attribute are either true or false. If the `merge` attribute is not provided it defaults to true.

### merge="true"

This is the default mode for DashO. When `merge="true"` either a `<dir path="…"/>` or `<jar path="…"/>` may be used for output. DashO will combine all obfuscated classes into the indicated jar or output directory.

### merge="false"

When `merge="false"` is specified only a `<dir path="…"/>` may be used for output. DashO will preserve the original packaging of the input classes in the output directory. Classes that came from jars will be placed in identically named jars in the output directory. In addition, the manifest and non-class files from the jars will be copied to the obfuscated jars. Classes that came from directories will be placed in subdirectories in the output directory. DashO will try to preserve relative paths between jars and directories that come from a common root location.

**Note**

The `merge="false"` option requires that a `<dir path="…"/>` tag. If a `<jar path="…"/>` tag is provided then the `merge="false"` setting is ignored. In Quick Jar mode the merging always takes place.

## Autocopy Attribute

When `merge="false"` is specified you can also specify the `autocopy` attribute. When `autocopy="true"` is specified non-class files in input jars input are automatically copied to their respective output. Non-class files that appear in input directories are *never* copied.

**Example**

```
<output merge="false" autocopy="true">
    <dir path="obfuscated"/>
</output>
```

## <constpooltag>

You can covertly add constant pool entries for your class files to mark them. This string will be placed in every class DashO outputs and will not be printed or evident to the casual user. Only those using a class disassembly tool will be able to view this string. The value attribute can contain property references including dynamic class properties.

**Example**

```
<output>
    <constpooltag value="Copyright 1984 Yoyodyne Engineering, Inc."/>
</output>
```

## <sourcefile>

This tag allows you to set the value of Java's `SourceFile` attribute that is used in stack traces. The value attribute can contain property references including dynamic class properties.

**Example**

```
<output>
    <sourcefile value="${CLASS_SIMPLENAME}-v${ver}"/>
</output>
```

# <removal> Section

The removal option allows you to specify what level of granularity you want for class, method and/or field removal, and metadata removal. For class and member removal there are two attributes on the tag `classes` and `members`. The options for these are:

- `none` - no removal
- `unused-non-public` - only remove unused items that are not public
- `unused` - remove all unused items

If both attributes are omitted or you do not specify `<removal>,` removal will not occur.

If you are packaging a true application - not something that's sub-classed or called by other classes - then the `unused` option is the best choice.

Pursuant to the license agreements you have with the third-party API libraries you use, it is best practice to allow DashO to include all classes your application needs. That way the resulting output would be one jar or directory that contains every class your application needs, tailored specifically to how your application uses it.

**Example**

```
<removal classes="unused-non-public" members="unused"/>
```

**Note**

Removal is off in Quick Jar mode. Quick Jar mode ignores all the options set in the removal section.

## <debug> Section

This section instructs DashO to remove debug information inserted into the class files by a compiler. The `type` attribute is used to specify the types of information to be removed. The following types can be removed:

- `SourceFile` - The name of the source file from which the class was compiled.
- `SourceDirectory` - The location of the source file from which the class was compiled.
- `SourceDebugExtension` - A tool specific string that is interpreted as extended debugging information.
- `LineNumberTable` - Maps byte codes to a given line number in the original source file. Used by debuggers and stack traces.
- `LocalVariables` - Used by debuggers to determine the name and type of local variables during the execution of a method.
- `LocalVariableTypes` - Signatures for local variables that use generics.

Multiple items are separated by spaces.

If the `<debug>` section is not present then all debugging information is retained. If it is present but the `type` attribute is not present then all debugging information is removed.

**Examples**

```
<debug/>

<debug types="SourceDirectory SourceDebugExtension" />
```

**Note**

The use of some obfuscation transforms requires the removal of local variable information even if the <debug> section does not request their removal.

## <attributes> Section

The compiler stores additional metadata in attributes inside the class file. DashO lets you determine the disposition of these attributes individually. The `types` attribute of the tag is used to specify the type of attributes to be removed. The following types can be removed:

- `Exceptions` - Indicates which checked exceptions a method may throw.
- `Signature` - Indicates generic types, method declarations with generics, and parameterized types.
- `Deprecated` - Indicates that the class, interface, method, or field has been superseded.
- `Synthetic` - Indicates a class member that does not appear in the source code.
- `EnclosingMethod` - Indicates the enclosing method for a local or anonymous class.
- `RuntimeVisibleAnnotations` - Holds the annotations on a class, method, or field that are visible with reflection.
- `RuntimeInvisibleAnnotations` - Holds the annotations on a class, method, or field that are *not* visible with reflection.
- `RuntimeVisibleParameterAnnotations` - Holds the annotations on the parameters to a method that are visible with reflection.
- `RuntimeInvisibleParameterAnnotations` - Holds the annotations on the parameters to a method that are *not* visible with reflection.
- `AnnotationDefault` - Default values for annotation elements.
- `InnerClasses` - Indicates relationships between inner and outer classes.
- `Unknown` - All other attribute types.

Multiple items are separated by spaces.

If the `<attributes>` section is not present then all attributes are retained. If it is present but the `type` attribute is not present then the following attributes are removed: `Deprecated`; `Synthetic`; `RuntimeInvisibleAnnotations`; `RuntimeInvisibleParameterAnnotations`.

**Examples**

```
<attributes/>

<attributes types="Deprecated Synthetic" />
```

# <renaming> Section

DashO can rename classes, methods, and fields to short meaningless names. This is significant in class size reduction and as an obfuscation technique. Subsequent sections allow you to exclude given classes and members from being renamed.

**Note**

See the Advanced Topics regarding DashO's renaming algorithm and its ramifications.

This tag allows for the global control of renaming using the option attribute. Valid values are `on` and `off`.

**Example**

```
<renaming option="on"/>
```

## <class-options> Section

The attributes of this tag controls the renaming of classes.

- `rename` – This `boolean` option turns the renaming of classes on or off. When `false` then classes will retain their original names.
- `keeppackages` – This `boolean` option allows you to rename the classes itself while keeping the original package names and hierarchy.
- `randomize` – The new names for classes can be assigned in either a sequential or random order. When this option is `true` identifiers are assigned in a random order. Two character identifiers are assigned after all single character identifiers have been used, and so on.
- `prefix` – This attribute specifies a prefix that is added to all renamed classes. If it contains a period then the class is effectively placed in a new package.

### prefix attribute

The prefix is appended to all renamed classes. By defining a `prefix` that contains a period the renamed classes can be placed in a custom package.

**Example**

```
<renaming option="on"/>
    <class-options prefix="pkg.X_"/>
</renaming>
```

The following table shows the renaming possibilities using a prefix:

| Prefix | New Name |
|---|---|
| C | Ca |
| pkg. | pkg.a |
| pkg.X_ | pkg.X_a |

### keeppackage attribute

When this option is true the name of the class is changed but the package portion of its name remains unchanged.

**Example**

```
<renaming option="on"/>
    <class-options keeppackages="true"/>
</renaming>
```

An example of this type of renaming is:

| Original Name | New Name |
|---|---|
| yoyodyne.application.Main | yoyodyne.application.a |
| yoyodyne.application.LoadData | yoyodyne.application.b |
| yoyodyne.tools.BinaryTree | yoyodyne.tools.c |
| yoyodyne.tools.LinkedList | yoyodyne.tools.d |

When used with a `prefix` the original package name appears before the portion added by the `prefix`.

**Example**

```
<renaming option="on"/>
    <class-options keeppackages="true" prefix="x_"/>
</renaming>
```

This would result in:

| Original Name | New Name |
|---|---|
| yoyodyne.application.Main | yoyodyne.application.x_a |
| yoyodyne.application.LoadData | yoyodyne.application.x_b |
| yoyodyne.tools.BinaryTree | yoyodyne.tools.sub.x_c |
| yoyodyne.tools.LinkedList | yoyodyne.tools.sub.x_d |

## <member-options> Section

This section controls the renaming of methods and fields.

- `keeppublics` – When set to true all public methods and fields will retain their original names. Usage of the `library` option in the <entrypoints> section treats all public methods as entry points inherently retaining their original names. Specifying this option would be redundant.
- `randomize` – The new names for members can be assigned in either a sequential or random order. When this option is `true` identifiers are assigned in a random order. Two character identifiers are assigned after all single character identifiers have been used, and so on.

WTF The `onlynonpublics` option is suitable for API libraries or other cases where you can accept renaming of access-controlled methods and/or fields but not public ones. However, usage of the `library` option in the <entrypoints> section treats all public methods as entry points inherently making them immune from renaming. Specifying them here is redundant.

## <renaming> Exclude List

This section provides a dynamic way to fine tune the renaming of the input class files. It can contain a list of exclude rules that are applied at runtime. If a rule selects a given class, method, or field, then that item is not renamed.

**Note**

These rules are applied in addition to renaming restrictions defined by entry points.

The rules are logically `OR`-ed together: any item selected by at least one rule is not renamed. The `<excludelist>` has support for excluding names by class, method, and field.

**Example**

```
<renaming option="on">
    <excludelist>
        <classes name="samples.SimpleApp" excludeclass="true"/>
    </excludelist>
</renaming>
```

## <mapreport> Section

DashO can produce a report of all the renaming it has performed as well as statistics about the renamed results. This is created using the nested `<mapreport>` tag.

**Example**

```
<renaming option="on">
    <mapping>
        <mapreport path="c:\workproject-mapreport.txt"/>
    </mapping>
</renaming>
```

**Note**

The `path` attributes support properties.

An example of the listing is:

**Example**

```
one.A (b)
================================================================================
  a   pub1(int)
  b   def1(int)
  c   pub2(int)

two.B (c)
================================================================================
  a   pub1(int)
  b   pub2(int)
  c   def1(int)
```

The new names of the classes and methods are shown. Bug tracking becomes difficult after renaming, especially with a high incidence of method overloading, making the map file essential. The map file also provides statistics regarding the success of overload-induction:

**Example**

```
Number of Methods : 7095
  Renamed to 'a' : 2031 (28.6%)
  Renamed to 'b' :  786 (11.0%)
  Renamed to 'c' :  484 (6.8%)
  Renamed to 'd' :  327 (4.6%)
  Renamed to 'e' :  230 (3.2%)
  Renamed to 'f' :  169 (2.4%)
  Renamed to 'g' :  131 (1.8%)
  Renamed to 'h' :  120 (1.7%)
  Renamed to 'i' :  106 (1.5%)
```

These statistics represent the total number of methods that were renamed to each given name.

## <mapoutput> Section

Specifying the `<mapoutput>` file option instructs DashO's renamer to keep track of how things were renamed for both your immediate review and to be used as input in a future DashO run. This option creates a file that is used in the map input file to do incremental renaming and decode obfuscated stack traces.

Accidental loss of this file can destroy your chances of incrementally updating your application in the future. Therefore, proper backup of this file is crucial. For this reason, DashO does not automatically overwrite this file if an existing one is found.

The attribute `overwrite="true"` instructs DashO to allow overwriting an existing file.

**Note**

The `overwrite` attribute is optional and if omitted, it defaults to `false`.

**Example**

```
<renaming option="on">
    <mapping>
        <mapoutput path="c:\work\project.map" overwrite="true"/>
    </mapping>
</renaming>
```

## <mapinput> Section

A file created from the `<mapinput>` option can be used in the incremental input file option.

**Example**

```
<renaming option="on">
    <mapping>
        <mapinput path="c:\work\project.map"/>
    </mapping>
</renaming>
```

## Suffix Attribute

The `mapinput` has an optional suffix option that can be used to immediately track changes across incremental obfuscations (*i.e.*, the suffix could be the date or some other identifying string).

**Example**

```
<renaming option="on">
    <mapping>
        <mapinput suffix="new">
            <file path="c:\work\project.map"/>
        </mapinput>
    </mapping>
</renaming>
```

# <optimization> Section

The optimization section allows you to specify options that are specific to byte code optimization including fine-grained rules for including and excluding items. When the `option` attribute is set to `off`, DashO skips optimization altogether, regardless of what is in the rest of the section.

**Example**

```
<optimization option="on"/>
```

To fine tune where optimization takes paces the `<optimization>` section can contain both a `<includelist>` and `<excludelist>` which contain rules that select classes and methods. These are explained in the section on <includelist> and <excludelist> Rules.

**Example**

```
<optimization option="on">
    <includelist>
        <classes name="samples.**"/>
    </includelist>
    <excludelist>
        <classes name="samples.SimpleApp"/>
    </excludelist>
</optimization>
```

**Note**

Quick Jar mode ignores all includes and excludes in the `<optimization>` section.

# <controlflow> Section

The control flow section allows the user to specify options that are specific to control flow obfuscation including fine-grained rules for including and excluding items. When the `option` attribute is set to `off`, DashO skips control flow obfuscation altogether, regardless of what is in the rest of the section.

**Example**

```
<controlflow option="on"/>
```

Control flow obfuscation adds an extra level of protection for your Java code but at times, this transformation is drastic and can affect performance. To fine tune where control flow obfuscation is performed the `<controlflow>` tag can contain both a `<includelist>` and `<excludelist>` which contain rules that select classes and methods. These are explained in the section on <includelist> and <excludelist> Rules.

**Example**

```
<controlflow option="on">
    <excludelist>
        <classes name="SimpleApp"/>
    </excludelist>
</controlflow>
```

**Note**

Quick Jar mode ignores all includes and excludes in the `<controlflow>` section.

## <stringencrypt> Section

The string encryption section allows the user to specify options that are specific to string encryption obfuscation including fine-grained rules for including and excluding items. When the `option` attribute is set to `off`, DashO skips string encryption altogether, regardless of what is in the rest of the section.

**Example**

```
<stringencrypt option="on"/>
```

String encryption hinders examination of your code by making it more difficult to use simple string searches to locate critical parts of your program but decrypting the strings at runtime does add some performance overhead. To fine tune where strings are encrypted the `<stringencrypt>` tag can contain both a `<includelist>` and `<excludelist>` which contain rules that select classes and methods. These are explained in the section on <includelist> and <excludelist> Rules.

**Example**

```
<stringencrypt option="on">
    <includelist>
        <classes name="com.yoyodyne.**"/>
    </includelist>
    <excludelist>
        <classes name="com.yoyodyne.ui.**"/>
    </excludelist>
</stringencrypt>
```

**Note**

Quick Jar mode ignores all includes and excludes in the `<stringencrypt>` section.

## <decrypter> Section

This section lets you control where DashO will place the method that is used to decrypt the strings at runtime. This tag is similar to the `<classes>` tag used in <includelist> and <excludelist> Rules. It has three attributes that select the class where the decrypter method can be placed:

- `name` – The name of the class where the method can be placed. This can be the name of the class, a pattern that selects the class, or a regular expression.
- `regex` – Determines the interpretation of the name attribute. If true then name is a regular expression.
- `modifiers` – the modifiers used to select the class where the method can be placed. See Modifiers attribute for details.

If the `<decrypter>` section is omitted then DashO will determine the location automatically.

**Example**

```
<decrypter modifiers="static class" name="com.yoyodyne.**"/>
```

# <premark> Section

This section explains how to specify options that are specific to software watermarking. If the option set to `off`, DashO skips PreMark altogether, regardless of what's in the rest of the `premark` section. When it is `on`, DashO watermarks your application using the specified encoding and watermark string.

**Example**

```
<premark option="on"/>
```

## Truncate Attribute

It is not possible for DashO to predict the maximum watermark string length until the output jar has been generated. You can tell DashO what to do during a build when your watermark string will not fit in the output jar. The default setting stops the build with an error message. When set to `on` DashO truncates the string so it fits and prints a warning message. In both cases, the message will indicate the maximum watermark size.

**Example**

```
<premark truncate="on" option="on"/>
```

## <encoding>

DashO uses character encodings, called character maps, to minimize the number of bits required to encode a character. A small character encoding allows you to create a longer watermark string.

**Example**

```
<premark option="on">
    <encoding name="7bit-a"/>
</premark>
```

DashO defines 5 character maps you can choose from to encode your watermark string.

| Name | Description | Bits/Character |
|---|---|---|
| `6bit-a` | 6 bit Uppercase Alphanumeric and symbols | 6 |
| `6bit-b` | 6 bit Alphanumeric and symbols | 6 |
| `7bit-a` | 7 bit Alphanumeric and symbols | 7 |
| `4bit-a` | 4 bit Hexadecimal | 4 |
| `utf8` | Any Character | 8 |

The watermark string can have only those characters that are legal for the specified encoding. For example, if your string contains lower case letters, you cannot use an encoding such as `6bit-a` which only holds upper case letters.

**Note**

The user interface displays the specific characters defined in each character map.

### <watermark>

This option sets the watermark to be embedded in the output jar. The characters in the watermark string must comply with the character set permitted for the specified encoding.

The maximum size of the watermark string is governed by your configuration options and by the complexity of the target jar. In general, you can fit bigger strings in bigger jars.

**Example**

```
<premark option="on">
    <watermark>Copyright Yoyodyne Engineering, Inc.</watermark>
</premark>
```

### <passphrase>

In addition, the encryption algorithm has a fixed block size. If you choose to encrypt the watermark string, it will require more space. As a result, the maximum length of your watermark string may be smaller than it is without encryption.

**Example**

```
<premark option="on">
    <passphrase>secret</passphrase>
</premark>
```

## <includenonclassfiles> Section

DashO can copy related non-class files into its destination directory to jar as part of the run. For example, assume your application is embedded within a jar file that contains gif files scattered throughout the directory hierarchy in the jar. In addition to putting obfuscated class files into the destination, it can also copy the gifs to any other non-class files into the destination you specified.

It is also possible with non-class file includes to specify a relative path from the root of the destination directory or root of the jar to which the non-class files are copied. This relative path is optional. If a relative path is not specified, individual non-class files are copied to the root of the destination directory or jar.

In the following example DashO copies the non-class file to the root of the destination directory or jar.

**Example**

```
<includenonclassfiles>
    <copy source="c:\gifs\important.gif"/>
</includenonclassfiles>
```

In the following example DashO will copy the `.gif` files in the directory `c:\gifs` to the root of the destination directory or jar. Other directories in the source will not be searched for the .gif files.

**Example**

```
<includenonclassfiles>
    <copy source="c:\gifs\*.gif"/>
</includenonclassfiles>
```

In the following example the non class file will be copied to the directory `c:\test\dashoed\gifs`. A sub directory `gifs` will be created in the output directory `c:\test\dashoed`.

**Example**

```
<output>
    <dir path="c:\test\dashoed"/>
</output>

<includenonclassfiles>
    <copy source="c:\gifs\important.gif" relativedest="/gifs"/>
</includenonclassfiles>
```

If a directory is specified as the source, all non-class files, found through a recursive decent, are copied to the destination while preserving the hierarchy.

**Example**

```
<includenonclassfiles>
    <copy source="c:\nonclassfiles\"/>
</includenonclassfiles>
```

If a jar or zip file is specified, all non-classes are copied while preserving the internal hierarchy.

**Example**

```
<includenonclassfiles>
    <copy source="c:\test\nonclassfiles.jar"/>
</includenonclassfiles>
```

If a relative path is specified with a jar or zip file, the hierarchy is recreated under the specified relative path.

**Example**

```
<includenonclassfiles>
    <copy source="c:\test\nonclassfiles.jar" relativedest="misc"/>
</includenonclassfiles>
```

**Note**

All non-class files from a jar specified using `<quickjar>` entry points are automatically copied to the destination jar.

# <preverifier> Section

If you are running a J2ME CLDC application, DashO allows you to run the preverifier on the class files after DashO has finished processing the application. If you have set the `run` attribute to `true`, you can specify the path to the preverifier program. If you specify only a path, DashO assumes that the program name is `preverify`.

The `<preverifier>` tag also contains the following attributes which pass additional options to the preverifier:

- `nofinalize="true/false"` – Pass `-nofinalize` to the preverifier: no finalizers allowed.

- `nonative="true/false"` – Pass `-nonative` to the preverifier: no native methods allowed.
- `nofp="true/false"` – Pass `-nofp` to the preverifier: no floating point operations allowed.

**Example**

```
<preverifier run="true" nonative=”true” nofp=”true”>
    ${wtk.home}/bin/preverify.exe
</preverifier>
```

## <signjar> Section

This section lets you run the `jarsigner` tool on the jars created by DashO. Additional details on jar signing can be found in jarsigner - JAR Signing and Verification Tool. The `<signjar>` tag has the following attributes:

- `option="on/off"` – Turns signing on or off. If not present, the default is `on`.
- `keystore="…"` – The URL to the key store. Optional, defaults to `.keystore` in the user's home directory. If the URL does not include a protocol the key store is assumed to be a file.
- `storepass="…"` – Password for the key store. Required. This is also the default value for the private key if `keypass` is not specified. The user interface stores this in an encoded form but the value can be in plain text and may contain property references.
- `storetype="…"` – The type of the key store. Optional, defaults to the value set for `keystore.type` in the Java security properties file.
- `alias="…"` – Alias used to store the private key in the key store. Required.
- `keypass="…"` – Password for the private key used to sign the jar. Optional, defaults to the password for the key store. The user interface stores this in an encoded form but the value can be in plain text and may contain property references.
- `sigfile="…"` – Base name for the `.SF` and `.DSA` files. Optional, defaults to value derived from the `alias`.
- `internalsf="true/false"` – Include a copy of the signature file in the `.DSA`. Optional, defaults to `false`.
- `sectionsonly="true/false"` – The signature file will not include a header containing a hash of the manifest file. Optional, defaults to `false`.

**Example**

```
<signjar option="on"
  keystore="../dev/keystore" storepass="${keystore.psw}"
  alias="lazardo">
    ${jdk.home}/bin/jarsigner
</signjar>
```

## <instrumentation> Section

This section describes how to specify instrumentation for the Runtime Intelligence System. This section includes options to define instrumentation properties, the handling of annotations, and the definition of virtual annotations.

The `<instrumentation>` tag has the following attributes:

- `option="on/off"` – Turns DashO's instrumentation feature on or off. If not present, the default is `on`.
- `honorAnnotations="true/false"` – Determines if instrumentation annotations present in the compiled classes will be acted upon. If `true`, then DashO will process the instrumentation annotations in the classes. If not preset, then the default is `true`.
- `stripAnnotations="true/false"` – Determines if instrumentation annotations present in the compiled class will be retained in the output. If `true`, DashO will remove the annotations. If not present, then the default is `true`.
- `sendMessages="true/false"` – When set to `false` no messages will be sent to the Runtime Intelligence server. If `supportOffline` is `true` then the messages will be saved for later transmission. This feature can be controlled by the OfflineMode and OfflineModeSource annotations or programmatically. If not present, then the default is `true`.
- `supportOffline="true/false"` – Determines the disposition of messages that cannot be immediately sent to the Runtime Intelligence server. If set to `true` then the messages will be stored locally until they can be sent to the server. Messages may be stored locally when `sendMessages` is `false` or communication to the server is not possible. This feature can be controlled by the OfflineModeSupport and OfflineModeSupportSource annotations or programmatically. If not present, the default is `true`
- `fullData="true/false"` – Determines how much information is sent that identifies the user/host and the data sent with a system profile message. Setting this to `false` will send the minimal amount of information which can reduce startup and shutdown time. If not present, the default is `true`.

If the instrumentation tag is not present then annotations in the compiled classes will be ignored, but retained in the output. If the option attribute is `off`, then the entire instrumentation tag is ignored regardless of its contents. Since the attributes have default values, the following tags are equivalent:

**Example**

```
<instrumentation />

<instrumentation option="on"
    honorAnnotations="true"
    stripAnnotations="true"
    sendMessages="true"
    supportOffline="true"
    fullData="true"/>
```

**Note**

Instrumentation is not available in Quick Jar mode. Quick Jar mode ignores all the options set in the instrumentation section.

The <instrumentation> tag only processes or removes the annotations from the `com.preemptive.annotation.instrumentation` package. For information on these annotations see the related javadoc.

## <endpoint> Section

The <endpoint> defines where the runtime information will be sent. The endpoint tag has the following attributes:

- `name="name"` – This is the location of the Runtime Intelligence server. The end point is like a URL but does not include the protocol. If not specified the commercial server is used.
- `ssl="true/false"` – Should HTTP or HTTPS protocol be used when sending data to the endpoint. The default value is true.

These values can also be set by the Endpoint and UseSSL annotations.

## <runtime> Section

The instrumentation tag can contain an optional runtime tag that is used to specify which Runtime Intelligence System implementation jar will be used with the application and how it will be handled. If the tag is omitted then the default values for its attributes will be used. The runtime tag has the following attributes:

- `target="java14"` – The execution environment for the application. The supported values are: `java14` for Java 1.4 through 1.6; `android4` for Android SDK 1.6 and up.
- `merge="true/false"` – Will the runtime library be merged with the application's classes. The default value is `true` which allows DashO to merge the classes into the output allowing for full renaming and pruning of the implementation's classes. If `false`, then the implementations jar will need to be shipped with the application and added to its class path.

**Example**

```
<instrumentation>
    <runtime target="java14" merge="true" />
</instrumentation>
```

## <company> and <application> Sections

The instrumentation tag can contain optional company and application tags. These define property values that are used by instrumentation. The tags and all their attributes are optional.

### <company>

- `name="name"` – the name of the application.
- `id="id"` The ID of the company providing the application. This value must be specified as a GUID and must be a value obtained from PreEmptive Solutions. For more information about obtaining an activation code, see How Do I Get an Activation Code.

### <application>

- `name="name"` – The name of the application.
- `id="id"` – The ID of the application. This value must be specified as a GUID.
- `version="version"` – The version of the application. This version can be expressed in any format.
- `type="type"` – The type of application. The type can be any user defined string.

**Example**

```
<instrumentation>
    <company name="Yoyodyne Engineering, Inc."
        id="DF29A894-C1AB-5947-E0A2-0D9779CFFB63" />
    <application id="40A80B91-FB16-BB0F-96CF-6931B4472204"
        version="9.3.4" type="Swing App" />
</instrumentation>
```

## <expiry> Section

The instrumentation tag can contain optional expiration information. These define the values that will be used by the `ExpiryCheck` annotation to create an expiration token that is placed in the application. Note that all of the attributes can contain property references that are expanded at the time the injection takes place.

- `key="file"` – The Shelf Life key file obtained from PreEmptive Solutions.
- `date="date"` – A fixed expiration date in `MM/DD/YYYY` format. This is the date at which the application will be considered expired.
- `warningdate="date"` – A fixed warning date in `MM/DD/YYYY` format. This is the date on which warnings about expiration will be begin.
- `period="days"` – An expiration period. This is the number of days from a starting date on which the application will be considered expired. The starting date is provided by the application with the `StartDateSource` annotation.
- `warningperiod="days"` – A warning period. This is the number of days before the expiration when the expiration warning period starts.

Combinations of fixed dates and periods are allowed. If values for both the fixed date and period are present, the fixed date is used. Annotations that appear in the application code or are defined via virtual annotations can override or augment these values.

**Example**

```
<instrumentation>
    <expiry key="../yoyodyne.slkey"
            date="10/25/${EXP_YR}"
            warningperiod="90"/>
</instrumentation>
```

### Expiration Token Properties

User defined properties may be added to the expiration token. These properties have the same form as other DashO property tags. The properties may be examined by the application when a user action is specified with the `ExpiryCheck` annotation.

**Example**

```
<instrumentation>
    <expiry key="…" date="10/25/2015">
        <property name="REGION" value="2"/>
        <property name="COUNTRY" value="GB"/>
    </expiry>
</instrumentation>
```

Both the name and value attributes may contain property references and are expanded at the time the `ExpiryCheck` is injected.

## Virtual Annotations

The instrumentation tag can contain one or more virtual annotation definitions. DashO acts on these annotations as if they were in the compiled class files. Virtual annotations can be used to augment existing annotations or to override their values. The virtual annotations are associated with the compiled classes by using one or more `<classes>` tags. These tags follow the same syntax as those found in include and exclude lists. See the section on <includelist> and <excludelist> Rules for more information.

**Note**

The classes tag does not support the `excludeclass` attribute, nor can it contain `field` tags.

One or more annotations tags may appear inside a `<classes>` tag or its contained `<method>` tag.

### <annotation> Tag

The annotation tag defines the virtual annotation that will be applied to a class or method. An annotation has two attributes and can have any number of nested properties.

- `name="name"` – The name of the annotation. Although any name can be used here, DashO only processes the annotations that are found in the `com.preemptive.annotation.instrumentation` package. Annotations can be referenced by their simple name, e.g., ApplicationStart, rather than their fully qualified name.
- `value="value"` – An optional value for the annotation. Some annotations such as SystemProfile do not use values, while others such as FeatureTick require one. Values can contain property references which will be expanded when the annotation is applied.

**Example**

```
<classes name="com.yoyodyne.Overthruster">
  <annotation name="CompanyId"
              value="DF29A894-C1AB-5947-E0A2-0D9779CFFB63"/>
  <method name="main" signature="java.lang.String[]">
  <annotation name="ApplicationStart"/>
  <annotation name="ApplicationStop"/>
  <annotation name="PropertySource" value="staticProps"/>
  </method>
</classes>
```

In this example, the main method bounds the application's start and stop. Both feature messages send along additional properties from the static field `staticProps`. Note that the company name has been set on the class, but is then used by the `ApplicationStart` in the main method. The order of the annotations is not important - DashO sorts out the details when the code is instrumented.

**Example**

```
<classes name="com.yoyodyne.Overthruster">
  <method name="start" signature="">
    <annotation name="ApplicationStart">
      <property name="where" value="End"/>
    </annotation>
    <annotation name="Company">
      <property name="name" value="Yoyodyne Engineering, Inc."/>
      <property name="id" value="DF29A894-C1AB-5947-E0A2-0D9779CFFB63"/>
    </annotation>
  </method>
  <method name="stop" signature="">
    <annotation name="ApplicationStop"/>
  </method>
  <method name="testOscillation " signature="">
    <annotation name="FeatureStart" value="Oscillation Test"/>
    <annotation name="FeatureStop" value="Oscillation Test"/>
    <annotation name="PropertySource" value="getTestParameters()"/>
  </method>
</classes>
```

This example shows the use of annotations that contain properties. The `ApplicationStart` is performed at the *end* of the start method. Although the `Company` annotation does not have a value, it consists of two properties.

Annotation values can use both class and method dynamic properties. You can use `METHOD_NAME` and `PROP_NAME` in annotations used at the class level. The actual values will be expanded only when the annotation is applied to a specific method.

## Specifying Sources

Several annotations specify sources for dynamic information that will be used with the generated information. These sources can reference either a field or a method defined in the current class. You can use the following format for specifying a source:

- `field` - use a field in the current class as the source. If the source is used from a static method it must be static, otherwise it must be an instance field.
- `@field` - use a static field in the current class as the source. This can be used from static or instance methods.
- `class.field` - use a static field in the given class as the source. Class is a fully qualified Java class name. This can be used from static or instance methods.
- `method()` - use a method in the current class as the source. If the source is used from a static method it must be static, otherwise it must be an instance method.
- `@method()` - use a static method in the current class as the source. This can be used from static or instance methods.
- `class.method()` - use a static method in the given class as the source class is a fully qualified Java class name. This can be used from static or instance methods.

# <includelist> and <excludelist> Rules

Some tags in the project file use `<includelist> and/`or `<excludelist>` to fine tune the items to which an operation is applied. These tags specify a list of rules that are applied to select a given class, method, or field.

For tags that use both includes and excludes includes are determined first. If the `<includelist>` is empty then **all** item are included. If an item is included then the exclude rules are checked. If the `<excludelist>` is empty then **no** items are excluded. Rules within each list are applied in the order that that they are specified in the project file. Additionally, internal rules of DashO, the requirements of other options, and the classes themselves may cause items to be excluded.

The name of classes and members and well as method signatures may be specified as literals, patterns or regular expressions. See Names: Literals, Patterns, and Regular Expressions for details. The modifiers of the item can also be used as criteria, see Modifiers attribute for details.

## <classes> Tag

The `<classes>` tag is used to define a rule that selects one or more classes. Note that the class name should be fully qualified names and inner classes are specified by using a `$` as the separator between outer and inner class names.

The `<classes>` tag selects a class in order to specify additional rules for selecting fields and methods. If the tag does not contain any `<field>` or `<method>` tags, then it can be used to apply to either all members of the class or the class itself. This behavior is determined by the option that is using the rule.

Some exclude lists allow a `<classes>` name to be applied to the class itself rather than the members of the class. This is controlled by the optional, `excludeclass` attribute. The default value for the `excludeclass` attribute is `true`. Please consult the individual tags that use `<excludelist>` to see if the `excludeclass` attribute is used by that option.

**Examples**

```
<classes name=".*" regex="true"/>

<classes name="library.Class1$NestedClass"/>

<classes name="myco.Test.MyOtherTest" excludeclass="false”>
```

## <method> Tag

`<method>` tags are used inside the `<classes>` tag. Methods may be selected by name and signature. The setting for the `<method>`'s `regex` is inherited from the `<classes>` tag: if the value of `regex` for the enclosing `<classes>` is true, the `name` and `signature` attributes are regular expressions. The following example selects all methods beginning with `set` with any number of parameters using a regular expression:

**Example**

```
<classes name=".*" regex="true"/>
    <method name="set.*" signature=".*"/>
</classes>
```

The `signature` attribute can be used as criteria for selection. The signature attributes is a comma separated list of Java types that match the types in the method's parameter list. The class names of the parameters must be fully qualified. Use an empty string to specify a method that has no parameters.

**Example**

```
<classes name=".*" regex="true"/>
    <method name="get[A-Z].*" signature=""/>
</classes>

<classes name=".*"/>
    <method name="set*" signature="int,MyClass,MyClass[]"/>
</classes>

<classes name="AnalysisEngine" />
    <method name="compute" signature="int,java.util.List,float[]"/>
</classes>
```

## <field> Tag

`<field>` tags are used inside the `<classes>` tag. The setting for the `<field>'s regex` is inherited from the `<classes>` tag: if the value of `regex` for the enclosing `<classes>` is true, the `name` attribute is a regular expression.

The `<field>` tag is not applicable to all include or exclude lists as the actions of some options only apply to methods. Please consult the individual tags that use include or exclude lists to see if the `<field>` tag can be used. The following example selects all fields starting with `counter` using a regular expression:

**Example**

```
<classes name=".*" regex="true"/>
    <field regex="true" name="counter.*"/>
</classes>
```

## Combining <method> and <field>

A `<classes>` tag can contain multiple `<method>` and `<field>` tags to create a rule that selects many items in your project. For example:

**Example**

```
<classes name="com\.yoyodyne\.beans\..*" regex="true">
    <method name="get[A-Z].*" signature=""/>
    <method name="set[A-Z].*" signature=".*"/>
    <method name="is[A-Z].*" signature=""/>
    <field name="CONST_.*"/>
</classes>
```

## Modifiers attribute

The `<classes>`, `<method>` and `<field>` tags all have a `modifiers` attribute. The attribute is used to match the item by its Java modifiers or keywords. Multiple modifiers can be specified by separating them with spaces. If `modifiers` is omitted then the modifiers of the item are not used as part of the matching criteria. The modifiers are:

- `public` – the visibility of the item is `public` in the source code.
- `protected` – the visibility of the item is `protected` in the source code.
- `private` – the visibility of the item is `private` in the source code.
- `default` – this represents the default visibility given to an item when neither `public`, `protected`, nor `private` has been specified in the source code.
- `abstract` – the item has been marked `abstract` in the source code. It has no meaning when used with `<field>`.
- `final` – the item has been marked `final` in the source code.
- `static` – the item has been marked `static` in the source code.
- `native` – a method has been marked as `native` in the source code. It has no meaning when used with `<classes>` or `<field>`.
- `strictfp` – the item has been marked as `strictfp` in the source code.
- `synchronized` – the method has been marked as `synchronized` in the source code. It has no meaning when used with `<classes>` or `<field>`.
- `transient` – the field has been marked as `transient` in the source code. It has no meaning when used with `<classes>` or `<method>`.
- `volatile` - the field has been marked as `volatile` in the source code. It has no meaning when used with `<classes>` or `<method>`.
- `class` – the item is a `class`. This only has meaning when used with `<classes>`.
- `interface` – the item is an `interface`. This only has meaning when used with `<classes>`.
- `enum` – the item is a `enum`. This only has meaning when used with `<classes>`.
- `annotation` – the item is a Java `annotation`. This only has meaning when used with `<classes>`.
- `synthetic` – the Java compiler has created this item as an implementation detail and it does not appear as part of the source code.

Unrecognized modifiers are ignored. Modifiers can also be specified as a negation by adding an `!` before the modifier. Modifiers are not case sensitive.

**Examples**

```
<classes modifiers="public class" name="com.yoyodyne.*" >
    <method modifiers="!private !native" name="*" signature="**"/>
    <field modifiers="!public final" name="*" />
</classes>

<classes modifiers="!default !private !enum !annotation" name="**" >
    <method modifiers="!default !private" name="*" signature="**"/>
    <field modifiers="!default !private" name="*" />
</classes>
```

# Names: Literals, Patterns, and Regular Expressions

Name of classes and members may be specified as either a literal value, a pattern, or as a regular expression. A literal value lets you specify exactly what item to match while patterns and regular expressions let you match one or more items with a single entry. By default names are treated as literal values unless they contain a `?` or `*`. To specify a regular expression, the `regex="true"` attribute must be added to the tag.

## Using Patterns

Patterns are just like literal values but contain one or more of the following pattern indicators:

- `?` – Matches a single character.
- `*` – Matches zero or more characters, with limits. What can be matched depends upon the type of item you are matching - this is discussed in the following sections.
- `**` - Matches zero or more characters without limits.

### Patterns in Class Names

When a `*` is used in a class name it will match items within a single package, but not in sub-packages. The `**` pattern will match items within the package or any sub-package.

### Patterns in Method and Field Names

There is no difference between a `*` and `**` used in method and field names. Both match zero or more characters.

### Patterns in Method Signatures

When patterns are used in method signatures, there is a difference between the `*` and `**`. The `*` pattern will match zero or one argument to the method while the `**` will match any number of arguments.

For example:

| Arguments | Patterns | | | |
|---|---|---|---|---|
| | * | ** | long,* | long,** |
| **No args** | ✓ | ✓ | | |
| **Int** | ✓ | ✓ | | |
| **java.lang.String** | ✓ | ✓ | | |
| **long,int** | | ✓ | ✓ | ✓ |
| **long,boolean,int** | | ✓ | | ✓ |