Java

Overview

This document covers common coding styles and guidelines for all ForgeRock products.

The ForgeRock Java coding style is loosely based on the Sun Java conventions. We provide a set of Checkstyle rules which can be used to enforce the code style, as well as integration for Maven based projects.

Copyright notices

All new source files must begin with the following copyright notice, which should be adapted accordingly for non-Java source code (e.g. XML, properties, etc):

```
/*
 * Copyright 2020 ForgeRock AS. All Rights Reserved
 *
 * Use of this code requires a commercial software license with ForgeRock AS.
 * or with one of its affiliates. All use shall be exclusively subject
 * to such license between the licensee and ForgeRock AS.
 */
```

All source files should contain copyright attributions for the people or organizations who have contributed the code. For new files this should be of the form:

```
Copyright [year] [owner]
```

The attribute should be prefixed with "Portions" for changes to existing files:

```
Portions copyright [year] [owner]
```

When multiple contributions have been made in different years by the same contributor a year range should be used and kept up to date, for example:

```
Portions copyright 2011-2016 ForgeRock AS.
```

Note that copyright attributions DO NOT need to include a (c) symbol nor the phrase "All rights reserved". In addition, projects MUST include the CDDL license in its entirety in the project relative location `legal/CDDLv1.0.txt`.

The ForgeRock Java coding style

Summary

The ForgeRock Java code style rules are based on the Sun Java conventions with some deviations, the key points being:

- source files MUST contain a valid copyright notice
- source files MUST NOT contain tab characters
- source files MUST NOT contain lines longer than 120 characters, except for `import` statements
- source files MUST NOT contain lines with trailing white-space
- all packages MUST have a `package-info.java` containing at least a short summary of the package
- all public and protected types, methods, and fields MUST have full Javadoc
- code MUST be formatted using 4 character indents
- curly braces MUST be on the same line as their associated statement

In detail

Source files

- MUST contain a new line at the end of the file
- MUST NOT contain any TAB characters
- MUST NOT contain trailing white space
- MUST contain a valid copyright notice (see above)
- MUST NOT contain lines longer than 120 characters, with the exception of `import` statements
- MUST NOT contain more than one outer type (class, interface, etc)
- package declaration MUST correspond to the source file's directory
- outer type name MUST correspond to the source file's name

Documentation (Javadoc)

- packages MUST contain a `package-info.java` file with at least a simple one line summary of the package's content
- type declarations (e.g. class, interface, etc) MUST have Javadoc if the type has protected or public scope. Although not required, consider also adding type Javadoc for package private types
- method declarations MUST have Javadoc if the method has protected or public scope, and it MUST include a description, type parameters, parameters, return type, and checked exceptions. Unchecked exceptions MAY be documented at the developer's discretion even if the exception is not declared (unchecked exceptions SHOULD NOT be specified by a method's throws declaration)
- fields MUST have Javadoc if the field has protected or public scope
- where Javadoc is required, empty or stubbed out Javadoc IS NOT acceptable
- source files SHOULD NOT include @author annotations: they provide no benefit and are often incomplete and/or misleading in mature projects.
- comments MAY cite tested code samples. See Citing Code Samples in Javadoc Comments for details.

NOTE: Javadoc is NOT required for unit tests.

Naming conventions

- generic type parameters (e.g. `T` in `List<T>`) MUST begin with a single upper-case letter and be followed by zero or more digits.
- constants (declared using `static final`) MUST contain only upper-case letters, digits, or underscores
- package names MUST contain only alphanumeric characters and underscores. Ideally, they SHOULD only contain lower case letters
- all other names MUST be alphanumeric camel-case and begin with a lower-case letter

Imports

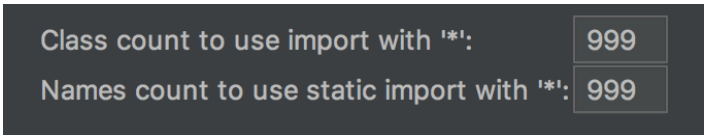
- wild-cards MUST NOT be used when importing classes, e.g. `import java.util.*` is not allowed. (hint for IntelliJ)
- wild-cards MUST NOT be used when importing static members, e.g. `import static org.junit.Assert.*` is no longer allowed
- redundant, illegal, and unused imports MUST NOT be present
- static imports MUST be grouped together, above the non-static imports

Imports - IntelliJ

To apply the agreed coding style of using explicit imports (i.e. using no wildcard imports), please update your Intelli-J settings as follows:

File > Other Settings > Default Settings > Editor > Code Style > Java > Imports (tab)

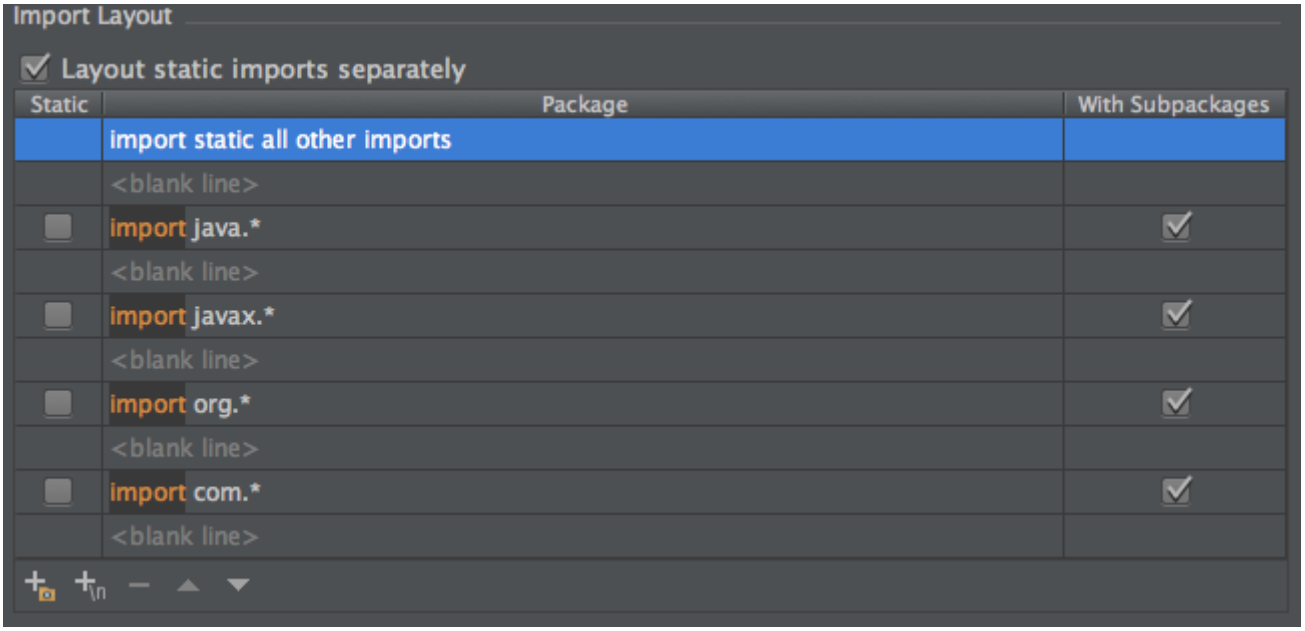
- Class count to use import with '*': 999
- Names count to use static import with '*': 999



To assist with cross product development, the following order has been devised as a standard for laying out import statements:

File > Other Settings > Default Settings > Editor > Code Style > Java > Imports (tab) > Import Layout

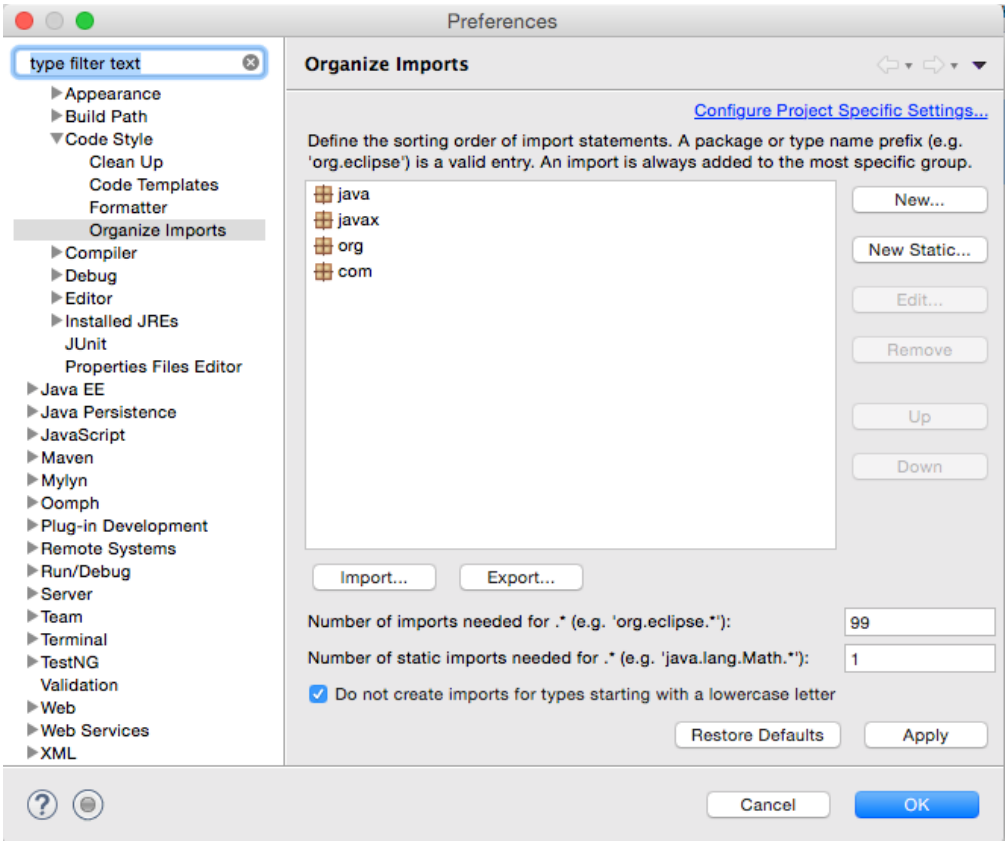
- Import static all other imports
- <blank line>
- import java.*
- <blank line>
- import javax.*
- <blank line>
- import org.*
- <blank line>
- import com.*
- <blank line>
- import all others imports



You can prompt IntelliJ to organize imports for a given file using **Code > Optimize Imports** or by clicking **ALT + SHIFT + O or Control + Option + O.**

Imports - Eclipse

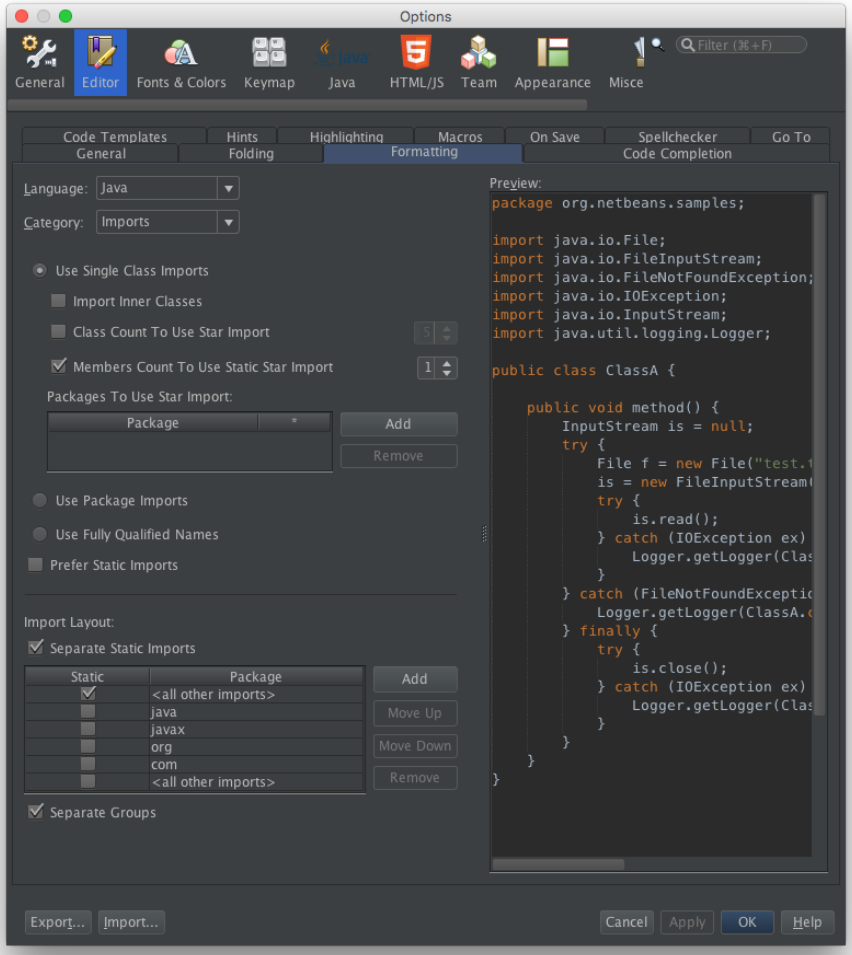
> In eclipse these may be set by going to either **Eclipse** (OSx) or **Window** (Windows & Linux) **> Preferences > Java > Code Style > Organize Imports** and making the following selections;



Imports - Netbeans

- In Netbeans got to Netbeans (OSx) > Preferences > Editor > Formatting
- Select language = Java
- Select Category = Imports

- Set up your preferences as follows:



- Go to your project properties
- Under formatting select "Use global options"

Code formatting

- blocks of code MUST be indented 4 characters, `case` statements aligned with their associated `switch`, and continuation lines 8 characters
- code MUST NOT contain nested blocks except in `switch-case` statements
- single line blocks MUST be surrounded by curly braces
- curly braces MUST appear on the same line as their associated statement, NOT on a new line
- declarations and statements MUST NOT contain unnecessary white-space. A single-white space character is permitted between a comma and the following parameter in parameter lists, but that's all
- there MUST be at most one statement per line

Coding and class design

- empty blocks of code MUST be commented, e.g. empty uncommented `catch` clauses are not allowed
- classes that override `equals()` MUST also override `hashCode()`
- strings MUST be checked for equality using `equals()` or `equalsIgnoreCase()`
- classes with private constructors MUST be final
- utility classes MUST NOT have a public constructor
- interfaces MUST define a type. A utility class SHOULD be used for constants
- classes MUST NOT contain public member variables

Using Checkstyle to enforce the coding style

A complete set of Checkstyle rules which enforce the ForgeRock coding style can be found here:

- Checkstyle rules
- Checkstyle suppressions for unit tests
- Copyright notice template

Sometimes Checkstyle validation incorrectly reports an error and there is no easy workaround. In these rare cases you may choose to temporarily disable Checkstyle from within your source code using one of the following source code annotations:

- toggle Checkstyle on and off

```
// @Checkstyle:off
... ignored
// @Checkstyle:on
```

- instruct Checkstyle to ignore the next line

```
// @Checkstyle:ignore
... ignored
... checked
```

- instruct Checkstyle to ignore next N lines (-ve means previous lines)

```
// @Checkstyle:ignoreFor 2
... ignored
... ignored
... checked
```

Maven integration

Full Maven integration has been provided for projects which declare `forgerock-parent` version 1.1.0-SNAPSHOT or later as their parent (directly or indirectly):

```
<parent>
    <groupId>org.forgerock</groupId>
    <artifactId>forgerock-parent</artifactId>
    <version>1.1.0-SNAPSHOT</version>
</parent>
```

Once the dependency is declared Checkstyle is automatically available via the `precommit` profile:

```
mvn -Pprecommit clean install
```

By default Checkstyle will use the default set of rules and copyright notice listed above . In addition, the `precommit` profile will fail the build if any validation errors are encountered.

For some existing projects it may not be feasible to fix all of the pre-existing Checkstyle violations, in which case the `precommit` profile should be configured to not fail the build. This can be achieved by setting the `checkstyleFailOnError` property to `false`:

```
mvn -P precommit -DcheckstyleFailOnError=false clean install
```

Or in the Maven `pom.xml` file:

```
<properties>
    <checkstyleFailOnError>false</checkstyleFailOnError>
</properties>
```

In some very rare cases it may be necessary to use an alternative set of Checkstyle rules or an alternative copyright template. This can be achieved using the following Maven properties (shown below with their default values):

```
<properties>
    <checkstyleSourceConfigLocation>org/forgerock/checkstyle/check-src-default.xml</checkstyleSourceConfigLocation>
    <checkstyleUnitTestSuppressionsLocation>org/forgerock/checkstyle/unit-test-suppressions.xml<
/checkstyleUnitTestSuppressionsLocation>
    <checkstyleHeaderLocation>org/forgerock/checkstyle/default-java-header</checkstyleHeaderLocation>
</properties>
```

IDE integration

Here is the the Eclipse formatter for the ForgeRock coding style : eclipse_profile_formatter_ForgeRock.xml .

And here is the IntelliJ code style settings: ForgeRock.xml (installation instructions)

Commit messages

When committing code it is essential that others can quickly get an idea of what the commit relates to, and also find more information on the issue and review. A message must always be provided when committing to trunk, sustaining or release branches/tags and we have some simple guidelines for writing them.

1. Start with the JIRA issue ID for the story or bug, and the ID of the crucible review for the code.
2. State in up to 50 chars how this commit changes the product. Begin with a capital letter and don't end with a full stop. Write as if completing the sentence *"If applied, this commit will..."*
3. If you really need to provide further info in the commit message (info about the fix should be captured in the JIRA issue), then leave a blank line below the summary before adding the details.

Example: *AME-9876 CR-1234 Add new authentication module for device auth*

Citing Code Samples in Javadoc Comments

Avoid broken code samples in Javadoc by using code citations rather than inline code in comments. The code citations are resolved when the Javadoc is built.

To cite a code samples in a Javadoc comment, use the JCite code citation system.

1. Include the sample code in your tests.
2. Add JCite as a dependency of the Javadoc Maven plugin, as in the following example.

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-javadoc-plugin</artifactId>
 <configuration>
  <taglet>ch.arrenbrecht.jcite.JCiteTaglet</taglet>
  <tagletArtifact>
   <groupId>org.mcraig</groupId>
   <artifactId>jcite</artifactId>
   <version>1.13.0</version>
  </tagletArtifact>
  <sourcepath>src/test/java</sourcepath>
  <additionalJOption>-J-Djcitesourcepath=src/test/java</additionalJOption>
 </configuration>
</plugin>
```

3. Cite the sample code as described in http://www.arrenbrecht.ch/jcite/javadoc.htm.
4. Build and check the Javadoc.

Known Issues

1. Before reformatting your code in order to comply with the coding conventions be aware of the following:
    a. reformatting will not add missing Javadoc!
    b. reformatting may introduce widespread changes which will make it harder to back-port fixes to branches because it will not be possible to apply patches
    c. widespread changes resulting from formatting will make tools like Fisheye less useful because the revision annotations will all refer to the same reformat revision, giving the appearance that the file was entirely written by a single user. To see the remaining revision history it will be necessary to step back to a revision before the reformatting occurred
    d. IDEs such as Eclipse will attempt to rewrap single-line comments. If multiple single lined comments have been used instead of a block comment, then the comments may become mangled during reformatting. For example:

    ```
    // This is a comment which uses multiple single-line comments "//" instead
    // of a multi-line comment block using "/* ... */"
    ```

    Could become:

    ```
    // This is a comment which uses multiple single-line comments "//"
    // instead
    // of a multi-line comment block using "/* ... */"
    ```

2. The Checkstyle indentation rule contains a bug which causes it to incorrectly flag correctly indented throws declarations as errors. The Checkstyle Maven integration includes a fix for this, however it will not be available if running Checkstyle outside of Maven, e.g. within an IDE or as part of an Ant build.