Spring Framework Code Style

Stéphane Nicoll edited this page on 24 Jul 2014 · 7 revisions

Introduction

This document defines the coding standards for source files in the Spring Framework. It is primarily intended for the Spring Framework team but can be used as a reference by contributors.
The structure of this document is based on the Google Java Style reference and is *work in progress*.

Source File Basics

File encoding: ISO-8859-1

Source files must be encoded using `ISO-8859-1`. (see SPR-11569 for a suggestion to move to `UTF-8`)

Indentation

- Indentation uses *tabs* (not spaces)
- Unix (LF), not DOS (CRLF) line endings
- Eliminate all trailing whitespace
    - On Linux, Mac, etc.: `find . -type f -name "*.java" -exec perl -p -i -e "s/[ \t]$//g" {} \;`

Source file structure

A source file consists of the following, in this exact order:

- License
- Package statement
- Import statements
- Exactly one top-level class

Exactly one blank line separates each of the above sections.

License

Each source file must specify the following license at the very top of the file:

```
/*
 * Copyright 2002-2014 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
```

```
 * limitations under the License.
 */
```

Always check the date range in the license header. For example, if you've modified a file in 2014 whose header still reads:

```
* Copyright 2002-2011 the original author or authors.
```

Then be sure to update it to 2014 accordingly:

```
* Copyright 2002-2014 the original author or authors.
```

Import statements

The import statements are structured as follow:

- import `java.*`
- import `javax.*`
- blank line
- import all other imports
- blank line
- import `org.springframework.*`
- blank line
- import static all other imports

Also, static imports should not be used in production code. They should be used in test code, especially for things like `org.junit.Assert`.

Java source file organization

The following governs how the elements of a source file are organized:

1. static fields
2. normal fields
3. constructors
4. (private) methods called from constructors
5. static factory methods
6. JavaBean properties (i.e., getters and setters)
7. method implementations coming from interfaces
8. private or protected templates that get called from method implementations coming from interfaces
9. other methods

10. `equals`, `hashCode`, and `toString`

Note that private or protected methods called from method implementations should be placed immediately below the methods where they're used. In other words if there 3 interface method implementations with 3 private methods (one used from each), then the order of methods should include 1 interface and 1 private method in sequence, not 3 interface and then 3 private methods at the bottom. Above all, the organization of the code should feel *natural*.

Formatting

Braces

Block-like constructs: K&R style

Braces mostly follow the *Kernighan and Ritchie style* (a.k.a., "Egyptian brackets") for nonempty blocks and block-like constructs:

- No line break before the opening brace but prefixed by a single space
- Line break after the opening brace
- Line break before the closing brace
- Line break after the closing brace if that brace terminates a statement or the body of a method, constructor, or named class with the exception of the `else`, `catch` and `finally` statements that also lead to a line break

Example:

```
return new MyClass() {
    @Override
    public void method() {
        if (condition()) {
            something();
        }
        else {
            try {
                alternative();
            }
            catch (ProblemException ex) {
```

```
                recover();
            }
        }
    }
};
```

Line wrapping: around 90 characters

Aim to wrap code and Javadoc at 90 characters but favor readability over wrapping as there is no deterministic way to line-wrap in every situation.

When wrapping a lengthy expression put the separator symbol at the end of the line, rather on the next line (think comma separated arguments as an exemple). For instance:

```
if (thisLengthyMethodCall(param1, param2) && anotherCheck() &&
        yetAnotherCheck()) {
....
}
```

Blank Lines

Add two blank lines before the following elements:

- `static {}` block
- Fields
- Constructors
- Inner classes

Add one blank line after a method signature that is multiline, i.e.

```
@Override
protected Object invoke(FooBarOperationContext<FooBarOperation>
context,
            FooBarInvoker invoker) {

    CacheKeyInvocationContext<CachePut> invocationContext = ...
```

Class declaration

Try as much as possible to put the `implemnts`, `extends` section of a class declaration on the same line as the class itself.

Order the classes so that the most important comes first.

Naming

Constant names

Constant names use `CONSTANT_CASE`: all uppercase letters, with words separated by underscores.
Every constant is a `static final` field, but not all `static final` fields are constants. Constant case should therefore be chosen only if the field **is really** a constant.
Example:

```
// Constants
private static final Object NULL_HOLDER = new NullHolder();
public static final int DEFAULT_PORT = -1;

// Not constants
private static final ThreadLocal<Executor> executorHolder = new
ThreadLocal<Executor>();
private static final Set<String> internalAnnotationAttributes =
new HashSet<String>();
```

variable names

Avoid using variable name for a single character. For instance prefer `Method method` to`Method m`.

Programming Practices

File history

- A file should look like it was crafted by a single author, not like a history of changes
- Don't artificially spread things out that belong together

Organization of setter methods

Choose wisely where to add a new setter method; it should not be simply added at the end of the list. Perhaps the setter is related to another setter or relates to a group. In that case it should be placed near related methods.

- Setter order should reflect order of importance, not historical

order

- Ordering of *fields* and *setters* should be **consistent**

Ternary Operator

Wrap the ternary operator within parentheses, i.e. `return (foo != null ? foo : "default");`

Also make sure that the *not null* condition comes first.

Null Checks

Use the `org.springframework.util.Assert.notNull` static method to check that a method argument is not `null`. Format the exception message so that the name of the parameter comes first with its first character capitalized, followed by "*must not be null*". For instance

```
public void handle(Event event) {
    Assert.notNull(event, "Event must not be null");
    //...
}
```

Use of @Override

Always add `@Override` on methods overriding or implementing a method declared in a super type.

Use of @since

- `@since` should be added to every new class with the version of the framework in which it was introduced
- `@since` should be added to any *new* **public** and **protected** methods of an existing class

Utility classes

A class that is only a collection of static utility methods must be named with a `Utils`suffix, must have a private default constructor, and must be abstract. Making the class abstract and providing a `private` *default* constructor prevent anyone from instantiating it. For example:

```
public abstract MyUtils {

    private MyUtils() {
        /* prevent instantiation */
```

```
    }

    // static utility methods
}
```

Field and method references

A field of a class should *always* be referenced using `this`. A method of class, however, should never be referenced using `this`.

Javadoc

Javadoc formatting

The following template summarizes a typical use for the Javadoc of a method.

```
/**
 * Parse the specified {@link Element} and register the
resulting
 * {@link BeanDefinition BeanDefinition(s)}.
 * <p>Implementations must return the primary {@link
BeanDefinition} that results
 * from the parsing if they will ever be used in a nested
fashion (for example as
 * an inner tag in a {@code <property/>} tag). Implementations
may return
 * {@code null} if they will <strong>not</strong> be used in a
nested fashion.
 * @param element the element that is to be parsed into one or
more {@link BeanDefinition BeanDefinitions}
 * @param parserContext the object encapsulating the current
state of the parsing process;
 * provides access to a {@link
org.springframework.beans.factory.support.BeanDefinitionRegistry
}
 * @return the primary {@link BeanDefinition}
 */
BeanDefinition parse(Element element, ParserContext
parserContext);
```

In particular, please note:

- Use an imperative style (i.e. *Return* and not *Returns*) for the first

sentence.

- No blank lines between the description and the parameter descriptions.
- If the description is defined with multiple paragraphs, start each of them with `<p>`.
- If a parameter description needs to be wrapped, do not indent subsequent lines (see`parserContext`).

The Javadoc of a class has some extra rules that are illustrated by the sample below:

```
/*
 * Interface used by the {@link
DefaultBeanDefinitionDocumentReader} to handle custom,
 * top-level (directly under {@code <beans/>}) tags.
 *
 * <p>Implementations are free to turn the metadata in the
custom tag into as many
 * {@link BeanDefinition BeanDefinitions} as required.
 *
 * <p>The parser locates a {@link BeanDefinitionParser} from the
associated
 * {@link NamespaceHandler} for the namespace in which the
custom tag resides.
 *
 * @author Rob Harrop
 * @since 2.0
 * @see NamespaceHandler
 * @see AbstractBeanDefinitionParser
 */
```

- Each class must have a `@since` tag with the version in which the class was introduced.
- The order of tags for class-level Javadoc is `@author`, `@since` and `@see`.
- In contrast to method-level Javadoc, the paragraphs of a class description *are*separated by blank lines.

The following are additional general rules to apply when writing Javadoc:

- Use `{@code}` to wrap code statements or values such as `null`.
- If a type is only referenced by a `{@link}` element, use the fully qualified name in order to avoid an unnecessary `import` declaration.

Tests

Naming

Each test class must end with a `Tests` suffix.

Mocking

Use the BDD Mockito support.