

# Checkstyle

Raúl Estrada

Octubre 2020

## 1. Overview

Checkstyle is an open source tool that checks code against a configurable set of rules.

In this tutorial, we're going to look at **how to integrate Checkstyle into a Java project via Maven and by using IDE plugins.**

The plugins mentioned in below sections aren't dependent on each other and can be integrated individually in our build or IDEs. For example, the Maven plugin isn't needed in our *pom.xml* to run the validations in our Eclipse IDE.

## 2. Checkstyle Maven Plugin

### 2.1. Maven Configuration

To add Checkstyle to a project, we need to add the plugin in the reporting section of a *pom.xml*:

```
1  <reporting>
2    <plugins>
3      <plugin>
4        <groupId>org.apache.maven.plugins</groupId>
5        <artifactId>maven-checkstyle-plugin</artifactId>
6        <version>3.0.0</version>
7        <configuration>
8          <configLocation>checkstyle.xml</configLocation>
9        </configuration>
10     </plugin>
11   </plugins>
12 </reporting>
```

**This plugin comes with two predefined checks, a Sun-style check, and a Google-style check.** The default check for a project is *sun\_checks.xml*.

To use our custom configuration, we can specify our configuration file as shown in the sample above. Using this config, the plugin will now read our custom configuration instead of the default one provided.




The latest version of the plugin can be found on [Maven Central](#).

## 2.2. Report Generation



Now that our Maven plugin is configured, we can generate a report for our code by running the *mvn site* command. **Once the build finishes, the report is available in the *target/site* folder under the name *checkstyle.html*.**

There are three major parts to a Checkstyle report:

**Files:** This section of the report provides us with **the list of files in which the violations have happened**. It also shows us the counts of the violations against their severity levels. Here is how the files section of the report looks like:

Files			
File	 I	 W	 E
org/baeldung/web/controller/HeavyResourceController.java	0	1	0
org/baeldung/web/controller/mediatypes/CustomMediaTypeController.java	0	1	0

**Rules:** This part of the report gives us an **overview of the rules that were used to check for violations**. It shows the category of the rules, the number of violations and the severity of those violations. Here is a sample of the report that shows the rules section:

Rules			
Category	Rule	Violations	Severity
imports	<a href="#">AvoidStarImport</a> 	2	 Warning

**Details:** Finally, the details section of the report provides us the **details of the violations that have happened**. The details provided are at line number level. Here is a sample details section of the report:

Details				
org/baeldung/web/controller/HeavyResourceController.java				
Severity	Category	Rule	Message	Line
 Warning	Imports	AvoidStarImport	Using the '*' form of import should be avoided - org.springframework.web.bind.annotation.*.	9



## 2.3. Build Integration

If there's a need to have stringent checks on the coding style, **we can configure the plugin in such a way that the build fails when the code doesn't adhere to the standards.**

We do this by adding an execution goal to our plugin definition:

```
1  <plugin>
2    <groupId>org.apache.maven.plugins</groupId>
3    <artifactId>maven-checkstyle-plugin</artifactId>
4    <version>${checkstyle-maven-plugin.version}</version>
5    <configuration>
6      <configLocation>checkstyle.xml</configLocation>
7    </configuration>
8    <executions>
9      <execution>
10        <goals>
11          <goal>check</goal>
12        </goals>
13      </execution>
14    </executions>
15  </plugin>
```

The *configLocation* attribute defines which configuration file to refer to for the validations.

In our case, the config file is *checkstyle.xml*. The goal *check* mentioned in the execution section asks the plugin to run in the verify phase of the build and forces a build failure when a violation of coding standards occurs.

Now, if we run the *mvn clean install* command, it will scan the files for violations and the build will fail if any violations are found.

We can also run only the *check* goal of the plugin using *mvn checkstyle:check*, without configuring the execution goal. Running this step will result in a build failure as well if there are any validation errors.

## 4. IntelliJ IDEA Plugin

### 4.1. Configuration

Like Eclipse, IntelliJ IDEA also enables us to use our own custom configurations with a project.

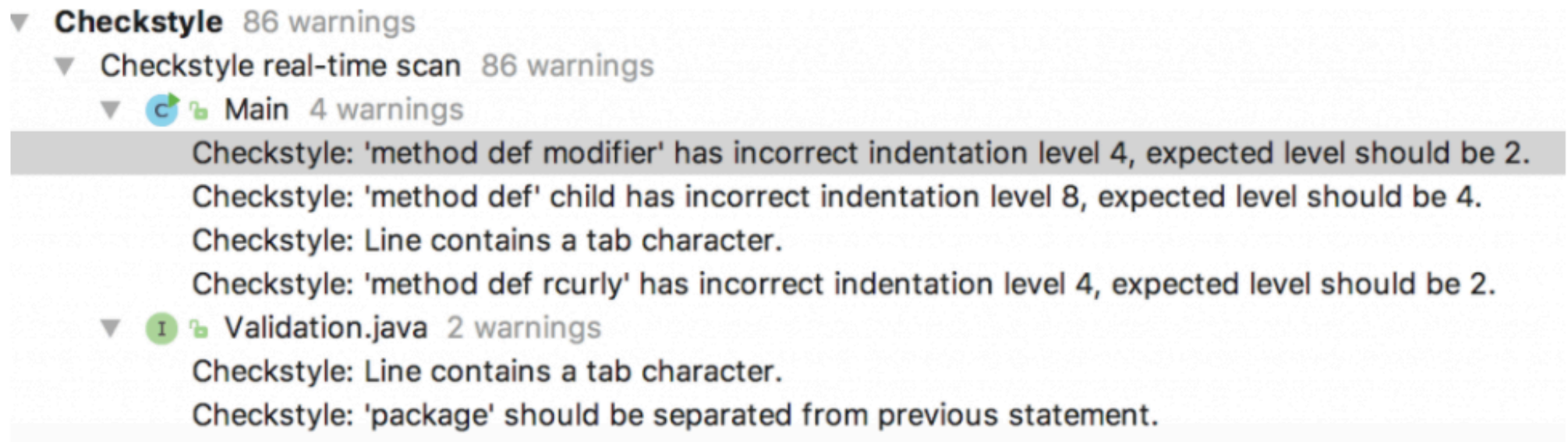
In the IDE open Settings and search for Checkstyle. A window is shown that has the option to select our checks. Click on the + button and a window will open which will let us specify the location of the file to be used.

Now, we select a configuration XML file and click Next. This will open up the previous window and show our newly added custom configuration option. We select the new configuration and click on OK to start using it in our project.



## 4.2. Reports Browsing

Now that our plugin is configured, let's use it to check for violations. To check for violations of a particular project, go to *Analyze -> Inspect Code*.

**The Inspections Results will give us a view of the violations under the Checkstyle section.** Here is a sample report:



The screenshot shows the 'Inspections Results' window in IntelliJ IDEA. It displays a tree view of inspections. The 'Checkstyle' section is expanded, showing 86 warnings. Under 'Checkstyle real-time scan', there are 86 warnings. Under 'Main', there are 4 warnings. The first warning is expanded, showing four Checkstyle messages: 'method def modifier' has incorrect indentation level 4, expected level should be 2; 'method def' child has incorrect indentation level 8, expected level should be 4; Line contains a tab character; and 'method def rcurly' has incorrect indentation level 4, expected level should be 2. Below this, 'Validation.java' has 2 warnings, which are also expanded: Line contains a tab character, and 'package' should be separated from previous statement.

- ▼ **Checkstyle** 86 warnings
  - ▼ Checkstyle real-time scan 86 warnings
    - ▼  Main 4 warnings
      - Checkstyle: 'method def modifier' has incorrect indentation level 4, expected level should be 2.
      - Checkstyle: 'method def' child has incorrect indentation level 8, expected level should be 4.
      - Checkstyle: Line contains a tab character.
      - Checkstyle: 'method def rcurly' has incorrect indentation level 4, expected level should be 2.
    - ▼  Validation.java 2 warnings
      - Checkstyle: Line contains a tab character.
      - Checkstyle: 'package' should be separated from previous statement.

Clicking on the violations will take us to the exact lines on the file where the violations have happened.

## 5. Custom Checkstyle Configuration

In the Maven report generation section (Section 2.2), we used a custom configuration file to perform our own coding standard checks.

**We have a way to create our own custom configuration XML file** if we don't want to use the packaged Google or Sun checks.

Here is the custom configuration file used for above checks:

```
1  <!DOCTYPE module PUBLIC
2    "-//Puppy Crawl//DTD Check Configuration 1.3//EN"
3    "http://www.puppcrawl.com/dtds/configuration_1_3.dtd">
4  <module name="Checker">
5    <module name="TreeWalker">
6      <module name="AvoidStarImport">
7        <property name="severity" value="warning" />
8      </module>
9    </module>
10 </module>
```

## 5.1. DOCTYPE Definition

The first line of the i.e. the DOCTYPE definition is an important part of the file and it tells where to download the DTD from so that the configurations can be understood by the system.

**If we don't include this definition in our configuration file won't be a valid configuration file.**

## 5.2. Modules

A config file is primarily composed of Modules. **A module has an attribute *name* which represents what the module does.** The value of the *name* attribute corresponds to a class in the plugin's code which is executed when the plugin is run.

Let's learn about the different modules present in the config above.

## 5.3. Module Details

- **Checker:** Modules are structured in a tree that has the Checker module at the root. This module defines the properties that are inherited by all other modules of the configuration.
- **TreeWalker:** This module checks the individual Java source files and defines properties that are applicable to checking such files.
- **AvoidStarImport:** This module sets a standard for not using Star imports in our Java code. It also has a property that asks the plugin to report the severity of such issues as a warning. Thus, whenever such violations are found in the code, a warning will be flagged against them.

To read more about custom configurations follow this [link](#).



## 6. Report Analysis for the Spring-Rest Project

In this section, we're going to shed some light on an analysis done by Checkstyle, using the custom configuration created in section 5 above, on the [spring-rest project available on GitHub](#) as an example.

## 6.1. Violation Report Generation

We've imported the configuration to Eclipse IDE and here is the violation report that is generated for the project:

Spring Explorer History Git Staging Checkstyle violations chart			
Details of Checkstyle violation "Using the '*' form of import should be avoided - X." - 2 occurrences			
Resource	In Folder	Line	Message
HeavyResourceController...	/spring-rest/src/main/java/org/baeldung/web/controller	9	Using the '*' form of import should be avoided - org.springframework.web.bind.annotation.*
CustomMediaTypeControl...	/spring-rest/src/main/java/org/baeldung/web/controlle...	5	Using the '*' form of import should be avoided - org.springframework.web.bind.annotation.*

The warnings reported here says that wildcard imports should be avoided in the code. We have two files that don't comply with this standard. When we click on the warning it takes us to the Java file which has the violation.

Here is how the *HeavyResourceController.java* file shows the warning reported:

```
import org.baeldung.repository.HeavyResourceRepository;
import org.baeldung.web.dto.HeavyResource;
import org.baeldung.web.dto.HeavyResourceAddressOnly;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
```

## 6.2. Issue Resolution

Using Star imports is not a good practice in general as it can create conflicts when two or more packages contain the same class.

As an example, consider the class *List*, which *is* available in packages *java.util* and *java.awt* both. If we use both the imports of *java.util.\** and *java.awt.\** our compiler will fail to compile the code, as *List* is available in both packages.

**To resolve the issue mentioned above we organize the imports in both files and save them.** Now when we run the plugin again we don't see the violations and our code is now following the standards set in our custom configuration.

## 7. Conclusion

In this article, we've covered basics for integrating Checkstyle in our Java project.

We've learned that it is a simple yet powerful tool that's used to make sure that developers adhere to the coding standards set by the organization.