Checkstyle is an automated static code analysis tool for Java. It was originally developed by Oliver Burn in 2001, but is now maintained on an open source remote repository. It is available at no cost to the user from a variety of sources including sourceforge.net, GitHub, or even the Eclipse IDE marketplace.

Coding rules are an important part of developing stable, maintainable software. Following them can lead to a reduction in faults, ease debugging, and produce more reusable code. Unfortunately, adhering to coding rules can be tedious for developers, and ensuring compliance can consume a lot of valuable time, especially during code reviews.

Checkstyle solves this issue. It will automatically check code and ensure that it adheres to the defined coding rules and style. It comes preset with Google Java Style and Sun Code Conventions, but is highly configurable to accommodate nearly any set of coding rules. It is highly capable, and can check things like:

* Javadoc enforcement
* Method argument count
* Tab width
* The ever popular opening brace at the end of a line, or beginning of next line
* Braces on single line complex statements (e.g. ifs and fors)
* Spacing around mathematical operators
* Much more

Checkstyle is available to support several development environments. It is available as a standalone java app, Ant task, or Eclipse plugin. If used as a plugin, it can automatically highlight offending pieces of code, and provide a concise explanation of which coding rule is being broken (screenshot 1). The plugin also comes with a user friendly GUI that can be used to configure the tool (screenshot 2). If used outside of Eclipse, configuration is done in an XML file.

Checkstyle is most useful if used from the beginning of a project. If developers are consistent about following coding rules from the beginning, they become almost a non-issue. Checkstyle makes it obvious when rules are not being followed. If the violations are rectified as they arise, valuable time can be saved later during code reviews. Checkstyle also offers a pie chart view to easily see the violation distribution among categories. This allows developers and reviewers to prioritize among rules, and focus on the ones that are considered most important (screenshot 3).

Of course, no rule is absolute, and there will always need to be exceptions. This is one of the shortcomings of Checkstyle. While it is possible to customize rules within an entire project, there is no way to add a one-off exception or an “ignore this instance of this violation” option. That means that any time a rule is not followed, Checkstyle will always flag it as a violation. Fortunately, the scenarios that involve purposefully disregarding coding rules are quite limited, and this will not be an issue under most circumstances.

The most obvious limitation of Checkstyle is that it lacks any utility to automatically fix violations. Imagine a scenario where a team is reusing legacy code with new code rules, or code rules change in the middle of a project. It would be extremely helpful if Checkstyle had a “Fix All” option. Of course, such a feature would not be feasible for every category of violation, but for many (‘{‘ must be preceded with whitespace, tab width, imports must be in lexicographical order, etc), this feature is not just feasible, but almost trivial to implement. Such automated violation remediation is not possible with Checkstyle, although there may be other tools that can accomplish this.

Clearly, Checkstyle is a powerful tool to ensure consistent, uniform, and readable code. Our group project would benefit from using this tool, and it is highly likely that we will begin using it shortly.





