Google Summer of Code

checkstyle

Project Name: Improve Modules used in Google Style

Name: Mohit Attry
Github: www.github.com/mohitsatr
LinkedIn: www.linkedin.com/in/mohitsatr
Time Zone: Indian Standard Time (IST), UTC +5:30
Project Size: Large (350 hours)
Mentors: Roman Ivanov, Mauryan Kansara, Daniel Mühlbachler,

About Me

I’m Mohit Attry. I’m a second year student pursuing Bachelor's in Computer Application at S.M.H.S Government College Mohali, India.

I’m currently learning Android development. I’m also interested in compilers and low level programming.

Why Checkstyle: I had studied about Abstract Syntax Trees (ASTs) in an online compiler course, but it was only while contributing to Checkstyle that I really saw them in action—how they work and how they’re used in practice. That hands-on experience made everything click.
Another big reason is the amazing Checkstyle community. The maintainers are super helpful and patient, especially with newcomers, which made the whole experience even better.

Project Details

During GSoC'24, a huge effort was made to improve Checkstyle's Google config coverage and updated our implementation of google style guide to the latest version (03 Feb 2022). but there's still a lot of work left. google_checks.xml is the configuration file where our google java style guide is implemented. Though most of the implementation work has been done and we have covered almost all the rules in google java style guide, users have reported a bunch of issues pointing out flaws in our implementation, these issues are labeled as google style issues, we need to solve them. On top of this, we have few rules which we are not able to completely follow/implement, they are marked with "Blue Tick" in our coverage page, we need to find a way to reduce such blue ticks rules and improve our coverage.

Deliverables

- Resolve all issues labeled as google style
- Reduce "Blue Tick" rules by analyzing the rule and our coverage for that rule and provide solutions for them.
- We have already started the process of triaging issues which affect user experience, they're listed at: https://github.com/orgs/checkstyle/projects/10, this work should be continued and finished.

Reduce "Blue Tick" Rules

We did a huge amount of work last year to improve Checkstyle's Google config coverage, but there's still a lot of work left. A few rules were not fully implemented or followed. These are marked with blue-tick checks on the coverage page.

It is discovered that some sections with blue-tick checks are missing proper descriptions as to what part of the rule is blocking from being Green ticked. These are:

4.6.2 Horizontal whitespace

This section has three blue-ticked checks–**WhitespaceAfter**, **WhitespaceAround** and **WhitespaceBefore** Checks.

We are facing problems with following rules in this section.

> ***Optional*** *[space] just inside both braces of an array initializer*

a. `new int[] {5, 6}` and `new int[] { 5, 6 }` *are both valid*

According to the guide, both styles are valid. But there is a case which seems to be lying in the middle of two possibilities. Take a look:

```
public class Foo {
   public final int[] COLORS = new int[] { 2, 1};
}
```

Google-style config shows no errors in this case. But it seems odd, a space should be either present after 1 or removed before 2. We need to decide whether to be fanatic about the case or declare this as also valid. Taken from this issue.

> *Between a type annotation and* ***[ ]*** *or* ***...*** .

The above rule is not being covered properly as users have reported many false-negative cases.

```
@NonNull int @NonNull[] @NonNull[] var1;       //false-negative
@NonNull int @NonNull [] @NonNull [] var2;     // correct

void test2(String @NonNull... param) {}        // false-negative
void test1(String @NonNull ... param) {}       // correct

void test4(String... param) {}                // false-negative
void test3(String ... param) {}               // correct

public void foo3(final char @NonNull[] param) {} // false-negative
public void foo4(final char @NonNull [] param) {} // correct
```

We need to find a way to resolve all these false-negatives.

Multiple Whitespaces:

>*Beyond where required by the language or other style rules, and apart from literals, comments and Javadoc, a single ASCII space also appears in the following places* ***only****.*

According to config, only single whitespace is allowed. But Checkstyle currently does not cover this rule and allows multiple whitespaces to pass through.
Example of such cases from issue:

```
int[][] arr = { { 1, 2, 3, } };
int max = (a > b) ?  a : b;
```

We might consider adding **SingleSpaceSeparator** Check in the config, we have to test it properly first.

5.2.2 Class names

This section has one blue-ticked Check–**TypeName** Check

> Class names are written in UpperCamelCase.

This rule is covered.

> Class names are typically nouns or noun phrases. For example, `Character` or `ImmutableList`. Interface names may also be nouns or noun phrases (for example, `List`), but may sometimes be adjectives or adjective phrases instead (for example, `Readable`).

It's impossible to validate for such requirements because Checkstyle cannot determine whether a name is a noun, an adjective, or any other grammatical category.

> There are no specific rules or even well-established conventions for naming annotation types.

There is no specific rule for naming annotations. This is covered then.

> A test class has a name that ends with `Test`, for example, `HashIntegrationTest`. If it covers a single class, its name is the name of that class plus `Test`, for example `HashImplTest`.

For example:

Even though this rule is being followed in Checkstyle's own codebase (at least in IDEA) through intellij-idea-insections.xml config.

```
<inspection_tool class="JUnitTestClassNamingConvention"
enabled="true" level="ERROR"
                enabled_by_default="true">
 <option name="m_regex" value="[A-Z][A-Za-z\d]*Test"/>
```

It is not covered by **TypeName** Check or any other Check, which means it is not covered by `google_checks.xml` either.

I need to clarify this with maintainers whether it was intentional to cover the rule this way due to some limitation in Checkstyle or we can modify the check to cover this. I feel the latter is the case. It could be easily by modify regexp pattern in the Check and adding some logic to detect if method is a test method.

5.2.3 Method names

This section has one blue-tick check–**MethodName** Check.

> *Method names are written in lowerCamelCase.*

This case is covered.

> *Method names are typically verbs or verb phrases. For example, `sendMessage` or `stop`.*

It's impossible to validate for this rule because Checkstyle cannot determine whether a name is a verb, an adjective, or any other grammatical category.

> *Underscores may appear in JUnit test method names to separate logical components of the name, with each component written in lowerCamelCase, for example `transferMoney_deductsFromSource`.* ***There is no One Correct Way to name test methods.***

This rule is currently not being followed as Checkstyle log violation on the following code.

```
@Test
public void test_Default()
       throws Exception {
  ...
}
```

```
[WARN] /mnt/567/google/TestFile.java:49:17: Method name
'test_Default' must match pattern '^[a-z][a-z0-9][a-zA-Z0-9]*$'.
[MethodName]
```

6.2 Caught exceptions: not ignored

This section has one blue-tick check–**EmptyCatchBlock** Check.

> *Except as noted below, it is very rarely correct to do nothing in response to a caught exception. (Typical responses are to log it, or if it is considered "impossible", rethrow it as an `AssertionError`.) When it truly is appropriate to take no action whatsoever in a catch block, the reason this is justified is explained in a comment.*

Example:

```
try {
  int i = Integer.parseInt(response);
  return handleNumericResponse(i);
} catch (NumberFormatException ok) {
  // it's not numeric; that's fine, just continue
}
return handleTextResponse(response);
```

This rule is properly covered through this check.

> ***Exception:*** *In tests, a caught exception may be ignored without comment if its name is or begins with `expected`. The following is a very common idiom for ensuring that the code under test does throw an exception of the expected type, so a comment is unnecessary here.*

Example:

```
try {
  emptyStack.pop();
  fail();
} catch (NoSuchElementException expected) {
}
```

This is covered through the following property in google_checks.xml

```
<module name="EmptyCatchBlock">
   <property name="exceptionVariableName" value="expected"/>
</module>
```

There is no issue related to **EmptyCatchBlock** Check in the list of polish-google-style. So I think this section of the google-style-guide is fully covered and we can put a green tick on the Check.

7.1.1 General form
7.3.4 Non-required Javadoc
7.1.2 Paragraphs
7.3.2-javadoc-exception-overrides

The above sections are related to Javadoc and they have three blue-ticked Checks–**InvalidJavadocPosition**, **JavadocParagraph**, **JavadocMethod**.
Unfortunately there's no easy way to find missed cases in the Javadoc because of the complexity and it’s really efficient and (and tiring) to consider all the possibilities where config might fail and then test them out to see if it actually fails. (There is more on this below).
So in order to mark the above Checks green, we should strive to fix every issue related to these three Checks in the issue-tracker.

Improving Testing

Currently the only way we find faults in the implementation of the google-style is either users report them to us, or we find them accidentally. There is no systematic way to go about this and, like I said above, it is a very inefficient approach.

There are some ways we can make it easy for ourselves to find error errors:

Regression-testing for google-style

Regression testing is an important part of Checkstyle's workflow as it helps Checkstyle identify any unintended differences or side effects introduced by PRs across a wide range of projects. This is especially important because tests alone cannot not always find whether the code behaves as expected in real-world scenarios.

For Checkstyle, regression testing is even more important, as it allows us to validate changes against a diverse set of Java codebases. It also allows us to find and copy complex or edge-case java code into our internal test input files. The more we do this, the more robust and comprehensive our input files become to be fed by tests—making us less dependent on external projects for validation.

Following is the list of projects we have for regression testing

- android-launcher
- apache-ant
- apache-jsecurity
- apache-struts
- camunda
- checkstyle
- checkstyle-sonar
- elasticsearch
- guava

- Hartshorn
- Hbase
- hibernate-orm
- infinispan
- java-design-patterns
- jOOL
- lombok-ast
- MaterialDesignLibrary
- openjdk21
- Orekit
- pmd
- protonpack
- RxJava
- sevntu-checkstyle
- spoon
- spotbugs
- spring-framework
- Vavr

Of all these, only three projects follow google-java-style guidelines–Camunda, Hbase and guava. This is not good as we do not have a variety of real world code to test our changes on. Ironically, to become less reliant on external projects for validation, we must first become really reliant on them.
We need to add more projects which use google-style-guidelines to that list. Ideally we should have a different list of projects just for testing "google-style" PRs.

But it is not an easy task, in order to find the perfect candidate for our purpose, we need to ensure that google-style config of checkstyle is used in the build process of the project and it also should have all Checkstyle violations resolved. There must be such projects on the web and I need to discuss with maintainers what are the potential candidates and how we should go about doing this.

We might also require making CI workflows to validate that a project is free of all violations and is ready to be used for regression testing

purposes. During the community bonding period, I'm gonna have detailed discussion on this with maintainers.

Helping Guava implement Checkstyle's google-style

Guava is a core set of Java libraries from Google. It is widely used across most Java projects within Google and by many other organizations as well. It is also one of the projects included in our regression testing suite. Since it is a Google project, it follows the Google-java-style-guide. However, it does not currently have Checkstyle's google_checks.xml enabled.

If the configuration was enabled, it'd result in hundreds of violations during the build process. Given the scale of the project, developers may be reluctant to address these style issues due to the effort required(we are lazy).

We can assist them in resolving these violations and bring the project to zero Checkstyle errors. Once the codebase is fully compliant, Guava can begin enforcing the Checkstyle google-style config on each commit, ensuring ongoing adherence to the style guide.

This would really benefit us—having a widely-used, real world project fully compliant to google-style in our regression projects suite would help us find more diffs in regression reports and would make our implementations more robust and solid.

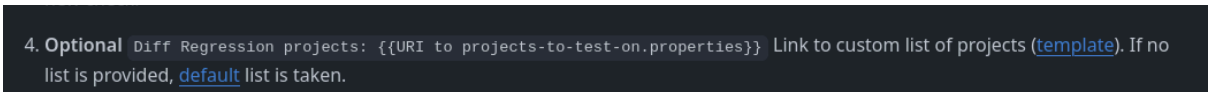
Improving Documentation

When I started contributing to Checkstyle, specifying violations in input files, writing testing and generating reports was the most unintuitive part for me. Since we use regexp so heavily in the codebase, very minute details like putting an extra whitespace mistakenly gets rejected by regexp format matcher and causes build errors. we must mention all

such nuances and more in docs. This will help avoid repetitive questions from the newcomers and save everyone's time.

Currently, there is no information in our docs on how to make tests for google-style related changes. Following are some of the suggestions we could add to improve our docs:

- On the report generation page, we need to add a list of projects(camunda and Hbase as of now) to be used when PR is related to google-style.
  Since other projects do not use the same config, it's no use to find diffs on them.



- How to use …Correct.java suffix in input file names.
- Mention that while report generation code changes must not be tested on the entire config, only the module for which changes are made. Testing the entire config burdens the CIs and sometimes fails the report generation in case it exceeds 6 hours.
- How to specify violations in the input files.

Fixing “google-style” bugs

We have compiled a list of issues related to Checks defined in `google_checks.xml`. Each issue needs to be reviewed carefully to determine whether it’s related to google-style-guide. Once confirmed, the necessary fixes should be implemented accordingly. This effort is already underway, and I’ve contributed by helping resolve/close the following issues from the list so far.

- 1
- 2
- 3
- 4
- 5

Following are some of the highlighted issues from the list:

Revaluate-tokens issues

To resolve such issues, we need to thoroughly review the Check mentioned in the issue, along with all the tokens listed. We must identify the kind of code each token is responsible for checking and what kind of violations it flags. Then, we need to verify what google-style guide says about those specific cases. Does the style guide explicitly forbid them, imply they’re discouraged, allow them, or say nothing at all?

We must clearly outline:

- Which tokens are being discussed
- Where in the style guide we’re referencing
- And our final judgment on whether the token's behavior aligns with the style guide

Once a consensus is reached:

- If a token *does* follow the style guide, we should add it to the google-config and write test cases to show that compliance.

- If it *doesn't* align with the style guide, we must update the original issue with a comment explaining why that token can't be added to the google-config.

1. **IllegalTokenText**

```
<module name="IllegalTokenText">
 <property name="tokens" value="STRING_LITERAL, CHAR_LITERAL"/>
 <property name="format" value="\\u00(09|0(a|A)|0(c|C)|...
       0(d|D)|22|27|5(C|c))|\\(0(10|11|12|14|15|42|47)|134)"/>
 <property name="message"
          value="Consider using special escape sequence instead
of octal value or Unicode escaped value."/>
</module>
```

Currently we have a few tokens that have text content, such as `TEXT_BLOCK_CONTENT` and `STRING_TEMPLATE_CONTENT`, that must be checked and should be added to the config.

2. **AnnotationLocationCheck**

```
<module name="AnnotationLocation">
     <property name="id" value="AnnotationLocationMostCases"/>
     <property name="tokens"value="CLASS_DEF, INTERFACE_DEF,
ENUM_DEF, METHOD_DEF, CTOR_DEF, RECORD_DEF, COMPACT_CTOR_DEF"/>

</module>
```

Following tokens must be checked:
`TYPECAST`, `DOT`, `TYPE_ARGUMENT`, `LITERAL_NEW`, `LITERAL_THROWS`, `IMPLEMENTS_CLAUSE`, `CLASS_DEF`,

`CTOR_DEF`, `ENUM_DEF`, `INTERFACE_DEF`, `METHOD_DEF`, `VARIABLE_DEF`

3. **EmptyBlockCheck**
   After PR, This check is no longer in use in google-config so we must close this issue.

RightCurlyAloneOrEmpty Check:

This new Check is required to avoid inlined suppression and once properly implemented, it will be a successor to **RightCurly** Check completely in `google_checks.xml`. It is still stuck in designing phases and according to this discussion, it is expected to be more than one Check.

To give an idea of the problem, according to google-style:

> *An empty block or block-like construct may be in K & R style (as described in Section 4.1.2). Alternatively, it may be closed immediately after it is opened, with no characters or line break in between ({}), unless it is part of a multi-block statement (one that directly contains multiple blocks: if/else or try/catch/finally).*

So code in the following example is valid according to config

```
public class RightCurly {
  public RightCurly() {}
}
```

But RightCurly check with `option` property set to `alone` logs violation on this code:

```
'}' at column 24 should be alone on a line.
[RightCurlyAlone]
```

To overcome this problem, we use the following suppression in the config

```
<module name="SuppressionXpathSingleFilter">
     <!-- suppression is required till
https://github.com/checkstyle/checkstyle/issues/7541 -->
     <property name="id" value="RightCurlyAlone"/>
     <property name="query"
value="//RCURLY[parent::SLIST[count(./*)=1]
                                        or
preceding-sibling::*[last()][self::LCURLY]]"/>
   </module>
```

The above suppress is not perfect and suppresses too much. For example, the following code is not correct but it escapes because of the above suppression.

```
public @interface TestAnnotation2 {
    String someValue(); }  //it should be violation here

 enum TestEnum2 {
    SOME_VALUE; } // it should be violation here
}
```

Here's another case:
There should be a violation on the following code, but this too escapes the RightCurly Check because of the suppression

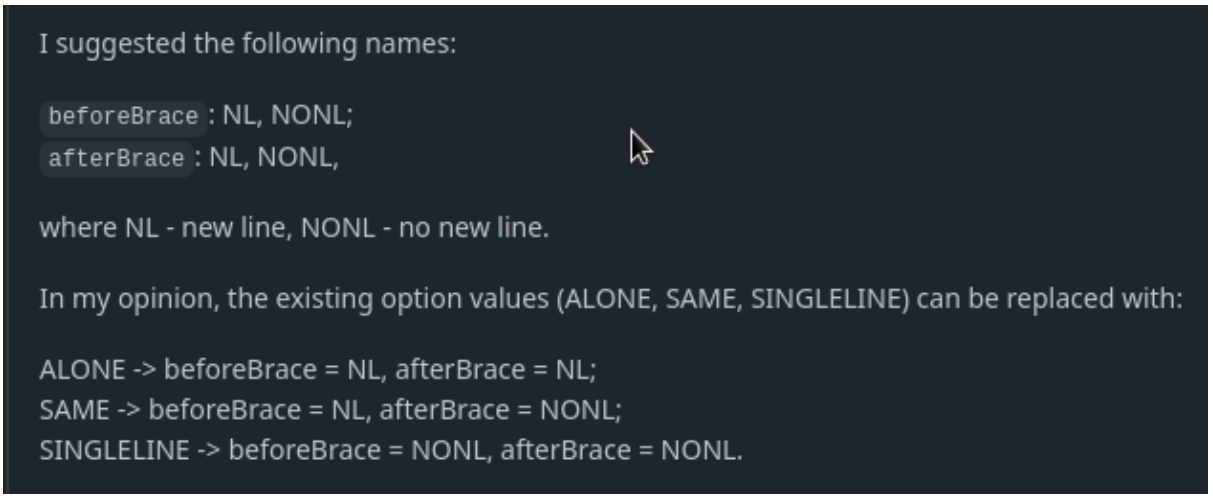
```
   if (a == 2) {
   } else {} // expected violation, part of a multi-block
statement
```

So to make it log violations, we have yet another suppression.

```
<module name="RegexpSinglelineJava">
     <property name="format" value="\{[ ]+\}"/>
     <property name="message" value="Empty blocks should have no
spaces. Empty blocks
                                     may only be represented as '{}'
when not part of a
                                     multi-block statement (4.1.3)"/>
</module>
```

A new right-curly check should help us get rid of all these suppression.

For this check, I feel like we've still not reached a consensus over whether we should make a new check for this, improve it incrementally over time and then replace it with RightCurly check in google-config or refactor the existing check with new properties with names to avoid confusion.



During the bonding period, I will look more into the design, discuss the implementation and how we can learn from our past mistakes in the new check with maintainers.

Other issues

There are many unapproved issues with different labels like new feature, bug, javadoc, miscellaneous, etc. There are issues with incomplete PRs for example **EnumTrailingComma** Check is almost finished but the original contributor left it incomplete. During the summer, we must review all of these and approve/close or fix them.

Timeline

Weeks	Start date	End date	Tasks to be completed
-	9 May	1 June	Community Bonding period: Doubts clearing Review issues from the list and determine their category
1	June 2	June 8	Resolving blue-tick issues
2	June 9	June 15	Reevaluate token issues
3	June 16	June 22	Finding projects for regression testing and making CI workflows
4	June 23	June 29	New checks implementation: right curly
5	June 30	July 6	Keep working new check implementations
6	July 7	July 13	helping guava implement check style Keep working on things from previous weeks
7	July 14	July 20	Keep working on things from previous weeks
8	July 21	July 27	Fixing bugs and reviewing abandoned PRs
9	July 28	August 3	Other issues: fixing abandoned PRs and issues
10	August 4	August 10	Other issues: fixing bugs for different labels
11	August 11	August 17	Documentation changes and updating coverage page
12	August 18	August 24	Left over work, tests and preparing final report

During the community bonding period, I need to get clarification on all the doubts presented in the proposal and since this project involves more of fixing stuff than implementing new things, I will make sure at the end of the project we would have improved our google-style implementations by a magnitude.

I'll be able to commit 24-28 hours per week on this project during the entire duration of the project. I'll have my end-semester exams in May, but those will be covered in the community bonding period.

CONTRIBUTIONS IN Checkstyle

- **Merged PRs**
- **All PRs** (including failed and opened PRs)
- **Raised issues**

Why Me:

I raised my first PR in September 2024 and have been contributing to Checkstyle ever since. From the beginning, I've focused on tackling complex issues and making meaningful contributions. Over this time, I've worked on several code refactors and bug fixes—including indentation-related issues and a grammar-related issue, both of which are among the most complex parts of the codebase. This experience has given me a strong understanding of the project, and I'm confident that I can navigate the codebase smoothly during the summer.

AFTER GSOC:

I'm especially interested in grammar and ANTLR-related issues. By the end of the GSoC period, I'll have a much deeper understanding of the codebase and coding practices, which will better prepare me to start tackling grammar-related issues.

I will also help newcomers to get onboarded and pass down my learnings to them.

THANK-YOU