Google Summer of Code

checkstyle

Project Name: OpenJDK Code convention coverage

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

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

OpenJdk Code Convention was one of the first guidelines on how to write Java code. OpenJdk Code Convention is marked as outdated(last updated in **1999**) but best practices described there do not have an expiration date. New OpenJDK Java Style Guidelines is close to the final version and most likely will be successor of the OpenJdk Code Convention. But there are a number of projects in Apache that still follow OpenJdk rules, so both configurations are in need by the community.

OpenJdk Code Convention is already partly covered by Checkstyle, known as Sun Code Convention. `sun_checks.xml` is the config file which implements rules of now-outdated OpenJdk code conventions..

The goal of this project is to implement rules described in new OpenJdk Java Style Guidelines. A New config file needs to be created, missed checks and properties need to be created and bugs need to be fixed. At the end, New page "New OpenJDK's Java Style Checkstyle Coverage" needs to be created.

We also need to review the old guidelines in detail to ensure full coverage and complete the currently-incomplete `sun_checks.xml` file and update the coverage page.

Deliverables

- `openjdk_checks.xml` config file with all checks that are required for coverage.
- "OpenJDK's Java Style Checkstyle Coverage" page at checkstyle's website that explains how each paragraph in style guide is covered by Checkstyle
- Update the sun-style config and coverage page.

New OpenJdk Java Code Style

Java Source Files

> *There may be no trailing white space at the end of a line.*

Checkstyle covers this rule through **RegexpSingleLine** Check.

Special Characters

> *`\'`, `\"`, `\\`, `\t`, `\b`, `\r`, `\f`, and `\n` should be preferred over corresponding octal (e.g. `\047`) or Unicode (e.g. `\u0027`) escaped characters.*

This rule is partially covered as Checkstyle avoids through **AvoidEscapedUnicodeCharacters** Check.
We currently do not cover Octal escaped characters. An issue has been raised on this.

Formatting

> *A `.java` source file should be structured as follows:*

- *The copyright notice*
- *Package declaration*
- *Import statements*
- *Class declaration*

Checkstyle currently does not validate this rule.

> *There may be only one top level class declaration per file.*

Checkstyle already covers this rule through **OneTopLevelClass** Check but it is not part of the existing sun_checks config.

Copyright notice

> *All source files should have an appropriate copyright notice at the top of the file.*

> *For files under Oracle copyright, the copyright notice must follow the standard wording and format. In particular the first two lines should be*

```
/*
 * Copyright (c) 2011, Oracle and/or its affiliates. All rights
reserved.
```

or

```
/*
 * Copyright (c) 2011, 2015, Oracle and/or its affiliates. All rights
reserved.
```

Checkstyle currently does not validate the format of copyright notice. We can easily support this through regex expressions.

Package declaration

> *The package declaration should not be line wrapped, regardless of whether it exceeds the recommended maximum length of a line.*

Checkstyle already covers this rule through **NoLineWrap** Check**.**

Import statements

> Import statements should be sorted…

a. …primarily by non-static / static with non-static imports first.
b. …secondarily by package origin according to the following order
    - `java` packages
    - `javax` packages

        - external packages (e.g. `org.xml`)
        - internal packages (e.g. `com.sun`)
    c. …tertiary by package and class identifier in lexicographical order

Checkstyle covers this rule through **CustomImportOrder** Check. With the help of this check, we can define custom order of import statements

> Import statements should not be line wrapped, regardless of whether it exceeds the recommended maximum length of a line.

Checkstyle already covers this rule through **NoLineWrap** Check.

> *No unused imports should be present.*

Checkstyle already covers this rule through **UnusedImports** Check.

Wildcard Imports

> *Wildcard imports should in general not be used.*

> *When importing a large number of closely-related classes (such as implementing a visitor over a tree with dozens of distinct “node” classes), a wildcard import may be used.*

Checkstyle already covers the above two rules through **AvoidStarImport** Check.

For the second rule, AvoidStarImport’s `allowClassImports` property can be used to specify closely-related classes.

> *In any case, no more than one wildcard import per file should be used.*

Currently, there is no way to check how many wildcard-imports are used in a file. For this rule, either a new check could be created or AvoidStarImport Check could be expanded to allow X number of star imports per file.

Class Structure

> *The recommended order of class members is the following:*

- *Fields*
- *Constructors*
- *Factory methods*
- *Other Methods*

Checkstyle already covers this rule through **DeclarationOrder** Check.

> *Related fields should be grouped together. Ordering fields primarily according to access modifiers or identifier is not required. The same applies to methods.*

I don't understand this properly. I will discuss this with mentors.

> *Nested types are put at the top of the class or right before it is first used.*

Nested types include `class`, `enum`, `interface` or `record`. No check exists for this rule.

InnerTypeLast Check has somewhat similar functionality as it requires nested types to be declared at the bottom in the class.

If a new check is needed to be made for this rule, it would be more flexible than **InnerTypeLast** Check as according to the rule, nested types can be declared anywhere close to its first use, whereas the current check requires them only at the bottom. We have other checks to ensure other class element's orders are not affected due to this. They are discussed below in their relevant sections.

Order of Constructors and Overloaded Methods

> *Constructors and overloaded methods should be grouped together by functionality and ordered with increasing arity. This implies that delegation among these constructs flows downward in the code.*

The rule states that methods with fewer parameters (**lower arity**) should come first, followed by methods with more parameters.

Example code in case of constructors :

```
/** Constructor with the lowest arity (1 parameter) */
public Person(String name) {
     ...
}
/** Constructor with 2 parameters */
public Person(String name, int age) {
     ...
}
/** Constructor with the highest arity (3 parameters) */
public Person(String name, int age, String city) {
    ...
}
```

Example code in case of overloaded methods :

```
/** Overloaded method with lower arity */
public void greet() {
     ...
}
/** Overloaded method with higher arity */
public void greet(String message) {
     ...
}
```

No Checks currently exist for this rule. We need to create two new checks for each one. **MethodsArityOrderCheck** could be one of the possible names.

> *Constructors and overloaded methods should not be split apart by other members.*

Checkstyle already covers this rule through **ConstructorsDeclarationGrouping** and **OverloadMethodsDeclarationOrder** Checks.

Modifiers

> *Modifiers should go in the following order*

- *Access modifier* (`public` / `private` / `protected`)
- `abstract`
- `static`
- `final`
- `transient`
- `volatile`
- `default`
- `synchronized`
- `native`
- `Strictfp`

Checkstyle already covers the above two rules through **ModifierOrder**.

> *Modifiers should not be written out when they are implicit. For example, interface methods should neither be declared public nor abstract, and nested enums and interfaces should not be declared static.*

Checkstyle already covers the above two rules through **RedundantModifier** Check.

> *Method parameters and local variables **should not be declared final** unless it improves readability or documents an actual design decision (Motivation: Although method parameters should typically not be mutated, consistently marking all parameters in every method as `final` is an exaggeration.)*

Checkstyle has **FinalParameters** and **FinalLocalVariable** checks to ensure method parameters and local variables are declared final. A new Check, **illegalFinal** maybe, could be created to do the opposite–making sure parameters are local variables are not final.

There is a rule related to above parameters defined in “Programming practices” section:

> *Mutating method parameters is discouraged. This is because parameters are typically associated with the input values and by mutating these the code may be harder to understand, especially when only looking at parts of the method below the mutation.*

We need to find a way to ensure parameters are not being modified, without them being declared `final`.

> *Fields should be declared final unless there is a compelling reason to make them mutable.*

Covered through ImmutabilityTest

Braces

> *Opening braces should be put on the end of the current line rather than on a line by its own.*

> *There should be a new line in front of a closing brace unless the block is empty (see Short Forms below)*

> *The `else`, `catch` and the `while` keyword in `do`…`while` loops go on the same line as the closing brace of the preceding block.*

Checkstyle already covers the above rules through **LeftCurly** and **RightCurly** Checks.

> *Braces are recommended even where the language makes them optional, such as single-line if and loop bodies.*

a. *If a block spans more than one line (including comments) it must have braces.*
b. *If one of the blocks in a if / else statement has braces, the other block must too.*
c. *If the block comes last in an enclosing block, it must have braces.*

NeedBraces check makes sure braces are not omitted by the user. So the above conditions are covered by Checkstyle.

Short Forms

This is a small section in the guideline about short forms. It is obvious from the name that if the code for an enum or method is trivial, we can just finish them in a single line.

For Example:

```
enum Response { YES, NO, MAYBE }
public boolean isReference() { return true; }
```

It does not seem like an important rule to me. I think we can ignore this until we are done with other important rules.

Indentation

> *Indentation level is four spaces.*

This means that the indentation inside the code block must be 4. This rule is covered as the `Basic offset` property of the **Indentation** check has default value of 4.

> *Empty lines must not be indented. (This is implied by the no trailing white space rule)*

This rule is covered through **RegexpSingleLine** Check.

> *case lines should be indented with four spaces, and statements within the case should be indented with another four spaces.*

Example for the rule:

```
switch (var) {
    case TWO:
        setChoice("two");
        break;
    case THREE:
        setChoice("three");
        break;
    default:
        throw new IllegalArgumentException();
}
```

This rule is covered as default value of `caseIndent` property of Indentation check is 4.

Wrapping Lines (how to indent continuous lines)

> *Source code and comments should generally not exceed 80 characters per line and rarely if ever exceed 100 characters per line, including indentation.*

I don't understand what it is about. Does it mean even though 80 characters per line is the limit, it's okay if in some cases the line goes beyond even 100 characters as long as it improves readability ? There is more on this below.

> *There should be at most 1 statement per line.*

Checkstyle already covers this rule through **OneStatementPerLine** Check.

> *URLs or example commands should not be wrapped.*

> *A continuation line should be indented in one of the following four ways:*

> > ***Variant 1:** With 8 extra spaces relative to the indentation of the previous line.*
> >
> > ***Variant 2:** With 8 extra spaces relative to the starting column of the wrapped expression.*
> >
> > ***Variant 3:** Aligned with previous sibling expression (as long as it is clear that it's a continuation line)*
> >
> > ***Variant 4:** Aligned with previous method call in a chained expression.*

Checkstyle covers this rule by specifying IndentationCheck's `lineWrappingIndentation` property to 8.

Wrapping Method Declarations

> *Method declarations can be formatted by listing the arguments vertically, or by a new line and +8 extra spaces.*
> *(if aligning the parameters vertically, don't put two parameters on one line)*

For Example:

```
int Vertically(String aString,
               int num,
               long aLong,
               Set<Number> aSet,
               double aDouble) {
    ...
}
```

First part of the rule only talks of parameters being list vertically and does not specify the indentation of parameters in new lines so we need to interpret this in our own way.
It seems like a complex case to handle to me. I require clarification on this rule from mentors.

> *If a `throws` clause needs to be wrapped, put the line break in front of the throws clause and make sure it stands out from the argument list, either by indenting +8 relative to the function declaration, or +8 relative to the previous line.*

To put it simply, the Indentation value of arguments and `throws` must be different to make `throws` stand out. Example:

```
int someMethod1(String aString, List<Integer> aList, //indentation:0
        Map<String, String> aMap, int anInt)         //indentation:8
                throws IllegalArgumentException {    //indentation: 16
    ...
}

int someMethod2(String aString,                                 //indentation:0
                List<Map<Integer, Buffer>> aListOfMaps,         //indentation:16
                Map<String, String> aMap)                       //indentation:16
        throws IllegalArgumentException {                       //indentation: 8
    ...
}
```

Checkstyle currently doesn't check such conditions. We only do the simple indentation check for `throws.`

```java
/**
 * Check the indentation level of the throws clause.
 */
private void checkThrows() {  1 usage   rnveach
    final DetailAST throwsAst = getMainAst().findFirstToken(TokenTypes.LITERAL_THROWS);

    if (throwsAst != null) {
        checkWrappingIndentation(throwsAst, throwsAst.getNextSibling(), getIndentCheck()
                .getThrowsIndent(), getLineStart(getMethodDefLineStart(getMainAst())),
                !isOnStartOfLine(throwsAst));
    }
}
```

We might need to create a **ThrowsHandler** just like other indentation check indentation handlers to validate for more complex cases.

Wrapping Class Declarations

> *A class header should not be wrapped unless it approaches the maximum column limit. If it does, it may be wrapped before `extends` and/or `implements` keywords.*

If it is a class with a long name, then it is possible to wrap the class header. Trying to wrap it without exceeding the line limit should trigger violation of this rule. Checkstyle currently does not have any check to check if wrap within line-length limit.

> *Declarations of type parameters may, if necessary, be wrapped the same way as method arguments.*
Covered in above sections.

Wrapping Expression

> *Break before operators.*
> *Break before the `.` in chained method calls.*

For Example:

```
aMethodCall(withMany(arguments, that, needs),
        to(be, (wrapped - to) * avoid / veryLong - lines));
```

Checkstyle covers above two rules through **OperatorWrap** Check.

Whether to break line after opening brace ?
We also allow line breaks after an opening brace (in case of chained method calls. Take a look at following cases:

```
void foo() {
      new String()
          .substring(
              0, 100 )
```

```
toString().contains(//indent:8 exp:8
       new String(//indent:12 exp:12
               "a" //indent:20 exp:20
       )//indent:12 exp:12
```

Since It isn't mentioned in the documentation if such line breaks are allowed or not, I need to discuss nuisance with mentors.

Confusion over column Length:
> *The character limit must be judged on a case by case basis. What really matters is the semantical “density” and readability of the line. Making lines gratuitously long makes them hard to read; similarly, making “heroic attempts” to fit them into 80 columns can also make them hard to read. The flexibility outlined here aims to enable developers to avoid these extremes, not to maximize use of monitor real-estate.*

Simply put, breaking a line just to stay within the 80-character limit can sometimes make the code harder to read. For Example:

```
Error e = isTypeParam
        ? Errors.InvalidRepeatableAnnotationNotApplicable(
                targetContainerType, on)
```

```
        : Errors.InvalidRepeatableAnnotationInContext(
                targetContainerType);
```

In such cases, some flexibility should be allowed. The following example should be considered valid, even if it exceeds the line limit, as it improves readability.

```
Error e = isTypeParam
        ? Errors.InvalidRepeatableAnnotationNotApplicable(targetContainerType, on)
        :Errors.InvalidRepeatableAnnotationInContext(targetContainerType);
```

I'm not sure about the feasibility of this rule. I need to discuss it properly with mentors.
I think dealing with rules related to Indentation Check is going to be the most difficult part of this project Because it is one of the most buggy checks in the codebase. Just look at the number of bugs that get reported on this check. This is due to the way check is designed.

Whitespace

Vertical Whitespace

> *A single blank line should be used to separate…*

a. *Copyright notice*
b. *Package declaration*
c. *Class declarations*
d. *Constructors*
e. *Methods*
f. *Static initializers*
g. *Instance initializers*

…and may be used to separate logical groups of

h. *import statements*
i. *Fields*
j. *statement*

Checkstyle already covers the above two rules through **EmptyLineSeparator** Check.

> *Multiple consecutive blank lines should only be used to separate groups of related members and not as the standard inter-member spacing.*

We need to discuss what *groups of related members* might mean. Related by functionality, return type or number of parameters.

EmptyLineSeparator Check has some properties on multiple blank lines such as `allowMultipleEmptyLines`, `allowMultipleEmptyLinesInsideClassMembers`. We can play around with these or make a new property once we are sure what needs to be done.

Horizontal Whitespace

> *A single space should be used…*

a. *To separate keywords from neighboring opening or closing brackets and braces*
b. *Before and after all binary operators and operator like symbols such as arrows in lambda expressions and the colon in enhanced for loops (but not before the colon of a label)*
c. *After `//` that starts a comment.*
d. *After commas separating arguments and semicolons separating the parts of a for loop.*
e. *After the closing parenthesis of a cast.*

Checkstyle covers all the above cases through **WhitespaceAround** and **WhitespaceAfter** Checks.

> *In variable declarations it is not recommended to align types and variables.*

For Example:

```
int    firstInt;
String secondString;
char   thirdChar;
long   fourLong;
```

Checkstyle covers this rule through **SingleSpaceSeparator** Check which ensures that there is only one whitespace between datatype and variable name.

Variable Declarations

> *One variable per declaration (and at most one declaration per line)*

Checkstyle already covers this rule through **MultipleVariableDeclarations** Check.

> *Square brackets of arrays should be at the type (`String[] args`) and not on the variable (`String args[]`).*

Checkstyle already covers this rule through **ArrayTypeStyle**.

> *Declare a local variable right before it is first used, and initialize it as close to the declaration as possible.*

Checkstyle already covers this rule through **VariableDeclarationUsageDistance**.

Annotations

Annotations were not part of the original Java language. They were introduced later in Java in 2005. Following rules on annotation are only present in the updated guidelines.

> *Declaration annotations should be put on a separate line from the declaration being annotated.*

> *Few/short annotations annotating a single-line method may however be put on the same line as the method if it improves readability.*

Currently there is no check to validate the above rules. Based on this discussion, these are complicated cases and need a lot of experimenting to create Check(s) to adhere to above rules. How to handle the above rules will be figured out during the bonding period.

> *Either all annotations should be put on the same line or each annotation should be put on a separate line.*

It states that some annotations are on the same line, while others on separate lines cause inconsistent formatting and make code hard to read. Example of violation of this rule:

```
@Override @Deprecated
@SuppressWarnings("unchecked")
public void someMethod() {
      ...
}
```

No Check currently exists for this rule. So we need a new Check. **MultipleAnnotationsLocationCheck** could be one of the possible names.

There is a rule related to annotations specified in the Programming practices section:

> *Always use `@Override` where it is possible to do so.*

In Java `@Override` is **optional** for overriding a method, but it is highly recommended because it helps prevent subtle bugs. **MissingOverride** Check has somewhat similar functionality, but it only looks for `@Override` when the `@inheritDoc` javadoc tag is used. Maybe this check could be expanded to make it cover all cases of missing `@Override`.

Lambda Expressions

Lambdas were not part of the original Java language. They were introduced later in Java 8 (2014). Following rules on lambda expression are only present in the updated guideline:

> *Expression lambdas are preferred over single-line block lambdas.*

If a lambda body is only single-line, it's better to use the expression lambda instead. For Example:

```
public static void main1(String[] args) {
     List<String> names = List.of("x", "y", "z");

     // single-line block lambdas
     names.forEach(n -> { System.out.println(n); });
}

public static void main2(String[] args) {
     List<String> names = List.of("x", "y", "z");
     // Expression lambda (concise and readable)
     names.forEach(name -> System.out.println(name));
}
```

No Check currently exists for this rule.

> *Method references should generally be preferred over lambda expressions.*

For bound instance method references, or methods with arity greater than one, a lambda expression may be easier to understand and therefore preferred. Especially if the behavior of the method is not clear from the context.

Bound instance method references means making an object/instance of a class, and using that object/instance to make reference to class methods (Reference is bound to a specific instance). Reference is made using `::` operator.

> *The parameter types should be omitted unless they improve readability.*

Based on this discussion, it might not be possible to cover all above rules because of the limits of checkstyle. These rules require detailed discussions before anything could be finalised.

> *If a lambda expression stretches over more than a few lines, consider creating a method.*

Checkstyle already covers this rule through **LambdaBodyLength** Check.

Redundant Parentheses

Checkstyle has **UnnecessaryParentheses** Check to check for redundant parentheses, but it's not part of the sun_checks config.

> *The entire expression following a return keyword must not be surrounded by parentheses.*

This rule is covered by UnnecessaryParentheses check.

In the previous version of the OpenJDK guidelines, there were still cases where parenthesizing the return expression was considered valid. According to previous version:

> *A `return` statement with a value should not use parentheses unless they make the return value more obvious in some way.*

For example:

```
return (size ? size : defaultSize);
```

But the new rule makes it absolutely clear that the entire expression following a return is not to be parentheses-ed.

> *Redundant grouping parentheses (i.e. parentheses that do not affect evaluation) may be used if they improve readability.*

I need further clarification on such rules.[1]

---

[1] Something I've learned from contributing to checkstyle over the past few months is that the choice of parentheses is very subjective. Some users may prefer to use them everywhere, yet some would use them only occasionally or whenever expression becomes complex. So it is a difficult task to turn words like "*if improves readability*" into specific checks/rules.

> *Redundant grouping parentheses should typically be left out in shorter expressions involving common operators but included in longer expressions or expressions involving operators whose precedence and associativity is unclear without parentheses.*

For shorter expressions, UnnecessaryParentheses check already covers this rule. it registers violations for all cases in the following examples:

```
// short-expressions: Redundant parentheses
i = ((int) arg2);
int square = (a * b);
return (square);
```

> Ternary e*xpressions with non-trivial conditions belong to the latter (longer expressions or expressions involving operators whose precedence and associativity is unclear without parentheses.*

Example with non-trivial conditions(parentheses required)

```
String cmp = (flag1 != flag2) ? "not equal" : "equal";
```

Example with trivial condition(parentheses not required)

```
String cmp2 = size ? size : defaultSize; // simple condition,
//                                       parentheses not required
```

UnnecessaryParentheses check only validates if parentheses are unnecessary(where they shouldn't belong), and does not check if they are mandatory(where they must belong). So it does not log violation if parentheses around (`flag1 != flag2`) get removed in example 1.

I need to discuss how to workaround this limitation with maintainers. UnnecessaryParentheses Check has some design flaws. So It might not be ideal to try to modify this check.

Literals

> `long` *literals should use the upper case letter* `L` *suffix.*

This rule is implemented with **UpperEllCheck**.

> *Hexadecimal literals should use upper case letters `A-F`.*

> *All other numerical prefixes, infixes, and suffixes should use lowercase letters.*

Example of violation of above two rules:

```
int i = 0X123 + 0xabc; // expected 0xABC
byte b = 0B1010;       // expected 0b1010
float f1 = 1 / 5432F;  // expected 5432f
float f2 = 0.123E4f;   // expected 0.123e5f
double d1 = 1 / 5432D; // 5432d
double d2 = 0x1.3P2;   // 0x1.3p2
```

No Checks currently exist for this rule. Based on discussion, We need to create two new checks for this to cover each of the above rules. An issue is raised and I've already started working on it .

Javadoc

> *Start longer comments with a short summarizing sentence since Javadoc includes this in the method summary table.*

> *Prefer inline tags (such as `{@code …}` and `{@link …}` etc) over corresponding HTML tags (`<code>…</code>`, `<a href="…">…</a>` etc).*

No Check currently exists to validate for this rule.

> *Use `<p>` to separate paragraphs (closing `</p>` tags are not needed and should not be used)*

Currently checkstyle is not valid for this rule.
After this Pr, **JavadocParagraph** Check now has the ability to detect closing paragraph tags too. This check could be expanded to have a new property– `paragraphClosingTagAllowed`–to check for `</p>` tags.

Naming

> *Avoid hiding/shadowing methods, variables and type variables in outer scopes.*

Checkstyle already covers this rule through **HiddenField**.

> *Let the verbosity of the name correlate to the size of the scope. (For instance, use descriptive names for fields of large classes and brief names for local short-lived variables.)*

> *When naming public static members, let the identifier be self descriptive if you believe they will be statically imported.*

I'm not sure about the above two rules. There is no way to check if variable names are *self-descriptive enough* based on their surroundings.

Package Names

> *Package names should be all lower case without underscores or other special characters.*

PackageName is a general purpose check to validate the package name against a specified format.

> *Don't use plural form. Follow the convention of the standard API which uses for instance* `java.lang.annotation` *and not* `java.lang.annotations`*.*

I'm skipping this rule for now, and I'll go over them during the community bonding phase.

Class, Interface and Enum Names

> *Class and enum names should typically be nouns.*

Checkstyle cannot determine whether a name is a noun, an adjective, or any other grammatical category.

Google style guide also has a similar rule: "*Class names are typically nouns or noun phrases*" and **TypeName** Check is used there to validate the above rule.

So I think we can have our rule covered as well through that check.

> *Interface names should typically be nouns or adjectives ending with …able. Use mixed case with the first letter in each word in upper case.*

Covered.

> *Use whole words and avoid using abbreviations unless the abbreviation is more widely used than the long form.*

> *Format an abbreviation as a word if it is part of a longer class name.*

I'm not sure what these two rules mean. Maybe we can ignore them until the end of the program.

Method Names

> *Method names should typically be verbs or other descriptions of actions.*

Covered above.

> *Use mixed case with the first letter in lower case.*

I'm assuming mixed case here means lowerCamelCase. Checkstyle covers this rule through **MethodName** Check.

Variables

> *Variable names should be in mixed case with the first letter in lower case.*

> Though it is not said explicitly, Variable names should not contain an underscore.

Checkstyle covers this rule.

Type Variables

> *For simple cases where there are few type variables involved use a single upper case letter. For example:*

```
// single-uppercase letter type variable names for simple and
obvious case
// good
interface SpecialMap<K, V> extends Map<K, V> {
    ...
}
```

```
// long type variable names for simple and obvious case
// not good
interface SpecialMap<Key, Value> extends Map<Key, Value> {
    ...
}
```

> *If one letter is more descriptive than another (such as K and V for keys and values in maps or R for a function return type) use that, otherwise use T.*

> *For complex cases where single letter type variables become confusing, use longer names written in all capital letters and use underscore (_) to separate words.*

It's not possible to validate such rules because they are very context-based. For example: there is no way to check if K is more descriptive than R. It is also impossible to distinguish between a simple *case* and a complex *case.* For Checkstyle, every case is created equal.

Constants

> *Constants (static final fields whose content is immutable, by language rules or by convention) should be named with all capital letters and underscore (_) to separate words.*

Checkstyle already covers the above two rules through **ConstantName**.

Programming Practices

> *Always use `@Override` where it is possible to do so.*
> *Mutating method parameters is discouraged. This is because parameters are typically associated with the input values and by mutating these the code may be harder to understand, especially when only looking at parts of the method below the mutation.*

Already covered in above sections.

> *Document any non-obvious pre and post conditions (also for private methods). In particular, all parameters / return values are assumed to be non-null unless explicitly stated otherwise.*

Requirements are not possible to check by Checkstyle.

> *Avoid checking in "`TODO`" comments. If the missing code doesn't affect correctness or performance, skip the comment, otherwise file an issue for it.*

Checkstyle already covers this rule through **TodoComment** Check.

It's a general-purpose check for matching patterns in Java comments. we can customize the error message in the config to enforce the rule more precisely.

> Avoid checking in dead code.

Not possible as Checkstyle cannot recognize dead code.

> Static mutable state should be used judiciously. In particular enums should not have mutable state, and any public static variable must also be final. Make fields final unless they have a need to be mutable.

Qualify static method invocations and fields with class identifiers and not variable identifiers.

I'm not sure about the above two rules. Better to understand them from mentors.

> *Fall through on non-empty cases in switch statements should be documented, typically with a "`// fall through`" comment.*

FallThrough Check could be used here with a custom violation message to warn users about "missing fall through comment" . According to check's description:

" Checks for fall-through in `switch` statements. Finds locations where a `case` **contains** Java code but lacks a `break`, `return`, `yield`, `throw` or `continue` statement."

Message might be something like, "Expected // fall through comment.".

> *Keep methods short. Consider splitting up long methods into smaller ones to limit variable scopes and simplify structure of the code.*

Checkstyle already covers this rule through **MethodLength** Check.

Commenting Code

> *First and foremost, try to make the code simple enough that it's self explanatory. While comments explaining the code are good, not having to explain the code is better.*

> *Avoid comments that run the risk of getting out of sync as the code evolves. (If a comment runs the risk of getting out of sync, it's often a sign that it comments how the code works rather than what the code achieves.)*

> *Comments should be grammatically correct and follow general recommendations of technical writing.*

Above requirements are not possible to check by Checkstyle.

> *For single line comments, use end-of-line comments (`//`) otherwise use multiline comments (`/* … */`).*

Checkstyle currently does not cover this rule.

> *Small well documented methods are preferred over longer methods with comments in the body of the method.*

Both parts of the rule are covered and already discussed above.

> *Don’t check in code that’s commented out.*
> *IDE/tool-specific comments should be avoided wherever possible, especially when there are reasonable alternatives, such as external settings files, etc.*

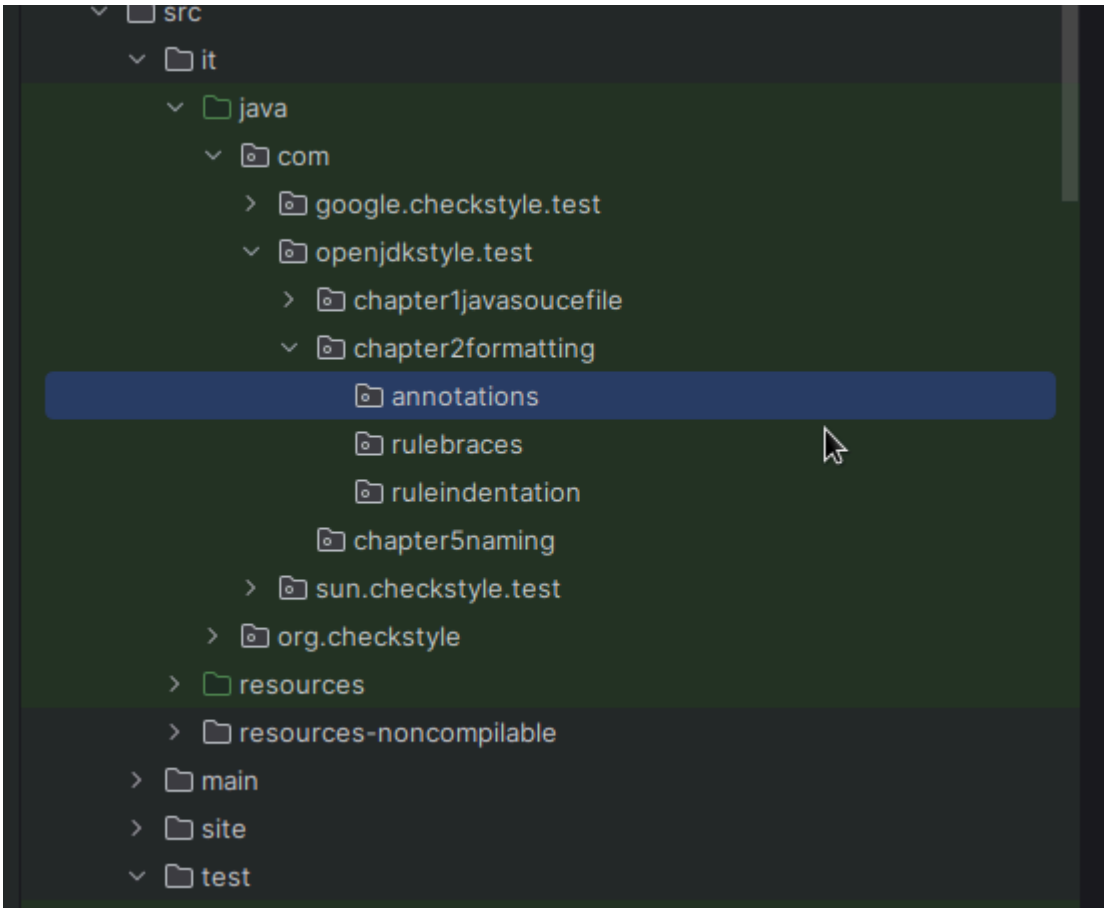
Because of time constraints, I’m leaving these rules out for now. I’ll revisit and discuss them during the community bonding period.

Testing and Documents

Chapter-wise testing

It involves dividing all the rules in the guidelines into chapters and subchapters. All the rules specified in a particular subchapter can be tested with a single java test file. This is a better approach than using different test files for different checks.
Similar type of refactoring was done for Google-style implementation in last year's Gsoc. Here. We can use the same idea for this project too.

Folder structure might look something like below:



Config file

`openjdk_checks.xml` config file will be the final product of this project. It will contain all the new checks to-be-created, checks present in `sun_checks.xml` file and the following checks, for rules we have coverage but, missing from `sun_checks.xml`:

- OneTopLevelClassCheck
- NoLineWrapCheck
- DeclarationOrderCheck
- ConstructorsDeclarationGroupingCheck
- OverloadMethodsDeclarationOrderCheck
- OneStatementPerLineCheck
- IndentationCheck
- VariableDeclarationUsageDistanceCheck
- UnnecessaryParenthesesCheck
- SingleSpaceSeparator

New OpenJDK's Java Style Checkstyle Coverage Page

This page will contain list of rules with coverage details just like Google-style coverage page.

Sun style

Sun_checks.xml config is currently incomplete. Since it is based on now-outdated OpenJDK guidelines and Checkstyle already covers most of those rules, only the tedious task of adding modules to the config file is left.
While analysing new guideline rules, we can cross-check them with rules in the old one just to ensure full coverage.

I also need to discuss whether the checks related to annotation and lambda expressions should also be added in sun config as there is

Coverage Page for Sun-Style also needs to be updated.

Timeline

Weeks	Start date	End date	Tasks to be completed
-	9 May	1 June	Community Bonding period: Doubts clearing Deciding what to do and what to leave out
1	June 2	June 8	Class structure section
2	June 9	June 15	Modifier section
3	June 16	June 22	Lambda section or annotation section
4	June 23	June 29	Redundant parentheses section (not sure if lambda/annotations could be handled in a single week)
5	June 30	July 6	Indentation
6	July 7	July 13	Wrapping lines (also part of Indentation)
7	July 14	July 20	Javadoc section(not sure how much javadoc check are going to take, because they are complex in nature)
8	July 21	July 27	Copyright section and Naming section (and other similar rules which deal with regex.)
9	July 28	August 3	Rules remaining in Formatting section
10	August 4	August 10	Programming practices section
11	August 11	August 17	Commenting code section
12	August 18	August 24	Left over work, tests and preparing final report Deciding if coding period to be extended

During the community bonding period, I will get clarification on all the doubts presented in the proposal and since this is a very very big project, we need to prioritize important sections from every rule and work on them first and leave the trivial parts until the end of the coding period.

I'll be able to commit around 25-30 hours per week to this project throughout the entire duration of the project. This is a very big project, and I'm okay if we require extending the coding period. I'll have my end-semester exams in May, but those will be covered in the community bonding period so issues on the school side.

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