P3

Programming Assignment 3: Program Linting

Background

Note: You do not need to read this section to complete the assessment, but it provides some helpful context that may make the assessment easier to understand.

What is *Linting*?

We've all encountered bugs when writing code, and explored strategies to debug these errors. Compilers help greatly with syntax errors, and other tools like debuggers help us find functional bugs. However, these tools can't help us with *code quality*; that's where *linting* comes in.

Linters are programs that involve reading over other programs to highlight formatting or some other violations based on some ***opinionated set of style guidelines*** that define how programs should be written.

Some examples are programs called checkstyle for linting Java programs and flake8 for linting Python programs.

Stephen C. Johnson, a computer scientist at Bell Labs coined the term linting in 1978 while dealing with code portability issues

> The term "lint" was derived from
>
> lint

Why does linting matter?

Writing Java code isn't just about writing code that compiles and works. If we wanted code that just got a computer to follow our directions, why not just write it in binary? Truth is, code isn't written for computers; it's written for people.

Programming in the real world is a highly collaborative activity, so it's very important to be able to write code that is easy to work with and understand. Having clean, readable code also makes finding bugs and errors in your code significantly easier. There are numerous code quality guides that define their own set of rules that programmers should follow.

Linters help us simplify this process and enforces a set of style guidelines by highlighting instances of incorrect formatting or other violations.

Who decides the style guidelines?

There isn't "one true style guide" that all programmers follow! Style guides are extremely customizable and rules can be switched in and out easily. These guidelines are somewhat arbitrary (decided on by people based on preference), but organizations use them up to ensure consistency among their codebase.

This assignment will present some rules for you to implement, but these rules are not all-inclusive! You all can come up with other rules that might be important to you as a programmer 😃

Specification

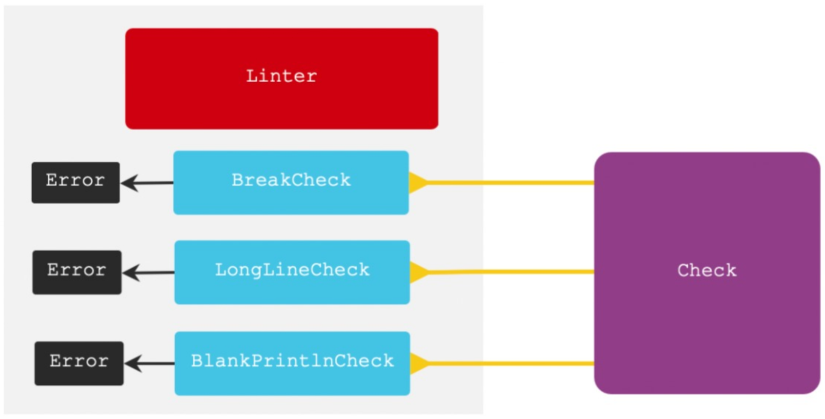
Learning Objectives

By completing this assignment, students will be able to:

- Implement a well-designed Java class to meet a given specification
- Implement correctly encapsulated objects with instance methods and constructors
- Understand relationships between classes and interfaces
- Follow prescribed conventions for code quality, documentation, and readability

1️⃣ Program Behavior

In this assignment, you will implement a **program linter** that analyzes Java files. This linter will be driven by multiple classes that work in tandem to detect errors and present them to the user.



As you read through the descriptions of each class below, you may find it helpful to refer back to the diagram above to understand how these different classes work together.

The program is driven by a class `Linter` that lints a file given a list of checks. You will need to iterate through files line by line and run checks on each line of the Java file.

Like we mentioned in the background section, these checks are completely customizable, and we've come up with 3 such checks: `BreakCheck`, `LongLineCheck`, and `BlankPrintlnCheck` and each check should return an `Error`.

To define these checks in Java, we've defined an interface called `Check` that these checks should implement.

2️⃣ Checker Classes

You will have to implement specific checks to highlight errors found in the source files. We provided an ***interface*** `Check.java` that defines a single method `public Optional<Error> lint(String line, int lineNumber)` . **All the checkers you write should implement this interface and hence, you need to implement the `lint` method**.

For this assignment, you'll implement the following checks:

1. `LongLineCheck`
    - Should return an error (with a custom message) if the given line length is `100` characters or greater
    - Has error code `1`
2. `BreakCheck`
    - Should return an error (with custom message) if the given line contains the `break` keyword **outside of a single line comment** (comments that start with `//`). i.e, we don't care about the word `break` inside comments, and only in the actual java code.
        - Your check should only look for `break` specifically in all-lowercase (so occurrences of "Break" or "BReaK" outside of a single line comment should not be flagged).
        - Note that this check is overly-simplistic in that it might flag some false uses of `break` such as `System.out.println("break");`. You do not need to handle this case specially; you should flag any use of the word break outside of a single line comment.
    - Has error code `2`
3. `BlankPrintlnCheck`
    - Should return an error (with custom message) if the given line contains the pattern
    - System.out.println("")
    - Has error code `3`

All of these should return an Error when one is present with a custom message of your choosing to describe what the error means to the user. If the condition a `Check` is looking for is not present for a line, should return `Optional.empty()`.

3️⃣ Other Classes

For each file below, implement the methods and constructors defined.

`Error.java`

`public Error(int code, int lineNumber, String message)`

Constructs an `Error` given the error code, line number and a message.

`public String toString()`

Returns a String representation of the error with the line number, error code and message. The representation should be formatted as (replace curly braces with the actual values)

(Line: {lineNumber}) has error code {code}
{message}

`public int getLineNumber()`

Returns the line number of this error

`public int getCode()`

Returns the error code of this error

`public String getMessage()`

Returns the message for this error

`Linter.java`

`public Linter(List<Check> checks)`

Constructs a `Linter` given a list of checks to run through.

`public List<Error> lint(String fileName) throws FileNotFoundException`

Lints the given files line-by-line by running through all the checks passed into the constructor. You should first open the file (specified by `fileName`) and read through the file line-by-line.

You can use the `File` and `Scanner` classes to help open and read the file line-by-line. For each line, run through all the checks and add errors if they are present. Returns a list of all errors found.

Development Strategy

We recommend developing classes in the following order:

- `Error.java`
- The classes that implement the `Check` interface
  - `LongLineCheck`
  - `BlankPrintlnCheck`
  - `BreakCheck`
- `Linter`

You will likely be able to debug your work more easily if you test each class individually. Note that you **should** be editing `LinterMain` to test all of your `Check`s. Before submitting, make sure your `LinterMain` is testing all of your `Check`s!

As you work on your code, be sure to think carefully about the **state** (i.e. the fields) necessary for each class. It will be difficult or impossible to achieve the correct behavior if you are not storing the correct state in your objects.

In particular, you will likely need to "remember" values passed to constructors in fields to ensure they are available for later use.

4️⃣ Implementation Details

Notice that to implement these checks, you should implement the `Check.java` interface and write code to detect these errors in the `lint` method. This method has a return type of `Optional<Error>` . `Optional` in Java, is a class that represents optional values (null or an error).

There are a few ways to create `Optional` objects, but you should use the methods described below:

1. `Optional.empty()`: Creates an empty optional object. You should use this method if you **don't** detect any errors in your lint method, to signify that this line is *empty* of errors.
2. `Optional.of(error)`: Creates an optional object with a specific error. You should use this method if you detect any error in your lint method.

In your `Linter` class, you'll need to detect if an error was present after running a check, and get the error itself from the `Optional`. To do this, you can use:

1. `Optional.isPresent()`: Returns `true` if there was an error present and `false` otherwise.
2. `Optional.get()` : Gets the `Error` object from the `Optional`

Look at the sample code below to see how you can use all **4** methods described above!

Sample Check

Below is an example for a checker class that implements a check for bad boolean zen. We want to return an error (with error code `4`) if we see any lines of code that include `== true` or `== false`. For example, lines like: `x == true` or `y == false` would return an error.

```
// Checks if a line contains bad boolean zen
public class BooleanZenCheck implements Check {
    public Optional<Error> lint(String line, int lineNumber) {
        if (line.contains("== true") || line.contains("== false")) {
            return Optional.of(new Error(4, lineNumber, "Line has bad boolean zen"));
        }
        return Optional.empty();
    }
}
```

Sample `Linter`

Below is an example for a `Linter` that runs a sample check (called `check`). Suppose we want to see if there was an error present, we can use the `isPresent()` method, and get the actual `Error` object using the `get()` method.

```
// Code to get line and lineNumber and setup checker (check)
Optional<Error> error = check.lint(line, lineNumber);
if (error.isPresent()) { // error was found
    Error e = error.get(); // get the error from the Optional
    ...
}
```

In your solution, you will likely need to use some *escape sequences* which are specific sequences of characters in Java to represent special characters in `String`s. These include

- `\n` to insert a line break in a `String`
- `\"` to insert a double-quote " in a `String`

5️⃣ Assignment Requirements

Make sure your code follows the Code Quality Guide. In particular, pay attention to the following bullets:

- You should make all of your fields `private` and you should reduce the number of fields only to those that are necessary for solving the problem.
- You should use interface types where appropriate and properly use type parameters when specifying the types of data structures in your solution.
- You should comment your code following Commenting Guide for each file you write. You should write comments with basic info (a header comment at the top of your file), a class comment for every class, and a comment for every method other than `main`.
    - In particular, don't forget to avoid mentioning implementation details (such as private fields) and make sure to comment on any interface(s) your classes implement (when applicable)!
- As a general rule, you have learned everything necessary to implement a solution to an assignment by the time it is released. If you are tempted to use any concepts or tools that have *not* been covered in class, check the **Forbidden Features** section of the Code Quality Guide first to make sure they do not appear there!