CSDS 293
Software Craftsmanship
2025 Spring Semester

Programming Assignment 3

Due at the beginning of your discussion session on

February 3-7, 2025

Reading

In addition to the following topics, the quiz syllabus includes any material covered in the lectures:

- Section 16.1 ("Normal Loop-With-Exit Loops" only) in Code Complete
- Section 16.2 (Except "Enter the loop from one location only", "Checking Endpoints", "Using Loop Variables", and "How Long …") in Code Complete.
- Section 19.6 in Code Complete
- Item 58, 59, and 67 in Effective Java.

Programming

First, make all the changes discussed in your discussion section. Additionally, you should refactor your code to make sure that it adopts the principles covered in the reading assignments. You should modify your code to avoid methods with high McCabe's complexity.

In this programming assignment, you will implement a filtering and matching mechanism to compare n-grams.

Error Handling

As a reminder, the following rules apply to error handling.

Exceptions and Assertions

Error checking will be a topic of a future lecture. For the time being, the following guidelines should be implemented:

- If a method is *visible* outside the package (e.g., public methods in public classes), failed pre-conditions should cause the code to throw an appropriate exception. An exception can carry supplemental information that is passed as the constructor's argument(s).
- If a method is *invisible* from outside the package (e.g., private, or package-private methods), failed pre-conditions should be verified by an `assert`. An `assert` can specify supplemental information as the expression following the colon.

Supplemental Information

In both cases, the error handling should be supplemented by additional information. In Java, the supplemental information can take the form of one or more of the following:

- *Nothing*, if the source of the error is clear from other information already available (such as a stack trace).
- A `String` that outlines the reasons for the error. Strings should only be used if their use conforms to Item 62 in the reading assignment.
- A `Throwable` that describes the cause of the error.

Null Arguments

A common occurrence of failed pre-conditions are arguments that are *null* even though null parameters are not expected or meaningful. Unless stated otherwise, you should assume through this series of assignments that null parameters constitute a failed pre-condition that necessitates error handling.

The Java Language Features module on canvas contains pointers on assertions, exceptions, and null parameter checking.

Filters

After the player has made a guess, she receives feedback on correct and incorrect characters. The feedback is represented as a predicate consisting of the logical conjunction of `NGram` methods. For example, if the correct word is "rebus" and the player guesses "route", the feedback is the predicate

```
matches(0, "r") and
!contains(1, "o") and
containsElsewhere(2, "u") and
!contains(3, "t") and
containsElsewhere(4, "e")
```

An n-gram such as "redux" is consistent with the predicate whereas "regex" is not. The key "rebus" is always consistent with the predicate. The player should strive to guess an n-gram that is consistent with the predicate.

The `public final class Filter` represents a predicate on n-grams. It contains a `private final Predicate<NGram> predicate` and a private constructor that sets its value. The `public static Filter from(Predicate<NGram> predicate)` returns a new `Filter` with the given predicate or throws a `NullPointerException` if the predicate is null. The filter delegates the test method to the predicate. The `public static final Filter FALSE` defines a filter that always evaluates to `false`. To build the feedback incrementally, `Filter` has a `public Filter and(Optional<Filter> other)` method that returns the logical conjunction of this filter and the `other`, or just this filter if the `other` is empty.

The `Corpus` supports the additional `public long size(Filter filter)` that returns the number of n-grams consistent with the `filter`. For example, a Corpus containing "route", "rebus", "redux", and "hello" would return 2 because "rebus" and "redux" are the two n-grams consistent with the filter. The `Corpus' Builder` supports a `public Builder filter(Filter filter)` that returns a builder with the n-grams that are consistent with the `filter`.

Matching

Filters are generated by attempting to match the individual characters in the player's guess and the key. First, identical characters in the same position are matched with each other, reported to the player, and removed from further consideration. Among the remaining characters, identical misplaced characters are matched with each other, reported, and removed from further consideration. The other characters are reported as absent. The matching algorithm is therefore:

```
**Algorithm** MATCH(key, guess)
**Input**: two n-grams key and guess of the same length n
**Output**: predicate representing the similarity of key and guess

key-match[i], guess-match[i] ← false (i = 0, 1, .. , n-1)
// match identical characters in the same location
for i ← 0 to n-1 do
    if key[i]=guess[i] then
        report the match to the player
        key-match[i], guess-match[i] ← true
// match characters in different positions
for i←0 to n-1 do
    if !guess-match[i] and
        there is a j such that !key-match[j] and key[j]=guess[i]
    then
        report i to the player
        key-match[j], guess-match[i] ← true
// absent characters
for i←0 to n-1 do
    if !guess-match[i] then
        report i as absent
```

The package-private `final class NGramMatcher` implements the MATCH algorithm. It has a `public static final NGramMatcher of(NGram key, NGram guess)` that initializes the two n-grams. Its primary method is `Filter match()` that returns the corresponding filter. However, it returns `FALSE` if the `key` and `guess` have different length. The `NGramMatcher` implementation details are intentionally left up to you.

General Considerations

These classes may contain as many auxiliary private and package-private methods as you see fit, and additional package-private helper classes may be defined. However, any modification or addition to public classes and methods must be approved by the instructors at the discussion board.

You should write JUnit tests to make sure that your primary methods work as intended. However, we will revisit testing later in the course, so extensive testing is not yet recommended. If you have never used JUnit, the JUnit module on canvas lists resources to get you started. Similarly, your code should have a reasonable number of comments, but documentation is going to be the topic of a future assignment. As a general guideline at this stage of the course, comments and tests should be like those accepted in EECS 132. Additionally, comments should only be applied to the code sections that you feel are not self-documenting.

Discussion Guidelines

Although the discussion can range through the whole reading assignment, the emphasis will be on:

- Functions that exceed McCabe's complexity of 4 (if any)
- Non-structured programming constructs and break statements (if any)

Submission

Submit an electronic copy of your program to Canvas. In addition to your code, include a README file explaining how to compile and run the code. The code should be handed in a zip, tar.bz2, or tar.gz archive. Archives in 7z cannot be accepted. You can either bring a laptop to discussion to display your project on a projector, or present your project from the canvas submission.

Grading Guidelines

Advance Warning: starting with Programming Assignment 4, an automatic C (or less) is triggered by any routine with complexity greater than 4 or by any substantially repeated piece of code.