CIT 594 Module 3 Programming Assignment

Stacks & Queues

In this assignment, you will use Stacks and Queues to check documents for correct nesting. After briefly explaining how this can be used as part of the syntax validation in multiple settings including arithmetic expressions and computer programming languages, you will build a function that applies this knowledge to check strings of nestable characters for valid delimiter nesting.

Learning Objectives

In completing this assignment, you will:

- Become familiar with the methods in the java.util.Stack class and java.util.Queue interface
- Work with an abstract data type (specifically, Queues) by using only the interface of an implementation
- Apply what you have learned about how Stacks and Queues work

Background

Consider the following valid Python expression:

```
1 [0, (1, 2, {3, (4, 5, 6), (7, 8)}, 9), 10, 11, 12 ]
```

The expression contains 3 types of open and close elements:

Open	Close
(	)
[	]
{	}

There are two rules:

1. A closing element must close the *current context*
2. At the end of input, all contexts must be closed

An open element starts a context and the complementary close element ends that context. Note from the above example that contexts can be *nested* inside each other, but we can't close an outer context until the current context is closed, so this expression would be **invalid** – notice '})' instead of ')}':

```
[0, (1, 2, {3, (4, 5, 6), (7, 8}), 9), 10, 11, 12 ]
```


Now suppose we want to create a syntax checking algorithm that reads a source code file and determines if the above two rules are respected (keep in mind we are just checking if the nesting is correct, and not attempting to perform a full syntax validation). First, we would need a parser that understands the language whose syntax is being checked. It would look at each character in the file, classify its effect on the nesting context as either *opening*, *closing*, or *neutral*, and return a **Queue** of only the opening and closing elements. Neutral elements have no effect on the nesting context, so they are ignored. In the first (valid) example above, the following characters would be enqueued: `[({()()})]`. We must also define a bidirectional "matching" of elements, such that '(' matches ')', '[' matches ']', etc.

Given this Queue, we repeatedly dequeue elements. If the dequeued element's nesting effect is *opening*, we push it to a **Stack**. If the dequeued element's nesting effect is *closing*, we pop the Stack only if the top of the Stack matches the dequeued element; if the top of the Stack doesn't match, we know there must be a syntax error. Finally, we know the entire file is valid if, upon emptying our Queue, we find that our Stack is empty.

Try this algorithm on pencil and paper using some simple examples:

```
# valid; stack should be empty
[0, (1, 2, {3, (4, 5, 6), (7, 8)}, 9), 10, 11, 12 ]

# invalid close on }; stack should be '[({('
[0, (1, 2, {3, (4, 5, 6), (7, 8}), 9), 10, 11, 12 ]

```

```
# not terminated; stack should be '['
[0, (1, 2, {3, (4, 5, 6), (7, 8)}, 9), 10, 11, 12

# invalid close on }; stack should be empty
[0, (1, 2, {3, (4, 5, 6), (7, 8)}, 9), 10, 11, 12 ]}

# valid; we are only considering nesting and not other aspects of syntax
(+ 1 1)
```

Once you understand how the above algorithm works, you will be ready to tackle this assignment.

This was just a warmup; the algorithm you will implement for this assignment will be slightly more advanced in that the given Queue may contain neutral elements, null/uninitialized elements, and must support any kind of "nestable" object, not just characters. More on this below.

Interfaces, abstract classes, and implementations

One of the key ideas of this assignment is that an interface can have multiple conformant implementations. Queues are a concept that can be implemented using linked lists, circular dynamic arrays, and other concrete data structures; likewise, Stacks can be implemented using linked lists, array lists, etc.

To reinforce this point, we have provided an interface called *Nestable*. (Though, due to limitations in the Java language, it's actually an abstract class in the given code, but we'll often say "interface" since this is the role it plays.) To make a class "Nestable" – i.e., `MyClass extends Nestable` – asserts that every instance of that class obeys certain properties, principally that it has some sort of nesting effect (opening, closing, neutral) and that it defines a notion of "matching" elements.

You will write a method that must work for *all* Nestable objects, and in particular, we have provided two kinds of Nestables: characters (*NestableCharacter*) and something a little more abstract (*VanNest*). The *NestableCharacter* class would be used for strings like the Python snippets given in the previous section. In this case, each individual character in the string is Nestable. Objects of the *VanNest* class will also follow the same nesting behavior as other subclasses of *Nestable*, but otherwise exist primarily for artistic purposes.

For more details about interfaces, abstract classes, and Java inheritance in general, see the Interfaces and Inheritance chapter in Java™ Tutorials. The "Abstract Classes" section has a comparison and guidance for how to decide between using an abstract class or interface.

Getting Started

Download the zip file, which contains the following:

- **NestingChecker.java**, which contains the unimplemented method `checkNesting` that you will write in this assignment.
- **Nestable.java**, which is an abstract class representing the concept of a "nestable" entity. You may find `enum NestEffect`, `getEffect()`, and `matches(Nestable other)` useful in your implementation.
- **NestableCharacter.java**, whose class *NestableCharacter* extends *Nestable* to represent single characters as nestable entities. The `matches` method, required by *Nestable*, is defined such that '(' matches ')', '[' matches ']', and '{' matches '}', all bi-directionally.
- **VanNest.java**, whose class *VanNest* extends *Nestable*.
- **NestingReport.java**, whose class *NestingReport* is the compound datatype that you will need to return from your implementation of `NestingChecker.checkNesting`.
- **Playground.java**, whose class *Playground* is an interactive program to help test your implementation of `NestingChecker.checkNesting`. This is for your use and will not be graded. Instructions for use are provided in the file.

You do not need to understand the implementation details of *NestableCharacter* or *VanNest*. You can treat them as black boxes, where the only thing you know about is their *Nestable* interface.

Activity

In **NestingChecker.java**, implement the `checkNesting` method. `checkNesting` takes as input a Queue of *Nestable*s and returns a *NestingReport* that describes the nesting structure, according to the specification described below.

- If the input represents a valid nesting, then return a *NestingReport* with `status` set to `VALID`, `badItem` set to `null`, and `stackState` set to the current contents of the Stack.
- If the input queue is `null`, return a *NestingReport* with `status` set to `NULL_INPUT`, `badItem` set to `null`, and `stackState` set to the empty Stack.

- When an element b in the Queue is `null`, return a *NestingReport* with `status` set to `NULL_ITEM`, `badItem` set to b (null), and `stackState` set to the current contents of the Stack.
- When an element b in the Queue is an invalid closing element, return a *NestingReport* with `status` set to `INVALID_CLOSE`, `badItem` set to b, and `stackState` set to the current contents of the Stack (which should not include b).
- When the Queue represents an invalid nesting because it's empty but there are still elements remaining in the Stack that were never closed, return a *NestingReport* with `status` set to `NOT_TERMINATED`, `badItem` set to `null`, and `stackState` set to the current contents of the Stack.

Java doesn't allow multi-value returns, so to get around this, we return a single object of type *NestingReport* that contains the fields referenced above. An example of returning one from the assigned function is given in the starter code. Every component of your *NestingReport* will be tested for correctness.

Other requirements

Please do not change the signature of the `checkNesting` method (its parameter list, name, and return value type). Also, do not create any additional .java files for your solution, and **do not modify any java file other than NestingChecker.java**. If you need additional classes, you can define them in **NestingChecker.java**.

Last, be sure that your *NestingChecker* class is in the default package, i.e. there is no "package" declaration at the top of the source code.

Tips

- This is a short and simple assignment. A correct implementation should only need 20-30 lines of code.
- You don't need to read or understand all of the starter code. Focus only on what you need. Along with *NestingReport*, you should only need the methods from the *Nestable*, *Stack*, and *Queue* interfaces.

Before You Submit

Please be sure that:

- your *NestingChecker* class is in the default package, i.e. there is no "package" declaration at the top of the source code
- your *NestingChecker* class compiles and you have not changed the signature of the `checkNesting` method
- you have not created any additional .java files and have not made any changes to other java files
- you have filled out the required academic integrity signature in the comment block at the top of your submission file

How to Submit

After you have finished implementing the *NestingChecker* class, go to the "Module 3 Programming Assignment Submission" item and click the "Open Tool" button to go to the Codio platform.

Once you are logged into Codio, read the submission instructions in the README file. Be sure you upload your code to the "submit" folder.

To test your code before submitting, click the "Run Test Cases" button in the Codio toolbar.

Unlike the Module 1 Programming Assignment, this will run some but not all of the tests that are used to grade this assignment. That is, there are "hidden tests" on this assignment!

The test cases we provide here are "sanity check" tests to make sure that you have the basic functionality working correctly, but it is up to you to ensure that your code satisfies all of the requirements described in this document. Just because your code passes all the tests when you click "Run Test Cases" doesn't mean you'd get 100% if you submit the code for grading!

You are responsible for writing and running any additional test cases required to ensure your implementation satisfies the requirements of this assignment.

When you click "Run Test Cases," you'll see quite a bit of output, even if all tests pass, but at the bottom of the output you will see the number of successful test cases and the number of failed test cases.

You can see the name and error messages of any failing test cases by scrolling up a little to the "Failures" section.

Assessment

This assignment is scored out of a total of 23 points, based on the correctness of the return value for different inputs representing correct and incorrect nestings of nestable entities.

There is a hidden implementation of Nestable that you are not given but will nevertheless be tested on.

As noted above, the tests that are executed when you click "Run Test Cases" are **not** all of the tests that are used for grading. There are hidden tests for each of the three methods described here.

After submitting your code for grading, you can go back to this assignment in Codio and view the "results.txt" file, which should be listed in the Filetree on the left. This file will describe any failing test cases.

Frequently Asked Questions

Where can I learn more about Java's Stack and Queue?

Documentation about the methods in the Stack class and Queue interface in Java are available at:

- https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Stack.html
- https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Queue.html

Refer to this documentation if you need help understanding the methods that are available to you.

Why is Queue an interface, while Stack is a class?

Just historical reasons. In modern software design we would want abstract concepts like Queues and Stacks to be interfaces, but Java already had a legacy Stack class, so we're stuck with it.

Will my code be tested with other *Nestable* subclasses?

Yes.

I don't understand the signature of `checkNesting`.

Here is the signature:

```
1 public static NestingReport checkNesting(Queue<? extends Nestable> elements)
```

Let's walk through each component of this signature:

- `public`, as you should know, means that this method is accessible to the outside world (i.e. classes outside the current package).
- `static`, as you should know, means that this is a class method, not an instance method. So you access this method by calling `NestingChecker.checkNesting`, not by instantiating a `new NestingChecker`.
- `<? extends Nestable>` means that the type of element in the Queue is a *Nestable* or a subclass of *Nestable*. The '?' is just a wildcard, meaning we aren't ever going to reference the name of that class, so we don't need a type variable.

  (If we had instead written `<E extends Nestable>`, this would mean the same thing but it would allow us to use the type variable E to mean the name of the subclass. It's like writing $\forall E \subseteq$ Nestable in set notation.)

I don't understand the *Nestable* class.

Nestable represents the abstract notion of an entity that has the effect of modifying the nesting context. This entity could be a single character (as in the opening/closing characters we showed in the Python code in the Background section above), or a multi-character element like an HTML tag or an S-expression, etc. All of these have the common property that they can affect the nesting context by either opening a new one, closing the current one, or having no effect at all. We represent these three possible effects using the enum `NestEffect`.

A variable of type `NestEffect` can only take on the values of `OPEN`, `NEUTRAL`, or `CLOSE`. For example, we would encode the opening parenthesis character '(' as having the `OPEN` effect, the closing parenthesis character ')' as having the `CLOSE` effect, and all other characters as `NEUTRAL`.

We recommend looking at the implementation of *NestEffect*'s `matches()` method to see how you can use a `switch` statement on an `enum`. This syntax might be useful to you.

Should my `checkNesting` method have special code for handling *VanNest*s versus *NestableCharacter*s?

Nope. We suggest writing your implementation as if you're expecting a Queue of *NestableCharacter*s, because it's simpler to think about and you should have done a warmup exercise on pencil and paper in the Background section. However, if your implementation of the algorithm is correct, it should work just as well for other *Nestable* classes and objects. That's the beauty of coding against an interface.

I don't understand the *VanNest* class.

Does anyone? Some things exist to be appreciated without complete understanding.

How should I process the Queue?

These are what we would consider good practices (choose one):

- `remove`/`poll` (i.e. dequeuing, not iterating) because Queues are meant to be drained
- `isEmpty` preferred over `size()` because you really only care if there are more or not, not how many more

These are what we would consider bad practices:

- `peek` when you know you're going to remove the element, as then you're just doing repeated work
- anything that involves converting the Queue to a different data structure; use the Queue interface's methods instead
- for loops, for each loops, iterators, or anything else that iterates through the Queue without actually consuming it; Queues are meant to be treated as a data structure to which you only have access to the front at any given time

We encourage arguments on Ed Discussion about why each of these are good or bad.

Can I use the textbook's solution?

There is a similar program described in Section 6.1.5 of your course textbook. You may refer to this solution and even reuse parts of it as you see fit, but keep in mind that the solution in the book is **not** a complete solution to this particular problem.

How can I test my implementation easily?

Check out jshell, which should be built-in with your installation of Java.

To load all of your classes into jshell, you can run this command on the command line:

```
jshell Nestable.java NestingReport.java NestableCharacter.java NestingChecker.java
 Playground.java
```


It has to be in that order because when loading classes into jshell in this manner, they have to be loaded in dependency order. However there are other ways to do it that may be easier – see the jshell documentation.

Then you'll be able to call `Playground.interact()` to see how your implementation handles various lines of input. That method is just a convenience, you can also work directly with the other methods to interactively see how they respond to various cases. This is not entirely a replacement for unit testing, but it can be useful for figuring out some of the unit tests to write.

Optional Challenge

Implement an Reverse Polish Notation (RPN) Calculator

Postfix notation entirely avoids nesting by using a stack to manage operands and results. Operands are added to the stack in the order given, and operators pop however many values they require for input, compute the result, and push the result back onto the stack. For example:

```
1 1 + => 2
2 1 3 + * => 8
2 1 + 3 * => 9
```

If you want to take the exercise further, try pre-fix notation, where the operator is written before the operand (famously the basis for the entire syntax of the Lisp programming language).

```
(+ 1 1) => 2
(* 2 (+ 1 3)) => 8
(* (+ 2 1) 3)  => 9
(+ 1 1 1 1) => 4
```

Notice the last example, unlike infix and postfix, prefix makes it trivial to support arbitrary numbers of operands. A simple "add" becomes a full "sum" function.