CIT 594 Module 3 Programming Assignment
Frequently Asked Questions

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:

```java
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?

No. 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.