Annotations

Learn the use of various annotations used in concurrent code.

*If you are interviewing, consider buying our **number#1** course for **Java Multithreading Interviews**.*

Overview

In this lesson, we'll explore the various Java annotations that are related to concurrency. These are:

1. `@ThreadSafe`
2. `@NotThreadSafe`
3. `@Immutable`
4. `@GuardedBy(lock)`

Class-level annotations

Note that these annotations serve as documentation about class behavior but don't change the ability or functionality of class in any way. Also, these class-level annotations become part of the public documentation of a class.

`@ThreadSafe`

The annotation `@ThreadSafe` can be applied to a class to indicate to users and maintainers that the class is thread safe and multiple threads can concurrently interact with an instance of the class without worrying about synchronization. Newbies should not confuse the annotation to mean that it makes a class thread safe. The annotation only serves as documentation.

`@NotThreadSafe`

The `@NotThreadSafe` annotation is the opposite of `@ThreadSafe` and is intended to explicitly communicate to the users and maintainers that the class requires synchronization effort on part of the users to ensure thread-safe concurrent access to instances of the class.

`@Immutable`

The `@Immutable` annotation indicates that once an object of the class is instantiated, it can't be changed. All operations on such an instance of the class become read-only operations and consequently the class becomes thread safe. In other words, `@Immutable` implies `@ThreadSafe`. The `String` class is, perhaps, the most well-known Java immutable class. Other examples include wrapper classes such as `Integer`, `Byte`, `Short, and` Boolean` Java classes.

Field and method-level annotations

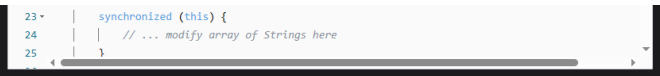
Unlike class-level annotations the field and method-level annotations don't appear in the public documentation of a class and are targeted towards informing the maintainers and extenders of a class.

`@GuardedBy(lock)`

The annotation `@GuardedBy(lock)` can be used to document that a certain field or method should only be accessed while holding the lock argument in the annotation. The `lock` argument passed-in to the annotation can take-on different values which we'll discuss below:

Consider the class `AnnotationsExampleClass` in the widget below:

```
public class AnnotationsExampleClass {

    // field is protected by the method instance
    @GuardedBy("this") // example of @GuardedBy("this")
    private String[] stringValues;

    private Object customLock = new Object();

    private static ReentrantLock counterLock = new ReentrantLock();

    @GuardedBy("AnnotationsExampleClass.class") // example of @GuardedBy("ClassName.class")
    private static long lastAccess = System.currentTimeMillis();

    private void updateAccessTime() {
        synchronized (AnnotationsExampleClass.class) {
            lastAccess = System.currentTimeMillis();
        }
    }

    public void manipulateStringValues() {
        // This method holds the "this" lock before
        // accessing the stringValues variable.
```



`@GuardedBy("this")`

Indicates that the field or method should be accessed while holding the intrinsic lock, i.e. the object itself of which the field or method is a member. In our example above the field `stringValues` uses this annotation and in the method `manipulateStringValues()` the intrinsic lock is acquired before manipulating `stringValues`.

`@GuardedBy("field")`

Indicates that the field or method should be accessed while synchronizing on the lock associated with the named field in the annotation. In our example class `AnnotationsExampleClass`, the field `customLock` appears as the named field in the annotation for the method `getStringValue()`. This method is in turn invoked by `printValues()` which acquires the intrinsic lock associated with the `customLock` field before invoking the `getStringValue()` method.

If the named field in the annotation were a lock object itself, e.g. if `customLock` were an instance of the `ReentrantLock` class the we'd acquire the explicit lock, i.e. `customLock.lock()` rather than the intrinsic lock associated with the object.

`@GuardedBy("ClassName.fieldName")`

This form of annotation is similar to the `@GuardedBy("field")`, however, the lock argument represents an object held in a static field of another class. In our example, the inner `Helper` class's `counter` field uses the annotation to indicate to the users to acquire the `counterLock` of the `AnnotationsExampleClass` before manipulating the `counter` object.

`@GuardedBy("methodName()")`

Indicates that the lock returned by the named method should be acquired before interacting with the annotated field or method. In our example class, the method `lockMaster()` is named in the annotation for the method `transforms()`. The annotation implies that `lockMaster(1)` should be invoked, the returned lock acquired and then only `transforms()` should be invoked.

`@GuardedBy("ClassName.class")`

Indicates that the annotated field or method should be accessed only after acquiring the class literal object for the named class. In our example, the field `lastAccess` is annotated with the `@GuardedBy` annotation with the class object `AnnotationsExampleClass`. The method `updateAccessTime()` correctly acquires the lock on the class object before accessing the `lastAccess` field.

Summary

The annotations we discussed in this lesson only serve to document notes and information for users and maintainers but can also be beneficial for static analysis tools that can programmatically verify if a class behaves as advertised. For instance, tools can verify if a class marked as `@Immutable` is truly immutable or not. In general, it is a good practise to annotate where appropriate for improving readability of one's code.

← Back lesson
BrokenBarrierException

Mark As Completed Next →
Quiz 1