StampedLock

Learn how to use the StampedLock class for optimistic read, write and read access.

> Code widgets in this lesson may fail to execute, if Educative.io's code execution platform doesn't use Java version 9 or above. This is because the `StampedLock` class was introduced in Java 9.

Overview

The `StampedLock` was introduced in Java 9 and is a capability-based lock with three modes for controlling read/write access. The `StampedLock` class is designed for use as an internal utility in the development of other thread-safe components. Its use relies on knowledge of the internal properties of the data, objects, and methods it protects.

Modes

The state of a `StampedLock` is defined by a version and mode. There are three modes the lock can be in:

- Writing

for controlling read/write access. The `StampedLock` class is designed for use as an internal utility in the development of other thread-safe components. Its use relies on knowledge of the internal properties of the data, objects, and methods it protects.

Modes

The state of a `StampedLock` is defined by a version and mode. There are three modes the lock can be in:

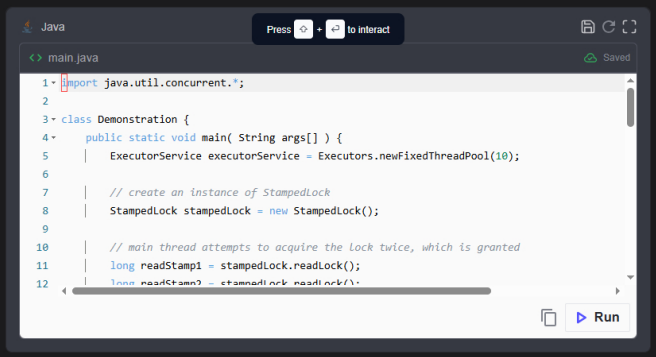
- Writing
- Reading
- Optimistic Reading

On acquiring a lock, a stamp (long value) is returned that represents and controls access with respect to a lock state. The stamp can be used later on to release the lock or convert the existing acquired lock to a different mode.

Reading

The read mode can be acquired using the `readLock()` method. When reading a thread isn't expected to make any changes to the shared state and therefore it makes sense to have multiple threads acquire the read lock at the same time. The method returns a stamp that can be used to unlock the read lock. Timed and untimed versions of `tryReadLock()` also exist.

The code widget below demonstrates multiple threads acquiring the `StampedLock` in read mode.



While read lock is held by a reader thread, all attempts to acquire the write lock will be blocked.

Writing

The write mode can be acquired by invoking the `writeLock()` method. When the write lock is held, all threads attempting to acquire the read lock will be blocked. Similar to the read lock, timed and untimed versions of `tryWriteLock()` exist. The following widget demonstrates the

use of the write lock:

```java
import java.util.concurrent.*;
class Demonstration {

    public static void main( String args[] ) {
        ExecutorService executorService = Executors.newFixedThreadPool(10);

        // create an instance of StampedLock
        StampedLock stampedLock = new StampedLock();

        // main thread acquires the write lock before spawning read thread
        long stamp = stampedLock.writeLock();

    }
}
```

In the above program, the print statement for releasing the write lock is always output before the statement for acquiring the read lock.

Remember that the `writeLock()` is not re-entrant. The following program results in a deadlock:

```java
class Demonstration {
    public static void main( String args[] ) {

        // create an instance of StampedLock
        StampedLock stampedLock = new StampedLock();

        stampedLock.writeLock();

        stampedLock.writeLock();
    }
}
```

Optimistic Read

Consider a scenario, where we never want to block a thread from acquiring the write lock if the lock isn't already acquired. Recall, that if the read lock is already held then a thread attempting to acquire the write lock will block. `StampedLock` makes this scenario possible with the optimistic reading mode. A thread can invoke the `tryOptimisticRead()` method which returns a non-zero value if the write lock isn't exclusively locked. The thread can then proceed to read but once the thread reads the desired value it must validate if the lock wasn't acquired for a write in the meanwhile. If validation returns true then the reader thread can safely assume that from the time it was returned a stamp from invoking the `tryOptimisticRead()` method till the time the thread validated the stamp, the lock hasn't been acquired for writing and the data hasn't changed. In case, if the lock was acquired for a write in the intervening period, then the validation will return false. The validation can be performed using the method `validate(long)` which takes in the stamp issued at the time of invoking `tryOptimisticRead()`.

This mode can be broken at any time by a thread acquiring a write lock. Another way to think of this mode is as an extremely weak version of a read-lock. Using optimistic reads in cases where the read happens briefly can improve throughput and reduce contention.

Note that `tryOptimisticRead()` doesn't acquire the read lock nor is the read lock count incremented as the following snippet demonstrates:

```java
class Demonstration {
    public static void main( String args[] ) {

        ExecutorService executorService = Executors.newFixedThreadPool(10);

        // create an instance of StampedLock
        StampedLock stampedLock = new StampedLock();

        // optimistic read
        stampedLock.tryOptimisticRead();

        // outputs "read lock count 0 is read locked false"
```

The sequence of operations in the following code widget illustrates when a stamp returned from a `tryOptimisticRead()` call becomes invalid:

main.java Saved

```java
class Demonstration {
    public static void main( String args[] ) {
        // create an instance of StampedLock
        StampedLock stampedLock = new StampedLock();

        // try optimistic read
        long stamp = stampedLock.tryOptimisticRead();

        // check for validation, prints true
        System.out.println(stampedLock.validate(stamp));

        // acquire the write lock which will invalidate the previous stamp
```

Run

In the above program, instead of the write lock if we acquire the read lock, the validation will come out to be true. The official Java documentation marks optimistic reads as inherently fragile and cautions *"optimistic read sections should only read fields and hold them in local variables for later use after validation. Fields read while in optimistic mode may be wildly inconsistent, so usage applies only when you are familiar enough with data representations to check consistency and/or repeatedly invoke method validate(). For example, such steps are typically required when first reading an object or array reference, and then accessing one of its fields, elements or methods."*

The following program demonstrates the idiomatic use of optimistic read. Consider a function that is always passed-in an integer array with three elements. The function computes the product of the three numbers and returns the answer. We'll also assume that the multiplication doesn't result in an overflow exception. The important detail is to remember to read in the array elements into local variables, validate the stamp and then compute the product using the local variables. The reason to store values in local variables is because just after stamp validation, we can't be sure if the original passed-in array remains unmodified. With storing the array values in local variables, we have in a sense, preserved a snapshot of the state of the data, i.e. the array and computed a product for that snapshot. In case the stamp validation fails, then we are compelled to acquire the read lock and compute the product. The code with comments explaining the program flow appears below:

Java

main.java Saved

```java
import java.util.concurrent.*;

class Demonstration {

    static StampedLock stampedLock = new StampedLock();

    public static void main( String args[] ) {
        int[] array = new int[3];
        array[0] = 3;
        array[1] = 5;
        array[2] = 7;
```

Run

Coverting modes

The `StampedLock` also provides support for converting locks from one mode to another if certain conditions are met. For instance, the method `tryConvertToWriteLock(long)` upgrades to a write lock if one of the following is true:

- The write lock is currently held.
- The read lock is current held and no other readers exists
- The optimistic mode is in progress and the write lock is available

Here are some examples of upgrading and downgrading across modes:

Upgrade to write lock after optimistic read

```java
// create an instance of StampedLock
StampedLock stampedLock = new StampedLock();

// get into optimistic read mode
long stamp = stampedLock.tryOptimisticRead();

// upgrade to write lock
stampedLock.tryConvertToWriteLock(stamp);

/*
* output of program is
* Read locks held : 0
* Write lock held : true
*/
System.out.println("Read locks held : " + stampedLock.getReadLockCount() + "\nWrite lock held :
```

Upgrade to write lock from read lock

```java
// create an instance of StampedLock
StampedLock stampedLock = new StampedLock();
```

```
// acquire read lock
long stamp = stampedLock.readLock();

// upgrade to write lock
stampedLock.tryConvertToWriteLock(stamp);

/*
* output of program is
* Read locks held : 0
* Write lock held : true
*/
System.out.println("Read locks held : " + stampedLock.getReadLockCount() + "\nWrite lock held :
```

Downgrade to read lock from write lock

```
// create an instance of StampedLock
StampedLock stampedLock = new StampedLock();

// acquire write lock
long stamp = stampedLock.writeLock();

// downgrade to read lock
stampedLock.tryConvertToReadLock(stamp);

/*
* output of program is
* Read locks held : 1
* Write lock held : false
*/
System.out.println("Read locks held : " + stampedLock.getReadLockCount() + "\nWrite lock held :
```

Downgrade to optimistic read from read lock

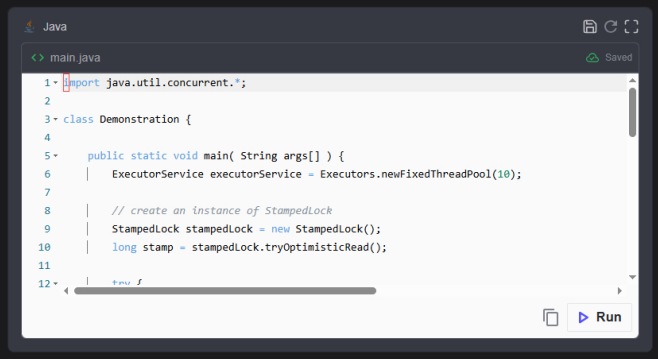
```
// create an instance of StampedLock
StampedLock stampedLock = new StampedLock();

// acquire read lock
long stamp = stampedLock.readLock();

// downgrade to optimistic read mode
stampedLock.tryConvertToOptimisticRead(stamp);

/*
* output of program is
* Read locks held : 0
* Write lock held : false
*/
System.out.println("Read locks held : " + stampedLock.getReadLockCount() + "\nWrite lock held :
```

Failed upgrade to write lock example

The final example demonstrates a failed attempt by the main thread to upgrade to the write lock from optimistic read mode. Another spawned thread holds the write lock and prevents the main thread from upgrading. The output of the program appears below:

```
Stamp : 0
Read locks held : 0
Write lock held : true
spawned thread exiting.
```

The code widget below includes comments that explain the failed lock upgrade example:



Characteristics

Some of the salient notes/features about the `StampedLock` include:

- Similar to `Semaphore` the `StampedLock` has no notion of ownership. Lock acquired by one thread can be released by another thread, which isn't possible for implementation of the `Lock` interface.
- The `StampedLock` class doesn't implement the `Lock` or `ReadWriteLock` interfaces but applications that desire to use the functionality offered by these interfaces out of an instance of `StampedLock` can do so using the methods `asReadLock()`, `asWriteLock()` and `asReadWriteLock()`. The following statements are possible

```
// create an instance
StampedLock stampedLock = new StampedLock();

// use functionality as a ReadLock
Lock readLock = stampedLock.asReadLock();

// use functionality as a WriteLock
Lock writeLock = stampedLock.asWriteLock();
```

```
    // use functionality as a ReadWriteLock
    ReadWriteLock readWriteLock = stampedLock.asReadWriteLock();
```

- The scheduling policy of `StampedLock` does not consistently prefer readers over writers or vice versa. All `try*` methods are best-effort and do not necessarily conform to any scheduling or fairness policy.
- A zero return from any `try*` method for acquiring or converting locks does not imply or carry any information about the state of the lock; a subsequent invocation may succeed.
- Stamp values can recycle after a year but not earlier. This implies that if a stamp is held without use or validation for longer than this period may fail to validate correctly.
- The `StampedLock` class is serializable, but always deserializes into the initial unlocked state, so its instances are not useful for remote locking.
- Stamps aren't cryptographically secure and a valid stamp can be guessed by malicious participants in a system.
- The `StampedLock`'s write lock is not reentrant i.e. recursively attempting to acquire the lock will result in a deadlock as the following widget demonstrates. The execution times out because the main thread deadlocks itself by acquiring the write lock twice.

Java

main.java — Saved

```java
import java.util.concurrent.*;

class Demonstration {
    public static void main( String args[] ) {

        // create an instance
        StampedLock stampedLock = new StampedLock();

        stampedLock.writeLock();
        stampedLock.writeLock();

    }
```

Run

- Don't use optimistic read locks without validation if the read sections aren't tolerant to potential inconsistencies.

Conclusion

`StampedLock` is an advanced locking construct, which a typical Java developer is unlikely to encounter in her day to day work. `StampedLock` class offers flexible usage and functionality but requires careful thought in its application to avoid bugs that can be introduced in a program especially when using the optimistic read mode.

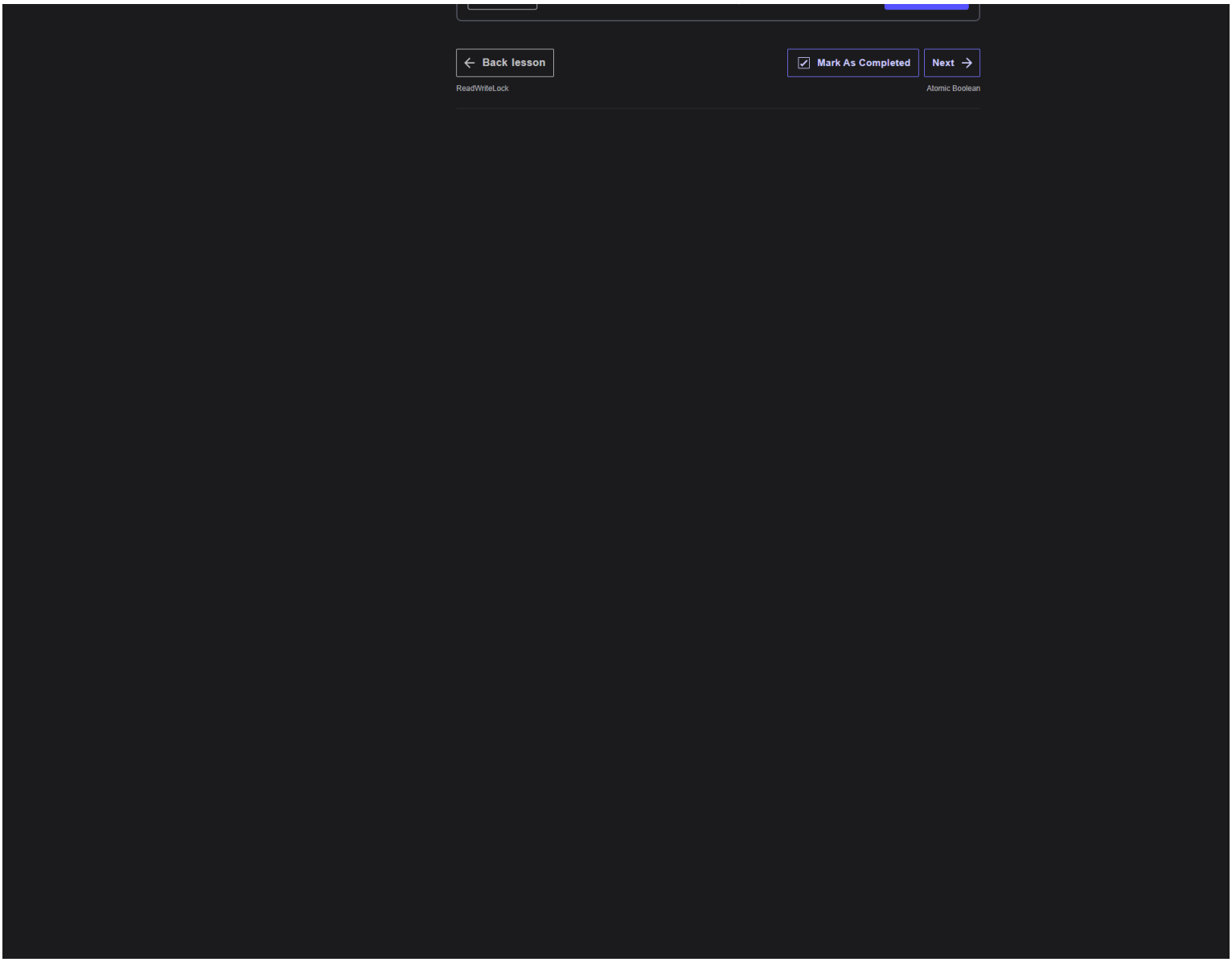
Technical Quiz

What is the outcome of running the below snippet:

1.
```
void demoWriteLock() throws Exception {

    ExecutorService executorService = Executors.newFixedThreadPool(10);

    StampedLock stampedLock = new StampedLock();

    // main thread acquires the write lock before spawning read thread
    long stamp = stampedLock.writeLock();

    try {
        executorService.submit(new Runnable() {
            @Override
            public void run() {
                System.out.println("Attempting to acquire read lock");
                long readStamp = stampedLock.readLock();
                System.out.println("Read lock acquired");
                stampedLock.unlock(readStamp);
            }
        });

        Thread.sleep(3000);

    } finally {
        // wait for read thread to exit
        executorService.shutdown();
        executorService.awaitTermination(1, TimeUnit.HOURS);

        // unlock write
        System.out.println("Write lock being released");
        stampedLock.unlock(stamp);
    }
}
```

- A. The program results in a deadlock
- B. The "Write lock being released" is printed after the "Read lock acquired"
- C. The "Read lock acquired" is printed after the "Write lock being released"
- D. Program throws an exception

Reset Quiz Q1 / Q1 Submit Answer


Back lesson
ReadWriteLock
Mark As Completed
Next
Atomic Boolean