Nonblocking Stack

Problem

Design a stack that doesn't use locks or `synchronized` and is thread-safe. You may assume that you are provided with an application-level API that mocks the hardware instruction compare-and-swap, to atomically compare and swap values at a memory location.

Solution

This is a slightly advanced question that requires knowledge about atomic classes in Java. You can read the section on Atomics in this course and then come back to this question. Very briefly, a stack is a data structure that implements first-in, last-out semantics for stored items.

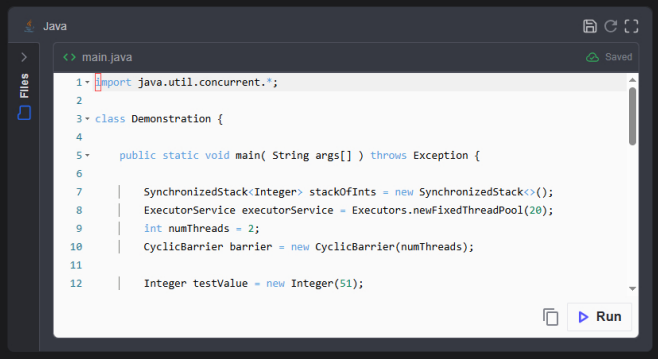
Usually, we want to atomically execute steps that require checking a value and then taking action based on the observed value. This is usually expressed as an `if-clause`. For instance, in the stack problem we would have a variable `top` that points to the top of the stack. Initially, this variable is `null` and in a single-threaded environment we would perform the following check when pushing the first element in the stack.

```
··if (top == null) {
  top = new StackNode();
  // ... other initialization steps
}
```

Usually, we want to atomically execute steps that require checking a value and then taking action based on the observed value. This is usually expressed as an `if-clause`. For instance, in the stack problem we would have a variable `top` that points to the top of the stack. Initially, this variable is `null` and in a single-threaded environment we would perform the following check when pushing the first element in the stack.

```
··if (top == null) {
  top = new StackNode();
  // ... other initialization steps
}
```

As you can see, the above snippet is a check-then-act scenario - We check if `top` is already `null`, if so we initialize it to an object of `StackNode`. The above code fails in a multi-threaded environment because the check-then-act sequence can't be performed atomically. The straightforward path to make the code thread-safe is to either use locks or simply mark the methods `synchronized`. The code for a thread-safe stack using `synchronized` is presented below for context and before we delve into a solution without locking.



There are several reasons to avoid locks which we described in detail in the Atomics section. The crux is that locking is a heavyweight synchronization mechanism that incurs significant overhead. In low to moderate contention situations, it may be better to execute critical sections of code speculatively in the hope that no other thread interferes. If interference is detected then actions are retried.

In interviews, this question can be solved either using the `AtomicReference` class or assuming that an API is provided that simulates the compare-and-swap (CAS) processor instruction. We'll first solve the problem using the mocked CAS API and then using `AtomicReference`.

CAS-based solution

Please read the CAS section in the Atomic Classes lesson before proceeding forward. We'll use the following class `SimulatedCompareAndSwap` to simulate the CAS processor instruction. Particularly, the method `compareAndSet()` can be used to determine if the check-then-act steps were successfully executed atomically or not. Take some time to go through the listing

of `SimulatedCompareAndSwap` and its associated tests in the code widget below:

The above program shows 7 threads attempting to change the value of a shared integer value that is stored in an instance of `SimulatedCompareAndSwap`. Each thread spins in a loop until it is able to update the integer value.

The `SimulatedCompareAndSwap` is a generic class and an instance of the class stores value for the type argument, which is an integer in the above widget. Consider the following snippet:

```
SimulatedCompareAndSwap<Integer> myInt = new SimulatedCompareAndSwap<>(5);
```

The instance of `SimulatedCompareAndSwap` is storing an integer object which has a value of 5. Even though we are passing an `int` literal in the constructor but value is implicitly converted into an `Integer` object. Next, we can use attempt to change the value as follows:

```
SimulatedCompareAndSwap<Integer> myInt = new SimulatedCompareAndSwap<>(5);

// attempt to change the value when the expected value is incorrect
System.out.println(myInt.compareAndSet(5, 9));

// attempt to change the value with correct expected value
System.out.println(myInt.compareAndSet(5, 9));

// retrieve the current value
System.out.println(myInt.getValue());
```

The act of checking the current value of the typed parameter stored in the instance of `SimulatedCompareAndSwap` and then updating it to a new value is executed atomically using the method `compareAndSet()`. If the execution is successful a true boolean value is returned. However, it is possible that another thread reads and updates the stored value before the main thread has a chance to execute `compareAndSet()`. In that instance, the expected value parameter passed in by the main thread will not match the currently stored value and the attempt to change the stored value will be rejected, indicated by a false boolean return value.

When implementing a stack using `SimulatedCompareAndSwap` we'll store the current value of the `top` of the stack in the `SimulatedCompareAndSwap` instance. The trick is to let a thread read the current value of the `top` of stack and then attempt to change it using `compareAndSet()`. If the attempt is unsuccessful we simply re-read the `top` value again, since the value we previously had didn't match the expected value and try again in a loop until the operation succeeds. Let's see how the push operation for the stack will be implemented using this strategy.

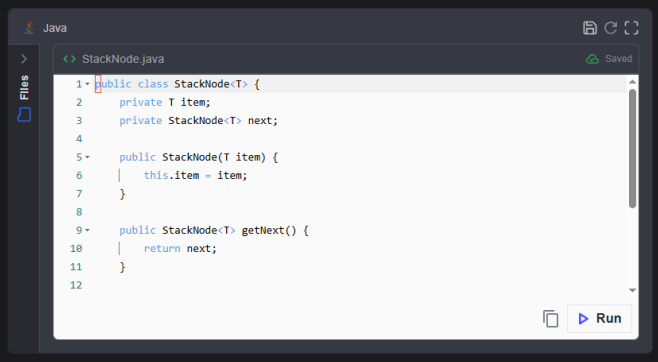
```
    do {
        // retrieve the current value of top
        oldHead = simulatedCAS.getValue();
        // create a new StackNode for the passed-in item.
        newHead = new StackNode<>(item);
        // Adjust the pointer
        newHead.setNext(oldHead);
    } while (!simulatedCAS.compareAndSet(oldHead, newHead));
    // attempt to atomically check and update
```

The above code is a partial snippet from the complete code for the `push()` method. The intent is to demonstrate how we loop until the expected value matches the stored value and the new value is updated. Similarly, the `pop()` method uses a loop to remove the top of the stack and update it with the next element atomically. The partial snippet from the `pop()` is shown below:

```
    do {
        // get the current top of the stack
        returnValue = simulatedCAS.getValue();
        // if the top is null then simply return null
        if (returnValue == null) return null;
        // compute the new top of stack
        newHead = returnValue.getNext();
    } while (!simulatedCAS.compareAndSet(returnValue, newHead));
    // attempt to update the new top of stack
```

Note, that these loops run a finite number of times. The unluckiest of threads will eventually have its expected value match the current value and the `compareAndSet()` method will

have its expected value match the current value and the compareAndSet() method will succeed. The complete code for the stack class `CASBasedStack` appears below:



`AtomicReference` solution

In the previous solution we mocked the compare-and-swap processor instruction with the class `SimulatedCompareAndSwap`. Fortunately, Java provides classes that can be used to atomically execute compare-and-swap operations for a variety of types. Chief among them are `AtomicInteger`, `AtomicLong` and `AtomicReference`.

We discuss the `AtomicReference` class in detail here and suggest the reader to go through that lesson, if not already familiar with the `AtomicReference` class before proceeding forward.

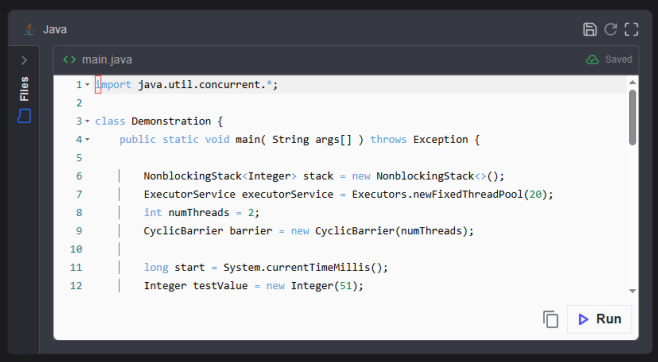
The solution using the `AtomicReference` class will resemble the solution we implemented using the `SimulatedCompareAndSwap` class. The trick is to be able to atomically update the `top` of the stack and for that we'll store the current top of the stack, which is of type `StackNode`, in an instance of `AtomicReference`. The `push()` method's core loop looks like the following:

```
do {
   oldTop = top.get();
   newTop = new StackNode<>(newItem);
   newTop.setNext(oldTop);
} while (!top.compareAndSet(oldTop, newTop));
```

A thread reads the current `top` of the stack and creates a new `StackNode` as the new top of the stack. The `next` pointer of the new top is set to the old top and then the thread makes an attempt to update the `top` value. It does so in a loop until successful. The core loop for the `pop()` operation is shown below:

```
do {
    oldTop = top.get();
    if (oldTop == null) return null;
    newTop = oldTop.getNext();
   } while (!top.compareAndSet(oldTop, newTop));
```

The complete code for the non-blocking stack using `AtomicReference` is shown below:



Caveats

One of the important caveats to recognize is that though our stack implementation `NonblockingStack` is thread-safe and may not use locking it doesn't guarantee ordering for either the `push()` or the `pop()` methods. Say the stack holds a single element and a thread T1 invokes the `pop()` method but gets suspended. Another thread T2 comes along and invokes the same method and is returned the only element in the stack. When T1 resumes, it'll be returned null even though at the time it made the `pop()` call the stack held one item. Similarly, if threads T1 and T2 attempt to `push()` two integers, say 1 and 2 then the order in which elements get pushed into the stack could be either (1 and then 2) or (2 and then 1).

Another observation to note is the use of `AtomicInteger` to keep an accurate count of the number of items in the stack. The methods `push()` and `pop()` as a whole aren't thread-safe, only the loops that update the `top` of the stack using compare-and-swap instructions. Other operations within these methods have to be made thread-safe such as incrementing a counter.