... continued

This lesson explains how to solve the token bucket filter problem using threads.

Using a Background Thread

The previous solution consisted of manipulating pointers in time, thus avoiding threads altogether. Another solution is to use threads to solve the token bucket filter problem. We instantiate one thread to add a token to the bucket after every one second. The user thread invokes the `getToken()` method and is granted one if available.

One simplification as a result of using threads is we now only need to remember the current number of tokens held by the token bucket filter object. We'll add an additional method `daemonThread()` that will be executed by the thread that adds a token every second to the bucket. The skeleton of our class looks as follows:

```
public class MultithreadedTokenBucketFilter {
    private long possibleTokens = 0;
    private final int MAX_TOKENS;
    private final int ONE_SECOND = 1000;

    public MultithreadedTokenBucketFilter(int maxTokens) {

        MAX_TOKENS = maxTokens;
    }

    private void daemonThread() {
    }

    void getToken() throws InterruptedException
    {
    }
}
```

The logic of the daemon thread is simple. It sleeps for one second, wakes up, checks if the number of tokens in the bucket is less than the maximum allowed tokens, if yes increments the `possibleTokens` variable and if not goes back to sleep for a second again.

The implementation of the `getToken()` is even simpler. The user thread checks if the number of tokens is greater than zero, if yes it simulates taking away a token by decrementing the variable `possibleTokens`. If the number of available tokens is zero then the user thread must wait and be notified only when the daemon thread has added a token. We can use the current token bucket object `this` to wait and notify. The implementation of the `getToken()` method is shown below:

```
    void getToken() throws InterruptedException {

        synchronized (this) {
            while (possibleTokens == 0) {
                this.wait();
            }
            possibleTokens--;
        }

        System.out.println(
                "Granting " + Thread.currentThread().getName() + " to
ken at " + System.currentTimeMillis() / 1000);
    }
```

Note that we are manipulating the shared mutable object `possibleTokens` in a synchronized block. Additionally, we `wait()` when the number of tokens is zero. The implementation of the daemon thread is given below. It runs in a perpetual loop.

```
    private void daemonThread() {

        while (true) {
            synchronized (this) {
                if (possibleTokens < MAX_TOKENS) {
                    possibleTokens++;
                }
                this.notify();
```

```
            }
            try {

                Thread.sleep(ONE_SECOND);
            } catch (InterruptedException ie) {
                // swallow exception
            }
        }
    }
```

The complete implementation along with a test-case appears in the code widget below:

```
import java.util.HashSet;
import java.util.Set;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        Set<Thread> allThreads = new HashSet<Thread>();
        final MultithreadedTokenBucketFilter tokenBucketFilter = new MultithreadedTokenBucket

        for (int i = 0; i < 10; i++) {

            Thread thread = new Thread(new Runnable() {

                public void run() {
                    try {
                        tokenBucketFilter.getToken();
                    } catch (InterruptedException ie) {
                        System.out.println("We have a problem");
                    }
                }
            });
            thread.setName("Thread_" + (i + 1));
            allThreads.add(thread);
        }

        for (Thread t : allThreads) {
            t.start();
        }

        for (Thread t : allThreads) {
            t.join();
        }

    }
}

class MultithreadedTokenBucketFilter {
    private long possibleTokens = 0;
    private final int MAX_TOKENS;
    private final int ONE_SECOND = 1000;

    public MultithreadedTokenBucketFilter(int maxTokens) {
```

```
        MAX_TOKENS = maxTokens;

        // Never start a thread in a constructor
        Thread dt = new Thread(() -> {
            daemonThread();
        });
        dt.setDaemon(true);
        dt.start();
    }

    private void daemonThread() {

        while (true) {

            synchronized (this) {
                if (possibleTokens < MAX_TOKENS) {
                    possibleTokens++;
                }
                this.notify();
            }

            try {
                Thread.sleep(ONE_SECOND);
            } catch (InterruptedException ie) {
                // swallow exception
            }
        }
    }

    void getToken() throws InterruptedException {

        synchronized (this) {
            while (possibleTokens == 0) {
                this.wait();
            }
            possibleTokens--;
        }

        System.out.println(
                "Granting " + Thread.currentThread().getName() + " token at " + System.currer
    }
}
```




We reuse the test-case from the previous lesson, where we create a token bucket with max tokens set to 1 and have ten threads request for a token. The threads are shown being granted tokens at exactly 1-second intervals instead of all at once. The program output displays the timestamps at which each thread gets the token and we can verify the timestamps are 1 second apart. Additionally, wee mark the daemon thread as background so that it exits when the application terminates.

Using a Factory

The problem with the above solution is that we start our thread in the constructor. **Never start a thread in a constructor as the child thread can attempt to use the not-yet-fully constructed object using `this`**. This is an anti-pattern. Some candidates present this solution when attempting to solve token bucket filter problem using threads. However, when checked, few candidates can reason why starting threads in a constructor is a bad choice.

There are two ways to overcome this problem, the naive but correct solution is to start the daemon thread outside of the `MultithreadedTokenBucketFilter` object. However, the con of this approach is that the management of the daemon thread spills outside the class. Ideally, we want the class to encapsulate all the operations related with the management of the token bucket filter and only expose the public API to the consumers of our class, as per good object orientated design. This situation is a great for using the **Simple Factory** design pattern. We'll create a factory class which produces token bucket filter objects and also starts the daemon thread only when the object is full constructed. If you are unaware of this pattern, I'll take the liberty insert a shameless marketing plug here and refer you to this design patterns course to get up to speed.

Our token bucket filter factory will expose a method `makeTokenBucketFilter()` that will return an object of type token bucket filter. Before returning the object we'll start the daemon thread. Additionally, we don't want consumers to be able to instantiate the token bucket filter objects without interacting with the factory. For this reason, we'll make the class `MultithreadedTokenBucketFilter` private and nest it within the factory class. We'll also add an abstract `TokenBucketFilter` class that consumers can use to reference the object returned from our `makeTokenBucketFilter()` method. The class `MultithreadedTokenBucketFilter` will extend the abstract class `TokenBucketFilter`.

The complete code with the same test case appears below.

main.java

TokenBucketFilter.java

TokenBucketFilterFactory.java

```
import java.util.HashSet;
import java.util.Set;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        Set<Thread> allThreads = new HashSet<Thread>();
        TokenBucketFilter tokenBucketFilter = TokenBucketFilterFactory.makeTokenBucketFilter

        for (int i = 0; i < 10; i++) {

            Thread thread = new Thread(new Runnable() {

                public void run() {
                    try {
                        tokenBucketFilter.getToken();
                    } catch (InterruptedException ie) {
                        System.out.println("We have a problem");
                    }
                }
            });
            thread.setName("Thread_" + (i + 1));
            allThreads.add(thread);
        }

        for (Thread t : allThreads) {
            t.start();
        }

        for (Thread t : allThreads) {
            t.join();
        }

    }
}
```