

ASSIGNMENT4: THREADS

Evaluate the existing code and identify segments that lack thread safety. Provide detailed analysis in a PDF or text document.

1. **addVet** method: Multiple threads can call this method concurrently, leading to a race condition when adding a veterinarian to the **vetMap**.

At this moment, the code did not have an error

```
1 public void addVet(String name, String dep) {
2     Veterinarian vet = createDoctor(name, dep);
3     if (vet != null) {
4         PriorityQueue<Veterinarian> vets = vetMap.getOrDefault(dep, new PriorityQueue<>(Comparator.comparing(Veterinarian::getAvailability)));
5         vets.add(vet);
6         vetMap.put(dep, vets);
7     }
8 }
```

This method is not thread-safe because multiple threads can call it concurrently, leading to a race condition when adding a veterinarian to the **vetMap**.

```
38 Thread thread3 = new Thread(appointment3);
39 Thread thread4 = new Thread(appointment4);
40 Thread thread5 = new Thread(appointment5);
41
42 thread1.start();
43 thread2.start();
44 thread3.start();
45 thread4.start();
46 thread5.start();
47
48
49
50 VetBookingThread thread11 = new VetBookingThread(clinic, "Garfield", 5, "feline");
51 VetBookingThread thread12 = new VetBookingThread(clinic, "Rio", 3, "avian");
52 VetBookingThread thread13 = new VetBookingThread(clinic, "Lassie", 7, "canine");
53
54 thread11.start();
55 thread12.start();
56 thread13.start();
57
58 // Adding veterinarians
59 clinic.addVet("Dr. Smith", "avian");
60 clinic.addVet("Dr. Johnson", "feline");
61
62 // Creating multiple threads to simulate concurrent access
63 Thread thread21 = new Thread(() -> {
64     clinic.bookAppointment("Buddy", 5, "avian");
65 });
66
67 Thread thread22 = new Thread(() -> {
68     clinic.bookAppointment("Max", 3, "feline");
69 });
70
71 // Starting the threads
72 thread21.start();
73 thread22.start();
74
75 }
```

Problems • Javadoc • Declaration • Console • Coverage

<terminated> Driver [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Jun 4, 2024, 9:01:19 a.m. - 9:01:19 a.m.) [pid: 31672]

Pets in Pet Care Clinic:
Appointment scheduled with Bob for Goofy
Appointment scheduled with Mike for Garfield
Appointment scheduled with David for Buddy
Appointment scheduled with Bob for Lassie
Appointment scheduled with Bob for Lassie
Appointment scheduled with Bob for Scooby-Doo
Appointment scheduled with David for Rio
Appointment scheduled with Mike for Max
Appointment scheduled with Mike for Garfield
Exception in thread "Thread-2": Appointment scheduled with David for Rio
java.lang.ArrayIndexOutOfBoundsException: Index -1 out of bounds for length 11
at java.base/java.util.PriorityQueue.poll(PriorityQueue.java:578)
at assignment4_threads.Clinic.bookAppointment(assignment4.java:213)
at assignment4_threads.Driver.lambda\$2(Driver.java:32)
at java.base/java.lang.Thread.run(Thread.java:1583)

When I run the code more times then I get a different error but a similar type. This is because of

2. **bookAppointment** method: Multiple threads can call this method concurrently, leading to a race condition when booking an appointment with a veterinarian and updating their availability.

```
public void bookAppointment(String name, int age, String vetType) {
    PriorityQueue<Veterinarian> vets = vetMap.get(vetType);
    if (vets != null && !vets.isEmpty()) {
        Veterinarian vet = vets.peek();
        if (vet != null && vet.getAvailability()) {
            System.out.println("Appointment scheduled with " + vet.getName() + " for " + name);
            vet.setAvailability(false);
            Pet pet = new Pet(name, age, Pet.totalPets + 1);
            addPet(pet);
            vets.poll();
        } else {
            System.out.println("No available doctor for the specified type:" + vetType + " for " + name);
        }
    } else {
        System.out.println("No doctor available for the specified type:" + vetType + " for " + name);
    }
}
```

This method is not thread-safe because multiple threads can call it concurrently, leading to a race condition when booking an appointment with a veterinarian and updating their availability.

3. **vetMap**: The **PriorityQueue** used in the **vetMap** is not thread-safe, and multiple threads can modify it concurrently, leading to inconsistencies.

```
1 private Map<String, PriorityQueue<Veterinarian>> vetMap;
```

The PriorityQueue used in the vetMap is not thread-safe, and multiple threads can modify it concurrently, leading to inconsistencies.

```

125 //=====
126 // Overloaded method to add a veterinarian without synchronization
127* public void addVetUnsynchronized(String name, String dep) {
128     Veterinarian vet = createDoctor(name, dep);
129     if (vet != null) {
130         PriorityQueue<Veterinarian> vets = vetMap.getOrDefault(dep, new PriorityQueue<>(Comparator.comparing(Veterinarian::getAvailability)));
131         vets.add(vet);
132         vetMap.put(dep, vets);
133     }
134 }
135
136 // Overloaded method to book an appointment without synchronization
137* public void bookAppointmentUnsynchronized(String name, int age, String vetType) {
138     PriorityQueue<Veterinarian> vets = vetMap.get(vetType);
139     if (vets != null && !vets.isEmpty()) {
140         Veterinarian vet = vets.peek();
141         if (vet != null && vet.getAvailability()) {
142             System.out.println("Appointment scheduled with " + vet.getName() + " for " + name);
143             vet.setAvailability(false);
144             Pet pet = new Pet(name, age, Pet.totalPets + 1);
145             addPet(pet);
146             vets.poll();
147         } else {
148             System.out.println("No available doctor for the specified type:" + vetType + " for " + name);
149         }
150     } else {
151         System.out.println("No doctor available for the specified type:" + vetType + " for " + name);
152     }
153 }
154
155* public void addVet(String name, String dep) {
156     Veterinarian vet = createDoctor(name, dep);
157     if (vet != null) {
158         PriorityQueue<Veterinarian> vets = vetMap.getOrDefault(dep, new PriorityQueue<>(Comparator.comparing(Veterinarian::getAvailability)));
159         vets.add(vet);
160         vetMap.put(dep, vets);
161     }
162 }
163
164
165 //=====

```

Simulation of concurrent access with no synchronization:

To ensure thread safety for the class, we can use **synchronized** methods and blocks.
Updated **Clinic** class with synchronized methods:

```

class Clinic {
    // ...

    public synchronized void addVet(String name, String dep) {
        // ...
    }

    public synchronized void bookAppointment(String name, int age, String vetType) {
        // ...
    }

    public synchronized void addPet(Pet pet) {
        // ...
    }

    public synchronized void printPets() {
        // ...
    }
}

```

By using synchronized methods, we can ensure that only one thread can execute the method at a time, preventing concurrent access issues.

2. Evaluate synchronization mechanisms:

- **Synchronized methods or blocks:** These can be used to synchronize access to critical sections of code, ensuring only one thread can execute the synchronized block at a time. However, it can lead to contention if the synchronized block is too large or if multiple synchronized blocks need to be executed sequentially.
- **ReentrantLock:** Provides more flexibility than synchronized blocks, allowing for try-lock operations, which can be useful for avoiding deadlocks. However, it requires explicit lock acquisition and release, which can be error-prone if not used correctly.
- **Atomic variables:** Can be used for simple operations that require atomicity, like incrementing a counter. However, it might not be suitable for more complex synchronization scenarios.
- **Concurrent collections:** Offer built-in thread safety for common data structures like lists, maps, and queues. They are easy to use and often perform well in multi-threaded environments.