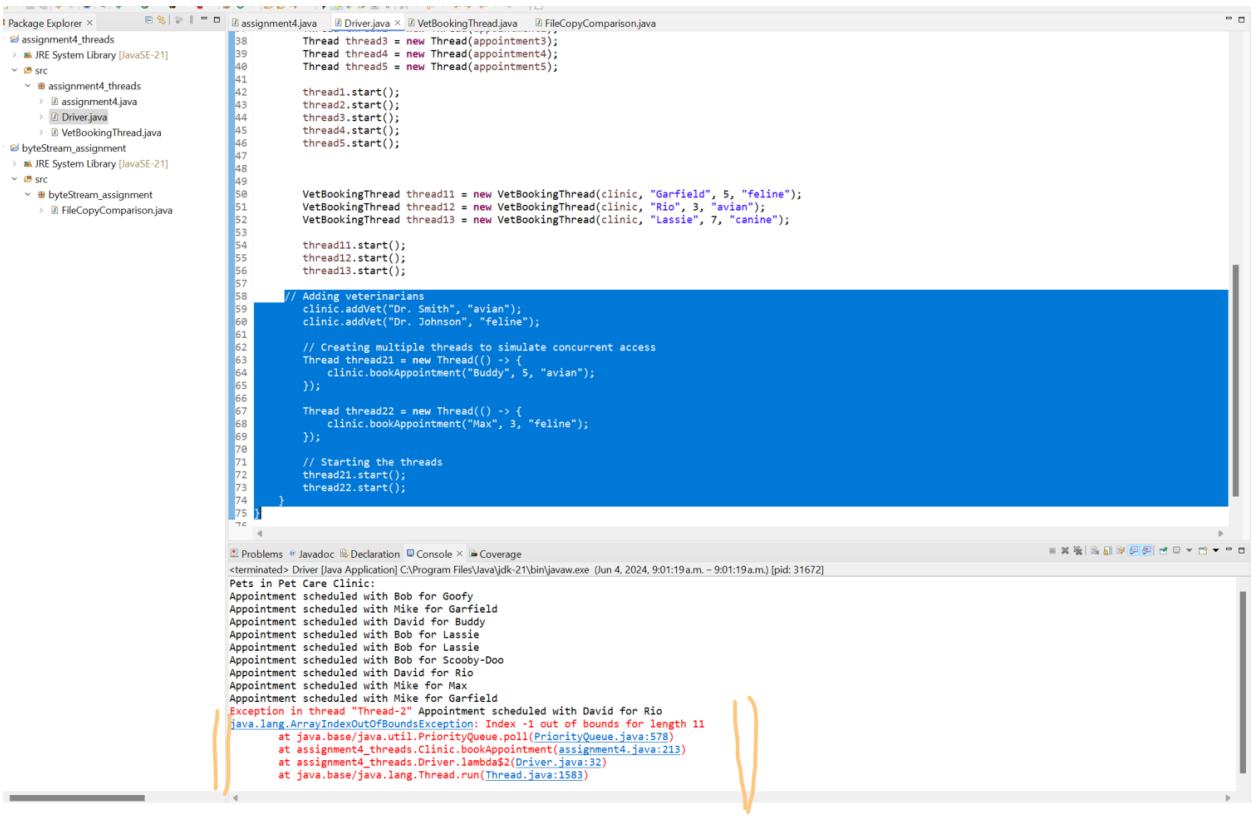
ASSIGNMENT4: THREADS

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**.



When I run the code more tiomes 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.

*assignment4.java × Driver.java VetBookingThread.java FileCopyComparison.java

```java
//=============================================================================================================
  // Overloaded method to add a veterinarian without synchronization
  public void addVetUnsynchronized(String name, String dep) {
      Veterinarian vet = createDoctor(name, dep);
      if (vet != null) {
          PriorityQueue<Veterinarian> vets = vetMap.getOrDefault(dep, new PriorityQueue<>(Comparator.comparing(Veterinarian::getAvailability)));
          vets.add(vet);
          vetMap.put(dep, vets);
      }
  }


  // Overloaded method to book an appointment without synchronization
  public void bookAppointmentUnsynchronized(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);
      }
  }

  public void addVet(String name, String dep) {
      Veterinarian vet = createDoctor(name, dep);
      if (vet != null) {
          PriorityQueue<Veterinarian> vets = vetMap.getOrDefault(dep, new PriorityQueue<>(Comparator.comparing(Veterinarian::getAvailability)));
          vets.add(vet);
          vetMap.put(dep, vets);
      }
  }

//=============================================================================================================
```

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:

```java
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.