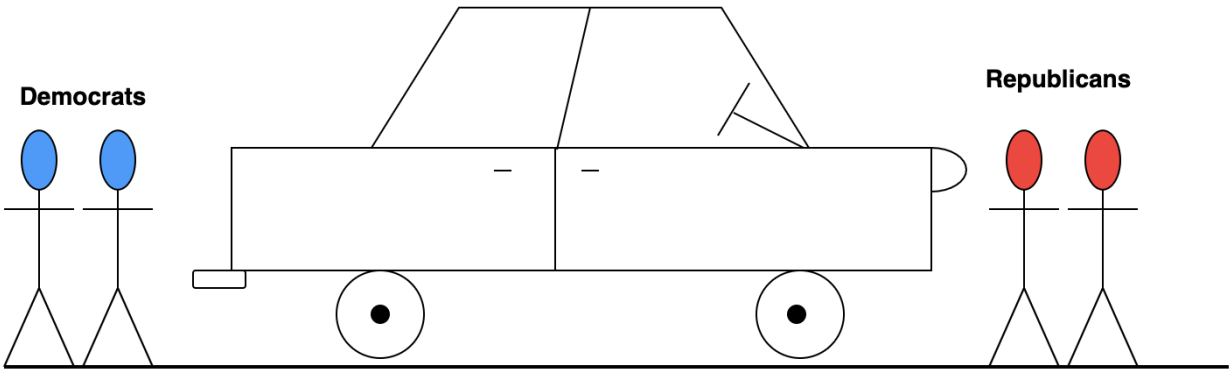
Uber Ride Problem

This lesson solves the constraints of an imaginary Uber ride problem where Republicans and Democrats can't be seated as a minority in a four passenger car.

Problem

Imagine at the end of a political conference, republicans and democrats are trying to leave the venue and ordering Uber rides at the same time. However, to make sure no fight breaks out in an Uber ride, the software developers at Uber come up with an algorithm whereby either an Uber ride can have all democrats or republicans or two Democrats and two Republicans. All other combinations can result in a fist-fight.

Your task as the Uber developer is to model the ride requestors as threads. Once an acceptable combination of riders is possible, threads are allowed to proceed to ride. Each thread invokes the method `seated()` when selected by the system for the next ride. When all the threads are seated, any one of the four threads can invoke the method `drive()` to inform the driver to start the ride.



Uber Seating Problem

Solution

First let us model the problem as a class. We'll have two methods one called by a Democrat and one by a Republican to get a ride home. When either one gets a seat on the next ride, it'll call the `seated()` method.

To make up an allowed combination of riders, we'll need to keep a count of Democrats and Republicans who have requested for rides. We create two variables for this purpose and modify them within a lock/mutex. In this problem, we'll use the `ReentrantLock` class provided by java's `util.concurrent` package when manipulating counts for democrats and republicans.

Realize we'll also need a barrier where all the four threads, that have been selected for the Uber ride arrive at, before riding away. This is analogous to the four riders being seated in the car and the doors being shut.

Once the doors are shut, one of the riders has to tell the driver to drive which we simulate with a call to the `drive()` method. Note that exactly one thread makes the shout-out to the driver to `drive()`.

The initial class skeleton looks like the following:

```
public class UberSeatingProblem {

    private int republicans = 0;
    private int democrats = 0;

    CyclicBarrier barrier = new CyclicBarrier(4);
    ReentrantLock lock = new ReentrantLock();

    void seatDemocrat() throws InterruptedException, BrokenBarrierExc
eption {
    }

    void seatRepublican() throws InterruptedException, BrokenBarrierE
xception {
    }

    void seated() {
```

```
    }


    void drive() {
    }
}
```

Let's focus on the `seatDemocrat()` method first. For simplicity imagine the first thread is a democrat and invokes `seatDemocrat()`. Since there's no other rider available, it should be put to wait. We can use a semaphore to make this thread wait. We'll not use a barrier, because we don't know what party loyalty the threads arriving in future would have. It might be that the next four threads are all republican and this Democrat isn't placed on the next Uber ride. To differentiate between waiting democrats and waiting republicans, we'll use two different semaphores `demsWaiting` and `repubsWaiting`. Our first democrat thread will end up `acquire()`-ing the `demsWaiting` semaphore.

Now it's easy to reason about how we select the threads for a ride. A democrat thread has to check the following cases:

- If there are already 3 waiting democrats, then we signal the `demsWaiting` three times so that all these four democrats can ride together in the next Uber ride.
- If there are two or more republican threads waiting and at least two democrat threads (including the current thread) waiting, then the current democrat thread can signal the `repubsWaiting` semaphore twice to release the two waiting republican threads and signal the `demsWaiting` semaphore once to release one more democrat thread. Together the four of them would make up the next ride consisting of two republican and two democrats.
- If the above two conditions aren't true then the current democrat thread should simply wait itself at the `demsWaiting` semaphore and release the lock object so that other threads can enter the critical sections.

The logic we discussed so far is translated into code below:

```
    void seatDemocrat() throws InterruptedException, BrokenBarrierExc
eption {

        boolean rideLeader = false;
        lock.lock();

        democrats++;

        if (democrats == 4) {
            // Seat all the democrats in the Uber ride.
            demsWaiting.release(3);
            democrats -= 4;
            rideLeader = true;
        } else if (democrats == 2 && republicans >= 2) {
            // Seat 2 democrats & 2 republicans
            demsWaiting.release(1);
            repubsWaiting.release(2);
            rideLeader = true;
            democrats -= 2;
            republicans -= 2;
        } else {
            lock.unlock();
            demsWaiting.acquire();
        }

        seated();
        barrier.await();

        if (rideLeader == true) {
            drive();
            lock.unlock();
        }
    }
```

The thread that signals other threads to come along for the ride marks itself as the `rideLeader`. This thread is responsible for informing the driver to `drive()`. We can come up with some other criteria to choose the rider leader but given the logic we implemented, it is easiest to make the thread that determines an acceptable ride combination as the ride leader.

The republicans' `seatRepublican()` method is analogous to the `seatDemocrat()` method.

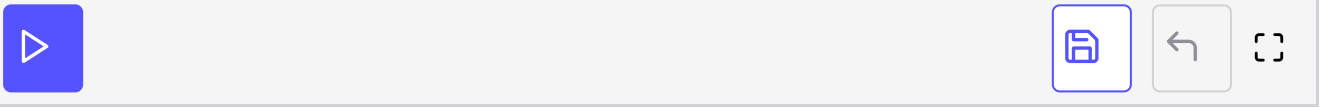
Complete Code

The complete code appears below:

```
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.ReentrantLock;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        UberSeatingProblem.runTest();
    }
}

class UberSeatingProblem {

    private int republicans = 0;
    private int democrats = 0;

    private Semaphore demsWaiting = new Semaphore(0);
    private Semaphore repubsWaiting = new Semaphore(0);

    CyclicBarrier barrier = new CyclicBarrier(4);
    ReentrantLock lock = new ReentrantLock();

    void drive() {
        System.out.println("Uber Ride on Its wayyyy... with ride leader " + Thread.currentThr
        System.out.flush();
    }

    void seatDdemocrat() throws InterruptedException, BrokenBarrierException {

        boolean rideLeader = false;
        lock.lock();

        democrats++;

        if (democrats == 4) {
            // Seat all the democrats in the Uber ride.
            demsWaiting.release(3);
            democrats -= 4;
            rideLeader = true;
        } else if (democrats == 2 && republicans >= 2) {
            // Seat 2 democrats & 2 republicans
            demsWaiting.release(1);
            repubsWaiting.release(2);
            rideLeader = true;
            democrats -= 2;
            republicans -= 2;
```

```
        } else {
            lock.unlock();
            demsWaiting.acquire();

        }

        seated();
        barrier.await();

        if (rideLeader == true) {
            drive();
            lock.unlock();
        }
    }

    void seated() {
        System.out.println(Thread.currentThread().getName() + "  seated");
        System.out.flush();
    }

    void seatRepublican() throws InterruptedException, BrokenBarrierException {

        boolean rideLeader = false;
        lock.lock();

        republicans++;

        if (republicans == 4) {
            // Seat all the republicans in the Uber ride.
            repubsWaiting.release(3);
            rideLeader = true;
            republicans -= 4;
        } else if (republicans == 2 && democrats >= 2) {
            // Seat 2 democrats & 2 republicans
            repubsWaiting.release(1);
            demsWaiting.release(2);
            rideLeader = true;
            republicans -= 2;
            democrats -= 2;
        } else {
            lock.unlock();
            repubsWaiting.acquire();
        }

        seated();
        barrier.await();

        if (rideLeader) {
            drive();
            lock.unlock();
        }
    }

    public static void runTest() throws InterruptedException {


        final UberSeatingProblem uberSeatingProblem = new UberSeatingProblem();
        Set<Thread> allThreads = new HashSet<Thread>();

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

            Thread thread = new Thread(new Runnable() {
                public void run() {
```

```
                    try {
                        uberSeatingProblem.seatDemocrat();
                    } catch (InterruptedException ie) {
                        System.out.println("We have a problem");

                    } catch (BrokenBarrierException bbe) {
                        System.out.println("We have a problem");
                    }

                }
            });
            thread.setName("Democrat_" + (i + 1));
            allThreads.add(thread);

            Thread.sleep(50);
        }

        for (int i = 0; i < 14; i++) {
            Thread thread = new Thread(new Runnable() {
                public void run() {
                    try {
                        uberSeatingProblem.seatRepublican();
                    } catch (InterruptedException ie) {
                        System.out.println("We have a problem");

                    } catch (BrokenBarrierException bbe) {
                        System.out.println("We have a problem");
                    }
                }
            });
            thread.setName("Republican_" + (i + 1));
            allThreads.add(thread);
            Thread.sleep(20);
        }

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

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



The output of the program will show the members of each ride. Since we create four more republican threads than democrat threads, you should see at least one ride with all republican riders.

The astute reader may wonder what factor determines that a ride is evenly split between members of the two parties or entirely made up of

the members of the same party, given enough riders exist that both combinations can be possible. The key is to realize that each thread enters

the critical sections `seatDemocrat()` or `seatRepublican()` one at a time because of the lock at the beginning of the two methods. Whether a ride is evenly split between the two types of riders or consists entirely of one type of riders depends upon the order in which the threads enter the critical section. For instance, if we create four democrat and four republican threads then we can either get two rides each split between the two types or two rides each made of the same type. If the first four threads to sequentially enter the critical sections are democrat, then the two rides will be made up of either entirely democrats or republicans. Since threads are scheduled for execution non-deterministically, we can't be certain what would be the makeup of each ride.