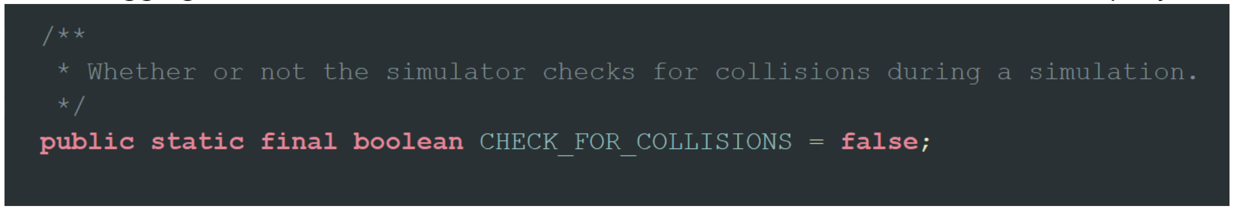
AIM4 – Release 1.0.4 – Fixed collisions – Code changes notes

Debug.java:

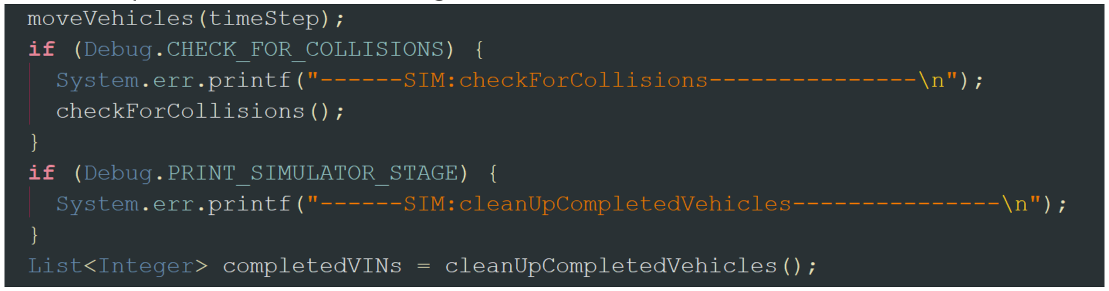
- Added a constant used for debugging collisions. If on (true), the simulator will check for collisions at each step. By default, set to false.



```java
/**
 * Whether or not the simulator checks for collisions during a simulation.
 */
public static final boolean CHECK_FOR_COLLISIONS = false;
```

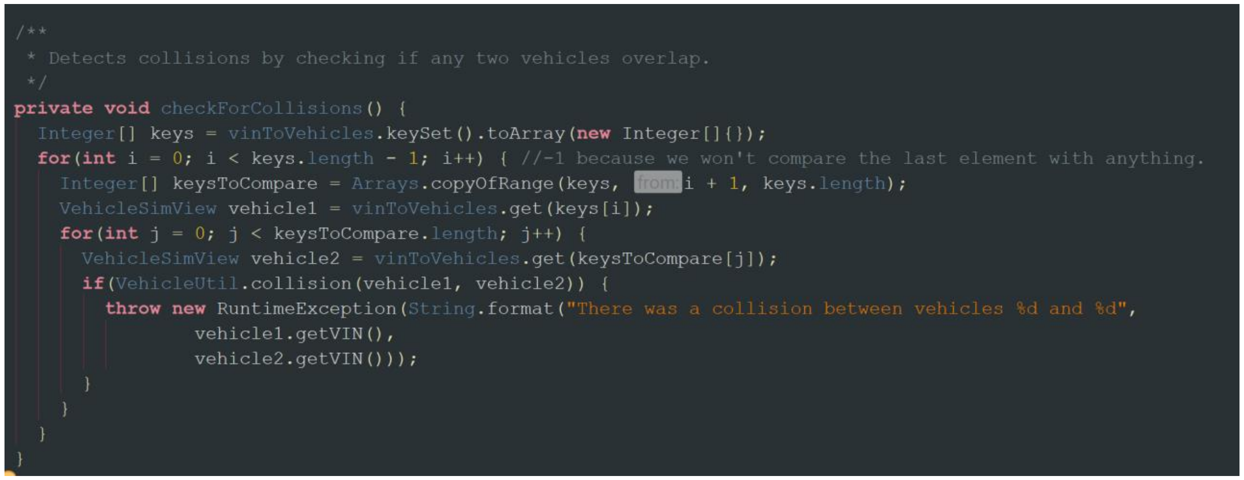
AutoDriverOnlySimulator.java:

- Added a check for collisions in the step() method after moving the vehicles:



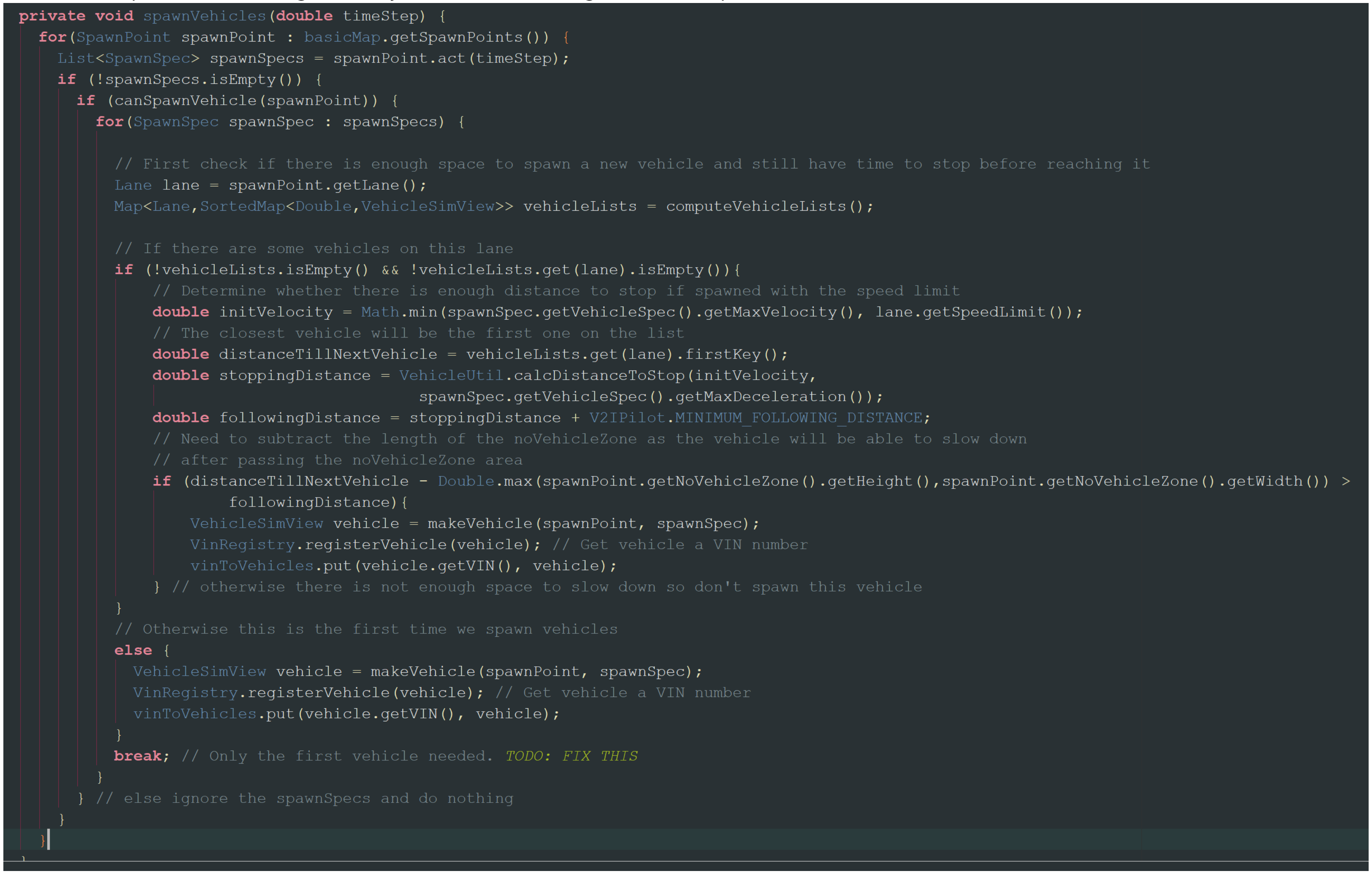
```java
moveVehicles(timeStep);
if (Debug.CHECK_FOR_COLLISIONS) {
  System.err.printf("------SIM:checkForCollisions---------------\n");
  checkForCollisions();
}
if (Debug.PRINT_SIMULATOR_STAGE) {
  System.err.printf("------SIM:cleanUpCompletedVehicles---------------\n");
}
List<Integer> completedVINs = cleanUpCompletedVehicles();
```

- The checkForCollisions() method will check if a collision occurs between any two vehicles. Does an overlap of the shape bounds of all vehicles.



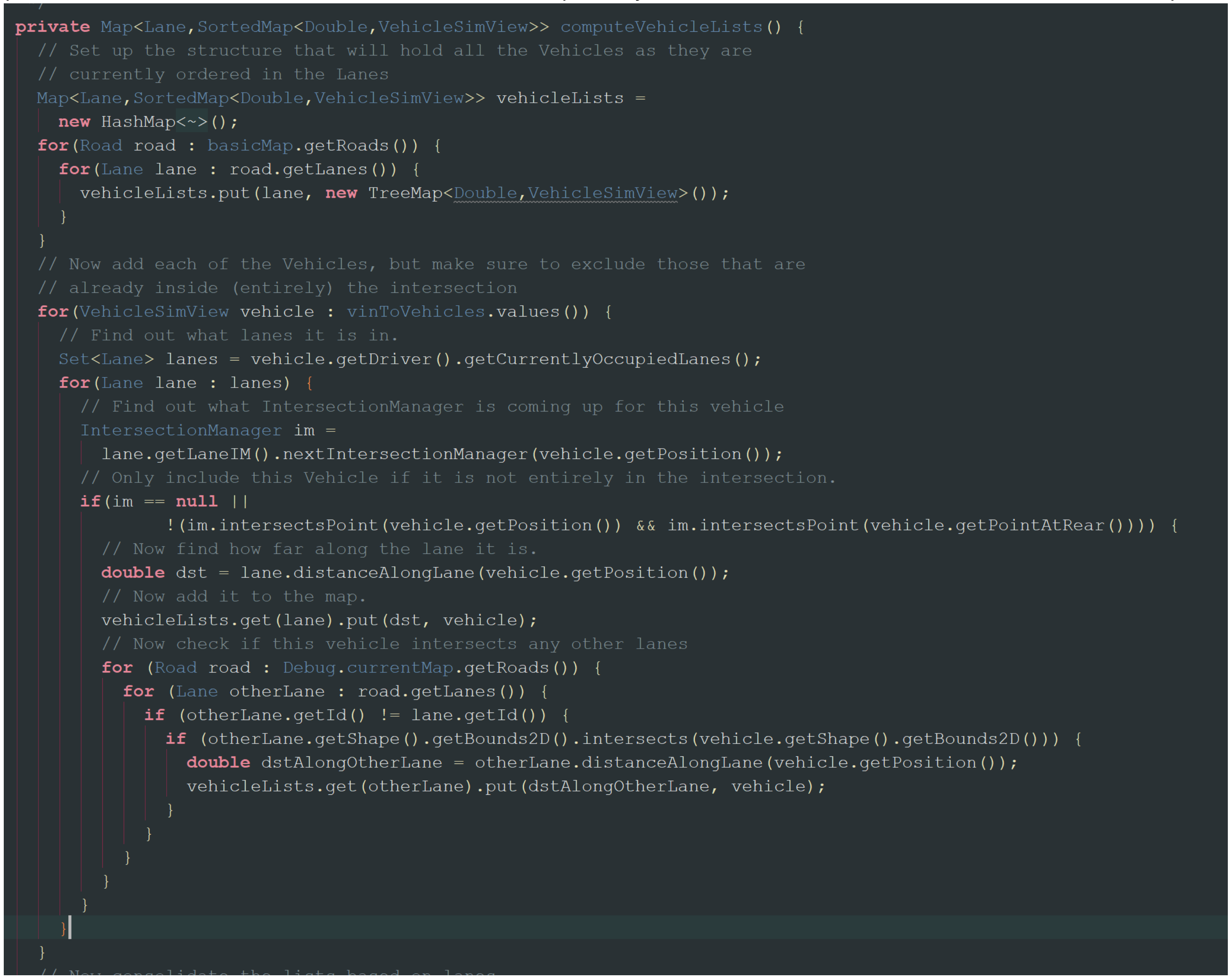
```java
/**
 * Detects collisions by checking if any two vehicles overlap.
 */
private void checkForCollisions() {
  Integer[] keys = vinToVehicles.keySet().toArray(new Integer[]{});
  for(int i = 0; i < keys.length - 1; i++) { //-1 because we won't compare the last element with anything.
    Integer[] keysToCompare = Arrays.copyOfRange(keys, from: i + 1, keys.length);
    VehicleSimView vehicle1 = vinToVehicles.get(keys[i]);
    for(int j = 0; j < keysToCompare.length; j++) {
      VehicleSimView vehicle2 = vinToVehicles.get(keysToCompare[j]);
      if(VehicleUtil.collision(vehicle1, vehicle2)) {
        throw new RuntimeException(String.format("There was a collision between vehicles %d and %d",
                vehicle1.getVIN(),
                vehicle2.getVIN()));
      }
    }
  }
}
```

- Changed the spwanVehicles() method such that not to spawn vehicles if there is not enough space for them to stop before reaching the next vehicle. Collisions happened before this change when at high velocities and very high traffic volume, the map will be filled with vehicles and a new vehicle spawned with a high velocity did not have enough time to stop and collided into the vehicle in front:



```java
private void spawnVehicles(double timeStep) {
  for(SpawnPoint spawnPoint : basicMap.getSpawnPoints()) {
    List<SpawnSpec> spawnSpecs = spawnPoint.act(timeStep);
    if (!spawnSpecs.isEmpty()) {
      if (canSpawnVehicle(spawnPoint)) {
        for(SpawnSpec spawnSpec : spawnSpecs) {

          // First check if there is enough space to spawn a new vehicle and still have time to stop before reaching it
          Lane lane = spawnPoint.getLane();
          Map<Lane,SortedMap<Double,VehicleSimView>> vehicleLists = computeVehicleLists();

          // If there are some vehicles on this lane
          if (!vehicleLists.isEmpty() && !vehicleLists.get(lane).isEmpty()){
              // Determine whether there is enough distance to stop if spawned with the speed limit
              double initVelocity = Math.min(spawnSpec.getVehicleSpec().getMaxVelocity(), lane.getSpeedLimit());
              // The closest vehicle will be the first one on the list
              double distanceTillNextVehicle = vehicleLists.get(lane).firstKey();
              double stoppingDistance = VehicleUtil.calcDistanceToStop(initVelocity,
                                            spawnSpec.getVehicleSpec().getMaxDeceleration());
              double followingDistance = stoppingDistance + V2IPilot.MINIMUM_FOLLOWING_DISTANCE;
              // Need to subtract the length of the noVehicleZone as the vehicle will be able to slow down
              // after passing the noVehicleZone area
              if (distanceTillNextVehicle - Double.max(spawnPoint.getNoVehicleZone().getHeight(),spawnPoint.getNoVehicleZone().getWidth()) >
                      followingDistance){
                  VehicleSimView vehicle = makeVehicle(spawnPoint, spawnSpec);
                  VinRegistry.registerVehicle(vehicle); // Get vehicle a VIN number
                  vinToVehicles.put(vehicle.getVIN(), vehicle);
              } // otherwise there is not enough space to slow down so don't spawn this vehicle
          }
          // Otherwise this is the first time we spawn vehicles
          else {
            VehicleSimView vehicle = makeVehicle(spawnPoint, spawnSpec);
            VinRegistry.registerVehicle(vehicle); // Get vehicle a VIN number
            vinToVehicles.put(vehicle.getVIN(), vehicle);
          }
          break; // Only the first vehicle needed. TODO: FIX THIS
        }
      } // else ignore the spawnSpecs and do nothing
    }
  }
```

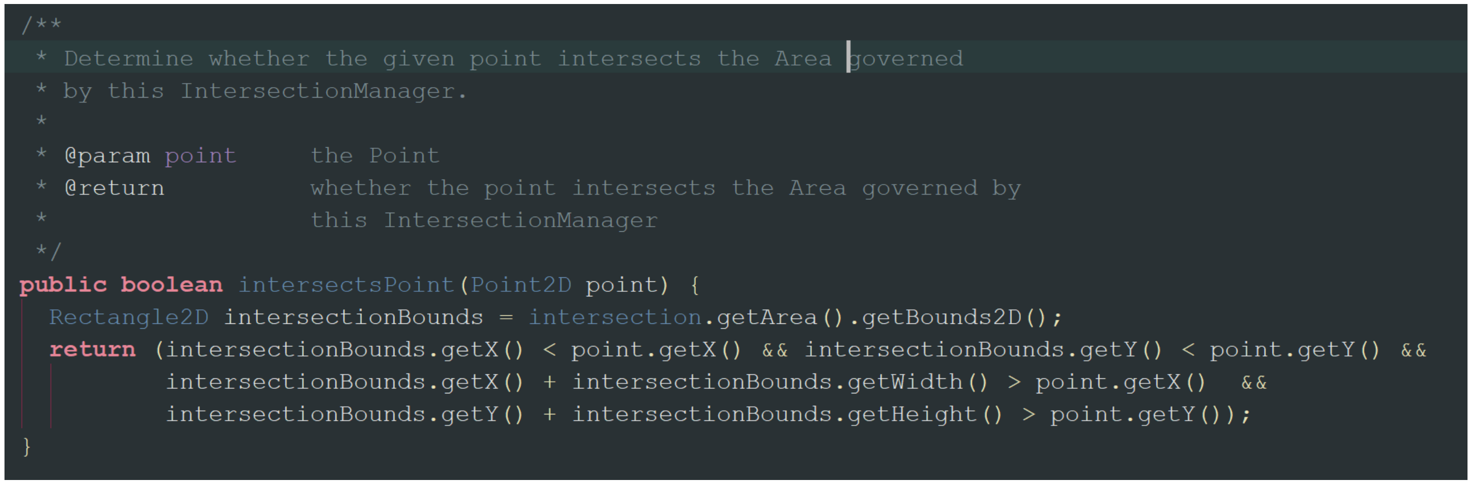
- Changed computeVehicleLists() method such that to also include partially entered vehicles and all the lanes a vehicle occupies.



```
private Map<Lane,SortedMap<Double,VehicleSimView>> computeVehicleLists() {
  // Set up the structure that will hold all the Vehicles as they are
  // currently ordered in the Lanes
  Map<Lane,SortedMap<Double,VehicleSimView>> vehicleLists =
    new HashMap<~>();
  for(Road road : basicMap.getRoads()) {
    for(Lane lane : road.getLanes()) {
      vehicleLists.put(lane, new TreeMap<Double,VehicleSimView>());
    }
  }
  // Now add each of the Vehicles, but make sure to exclude those that are
  // already inside (entirely) the intersection
  for(VehicleSimView vehicle : vinToVehicles.values()) {
    // Find out what lanes it is in.
    Set<Lane> lanes = vehicle.getDriver().getCurrentlyOccupiedLanes();
    for(Lane lane : lanes) {
      // Find out what IntersectionManager is coming up for this vehicle
      IntersectionManager im =
        lane.getLaneIM().nextIntersectionManager(vehicle.getPosition());
      // Only include this Vehicle if it is not entirely in the intersection.
      if(im == null ||
              !(im.intersectsPoint(vehicle.getPosition()) && im.intersectsPoint(vehicle.getPointAtRear()))) {
        // Now find how far along the lane it is.
        double dst = lane.distanceAlongLane(vehicle.getPosition());
        // Now add it to the map.
        vehicleLists.get(lane).put(dst, vehicle);
        // Now check if this vehicle intersects any other lanes
        for (Road road : Debug.currentMap.getRoads()) {
          for (Lane otherLane : road.getLanes()) {
            if (otherLane.getId() != lane.getId()) {
              if (otherLane.getShape().getBounds2D().intersects(vehicle.getShape().getBounds2D())) {
                double dstAlongOtherLane = otherLane.distanceAlongLane(vehicle.getPosition());
                vehicleLists.get(otherLane).put(dstAlongOtherLane, vehicle);
              }
            }
          }
        }
      }
    }
  }
```

IntersectionManager.java:

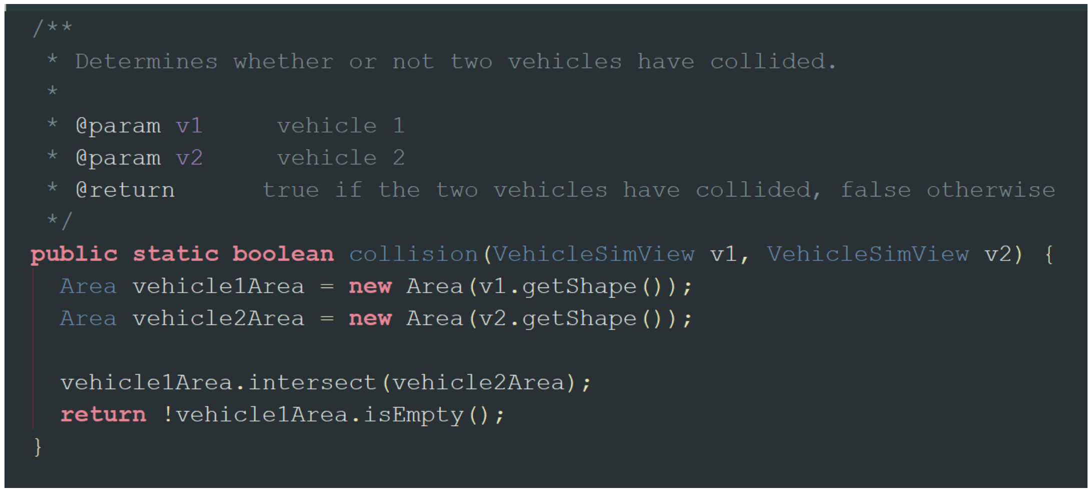
- Added method intersectsPoint() used in the simulator when computing vehicle lists to see if certain point parts of a vehicle is inside the area of the intersection:



```java
/**
 * Determine whether the given point intersects the Area governed
 * by this IntersectionManager.
 *
 * @param point     the Point
 * @return          whether the point intersects the Area governed by
 *                  this IntersectionManager
 */
public boolean intersectsPoint(Point2D point) {
  Rectangle2D intersectionBounds = intersection.getArea().getBounds2D();
  return (intersectionBounds.getX() < point.getX() && intersectionBounds.getY() < point.getY() &&
          intersectionBounds.getX() + intersectionBounds.getWidth() > point.getX()  &&
          intersectionBounds.getY() + intersectionBounds.getHeight() > point.getY());
}
```

VehicleUtil.java:

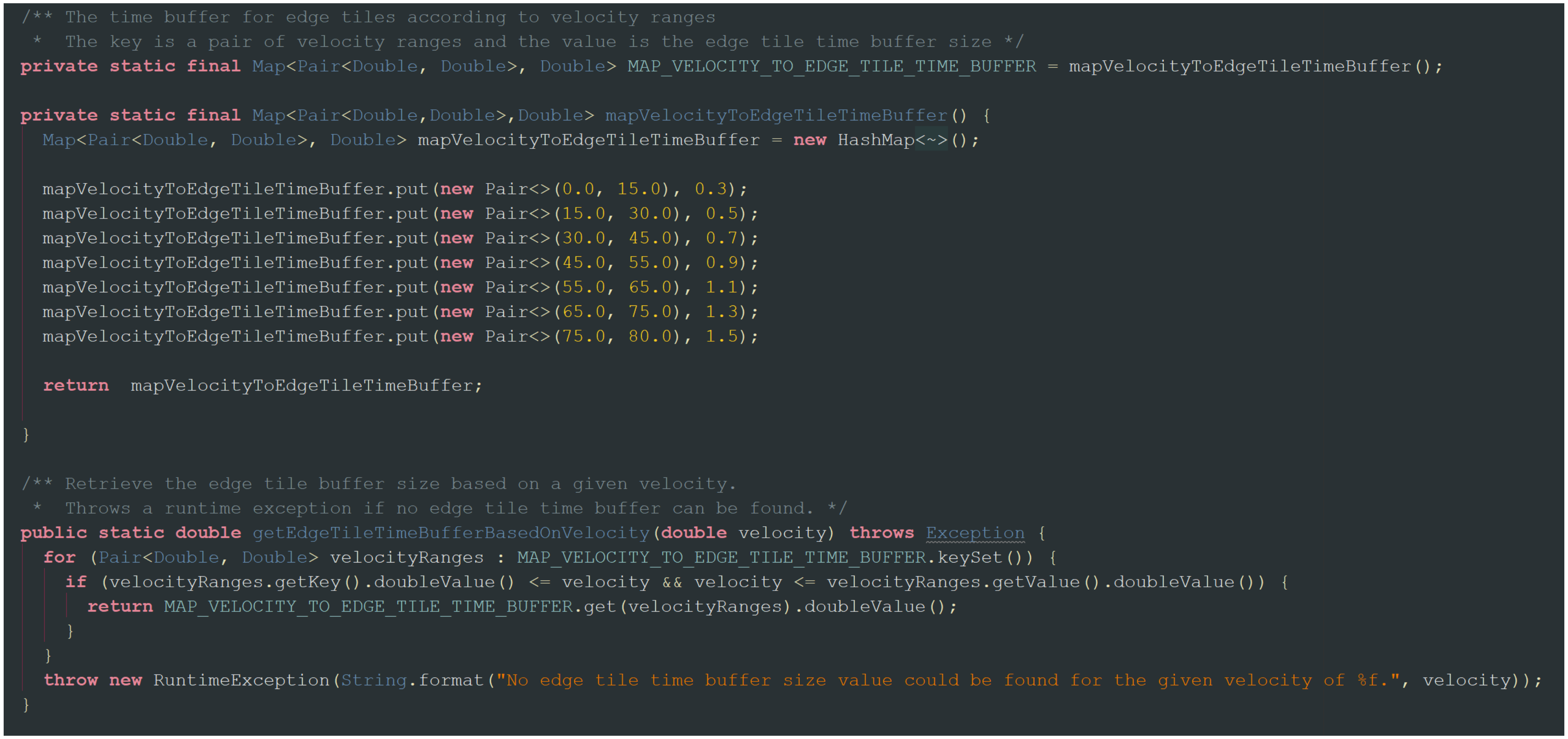
- Added a method to check the overall between two vehicles area:



```java
/**
 * Determines whether or not two vehicles have collided.
 *
 * @param v1     vehicle 1
 * @param v2     vehicle 2
 * @return      true if the two vehicles have collided, false otherwise
 */
public static boolean collision(VehicleSimView v1, VehicleSimView v2) {
  Area vehicle1Area = new Area(v1.getShape());
  Area vehicle2Area = new Area(v2.getShape());

  vehicle1Area.intersect(vehicle2Area);
  return !vehicle1Area.isEmpty();
}
```

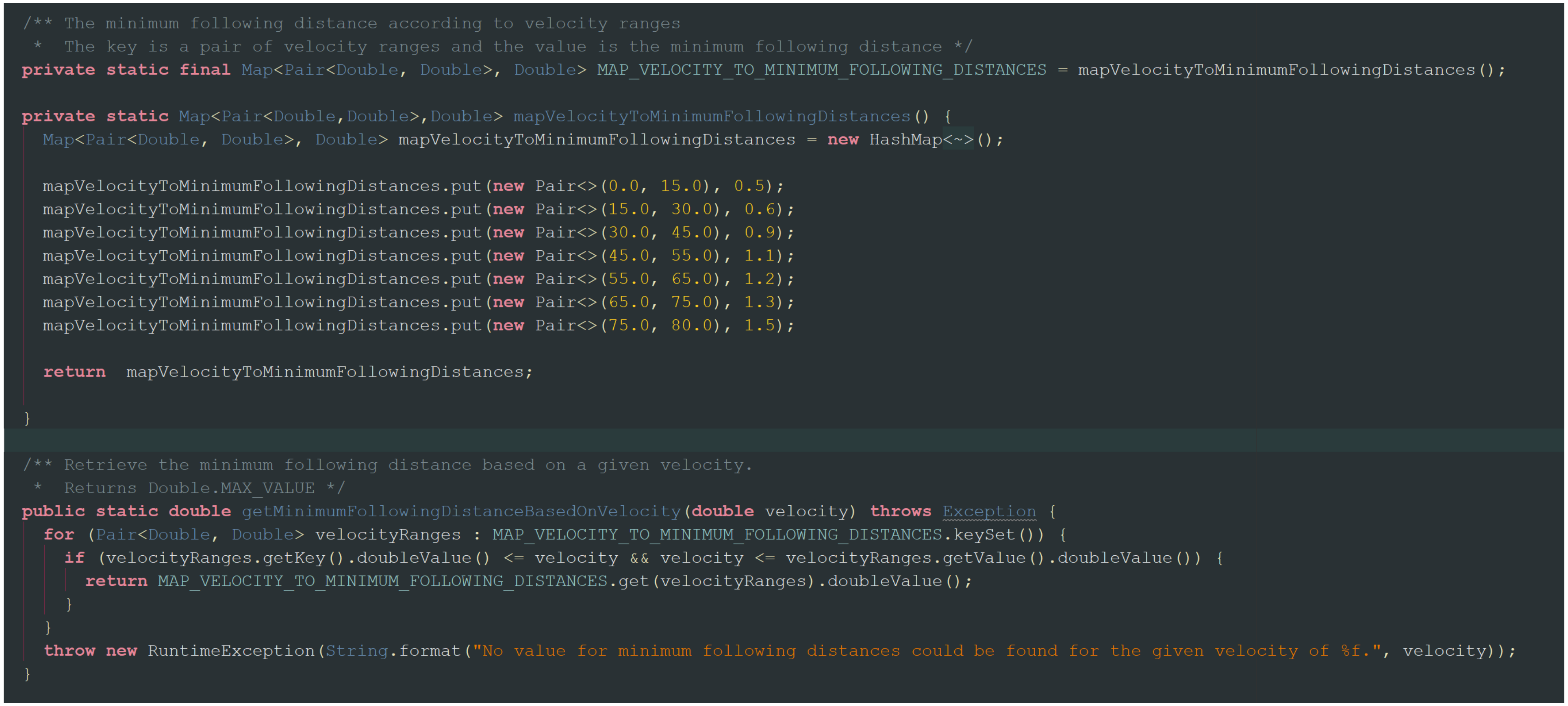
Constants.java:

- Added a Map to hold edge tile time buffer sizes for a range of velocities. The key of the map is a pair where the first element is the lower bound velocity and second element is the upper bound velocity. These two elements define the range. The values in the map define the edge tile time buffer sizes.
- Also added a get method to retrieve the edge tile time buffer size based on a given velocity. It throws an error is no such value is found.



```java
/** The time buffer for edge tiles according to velocity ranges
 *  The key is a pair of velocity ranges and the value is the edge tile time buffer size */
private static final Map<Pair<Double, Double>, Double> MAP_VELOCITY_TO_EDGE_TILE_TIME_BUFFER = mapVelocityToEdgeTileTimeBuffer();

private static final Map<Pair<Double,Double>,Double> mapVelocityToEdgeTileTimeBuffer() {
  Map<Pair<Double, Double>, Double> mapVelocityToEdgeTileTimeBuffer = new HashMap<~>();

  mapVelocityToEdgeTileTimeBuffer.put(new Pair<>(0.0, 15.0), 0.3);
  mapVelocityToEdgeTileTimeBuffer.put(new Pair<>(15.0, 30.0), 0.5);
  mapVelocityToEdgeTileTimeBuffer.put(new Pair<>(30.0, 45.0), 0.7);
  mapVelocityToEdgeTileTimeBuffer.put(new Pair<>(45.0, 55.0), 0.9);
  mapVelocityToEdgeTileTimeBuffer.put(new Pair<>(55.0, 65.0), 1.1);
  mapVelocityToEdgeTileTimeBuffer.put(new Pair<>(65.0, 75.0), 1.3);
  mapVelocityToEdgeTileTimeBuffer.put(new Pair<>(75.0, 80.0), 1.5);

  return  mapVelocityToEdgeTileTimeBuffer;

}

/** Retrieve the edge tile buffer size based on a given velocity.
 *  Throws a runtime exception if no edge tile time buffer can be found. */
public static double getEdgeTileTimeBufferBasedOnVelocity(double velocity) throws Exception {
  for (Pair<Double, Double> velocityRanges : MAP_VELOCITY_TO_EDGE_TILE_TIME_BUFFER.keySet()) {
    if (velocityRanges.getKey().doubleValue() <= velocity && velocity <= velocityRanges.getValue().doubleValue()) {
      return MAP_VELOCITY_TO_EDGE_TILE_TIME_BUFFER.get(velocityRanges).doubleValue();
    }
  }
  throw new RuntimeException(String.format("No edge tile time buffer size value could be found for the given velocity of %f.", velocity));
}
```

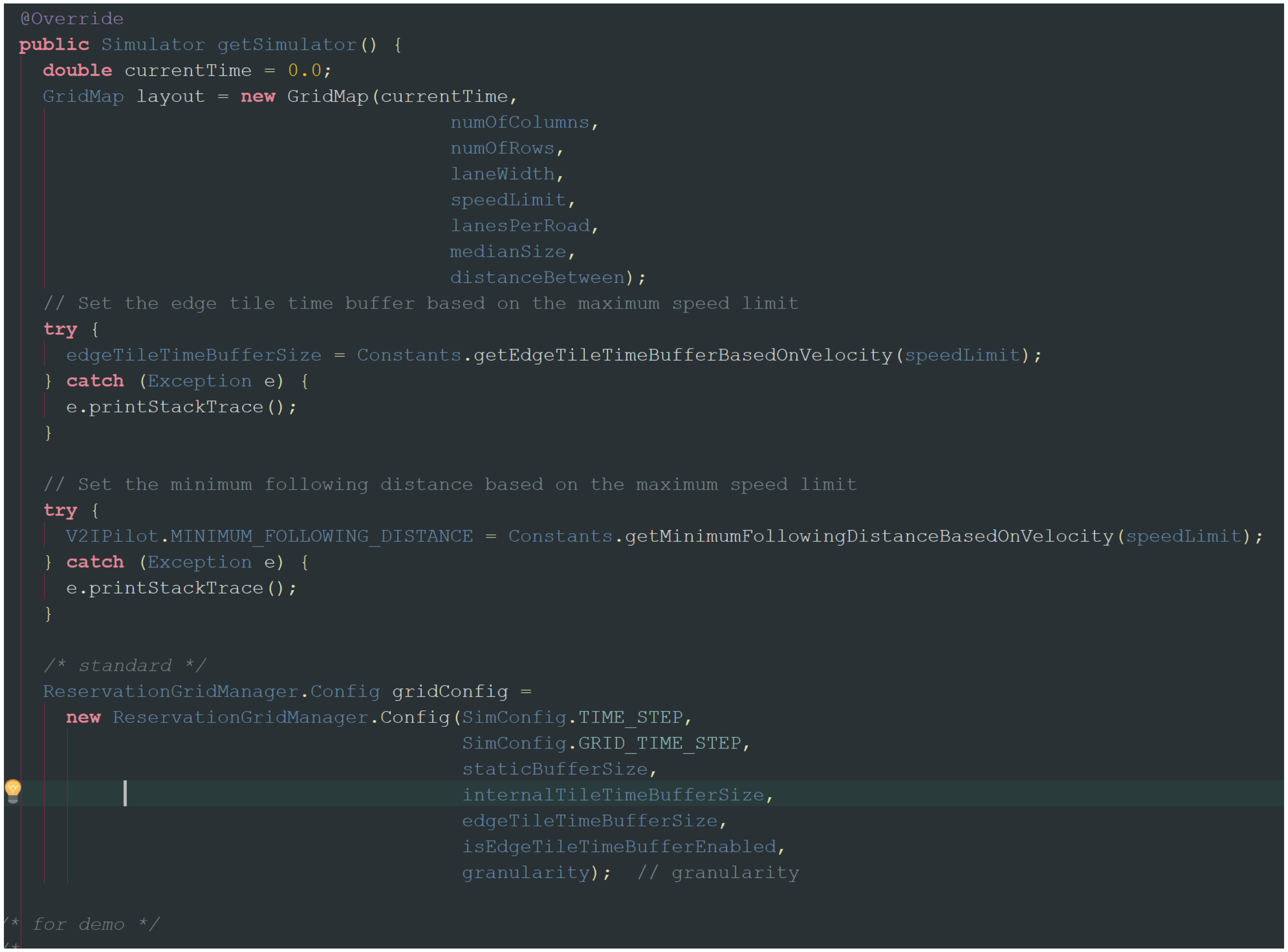
- Added a Map to hold minimum following distances for a range of velocities. The key of the map is a pair where the first element is the lower bound velocity and second element is the upper bound velocity. These two elements define the range. The values in the map define the minimum following distances.
- Also added a get method to retrieve the minimum following distance based on a given velocity. It throws an error is no value is found.



```java
/** The minimum following distance according to velocity ranges
 *  The key is a pair of velocity ranges and the value is the minimum following distance */
private static final Map<Pair<Double, Double>, Double> MAP_VELOCITY_TO_MINIMUM_FOLLOWING_DISTANCES = mapVelocityToMinimumFollowingDistances();

private static Map<Pair<Double,Double>,Double> mapVelocityToMinimumFollowingDistances() {
  Map<Pair<Double, Double>, Double> mapVelocityToMinimumFollowingDistances = new HashMap<~>();

  mapVelocityToMinimumFollowingDistances.put(new Pair<>(0.0, 15.0), 0.5);
  mapVelocityToMinimumFollowingDistances.put(new Pair<>(15.0, 30.0), 0.6);
  mapVelocityToMinimumFollowingDistances.put(new Pair<>(30.0, 45.0), 0.9);
  mapVelocityToMinimumFollowingDistances.put(new Pair<>(45.0, 55.0), 1.1);
  mapVelocityToMinimumFollowingDistances.put(new Pair<>(55.0, 65.0), 1.2);
  mapVelocityToMinimumFollowingDistances.put(new Pair<>(65.0, 75.0), 1.3);
  mapVelocityToMinimumFollowingDistances.put(new Pair<>(75.0, 80.0), 1.5);

  return  mapVelocityToMinimumFollowingDistances;

}

/** Retrieve the minimum following distance based on a given velocity.
 *  Returns Double.MAX_VALUE */
public static double getMinimumFollowingDistanceBasedOnVelocity(double velocity) throws Exception {
  for (Pair<Double, Double> velocityRanges : MAP_VELOCITY_TO_MINIMUM_FOLLOWING_DISTANCES.keySet()) {
    if (velocityRanges.getKey().doubleValue() <= velocity && velocity <= velocityRanges.getValue().doubleValue()) {
      return MAP_VELOCITY_TO_MINIMUM_FOLLOWING_DISTANCES.get(velocityRanges).doubleValue();
    }
  }
  throw new RuntimeException(String.format("No value for minimum following distances could be found for the given velocity of %f.", velocity));
}
```

AutoDriverOnlySimSetup.java:

- In the getSimulator() method call the methods from Constant.java class to retrieve the edge tile time buffer size needed and the minimum following distance needed.



```
@Override
public Simulator getSimulator() {
  double currentTime = 0.0;
  GridMap layout = new GridMap(currentTime,
                                   numOfColumns,
                                   numOfRows,
                                   laneWidth,
                                   speedLimit,
                                   lanesPerRoad,
                                   medianSize,
                                   distanceBetween);
  // Set the edge tile time buffer based on the maximum speed limit
  try {
    edgeTileTimeBufferSize = Constants.getEdgeTileTimeBufferBasedOnVelocity(speedLimit);
  } catch (Exception e) {
    e.printStackTrace();
  }

  // Set the minimum following distance based on the maximum speed limit
  try {
    V2IPilot.MINIMUM_FOLLOWING_DISTANCE = Constants.getMinimumFollowingDistanceBasedOnVelocity(speedLimit);
  } catch (Exception e) {
    e.printStackTrace();
  }

  /* standard */
  ReservationGridManager.Config gridConfig =
    new ReservationGridManager.Config(SimConfig.TIME_STEP,
                                      SimConfig.GRID_TIME_STEP,
                                      staticBufferSize,
                                      internalTileTimeBufferSize,
                                      edgeTileTimeBufferSize,
                                      isEdgeTileTimeBufferEnabled,
                                      granularity);  // granularity

/* for demo */
```