1 CyclingPortal.java

```java
package cycling;

import java.util.Arrays;
import java.util.Comparator;
import java.util.HashMap;
import java.io.IOException;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.ArrayList;
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.FileInputStream;



/**
 * CyclingPortal implements CyclingPortalInterface; contains
    methods for
 * handling the following classes: Race, Stage, Segment,
    RiderManager (and in
 * turn Rider and Team), and Result.
 * These classes are used manage races and their subdivisions,
    teams and their
 * riders, and to calculate and assign points.
 * Also contains methods for saving and loading
    MiniCyclingPortalInterface to
 * and from a file.
 *
 * @author Ethan Ray & Thomas Newbold
 * @version 1.0
 *
 */
public class CyclingPortal implements CyclingPortalInterface {
    public RiderManager riderManager = new RiderManager();

    @Override
    public int[] getRaceIds() {
        return Race.getAllRaceIds();
    }

    @Override
    public int createRace(String name, String description)
       throws IllegalNameException, InvalidNameException {
        Race r = new Race(name, description);
        return r.getRaceId();
    }

    @Override
    public String viewRaceDetails(int raceId) throws
        IDNotRecognisedException {
        double sum = 0.0;
        for(int id : Race.getStages(raceId)) {
            sum += Stage.getStageLength(id);
        }
        return Race.toString(raceId)+Double.toString(sum)+";";
    }

    @Override
    public void removeRaceById(int raceId) throws
        IDNotRecognisedException {
        Race.removeRace(raceId);
    }

    @Override
    public int getNumberOfStages(int raceId) throws
        IDNotRecognisedException {
        int[] stageIds = Race.getStages(raceId);
        return stageIds.length;
    }

    @Override
    public int addStageToRace(int raceId, String stageName,
        String description, double length, LocalDateTime
        startTime,
            StageType type)
            throws IDNotRecognisedException,
                IllegalNameException, InvalidNameException,
                InvalidLengthException {
        return Race.addStageToRace(raceId, stageName,
            description, length, startTime, type);
    }

    @Override
    public int[] getRaceStages(int raceId) throws
        IDNotRecognisedException {
        return Race.getStages(raceId);
    }

    @Override
    public double getStageLength(int stageId) throws
        IDNotRecognisedException {
        return Stage.getStageLength(stageId);
    }

    @Override
    public void removeStageById(int stageId) throws
```

```
        IDNotRecognisedException {
        Race.removeStage(stageId);
    }

    @Override
    public int addCategorizedClimbToStage(int stageId, Double
        location, SegmentType type, Double averageGradient,
            Double length) throws IDNotRecognisedException,
                InvalidLocationException,
                InvalidStageStateException,
            InvalidStageTypeException {
        return Stage.addSegmentToStage(stageId, location, type,
             averageGradient, length);
    }

    @Override
    public int addIntermediateSprintToStage(int stageId, double
         location) throws IDNotRecognisedException,
            InvalidLocationException,
                InvalidStageStateException,
                InvalidStageTypeException {
        return Stage.addSegmentToStage(stageId, location,
            SegmentType.SPRINT, 0.0, 0.0);
    }

    @Override
    public void removeSegment(int segmentId) throws
        IDNotRecognisedException, InvalidStageStateException {
        Stage.removeSegment(segmentId);
    }

    @Override
    public void concludeStagePreparation(int stageId) throws
        IDNotRecognisedException, InvalidStageStateException {
        Stage.updateStageState(stageId);
    }

    @Override
    public int[] getStageSegments(int stageId) throws
        IDNotRecognisedException {
        return Stage.getSegments(stageId);
    }

    @Override
    public int createTeam(String name, String description)
        throws IllegalNameException, InvalidNameException {
        return riderManager.createTeam(name, description);
    }

    @Override
```

```java
    public void removeTeam(int teamId) throws
        IDNotRecognisedException {
        riderManager.removeTeam(teamId);

    }

    @Override
    public int[] getTeams() {
        return riderManager.getTeams();
    }

    @Override
    public int[] getTeamRiders(int teamId) throws
        IDNotRecognisedException {
        return riderManager.getTeamRiders(teamId);
    }

    @Override
    public int createRider(int teamID, String name, int
        yearOfBirth) throws IDNotRecognisedException,
        IllegalArgumentException {
        return riderManager.createRider(teamID, name,
            yearOfBirth);

    }

    @Override
    public void removeRider(int riderId) throws
        IDNotRecognisedException {
        riderManager.removeRider(riderId);

    }

    @Override
    public void registerRiderResultsInStage(int stageId, int
        riderId, LocalTime... checkpoints)
            throws IDNotRecognisedException,
                DuplicatedResultException,
                InvalidCheckpointsException,
            InvalidStageStateException {
        if(Stage.getStageState(stageId).equals(StageState.
            BUILDING)) {
            throw new InvalidStageStateException("stage is not
                waiting for results");
        } else if(Stage.getSegments(stageId).length+2 !=
            checkpoints.length) {
            throw new InvalidCheckpointsException("checkpoint
                count mismatch");
        }
        try {
            Result.getResult(stageId, riderId);
            throw new DuplicatedResultException();
        } catch(IDNotRecognisedException ex) {
            Stage.getStage(stageId);
            riderManager.getRider(riderId);
            // above should throw exceptions if IDs are not in
                system
            new Result(stageId, riderId, checkpoints);
        }
    }

    @Override
    public LocalTime[] getRiderResultsInStage(int stageId, int
        riderId) throws IDNotRecognisedException {
        Stage.getStage(stageId);
        riderManager.getRider(riderId);
        // above should throw exceptions if IDs are not in
            system
        Result result = Result.getResult(stageId, riderId);
        LocalTime[] checkpointTimes = result.getCheckpoints();
        LocalTime[] out = new LocalTime[checkpointTimes.length
            +1];
        for(int i=0; i<checkpointTimes.length; i++) {
            out[i] = checkpointTimes[i];
        }
        out[-1] = result.getTotalElasped();
        return out;
    }

    @Override
    public LocalTime getRiderAdjustedElapsedTimeInStage(int
        stageId, int riderId) throws IDNotRecognisedException {
        Stage.getStage(stageId);
        riderManager.getRider(riderId);
        // above should throw exceptions if IDs are not in
            system
        LocalTime[] adjustedTimes = Result.getResult(stageId,
            riderId).adjustedCheckpoints();
        LocalTime elapsedTime = adjustedTimes[0];
        for(int i=1; i<adjustedTimes.length; i++) {
            LocalTime t = adjustedTimes[i];
            //elapsedTime.plusHours(t.getHour()).plusMinutes(t.
                getMinute()).plusSeconds(t.getSecond()).
                plusNanos(t.getNano());
            elapsedTime = elapsedTime.plusHours(t.getHour());
            elapsedTime = elapsedTime.plusMinutes(t.getMinute()
                );
            elapsedTime = elapsedTime.plusSeconds(t.getSecond()
                );
        }
        return elapsedTime;
    }

    @Override
    public void deleteRiderResultsInStage(int stageId, int
        riderId) throws IDNotRecognisedException {
        Stage.getStage(stageId);
        riderManager.getRider(riderId);
        // above should throw exceptions if IDs are not in
            system
        Result.removeResult(stageId, riderId);
    }

    @Override
    public int[] getRidersRankInStage(int stageId) throws
        IDNotRecognisedException {
        Result[] results = Result.getResultsInStage(stageId);
        int[] riderRanks = new int[results.length];
        Arrays.fill(riderRanks, -1);
        for(Result r : results) {
            for(int i=0; i<riderRanks.length; i++) {
                if(riderRanks[i] == -1) {
                    riderRanks[i] = r.getRiderId();
                    break;
                } else if(r.getTotalElasped().isBefore(Result.
                    getResult(stageId, riderRanks[i]).
                    getTotalElasped())) {
                    int temp;
                    int prev = r.getRiderId();
                    for(int j=i; j<riderRanks.length; j++) {
                        temp = riderRanks[j];
                        riderRanks[j] = prev;
                        prev = temp;
                        if(prev == -1) {
                            break;
                        }
                    }
                    break;
                }
            }
        }
        return riderRanks;
    }

    @Override
    public LocalTime[] getRankedAdjustedElapsedTimesInStage(int
         stageId) throws IDNotRecognisedException {
        int[] riderRanks = this.getRidersRankInStage(stageId);
        LocalTime[] out = new LocalTime[riderRanks.length];
        for(int i=0; i<out.length; i++) {
```

```java
            Result r = Result.getResult(stageId, riderRanks[i]);
            LocalTime[] checkpoints = r.getCheckpoints();
            LocalTime[] adjustedTimes = r.adjustedCheckpoints();
            out[i] = adjustedTimes[0];
            LocalTime adjustedSplit;
            for(int j=0; j<adjustedTimes.length; j++) {
                adjustedSplit = Result.getElapsed(adjustedTimes[j], checkpoints[j]);
                System.out.println(adjustedSplit.toString());
                out[i] = out[i].plusHours(adjustedSplit.getHour());
                out[i] = out[i].plusMinutes(adjustedSplit.getMinute());
                out[i] = out[i].plusSeconds(adjustedSplit.getSecond());
            }
        }
        return out;
    }

    @Override
    public int[] getRidersPointsInStage(int stageId) throws IDNotRecognisedException {
        StageType type = Stage.getStageType(stageId);
        int[] points = new int[Result.getResultsInStage(stageId).length];
        int[] distribution = new int[15];
        // distributions from https://en.wikipedia.org/wiki/Points_classification_in_the_Tour_de_France
        switch(type) {
            case FLAT:
                distribution = new int[]{50,30,20,18,16,14,12,10,8,7,6,5,4,3,2};
                break;
            case MEDIUM_MOUNTAIN:
                distribution = new int[]{30,25,22,19,17,15,13,11,9,7,6,5,4,3,2};
                break;
            case HIGH_MOUNTAIN:
                distribution = new int[]{20,17,15,13,11,10,9,8,7,6,5,4,3,2,1};
                break;
            case TT:
                distribution = new int[]{20,17,15,13,11,10,9,8,7,6,5,4,3,2,1};
                break;
        }
        for(int i=0; i<Math.min(points.length, distribution.
            length); i++) {
            points[i] = distribution[i];
        }
        return points;
    }

    @Override
    public int[] getRidersMountainPointsInStage(int stageId)
        throws IDNotRecognisedException {
        Result[] results = Result.getResultsInStage(stageId);
        // All results refering to the stage with id *stageId*
        int[] riders = getRidersRankInStage(stageId);
        // An int array of rider ids, from first to last
        int[] segments = Stage.getSegments(stageId);
        // An int array of the segment ids in the stage
        int[] points = new int[riders.length];
        // The int in position i is the number of points to be
            awarded to the rider with id riders[i]
        for(int s=0; s<segments.length; s++) {
            SegmentType type = Segment.getSegmentType(segments[
                s]);
            int[] distribution = new int[1];
            // The points to be awarded in order for the
                segment
            switch(type) {
                case C4:
                    distribution = new int[]{1};
                    break;
                case C3:
                    distribution = new int[]{2,1};
                    break;
                case C2:
                    distribution = new int[]{5,3,2,1};
                    break;
                case C1:
                    distribution = new int[]{10,8,6,4,2,1};
                    break;
                case HC:
                    distribution = new int
                        []{20,15,12,10,8,6,4,2};
                    break;
                case SPRINT:
            }
            // get ranks for segment
            int[] riderRanks = new int[results.length];
            Arrays.fill(riderRanks, -1);
            for(Result r : results) {
                for(int i=0; i<riderRanks.length; i++) {
                    if(riderRanks[i] == -1) {
                        riderRanks[i] = r.getRiderId();
```

```java
                        break;
                    } else if(r.getCheckpoints()[s].isBefore(Result.getResult(stageId, riderRanks[i]).getCheckpoints()[s])) {
                        int temp;
                        int prev = r.getRiderId();
                        for(int j=i; j<riderRanks.length; j++) {
                            temp = riderRanks[j];
                            riderRanks[j] = prev;
                            prev = temp;
                            if(prev == -1) {
                                break;
                            }
                        }
                        break;
                    }
                }
            }
            ArrayList<Integer> ridersArray = new ArrayList<Integer>();
            for(int r : riders) { ridersArray.add(r); }
            for(int i=0; i<Math.min(points.length, distribution.length); i++) {
                int overallPos = ridersArray.indexOf(riderRanks[i]);
                if(overallPos<points.length && overallPos!=-1) {
                    points[overallPos] += distribution[i];
                }
            }
        }
        return points;
    }

    @Override
    public void eraseCyclingPortal() {

        Team.teamNames.clear();
        Team.teamTopId = 0;
        Rider.ridersTopId = 0;

        RiderManager.allRiders.clear();
        RiderManager.allTeams.clear();


        Race.allRaces.clear();
        Race.removedIds.clear();
        Race.loadId();

        Segment.allSegments.clear();
        Segment.removedIds.clear();
        Segment.loadId();

        Stage.allStages.clear();
        Stage.removedIds.clear();
        Stage.loadId();

        Result.allResults.clear();


    }

    @Override
    public void saveCyclingPortal(String filename) throws
        IOException {
        try {
            FileOutputStream fos = new FileOutputStream(
                filename);
            ObjectOutputStream oos = new ObjectOutputStream(fos
                );
            ArrayList<ArrayList> allObj = new ArrayList<>();
            allObj.add(RiderManager.allTeams);
            allObj.add(RiderManager.allRiders);
            allObj.add(Stage.allStages);
            allObj.add(Stage.removedIds);
            allObj.add(Race.allRaces);
            allObj.add(Race.removedIds);
            allObj.add(Result.allResults);
            allObj.add(Segment.allSegments);
            allObj.add(Segment.removedIds);

            oos.writeObject(allObj);

            oos.flush();
            oos.close();

        } catch (IOException ex) {
            ex.printStackTrace();
        }

    }

    @Override
    public void loadCyclingPortal(String filename) throws
        IOException, ClassNotFoundException {
        try {

            FileInputStream fis = new FileInputStream(filename)
                ;
```

```java
ObjectInputStream ois = new ObjectInputStream(fis);
ArrayList<Object> allObjects = new ArrayList<>();
ArrayList<Team> allTeams = new ArrayList<>();
ArrayList<Rider> allRiders = new ArrayList<>();
ArrayList<Result> allResults = new ArrayList<Result>();
ArrayList<Race> allRaces = new ArrayList<Race>();
ArrayList<Stage> allStages = new ArrayList<Stage>();
ArrayList<Segment> allSegments = new ArrayList<Segment>();
ArrayList<Integer> removedIds = new ArrayList<>();

Class<?> classFlag = null;

allObjects = (ArrayList) ois.readObject();
for (Object tempObj : allObjects){
    ArrayList Objects = (ArrayList) tempObj;
for (Object obj : Objects){
    if (classFlag != null){
        if (obj.getClass() != classFlag && obj.getClass() != Integer.class){
            if (classFlag == Race.class){
                Race.removedIds = removedIds;
            }
            if (classFlag == Segment.class){
                Segment.removedIds = removedIds;
            }
            if (classFlag == Stage.class){
                Stage.removedIds = removedIds;
            }
            classFlag = null;
            removedIds.clear();


        }
        else{
            Integer removedId = (Integer) obj;
            removedIds.add(removedId);

        }
    }
    String objClass = obj.getClass().getName();
    System.out.println(objClass);
    if (obj.getClass() == Rider.class){
        Rider newRider = (Rider) obj;
        allRiders.add(newRider);
        System.out.println("NEW RIDER");
    }
    if (obj.getClass() == Team.class){
```

```
                    Team newTeam = (Team) obj;
                    allTeams.add(newTeam);
                    System.out.println("NEW TEAM");
                }
                if (obj.getClass() == Result.class){
                    Result newResult = (Result) obj;
                    allResults.add(newResult);
                    System.out.println("NEW RESULT");
                }
                if (obj.getClass() == Stage.class){
                    Stage newStage = (Stage) obj;
                    allStages.add(newStage);
                    System.out.println("NEW STAGE");
                    classFlag = Stage.class;
                }
                if (obj.getClass() == Race.class){
                    Race newRace = (Race) obj;
                    allRaces.add(newRace);
                    System.out.println("NEW Race");
                    classFlag = Race.class;
                }
                if (obj.getClass() == Segment.class){
                    Segment newSeg = (Segment) obj;
                    allSegments.add(newSeg);
                    System.out.println("NEW SEGMENT");
                    classFlag = Segment.class;
                }


                System.out.println(obj.getClass());
            }
        }
        if (classFlag == Race.class){
            Race.removedIds = removedIds;
        }
        if (classFlag == Segment.class){
            Segment.removedIds = removedIds;
        }
        if (classFlag == Stage.class){
            Stage.removedIds = removedIds;
        }

            this.riderManager.setAllTeams(allTeams);
            this.riderManager.setAllRiders(allRiders);
            Race.allRaces = allRaces;
            Race.loadId();
            Stage.allStages = allStages;
            Stage.loadId();
            Segment.allSegments = allSegments;
            Segment.loadId();
            Result.allResults = allResults;
            ois.close();

        }
        catch (Exception ex) {
            ex.printStackTrace();
        }

    }

    @Override
    public void removeRaceByName(String name) throws
        NameNotRecognisedException {
        boolean found = false;
        for (int raceId : Race.getAllRaceIds()){
            try {
                if (name == Race.getRaceName(raceId)){
                    Race.removeRace(raceId);
                }
            }
            catch(Exception c){
                assert(false); // Exception will not throw by
                    for each condition
                // This try catch is easier than moving
                    exceptions to CyclingPortal level
            }

        }
        if (!found){ throw new NotRecognisedException("Name
             not in System.");}

    }

    @Override
    public LocalTime[] getGeneralClassificationTimesInRace(int
        raceId) throws IDNotRecognisedException {
        Race currentRace = Race.getRace(raceId);
        int[] stageIds = currentRace.getStages();
        int[] riderIds = this.riderManager.getRiderIds();
        HashMap<Integer,Long> riderElaspedTime = new HashMap<
            Integer,Long>(); //Rider Id -> totalTime (long)
        for (int riderId : riderIds){
            riderElaspedTime.put(riderId,0L);
        }
        for (int stageId : stageIds){
            Result[] temp = Result.getResultsInStage(stageId);
            for(Result result: temp){
                int riderId = result.getRiderId();
                LocalTime getTotalElasped = result.
                    getTotalElasped();
                long timeTaken = getTotalElasped.toNanoOfDay();
                Long newTime = (Long)riderElaspedTime.get(
                    riderId)+timeTaken;
                riderElaspedTime.put(riderId,newTime);
            }

        }
        long[][] riderTimePos = new long[riderIds.length][2];
        int count = 0;
        for (int riderId : riderIds){
            Long finalRiderTime = riderElaspedTime.get(riderId)
                ;// ## -> [[time,riderId],....] sort by time!
            riderTimePos[count][0] = riderId;
            riderTimePos[count][1] = finalRiderTime;
            count++;
        }
        Arrays.sort(riderTimePos, Comparator.comparingDouble(o
            -> o[1]));
        LocalTime[] finalTimes = new LocalTime[riderIds.length
            ];
        count = 0;
        for (long[] items : riderTimePos){
            finalTimes[count]= LocalTime.ofNanoOfDay(items[1]);
            count++;
        }



        return finalTimes;
    }

    @Override
    public int[] getRidersPointsInRace(int raceId) throws
        IDNotRecognisedException {
        ArrayList<Integer> order = new ArrayList<Integer>();
        for(int riderId : getRidersGeneralClassificationRank(
            raceId)) {
            order.add(riderId);
        }
        int[] out = new int[order.size()];
        int[] stageRank, stagePoints;
        for(int stageId : Race.getStages(raceId)) {
            stageRank = getRidersRankInStage(stageId);
            stagePoints = getRidersPointsInStage(stageId);
            for(int i=0; i<stageRank.length; i++) {
                out[order.indexOf(stageRank[i])] += stagePoints
                    [i];
            }
        }
        return out;
```

```java
    }

    @Override
    public int[] getRidersMountainPointsInRace(int raceId)
        throws IDNotRecognisedException {
        ArrayList<Integer> order = new ArrayList<Integer>();
        for(int riderId : getRidersGeneralClassificationRank(
            raceId)) {
            order.add(riderId);
        }
        int[] out = new int[order.size()];
        int[] stageRank, stagePoints;
        for(int stageId : Race.getStages(raceId)) {
            stageRank = getRidersRankInStage(stageId);
            stagePoints = getRidersMountainPointsInStage(
                stageId);
            for(int i=0; i<stageRank.length; i++) {
                out[order.indexOf(stageRank[i])] += stagePoints
                    [i];
            }
        }
        return out;
    }

    @Override
    public int[] getRidersGeneralClassificationRank(int raceId)
         throws IDNotRecognisedException {
        Race currentRace = Race.getRace(raceId);
        int[] stageIds = currentRace.getStages();
        int[] riderIds = this.riderManager.getRiderIds();
        HashMap<Integer,Long> riderElaspedTime = new HashMap<
            Integer,Long>(); //Rider Id -> totalTime (long)
        for (int riderId : riderIds){
            riderElaspedTime.put(riderId,0L);
        }
        for (int stageId : stageIds){
            Result[] temp = Result.getResultsInStage(stageId);
            for(Result result: temp){
                int riderId = result.getRiderId();
                LocalTime getTotalElasped = result.
                    getTotalElasped();
                long timeTaken = getTotalElasped.toNanoOfDay();
                Long newTime = (Long)riderElaspedTime.get(
                    riderId)+timeTaken;
                riderElaspedTime.put(riderId,newTime);
            }

        }
        long[][] riderTimePos = new long[riderIds.length][2];
        int count = 0;
        for (int riderId : riderIds){
            Long finalRiderTime = riderElaspedTime.get(riderId)
                ;// ## -> [[time,riderId],....] sort by time!
            riderTimePos[count][0] = riderId;
            riderTimePos[count][1] = finalRiderTime;
            count++;
        }
        Arrays.sort(riderTimePos, Comparator.comparingDouble(o
            -> o[1]));
        int[] finalPos = new int[riderIds.length];
        count = 0;
        for (long[] items : riderTimePos){
            finalPos[count]= (int)items[0];
            count++;
        }

        return finalPos;
    }

    @Override
    public int[] getRidersPointClassificationRank(int raceId)
        throws IDNotRecognisedException {
        int[] order = getRidersGeneralClassificationRank(raceId
            );
        int[] points = getRidersPointsInRace(raceId);
        int[] out = new int[order.length];
        for(int i=0; i<out.length; i++) {
            int maxPoints = -1;
            int nextId = -1;
            for(int j=0; j<order.length; j++) {
                int id = order[j];
                if(id<0) { continue; }
                if(points[id] > maxPoints) {
                    maxPoints = points[j];
                    nextId = id;
                }
            }
            if(maxPoints < 0) {
                break;
            } else {
                out[i] = nextId;
                order[nextId] = -1;
            }
        }
        return out;
    }

    @Override
    public int[] getRidersMountainPointClassificationRank(int
        raceId) throws IDNotRecognisedException {
```

```java
        // effectively a clone of the method above
        int[] order = getRidersGeneralClassificationRank(raceId
            );
        int[] points = getRidersMountainPointsInRace(raceId);
        int[] out = new int[order.length];
        for(int i=0; i<out.length; i++) {
            int maxPoints = -1;
            int nextId = -1;
            for(int j=0; j<order.length; j++) {
                int id = order[j];
                if(id<0) { continue; }
                if(points[id] > maxPoints) {
                    maxPoints = points[j];
                    nextId = id;
                }
            }
            if(maxPoints < 0) {
                break;
            } else {
                out[i] = nextId;
                order[nextId] = -1;
            }
        }
        return out;
    }
}
```

2 Race.java

```java
package cycling;

import java.util.ArrayList;
import java.io.Serializable;
import java.time.LocalDateTime;

/**
 * Race encapsulates tour races, each of which has a number of
     associated
 * Stages.
 *
 * @author Thomas Newbold
 * @version 2.0
 *
 */
public class Race implements Serializable {
    // Static class attributes
    private static int idMax = 0;
    public static ArrayList<Integer> removedIds = new ArrayList
        <Integer>();
    public static ArrayList<Race> allRaces = new ArrayList<Race
        >();

    /**
     * Loads the value of idMax.
     */
    public static void loadId(){
        if(Race.allRaces.size()!=0) {
            Race.idMax = Race.allRaces.get(Race.allRaces.size()
                -1).getRaceId() + 1;
        } else {
            Race.idMax = 0;
        }
    }

    /**
     * @param raceId The ID of the race instance to fetch
     * @return The race instance with the associated ID
     * @throws IDNotRecognisedException If no race exists with
         the requested ID
     */
    public static Race getRace(int raceId) throws
        IDNotRecognisedException {
        boolean removed = Race.removedIds.contains(raceId);
        if(raceId<Race.idMax && raceId >= 0 && !removed) {
            int index = raceId;
            for(int j=0; j<Race.removedIds.size(); j++) {
                if(Race.removedIds.get(j) < raceId) {
                    index--;
                }
            }
            return allRaces.get(index);
        } else if (removed) {
            throw new IDNotRecognisedException("no race
                instance for raceID");
        } else {
            throw new IDNotRecognisedException("raceID out of
                range");
        }
    }

    /**
     * @return An integer array of the race IDs of all races
     */
    public static int[] getAllRaceIds() {
        int length = Race.allRaces.size();
        int[] raceIdsArray = new int[length];
        int i = 0;
        for(Race race : allRaces) {
            raceIdsArray[i] = race.getRaceId();
            i++;
        }
        return raceIdsArray;
    }

    /**
     * @param raceId The ID of the race instance to remove
     * @throws IDNotRecognisedException If no race exists with
         the requested ID
     */
    public static void removeRace(int raceId) throws
        IDNotRecognisedException {
        boolean removed = Race.removedIds.contains(raceId);
        if(raceId<Race.idMax && raceId >= 0 && !removed) {
            Race r = getRace(raceId);
            for(int id : r.getStages()) {
                r.removeStageFromRace(id);
            }
            allRaces.remove(r);
            removedIds.add(raceId);
        } else if (removed) {
            throw new IDNotRecognisedException("no race
                instance for raceID");
        } else {
            throw new IDNotRecognisedException("raceID out of
                range");
        }
    }

    // Instance attributes
    private int raceId;
    private String raceName;
    private String raceDescription;
    private ArrayList<Integer> stageIds;

    /**
     * @param name String to be checked
     * @return true if name is valid for the system
     */
    private static boolean validName(String name) {
        if(name==null || name.equals("")) {
            return false;
        } else if(name.length()>30) {
            return false;
        } else if(name.contains(" ")) {
            return false;
        } else {
            return true;
        }
    }

    /**
     * Race constructor; creates new race and adds to allRaces array.
     *
     * @param name The name of the new race
     * @param description The description for the new race
     * @throws IllegalNameException If name already exists in the system
     * @throws InvalidNameException If name is empty/null, contains whitespace,
     *                              or is longer than 30 characters
     */
    public Race(String name, String description) throws IllegalNameException,
                InvalidNameException {
        for(Race race : allRaces) {
            if(race.getRaceName().equals(name)) {
                throw new IllegalNameException("name already exists");
            }
        }
        if(!validName(name)) {
            throw new InvalidNameException("invalid name");
        }
        if(Race.removedIds.size() > 0) {
            this.raceId = Race.removedIds.get(0);
            Race.removedIds.remove(0);
        } else {
            this.raceId = idMax++;
        }
        this.raceName = name;
        this.raceDescription = description;
        this.stageIds = new ArrayList<Integer>();
        Race.allRaces.add(this);
    }

    /**
     * @return A string representation of the race instance
     */
    public String toString() {
        String id = Integer.toString(this.raceId);
        String name = this.raceName;
        String description = this.raceDescription;
        String list = this.stageIds.toString();
        return String.format("Race[%s]: %s; %s; StageIds=%s;", id, name,
                             description, list);
    }

    /**
     * @param id The ID of the race
     * @return A string representation of the race instance
     * @throws IDNotRecognisedException If no race exists with
         the requested ID
     */
    public static String toString(int id) throws
        IDNotRecognisedException {
        return getRace(id).toString();
    }

    /**
     * @return The integer raceId for the race instance
     */
    public int getRaceId() { return this.raceId; }

    /**
     * @return The string raceName for the race instance
     */
    public String getRaceName() { return this.raceName; }

    /**
     * @param id The ID of the race
     * @return The string raceName for the race with the
         associated id
     * @throws IDNotRecognisedException If no race exists with
         the requested ID
     */
    public static String getRaceName(int id) throws
        IDNotRecognisedException {
        return getRace(id).raceName;
    }

    /**
     * @return The string raceDescription for the race instance
     */
    public String getRaceDescription() { return this.
        raceDescription; }

    /**
     * @param id The ID of the race
     * @return The string raceDescription for the race with the
          associated id
     * @throws IDNotRecognisedException If no race exists with
         the requested ID
     */
    public static String getRaceDescription(int id) throws
                                            IDNotRecognisedException
                                                 {
```

```
        return getRace(id).raceDescription;
    }

    /**
     * @return An integer array of stage IDs for the race
         instance
     */
    public int[] getStages() {
        int length = this.stageIds.size();
        int[] stageIdsArray = new int[length];
        for(int i=0; i<length; i++) {
            stageIdsArray[i] = this.stageIds.get(i);
        }
        return stageIdsArray;
    }

    /**
     * @param id The ID of the race
     * @return An integer array of stage IDs for the race
         instance
     * @throws IDNotRecognisedException If no race exists with
         the requested ID
     */
    public static int[] getStages(int id) throws
        IDNotRecognisedException {
        Race race = getRace(id);
        int length = race.stageIds.size();
        int[] stageIdsArray = new int[length];
        for(int i=0; i<length; i++) {
            stageIdsArray[i] = race.stageIds.get(i);
        }
        return stageIdsArray;
    }

    /**
     * @param name The new name for the race instance
     */
    public void setRaceName(String name) {
        this.raceName = name;
    }

    /**
     * @param id The ID of the race to be updated
     * @param name The new name for the race instance
     * @throws IDNotRecognisedException If no race exists with
         the requested ID
     */
    public static void setRaceName(int id, String name) throws
                                    IDNotRecognisedException {
        getRace(id).setRaceName(name);
    }

    /**
     * @param description The new description for the race
         instance
     */
    public void setRaceDescription(String description) {
        this.raceDescription = description;
    }

    /**
     * @param id The ID of the race to be updated
     * @param description The new description for the race
         instance
     * @throws IDNotRecognisedException If no race exists with
         the requested ID
     */
    public static void setRaceDescription(int id, String
        description) throws
                                         IDNotRecognisedException
                                              {
        getRace(id).setRaceDescription(description);
    }

    /**
     * Creates a new stage and adds the ID to the stageIds
         array.
     *
     * @param name The name of the new stage
     * @param description The description of the new stage
     * @param length The length of the new stage (in km)
     * @param startTime The date and time at which the stage
         will be held
     * @param type The StageType, used to determine the point
         distribution
     * @return The ID of the new stage
     */
    public int addStageToRace(String name, String description,
        double length,
                               LocalDateTime startTime,
                                   StageType type) throws
                               IllegalNameException,
                                   InvalidNameException,
                               InvalidLengthException {
        Stage newStage = new Stage(name, description, length,
            startTime, type);
        this.stageIds.add(newStage.getStageId());
        return newStage.getStageId();
    }

```

```java
    /**
     * Creates a new stage and adds the ID to the stageIds array.
     *
     * @param id The ID of the race to which the stage will be added
     * @param name The name of the new stage
     * @param description The description of the new stage
     * @param length The length of the new stage (in km)
     * @param startTime The date and time at which the stage will be held
     * @param type The StageType, used to determine the point distribution
     * @return The ID of the new stage
     * @throws IDNotRecognisedException If no race exists with the requested ID
     */
    public static int addStageToRace(int id, String name, String description,
                                     double length, LocalDateTime startTime,
                                     StageType type) throws
                                     IDNotRecognisedException,
                                     IllegalNameException, InvalidNameException,
                                     InvalidLengthException  {
        return getRace(id).addStageToRace(name, description, length, startTime, type);
    }

    /**
     * Removes a stageId from the array of stageIds for a race instance,
     * as well as from the static array of all stages in the Stage class.
     *
     * @param stageId The ID of the stage to be removed
     * @throws IDNotRecognisedException If no stage exists with the requested ID
     */
    private void removeStageFromRace(int stageId) throws IDNotRecognisedException {
        if(this.stageIds.contains(stageId)) {
            this.stageIds.remove(stageId);
            Stage.removeStage(stageId);
        } else {
            throw new IDNotRecognisedException("stageID not found in race");
        }
    }

    /**
     * Removes a stageId from the array of stageIds for a race instance,
     * as well as from the static array of all stages in the Stage class.
     *
     * @param id The ID of the race to which the stage will be removed
     * @param stageId The ID of the stage to be removed
     * @throws IDNotRecognisedException If no stage exists with the requested ID
     */
    public static void removeStageFromRace(int id, int stageId) throws
                                                     IDNotRecognisedException {
        getRace(id).removeStageFromRace(stageId);
    }

    /**
     * Removes a stageId from the array of stageIds for a race instance,
     * as well as from the static array of all stages in the Stage class.
     *
     * @param stageId The ID of the stage to be removed
     * @throws IDNotRecognisedException If no stage exists with the requested ID
     */
    public static void removeStage(int stageId) throws IDNotRecognisedException {
        for(Race race : allRaces) {
            if(race.stageIds.contains(stageId)) {
                race.removeStageFromRace(stageId);
                break;
            }
        }
    }
}
```

3 Stage.java

```java
package cycling;

import java.util.ArrayList;
import java.io.Serializable;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

/**
 * Stage encapsulates race stages, each of which has a number of associated
 * Segments.
 *
 * @author Thomas Newbold
 * @version 2.0
 *
 */
public class Stage implements Serializable {
    // Static class attributes
    private static int idMax = 0;
    public static ArrayList<Integer> removedIds = new ArrayList<Integer>();
    public static ArrayList<Stage> allStages = new ArrayList<Stage>();

    /**
     * Loads the value of idMax.
     */
    public static void loadId(){
        if(Stage.allStages.size()!=0) {
            Stage.idMax = Stage.allStages.get(Stage.allStages.size()-1).getStageId() + 1;
        } else {
            Stage.idMax = 0;
        }
    }

    /**
     * @param stageId The ID of the stage instance to fetch
     * @return The stage instance with the associated ID
     * @throws IDNotRecognisedException If no stage exists with the requested ID
     */
    public static Stage getStage(int stageId) throws IDNotRecognisedException {
        boolean removed = Stage.removedIds.contains(stageId);
        if(stageId<Stage.idMax && stageId >= 0 && !removed) {
            int index = stageId;
            for(int j=0; j<Stage.removedIds.size(); j++) {
                if(Stage.removedIds.get(j) < stageId) {
                    index--;
                }
            }
            return allStages.get(index);
        } else if (removed) {
            throw new IDNotRecognisedException("no stage
                instance for stageID");
        } else {
            throw new IDNotRecognisedException("stageId out of
                range");
        }
    }

    /**
     * @return An integer array of the stage IDs of all stage
     */
    public static int[] getAllStageIds() {
        int length = Stage.allStages.size();
        int[] stageIdsArray = new int[length];
        int i = 0;
        for(Stage stage : allStages) {
            stageIdsArray[i] = stage.getStageId();
            i++;
        }
        return stageIdsArray;
    }

    /**
     * @param stageId The ID of the stage instance to remove
     * @throws IDNotRecognisedException If no stage exists with
         the requested ID
     */
    public static void removeStage(int stageId) throws
        IDNotRecognisedException {
        boolean removed = Stage.removedIds.contains(stageId);
        if(stageId<Stage.idMax && stageId >= 0 && !removed) {
            Stage s = getStage(stageId);
            for(int id : s.getSegments()) {
                s.removeSegmentFromStage(id);
            }
            allStages.remove(s);
            removedIds.add(stageId);
        } else if (removed) {
            throw new IDNotRecognisedException("no stage
                instance for stageID");
        } else {
            throw new IDNotRecognisedException("stageId out of
                range");
        }
    }

    // Instance attributes
    private int stageId;
    private StageState stageState;
    private String stageName;
    private String stageDescription;
```

```java
    private double stageLength;
    private LocalDateTime stageStartTime;
    private StageType stageType;
    private ArrayList<Integer> segmentIds;

    /**
     * @param name String to be checked
     * @return true if name is valid for the system
     */
    private static boolean validName(String name) {
        if(name==null || name.equals("")) {
            return false;
        } else if(name.length()>30) {
            return false;
        } else if(name.contains(" ")) {
            return false;
        } else {
            return true;
        }
    }

    /**
     * Stage constructor; creates a new stage and adds to allStages array.
     *
     * @param name The name of the new stage
     * @param description The description of the new stage
     * @param length The total length of the new stage
     * @param startTime The start time for the new stage
     * @param type The type of the new stage
     * @throws IllegalNameException If name already exists in the system
     * @throws InvalidNameException If name is empty/null, contains whitespace,
     *                             or is longer than 30 characters
     * @throws InvalidLengthException If the length is less than 5km
     */
    public Stage(String name, String description, double length,
                 LocalDateTime startTime, StageType type) throws
                 IllegalNameException, InvalidNameException,
                 InvalidLengthException {
        for(Stage stage : allStages) {
            if(stage.getStageName().equals(name)) {
                throw new IllegalNameException("name already exists");
            }
        }
        if(!validName(name)) {
            throw new InvalidNameException("invalid name");
        }
        if(length<5) {
            throw new InvalidLengthException("length less than
                5km");
        }
        if(Stage.removedIds.size() > 0) {
            this.stageId = Stage.removedIds.get(0);
            Stage.removedIds.remove(0);
        } else {
            this.stageId = idMax++;
        }
        this.stageState = StageState.BUILDING;
        this.stageName = name;
        this.stageDescription = description;
        this.stageLength = length;
        this.stageStartTime = startTime;
        this.stageType = type;
        this.segmentIds = new ArrayList<Integer>();
        Stage.allStages.add(this);
    }

    /**
     * @return A string representation of the stage instance
     */
    public String toString() {
        String id = Integer.toString(this.stageId);
        String state;
        switch (this.stageState) {
            case BUILDING:
                state = "In preperation";
                break;
            case WAITING:
                state = "Waiting for results";
                break;
            default:
                state = "null state";
        }
        String name = this.stageName;
        String description = this.stageDescription;
        String length = Double.toString(this.stageLength);
        DateTimeFormatter formatter = DateTimeFormatter.
            ofPattern("HH:hh dd-MM-yyyy");
        String startTime = this.stageStartTime.format(formatter
            );
        String list = this.segmentIds.toString();
        String type;
        switch (this.stageType) {
            case FLAT:
                type = "Flat";
                break;
            case MEDIUM_MOUNTAIN:
                type = "Medium Mountain";
                break;
            case HIGH_MOUNTAIN:
                type = "High Mountain";
                break;
            case TT:
                type = "Time Trial";
                break;
            default:
                type = "null type";
        }
        return String.format("Stage[%s](%s): %s (%s); %s; %skm; %s; SegmentIds=%s;",
                             id, state, name, type, description, length,
                             startTime, list);
    }

    /**
     * @param id The ID of the stage
     * @return A string representation of the stage instance
     * @throws IDNotRecognisedException If no stage exists with the requested ID
     */
    public static String toString(int id) throws IDNotRecognisedException {
        return getStage(id).toString();
    }

    /**
     * @return The integer stageId for the stage instance
     */
    public int getStageId() { return this.stageId; }

    /**
     * @return The state of the stage instance
     */
    public StageState getStageState() { return this.stageState; }

    /**
     * @param id The ID of the stage
     * @return The state of the stage instance
     * @throws IDNotRecognisedException If no stage exists with the requested ID
     */
```

```java
    public static StageState getStageState(int id) throws
                                               IDNotRecognisedException
                                                    {
        return getStage(id).getStageState();
    }
    /**
     * @return The string raceName for the stage instance
     */
    public String getStageName() { return this.stageName; }

    /**
     * @param id The ID of the stage
     * @return The string stageName for the stage with the
         associated id
     * @throws IDNotRecognisedException If no stage exists with
          the requested ID
     */
    public static String getStageName(int id) throws
        IDNotRecognisedException {
        return getStage(id).stageName;
    }

    /**
     * @return The string stageDescription for the stage
         instance
     */
    public String getStageDescription() { return this.
        stageDescription; }

    /**
     * @param id The ID of the stage
     * @return The string stageDescription for the stage with
         the associated id
     * @throws IDNotRecognisedException If no stage exists with
          the requested ID
     */
    public static String getStageDescription(int id) throws
                                                 IDNotRecognisedException
                                                      {
        return getStage(id).stageDescription;
    }

    /**
     * @return The length of the stage instance
     */
    public double getStageLength() { return this.stageLength; }

    /**
     * @param id The ID of the stage
     * @return The length of the stage instance
     * @throws IDNotRecognisedException If no stage exists with
         the requested ID
     */
    public static double getStageLength(int id) throws
        IDNotRecognisedException {
        return getStage(id).stageLength;
    }

    /**
     * @return The start time for the stage instance
     */
    public LocalDateTime getStageStartTime() { return this.
        stageStartTime; }

    /**
     * @param id The ID of the stage
     * @return The start time for the stage instance
     * @throws IDNotRecognisedException If no stage exists with
         the requested ID
     */
    public static LocalDateTime getStageStartTime(int id)
        throws
                                                   IDNotRecognisedException
                                                        {
        return getStage(id).stageStartTime;
    }

    /**
     * @return The type of the stage instance
     */
    public StageType getStageType() { return this.stageType; }

    /**
     * @param id The ID of the stage
     * @return The type of the stage instance
     * @throws IDNotRecognisedException If no stage exists with
         the requested ID
     */
    public static StageType getStageType(int id) throws
        IDNotRecognisedException {
        return getStage(id).getStageType();
    }

    /**
     * @return An integer array of segment IDs for the stage
        instance
     */
    public int[] getSegments() {
        int length = this.segmentIds.size();
        int[] segmentIdsArray = new int[length];
        for(int i=0; i<length; i++) {
            segmentIdsArray[i] = this.segmentIds.get(i);
        }
        return segmentIdsArray;
    }

    /**
     * @param id The ID of the stage
     * @return An integer array of segment IDs for the stage
         instance
     * @throws IDNotRecognisedException If no stage exists with
          the requested ID
     */
    public static int[] getSegments(int id) throws
        IDNotRecognisedException {
        Stage stage = getStage(id);
        int length = stage.segmentIds.size();
        int[] segmentIdsArray = new int[length];
        for(int i=0; i<length; i++) {
            segmentIdsArray[i] = stage.segmentIds.get(i);
        }
        return segmentIdsArray;
    }

    /**
     * Updates the stage state from building to waiting for
         results.
     *
     * @throws InvalidStageStateException If the stage is
         already waiting for results
     */
    public void updateStageState() throws
        InvalidStageStateException {
        if(this.stageState.equals(StageState.WAITING)) {
            throw new InvalidStageStateException("stage is
                already waiting for results");
        } else if(this.stageState.equals(StageState.BUILDING))
            {
            this.stageState = StageState.WAITING;
        }
    }

    /**
     * Updates the stage state from building to waiting for
         results.
     *
     * @param id The ID of the stage to be updated
     * @throws IDNotRecognisedException If no stage exists with
          the requested ID
     * @throws InvalidStageStateException If the stage is
```

```
         already waiting for results
     */
    public static void updateStageState(int id) throws
        IDNotRecognisedException,
                                        InvalidStageStateException
                                             {
        getStage(id).updateStageState();
    }

    /**
     * @param name The new name for the stage instance
     */
    public void setStageName(String name) {
        this.stageName = name;
    }

    /**
     * @param id The ID of the stage to be updated
     * @param name The new name for the stage instance
     * @throws IDNotRecognisedException If no stage exists with
          the requested ID
     */
    public static void setStageName(int id, String name) throws
                                    IDNotRecognisedException {
        getStage(id).setStageName(name);
    }

    /**
     * @param description The new description for the stage
         instance
     */
    public void setStageDescription(String description) {
        this.stageDescription = description;
    }

    /**
     * @param id The ID of the stage to be updated
     * @param description The new description for the stage
         instance
     * @throws IDNotRecognisedException If no stage exists with
          the requested ID
     */
    public static void setStageDescription(int id, String
        description) throws
                                          IDNotRecognisedException
                                               {
        getStage(id).setStageDescription(description);
    }

    /**
```

```java
     * @param length The new length for the stage instance
     */
    public void setStageLength(double length) {
        this.stageLength = length;
    }

    /**
     * @param id The ID of the stage to be updated
     * @param length The new length for the stage instance
     * @throws IDNotRecognisedException If no stage exists with the requested ID
     */
    public static void setStageLength(int id, double length) throws
                                            IDNotRecognisedException {
        getStage(id).stageLength = length;
    }

    /**
     * @param startTime The new start time for the stage instance
     */
    public void setStageStartTime(LocalDateTime startTime) {
        this.stageStartTime = startTime;
    }

    /**
     * @param id The ID of the stage to be updated
     * @param startTime The new start time for the stage instance
     * @throws IDNotRecognisedException If no stage exists with the requested ID
     */
    public static void setStageStartTime(int id, LocalDateTime startTime)
                                            throws IDNotRecognisedException {
        getStage(id).stageStartTime = startTime;
    }

    /**
     * Creates a new stage and adds the ID to the stageIds array.
     *
     * @param location The location of the new segment
     * @param type The type of the new segment
     * @param averageGradient The average gradient of the new segment
     * @param length The length (in km) of the new segment
     * @throws InvalidLocationException If the segment finishes
          outside of the
     *                                  bounds of the stage
     * @throws InvalidStageStateException If the segment state
         is waiting for
     *                                    results
     * @throws InvalidStageTypeException If the stage type is a
          time-trial
     *                                   (cannot contain
         segments)
     */
    public int addSegmentToStage(double location, SegmentType
        type,
                                 double averageGradient, double
                                      length) throws
                                 InvalidLocationException,
                                 InvalidStageStateException,
                                 InvalidStageTypeException {
        if(location > this.getStageLength()) {
            throw new InvalidLocationException("segment
                finishes outside of stage bounds");
        }
        if(this.getStageState().equals(StageState.WAITING)) {
            throw new InvalidStageStateException("stage is
                waiting for results");
        }
        if(this.getStageType().equals(StageType.TT)) {
            throw new InvalidStageTypeException("time trial
                stages cannot contain segments");
        }
        Segment newSegment = new Segment(location, type,
            averageGradient, length);
        this.segmentIds.add(newSegment.getSegmentId());
        return newSegment.getSegmentId();
    }

    /**
     * Creates a new stage and adds the ID to the stageIds
         array.
     *
     * @param id The ID of the stage to which the segment will
         be added
     * @param location The location of the new segment
     * @param type The type of the new segment
     * @param averageGradient The average gradient of the new
         segment
     * @param length The length (in km) of the new segment
     * @throws IDNotRecognisedException If no stage exists with
          the requested ID
```

```java
     * @throws InvalidLocationException If the segment finishes outside of the
     *                                   bounds of the stage
     * @throws InvalidStageStateException If the segment state is waiting for
     *                                    results
     * @throws InvalidStageTypeException If the stage type is a time-trial
     *                                   (cannot contain segments)
     */
    public static int addSegmentToStage(int id, double location, SegmentType type,
                                        double averageGradient, double length) throws
                                        IDNotRecognisedException,
                                        InvalidLocationException,
                                        InvalidStageStateException,
                                        InvalidStageTypeException {
        return getStage(id).addSegmentToStage(location, type, averageGradient, length);
    }

    /**
     * Removes a segmentId from the array of segmentIds for a stage instance,
     * as well as from the static array of all segments in the Segment class.
     *
     * @param segmentId The ID of the segment to be removed
     * @throws IDNotRecognisedException If no segment exists with the requested
     *                                  ID
     */
    private void removeSegmentFromStage(int segmentId) throws
                                        IDNotRecognisedException {
        if(this.segmentIds.contains(segmentId)) {
            this.segmentIds.remove(segmentId);
            Segment.removeSegment(segmentId);
        } else {
            throw new IDNotRecognisedException("segmentID not found in race");
        }
    }

    /**
     * Removes a segmentId from the array of segmentIds for a
         stage instance,
     * as well as from the static array of all segments in the
         Segment class.
     *
     * @param id The ID of the stage to which the segment will
         be removed
     * @param segmentId The ID of the segment to be removed
     * @throws IDNotRecognisedException If no segment exists
         with the requested
     *                                  ID
     */
    public static void removeSegmentFromStage(int id, int
        segmentId) throws
                                                IDNotRecognisedException
                                                     {
        getStage(id).removeSegmentFromStage(segmentId);
    }

    /**
     * Removes a segmentId from the array of segmentIds for a
         stage instance,
     * as well as from the static array of all segments in the
         Segment class.
     *
     * @param segmentId The ID of the segment to be removed
     * @throws IDNotRecognisedException If no segment exists
         with the requested
     *                                  ID
     */
    public static void removeSegment(int segmentId) throws
        IDNotRecognisedException {
        for(Stage stage : allStages) {
            if(stage.segmentIds.contains(segmentId)) {
                stage.removeSegmentFromStage(segmentId);
                break;
            }
        }
    }
}
```

4 StageState.java

```
package cycling;

/**
 * This enum is used to represent the state of a stage.
 *
 * @author Thomas Newbold
 * @version 1.0
 *
 */
public enum StageState {

    /**
     * Used for stages still in preperation - i.e. segments are
        still being
     * added.
     */
    BUILDING,

    /**
     * Used for stages waiting for results
     */
    WAITING;
}
```

5 Segment.java

```
package cycling;

import java.io.Serializable;
import java.util.ArrayList;

/**
 * Segment encapsulates race segments
 *
 * @author Thomas Newbold
 * @version 2.0
 *
 */
public class Segment implements Serializable {
    // Static class attributes
    private static int idMax = 0;
    public static ArrayList<Integer> removedIds = new ArrayList
        <Integer>();
    public static ArrayList<Segment> allSegments = new
        ArrayList<Segment>();

    /**
     * Loads the value of idMax.
     */
    public static void loadId(){
        if(Segment.allSegments.size()!=0) {
            Segment.idMax = Segment.allSegments.get(-1).
                getSegmentId() + 1;
        } else {
            Segment.idMax = 0;
        }
    }

    /**
     * @param segmentId The ID of the segment instance to fetch
     * @return The segment instance with the associated ID
     * @throws IDNotRecognisedException If no segment exists
         with the requested
     *                                  ID
     */
    public static Segment getSegment(int segmentId) throws
                                    IDNotRecognisedException {
        boolean removed = Segment.removedIds.contains(segmentId
            );
        if(segmentId<Segment.idMax && segmentId >= 0 && !
            removed) {
            int index = segmentId;
            for(int j=0; j<Segment.removedIds.size(); j++) {
                if(Segment.removedIds.get(j) < segmentId) {
                    index--;
                }
            }
            return allSegments.get(index);
        } else if (removed) {
            throw new IDNotRecognisedException("no segment
                instance for "+
                                                "segmentId");
        } else {
            throw new IDNotRecognisedException("segmentId out
                of range");
        }
    }

    /**
     * @return An integer array of the segment IDs of all
         segment
     */
    public static int[] getAllSegmentIds() {
        int length = Segment.allSegments.size();
        int[] segmentIdsArray = new int[length];
        int i = 0;
        for(Segment segment : allSegments) {
            segmentIdsArray[i] = segment.getSegmentId();
            i++;
        }
        return segmentIdsArray;
    }

```

```java
    /**
     * @param segmentId The ID of the segment instance to remove
     * @throws IDNotRecognisedException If no segment exists with the requested
     *                                  ID
     */
    public static void removeSegment(int segmentId) throws
                                     IDNotRecognisedException {
        boolean removed = Segment.removedIds.contains(segmentId);
        if(segmentId<Segment.idMax && segmentId >= 0 && !removed) {
            Segment s = getSegment(segmentId);
            allSegments.remove(s);
            removedIds.add(segmentId);
        } else if (removed) {
            throw new IDNotRecognisedException("no segment instance for "+
                                                                    "segmentId");
        } else {
            throw new IDNotRecognisedException("segmentId out of range");
        }
    }

    // Instance attributes
    private int segmentId;
    private double segmentLocation;
    private SegmentType segmentType;
    private double segmentAverageGradient;
    private double segmentLength;

    /**
     * Segment constructor; creates a new segment and adds to allSegment array.
     *
     * @param location The location of the finish of the new segment in the stage
     * @param type The type of the new segment
     * @param averageGradient The average gradient of the new segment
     * @param length The length of the new segment
     */
    public Segment(double location, SegmentType type, double averageGradient,
                   double length) {
        if(Segment.removedIds.size() > 0) {
            this.segmentId = Segment.removedIds.get(0);
            Segment.removedIds.remove(0);
        } else {
            this.segmentId = idMax++;
        }
        this.segmentLocation = location;
        this.segmentType = type;
        this.segmentAverageGradient = averageGradient;
        this.segmentLength = length;
        Segment.allSegments.add(this);
    }

    /**
     * @return A string representation of the segment instance
     */
    public String toString() {
        String id = Integer.toString(this.segmentId);
        String location = Double.toString(this.segmentLocation)
            ;
        String type;
        switch (this.segmentType) {
            case SPRINT:
                type = "Sprint";
                break;
            case C4:
                type = "Category 4 Climb";
                break;
            case C3:
                type = "Category 3 Climb";
                break;
            case C2:
                type = "Category 2 Climb";
                break;
            case C1:
                type = "Category 1 Climb";
                break;
            case HC:
                type = "Hors Categorie";
                break;
            default:
                type = "null category";
        }
        String averageGrad = Double.toString(this.
            segmentAverageGradient);
        String length = Double.toString(this.segmentLength);
        return String.format("Segment[%s]: %s; %skm; Location=%
            s; Gradient=%s;",
                             id, type, length, location,
                                 averageGrad);
    }

    /**
```

```
     * @param id The ID of the segment
     * @return A string representation of the segment instance
     * @throws IDNotRecognisedException If no segment exists
         with the requested
     *                                  ID
     */
    public static String toString(int id) throws
        IDNotRecognisedException {
        return getSegment(id).toString();
    }

    /**
     * @return The integer segmentId for the segment instance
     */
    public int getSegmentId() { return this.segmentId; }

    /**
     * @return The integer representing the location of the
         segment instance
     */
    public double getSegmentLocation() { return this.
        segmentLocation; }

    /**
     * @param id The ID of the segment
     * @return The integer representing the location of the
         segment instance
     * @throws IDNotRecognisedException If no segment exists
         with the requested
     *                                  ID
     */
    public static double getSegmentLocation(int id) throws
                                            IDNotRecognisedException
                                                 {
        return getSegment(id).segmentLocation;
    }

    /**
     * @return The type of the segment instance
     */
    public SegmentType getSegmentType() { return this.
        segmentType; }

    /**
     * @param id The ID of the segment
     * @return The type of the segment instance
     * @throws IDNotRecognisedException If no segment exists
         with the requested
     *                                  ID
     */
    public static SegmentType getSegmentType(int id) throws
                                                IDNotRecognisedException
                                                     {
        return getSegment(id).segmentType;
    }

    /**
     * @return The average gradient of the segment instance
     */
    public double getSegmentAverageGradient() {
        return this.segmentAverageGradient;
    }

    /**
     * @param id The ID of the segment
     * @return The average gradient of the segment instance
     * @throws IDNotRecognisedException If no segment exists
         with the requested
     *                                   ID
     */
    public static double getSegmentAverageGradient(int id)
        throws
                                                       IDNotRecognisedException
                                                            {
        return getSegment(id).segmentAverageGradient;
    }

    /**
     * @return The length of the segment instance
     */
    public double getSegmentLength() { return this.
        segmentLength; }

    /**
     * @param id The ID of the segment
     * @return The length of the segment instance
     * @throws IDNotRecognisedException If no segment exists
         with the requested
     *                                   ID
     */
    public static double getSegmentLength(int id) throws
        IDNotRecognisedException {
        return getSegment(id).segmentLength;
    }

    /**
     * @param location The new location for the segment
         instance
     */
    public void setSegmentLocation(double location) {
        this.segmentLocation = location;
    }

    /**
     * @param id The ID of the segment to be updated
     * @param location The new location for the segment
         instance
     * @throws IDNotRecognisedException If no segment exists
         with the requested
     *                                  ID
     */
    public static void setSegmentLocation(int id, double
        location) throws
                                          IDNotRecognisedException
                                              {
        getSegment(id).setSegmentLocation(location);
    }

    /**
     * @param type The new type for the segment instance
     */
    public void setSegmentType(SegmentType type) {
        this.segmentType = type;
    }

    /**
     * @param id The ID of the segment to be updated
     * @param type The new type for the segment instance
     * @throws IDNotRecognisedException If no segment exists
         with the requested
     *                                  ID
     */
    public static void setSegmentType(int id, SegmentType type)
         throws
                                      IDNotRecognisedException
                                         {
        getSegment(id).setSegmentType(type);
    }

    /**
     * @param averageGradient The new average gradient for the
         segment instance
     */
    public void setSegmentAverageGradient(double
        averageGradient) {
        this.segmentAverageGradient = averageGradient;
    }

    /**
     * @param id The ID of the segment to be updated
     * @param averageGradient The new average gradient for the
         segment instance
     * @throws IDNotRecognisedException If no segment exists
         with the requested
     *                                  ID
     */
    public static void setSegmentAverageGradient(int id, double
         averageGradient)
                                                 throws
                                                     IDNotRecognisedException
                                                      {
        getSegment(id).setSegmentAverageGradient(
            averageGradient);
    }

    /**
     * @param length The new length for the segment instance
     */
    public void setSegmentLength(double length) {
        this.segmentLength = length;
    }

    /**
     * @param id The ID of the segment to be updated
     * @param length The new length for the segment instance
     * @throws IDNotRecognisedException If no segment exists
         with the requested
     *                                  ID
     */
    public static void setSegmentLength(int id, double length)
        throws
                                        IDNotRecognisedException
                                             {
        getSegment(id).setSegmentLength(length);
    }
}
```

6 Result.java

```
package cycling;

import java.util.ArrayList;
import java.util.Arrays;
import java.io.Serializable;
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;
import java.time.temporal.ChronoUnit;

/**
```

```
 * Result encapsulates rider results per stage, and handles
   time adjustments and
 * rankings (scoring is done externally based on points
   distributions defined in
 * Cycling Portal)
 *
 * @author Thomas Newbold
 * @version 1.1
 */
public class Result implements Serializable {
    // Static class attributes
    public static ArrayList<Result> allResults = new ArrayList<
        Result>();

    /**
     * @param stageId The ID of the stage
     * @return An array of all results for a stage
     */
    public static Result[] getResultsInStage(int stageId) {
        ArrayList<Result> stage = new ArrayList<Result>();
        for(Result r : allResults) {
            stage.add(r);
        }
        stage.removeIf(r -> r.getStageId()!=stageId);
        Result[] resultsForStage = new Result[stage.size()];
        for(int i=0; i<stage.size(); i++) {
            resultsForStage[i] = stage.get(i);
        }
        return resultsForStage;
    }

    /**
     * @param riderId The ID of the driver
     * @return An array of all results for a driver
     */
    public static Result[] getResultsForRider(int riderId) {
        ArrayList<Result> rider = new ArrayList<Result>(
            allResults);
        rider.removeIf(r -> r.getRiderId()!=riderId);
        Result[] resultsForRider = new Result[rider.size()];
        for(int i=0; i<rider.size(); i++) {
            resultsForRider[i] = rider.get(i);
        }
        return resultsForRider;
    }

    // Instance attributes
    private int stageId;
    private int riderId;
    private LocalTime[] checkpoints;
```

```java

    /**
     * Result constructor; creates a new result entry and adds
        to the
     * allResults array.
     *
     * @param sId The ID of the stage the result refers to
     * @param rId The ID of the rider who achieved the result
     * @param check An array of times at which the rider
        reached each
     *              checkpoint (including start and finish)
     */
    public Result(int sId, int rId, LocalTime... check) {
        this.stageId = sId;
        this.riderId = rId;
        this.checkpoints = check;
        Result.allResults.add(this);
    }

    /**
     * @return A string representation of the Result instance
     */
    public String toString() {
        String sId = Integer.toString(this.stageId);
        String rId = Integer.toString(this.riderId);
        int l = this.getCheckpoints().length;
        String times[] = new String[l];
        DateTimeFormatter formatter = DateTimeFormatter.
            ofPattern("HH:mm:ss");
        for(int i=0; i<l; i++) {
            times[i] = this.getCheckpoints()[i].format(
                formatter);
        }
        return String.format("Stage[%s]-Rider[%s]: SplitTimes=%
            s; Total=%s",
                             sId, rId, Arrays.toString(times),
                             getTotalElasped().format(formatter
                                 ));
    }

    /**
     * @param sId The ID of the stage of the result instance
     * @param rId The ID of the associated rider to the result
        instance
     * @return The Result instance
     * @throws IDNotRecognisedException If an instance for the
        rider/stage
     *                                  combination is not
        found in the
     *                                  allResults array
     */
    public static Result getResult(int sId, int rId) throws IDNotRecognisedException {
        for(Result r : allResults) {
            if(r.getRiderId()==rId && r.getStageId()==sId) {
                return r;
            }
        }
        throw new IDNotRecognisedException("results not found for rider in stage");
    }

    /**
     * @param sId The ID of the stage of the result instance to remove
     * @param rId The ID of the associated rider to the result instance to remove
     * @throws IDNotRecognisedException If an instance for the rider/stage
     *                                  combination is not found in the
     *                                  allResults array
     */
    public static void removeResult(int sId, int rId) throws IDNotRecognisedException {
        for(Result r : allResults) {
            if(r.getRiderId()==rId && r.getStageId()==sId) {
                allResults.remove(r);
                break;
            }
        }
        throw new IDNotRecognisedException("results not found for rider in stage");
    }

    /**
     * @return The stageId of the stage the result refers to
     */
    public int getStageId() { return this.stageId; }

    /**
     * @return The riderId of the rider associated with the result
     */
    public int getRiderId() { return this.riderId; }

    /**
     * @return An array of the split times between each checkpoint
     */
    public LocalTime[] getCheckpoints() {
        LocalTime[] out = new LocalTime[this.checkpoints.length
            -1];
        for(int n=0;n<this.checkpoints.length-1; n++) {
            out[n] = getElapsed(checkpoints[n],checkpoints[n
                +1]);
        }
        return out;
    }

    /**
     * @return The total time elapsed between the start and end
          checkpoints
     */
    public LocalTime getTotalElasped() {
        LocalTime[] times = this.checkpoints;
        return Result.getElapsed(times[0], times[times.length
            -1]);
    }

    /**
     * @param a Start time
     * @param b End time
     * @return The time difference between two times, a and b
     */
    public static LocalTime getElapsed(LocalTime a, LocalTime b
        ) {
        int hours = (int)a.until(b, ChronoUnit.HOURS);
        int minuites = (int)a.until(b, ChronoUnit.MINUTES);
        int seconds = (int)a.until(b, ChronoUnit.SECONDS);
        return LocalTime.of(hours%24, minuites%60, seconds%60);
    }

    /**
     * @return An array of the checkpoint times, adjusted to a
         threshold of
     *         one second
     */
    public LocalTime[] adjustedCheckpoints() {
        LocalTime[] adjusted = this.getCheckpoints();
        for(int n=0; n<adjusted.length; n++) {
            adjusted[n] = adjustedCheckpoint(n);
        }
        return adjusted;
    }

    /**
     * Recursive adjuster, used in {@link #adjustedCheckpoints
         ()}.
     *
```

```java
     * @param n The index of the checkpoint to adjust
     * @return The adjusted time for checkpoint n
     */
    public LocalTime adjustedCheckpoint(int n) {
        for(int i=0; i<allResults.size(); i++) {
            Result r = allResults.get(i);
            if(r.getRiderId()==this.getRiderId() && r.
                getStageId()==this.getStageId()) {
                continue;
            }
            LocalTime selfTime = this.getCheckpoints()[n];
            LocalTime rTime = r.getCheckpoints()[n];
            if(selfTime.until(rTime, ChronoUnit.SECONDS)<1) {
                return r.adjustedCheckpoint(n);
            } else {
                return selfTime;
            }
        }
        return null;
    }
}
```

7 Team.java

```java
package cycling;
import java.io.Serializable;
import java.util.ArrayList;
/**
 * Team Class holds the teamId,name,description and riderIds
     belonging to that team.
 *
 *
 * @author Ethan Ray
 * @version 1.0
 *
 */

public class Team implements Serializable {
    public static ArrayList<String> teamNames = new ArrayList
        <>();
    public static int teamTopId = 0;

    private int teamID;
    private String name;
    private String description;
    private ArrayList<Integer> riderIds = new ArrayList<>();


    /**
```

```java
     * @param name String - A name for the team, , If the name is null, empty, has more than 30 characters, or has white spaces will throw InvaildNameException.
     * @param description String - A description for the team.
     * @throws IllegalNameException name String - Is a duplicate name of any other Team, IllegalNameException will be thrown.
     * @throws InvailNameException name String - If the name is null, empty, has more than 30 characters, or has white spaces will throw InvaildNameException.
     */
    public Team(String name, String description) throws IllegalNameException, InvalidNameException
    {
        if (name == "" || name.length()>30 || name.contains(" ")){
            throw new InvalidNameException("Team name cannot be empty, longer than 30 characters , or has white spaces.");
        }
        for (int i = 0;i<teamNames.size();i++){
            if (teamNames.get(i) == name){
                throw new IllegalNameException("That team name already exsists!");
            }
        }

        teamNames.add(name);
        this.teamID = teamTopId++;
        this.name = name;
        this.description = description;
    }
    /**
     * @param rider Rider - A rider to add to the team.
     */
    public void addRider(Rider rider){

        this.riderIds.add(rider.getRiderId());
    }
    /**
     * @param riderId int - A riderId to be removed from the team.
     */
    public void removeRiderId(int riderId){
        for (int i =0;i<this.riderIds.size();i++){
            if (this.riderIds.get(i)==riderId){
                this.riderIds.remove(i);
                break;
            }
        }
    }
    /**
     * @return An Array of integers - which are the riderIds in
        that team.
     */
    public int[] getRiderIds(){
        int [] currentRiderIds = new int[this.riderIds.size()];
        for (int i=0; i<this.riderIds.size();i++){
            currentRiderIds[i]=this.riderIds.get(i);
        }
        return currentRiderIds;
    }
    /**
     * @return A Integer - teamId of the team.
     */
    public int getId(){
        return this.teamID;
    }
    /**
     * @return A String - Name of the team.
     */
    public String getTeamName(){
        return this.name;
    }
    /**
     * @return A String - The description of the team.
     */
    public String getDescription(){
        return this.description;
    }
}
```

8 Rider.java

```
package cycling;

import java.io.Serializable;

/**
 * Rider Class holds the riders teamId,riderId,name and
    yearOfBirth
 *
 *
 * @author Ethan Ray
 * @version 1.0
 *
 */


```

```java
public class Rider implements Serializable {
    public static int ridersTopId;
    private int riderId;
    private int teamID;
    private String name;
    private int yearOfBirth;


    /**
     * @param teamID int - A team Id that the rider will belong
         too
     * @param name String - A name for the rider, Has to be non
        -null or IllegalArgumentException is thrown.
     * @param yearOfBirth int - A year that the rider was born
        in. Has to be above 1900 or IllegalArgumentException is
         thrown.
     * @throws IllegalArgumentException name String - Has to be
         non-null or IllegalArgumentException is thrown.
     * @throws IllegalArgumentException yearOfBirth int - A
        year that the rider was born in. Has to be above 1900
        or IllegalArgumentException is thrown.
     */
    public Rider(int teamID, String name, int yearOfBirth)
        throws IllegalArgumentException
    {
        this.riderId = ridersTopId++;
        this.teamID = teamID;
        if (name == "" || name == null){
            throw new IllegalArgumentException("Illegal name
                entered for rider");
        }
        this.name = name;
        if (yearOfBirth < 1900){
            throw new IllegalArgumentException("Illegal value
                for yearOfBirth given please enter a value above
                 1900.");
        }
        this.yearOfBirth = yearOfBirth;
    }
    /**
     * @return The RiderId of the rider.
     */
    public int getRiderId(){
        return this.riderId;
    }
    /**
     * @return The team Id that the rider belongs to/
     */
    public int getRiderTeamId(){
        return this.teamID;
    }
    /**
     * @return The rider's name.
     */
    public String getRiderName(){
        return this.name;
    }
    /**
     * @return The the year of birth of the rider.
     */
    public int getRiderYOB(){
        return this.yearOfBirth;
    }

}
```

9 RiderManager.java

```java
package cycling;

import java.io.Serializable;
import java.util.ArrayList;

public class RiderManager implements Serializable{
    public static ArrayList<Rider> allRiders = new ArrayList
        <>();
    public static ArrayList<Team> allTeams = new ArrayList<>();


    /**
     * @param teamID int - A team Id that the rider will belong
         too. If the ID doesn't exist IDNotRecognisedException
        is thrown.
     * @param name String - A name for the rider, Has to be non
        -null or IllegalArgumentException is thrown.
     * @param yearOfBirth int - A year that the rider was born
        in. Has to be above 1900 or IllegalArgumentException is
         thrown.
     * @return riderId of the rider created.
     * @throws IDNotRecognisedException teamId int - If the ID
        doesn't exist IDNotRecognisedException is thrown.
     * @throws IllegalArgumentException yearOfBirth int - A
        year that the rider was born in. Has to be above 1900
        or IllegalArgumentException is thrown.
     */
    int createRider(int teamID, String name, int yearOfBirth)
        throws IDNotRecognisedException,IllegalArgumentException
        {
        int teamIndex = getIndexForTeamId(teamID);
        Rider newRider = new Rider(teamID,name,yearOfBirth);
        allRiders.add(newRider);
        Team ridersTeam = allTeams.get(teamIndex);
        ridersTeam.addRider(newRider);
        return newRider.getRiderId();
    }
    /**
     * @param riderId int - A riderId of a rider to be removed. If the ID doesn't exist IDNotRecognisedException is thrown.
     * @throws IDNotRecognisedException riderId int - If the ID doesn't exist IDNotRecognisedException is thrown.
     */
    void removeRider(int riderId) throws IDNotRecognisedException
    {
        int riderIndex = getIndexForRiderId(riderId);
        int teamId = allRiders.get(riderIndex).getRiderTeamId();
        int teamIndex = getIndexForTeamId(teamId);
        Team riderTeam = allTeams.get(teamIndex);
        riderTeam.removeRiderId(riderId);
        allRiders.remove(riderIndex);
    }
    /**
     * @param riderId int - A riderId of a rider to be searced for. If the ID doesn't exist IDNotRecognisedException is thrown.
     * @throws IDNotRecognisedException riderId int - If the ID doesn't exist IDNotRecognisedException is thrown.
     * @return An int which is the index that maps to the riderId.
     */
    int getIndexForRiderId(int riderId) throws IDNotRecognisedException{
        int index =-1;
        if (allRiders.size() == 0){
            throw new IDNotRecognisedException("No rider exists with that ID");
        }
        for (int i=0; i<allRiders.size();i++){
            if (allRiders.get(i).getRiderId()==riderId){
                index = i;
                break;
            }
        }
        if (index == -1){
            throw new IDNotRecognisedException("No rider exists with that ID");
        }
        return index;
    }
    /**
     * @param name String - A name for the team, , If the name
         is null, empty, has more than 30 characters, or has
         white spaces will throw InvaildNameException.
     * @param description String - A description for the team.
     * @throws IllegalNameException name String - Is a
         duplicate name of any other Team, IllegalNameException
         will be thrown.
     * @throws InvailNameException name String - If the name is
          null, empty, has more than 30 characters, or has white
          spaces will throw InvaildNameException.
     */
    int createTeam(String name, String description) throws
        IllegalNameException, InvalidNameException{
        Team newTeam = new Team(name,description);
        allTeams.add(newTeam);
        return newTeam.getId();
    }
    /**
     * @param teamId int - A teamId of a rider to be removed.
         If the ID doesn't exist IDNotRecognisedException is
         thrown.
     * @throws IDNotRecognisedException riderId int - If the ID
          doesn't exist IDNotRecognisedException is thrown.
     */
    void removeTeam(int teamId) throws IDNotRecognisedException
        { // Delete team and all riders in that team
        int teamIndex = getIndexForTeamId(teamId);
        Team currentTeam = allTeams.get(teamIndex);
        for (Integer riderId : currentTeam.getRiderIds()) {
            removeRider(riderId);
        }
        allTeams.remove(teamIndex);

    }
    /**
     * @return All the teamId's that are currently in the
         system as an int[]
     */

    int[] getTeams(){
        int [] allTeamIds = new int[allTeams.size()];
        for (int i=0; i<allTeams.size();i++){
            allTeamIds[i]=allTeams.get(i).getId();
        }
        return allTeamIds;
    }
    /**
```

```java
     * @param teamId int - A teamId to get RidersId in that
        team. If the ID doesn't exist IDNotRecognisedException
        is thrown.
     * @throws IDNotRecognisedException teamId int - If the ID
        doesn't exist IDNotRecognisedException is thrown.
     * @return All the riderId's in a team as an int[]
     */
    int[] getTeamRiders(int teamId) throws
        IDNotRecognisedException{
        Team currentTeam = getTeam(teamId);
        return currentTeam.getRiderIds();

    }
    /**
     * @return All team names in the system as an String[]
     */
    String[] getTeamsNames(){
        String [] allTeamNames = new String[allTeams.size()];
        for (int i=0; i<allTeams.size();i++){
            allTeamNames[i] = allTeams.get(i).getTeamName();
        }
        return allTeamNames;
    }
    /**
     * @return All rider names in the system as an String[]
     */
    String[] getRidersNames(){
        String [] allRiderNames = new String[allRiders.size()];
        for (int i=0; i<allRiders.size();i++){
            allRiderNames[i] = allRiders.get(i).getRiderName();
        }
        return allRiderNames;
    }
    /**
     * @param teamId int - A teamId of a team to search for its
         index. If the ID doesn't exist
        IDNotRecognisedException is thrown.
     * @throws IDNotRecognisedException teamId int - If the ID
        doesn't exist IDNotRecognisedException is thrown.
     * @return An int which is the index that maps to the
        teamId.
     */
    int getIndexForTeamId(int teamId) throws
        IDNotRecognisedException{
        int index =-1;
        if (allTeams.size() == 0){
            throw new IDNotRecognisedException("No Team exists
                with that ID");
        }
        for (int i=0; i<allTeams.size();i++){
            if (allTeams.get(i).getId()==teamId){
                index = i;
                break;
            }
        }
        if (index == -1){
            throw new IDNotRecognisedException("No rider exists
                 with that ID");
        }
        return index;
    }
    /**
     * @param teamId int - A teamId of a team to search for its
          object. If the ID doesn't exist
         IDNotRecognisedException is thrown.
     * @throws IDNotRecognisedException teamId int - If the ID
         doesn't exist IDNotRecognisedException is thrown.
     * @return A Team object with the teamId parsed.
     */
    Team getTeam(int teamId) throws IDNotRecognisedException{
        int teamIndex = getIndexForTeamId(teamId);
        return allTeams.get(teamIndex);
    }
    /**
     * @param riderId int - A riderId of a team to search for
         its object. If the ID doesn't exist
         IDNotRecognisedException is thrown.
     * @throws IDNotRecognisedException riderId int - If the ID
          doesn't exist IDNotRecognisedException is thrown.
     * @return A Rider object with the riderId parsed.
     */
    Rider getRider(int riderId) throws IDNotRecognisedException
        {
        int riderIndex = getIndexForRiderId(riderId);
        return allRiders.get(riderIndex);
    }
    void setAllTeams(ArrayList<Team> allTeams){

        RiderManager.allTeams = allTeams;
        if (allTeams.size() != 0){
        Team lastTeam = allTeams.get(allTeams.size()-1);
        Team.teamTopId = lastTeam.getId()+1;
        }
    }
    void setAllRiders(ArrayList<Rider> allRiders){
        RiderManager.allRiders = allRiders;
        if (allRiders.size() != 0){
            Rider lastRider = allRiders.get(allRiders.size()-1)
                ;
            Rider.ridersTopId = lastRider.getRiderId()+1;
        }
    }
    int [] getRiderIds(){
        int[] riderIdArray = new int[allRiders.size()];
        int count = 0;
        for (Rider rider : RiderManager.allRiders){
            riderIdArray[count] = rider.getRiderId();
            count++;

        }
        return riderIdArray;
    }

}
```