1 CyclingPortal.java

```java
package cycling;

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

/**
 * Cycling Portal implaments CyclingPortalInterface class
 *
 * @author Ethan Hofton
 * @author Jon Tao
 * @version 1.0
 */
public class CyclingPortal implements CyclingPortalInterface {

    private ArrayList<Team> teams;
    private ArrayList<Race> races;

    /**
     * CyclingPortal constructor initalises teams and races array list
     */
    public CyclingPortal() {
        // constructior to init lists
        teams = new ArrayList<>();
        races = new ArrayList<>();
    }

    private Team findTeam(int teamID) throws IDNotRecognisedException {
        // check if the list 'teams' has teamID
        // O(n)

        // loop throguh teams list and cheack the team class's id
        // against the given id teamID
        for (int i = 0; i < teams.size(); i++) {
            if (teams.get(i).getTeamId() == teamID) {
                return teams.get(i);
            }
        }

        // throw IDNotRecognisedException if not found
        throw new IDNotRecognisedException("Team Id '"+teamID+"' not found");
    }

    private Rider findRider(int riderID) throws IDNotRecognisedException {
        // check if the list 'teams' has teamID

        // loop through each team and check if any of the riders on that team
        // match the given rider id
        for (int i = 0; i < teams.size(); i++) {
            for (int j = 0; j < teams.get(i).getRiders().size(); j++) {
                if (teams.get(i).getRiders().get(j).getRiderId() == riderID) {
                    return teams.get(i).getRiders().get(j);
                }
            }
        }

        // throw IDNotRecognisedException if not found
        throw new IDNotRecognisedException("Rider Id '"+riderID+"' not found");
    }

    private Race findRace(int raceID) throws IDNotRecognisedException {
        // check if the list 'races' has raceID

        // loop through races list and check given raceID
        // against the race objects id
        for (int i = 0; i < races.size(); i++) {
            if (races.get(i).getRaceId() == raceID) {
                return races.get(i);
            }
        }

        // throw IDNotRecognisedException if not found
        throw new IDNotRecognisedException("Race Id '"+raceID+"' not found");
    }

    private Stage findStage(int stageId) throws IDNotRecognisedException {
        // check if the list 'races' has stageId

        // loop though each race and loop through each races' stages
        // if stage matches given id, return the stage
        for (int i = 0; i < races.size(); i++) {
            for (int j = 0; j < races.get(i).getStages().size(); j++) {
                if (races.get(i).getStages().get(j).getStageId() == stageId) {
                    return races.get(i).getStages().get(j);
                }
            }
        }

        throw new IDNotRecognisedException("Stage Id '"+stageId+"' not found");
    }

    private Segment findSegment(int segmentId) throws IDNotRecognisedException {
        // check if the list 'races' has Segment with id segmentId

        // loop through each races stages' segments
        // if the segment id matches the given id, return that segment
        for (int i = 0; i < races.size(); i++) {
            Race currentRace = races.get(i);
            for (int j = 0; j < currentRace.getStages().size(); j++) {
                Stage currentStage = currentRace.getStages().get(j);
                for (int m = 0; m < currentStage.getSegments().size(); m++) {
                    Segment currentSegment = currentStage.getSegments().get(m);
                    if (currentSegment.getSegmentId() == segmentId) {
                        return currentSegment;
                    }
                }
            }
        }

        throw new IDNotRecognisedException("Segment Id '"+segmentId+"' not found");
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int[] getRaceIds() {

        // loop thorugh each race in race list and add races id
        // to a list of ids, return this list
        int raceIds[] = new int[races.size()];
        for (int i = 0; i < races.size(); i++) {
            raceIds[i] = races.get(i).getRaceId();
        }

        return raceIds;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int createRace(String name, String description) throws IllegalNameException, InvalidNameException {

        // erronus arguments checking
        // check if the name is null, empty, contains wihitespace or is longer the 30 charicters
        if (name == null || name.equals("") || name.length() > 30 || name.contains(" ")) {
            // throw an error if name does not meet these paramiters
            throw new InvalidNameException("name cannot be null, empty, have more than 30 characters or contain white spaces");
        }

        // check if the name allready exists in the platform
        // loop through each race and check if the races name matches the given input name
        for (int i = 0; i < races.size(); i++) {
            if (name.equals(races.get(i).getName())) {
                // theow exception if the name allreadt exists on platform
                throw new IllegalNameException("name alrwdy exists in platform");
            }
        }

        // create a new race
        Race race = new Race(name, description);

        // add the race to the cycling portals array list of races
        races.add(race);

        // return the race id
        return race.getRaceId();
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public String viewRaceDetails(int raceId) throws IDNotRecognisedException {

        // find the race object in the system
        // throws IDNotRecognisedException if the id does not exist on the platform
        Race race = findRace(raceId);

        // find the total length
        // init total length to zero
        double totalLen = 0.0;

        // loop through each stage in the race and add the stage length to the total length
        for (Stage stage : race.getStages()) {
            totalLen += stage.getLength();
        }

        // stringify race details using race peramiters
        String raceDetails = "raceID="+raceId;
        raceDetails += ",name="+race.getName();
        raceDetails += ",description="+race.getDescription();
        raceDetails += ",numberOfStages="+race.getStages().size();
        raceDetails += ",totalLength="+totalLen;

        // return the stringified race detials
        return raceDetails;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void removeRaceById(int raceId) throws IDNotRecognisedException {
        // find the race class in the portal
        Race raceToRemove = findRace(raceId);

        // removing race from the system also removes all related data
        // since the race itself is the only thing that holds references to those
        // related data classes
        // remove the race class from the races array list
        races.remove(raceToRemove);
    }

```

```java
    /**
     * {@inheritDoc}
     */
    @Override
    public int getNumberOfStages(int raceId) throws IDNotRecognisedException {
        // find the race within the portal
        Race race = findRace(raceId);

        // return the size of the array that stores the stages
        return race.getStages().size();
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int addStageToRace(int raceId, String stageName, String description, double length,
         LocalDateTime startTime,
            StageType type)
            throws IDNotRecognisedException, IllegalNameException, InvalidNameException,
                InvalidLengthException {

        // find race in portal
        Race race = findRace(raceId);

        // loop throguh all the stages in the race
        for (int i = 0; i < race.getStages().size(); i++) {
            // check if the name allready exists in the race
            // compare each stage name to the new stage name
            if (race.getStages().get(i).getStageName().equals(stageName)) {
                // if stage name allready excists throw an IllegalNameException
                throw new IllegalNameException("name already exists on platform");
            }
        }

        // check if the stage name is null, empty or grater than 30 charicters
        if (stageName == null || stageName.equals("") || stageName.length() > 30) {
            // throw InvalidNameException if paramaters are met
            throw new InvalidNameException("Name cannot be null, empty or more than 30 characters");
        }

        // check if the stage length is less then 5km
        if (length < 5) {
            // throw InvalidLengthException
            throw new InvalidLengthException("Length cannot be less than 5km");
        }

        // create the new stage
        Stage stage = new Stage(race, stageName, description, length, startTime, type);

        // add the stage to the race
        race.addStage(stage);

        // return the stage id
        return stage.getStageId();
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int[] getRaceStages(int raceId) throws IDNotRecognisedException {
        // find the race in the portal
        Race race = findRace(raceId);

        // initalise stage id list to return
        // set array to the size of the number of stages for that stage
        int stageIds[] = new int[race.getStages().size()];

        // loop through all the stages in the race
        for (int i = 0; i < stageIds.length; i++) {
            // set each value of the array to the corrisponding stage id
            stageIds[i] = race.getStages().get(i).getStageId();
        }

        // return the list of stage ids
        return stageIds;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public double getStageLength(int stageId) throws IDNotRecognisedException {
        // find the stage in the system
        Stage stage = findStage(stageId);

        // return the length of the stage
        return stage.getLength();
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void removeStageById(int stageId) throws IDNotRecognisedException {
        // find the stage in the portal
        Stage stage = findStage(stageId);

        // removing the stage also removes all stage related data
        // this is because the stage class is the only class that stores a referance
        // to these classes
        //
        // remove the stage from the race
        stage.getRace().removeStage(stage);
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int addCategorizedClimbToStage(int stageId, Double location, SegmentType type, Double
         averageGradient,
            Double length) throws IDNotRecognisedException, InvalidLocationException,
                InvalidStageStateException,
            InvalidStageTypeException {

        // a climb segment cannot be a sprint
        if (type == SegmentType.SPRINT) {
            // throw an illigal argument exception if the given segment time is sprint
            throw new IllegalArgumentException("Segment type is not valid.");
        }

        // find stage in portal
        // throws IDNotRecognisedException
        Stage stage = findStage(stageId);

        // check if the segment location is out of bounds of the stage
        if (stage.getLength() < location) {
            // throw InvalidLocationException
            throw new InvalidLocationException("location is out of bounds of the stage length");
        }

        // check if the stage stage is correct
        // cannot add a new segment if the stage has conculded the stage preperation
        if (stage.getStageState() == StageState.WAITING_FOR_RESULTS) {
            // throw InvalidStageStateException
            throw new InvalidStageStateException("Stage cannot be added while waiting for results");
        }

        // time trial stages cannot contain a segment
        // check if the stage type is time trial
        if (stage.getType() == StageType.TT) {
            // if the type is a time trial, throw an InvalidStageTypeException
            throw new InvalidStageTypeException("Time-trial stages cannot contain any segment");
        }

        // create new climb segment with the paramiters
        ClimbSegment segment = new ClimbSegment(stage, location, type, averageGradient, length);

        // add the segment to the stage
        stage.addSegment(segment);

        // return the id of the new segment
        return segment.getSegmentId();
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int addIntermediateSprintToStage(int stageId, double location) throws IDNotRecognisedException,
            InvalidLocationException, InvalidStageStateException, InvalidStageTypeException {

        // find stage in portal
        // trows IDNotRecognisedException
        Stage stage = findStage(stageId);

        // check the location is in bounds of the stage
        if (stage.getLength() < location) {
            // throw InvalidLocationException if out of bounds
            throw new InvalidLocationException("location is out of bounds of the stage length");
        }

        // cannot add segment if stage has fininished stage preperation
        // check the stage state is not waiting for results
        if (stage.getStageState() == StageState.WAITING_FOR_RESULTS) {
            // throw InvalidStageStateException
            throw new InvalidStageStateException("Stage cannot be removed while waiting for results");
        }

        // time trial stages cannot have any segments
        // check the stage type is not time trial
        if (stage.getType() == StageType.TT) {
            // if the stage type is time trial, throw InvalidStageTypeException
            throw new InvalidStageTypeException("Time-trial stages cannot contain any segment");
        }

        // create a new sprint segment
        SprintSegment segment = new SprintSegment(stage, location);

        // add sprint segment to stage
        stage.addSegment(segment);

        // return the new segment id
        return segment.getSegmentId();
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void removeSegment(int segmentId) throws IDNotRecognisedException, InvalidStageStateException {

        // find segment in portal
        // throws IDNotRecognisedException
        Segment segmentToRemove = findSegment(segmentId);

        // get the stage the segment belongs to
        Stage stage = segmentToRemove.getStage();

        // cannot remove segment if stage preperation has finsihed
        // check the state of the stage is not waiting for results
        if (stage.getStageState() == StageState.WAITING_FOR_RESULTS) {
            // if stage state is wiating for results, throw InvalidStageStateException
            throw new InvalidStageStateException("Stage cannot be removed while waiting for results");
        }

        // remove segment from stage
        stage.removeSegment(segmentToRemove);
```

```java
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void concludeStagePreparation(int stageId) throws IDNotRecognisedException,
        InvalidStageStateException {
        // find the stage in the portal
        // throws IDNotRecognisedExceiption
        Stage stage = findStage(stageId);

        // conculde the stage preperation
        // throws InvalidStageStateException
        stage.concludeStagePreparation();
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int[] getStageSegments(int stageId) throws IDNotRecognisedException {

        // find the stage in the portal
        // throws IDNotRecognisedExceiption
        Stage stage = findStage(stageId);

        // init new array the size of the number of segments in the stage
        int[] stageSegmentIds = new int[stage.getSegments().size()];

        // loop through each segment in the stage
        for (int i = 0; i < stageSegmentIds.length; i++) {
            // add the segments id to the respective index in the array
            stageSegmentIds[i] = stage.getSegments().get(i).getSegmentId();
        }

        // return the segment ids
        return stageSegmentIds;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int createTeam(String name, String description) throws IllegalNameException,
        InvalidNameException {

        // check if team name allready exists
        // loop through each time
        for (Team team : teams) {
            // check if the team name is equal to the new team name
            if (name.equals(team.getTeamName())) {
                // if equal, throw IllegalNameException
                throw new IllegalNameException("Team name allready exisits");
            }
        }

        // check the desciption
        // the description has to be less then 30 chars, not null and not empty
        if (name.length() > 30 || name.equals("") || name == null) {
            // throw InvalidNameException if params are not met
            throw new InvalidNameException("Name cannot be null, empty or longer then 30");
        }

        // create a new team and add it to the teams array list
        Team newTeam = new Team(name, description);
        teams.add(newTeam);

        // return the new teams id
        return newTeam.getTeamId();
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void removeTeam(int teamId) throws IDNotRecognisedException {

        // find the team in the portal
        // throws IDNotRecognisedException
        Team teamToRemove = findTeam(teamId);

        // remove the team referance from the teams array list
        // the team is the only object that stores the team realted data
        // threfore, deleting the team also deletes all its related data
        teams.remove(teamToRemove);
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int[] getTeams() {
        // return the ids as an array of all the teams
        // init new array the size of the numnber of teams in the portal
        int[] teamsToReturn = new int[teams.size()];

        // loop through each value in the array
        for (int i = 0; i < teams.size(); i++) {
            // add the team id to the respective index in the array
            teamsToReturn[i] = teams.get(i).getTeamId();
        }

        // return the array
        return teamsToReturn;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int[] getTeamRiders(int teamId) throws IDNotRecognisedException {
        // find team in portal
        // Throws IDNotRecognisedException
        Team team = findTeam(teamId);

        // create an array the size of all the riders there are in the given team
        int teamRiders[] = new int[team.getRiders().size()];

        // loop through each rider in the team
        for (int i = 0; i < team.getRiders().size(); i++) {
            // add there id to the array to there corrisponding index
            teamRiders[i] = team.getRiders().get(i).getRiderId();
        }

        // return the array of rider ids
        return teamRiders;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int createRider(int teamID, String name, int yearOfBirth) throws IDNotRecognisedException,
         IllegalArgumentException {

        // check that the rider name is not null
        // and the year of birth is not before 1900
        if (name == null || yearOfBirth < 1900) {
            // if the name or year of birth breaks these paramiters, throw IllegalArgumentException
            throw new IllegalArgumentException("name cannot be null or year less then 1900");
        }

        // find the riders team in the portal
        // throws IDNotRecognisedException
        Team ridersTeam = findTeam(teamID);

        // create a new rider
        Rider newRider = new Rider(ridersTeam, name, yearOfBirth);

        // add the rider to the tema
        ridersTeam.addRider(newRider);

        // return the new riders id
        return newRider.getRiderId();
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void removeRider(int riderId) throws IDNotRecognisedException {

        // find rider in portal
        // throws IDNotRecognisedException
        Rider rider = findRider(riderId);

        // remove the rider from the team
        rider.getTeam().removeRider(rider);

        // remove rider race results
        // loop through each race in the portal
        for (int i = 0; i < races.size(); i++)
        {
            // store the race
            Race race = races.get(i);
            // loop through each races stages
            for (int j = 0; j < race.getStages().size(); j++)
            {
                // store the stage
                Stage stage = race.getStages().get(i);

                // create a tempory array to store all the results that need to be
                // removed from the stage as they referance rider
                ArrayList<Results> resultsToRemove = new ArrayList<>();
                for (int m = 0; m < stage.getResults().size(); m++)
                {
                    // store the result
                    Results result = stage.getResults().get(m);
                    // check if the riders id of the result matches the given rider id to remove
                    if (result.getRider().getRiderId() == riderId)
                    {
                        // if the result needs to be removed, add it to the remove list
                        resultsToRemove.add(result);
                    }
                }

                // loop through each result to remove
                for (Results result : resultsToRemove)
                {
                    // remove result from stage
                    stage.removeResults(result);
                }
            }
        }
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void registerRiderResultsInStage(int stageId, int riderId, LocalTime... checkpoints)
            throws IDNotRecognisedException, DuplicatedResultException, InvalidCheckpointsException,
            InvalidStageStateException {

        // find rider in portal
        // throws IDNotRecognisedException
        Rider rider = findRider(riderId);

        // find stage in portal
```

```java
        // throws IDNotRecognisedException
        Stage stage = findStage(stageId);

        // check rider does not have duplicate result
        // loop through each result in stage
        for (int i = 0; i < stage.getResults().size(); i++) {
            // check the rider does not have a result by
            // comparing the riders id with the stages riders id
            if (stage.getResults().get(i).getRider() == rider) {
                // duplicate found
                // throw DuplicatedResultException
                throw new DuplicatedResultException("Stage allready has results for rider");
            }
        }

        // check length of checkpoints is equal to n+2
        if (checkpoints.length != stage.getSegments().size() + 2) {
            // throw InvalidCheckpointsException
            throw new InvalidCheckpointsException("length of checkpoints is invalid");
        }

        // check if stage is "waiting for results"
        if (stage.getStageState() != StageState.WAITING_FOR_RESULTS) {
            // stage waiting for results, throwInvalidStageStateException
            throw new InvalidStageStateException("Invalid stage state");
        }

        // create a new result
        Results result = new Results(stage, rider, checkpoints);

        // add result to stage
        stage.addResults(result);
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public LocalTime[] getRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException {

        // find stage in portal
        // throws IDNotRecognisedException
        Stage stage = findStage(stageId);

        // find rider in portal
        // throws IDNotRecognisedException
        Rider rider = findRider(riderId);

        // init rider result to null
        Results riderResult = null;

        // find rider results
        // loop through each stages result
        for (int i = 0; i < stage.getResults().size(); i++) {
            // if the target riders id matches the stages results rider id
```

```java
                // then rider result found
                if (rider == stage.getResults().get(i).getRider()) {
                    // save the rider result
                    riderResult = stage.getResults().get(i);
                }
            }

            // if the rider result is still null, the result has not been found
            if (riderResult == null) {
                // return an empty localtime array
                return new LocalTime[0];
            }

            // initalise a rider results array that is the size of all of the riders results in the stage
            // add one at the end to store the elapsed time
            LocalTime[] riderResults = new LocalTime[riderResult.getTimes().length + 1];

            // loop through all the results times
            for (int i = 0; i < riderResult.getTimes().length; i++) {
                // store the result times in the array
                riderResults[i] = riderResult.getTimes()[i];
            }

            // store the elapsed time in the last spot of the array
            // elpased time calculated using result calculateElapsedTime() function
            riderResults[riderResult.getTimes().length] = riderResult.calculateElapsedTime();

            // return the results array
            return riderResults;
        }

        /**
         * {@inheritDoc}
         */
        @Override
        public LocalTime getRiderAdjustedElapsedTimeInStage(int stageId, int riderId) throws
             IDNotRecognisedException {

            // find stage in portal
            // throws IDNotRecognisedException
            Stage stage = findStage(stageId);

            // find the rider in the portal
            // throws IDNotRecognisedException
            Rider rider = findRider(riderId);

            // initalise rider result as null
            Results riderResult = null;

            // find rider results
            // loop through all the resutls in the stage
            for (int i = 0; i < stage.getResults().size(); i++) {
                // cheack if the results rider matches the target rider
                if (rider == stage.getResults().get(i).getRider()) {
                    // if the ids are the same, result is found
                    // store result in riderResult
                    riderResult = stage.getResults().get(i);
                }
            }

            // if riderResult is still null, no result found
            if (riderResult == null) {
                // if not result found, return null
                return null;
            }

            // otherwise, return the riders adjusted elapsed time
            // calculated using results calcaulateAdjustedElapsedTime()
            return riderResult.calculateAdjustedElapsedTime();
        }

        /**
         * {@inheritDoc}
         */
        @Override
        public void deleteRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException {

            // find stage in portal
            // throws IDNOtRecognisedException
            Stage stage = findStage(stageId);

            // find rider in stage
            // throws IDNOtRecognisedException
            Rider rider = findRider(riderId);

            // initalise rider result as null
            Results riderResult = null;

            // find rider results
            // loop through all the results in the stage
            for (int i = 0; i < stage.getResults().size(); i++) {
                // check the results rider id matches target rider id
                if (rider == stage.getResults().get(i).getRider()) {
                    // if ids match, rider result found
                    riderResult = stage.getResults().get(i);
                }
            }

            // if rider result still null, rider result does not excist
            if (riderResult == null) {
                // no results to be removed
                // return
                return;
            }

            // otherwise, remove results from stage
            stage.removeResults(riderResult);
        }

        /**
```

```java
     * {@inheritDoc}
     */
    @Override
    public int[] getRidersRankInStage(int stageId) throws IDNotRecognisedException {

        // find stage in portal
        // throws IDNotRecognisedException
        Stage stage = findStage(stageId);

        // create a list of results the size of all the results in the stage
        Results[] rankedResults = new Results[stage.getResults().size()];

        // loop throguh all results in stage
        for (int i = 0; i < rankedResults.length; i++) {
            // add the results to the results array
            rankedResults[i] = stage.getResults().get(i);
        }

        // sort array of results by there elapsed time
        // do this using custom comparitor class, ResultsElapsedTimeComparator.
        // this returns the differeance between the elapsed times of the results and orders by differance
        Arrays.sort(rankedResults, new ResultsElapsedTimeComparator());

        // create a new array to return the riders results
        // array the size of all theriders in the stage
        int[] riderRanks = new int[rankedResults.length];

        // loop throguh all the results
        for (int i = 0; i < riderRanks.length; i++) {
            // add the rider id to the array
            // since rankedResults is ordered by elapsed time, so will riderRanks
            riderRanks[i] = rankedResults[i].getRider().getRiderId();
        }

        // return the ranked list of riders
        return riderRanks;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId) throws IDNotRecognisedException {

        // find stage in portal
        // throws IDNotRecognisedException
        Stage stage = findStage(stageId);

        // get riders rank in stage
        // throws IDNotRecognisedException
        int[] ridersRanked = getRidersRankInStage(stageId);

        // create a new array of localtimes to store the ranked adjusted elpased times
        LocalTime[] riderAdjustedElapsedTimes = new LocalTime[ridersRanked.length];

        // loop through all the riders from ridersRanked
        for (int i = 0; i < riderAdjustedElapsedTimes.length; i++) {
            // get the rider
            Rider rider = findRider(ridersRanked[i]);

            // loop throguh the stages reults to find the rider result
            for (int x = 0; x < stage.getResults().size(); x++) {
                // cheack if the stage result belongs to the rider
                if (stage.getResults().get(x).getRider() == rider) {
                    // use the found result to calculate the adjusted elapsed time
                    // append to array at index i. Sicne ridersRanked is ordered by elapsed time,
                    // so will riderAdjustedElapsedTimes
                    riderAdjustedElapsedTimes[i] = stage.getResults().get(x).calculateAdjustedElapsedTime();

                    // break out of inner loop to save time
                    continue;
                }
            }
        }

        // return the array or ranked adjusted elapsed times
        return riderAdjustedElapsedTimes;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int[] getRidersPointsInStage(int stageId) throws IDNotRecognisedException {

        // find stage in portal
        // throws IDNotRecognisedException
        Stage stage = findStage(stageId);

        // get a ranked list of rider ids
        // throws IDNotRecognisedException
        int[] ridersRanked = getRidersRankInStage(stageId);

        // init an array to store the ranked riders points
        // the size of the number of riders in the stage
        int[] riderPoints = new int[ridersRanked.length];

        // loop through all the riders
        for (int i = 0; i < riderPoints.length; i++) {
            // get the rider
            Rider rider = findRider(ridersRanked[i]);

            // add the rider points to the riderPoins array at index i
            // since ridersRanked is ordered by elapsed time, so will riderPoints
            // rider points calculated using Rider.getPointsInStage() function
            riderPoints[i] = rider.getPointsInStage(stage, i+1);
        }

        // return the ordered array of rider points
        return riderPoints;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int[] getRidersMountainPointsInStage(int stageId) throws IDNotRecognisedException {

        // find stage in portal
        // throws IDNotRecognisedException
        Stage stage = findStage(stageId);

        // get the ranked list of riders
        // throws IDNotRecognisedException
        int[] ridersRanked = getRidersRankInStage(stageId);

        // init a new array to store the mountain points for a rider in a stage
        int[] riderPoints = new int[ridersRanked.length];

        // loop through each rider in the stage
        for (int i = 0; i < riderPoints.length; i++) {
            // get the rider
            Rider rider = findRider(ridersRanked[i]);

            // set the points at index i in riderPoints for the mountain points that rider has aquired
            // mountain points calculated using Stage.pointsForMountainClassification()
            // since ridersRanked is ordered by elapsed time, so will riderPoints
            riderPoints[i] = stage.pointsForMountainClassification(rider);
        }

        // return the ordered list of mountain points
        return riderPoints;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void eraseCyclingPortal() {

        // clear cycling portal lists
        // clear team
        // clear races
        teams.clear();
        races.clear();

        // reset counters
        // reset race id counter
        // reset rider id counder
        // reset segment id counter
        // reset stage id counter
        // reset ream id counter
        Race.resetCounter();
        Rider.resetCounter();
        Segment.resetCounter();
        Stage.resetCounter();
        Team.resetCounter();
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void saveCyclingPortal(String filename) throws IOException {

        // create a new output file stream
        ObjectOutputStream ostream = new ObjectOutputStream(new FileOutputStream(filename));

        // write the cycling portal to the output stream
        ostream.writeObject(this);

        // close and commit the output stream
        ostream.close();
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void loadCyclingPortal(String filename) throws IOException, ClassNotFoundException {

        // create a new input file stream
        ObjectInputStream istream = new ObjectInputStream(new FileInputStream(filename));

        // create a new object and assin it to the value in the file
        Object portalObject = istream.readObject();

        // chack if the portal is an instance of cycling portal
        if (!(portalObject instanceof CyclingPortal)) {
            // close input file stream
            istream.close();

            // throw exceiption
            throw new InvalidClassException("Object from file is not an instance of cycling portal");
        }

        // otherwise, upcast the portal object to a cycling portal
        CyclingPortal portal = (CyclingPortal)portalObject;

        // assin this cycling portals race list and team list to serialised portals
        this.races = portal.races;
        this.teams = portal.teams;

        // close the input file stream
        istream.close();
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void removeRaceByName(String name) throws NameNotRecognisedException {

        // initalise a race as null
        Race race = null;

        // loop through all the races
        for (int i = 0; i < races.size(); i++) {
            // cheack the target name and races name match
            if (races.get(i).getName() == name) {
                // if they match, assin race to the current race
                race = races.get(i);

                // break out of loop as race allready found
                break;
            }
        }

        // if race still null, race not found
        if (race == null) {
            // throw NameNotRecognisedException
            throw new NameNotRecognisedException("Race is not found with name " + name);
        }

        // remove the race from the portal
        races.remove(race);

        // since stage is stored in race and segments and results are stored in stage
        // deleting the race will also delete segments, results and stage
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int[] getRidersGeneralClassificationRank(int raceId) throws IDNotRecognisedException {
        // find race in portal
        // throws IDNotRecognisedException
        Race race = findRace(raceId);

        // if one of the stages does not have reaults, return an empty array
        // loop through each stage in race
        for (int i = 0; i < race.getStages().size(); i++) {
            // cheack if stage does not have results (results list empty)
            if (race.getStages().get(i).getResults().size() == 0) {
                // return empty array
                return new int[0];
            }
        }

        // initalize an array list of results
        ArrayList<Results> results = new ArrayList<>();

        // loop through each stage in the race
        for (int i = 0; i < race.getStages().size(); i++) {
            // loop throgh each result in the race
            for (int x = 0; x < race.getStages().get(i).getResults().size(); x++) {
                // add result to results array list
                results.add(race.getStages().get(i).getResults().get(x));
            }
        }

        // init a hash map to pair up the rider with there result
        Map<Rider, LocalTime> timesMap = new HashMap<Rider, LocalTime>();

        // loop through all results
        for (int i = 0; i < results.size(); i++) {
            // store the current rider
            Rider currentRider = results.get(i).getRider();

            // check weather the rider has allreayd been entered into the hash map
            if (timesMap.containsKey(currentRider)) {
                // if allready added, add the result adjusted elapsed time to the value allready
                // in the hash map

                // calculate the ammount of nannos the of the adjusted elapsed time
                long nanos = results.get(i).calculateAdjustedElapsedTime().toNanoOfDay();

                // add the nannos to the old value in the hash map
                LocalTime newTime = timesMap.get(currentRider).plusNanos(nanos);

                // relpace the old value in hash map with new value
                // (new value is old time + result time)
                timesMap.replace(currentRider, newTime);
            } else {
                // if the rider has not allready been entered into the hash map,
                // add the rider to the hash map paired with there adjusted elapsed time
                timesMap.put(currentRider, results.get(i).calculateAdjustedElapsedTime());
            }
        }

        // create an array list of map entrys
        ArrayList<Map.Entry<Rider, LocalTime>> sorted = new ArrayList<>(timesMap.entrySet());

        // sort the array of map entrys using the custom comparotor ResultsAdjustedElapsedTimeCompatiror
        // this comparotor comparse Map.Entry<Rider, LocalTime> by returning the differeance between
        // the local times
        sorted.sort(new ResultsAdjustedElapsedTimeCompatiror());

        // init array of rider ids, the size of all the riders in the portal
        int orderedRiderIds[] = new int[sorted.size()];

        // loop through all the riders in the portal
        for (int i = 0; i < orderedRiderIds.length; i++) {
            // add the riders id to the corrisponding index in the array
            // since the id is from the sorted array, orderedRiderIds will be ordered too
            orderedRiderIds[i] = sorted.get(i).getKey().getRiderId();
        }

        // return the list of ordered rider ids
        return orderedRiderIds;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public LocalTime[] getGeneralClassificationTimesInRace(int raceId) throws IDNotRecognisedException {
        // find race in portal
        // throws IDNotRecognisedException
        Race race = findRace(raceId);

        // check if any of the results are empty
        // loop through all the stages in the race
        for (int i = 0; i < race.getStages().size(); i++) {
            // check if the results list for any stage is empty
            if (race.getStages().get(i).getResults().size() == 0) {
                // if empty, return an empty local time array
                return new LocalTime[0];
            }
        }

        // initalize a new array list of results to store all the results for the race
        ArrayList<Results> results = new ArrayList<>();

        // loop through each stage in the race
        for (int i = 0; i < race.getStages().size(); i++) {
            // loop through eahc reuslt in the stage
            for (int x = 0; x < race.getStages().get(i).getResults().size(); x++) {
                // add the result to the results list
                results.add(race.getStages().get(i).getResults().get(x));
            }
        }

        // initalize a hash map to pair together the riders and there times
        Map<Rider, LocalTime> timesMap = new HashMap<Rider, LocalTime>();

        // loop through each result in the race
        for (int i = 0; i < results.size(); i++) {
            // store the current rider
            Rider currentRider = results.get(i).getRider();

            // check if the hash map allready contains an entry for the current rider
            if (timesMap.containsKey(currentRider)) {
                // if the hash map contains an enrty for the rider,
                // add the current results adjusted elapsed time to the value for the rider
                // allready in the hash map

                //calculate the adjusted elapsed time for the current result in nano seconds
                long nanos = results.get(i).calculateAdjustedElapsedTime().toNanoOfDay();

                // add the current results nano seconds to the riders current result
                LocalTime newTime = timesMap.get(currentRider).plusNanos(nanos);

                // replace the old time with the new time
                timesMap.replace(currentRider, newTime);
            } else {
                // if rider does not allreayd have a map entry
                // add them into the hashmap paired with there time
                timesMap.put(currentRider, results.get(i).calculateAdjustedElapsedTime());
            }
        }

        // create an array list of all the map entrys
        ArrayList<Map.Entry<Rider, LocalTime>> sorted = new ArrayList<>(timesMap.entrySet());

        // sort the map using the custom comparitor whitch compares map entrys based of
        // the differance between there LocalTimes
        sorted.sort(new ResultsAdjustedElapsedTimeCompatiror());

        // create an array of localTimes the size of all the riders in the race
        LocalTime orderedTimes[] = new LocalTime[sorted.size()];

        // loop throguh all the riders
        for (int i = 0; i < orderedTimes.length; i++) {
            // set the array to the local time of the sorted lists value at the same index
            // this means the ordered time list will also be sorted the same way the soreded array list is
            orderedTimes[i] = sorted.get(i).getValue();
        }

        // return the array of ordered times
        return orderedTimes;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException {
        // find race in portal
        // throws IDNotRecognisedException
        Race race = findRace(raceId);

        // check if any of the results are empty
        // loop through all the stages in the race
        for (int i = 0; i < race.getStages().size(); i++) {
            // check if the results list for any stage is empty
            if (race.getStages().get(i).getResults().size() == 0) {
                // if empty, return an empty local time array
                return new int[0];
            }
        }

        // initalize a new array list of results to store all the results for the race
        ArrayList<Results> results = new ArrayList<>();

        // loop through each stage in the race
        for (int i = 0; i < race.getStages().size(); i++) {
            // loop through eahc reuslt in the stage
            for (int x = 0; x < race.getStages().get(i).getResults().size(); x++) {
```

```java
                // add the result to the results list
                results.add(race.getStages().get(i).getResults().get(x));
            }
        }

        // initalize a hash map to pair together the riders and there times
        Map<Rider, LocalTime> timesMap = new HashMap<Rider, LocalTime>();

        // loop through each result in the race
        for (int i = 0; i < results.size(); i++) {
            // store the current rider
            Rider currentRider = results.get(i).getRider();

            // check if the hash map allready contains an entry for the current rider
            if (timesMap.containsKey(currentRider)) {
                // if the hash map contains an enrty for the rider,
                // add the current results adjusted elapsed time to the value for the rider
                // allready in the hash map

                //calculate the adjusted elapsed time for the current result in nano seconds
                long nanos = results.get(i).calculateAdjustedElapsedTime().toNanoOfDay();

                // add the current results nano seconds to the riders current result
                LocalTime newTime = timesMap.get(currentRider).plusNanos(nanos);

                // replace the old time with the new time
                timesMap.replace(currentRider, newTime);
            } else {
                // if rider does not allreayd have a map entry
                // add them into the hashmap paired with there time
                timesMap.put(currentRider, results.get(i).calculateAdjustedElapsedTime());
            }
        }

        // create an array list of all the map entrys
        ArrayList<Map.Entry<Rider, LocalTime>> sorted = new ArrayList<>(timesMap.entrySet());

        // sort the map using the custom comparitor whitch compares map entrys based of
        // the differance between there LocalTimes
        sorted.sort(new ResultsAdjustedElapsedTimeCompatiror());

        // init a new array the size of all the riders in the race
        int ridersPoints[] = new int[sorted.size()];
        // loop through all the riders in the race
        for (int i = 0; i < ridersPoints.length; i++) {
            // init there inital points to zero
            ridersPoints[i] = 0;
        }

        // for each rider, find the total points in all stages
        // loop through each stage in the race
        for (int i = 0; i < race.getStages().size(); i++) {
            // store the current stage
            Stage currentStage = race.getStages().get(i);

            // get a list of all the riders rank in that stage
            int ridersRanks[] = getRidersRankInStage(currentStage.getStageId());

            // loop through each riders rank
            for (int x = 0; x < ridersRanks.length; x++) {
                // find the riders id at rank x
                int id = ridersRanks[x];

                // add one to rank (so person with rank 0 is actually 1st)
                int rank = x + 1;

                // loop through the ordered list of adjusted elapsed times and riders
                for (int y = 0; y < sorted.size(); y++) {
                    // check if the rider id in the sorted list matches the current rider
                    if (id == sorted.get(y).getKey().getRiderId()) {
                        // add the points for that stage and rider to the riders points list
                        ridersPoints[y] += sorted.get(y).getKey().getPointsInStage(currentStage, rank);
                    }
                }
            }
        }

        // return the riders points
        return ridersPoints;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int[] getRidersMountainPointsInRace(int raceId) throws IDNotRecognisedException {
        // find race in portal
        // throws IDNotRecognisedException
        Race race = findRace(raceId);

        // check if any of the results are empty
        // loop through all the stages in the race
        for (int i = 0; i < race.getStages().size(); i++) {
            // check if the results list for any stage is empty
            if (race.getStages().get(i).getResults().size() == 0) {
                // if empty, return an empty local time array
                return new int[0];
            }
        }

        // initalize a new array list of results to store all the results for the race
        ArrayList<Results> results = new ArrayList<>();

        // loop through each stage in the race
        for (int i = 0; i < race.getStages().size(); i++) {
            // loop through eahc reuslt in the stage
            for (int x = 0; x < race.getStages().get(i).getResults().size(); x++) {
                // add the result to the results list
                results.add(race.getStages().get(i).getResults().get(x));
            }
        }

        // initalize a hash map to pair together the riders and there times
        Map<Rider, LocalTime> timesMap = new HashMap<Rider, LocalTime>();

        // loop through each result in the race
        for (int i = 0; i < results.size(); i++) {
            // store the current rider
            Rider currentRider = results.get(i).getRider();

            // check if the hash map allready contains an entry for the current rider
            if (timesMap.containsKey(currentRider)) {
                // if the hash map contains an enrty for the rider,
                // add the current results adjusted elapsed time to the value for the rider
                // allready in the hash map

                //calculate the adjusted elapsed time for the current result in nano seconds
                long nanos = results.get(i).calculateAdjustedElapsedTime().toNanoOfDay();

                // add the current results nano seconds to the riders current result
                LocalTime newTime = timesMap.get(currentRider).plusNanos(nanos);

                // replace the old time with the new time
                timesMap.replace(currentRider, newTime);
            } else {
                // if rider does not allreayd have a map entry
                // add them into the hashmap paired with there time
                timesMap.put(currentRider, results.get(i).calculateAdjustedElapsedTime());
            }
        }

        // create an array list of all the map entrys
        ArrayList<Map.Entry<Rider, LocalTime>> sorted = new ArrayList<>(timesMap.entrySet());

        // sort the map using the custom comparitor whitch compares map entrys based of
        // the differance between there LocalTimes
        sorted.sort(new ResultsAdjustedElapsedTimeCompatiror());

        // init a new array the size of all the riders in the race
        int ridersPoints[] = new int[sorted.size()];

        // loop through all the riders in the race
        for (int i = 0; i < ridersPoints.length; i++) {
            // init there inital points to zero
            ridersPoints[i] = 0;
        }

        // for each rider, find the total points in all stages
        // loop through each stage in the race
        for (int i = 0; i < race.getStages().size(); i++) {
            // store the current stage
            Stage currentStage = race.getStages().get(i);

            // loop through each rider in the sorted array
            for (int y = 0; y < sorted.size(); y++) {
                // add the rider at y's mountain points to the points array
                // points calculated using Rider.getMountainPointsInStage()
                ridersPoints[y] += sorted.get(y).getKey().getMountainPointsInStage(currentStage);
            }
        }

        // return the riders mouinain points ordered by adjusted elapsed time
        return ridersPoints;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int[] getRidersPointClassificationRank(int raceId) throws IDNotRecognisedException {
        // get riders ranks for race
        int riderIds[] = getRidersGeneralClassificationRank(raceId);

        // get riders points for race
        int riderPoints[] = getRidersPointsInRace(raceId);

        // create a map mapping rider to there points
        Map<Rider, Integer> pointsMap = new HashMap<Rider, Integer>();

        // loop through each rider in the race
        for (int i = 0; i < riderIds.length; i++) {
            // store the current rider
            Rider currentRider = findRider(riderIds[i]);

            // add the current rider to the hash map with there points in race
            pointsMap.put(currentRider, riderPoints[i]);
        }

        // create an array list of map entrys
        ArrayList<Map.Entry<Rider, Integer>> sorted = new ArrayList<>(pointsMap.entrySet());

        // sort the array list by comparing the points
        // points are compared using a custom comparitor and annonamys function which returs the differance
        // between the current result and the result after
        // p2 - p1 in order to get reverce order (p1 - p2 for acending order)
        sorted.sort(Comparator.comparing(Map.Entry<Rider, Integer>::getValue, (p1, p2) -> {
            return p2 - p1;
        }));

        // create an array to store the ids of all the riders
        int sortedIds[] = new int[riderIds.length];

        // loop through all the riders in the sorted array
        for (int i = 0; i < sortedIds.length; i++) {
            // store the riders id in the array matching the index of the sorted array
            sortedIds[i] = sorted.get(i).getKey().getRiderId();
        }

        // return the sorted list (by points aquired) of rider ids
        return sortedIds;
```

```java
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int[] getRidersMountainPointClassificationRank(int raceId) throws IDNotRecognisedException {
        // get riders ranks for race
        int riderIds[] = getRidersGeneralClassificationRank(raceId);

        // get riders mountain points for race
        int riderPoints[] = getRidersMountainPointsInRace(raceId);

        // create a map mapping rider to there points
        Map<Rider, Integer> pointsMap = new HashMap<Rider, Integer>();

        // loop through each rider in the race
        for (int i = 0; i < riderIds.length; i++) {
            // store the current rider
            Rider currentRider = findRider(riderIds[i]);

            // add the current rider to the hash map with there points in race
            pointsMap.put(currentRider, riderPoints[i]);
        }

        // create an array list of map entrys
        ArrayList<Map.Entry<Rider, Integer>> sorted = new ArrayList<>(pointsMap.entrySet());

        // sort the array list by comparing the points
        // points are compared using a custom comparitor and annonamys function which returs the differance
        // between the current result and the result after
        // p2 - p1 in order to get reverce order (p1 - p2 for acending order)
        sorted.sort(Comparator.comparing(Map.Entry<Rider, Integer>::getValue, (p1, p2) -> {
            return p2 - p1;
        }));

        // create an array to store the ids of all the riders
        int sortedIds[] = new int[riderIds.length];

        // loop through all the riders in the sorted array
        for (int i = 0; i < sortedIds.length; i++) {
            // store the riders id in the array matching the index of the sorted array
            sortedIds[i] = sorted.get(i).getKey().getRiderId();
        }

        // return the sorted list (by points aquired) of rider ids
        return sortedIds;
    }

}
```

2 ClimbSegment.java

```java
package cycling;
```

```java

/**
 * Class for ClimbSegemt extents {@link Segment}. Stores additional details requted if the segment is a
 * climbing segment.
 *
 * @author Ethan Hofton
 * @author Jon Tao
 * @version 1.0
 *
 */
public class ClimbSegment extends Segment {

    private Double averageGradient;
    private Double length;

    /**
     * The constructor for climb segment.
     *
     * @param stage the stage the segment is in
     * @param location the location of the segment within the stage
     * @param type the type of segment
     * @param averageGradient average gradient of segment
     * @param length length of segment
     */
    public ClimbSegment(Stage stage, double location, SegmentType type, Double averageGradient, Double
         length) {
        // call Segment custructor
        super(stage, location, type);

        // set gradient and length
        this.averageGradient = averageGradient;
        this.length = length;
    }

    /**
     * Getter for {@code this.averageGradient}
     *
     * @return the average gradient
     */
    public Double getAverageGradient() {
        return this.averageGradient;
    }

    /**
     * Getter for {@code this.length}
     *
     * @return the average gradient
     */
    public Double getLength() {
        return this.length;
    }

    /**
     * Returns if the segment is a climb segment.
     * Overrides {@link cycling.Segment.isClimb}
     *
     * @return wether the segment is a climb or not
     */
    @Override
    boolean isClimb() {
        return true;
    }

    /**
     * Returns if the segment is a sprint segment.
     * Overrides {@link cycling.Segment.isSprint}
     *
     * @return wether the segment is a sprint or not
     */
    @Override
    boolean isSprint() {
        return false;
    }

    /**
     * Calculates the points mountain points for the segment
     * Data from Figure 2 in coursework spesification
     *
     * @param rank the rank of the rider
     * @return the points the rider gets for the given rank
     */
    public int mountainPoints(int rank) {
        // switch the segment type
        // and return the points aquered for that type of segment given the riders rank
        switch (type) {
            case C1:
                // return points for C1
                return pointsFor1C(rank);
            case C2:
                // return points for C2
                return pointsFor2C(rank);
            case C3:
                // return points for C3
                return pointsFor3C(rank);
            case C4:
                // return points for C4
                return pointsFor4C(rank);
            case HC:
                // return points for HC
                return pointsForHC(rank);
            default:
                // if segment type is not as above
                // no points will be aquered so return 0
                return 0;
        }
    }

    /**
     * Calculates the points for HC Mountain segment
     * Data from Figure 2 in coursework spesification
```

```java
     *
     * @param rank the rank of the rider
     * @return the points the rider gets for the given rank
     */
    static public int pointsForHC(int rank) {
        // swicth the rank and return the points aquered for HC
        // data is from Figure 2 in coursework spec
        switch (rank) {
        case 1:
            return 20;
        case 2:
            return 15;
        case 3:
            return 12;
        case 4:
            return 10;
        case 5:
            return 8;
        case 6:
            return 6;
        case 7:
            return 4;
        case 8:
            return 2;
        default:
            return 0;
        }
    }

    /**
     * Calculates the points for 1C Mountain segment
     * Data from Figure 2 in coursework spesification
     *
     * @param rank the rank of the rider
     * @return the points the rider gets for the given rank
     */
    static public int pointsFor1C(int rank) {
        // swicth the rank and return the points aquered for C1
        // data is from Figure 2 in coursework spec
        switch (rank) {
        case 1:
            return 10;
        case 2:
            return 8;
        case 3:
            return 6;
        case 4:
            return 4;
        case 5:
            return 2;
        case 6:
            return 1;
        default:
            return 0;
        }
    }

    /**
     * Calculates the points for 2C Mountain segment
     * Data from Figure 2 in coursework spesification
     *
     * @param rank the rank of the rider
     * @return the points the rider gets for the given rank
     */
    static public int pointsFor2C(int rank) {
        // swicth the rank and return the points aquered for C2
        // data is from Figure 2 in coursework spec
        switch (rank) {
        case 1:
            return 5;
        case 2:
            return 3;
        case 3:
            return 2;
        case 4:
            return 1;
        default:
            return 0;
        }
    }

    /**
     * Calculates the points for 3C Mountain segment
     * Data from Figure 2 in coursework spesification
     *
     * @param rank the rank of the rider
     * @return the points the rider gets for the given rank
     */
    static public int pointsFor3C(int rank) {
        // swicth the rank and return the points aquered for C3
        // data is from Figure 2 in coursework spec
        switch (rank) {
        case 1:
            return 2;
        case 2:
            return 1;
        default:
            return 0;
        }
    }

    /**
     * Calculates the points for 4C Mountain segment
     * Data from Figure 2 in coursework spesification
     *
     * @param rank the rank of the rider
     * @return the points the rider gets for the given rank
     */
    static public int pointsFor4C(int rank) {
        // swicth the rank and return the points aquered for C4
        // data is from Figure 2 in coursework spec
        switch (rank) {
        case 1:
            return 1;
        default:
            return 0;
        }
    }
}
```

3 Race.java

```java
package cycling;

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

/**
 * Race class to store the race id and addtional details relevent
 * to the race
 *
 * @author Ethan Hofton
 * @author Jon Tao
 * @version 1.0
 */
public class Race implements Serializable {
    private static int raceCount = 0;

    private int raceId;
    private String name;
    private String description;
    private ArrayList<Stage> stages;

    /**
     * Race class constructor
     *
     * @param name the name of the race
     * @param description the description of the race
     */
    public Race(String name, String description) {
        // set the race id and increment the static race counter
        this.raceId = raceCount++;

        // set the rest of the class attributes
        this.name = name;
        this.description = description;

        // initalize stages array list
        this.stages = new ArrayList<>();
    }

    /**
     * getter for {@code this.raceId}
     *
     * @return the id of the race
     */
    public int getRaceId() {
        return raceId;
    }

    /**
     * getter for {@code this.name}
     *
     * @return the name of the race
     */
    public String getName() {
        return name;
    }

    /**
     * getter for {@code this.description}
     *
     * @return the description of the race
     */
    public String getDescription() {
        return description;
    }

    /**
     * getter for {@code this.stages}
     *
     * @return the list of stages in the race
     * @see cycling.Stage
     */
    public ArrayList<Stage> getStages() {
        return stages;
    }

    /**
     * adds a stage to the race
     *
     * @param stage the stage class to be added to the race
     * @see cycling.Stage
     */
    public void addStage(Stage stage) {
        // add stage to stages array list
        stages.add(stage);
    }

    /**
     * remove stage from race
     *
     * @param stage the stage class to be removed from the race
     * @throws IDNotRecognisedException if the stage is not in the race
     * @see cycling.Stage
     */
    public void removeStage(Stage stage) throws IDNotRecognisedException {
        // check if stages contains stage to remove
        if (!stages.contains(stage)) {
            // if stages array list does not contain a stage, throw IDNotRecognisedException
            throw new IDNotRecognisedException("stage does not exist in race with Id '"+raceId+"'");
        }
        // remove stage from stages array list
        stages.remove(stage);
    }

    /**
     * check if the race contains a given stage
     *
     * @param stage the stage to be checked
     * @return boolean wether the race contains the stage
     * @see cycling.Stage
     */
    public boolean containsStage(Stage stage) {
        // return weather stages contains array list
        return stages.contains(stage);
    }

    /**
     * Rest the static counter to set the ids
     */
    public static void resetCounter() {
        // reset static race counte to zero
        raceCount = 0;
    }
}
```

4 ResultsAdjustedElapsedTimeCompatiror.java

```java
package cycling;

import java.time.LocalTime;
import java.util.Comparator;
import java.util.Map;

/**
 * compatoror for results class compare by adjusted elasped time
 *
 * @author Ethan Hofton
 * @author Jon Tao
 * @version 1.0
 */
public class ResultsAdjustedElapsedTimeCompatiror implements Comparator<Map.Entry<Rider,LocalTime>> {
    /**
     * Compare 2 reuslts using {@code LocalTime.compareTo}
     *
     * @param result1 first result to compare
     * @param result2 second result to copmare
     * @return the value of result1 - result2
     */
    @Override
    public int compare(Map.Entry<Rider,LocalTime> result1, Map.Entry<Rider,LocalTime> result2) {
        // compare the value of each entry using LocalTime.compareTo
        return result1.getValue().compareTo(result2.getValue());
    }
}
```

5 ResultsElapsedTimeComparator.java

```java
package cycling;

import java.util.Comparator;

/**
 * compatoror for results class compare by elasped time
 *
 * @author Ethan Hofton
 * @author Jon Tao
 * @version 1.0
 */
public class ResultsElapsedTimeComparator implements Comparator<Results> {

    /**
     * Compare 2 reuslts using {@code LocalTime.compareTo}
     *
     * @param result1 first result to compare
     * @param result2 second result to copmare
     * @return the value of result1 - result2
     */
    @Override
    public int compare(Results result1, Results result2) {
        // compare 2 results by there adjusted elapsed time
        // using LocalTime.compareTo and Reuslt.calculateElapsedTime
        return result1.calculateElapsedTime().compareTo(result2.calculateElapsedTime());
    }
}
```

6 ResultsMountainTimeCompatoror.java

```java
package cycling;

import java.util.Comparator;

/**
 * compatoror for results class compare by elasped time
 *
 * @author Ethan Hofton
 * @author Jon Tao
 * @version 1.0
 */
public class ResultsMountainTimeCompatoror implements Comparator<Results> {

    private int pos;

    /**
     * Constructor for class
     *
     * @param pos the position the segment is in the checkpoint times
     */
    public ResultsMountainTimeCompatoror(int pos) {
        // set class attrivutes
        this.pos = pos;
    }

    /**
     * Compare 2 reuslts using {@code LocalTime.compareTo}
     *
     * @param result1 first result to compare
     * @param result2 second result to copmare
     * @return the value of result1 - result2
     */
    @Override
    public int compare(Results result1, Results result2) {
        // compare 2 results at a cetrain position using LocalTime.compareTo
        return result1.getTimes()[pos].compareTo(result2.getTimes()[pos]);
    }
}
```

7 ResultsSegmentTimeCompatitor.java

```java
package cycling;

import java.util.Comparator;

/**
 * Results class compatotor.
 * Used to compare 2 results based on the time to segment
 *
 * @author Ethan Hofton
 * @author Jon Tao
 * @version 1.0
 */
public class ResultsSegmentTimeCompatitor implements Comparator<Results> {

    private int pos;

    /**
     * Constructor for class
     *
     * @param pos the position the segment is in the checkpoint times
     */
    public ResultsSegmentTimeCompatitor(int pos) {
        // set class attrivutes
        this.pos = pos;
    }

    /**
     * Compare 2 reuslts using {@code LocalTime.compareTo}
     *
     * @param result1 first result to compare
     * @param result2 second result to copmare
     * @return the value of result1 - result2
     */
    @Override
    public int compare(Results result1, Results result2) {
        // compare 2 results at a cetrain position using LocalTime.compareTo and
             Result.calculateTimeToSegment
        return result1.calculateTimeToSegment(pos).compareTo(result2.calculateTimeToSegment(pos));
    }
}
```

8 Rider.java

```java
package cycling;

import java.io.Serializable;

/**
 * The rider class. Stores rider id and other data relevent to the rider
 *
 * @author Ethan Hofton
 * @author Jon Tao
 * @version 1.0
 */
public class Rider implements Serializable {

    private static int riderCount = 0;

    private int riderId;
    private String riderName;
    private int riderYearOfBirth;
    private Team riderTeam;

    /**
     * The rider constructor
     *
     * @param team the team the rider belongs to
     * @param riderName the name of the rider
     * @param riderYearOfBirth the year of bith of the rider
     * @see cycling.Team
     */
    public Rider(Team team, String riderName, int riderYearOfBirth) {
        // set rider id and increment rider count
        this.riderId = riderCount++;

        // set rider class attributes
        this.riderName = riderName;
        this.riderYearOfBirth = riderYearOfBirth;
        this.riderTeam = team;
    }

    /**
     * Getter for {@code this.riderId}
     *
     * @return the id of the rider
     */
    public int getRiderId() {
        return riderId;
    }

    /**
     * Getter for {@code this.riderTeam}
     *
     * @return the team of the rider
     * @see cycling.Team
     */
    public Team getTeam() {
        return riderTeam;
    }

    /**
     * Getter for {@code this.riderName}
     *
     * @return the name of the rider
     */
    public String getRiderName() {
        return riderName;
    }

    /**
     * Getter for {@code this.riderYearOfBirth}
     *
     * @return the year of birth of the rider
     */
    public int getRiderYearOfBirth() {
        return riderYearOfBirth;
    }

    /**
     * sums the rank points and sprint points for a rider and given stage
     *
     * @param stage the stage the rider accumlated points for
     * @param rank the rank the rider got
     * @return the total points accumlated for the given stage
     */
    public int getPointsInStage(Stage stage, int rank) {
        // initalize the points
        int points = 0;

        // add the rank points
        points += stage.pointsForRank(rank);

        // add the intermidiate sprint points
        points += stage.pointsForIntermediateSprints(this);

        return points;
    }

    /**
     * returns the mountain points for that rider in the given stage
```

```java
     *
     * @param stage the stage the rider accumlated points for
     * @return the total points accumlated for the given stage
     */
    public int getMountainPointsInStage(Stage stage) {
        // return the mountain points for this rider
        return stage.pointsForMountainClassification(this);
    }

    /**
     * Rest the static counter to set the ids
     */
    public static void resetCounter() {
        // reset static rider counter
        riderCount = 0;
    }
}
```

9 Segment.java

```java
package cycling;

import java.io.Serializable;

/**
 * Segment class. Stores information common to both
 * climb segemnts and sprint segments
 *
 * @author Ethan Hofton
 * @author Jon Tao
 * @version 1.0
 */
public class Segment implements Serializable {
    protected static int segmentCount;
    protected int segmentId;
    protected Stage stage;
    protected double location;
    protected SegmentType type;

    /**
     * Segment constructor
     *
     * @param stage the stage the segment belongs to
     * @param location the location of the segment within the stage
     * @param type the type of the segment
     * @see cycling.Stage
     * @see cycling.SegmentType
     */
    public Segment(Stage stage, double location, SegmentType type) {
        // set segment id and increment segment count
        this.segmentId = segmentCount++;

        // set the class attributes
        this.stage = stage;
        this.location = location;
        this.type = type;
    }

    /**
     * Getter for {@code this.segmentId}
     *
     * @return the id for the segment
     */
    public int getSegmentId() {
        return segmentId;
    }

    /**
     * Getter for {@code this.stage}
     *
     * @return the stage the segment belongs to
     * @see cycling.Stage
     */
    public Stage getStage() {
        return stage;
    }

    /**
     * Getter for {@code this.location}
     *
     * @return location of the segment within the stage
     */
    public double getLocation() {
        return location;
    }

    /**
     * Getter for {@code this.type}
     *
     * @return the type of segment
     * @see cycling.SegmentType
     */
    public SegmentType getType() {
        return type;
    }

    /**
     * Check wither the segment is a climb or not
     *
     * @return boolean of wether the segment is a climb or not
     */
    boolean isClimb() {
        return !isSprint();
    }

    /**
     * Check wither the segment is a sprint or not
     *
     * @return boolean of wether the segment is a sprint or not
```

```java
     */
    boolean isSprint() {
        return type == SegmentType.SPRINT;
    }

    /**
     * Rest the static counter to set the ids
     */
    public static void resetCounter() {
        // reset the static segment counter
        segmentCount = 0;
    }
}
```

10 SprintSegment.java

```java
package cycling;

/**
 * extends {@link cycling.Segment}
 * A special case of {@code Segment} where the type is {@code SegmentType.SPRINT}
 *
 * @author Ethan Hofton
 * @author Jon Tao
 * @version 1.0
 * @see cycling.Segment
 *
 */
public class SprintSegment extends Segment {

    /**
     * SprintSegment Constructor. call super construor explisitly passing {@code type} as {@code
         SegmentType.SPRINT}
     *
     * @param stage the stage the segment belongs to
     * @param location the location of the segment in the stage
     * @see cycling.Stage
     */
    public SprintSegment(Stage stage, double location) {
        // call segment constructor
        super(stage, location, SegmentType.SPRINT);
    }

    /**
     * Override of {@link cycling.Segment.isClimb} where the value is explisitly defined
     *
     * @return false
     * @see cycling.Segment.isClimb
     */
    @Override
    boolean isClimb() {
        return false;
    }

    /**
     * Override of {@link cycling.Segment.isSprint} where the value is explisitly defined
     *
     * @return true
     * @see cycling.Segment.isSprint
     */
    @Override
    boolean isSprint() {
        return true;
    }
}
```

11 Stage.java

```java
package cycling;

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

/**
 * Stage class to store stage id and data related to stage
 *
 * @author Ethan Hofton
 * @author Jon Tao
 * @version 1.0
 */
public class Stage implements Serializable {
    private static int stageCount = 0;
    private int stageId;
    private Race race;
    private String stageName;
    private String description;
    private double length; // in KM
    private LocalDateTime startTime;
    private StageType type;
    private StageState stageState;

    private ArrayList<Segment> segments;
    private ArrayList<Results> results;

    /**
     * Stage contrustor
     *
     * @param race the race the stage belongs to
     * @param stageName the name of the stage
     * @param description the stage description
     * @param length the length of the stage
     * @param startTime the time the stage will begin
     * @param type the type of stage
     * @see cycling.Race
     * @see cycling.StageType
     */
    public Stage(Race race, String stageName, String description, double length, LocalDateTime startTime,
         StageType type) {
        // set the stage id and increment the static stage counter
        this.stageId = stageCount++;

        // set class attributes
        this.race = race;
        this.stageName = stageName;
        this.description = description;
        this.length = length;
        this.startTime = startTime;
        this.type = type;
        this.stageState = StageState.STAGE_PREPERATION;

        // initalize the class array lists
        this.segments = new ArrayList<>();
        this.results = new ArrayList<>();
    }

    /**
     * Getter for {@code this.stageId}
     *
     * @return the id of the stage
     */
    public int getStageId() {
        return stageId;
    }

    /**
     * Getter for {@code this.race}
     *
     * @return the race the stage belongs to
     * @see cycling.Race
     */
    public Race getRace() {
        return race;
    }

    /**
     * Getter for {@code this.stageName}
     *
     * @return the name of the stage
     */
    public String getStageName() {
        return stageName;
    }

    /**
     * Getter for {@code this.description}
     *
     * @return the description of the stage
     */
    public String getDescriptiom() {
        return description;
    }

    /**
     * Getter for {@code this.length}
     *
     * @return the length of the stage
     */
    public double getLength() {
        return length;
    }

    /**
     * Getter for {@code this.startTime}
     *
     * @return the time the stage will begin
     */
    public LocalDateTime getStartTime() {
        return startTime;
    }

    /**
     * Getter for {@code this.type}
     *
     * @return the type of the stage
     */
    public StageType getType() {
        return type;
    }

    /**
     * Getter for {@code this.segments}
     *
     * @return a list of the segments the stage has
     * @see cycling.Segment
     */
    public ArrayList<Segment> getSegments() {
        return this.segments;
    }

    /**
     * Add a segment to the stage
     *
     * @param segment the segment to be added to the stage
     * @see cycling.Segment
     */
    public void addSegment(Segment segment) {
        // add segment to segment array list
        this.segments.add(segment);
    }

    /**
     * Remove a segment from the stage
     *
     * @param segment the segment to be removed from the stage
     * @see cycling.Segment
     */
    public void removeSegment(Segment segment) {
        // remove segment from segment array list
        this.segments.remove(segment);
    }

    /**
     * Getter for {@code this.stageState}
     *
     * @return the state of the stage
     * @see cycling.StageState
     */
    public StageState getStageState() {
        return this.stageState;
    }

    /**
     * Chage the state of the stage to waiting for results.
     * Function can only be called once
     *
     * @throws InvalidStageStateException if the function is called twice
     */
    public void concludeStagePreparation() throws InvalidStageStateException {
        // conculde stage preperation
        // if stage has allready been conculded throw error
        // check if stage type is allready waiting for results
        if (this.stageState == StageState.WAITING_FOR_RESULTS) {
            // throw InvalidStageStateException if stage state is allready waitng for results
            throw new InvalidStageStateException("Stage is allready waiting for results");
        }

        // set the stage state to waiting for resutls
        this.stageState = StageState.WAITING_FOR_RESULTS;
    }

    /**
     * add result to stage
     *
     * @param result the result to be added
     * @see cycling.Results
     */
    public void addResults(Results result) {
        // add result to resutls array list
        results.add(result);
    }

    /**
     * getter for {@code this.results}
     *
     * @return a list of results the stage contains
     * @see cycling.Results
     */
    public ArrayList<Results> getResults() {
        return results;
    }

    /**
     * remove result from stage
     *
     * @param result result to be removed
     * @throws IDNotRecognisedException if the result is not in the race
     * @see cycling.Results
     */
    public void removeResults(Results result) throws IDNotRecognisedException {
        // remove result from result array list
        // check if result array contains result
        if (!results.contains(result)) {
            // if the result array does not contain result, throw an IDNotRecognisedException
            throw new IDNotRecognisedException("result does not exist in race with Id '"+stageId+"'");
        }
        // remove result
        results.remove(result);
    }

    /**
     * Calculate the number of points for position in stage.
     * Segments are not considered in this funciton
     *
     * @param rank position rider finished in segment
     * @return points the rider gained for finishing position in stage
     */
    public int pointsForRank(int rank) {

        // return the points aquired for a riders given rank
        // switch the stage type and return the appriotiate points based on the rank and stage type
        switch (this.type) {
            case FLAT:
                return pointsForFlat(rank);
            case HIGH_MOUNTAIN:
                return pointsForHMTTIT(rank);
            case MEDIUM_MOUNTAIN:
                return pointsForMediumMountain(rank);
            case TT:
                return pointsForHMTTIT(rank);
            default:
                // if the stage type is not as above, no points were aquered and return zero
                return 0;
        }
    }

    /**
     * calculate the points for the intermiedete sprints in stage for a given rider.
     * Not including mountain points
     *
     * @param rider rider to calulcate points for
     * @return the points the rider accumulated over the stage
     */
    public int pointsForIntermediateSprints(Rider rider) {
        // initalize points to zero
        int points = 0;

        // loop through all the segments in the stage
        for (int i = 0; i < segments.size(); i++) {
            // check if the segment is a sprint segment
            if (segments.get(i).isSprint()) {
                // create an array for all the results
                Results[] rankedResults = new Results[getResults().size()];

                // loop through all the results in the stage
                for (int x = 0; x < rankedResults.length; x++) {
                    // add the result to the results array
                    rankedResults[x] = getResults().get(x);
                }

                // sort the results array based on the elapsed time to the point
                // sort using custom comparitor ResultsSegmentTimeCompatitor
                Arrays.sort(rankedResults, new ResultsSegmentTimeCompatitor(i+1));

                // loop through all the RANKED results
                for (int x = 0; x < rankedResults.length; x++) {
                    // if the result belongs to the rider
                    if (rankedResults[x].getRider() == rider) {
                        // add the intermediat points to the sum
                        points += pointsForHMTTIT(x+1);
                        continue;
                    }
                }
            }
        }

        // return the points aquered
        return points;
    }

    /**
     * Calculate the points for the mountain segments
     *
     * @param rider the rider to calculate the points for
     * @return the points the rider accumulated over the stage
     */
    public int pointsForMountainClassification(Rider rider) {

        // initalize points to zero
        int points = 0;

        // loop through all the segments in the stage
        for (int i = 0; i < segments.size(); i++) {
            // check if the segment is a climb
            if (segments.get(i).isClimb()) {
                // if the segment is a climb, it is safe to upcase the segment to a climbsegment
                ClimbSegment segment = (ClimbSegment)segments.get(i);

                // create a new array to store all the results in the stage
                Results[] rankedResults = new Results[getResults().size()];

                // loop through each result in the stage
                for (int x = 0; x < rankedResults.length; x++) {
                    // add the result to the list of results
                    rankedResults[x] = getResults().get(x);
                }

                // sore the ranked results using custom compatiror ResultsMountainTimeCompatoror to sort
                    based of
                // of the time at which the riders reached the segmenent finish
                Arrays.sort(rankedResults, new ResultsMountainTimeCompatoror(i+1));

                // loop through all the ranked results
                for (int x = 0; x < rankedResults.length; x++) {
                    // if the result belongs to the rider
                    if (rankedResults[x].getRider() == rider) {
                        // add the mountian points for that segment to the riders sum
                        points += segment.mountainPoints(x+1);
                        continue;
                    }
                }
            }
        }

        // return the points aquered
        return points;
    }

    /**
     * Calculates the points for flat finish stage
     * Data from Figure 1 in coursework spesification
     *
     * @param rank the rank of the rider
     * @return the points the rider gets for the given rank
     */
    static public int pointsForFlat(int rank) {
        // return the points aquered for the rank and if the stage type is flat
        // add data is taken from Figure 1 in coursework spec
        switch (rank) {
        case 1:
            return 50;
        case 2:
            return 30;
        case 3:
            return 20;
        case 4:
            return 18;
        case 5:
            return 16;
        case 6:
            return 14;
        case 7:
            return 12;
        case 8:
            return 10;
        case 9:
            return 8;
        case 10:
            return 7;
        case 11:
            return 6;
        case 12:
            return 5;
        case 13:
            return 4;
        case 14:
            return 3;
        case 15:
            return 2;
        default:
            return 0;
        }
    }

    /**
     * Calculates the points for Medium Mountain finish stage
     * Data from Figure 1 in coursework spesification
     *
     * @param rank the rank of the rider
     * @return the points the rider gets for the given rank
     */
    static public int pointsForMediumMountain(int rank) {
        // return the points aquered for the rank and if the stage type is medium
        // add data is taken from Figure 1 in coursework spec
        switch (rank) {
        case 1:
            return 30;
        case 2:
            return 25;
        case 3:
            return 22;
        case 4:
            return 19;
        case 5:
            return 17;
        case 6:
            return 15;
        case 7:
            return 13;
        case 8:
            return 11;
        case 9:
            return 9;
        case 10:
            return 7;
        case 11:
            return 6;
        case 12:
            return 5;
        case 13:
            return 4;
        case 14:
```

```java
            return 3;
        case 15:
            return 2;
        default:
            return 0;
        }
    }

    /**
     * Calculates the points for High Mountain, Time Trail, Individual Trial stage
     * Data from Figure 1 in coursework spesification
     *
     * @param rank the rank of the rider
     * @return the points the rider gets for the given rank
     */
    static public int pointsForHMTTIT(int rank) {
        // return the points aquered for the rank and if the stage type is high mountain, time trial or
            individual
        // trail
        // add data is taken from Figure 1 in coursework spec
        switch (rank) {
        case 1:
            return 20;
        case 2:
            return 17;
        case 3:
            return 15;
        case 4:
            return 13;
        case 5:
            return 11;
        case 6:
            return 10;
        case 7:
            return 9;
        case 8:
            return 8;
        case 9:
            return 7;
        case 10:
            return 6;
        case 11:
            return 5;
        case 12:
            return 4;
        case 13:
            return 3;
        case 14:
            return 2;
        case 15:
            return 1;
        default:
            return 0;
        }
    }

    /**
     * Rest the static counter to set the ids
     */
    public static void resetCounter() {
        // reset static counter of stage count
        stageCount = 0;
    }

}
```

12 StageState.java

```java
package cycling;

/**
 * This enum is used to represent the state of the stage.
 *
 * @author Ethan Hofton
 * @author Jon Tao
 * @version 1.0
 *
 */
public enum StageState {

    /**
    * Before the stage has concluded its preperation
    */
    STAGE_PREPERATION,

    /**
    * Stage is waiting for results to be entered
    */
    WAITING_FOR_RESULTS;
}
```

13 Team.java

```java
package cycling;

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


/**
 * Team class stores team ID and data relavent to team
 *
 * @author Ethan Hofton
 * @author Jon Tao
 * @version 1.0
 *
 */
public class Team implements Serializable {

    private static int teamCount = 0;

    private ArrayList<Rider> teamRiders;

    private int teamId;
    private String teamName;
    private String teamDescription;

    /**
     * Team construtor. initalises team ID
     *
     * @param teamName the name of the team
     * @param teamDescription the team description
     */
    Team(String teamName, String teamDescription) {
        // intialize team riders array list
        this.teamRiders = new ArrayList<>();

        // set team id and incriment static team counter
        this.teamId = teamCount++;

        // set class attributes
        this.teamName = teamName;
        this.teamDescription = teamDescription;
    }

    /**
     * Getter for {@code this.teamId}
     *
     * @return the id of the team
     */
    public int getTeamId() {
        return teamId;
    }

    /**
     * Getter for {@code this.teamName}
     *
     * @return the name of the team
     */
    public String getTeamName() {
        return teamName;
    }

    /**
     * Getter for {@code this.teamDescription}
     *
     * @return the desciption of the team
     */
    public String getTeamDescription() {
        return teamDescription;
    }

    /**
     * Getter for {@code this.teamRiders}
```

```java
     *
     * @return an array of the riders on the team
     * @see cycling.Rider
     */
    public ArrayList<Rider> getRiders() {
        return teamRiders;
    }

    /**
     * add rider to team
     *
     * @param newRider the rider to add to the team
     * @see cycling.Rider
     */
    public void addRider(Rider newRider) {
        // add rider to arraylist
        teamRiders.add(newRider);
    }

    /**
     * remove a rider from the team
     *
     * @param riderToRemove the rider to remove from the team
     * @throws IDNotRecognisedException if the rider is not in the team
     * @see cycling.Rider
     */
    public void removeRider(Rider riderToRemove) throws IDNotRecognisedException {
        // findRider throws IDNotRecognisedException
        // find rider position
        int riderPosition = findRider(riderToRemove);

        // remove rider at that index
        teamRiders.remove(riderPosition);

    }

    /**
     * return the index of the rider in {@code this.teamRiders}
     *
     * @param riderToFind the rider to find
     * @return the index of the rider in the rider array
     * @throws IDNotRecognisedException if the rider is not in the team
     * @see cycling.Rider
     */
    public int findRider(Rider riderToFind) throws IDNotRecognisedException {

        // loops through all team riders
        // checks id against given rider id
        // if ids match, return the position, id not throw exception
        for (int i = 0; i < teamRiders.size(); i++) {
            if (teamRiders.get(i).getRiderId() == riderToFind.getRiderId()) {
                return i;
            }
        }

        throw new IDNotRecognisedException("Rider id not found");
    }

    /**
     * Check if the rider is in the team
     *
     * @param riderToFind the rider to find
     * @return boolean wether the rider is in the team
     * @see cycling.Rider
     */
    public boolean containsRider(Rider riderToFind) {
        // try find the rider using findRider function
        // if the function throws an IDNotRecognisedException exception,
        // the rider does not exists and reutrn false,
        // otherwise return ture
        try {
            findRider(riderToFind);
        } catch (IDNotRecognisedException e) {
            return false;
        }

        return true;
    }

    /**
     * Rest the static counter to set the ids
     */
    public static void resetCounter() {
        // reset team counter to zero
        teamCount = 0;
    }
}
```