1 CyclingPortal.java

```java
package cycling;

import java.io.*;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.Objects;

/**
 * Compiling, functional implementor of the CyclingPortalInterface interface.
 *
 * @author Toby Slump and James Cracknell
 * @version 1.0
 * 03/2022
 */
public class CyclingPortal implements CyclingPortalInterface {
    private LinkedList<Race> listOfRaces = new LinkedList<>();
    private LinkedList<Team> listOfTeams = new LinkedList<>();
    private int[] staticAttributes = new int[5];
    @Override
    public int[] getRaceIds() {

        int[] RaceIDs = new int[listOfRaces.size()];

        for (int i = 0; i < listOfRaces.size(); i++) {
            RaceIDs[i] = listOfRaces.get(i).getRaceID();
        }

        return RaceIDs;

    }

    @Override
    public int createRace(String name, String description) throws IllegalNameException,
         InvalidNameException {
        // exception handling
        for (Race race : listOfRaces) {
            if (Objects.equals(race.getRaceName(), name)) { // if name already used
                throw new IllegalNameException
                        ("The given name has already been used on a race. Names must be unique./");
            }
        }

        if (name == null) { //if name null
            throw new InvalidNameException("Name must not be null");
        } else {
            if (name.length() > 30 || name.contains(" ")
                    || name.equals("")) { //if name does not meet specified criteria
                throw new InvalidNameException
                        ("Name must not be empty, must be a single word and cannot be over 30 characters");
            }
```

```
52          }
53
54
55          listOfRaces.add(new Race(name, description));
56          assert (listOfRaces.getLast().getRaceID() == Race.getNextRaceID())
57                  : "Race has not been created with correct ID";
58          return listOfRaces.getLast().getRaceID();
59
60      }
61
62      @Override
63      public String viewRaceDetails(int raceId) throws IDNotRecognisedException {
64          for (Race RaceObj : listOfRaces) {
65              if (RaceObj.getRaceID() == raceId) {
66                  return RaceObj.viewRaceDetails();
67              }
68          }
69
70          throw new IDNotRecognisedException("Couldn't find race with ID: " + raceId);
71      }
72
73      @Override
74      public void removeRaceById(int raceId) throws IDNotRecognisedException {
75          int prevListLength = listOfRaces.size();
76          for (int i = 0; i < listOfRaces.size(); i++){
77              Race RaceObj = listOfRaces.get((i));
78              if (RaceObj.getRaceID() == raceId){
79                listOfRaces.remove(RaceObj);
80              }
81          }
82
83          if (prevListLength == listOfRaces.size()){
84              throw new IDNotRecognisedException("Couldn't find race with ID: " + raceId);
85          }
86      }
87
88      @Override
89      public int getNumberOfStages(int raceId) throws IDNotRecognisedException {
90          for (Race RaceObj : listOfRaces) {
91              if (RaceObj.getRaceID() == raceId) {
92                  return RaceObj.getNumberOfStages();
93              }
94          }
95          throw new IDNotRecognisedException("Couldn't find race with ID: " + raceId);
96      }
97
98      @Override
99      public int addStageToRace(int raceId, String stageName, String description, double length,
            LocalDateTime startTime,
100                             StageType type)
101             throws IDNotRecognisedException, IllegalNameException, InvalidNameException,
                    InvalidLengthException {
102         // exception handling
103         for (Race race : listOfRaces) { //what should x be?
104             for (int j = 0; j < race.getStageIDs().length; j++) {
```

```
105                 if (Objects.equals(race.getStageNames()[j], stageName)) { // what should y be? // if name
                        already used
106                     throw new IllegalNameException
107                             ("The given name has already been used on a stage. Names must be unique.");
108                 }
109             }
110         }
111
112
113         if (stageName == null) { //if name null
114             throw new InvalidNameException("Name must not be null");
115         } else {
116             if (stageName.length() > 30 || stageName.contains(" ")
117                     || stageName.equals("")) { //if name does not meet specified criteria
118                 throw new InvalidNameException
119                         ("Name must not be empty, must be a single word and cannot be over 30 characters");
120             }
121         }
122
123         if (length < 5) { // Length has to be greater than 5kms
124             throw new InvalidLengthException("Length must be more than 5kms");
125         }
126
127
128         for (Race RaceObj : listOfRaces) {
129             if (RaceObj.getRaceID() == raceId) {
130                 return RaceObj.addStage(stageName, description, length,
131                         startTime, type);
132             }
133         }
134
135         throw new IDNotRecognisedException("No race found with ID: " + raceId);
136     }
137
138     @Override
139     public int[] getRaceStages(int raceId) throws IDNotRecognisedException {
140         for (Race RaceObj : listOfRaces) {
141             if (RaceObj.getRaceID() == raceId) {
142                 return RaceObj.getOrderedStageIDs();
143             }
144         }
145         throw new IDNotRecognisedException("Couldn't find race with ID: " + raceId);
146     }
147
148     @Override
149     public double getStageLength(int stageId) throws IDNotRecognisedException {
150         for (Race raceObj : listOfRaces) {
151             for (int j = 0; j < raceObj.getStageIDs().length; j++) {
152                 if (raceObj.getStageIDs()[j] == stageId) {
153                     return raceObj.getStageLength(stageId);
154                 }
155             }
156         }
157         throw new IDNotRecognisedException("Couldn't find stage with ID: " + stageId);
158     }
```

```
159
160     @Override
161     public void removeStageById(int stageId) throws IDNotRecognisedException {
162         boolean hasDeleted = false;
163
164         for (Race raceObj : listOfRaces) {
165             for (int j = 0; j < raceObj.getStageIDs().length; j++) {
166                 if (raceObj.getStageIDs()[j] == stageId) {
167                     raceObj.removeStageByID(stageId);
168                     hasDeleted = true;
169                 }
170             }
171         }
172
173         if (!hasDeleted){
174             throw new IDNotRecognisedException("Couldn't find stage with ID: " + stageId);
175         }
176     }
177
178     @Override
179     public int addCategorizedClimbToStage(int stageId, Double location, SegmentType type, Double
          averageGradient,
180                                           Double length) throws IDNotRecognisedException,
                                                  InvalidLocationException, InvalidStageStateException,
181             InvalidStageTypeException {
182         for (Race raceObj : listOfRaces) {
183             for (int j = 0; j < raceObj.getStageIDs().length; j++) {
184                 if (raceObj.getStageIDs()[j] == stageId) {
185                     if (Objects.equals(raceObj.getStageState(stageId), "waiting for results")) {
186                         throw new InvalidStageStateException
187                                 ("Stage is in invalid state: " + raceObj.getStageState(stageId));
188                     }
189                     if (0 > location || location > raceObj.getStageLength(stageId)) {
190                         throw new InvalidLocationException
191                                 ("The starting location of the climb is invalid. It must start and end within
                                      the stage.");
192                     }
193                     if (raceObj.getStageType(stageId) == StageType.TT) {
194                         throw new InvalidStageTypeException("Segments cannot be added to time trial stages");
195                     }
196                     return raceObj.addClimbToStage(stageId, location, type,
197                             averageGradient, length);
198                 }
199             }
200         }
201
202         throw new IDNotRecognisedException("No stage found with ID: " + stageId);
203     }
204
205     @Override
206     public int addIntermediateSprintToStage(int stageId, double location) throws IDNotRecognisedException,
207             InvalidLocationException, InvalidStageStateException, InvalidStageTypeException {
208         for (Race raceObj : listOfRaces) {
209             for (int j = 0; j < raceObj.getStageIDs().length; j++) {
210                 if (raceObj.getStageIDs()[j] == stageId) {
```

```
                if (raceObj.getStageState(stageId).equals("waiting for results")) {
                    throw new InvalidStageStateException
                            ("Stage is in invalid state: " + raceObj.getStageState(stageId));
                }
                if (0 > location || location > raceObj.getStageLength(stageId)) {
                    throw new InvalidLocationException
                            ("The starting location of the climb is invalid. It must start within the
                                stage.");
                }
                if (raceObj.getStageType(stageId) == StageType.TT) {
                    throw new InvalidStageTypeException("Segments cannot be added to time trial stages");
                }
                return raceObj.addSprintToStage(stageId, location);
            }
        }
    }

    throw new IDNotRecognisedException("No stage found with ID: " + stageId);
}

@Override
public void removeSegment(int segmentId) throws IDNotRecognisedException, InvalidStageStateException {
    boolean hasDeleted = false;

    for (Race raceObj : listOfRaces) {
        for (int j = 0; j < raceObj.getStageIDs().length; j++) {
            for (int k = 0; k < raceObj.getSegmentIds(raceObj.getStageIDs()[j]).length; k++) {
                if (raceObj.getSegmentIds(raceObj.getStageIDs()[j])[k] == segmentId) {
                    if (raceObj.getStageState(raceObj.getStageIDs()[j]).equals("waiting for results")) {
                        throw new InvalidStageStateException
                                ("Stage is in invalid state: " +
                                    (raceObj.getStageState(raceObj.getStageIDs()[j])));
                    }
                    raceObj.removeSegmentById(j, segmentId);
                    hasDeleted = true;
                }
            }
        }
    }

    if (!hasDeleted) {
        throw new IDNotRecognisedException("Couldn't find segment with ID: " + segmentId);
    }
}

@Override
public void concludeStagePreparation(int stageId) throws IDNotRecognisedException,
     InvalidStageStateException {
    boolean hasConcluded = false;

    for (Race raceObj : listOfRaces) {
        for (int j = 0; j < raceObj.getNumberOfStages(); j++) {
            if (raceObj.getStageIDs()[j] == stageId) {
                if (Objects.equals(raceObj.getStageState(stageId), "waiting for results")) {
                    throw new InvalidStageStateException
```

```
263                             ("Stage is in invalid state: " + raceObj.getStageState(stageId));
264                 }
265                 raceObj.concludeStatePreparation(stageId);
266                 hasConcluded = true;
267             }
268         }
269     }
270
271     if (!hasConcluded){
272         throw new IDNotRecognisedException("No stage found with Id: " + stageId);
273     }
274
275 }
276
277 @Override
278 public int[] getStageSegments(int stageId) throws IDNotRecognisedException {
279     for (Race raceObj : listOfRaces) {
280         for (int j = 0; j < raceObj.getStageIDs().length; j++) {
281             if (raceObj.getStageIDs()[j] == stageId) {
282                 return raceObj.getSegmentIds(stageId);
283             }
284         }
285     }
286
287     throw new IDNotRecognisedException("No stage found with ID: " + stageId);
288 }
289
290 @Override
291 public int createTeam(String name, String description) throws IllegalNameException,
      InvalidNameException {
292     for (Race race : listOfRaces) {
293         if (Objects.equals(race.getRaceName(), name)) { // if name already used
294             throw new IllegalNameException("The given name has already been used on a race. Names must be
                  unique./");
295         }
296     }
297
298     if (name == null) { //if name null
299         throw new InvalidNameException("Name must not be null");
300     } else {
301         if (name.length() > 30 || name.contains(" ") || name.equals("")) { //if name does not meet
               specified criteria
302             throw new InvalidNameException
303                     ("Name must not be empty, must be a single word and cannot be over 30 characters");
304         }
305     }
306
307     listOfTeams.add(new Team(name, description));
308     assert (listOfTeams.getLast().getTeamID() == Team.getNextTeamID())
309             : "Team was not created with correct ID";
310     return listOfTeams.getLast().getTeamID();
311
312 }
313
314 @Override
```

```
315     public void removeTeam(int teamId) throws IDNotRecognisedException {
316         boolean hasRemoved = false;
317         for (int i = 0; i < listOfTeams.size(); i++){
318             Team teamObj = listOfTeams.get(i);
319 
320             if (teamObj.getTeamID() == teamId){
321                 listOfTeams.remove(teamObj);
322                 hasRemoved = true;
323             }
324         }
325         if (!hasRemoved) {
326             throw new IDNotRecognisedException("No team found with ID: " + teamId);
327         }
328     }
329 
330     @Override
331     public int[] getTeams() {
332         int[] teamIds = new int[listOfTeams.size()];
333 
334         for (int i = 0; i < listOfTeams.size(); i++){
335             teamIds[i] = listOfTeams.get(i).getTeamID();
336         }
337 
338         return teamIds;
339     }
340 
341     @Override
342     public int[] getTeamRiders(int teamId) throws IDNotRecognisedException {
343         for (Team team : listOfTeams) {
344             if (team.getTeamID() == teamId) {
345                 return team.getRiderIds();
346             }
347         }
348 
349         throw new IDNotRecognisedException("No team found with Id:" + teamId);
350     }
351 
352     @Override
353     public int createRider(int teamID, String name, int yearOfBirth)
354             throws IDNotRecognisedException, IllegalArgumentException {
355         if (yearOfBirth < 1900) {
356             throw new IllegalArgumentException("Year of birth is less than 1900.");
357         } else if (name == null) {
358             throw new IllegalArgumentException("Name cannot be null.");
359         }
360 
361 
362         for (Team team : listOfTeams) {
363             if (team.getTeamID() == teamID) {
364                 team.addRider(name, yearOfBirth);
365                 assert (team.getNewRiderID() == Rider.getNextRiderID())
366                         : "Rider was not created with correct ID";
367                 return team.getNewRiderID();
368             }
369         }
```

```
370
371         throw new IDNotRecognisedException("No team found with ID: " + teamID);
372     }
373
374     @Override
375     public void removeRider(int riderId) throws IDNotRecognisedException {
376         boolean hasRemoved = false;
377
378         for (Team team : listOfTeams) {
379             for (int j = 0; j < team.getRiderIds().length; j++) {
380                 if (team.getRiderIds()[j] == riderId) {
381                     team.removeRider(riderId);
382                     for (Race race : listOfRaces){
383                         race.deleteAllRiderResults(riderId);
384                     }
385                     hasRemoved = true;
386                 }
387             }
388         }
389
390         if (!hasRemoved){
391             throw new IDNotRecognisedException("No rider found with ID: " + riderId);
392         }
393     }
394
395     @Override
396     public void registerRiderResultsInStage(int stageId, int riderId, LocalTime... checkpoints)
397             throws IDNotRecognisedException, DuplicatedResultException, InvalidCheckpointsException,
398             InvalidStageStateException {
399         boolean hasRegistered = false;
400
401         for (Race race : listOfRaces) {
402             for (int j = 0; j < race.getNumberOfStages(); j++) {
403                 if (race.getStageIDs()[j] == stageId) {
404                     if (race.isRiderInResults(stageId, riderId)) {
405                         throw new DuplicatedResultException("Rider already has results");
406                     }
407
408                     if (checkpoints.length != race.getNumberOfSegmentsInStage(stageId) + 2) {
409                         throw new InvalidCheckpointsException("Invalid length of checkpoints");
410                     }
411                     if (!Objects.equals(race.getStageState(stageId), "waiting for results")) {
412                         throw new InvalidStageStateException
413                                 ("Stage is in invalid state: " + race.getStageState(stageId));
414                     }
415                     race.registerRiderResultsInStage(stageId, riderId, checkpoints);
416                     hasRegistered = true;
417                 }
418             }
419         }
420         if (!hasRegistered){
421             throw new IDNotRecognisedException("No rider found with ID: " + riderId);
422         }
423     }
424
```

```
425     @Override
426     public LocalTime[] getRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException {
427         for (Race race : listOfRaces) {
428             for (int j = 0; j < race.getNumberOfStages(); j++) {
429                 if (race.getStageIDs()[j] == stageId) {
430                     if (!race.isRiderInResults(stageId, riderId)) {
431                         throw new IDNotRecognisedException
432                                 ("No rider with ID: " + riderId + " results found in stage with ID: " +
                                         stageId);
433                     }
434                     LocalTime[] results = race.getRiderResults(stageId, riderId);
435                     LocalTime[] resultsWithElapsedTime = new LocalTime[results.length + 1];
436                     LocalTime elapsedTime =
437                             LocalTime.ofSecondOfDay(results[results.length - 1].toSecondOfDay()
438                                     - results[0].toSecondOfDay());
439                     System.arraycopy(results, 0, resultsWithElapsedTime, 0, results.length);
440                     resultsWithElapsedTime[results.length] = elapsedTime;
441                     return resultsWithElapsedTime;
442                 }
443             }
444         }
445
446         throw new IDNotRecognisedException("No stage found with ID: " + stageId);
447     }
448
449     @Override
450     public LocalTime getRiderAdjustedElapsedTimeInStage(int stageId, int riderId) throws
          IDNotRecognisedException {
451         for (Race race : listOfRaces) {
452             for (int j = 0; j < race.getNumberOfStages(); j++) {
453                 if (race.getStageIDs()[j] == stageId) {
454                     if (!race.isRiderInResults(stageId, riderId)) {
455                         throw new IDNotRecognisedException
456                                 ("No rider with ID: " + riderId + " results found in stage with ID: " +
                                         stageId);
457                     }
458                     return race.getRiderAdjustedElapsedResults(stageId, riderId);
459                 }
460             }
461         }
462
463         throw new IDNotRecognisedException("No stage found with ID: " + stageId);
464     }
465
466     @Override
467     public void deleteRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException {
468         boolean hasDeleted = false;
469
470         for (Race race : listOfRaces) {
471             for (int j = 0; j < race.getNumberOfStages(); j++) {
472                 if (race.getStageIDs()[j] == stageId) {
473                     if (!race.isRiderInResults(stageId, riderId)) {
474                         throw new IDNotRecognisedException
475                                 ("No rider with ID: " + riderId + " results found in stage with ID: " +
                                         stageId);
```

```
                }
                race.deleteRidersResults(stageId, riderId);
                hasDeleted = true;
            }
        }
    }

    if (!hasDeleted){
        throw new IDNotRecognisedException("No stage found with ID: " + riderId);
    }
}

@Override
public int[] getRidersRankInStage(int stageId) throws IDNotRecognisedException {
    for (Race race : listOfRaces) {
        for (int j = 0; j < race.getNumberOfStages(); j++) {
            if (race.getStageIDs()[j] == stageId) {
                return race.getRidersRankInStage(stageId);
            }
        }
    }

    throw new IDNotRecognisedException("No stage found with ID: " + stageId);
}

@Override
public LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId) throws IDNotRecognisedException {
    for (Race race : listOfRaces) {
        for (int j = 0; j < race.getNumberOfStages(); j++) {
            if (race.getStageIDs()[j] == stageId) {
                return race.getRankedAdjustedElapsedTimesInStage(stageId);
            }
        }
    }

    throw new IDNotRecognisedException("No stage found with ID: " + stageId);
}

@Override
public int[] getRidersPointsInStage(int stageId) throws IDNotRecognisedException {
    // Get the number of points obtained by each rider in a stage.
    int [] ridersRanked = getRidersRankInStage((stageId));
    boolean pointsCalculated = false;
    if (ridersRanked.length == 0){ // if ID is not recognised, return will be empty
        throw new IDNotRecognisedException("Stage ID not recognised");
    }
    int [] riderPoints = new int [ridersRanked.length]; // create new array of riders to store points
    for (Race race : listOfRaces) { // loop through races
        if (!pointsCalculated) { // if data for points has not been collected
            for (int j = 0; j < ridersRanked.length; j++) { // loops through riders
                riderPoints[j] = race.getPointsFromStage(stageId, j, ridersRanked[j]); // add current
                    riders point to array of rider points
                if (riderPoints[j] != 0) { // data has been returned
                    pointsCalculated = true; // will not get data again
                }
```

```
530                 }
531             }
532         }
533         return riderPoints;
534     }
535 
536     @Override
537     public int[] getRidersMountainPointsInStage(int stageId) throws IDNotRecognisedException {
538         //Get the number of mountain points obtained by each rider in a stage.
539         int [] ridersRanked = getRidersRankInStage((stageId));
540         boolean pointsCalculated = false;
541         if (ridersRanked.length == 0){ // if ID is not recognised, return will be empty
542             throw new IDNotRecognisedException("Stage ID not recognised");
543         }
544         int [] riderPoints = new int [ridersRanked.length]; // create new array of riders to store points
545         for (Race race : listOfRaces) { // loop through races
546             if (!pointsCalculated) { // if data for points has not been collected
547                 for (int j = 0; j < ridersRanked.length; j++) { // for loops through riders
548                     riderPoints[j] = race.getMountainPointsFromStage(stageId, ridersRanked[j]); // add
                            current riders point to array of rider points
549                     if (riderPoints[j] != 0) { // data has been returned
550                         pointsCalculated = true; // will not get data again
551                     }
552                 }
553             }
554         }
555 
556 
557         return riderPoints;
558     }
559 
560     @Override
561     public void eraseCyclingPortal() {
562         listOfRaces.clear();
563         listOfTeams.clear();
564         Race.setNextRaceID(0);
565         Stage.setNextStageID(0);
566         SprintSegment.setNextSegmentID(0);
567         Rider.setNextRiderID(0);
568         Team.setNextTeamID(0);
569     }
570 
571     @Override
572     public void saveCyclingPortal(String filename) throws IOException {
573         String fileNameUsed = filename;
574         if (!filename.endsWith(".ser")){
575             fileNameUsed = filename + ".ser";
576         }
577 
578         staticAttributes[0] = Race.getNextRaceID();
579         staticAttributes[1] = Stage.getNextStageID();
580         staticAttributes[2] = SprintSegment.getNextSegmentID();
581         staticAttributes[3] = Team.getNextTeamID();
582         staticAttributes[4] = Rider.getNextRiderID();
583 
```

```
584         try (ObjectOutputStream out = new ObjectOutputStream(new
585                 FileOutputStream(fileNameUsed))) {
586             out.writeObject(listOfRaces);
587             out.writeObject(listOfTeams);
588             out.writeObject(staticAttributes);
589         }catch (IOException e){
590             throw new IOException("Couldn't save objects");
591         }
592 
593     }
594 
595     @Override
596     public void loadCyclingPortal(String filename) throws IOException, ClassNotFoundException {
597         String fileNameUsed = filename;
598         if (!filename.endsWith(".ser")){
599             fileNameUsed = filename + ".ser";
600         }
601 
602         try (ObjectInputStream in = new ObjectInputStream(new
603                 FileInputStream(fileNameUsed))) {
604             Object obj = in.readObject();
605             if (obj instanceof LinkedList<?>)
606                 listOfRaces = (LinkedList<Race>) obj;//downcast safely
607             obj = in.readObject();
608             if (obj instanceof LinkedList<?>)
609                 listOfTeams = (LinkedList<Team>) obj;//downcast safely
610             obj = in.readObject();
611             if (obj instanceof int[])
612                 staticAttributes = (int[]) obj;
613             Race.setNextRaceID(staticAttributes[0]);
614             Stage.setNextStageID(staticAttributes[1]);
615             SprintSegment.setNextSegmentID(staticAttributes[2]);
616             Team.setNextTeamID(staticAttributes[3]);
617             Rider.setNextRiderID(staticAttributes[4]);
618         }catch (IOException e){
619             throw new IOException();
620         }catch (ClassNotFoundException e){
621             throw new ClassNotFoundException("");
622         }
623     }
624 
625     @Override
626     public void removeRaceByName(String name) throws NameNotRecognisedException {
627         boolean hasRemoved = false;
628 
629         for (int i = 0; i < listOfRaces.size(); i++){
630             if (Objects.equals(listOfRaces.get(i).getRaceName(), name)){
631                 listOfRaces.remove(i);
632                 hasRemoved = true;
633             }
634         }
635 
636         if (!hasRemoved){
637             throw new NameNotRecognisedException();
638         }
```

```
639
640     }
641
642     @Override
643     public LocalTime[] getGeneralClassificationTimesInRace(int raceId) throws IDNotRecognisedException {
644         for (Race race : listOfRaces) {
645             if (race.getRaceID() == raceId) {
646                 return race.getGeneralClassificationTimes();
647             }
648         }
649
650         throw new IDNotRecognisedException("No race found with ID: " + raceId);
651     }
652
653     @Override
654     public int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException {
655         int[] ridersPoints;
656         // adjusted lap time not just lap time
657         for (Race race : listOfRaces) {
658             if (race.getRaceID() == raceId) {
659                 ridersPoints = race.getRidersOverallPoints(); //get points for race
660                 return ridersPoints;
661             }
662         }
663         throw new IDNotRecognisedException("No race found with ID: " + raceId);
664
665     }
666
667     @Override
668     public int[] getRidersMountainPointsInRace(int raceId) throws IDNotRecognisedException {
669         int[] ridersPoints;
670         for (Race race : listOfRaces) {
671             if (race.getRaceID() == raceId) {
672                 ridersPoints = race.getRidersOverallMountainPoints(); //get points for race
673                 return ridersPoints;
674             }
675         }
676         throw new IDNotRecognisedException("No race found with ID: " + raceId);
677     }
678
679     @Override
680     public int[] getRidersGeneralClassificationRank(int raceId) throws IDNotRecognisedException {
681         for (Race race : listOfRaces) {
682             if (race.getRaceID() == raceId) {
683                 return race.getRidersGeneralClassificationRank();
684             }
685         }
686
687         throw new IDNotRecognisedException("No race found with ID: " + raceId);
688     }
689
690     @Override
691     public int[] getRidersPointClassificationRank(int raceId) throws IDNotRecognisedException {
692         int[] ridersPoints = getRidersPointsInRace(raceId); //gets list of riders points
693         int[] ridersRank = getRidersGeneralClassificationRank(raceId);
```

```
        if (ridersPoints.length == 0){ //if no riders results were returned then race ID is invalid
            throw new IDNotRecognisedException("Race ID is not recognised.");
        }
        int[][] combinedRiders = new int[ridersRank.length][2];
        //ridersRank[i] is the rider ID with the score in ridersPoints[i]
        for (int i = 0; i < ridersRank.length; i++) { // create 2d array to allow for sorting
            combinedRiders[i][0] = ridersRank[i];
            combinedRiders[i][1] = ridersPoints[i];
        }
      // sort based on time
        Arrays.sort(combinedRiders, (first, second) -> {
            if (first[1] < second[1]) return 1;
            else return -1;
        });
        for (int i = 0; i < ridersRank.length; i++) {
            ridersPoints[i] = combinedRiders[i][0]; // make array of sorted IDs
        }
        return ridersPoints;
    }

    @Override
    public int[] getRidersMountainPointClassificationRank(int raceId) throws IDNotRecognisedException {
        int[] ridersPoints = getRidersMountainPointsInRace(raceId); //gets list of riders points
        int[] ridersRank = getRidersGeneralClassificationRank(raceId);
        if (ridersPoints.length == 0){ //if no riders results were returned then race ID is invalid
            throw new IDNotRecognisedException("Race ID is not recognised.");
        }
        int[][] combinedRiders = new int[ridersRank.length][2];
        //ridersRank[i] is the rider ID with the score in ridersPoints[i]
        for (int i = 0; i < ridersRank.length; i++) { // create 2d array to allow for sorting
            combinedRiders[i][0] = ridersRank[i];
            combinedRiders[i][1] = ridersPoints[i];
        }
        // sort based on time
        Arrays.sort(combinedRiders, (first, second) -> {
            if (first[1] < second[1]) return 1;
            else return -1;
        });
        for (int i = 0; i < ridersRank.length; i++) {
            ridersPoints[i] = combinedRiders[i][0]; // make array of sorted IDs
        }

        return ridersPoints;
    }

}
```

2 Race.java

```
package cycling;

import java.io.Serializable;
import java.time.LocalDateTime;
import java.time.LocalTime;
```

```java
import java.util.*;

/**
 * The java class for race. Contains methods relating to races within the cycling app.
 *
 * @author Toby Slump and James Cracknell
 * @version 1.0
 * 03/2022
 */
public class Race implements Serializable {
    private int raceID;
    private static int nextRaceID = 0;
    private String name;
    private String description;
    private LinkedList<Stage> listOfStages = new LinkedList<>();


    /**
     * Race class constructor.
     *
     * @param name      Race's name.
     * @param Description Races' description.
     */
    public Race(String name, String Description){
        this.name = name;
        this.description = Description;
        raceID = ++nextRaceID;
    }

    /**
     * Gets race ID.
     *
     * @return Unique race ID.
     */
    public int getRaceID(){
        return raceID;
    }

    /**
     * Gets race name.
     *
     * @return Race name.
     */
    public String getRaceName(){
        return name;
    }

    /**
     * Gets race details.
     *
     * @return Race details.
     */
    public String viewRaceDetails(){
        return description;
    }
```

```
61
62      /**
63       * Gets the number of stages for a race.
64       *
65       * @return number of stages in queried race.
66       */
67      public int getNumberOfStages(){
68          return listOfStages.size();
69      }
70
71      /**
72       * Gets the value of NextRaceID.
73       * This shows the ID of the last race created and is used when giving new races their ID.
74       *
75       * @return The value of nextRaceID.
76       */
77      public static int getNextRaceID(){
78          return nextRaceID;
79      }
80
81      /**
82       * Sets the value of nextRaceID to a specified value.
83       *
84       * @param nextRaceId The new value of nextRaceID.
85       */
86      public static void setNextRaceID(int nextRaceId){
87          nextRaceID = nextRaceId;
88      }
89
90      /**
91       * Gets the number of segments in a stage.
92       *
93       * @param stageID The ID of the stage being queried.
94       * @return The number of segments in the given stage.
95       */
96      public int getNumberOfSegmentsInStage(int stageID){
97          for (Stage stage : listOfStages) {
98              if (stage.getID() == stageID) {
99                  return stage.getSegmentsIds().length;
100             }
101         }
102         return 0;
103     }
104
105     /**
106      * Creates a new stage and adds it to this race.
107      *
108      * @param stageName      An identifier name for the stage.
109      * @param stageDescription A descriptive text for the stage.
110      * @param length         Stage length in kilometres.
111      * @param startTime      The date and time in which the stage will be raced.
112      *                       It cannot be null.
113      * @param type           The type of the stage.
114      * @return The unique ID of the stage.
115      */
```

```
116     public int addStage(String stageName, String stageDescription, double length,
117         LocalDateTime startTime, StageType type){
118
119         listOfStages.add(new Stage(stageName, stageDescription, length, startTime, type));
120         assert (listOfStages.getLast().getID() == Stage.getNextStageID())
121                 : "Stage was not created with correct ID";
122         return listOfStages.getLast().getID();
123     }
124
125     /**
126      * Retrieves the list of stage IDs for a race.
127      *
128      * @return The list of stage IDs.
129      */
130     public int[] getStageIDs(){
131         int[] ListOfStageIDs = new int[listOfStages.size()];
132         for (int i = 0; i < listOfStages.size(); i++){
133             ListOfStageIDs[i] = listOfStages.get(i).getID();
134         }
135         return ListOfStageIDs;
136     }
137
138     /**
139      * Retrieves the list of stage IDs for a race and orders them by start time.
140      *
141      * @return The list of ordered stage IDs.
142      */
143     public int[] getOrderedStageIDs(){
144         int[] orderedStageIds = new int[listOfStages.size()];
145         LinkedList<LocalDateTime> StageDates =
146                 new LinkedList<>();
147
148         for (Stage stage : listOfStages) {
149             StageDates.add(stage.getStartTime());
150         }
151         Collections.sort(StageDates);
152
153         for (int i = 0; i < listOfStages.size(); i++) {
154             for (Stage stage : listOfStages) {
155                 if (StageDates.get(i) == stage.getStartTime()) {
156                     orderedStageIds[i] = stage.getID();
157
158                     StageDates.set(i, null);
159                 }
160             }
161         }
162         return orderedStageIds;
163     }
164
165     /**
166      * The method gets a list of all stage names for a race.
167      *
168      * @return The list of stage names.
169      */
170     public String[] getStageNames(){
```

```
171         String[] ListOfStageNames = new String[listOfStages.size()];
172         for (int i = 0; i < listOfStages.size(); i++){
173             ListOfStageNames[i] = listOfStages.get(i).getStageName();
174         }
175         return ListOfStageNames;
176     }
177 
178     /**
179      * Retrieves the length of a stage in a race.
180      *
181      * @param stageID The ID of the stage being queried.
182      * @return The stage's length.
183      */
184     public double getStageLength(int stageID){
185         for (Stage stage : listOfStages) {
186             if (stage.getID() == stageID) {
187                 return stage.getLength();
188             }
189         }
190 
191         //should always find length
192         return 0;
193     }
194 
195     /**
196      * Retrieves the state of a stage in a race.
197      *
198      * @param stageID The ID of the stage being queried.
199      * @return The stage's state.
200      */
201     public String getStageState(int stageID){
202         for (Stage stage : listOfStages) {
203             if (stage.getID() == stageID) {
204                 return stage.getState();
205             }
206         }
207 
208         //should always find state
209         return null;
210     }
211 
212     /**
213      * Retrieves the type of a stage in a race.
214      *
215      * @param stageID The ID of the stage being queried.
216      * @return The stage's type.
217      */
218     public StageType getStageType(int stageID){
219         for (Stage stage : listOfStages) {
220             if (stage.getID() == stageID) {
221                 return stage.getStageType();
222             }
223         }
224 
225         //should always find state
```

```
226         return null;
227     }
228
229     /**
230      * Removes a stage and all its related data.
231      *
232      * @param stageID The ID of the stage being removed.
233      */
234     public void removeStageByID(int stageID){
235         for (int i = 0; i < listOfStages.size(); i++){
236             if (listOfStages.get(i).getID() == stageID){
237                 Stage stageToRemove = listOfStages.get(i);
238                 listOfStages.remove(stageToRemove);
239             }
240         }
241     }
242
243     /**
244      * Adds a climb segment to a stage.
245      *
246      * @param stageId       The ID of the stage to which the climb segment is
247      *                      to be added.
248      * @param location      The kilometre location where the climb finishes
249      *                      within the stage.
250      * @param type          The category of the climb - {@link SegmentType#C4},
251      *    *                     {@link SegmentType#C3}, {@link SegmentType#C2},
252      *    *                     {@link SegmentType#C1}, or {@link SegmentType#HC}.
253      * @param averageGradient The average gradient of the climb.
254      * @param length        The length of the climb in kilometres.
255      * @return The unique ID of the segment created.
256      */
257     public int addClimbToStage(int stageId, Double location, SegmentType type,
258                                Double averageGradient, Double length){
259         for (Stage stage : listOfStages) {
260             if (stage.getID() == stageId) {
261                 return stage.addClimb(location, type,
262                         averageGradient, length);
263             }
264         }
265
266         return 0;
267     }
268
269     /**
270      * Adds an intermediate sprint to a stage.
271      *
272      * @param stageId The ID of the stage to which the intermediate
273      *                sprint segment is being added.
274      * @param location The kilometre location where the intermediate sprint
275      *                finishes within a stage.
276      * @return The unique ID of the segment created.
277      */
278     public int addSprintToStage(int stageId, double location){
279         for (Stage stage : listOfStages) {
280             if (stage.getID() == stageId) {
```

```
281                 return stage.addSprint(location);
282             }
283         }
284 
285         return 0;
286     }
287 
288     /**
289      * The method retrieves a list of IDs for segments in a queried stage.
290      *
291      * @param stageId The ID of the stage being queried.
292      * @return The list of segment IDs.
293      */
294     public int[] getSegmentIds(int stageId){
295         for (Stage stage : listOfStages) {
296             if (stage.getID() == stageId) {
297                 return stage.getSegmentsIds();
298             }
299         }
300         return null;
301     }
302 
303     /**
304      * The method retrieves a list of IDs for segments in a queried stage.
305      * It then sorts the list by order they occur within the stage (by location).
306      *
307      * @param stageId The ID of the stage being queried.
308      * @return The sorted list of segment IDs.
309      */
310     public int[] getOrderedSegmentIds(int stageId){
311         double[] segmentLengths = new double[listOfStages.size()];
312         int[] orderedSegmentIds = new int[listOfStages.size()];
313         int[] segmentIds = new int[listOfStages.size()];
314 
315         //creates list of segment Ids and locations
316         for (Stage stage : listOfStages) {
317             if (stage.getID() == stageId) {
318                 segmentLengths = stage.getListOfSegmentLocations();
319                 segmentIds = stage.getSegmentsIds();
320             }
321         }
322         Arrays.sort(segmentLengths);
323 
324         //Matches up segment ID to its sorted length
325         for (int i = 0; i < listOfStages.size(); i++){
326             if (listOfStages.get(i).getID() == stageId){
327                 for (int j = 0; j < listOfStages.size(); j++){
328                     for (int k = 0; k < listOfStages.size(); k++){
329                         if (segmentLengths[j] ==
330                                 listOfStages.get(i).getSegmentLocation(segmentIds[k])){
331                             orderedSegmentIds[j] = segmentIds[k];
332                             orderedSegmentIds[j] = 0;
333                         }
334                     }
335                 }
```

```
            }
        }
        return orderedSegmentIds;
    }

    /**
     * Removes a segment from a stage.
     *
     * @param stageIndex The location of the queried stage in the list of stages.
     * @param segmentId The ID of the segment to be removed.
     */
    public void removeSegmentById(int stageIndex, int segmentId){
        Stage stageObj = listOfStages.get(stageIndex);
        stageObj.removeSegment(segmentId);
    }

    /**
     * Concludes preparation of a stage by setting the stage's state
     * to "waiting for results".
     *
     * @param StageId The ID of the stage being concluded.
     */
    public void concludeStatePreparation(int StageId){
        for (Stage stage : listOfStages) {
            if (stage.getID() == StageId) {
                stage.concludeStatePreparation();
            }
        }
    }

    /**
     * Records the times of a rider in a stage
     *
     * @param stageId   The ID of the stage the result refers to.
     * @param riderId   The ID of the rider.
     * @param checkpoints An array of times at which the rider reached each of the
     *                  segments of the stage, including the start time and the
     *                  finish line.
     */
    public void registerRiderResultsInStage(int stageId, int riderId, LocalTime... checkpoints){
        for (Stage stage : listOfStages) {
            if (stage.getID() == stageId) {
                stage.addRidersTime(riderId, checkpoints);
            }
        }
    }

    /**
     * Gets the times of a rider in a stage.
     *
     * @param stageId The ID of the stage the result refers to.
     * @param riderId The ID of the rider.
     * @return The list of riders results
     */
    public LocalTime[] getRiderResults(int stageId, int riderId){
```

```
391         for (Stage stage : listOfStages) {
392             if (stage.getID() == stageId) {
393                 return stage.getRiderTimes(riderId);
394             }
395         }
396         return null;
397     }
398 
399     /**
400      * Gets the adjusted elapsed times for a rider in a stage.
401      *
402      * @param stageId The ID of the stage the result refers to.
403      * @param riderId The ID of the rider.
404      * @return The adjusted elapsed time for the rider in the stage.
405      */
406     public LocalTime getRiderAdjustedElapsedResults(int stageId, int riderId){
407         for (Stage stage : listOfStages) {
408             if (stage.getID() == stageId) {
409                 int adjustedFinishTime = stage.getRiderAdjustedElapsedTimes(riderId).toSecondOfDay();
410                 int startTime = stage.getRiderTimes(riderId)[0].toSecondOfDay();
411                 return LocalTime.ofSecondOfDay(adjustedFinishTime - startTime);
412             }
413         }
414 
415         return null;
416     }
417 
418     /**
419      * Removes the stage results from a rider.
420      *
421      * @param stageId The ID of the stage that results are being removed from.
422      * @param riderId The ID of the rider.
423      */
424     public void deleteRidersResults(int stageId, int riderId){
425         for (Stage stage : listOfStages) {
426             if (stage.getID() == stageId) {
427                 stage.removeRidersResults(riderId);
428             }
429         }
430     }
431 
432     /**
433      * Removes the results of a rider from every stage in the race.
434      *
435      * @param riderID The ID of the rider whose results are being deleted.
436      */
437     public void deleteAllRiderResults(int riderID){
438         for (Stage stage : listOfStages){
439             stage.removeRidersResults(riderID);
440         }
441     }
442 
443     /**
444      * Queries whether a rider has results in a stage.
445      *
```

```
446         * @param stageId The stage being queried.
447         * @param riderId The rider being queried.
448         * @return A boolean result depending on whether the
449         *         rider has results in that stage.
450         */
451        public boolean isRiderInResults(int stageId, int riderId){
452            for (Stage stage : listOfStages) {
453                if (stage.getID() == stageId) {
454                    return stage.isRiderInResults(riderId);
455                }
456            }
457            //never reached
458            return false;
459        }
460
461        /**
462         * Get the riders finishing position in a stage.
463         *
464         * @param stageId The ID of the stage being queried.
465         * @return A list of riders ID sorted by their elapsed time.
466         */
467        public int[] getRidersRankInStage(int stageId){
468            for (Stage stage : listOfStages) {
469                if (stage.getID() == stageId) {
470                    return stage.getRidersRank();
471                }
472            }
473            // if no riders
474            int[] nullList = new int[]{};
475            return nullList;
476        }
477
478        /**
479         * Get the number of points obtained by a rider in a stage including intermediate sprints.
480         *
481         * @param stageId     The ID of the stage being queried.
482         * @param riderPosition The rank of the rider in the stage.
483         * @return The number of points the rider won in the stage
484         */
485        public int getPointsFromStage(int stageId, int riderPosition, int riderID) {
486            int riderPoints;
487            for (Stage stage : listOfStages) { // loop through stages in race
488                if (stage.getID() == stageId) { //if desired stage
489                    int numberOfRiders = stage.getNumberOfRiders(); // number of riders in race
490                    riderPoints = (stage.getPointsForStageRank(riderPosition)); // points from stage finish
491                    for (int j = 0; j < (numberOfRiders); j++) { // loop through riders
492                        int[][] PointsFromStage = (stage.getPointsFromStageSprints()); // get the points of stage
                                sprints
493                        if (PointsFromStage[j][0] == riderID) {
494                            riderPoints += PointsFromStage[j][1]; // points from sprints
495                        }
496                    }
497                    return riderPoints;
498                } else {
499                    break;
```

```
500             }
501
502         }
503         return 0;
504     }
505
506     /**
507      * Get the number of mountain points obtained by a rider in a stage.
508      *
509      * @param stageId     The ID of the stage being queried.
510      * @param riderID     The ID of the rider.
511      * @return The number of mountain points the rider won in the stage.
512      */
513     public int getMountainPointsFromStage(int stageId, int riderID) {
514         int riderPoints = 0;
515         for (Stage stage : listOfStages) { // loop through stages in race
516             if (stage.getID() == stageId) { //if desired stage
517                 int numberOfRiders = stage.getNumberOfRiders(); // number of riders in race
518                 for (int j = 0; j < (numberOfRiders); j++) { // loop through riders
519                     int[][] PointsFromStage = (stage.getPointsFromMountainStages()); // points from stage
520                     if (PointsFromStage[j][0] == riderID) {
521                         riderPoints = PointsFromStage[j][1];
522                     }
523                 }
524                 return riderPoints;
525             }
526         }
527         return 0;
528     }
529
530     /**
531      * Get the adjusted elapsed times of a riders in a stage.
532      *
533      * @param stageId The ID of the stage being queried.
534      * @return The ranked list of adjusted elapsed times sorted by their finish
535      *         time. An empty list if there is no result for the stage. These times
536      *         should match the riders returned by
537      *         {@link #getRidersRankInStage(int)}.
538      */
539     public LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId){
540         for (Stage stage : listOfStages) {
541             if (stage.getID() == stageId) {
542                 return stage.getRankedAdjustedElapsedTimes();
543             }
544         }
545         LocalTime[] nullList = new LocalTime[]{};
546         return nullList;
547     }
548
549     /**
550      * Creates and sorts a list of rider's total time to complete a race.
551      * The total time is the summation of the riders adjusted elapsed time for each stage in the race.
552      *
553      * @return The sorted list of riders total adjusted elapsed times.
554      */
```

```
555     public LocalTime[] getGeneralClassificationTimes(){
556         LinkedList<Integer> classificationTimes = new LinkedList<>();
557 
558         //Creates list of rider's total times
559         for (int i = 1; i <= Rider.getNextRiderID(); i++){
560             int totalTime = 0;
561             for (Stage stageObj : listOfStages) {
562                 if (isRiderInResults(stageObj.getID(), i)) {
563                     totalTime = totalTime + stageObj.getRiderAdjustedElapsedTimes(i).toSecondOfDay()
564                             - stageObj.getRiderStartTime(i);
565                 } else {
566                     totalTime = -1;
567                     break;
568                 }
569             }
570             if (totalTime != -1){
571                 classificationTimes.add(totalTime);
572             }
573         }
574         Collections.sort(classificationTimes);
575 
576         //Converts rider's times from Integer to LocalTime
577         LocalTime[] sortedClassificationTimes = new LocalTime[classificationTimes.size()];
578         for (int i = 0; i < sortedClassificationTimes.length; i++){
579             sortedClassificationTimes[i] = LocalTime.ofSecondOfDay(classificationTimes.get(i));
580         }
581 
582         return sortedClassificationTimes;
583     }
584 
585     /**
586      * Calculates the general classification of riders.
587      *
588      * @return A ranked list of riders' IDs sorted by total adjusted elapsed times in all race stages.
589      */
590     public int[] getRidersGeneralClassificationRank() {
591         int arrayLength = getGeneralClassificationTimes().length;
592         int[] classificationRank = new int[arrayLength];
593         int[][] classificationRiderTime = new int[arrayLength][3];
594 
595         //Fills array with rider Ids and their corresponding total times
596         for (int i = 1; i <= Rider.getNextRiderID(); i++) {
597             int totalAdjustedTime = 0;
598             int totalTime = 0;
599             for (Stage stageObj : listOfStages) {
600                 if (isRiderInResults(stageObj.getID(), i)) {
601                     totalAdjustedTime = totalAdjustedTime +
                            stageObj.getRiderAdjustedElapsedTimes(i).toSecondOfDay()
602                                - stageObj.getRiderStartTime(i);
603                     totalTime = totalTime +
                            stageObj.getRiderTimes(i)[stageObj.getRiderTimes(i).length-1].toSecondOfDay()
604                                - stageObj.getRiderStartTime(1);
605                 } else {
606                     totalAdjustedTime = -1;
607                     break;
```

```
608                 }
609             }
610             if (totalAdjustedTime != -1) {
611                 classificationRiderTime[i - 1][0] = i;
612                 classificationRiderTime[i - 1][1] = totalAdjustedTime;
613                 classificationRiderTime[i-1][2] = totalTime;
614             }
615         }
616 
617         //Sorts the array by the rider's times
618         Arrays.sort(classificationRiderTime, (first, second) -> {
619             if (first[2] > second[2]) return 1;
620             else return -1;
621         });
622 
623         //Sorts the array by the rider's adjusted times
624         Arrays.sort(classificationRiderTime, (first, second) -> {
625             if (first[1] >= second[1]) return 1;
626             else return -1;
627         });
628 
629         //Extracts sorted rider Ids
630         for (int i = 0; i < arrayLength; i++) {
631             classificationRank[i] = classificationRiderTime[i][0];
632         }
633 
634         return classificationRank;
635     }
636 
637 
638     /**
639      * Calculates the number of points for each rider across a whole race (sum of all stages).
640      * Points match riders in the order of getRidersGeneralClassificationRank.
641      *
642      * @return Array of riders points in a race.
643      */
644 
645     public int[] getRidersOverallPoints(){
646         int[] ridersSorted = getRidersGeneralClassificationRank();
647         int[] ridersPoints = new int[getRidersGeneralClassificationRank().length];
648         for (int i = 0; i <= ridersSorted.length-1; i++){ // loop through riders
649             for (Stage stageObj : listOfStages) { // loop through stages
650                 //ridersPoints[i] +=
                        stageObj.getPointsForStageRank(stageObj.getRidersRank()[ridersSorted[(i)]-1]);
651                 // score for position in race for rider in ridersSorted[i]
652                 for (int j =0; j < ridersSorted.length; j++){
653                     if (stageObj.getRidersRank()[j] == ridersSorted[i]) {
654                         ridersPoints[i] += stageObj.getPointsForStageRank(j);
655                     }
656                     if(stageObj.getPointsFromStageSprints()[j][0] == ridersSorted[i]){ //add riders sprint
                            points
657                         ridersPoints[i] += stageObj.getPointsFromStageSprints()[j][1];
658                     }
659                 }
660             }
```

```
        }
        return ridersPoints;
    }

    /**
     * Calculates the number of points in mountain stages for a rider in a race.
     *
     * @return array of riders mountain points within a race
     */
    public int[] getRidersOverallMountainPoints(){
        int[] ridersSorted = getRidersGeneralClassificationRank();
        int[] ridersPoints = new int[getRidersGeneralClassificationRank().length];
        for (int i = 0; i < ridersSorted.length; i++){ // loop through riders
            for (Stage stageObj : listOfStages) { // loop through stages
                for (int j =0; j < ridersSorted.length; j++) {
                    if (stageObj.getPointsFromMountainStages()[j][0] == ridersSorted[i]) {
                        ridersPoints[i] += stageObj.getPointsFromMountainStages()[j][1];
                    }
                }
            }
        }
        return ridersPoints;
    }

}
```

3 Stage.java

```
package cycling;

import java.io.Serializable;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.*;

/**
 *  The java class for stage. Contains methods relating to stages within races in the cycling app.
 *
 *  @author Toby Slump and James Cracknell
 *  @version 1.0
 *  03/2022
 */
public class Stage implements Serializable {
    private int stageID;
    private static int nextStageID = 0;
    private String stageName;
    private String description;
    private double length;
    private LocalDateTime startTime;
    private StageType type;
    private LinkedList<Segment> listOfSegments = new LinkedList<>();
    private String state;
    private Map<Integer, LocalTime[]> rawRiderResults = new HashMap<>(); //riderid, ridertimes

```

```
27      /**
28       * Stage Class constructor.
29       *
30       * @param stageName Name of the stage.
31       * @param description Description of the stage.
32       * @param length    Length of the stage (kms).
33       * @param startTime Start time of the stage in LocalDateTime format.
34       * @param type      Type of stage.
35       */
36      public Stage(String stageName, String description, double length,
37                   LocalDateTime startTime, StageType type) {
38          this.stageID = ++nextStageID;
39          this.stageName = stageName;
40          this.description = description;
41          this.length = length;
42          this.startTime = startTime;
43          this.type = type;
44          this.state = "constructing";
45      }
46
47      /**
48       * Gets the stage ID.
49       *
50       * @return The queried stage's unique ID.
51       */
52      public int getID() {
53          return stageID;
54      }
55
56      /**
57       * Gets the stage name.
58       *
59       * @return The queried stage's name.
60       */
61      public String getStageName() {
62          return stageName;
63      }
64
65      /**
66       * Gets the stage length.
67       *
68       * @return The queried stage's length.
69       */
70      public double getLength() {
71          return length;
72      }
73
74      /**
75       * Gets the stage state.
76       *
77       * @return The state of the queried stage.
78       */
79      public String getState() {
80          return state;
81      }
```

```
82
83      /**
84       * Gets stage type.
85       *
86       * @return The type of queried stage.
87       */
88      public StageType getStageType() {
89          return type;
90      }
91
92      /**
93       * Gets stage start time.
94       *
95       * @return The start time of the queried stage.
96       */
97      public LocalDateTime getStartTime() {
98          return startTime;
99      }
100
101     /**
102      * Gets the value of nextStageID.
103      *
104      * @return The ID of the last stage to be made.
105      */
106     public static int getNextStageID() {
107         return nextStageID;
108     }
109
110     /**
111      *Gets the list of times for a rider in a stage.
112      *
113      * @param riderId The ID of the queried rider.
114      * @return The list of the rider's times.
115      */
116     public LocalTime[] getRiderTimes(int riderId) {
117         return rawRiderResults.get(riderId);
118     }
119
120     /**
121      * Gets the number of riders present in the stage's results.
122      *
123      * @return The number of riders who are recorded in the queried stage's results.
124      */
125     public int getNumberOfRiders() {
126         return (rawRiderResults.size());
127     }
128
129     /**
130      * Retrieves a list of IDs for the segments in the stage.
131      *
132      * @return The list of segment IDs in a queried stage.
133      */
134     public int[] getSegmentsIds() {
135         int[] listOfSegmentIds = new int[listOfSegments.size()];
136         for (int i = 0; i < listOfSegments.size(); i++) {
```

```
            listOfSegmentIds[i] = listOfSegments.get(i).getSegmentID();
        }
        return listOfSegmentIds;
    }

    /**
     * Creates and returns a list of segment lengths in kilometers.
     *
     * @return List of the lengths of segments.
     */
    public double[] getListOfSegmentLocations() {
        double[] segmentLengths = new double[listOfSegments.size()];
        for (int i = 0; i < listOfSegments.size(); i++) {
            segmentLengths[i] = listOfSegments.get(i).getLocation();
        }
        return segmentLengths;
    }

    /**
     * Finds and returns the location of the requested segment.
     *
     * @param segmentId The ID of the segment being queried.
     * @return The location of the queried segment.
     */
    public double getSegmentLocation(int segmentId){
        for (Segment segment : listOfSegments) {
            if (segment.getSegmentID() == segmentId) {
                return segment.getLocation();
            }
        }
        return 0;
    }

    /**
     * Sets the value of nextStageID.
     *
     * @param nextStageId The new value of nextStageID.
     */
    public static void setNextStageID(int nextStageId) {
        nextStageID = nextStageId;
    }

    /**
     * Adds a climb segment to a stage.
     *
     * @param location     The location in which the climb starts.
     * @param type         The type of climb.
     * @param averageGradient The average gradient of the climb.
     * @param length       The length of the climb in kilometers.
     * @return The segment ID of the added climb segment.
     */
    public int addClimb(Double location, SegmentType type,
                        Double averageGradient, Double length) {
        listOfSegments.add(new ClimbSegment(location, type, averageGradient, length));
        assert (listOfSegments.getLast().getSegmentID() == ClimbSegment.getNextSegmentID())
```

```
192                 : "Segment was not created with correct ID";
193         return listOfSegments.getLast().getSegmentID();
194     }
195 
196     /**
197      * Adds a sprint segment to the stage.
198      *
199      * @param location The location in which the stage starts.
200      * @return The segment ID of the added sprint segment.
201      */
202     public int addSprint(Double location) {
203         listOfSegments.add(new SprintSegment(location));
204         assert (listOfSegments.getLast().getSegmentID() == SprintSegment.getNextSegmentID())
205                 : "Segment was not created with correct ID";
206         return listOfSegments.getLast().getSegmentID();
207     }
208 
209     /**
210      * Removes the given segment from the stage.
211      *
212      * @param segmentId The ID of the segment being removed.
213      */
214     public void removeSegment(int segmentId) {
215         for (int i = 0; i < listOfSegments.size(); i++) {
216             if (listOfSegments.get(i).getSegmentID() == segmentId) {
217                 listOfSegments.remove(listOfSegments.get(i));
218             }
219         }
220     }
221 
222     /**
223      * Indicates stage preparation has been completed, allowing results to be added.
224      */
225     public void concludeStatePreparation() {
226         state = "waiting for results";
227     }
228 
229     /**
230      * Adds a riders time to their results.
231      *
232      * @param riderId  The ID of the rider that results relate to.
233      * @param riderTimes The times the rider achieved that are to be added.
234      */
235     public void addRidersTime(int riderId, LocalTime[] riderTimes) {
236         rawRiderResults.put(riderId, riderTimes);
237     }
238 
239     /**
240      * Gets the adjusted elapsed times. Riders within one second of each other are grouped together into the
          same
241      * position and achieve same points.
242      *
243      * @param riderId The ID of the rider.
244      * @return The adjusted elapsed time for the rider in the stage.
245      */
```

```
246     public LocalTime getRiderAdjustedElapsedTimes(int riderId) {
247         boolean finishedAdjusting = false;
248         LocalTime finishTime = rawRiderResults.get(riderId)[rawRiderResults.get(riderId).length - 1];
249         boolean hasAdjusted;
250 
251         while (!finishedAdjusting) {
252             hasAdjusted = false;
253             for (Integer key : rawRiderResults.keySet()) {
254                 double riderTimeInSeconds = finishTime.toSecondOfDay();
255 
256                 if (key != riderId) {
257                     //If rider is within 1 second of a rider in front, lower their finish time
258                     if (rawRiderResults.get(key)[rawRiderResults.get(key).length - 1].toSecondOfDay() -
259                             riderTimeInSeconds >= -1 && rawRiderResults.get(key)
260                             [rawRiderResults.get(key).length - 1].toSecondOfDay() -
261                             riderTimeInSeconds < 0) {
262 
263                         finishTime = rawRiderResults.get(key)[rawRiderResults.get(key).length - 1];
264                         hasAdjusted = true;
265 
266                     }
267                 }
268             }
269 
270             if (!hasAdjusted) {
271                 finishedAdjusting = true;
272             }
273         }
274 
275         return finishTime;
276     }
277 
278     /**
279      * Removes the results of the requested rider.
280      *
281      * @param riderId The ID of the rider.
282      */
283     public void removeRidersResults(int riderId) {
284         rawRiderResults.remove(riderId);
285     }
286 
287     /**
288      * Queries if the rider has results for them.
289      *
290      * @param riderId The ID of the rider.
291      * @return A boolean, true when rider has results for them.
292      */
293     public boolean isRiderInResults(int riderId) {
294         return rawRiderResults.containsKey(riderId);
295     }
296 
297     /**
298      * Returns the time that the requested rider started the race.
299      *
300      * @param riderId The ID of the rider.
```

```
301         * @return The start time of the rider in the race.
302         */
303        public int getRiderStartTime(int riderId) {
304            return rawRiderResults.get(riderId)[0].toSecondOfDay();
305        }
306
307        /**
308         * Sorts the array of riders in the race based on their results (time) and ranking.
309         *
310         * @return sorted array of rider IDs.
311         */
312        public int[] getRidersRank() {
313            int[][] riderTimes = new int[rawRiderResults.size()][2];
314            int index = 0;
315
316            //Fills array with rider Ids and their corresponding elapsed time
317            for (Integer key : rawRiderResults.keySet()) {
318                LocalTime[] riderTimesList = rawRiderResults.get(key);
319                int riderFinishTime = riderTimesList[rawRiderResults.get(key).length - 1].toSecondOfDay()
320                        - riderTimesList[0].toSecondOfDay();
321
322                riderTimes[index][0] = key;
323                riderTimes[index][1] = riderFinishTime;
324                index += 1;
325            }
326
327            //Sorts array by elapsed time
328            Arrays.sort(riderTimes, (first, second) -> {
329                if (first[1] > second[1]) return 1;
330                else return -1;
331            });
332
333            //Extracts sorted rider Ids from array
334            int[] riderRanks = new int[rawRiderResults.size()];
335            for (int i = 0; i < rawRiderResults.size(); i++) {
336                riderRanks[i] = riderTimes[i][0];
337            }
338            return riderRanks;
339        }
340
341        /**
342         * Calculates the adjusted finish time of riders in a stage, riders within 1s of each other are grouped.
343         *
344         * @return Array of the adjusted finish times of riders in the stage.
345         */
346        public LocalTime[] getRankedAdjustedElapsedTimes() {
347            int[] RankedAdjustedElapsedTimesSeconds = new int[rawRiderResults.size()];
348            int index = 0;
349
350            //Fills array with rider adjusted finish times
351            for (Integer key : rawRiderResults.keySet()) {
352                RankedAdjustedElapsedTimesSeconds[index] = getRiderAdjustedElapsedTimes(key).toSecondOfDay()
353                        - rawRiderResults.get(key)[0].toSecondOfDay();
354                index += 1;
355            }
```

```
356         Arrays.sort(RankedAdjustedElapsedTimesSeconds);
357 
358         //Converts finish time from Int to LocalTime
359         LocalTime[] RankedAdjustedElapsedTimes = new LocalTime[rawRiderResults.size()];
360         for (int i = 0; i < rawRiderResults.size(); i++) {
361             RankedAdjustedElapsedTimes[i] = LocalTime.ofSecondOfDay(RankedAdjustedElapsedTimesSeconds[i]);
362 
363         }
364 
365         return RankedAdjustedElapsedTimes;
366     }
367 
368     /**
369      * Calculates the points for the rider based on their finishing position in the race.
370      *
371      * @param cyclistPosition The position of the cyclist in the race.
372      * @return The points scored by the rider.
373      */
374     public int getPointsForStageRank(int cyclistPosition) { // selects stage type
375         switch (this.type) {
376             case FLAT -> {
377                 // score for flat stage
378                 int[] flatPoints = {50, 30, 20, 18, 16, 14, 12, 10, 8,
379                         7, 6, 5, 4, 3, 2};
380                 if (cyclistPosition < 14) {
381                     return flatPoints[cyclistPosition];
382                 } else {
383                     return 0;
384                 }
385             }
386             case MEDIUM_MOUNTAIN -> {
387                 // score for medium mountain stage
388                 int[] mediumMountainPoints = {30, 25, 22, 19, 17, 15, 13, 11,
389                         9, 7, 6, 5, 4, 3, 2};
390                 if (cyclistPosition < 14) { //position is -1 due to 0 indexing. i.e., first place is position
                        0.
391                     return mediumMountainPoints[cyclistPosition];
392                 } else {
393                     return 0;
394                 }
395             }
396             // Score for high mountain stage
397             case HIGH_MOUNTAIN, TT -> {
398                 // score for TT mountain stage (Same as high mountain)
399                 int[] highMountainPoints = {20, 17, 15, 13, 11, 10, 9,
400                         8, 7, 6, 5, 4, 3, 2, 1};
401                 if (cyclistPosition < 14) {
402                     return highMountainPoints[cyclistPosition];
403                 } else {
404                     return 0;
405                 }
406             }
407             default -> {
408                 assert false : "Not a valid stage type";
409             }
```

```
410         }
411         return 0;
412     }
413 
414     /**
415      * Calculates the points earned by the riders based on their times in the intermediate sprints within
           the stage.
416      *
417      * @return Array of rider IDs and the points they earned through sprints.
418      */
419     public int[][] getPointsFromStageSprints() { //calculate points for intermediate sprints
420         LocalTime[] riderTimesList; //list of times for a rider
421         int[][] ridersTimes = new int[rawRiderResults.size()][2]; // list of times for that segment for the
               race
422         int index = 0;
423         int[][] ridersPoints = new int[rawRiderResults.size()][2]; //list of riders points
424         for (int i = 0; i < listOfSegments.size(); i++) {
425             if (listOfSegments.get(i).getSegmentType() == SegmentType.SPRINT) { // if sprint segment
426                 for (Integer key : rawRiderResults.keySet()) { // loop through riders
427                     riderTimesList = rawRiderResults.get(key); //to account for start time
428                     ridersTimes[index][0] = key;
429                     ridersTimes[index][1] = riderTimesList[i+1].toSecondOfDay();
430                     index += 1;
431                 }
432 
433                 //sort riders into order they crossed line
434                 Arrays.sort(ridersTimes, (o1, o2) -> {
435                     if (o1[1] > o2[1]) return 1;
436                     else return -1;
437                 });
438 
439                 int[] intermediateSprintPoints = {20, 17, 15, 13, 11, 10, 9,
440                         8, 7, 6, 5, 4, 3, 2, 1};
441                 for (int j = 0; j < rawRiderResults.size(); j++) {
442                     if (j < 14) {
443                         ridersPoints[j][0] = ridersTimes[j][0]; //rider ID
444                         ridersPoints[j][1] = intermediateSprintPoints[j];
445                     } else {
446                         ridersPoints[j][0] = ridersTimes[j][0];
447                         ridersPoints[j][1] = 0;
448                     }
449                 }
450 
451             }
452         }
453 
454         return ridersPoints;
455     }
456 
457     /**
458      * Calculates points earned within mountain stages by riders.
459      *
460      * @return Array of rider IDs and the points they earned through mountain stage.
461      */
462     public int[][] getPointsFromMountainStages() {
```

```
        LocalTime[] riderTimesList; //list of times for a rider
        int[][] ridersTimes = new int[rawRiderResults.size()][2]; // list of times for that segment for the race
        int index = 0;
        SegmentType typeOfSegment = null;
        int[][] ridersPoints = new int[rawRiderResults.size()][2]; //list of riders points
        for (int i = 0; i < listOfSegments.size(); i++) {
            if (listOfSegments.get(i).getSegmentType() != SegmentType.SPRINT) { // if not sprint segment, so climb segment
                typeOfSegment = listOfSegments.get(i).getSegmentType();
                for (Integer key : rawRiderResults.keySet()) { // loop through riders
                    riderTimesList = rawRiderResults.get(key); //to account for start time
                    ridersTimes[index][0] = key;
                    ridersTimes[index][1] = riderTimesList[i+1].toSecondOfDay(); //
                    index += 1;
                }

                //sort riders into order they crossed line
                Arrays.sort(ridersTimes, (o1, o2) -> {
                    if (o1[1] > o2[1]) return 1;
                    else return -1;
                });
            }
        }
        int[] HCPoints = {20, 15, 12, 10, 8, 6, 4, 2}; // Points that HC earns
        int[] OneCPoints = {10, 8, 6, 4, 2, 1};
        int[] TwoCPoints = {5, 3, 2, 1};
        int[] ThreeCPoints = {2, 1};
        int[] FourCPoints = {1};
        for (int i = 0; i < rawRiderResults.size(); i++) { // Determine points that each cyclist earns from position
            if (typeOfSegment != null) { //to prevent error, requires it to not be null
                switch (typeOfSegment) {
                    case HC -> {
                        ridersPoints[i][0] = ridersTimes[i][0];
                        if (i < HCPoints.length) {
                            ridersPoints[i][1] = HCPoints[i];
                        } else {
                            ridersPoints[i][1] = 0;
                        }
                    }
                    case C1 -> {
                        ridersPoints[i][0] = ridersTimes[i][0];
                        if (i < OneCPoints.length) {
                            ridersPoints[i][1] = OneCPoints[i];
                        } else {
                            ridersPoints[i][1] = 0;
                        }
                    }
                    case C2 -> {
                        ridersPoints[i][0] = ridersTimes[i][0];
                        if (i < TwoCPoints.length) {
                            ridersPoints[i][1] = TwoCPoints[i];
                        } else {
                            ridersPoints[i][1] = 0;
```

```
                    }
                }
                case C3 -> {
                    ridersPoints[i][0] = ridersTimes[i][0];
                    if (i < ThreeCPoints.length) {
                        ridersPoints[i][1] = ThreeCPoints[i];
                    } else {
                        ridersPoints[i][1] = 0;
                    }
                }
                case C4 -> {
                    ridersPoints[i][0] = ridersTimes[i][0];
                    if (i < FourCPoints.length) {
                        ridersPoints[i][1] = FourCPoints[i];
                    } else {
                        ridersPoints[i][1] = 0;
                    }
                }
                default -> {
                    assert false : "Not a valid climb";
                }
            }
        }
    }
    return ridersPoints;
  }
}
```

4 Segment.java

```
package cycling;

import java.io.Serializable;

/**
 *  The abstract java class for segement. Contains methods relating to segments within stages within races
 *  in the cycling app. Segments are lengths of races with an additional purpose such as intermediate
     sprints, or climbs.
 *
 *  @author Toby Slump and James Cracknell
 *  @version 1.0
 *  03/2022
 */
public abstract class Segment implements Serializable {
    protected int segmentID;
    protected static int nextSegmentID = 0;
    protected SegmentType type;
    protected double location;

    /**
     * Class constructor
     */
    public Segment(){
        this.segmentID = ++nextSegmentID;
```

```
24      }
25
26      /**
27       * Gets the type of a queried segment.
28       *
29       * @return The type of the segment.
30       */
31      public SegmentType getSegmentType() {
32          return type;
33      }
34
35      /**
36       * Gets the finish location of the segment in a stage (kms).
37       *
38       * @return The location of the queried segment.
39       */
40      public double getLocation(){
41          return location;
42      }
43
44      /**
45       * Gets the ID of the segment.
46       *
47       * @return The ID of the queried segment.
48       */
49      abstract int getSegmentID();
50
51  }
```

5 SprintSegment.java

```
1   package cycling;
2
3   import java.io.Serializable;
4
5   /**
6    *  The java class for sprint segments. Contains methods relating to sprint segments within stages of races
7    *  in the cycling app.
8    *
9    *  @author Toby Slump and James Cracknell
10   *  @version 1.0
11   *  03/2022
12   */
13  public class SprintSegment extends Segment implements Serializable {
14
15      /**
16       * Sprint Segment class constructor.
17       *
18       * @param Location The finish location of the segment in a stage.
19       */
20      public SprintSegment(Double Location){
21          super();
22          location = Location;
23          this.type = SegmentType.SPRINT;
```

```
24
25      }
26
27      /**
28       * Gets unique ID for a segment.
29       *
30       * @return Segment ID.
31       */
32      @Override
33      public int getSegmentID() {
34          return segmentID;
35      }
36
37      /**
38       * Gets the value of nextSegmentID.
39       *
40       * @return The ID of the last segment created.
41       */
42      public static int getNextSegmentID(){
43          return nextSegmentID;
44      }
45
46      /**
47       *Sets the value of nextSegmentID.
48       *
49       * @param nextSegmentId The new value of SegmentID.
50       */
51      public static void setNextSegmentID(int nextSegmentId){
52          nextSegmentID = nextSegmentId;
53      }
54  }
```

6 ClimbSegment.java

```
1   package cycling;
2
3   import java.io.Serializable;
4
5   /**
6    *  The java class for climb segment. Contains methods relating to climb segments within stages of races
7    *  in the cycling app.
8    *
9    *  @author Toby Slump and James Cracknell
10   *  @version 1.0
11   *  03/2022
12   */
13  public class ClimbSegment extends Segment implements Serializable {
14      private double averageGradient;
15      private double length;
16
17      /**
18       * Climb Segment class constructor.
19       *
20       * @param Location      The finish location of the segment in the stage.
```

```
     * @param type          The category of the climb segment.
     * @param averageGradient The average gradient of the segment.
     * @param length        The length of the segment in kilometres.
     */
    public ClimbSegment(Double Location, SegmentType type, Double averageGradient,
                        Double length){
        super();
        this.location = Location;
        this.type = type;
        this.averageGradient = averageGradient;
        this.length = length;

    }

    /**
     * Gets unique ID for a segment.
     *
     * @return Segment ID.
     */
    @Override
    public int getSegmentID(){
        return segmentID;
    }

    /**
     * Gets the value of nextSegmentID.
     *
     * @return The ID of the last segment created.
     */
    public static int getNextSegmentID(){
        return nextSegmentID;
    }

    /**
     *Sets the value of nextSegmentID.
     *
     * @param nextSegmentId The new value of SegmentID.
     */
    public static void setNextSegmentID(int nextSegmentId){
        nextSegmentID = nextSegmentId;
    }
}
```

7 Team.java

```
package cycling;

import java.io.Serializable;
import java.util.LinkedList;

/**
 *  The java class for teams. Contains methods relating to Teams in the cycling app.
 *
 *  @author Toby Slump and James Cracknell
```

```
 *  @version 1.0
 *  03/2022
 */
public class Team implements Serializable {

    private int teamID;
    private static int nextTeamID = 0;
    private String name;
    private String description;
    private LinkedList<Rider> listOfRiders = new LinkedList<>();

    /**
     * Team class constructor.
     *
     * @param name      Team's name.
     * @param Description Team's description.
     */
    public Team(String name, String Description){
        this.teamID = ++nextTeamID;
        this.name = name;
        this.description = Description;
    }

    /**
     * Creates a rider and adds them to the team.
     *
     * @param name      Rider's name.
     * @param yearOfBirth Rider's year of birth.
     */
    public void addRider(String name, int yearOfBirth){
        Rider newRider = new Rider(teamID, name, yearOfBirth);
        listOfRiders.add(newRider);
    }

    /**
     * Gets value of nextTeamID.
     *
     * @return The ID of the last team created.
     */
    public static int getNextTeamID(){
        return nextTeamID;
    }

    /**
     * Sets the value of nextTeamID.
     *
     * @param nextTeamId The new value of nextTeamID.
     */
    public static void setNextTeamID(int nextTeamId){
        nextTeamID = nextTeamId;
    }

    /**
     * Removes a rider from the team.
     *
```

```
     * @param riderId The ID of the rider being removed.
     */
    public void removeRider(int riderId){
        for (int i = 0; i < listOfRiders.size(); i++){
            if (listOfRiders.get(i).getId() == riderId){
                listOfRiders.remove(listOfRiders.get(i));
            }
        }
    }

    public int getTeamID(){
        return teamID;
    }

    /**
     * Retrieves a list of riders ID who are in the team.
     *
     * @return The list of rider IDs.
     */
    public int[] getRiderIds(){
        int[] listOfRiderIds = new int[listOfRiders.size()];

        for (int i = 0; i < listOfRiders.size(); i++){
            listOfRiderIds[i] = listOfRiders.get(i).getId();
        }

        return listOfRiderIds;
    }

    /**
     * Retrieves the unique ID of the rider newest to the team.
     *
     * @return The newest rider's ID.
     */
    public int getNewRiderID(){
        return listOfRiders.getLast().getId();
    }


}
```

8 Rider.java

```
package cycling;

import java.io.Serializable;

/**
 *  The java class for riders. Contains methods relating to riders within teams in the cycling app.
 *
 *  @author Toby Slump and James Cracknell
 *  @version 1.0
 *  03/2022
 */
```

```java
public class Rider implements Serializable {
    private int riderID;
    private static int nextRiderID = 0;
    private int teamID;
    private String name;
    private int yearOfBirth;

    /**
     * Rider class constructor.
     *
     * @param teamID    The ID of the team the rider will be added to.
     * @param name      Rider's name.
     * @param yearOfBirth Rider's year of birth.
     */
    public Rider(int teamID, String name, int yearOfBirth){
        this.teamID = teamID;
        this.name = name;
        this.yearOfBirth = yearOfBirth;
        this.riderID = ++nextRiderID;
    }

    /**
     * Gets unique rider ID.
     *
     * @return The ID of the rider.
     */
    public int getId(){
        return riderID;
    }

    /**
     * Gets the value of nextRiderID.
     *
     * @return The ID of the last rider created.
     */
    public static int getNextRiderID(){
        return nextRiderID;
    }

    /**
     * Sets the value of nextRiderID.
     *
     * @param nextRiderId The new value of nextRiderID.
     */
    public static void setNextRiderID(int nextRiderId){
        nextRiderID = nextRiderId;
    }
}
```