File - CyclingPortal.java

```java
package cycling;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;
import java.util.*;
import java.util.Map.Entry;
import java.util.stream.Collectors;

/**
 * CyclingPortal class which implements CyclingPortalInterface.
 * <p>
 *     The no-argument constructor of this class initialises the CyclingPortal
 *     as an empty platform with no initial racing teams nor races within it.
 * </p>
 *
 *
 * @author Joey Griffiths and Alexander Cairns
 *
 */
public class CyclingPortal implements CyclingPortalInterface {

    /**
     * A private, final, 2D array of integers, used to represent the points
     * earned for each rank in a stage, for different types of stages.<br>
     * To use: pointsTable[type][rank]
     */
    private final int[][] pointsTable = {
        {50, 30, 20, 18, 16, 14, 12, 10, 8, 7, 6, 5, 4, 3, 2},
        {30, 25, 22, 19, 17, 15, 13, 11, 9, 7, 6, 5, 4, 3, 2},
        {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
    };

    /**
     * A private, final, 2D array of integers, used to represent the points
     * earned for each rank in a mountain segment, for different types of
     * mountain segments.<br>
```

```
43      * To use: mountainPointsTable[rank][type]
44      */
45     private final int[][] mountainPointsTable = {
46         {1, 2, 5, 10, 20},
47         {0, 1, 3, 8, 15},
48         {0, 0, 2, 6, 12},
49         {0, 0, 1, 4, 10},
50         {0, 0, 0, 2, 8},
51         {0, 0, 0, 1, 6},
52         {0, 0, 0, 0, 4},
53         {0, 0, 0, 0, 2}
54     };
55 
56     /**
57      * An ArrayList of Race objects.<br>
58      * Stores all active races in the system.
59      */
60     private ArrayList<Race> races = new ArrayList<>();
61 
62     /**
63      * An ArrayList of Team objects.<br>
64      * Used to store all active teams in the system.
65      */
66     private ArrayList<Team> teams = new ArrayList<>();
67 
68     @Override
69     public int[] getRaceIds() {
70         // Initialise int[] of the same length as races
   ArrayList
71         int[] raceIds = new int[races.size()];
72         for (int i=0; i<races.size(); i++) {
73             // For each item in races, add this race's ID to
   raceIds
74             raceIds[i] = races.get(i).getId();
75         }
76         return raceIds;
77     }
78 
79     @Override
80     public int createRace(String name, String description)
   throws IllegalNameException, InvalidNameException {
81         // Race name input validation
82         if (name == null) { throw new InvalidNameException("
   Race name cannot be null"); }
83         if (name.isEmpty()) { throw new InvalidNameException("
   Race name cannot be an empty string"); }
84         if (name.length() > 30) { throw new
   InvalidNameException("Race name cannot be greater than 30
```

```
 84 characters"); }
 85         if (name.contains(" ")) { throw new
    InvalidNameException("Race name cannot contain white space"
    ); }
 86
 87         for (Race race : races) {
 88             // Searches through races to find one which has
    this same name
 89             if (race.getName().equals(name)) {
 90                 // If the name already exists, exception
    thrown
 91                 throw new IllegalNameException("Race name " +
    name + " already exists");
 92             }
 93
 94         }
 95
 96         // Checks passed, race in instantiated and added to
    the list of races
 97         Race race = new Race(name, description);
 98         races.add(race);
 99         assert (races.size() > 0);
100         return race.getId();
101     }
102
103     @Override
104     public String viewRaceDetails(int raceId) throws
    IDNotRecognisedException {
105         // Races have a method to return their details
106         return getRaceById(raceId).getDetails();
107     }
108
109     @Override
110     public void removeRaceById(int raceId) throws
    IDNotRecognisedException {
111         // Finds Race object and removes it from list of races
112         races.remove(getRaceById(raceId));
113     }
114
115     @Override
116     public int getNumberOfStages(int raceId) throws
    IDNotRecognisedException {
117         // Finds Race object and uses its built in
    getNoOfStages() method
118         return getRaceById(raceId).getNoOfStages();
119     }
120
121     @Override
```

```
122     public int addStageToRace(int raceId, String stageName,
    String description, double length, LocalDateTime startTime,
    StageType type)
123             throws IDNotRecognisedException,
    IllegalNameException, InvalidNameException,
    InvalidLengthException {
124         // Stage name and length validation
125         if (stageName == null) { throw new
    InvalidNameException("Stage name cannot be null"); }
126         if (stageName.isEmpty()) { throw new
    InvalidNameException("Stage name cannot be an empty string"
    ); }
127         if (stageName.length() > 30) { throw new
    InvalidNameException("Stage name cannot be greater than 30
    characters"); }
128         if (length<5) { throw new InvalidLengthException("
    Stage length cannot be less than 5(km)"); }
129         if (stageName.contains(" ")) { throw new
    InvalidNameException("Stage name cannot contain white space"
    ); }
130
131         Race raceToAddTo = null;
132         for (Race race : races) {
133             for (Stage stage : race.getStages()) {
134                 // Searches through stages to find one which
    has this same name
135                 if (stage.getName().equals(stageName)) {
136                     // If name already exists, exception
    thrown
137                     throw new IllegalNameException("Stage name
     " + stageName + " already exists");
138                 }
139             }
140             // Using this loop to find the race with this ID
    instead of getRaceById() saves computation
141             if (race.getId() == raceId) { raceToAddTo = race
    ; }
142         }
143         if (raceToAddTo == null) {
144             throw new IDNotRecognisedException("No race with
    an ID of "+ raceId + " exists");
145         }
146
147         // Checks passed, instantiates the stage and adds it
    to list of stages in race
148         Stage stage = new Stage(stageName, description, length
    , startTime,
149                 type);
```

```
150         raceToAddTo.addStage(stage);
151         assert (raceToAddTo.getNoOfStages() > 0);
152         return stage.getId();
153     }
154 
155     @Override
156     public int[] getRaceStages(int raceId) throws
    IDNotRecognisedException {
157         // Finds the correct Race object and retrieves its
    stages
158         Stage[] stages = getRaceById(raceId).getStages();
159 
160         // Converts array of Stage objects into array of
    corresponding IDs
161         int[] stageIds = new int[stages.length];
162         for (int i=0; i<stages.length; i++) {
163             stageIds[i] = stages[i].getId();
164         }
165         return stageIds;
166     }
167 
168     @Override
169     public double getStageLength(int stageId) throws
    IDNotRecognisedException {
170         // Finds Stage object and returns its length
171         return getStageById(stageId).getLength();
172     }
173 
174     @Override
175     public void removeStageById(int stageId) throws
    IDNotRecognisedException {
176         // Finds Race object and Stage object and uses the
    race's removeStage() method
177         getRaceByStageId(stageId).removeStage(getStageById(
    stageId));
178     }
179 
180     @Override
181     public int addCategorizedClimbToStage(int stageId, Double
    location, SegmentType type, Double averageGradient,
182             Double length) throws IDNotRecognisedException,
    InvalidLocationException, InvalidStageStateException,
183             InvalidStageTypeException {
184         Stage stage = getStageById(stageId);
185         // Input validation
186         if (stage.isPrepared()) {
187             throw new InvalidStageStateException("Stage is
    already 'waiting for results'");
```

```
188         }
189         if (location > stage.getLength() || location < 0) {
190             throw new InvalidLocationException("Location out
    of bounds");
191         }
192         if (stage.getType() == StageType.TT) {
193             throw new InvalidStageTypeException("Time-trial
    stages cannot contain segments");
194         }
195         // If arguments are valid, new Segment is instantiated
     and added to stage's list of segments
196         Segment segment = new Segment(location, type,
    averageGradient, length);
197         stage.addSegment(segment);
198         return segment.getId();
199     }
200 
201     @Override
202     public int addIntermediateSprintToStage(int stageId,
    double location) throws IDNotRecognisedException,
203             InvalidLocationException,
    InvalidStageStateException, InvalidStageTypeException {
204         Stage stage = getStageById(stageId);
205         // Input validation
206         if (stage.isPrepared()) {
207             throw new InvalidStageStateException("Stage is
    already 'waiting for results'");
208         }
209         if (location > stage.getLength() || location < 0) {
210             throw new InvalidLocationException("Location out
    of bounds");
211         }
212         if (stage.getType() == StageType.TT) {
213             throw new InvalidStageTypeException("Time-trial
    stages cannot contain segments");
214         }
215         // If arguments are valid, new Segment is instantiated
     and added to stage's list of segments
216         Segment segment = new Segment(location, SegmentType.
    SPRINT);
217         stage.addSegment(segment);
218         return segment.getId();
219     }
220 
221     @Override
222     public void removeSegment(int segmentId) throws
    IDNotRecognisedException, InvalidStageStateException {
223         // Finds the stage the segment is located in
```

```
224         Stage stage = getStageBySegmentId(segmentId);
225         if (stage.isPrepared()) {
226             throw new InvalidStageStateException("Stage is
    already 'waiting for results'");
227         }
228         // Removes segment from stage
229         stage.removeSegment(getSegmentById(segmentId));
230     }
231 
232     @Override
233     public void concludeStagePreparation(int stageId) throws
    IDNotRecognisedException, InvalidStageStateException {
234         // Finds the correct stage from the ID
235         Stage stage = getStageById(stageId);
236         if (stage.isPrepared()) {
237             throw new InvalidStageStateException("Stage is
    already 'waiting for results'");
238         }
239         // Prepares the stage
240         stage.prepare();
241     }
242 
243     @Override
244     public int[] getStageSegments(int stageId) throws
    IDNotRecognisedException {
245         Stage stage = getStageById(stageId);
246         // Retrieves an array of Segment objects
247         Segment[] segments = stage.getSegments();
248         // Method must return an int[] of IDs
249         int[] segmentIds = new int[segments.length];
250         for (int i=0; i<segments.length; i++) {
251             // Adds the ID of each segment to the new array
252             segmentIds[i] = segments[i].getId();
253         }
254         return segmentIds;
255     }
256 
257     @Override
258     public int createTeam(String name, String description)
    throws IllegalNameException, InvalidNameException {
259         // Name validation checks
260         if (name == null) { throw new IllegalNameException("
    Team name cannot be null"); }
261         if (name.isEmpty()) { throw new IllegalNameException("
    Team name cannot be an empty string"); }
262         if (name.length() > 30) { throw new
    IllegalNameException("Team name cannot be greater than 30
    characters"); }
```

```
263         if (name.contains(" ")) { throw new
    InvalidNameException("Team name cannot contain white space"
    ); }
264 
265         for (Team team : teams) {
266             if (team.getName() == name) {
267                 // Loops through each team to check the name
    is not already present
268                 throw new IllegalNameException("Team with name
     \"" + name + "\" already exists");
269             }
270         }
271 
272         // Instantiates new Team and adds it to the list of
    teams
273         Team team = new Team(name, description);
274         teams.add(team);
275         return team.getId();
276     }
277 
278     @Override
279     public void removeTeam(int teamId) throws
    IDNotRecognisedException {
280         // Finds the Team with this ID and removes it from the
     list of teams
281         teams.remove(getTeamById(teamId));
282     }
283 
284     @Override
285     public int[] getTeams() {
286         // Initialises a new int[] to store team IDs
287         int[] teamIds = new int[teams.size()];
288         for (int i=0; i<teams.size(); i++) {
289             // Adds the ID of each team in the teams list to
    teamIds
290             teamIds[i] = teams.get(i).getId();
291         }
292         return teamIds;
293     }
294 
295     @Override
296     public int[] getTeamRiders(int teamId) throws
    IDNotRecognisedException {
297         // Retrieves a Rider[] of all riders in the team
298         Rider[] riders = getTeamById(teamId).getRiders();
299         // Method needs to return an int[] of IDs
300         int[] riderIds = new int[riders.length];
301         for (int i=0; i<riders.length; i++) {
```

```
302             // Adds the ID of each rider to riderIds
303             riderIds[i] = riders[i].getId();
304         }
305         return riderIds;
306     }
307 
308     @Override
309     public int createRider(int teamID, String name, int
    yearOfBirth)
310             throws IDNotRecognisedException,
    IllegalArgumentException {
311         // Input validation checks
312         if (name == null) { throw new IllegalArgumentException
    ("Rider name cannot be null"); }
313         if (yearOfBirth < 1900) {
314             throw new IllegalArgumentException("Rider
    yearOfBirth cannot be less than 1900");
315         }
316 
317         // If arguments are valid, new Rider is instantiated
    and added to the team specified
318         Rider rider = new Rider(name, yearOfBirth);
319         getTeamById(teamID).addRider(rider);
320         return rider.getId();
321     }
322 
323     @Override
324     public void removeRider(int riderId) throws
    IDNotRecognisedException {
325         // Finds the correct team and removes this rider from
    it
326         getTeamByRiderId(riderId).removeRider(getRiderById(
    riderId));
327 
328     }
329 
330     @Override
331     public void registerRiderResultsInStage(int stageId, int
    riderId, LocalTime... checkpoints)
332             throws IDNotRecognisedException,
    DuplicatedResultException, InvalidCheckpointsException,
333             InvalidStageStateException {
334         Stage stage = getStageById(stageId);
335         Rider rider = getRiderById(riderId);
336         // Makes sure stage has finished preparation before
    results are registered
337         if (!stage.isPrepared()) {
338             throw new InvalidStageStateException("Stage is not
```

```
338  'waiting for results'");
339          }
340          // Checkpoints input validation
341          if (checkpoints.length != stage.getSegments().length+2
     ) {
342              throw new InvalidCheckpointsException("Number of
     checkpoints must be number of segments + 2");
343          }
344          // Rider can only have one StageResult per stage
345          if (getResultInStage(rider, stage) != null) {
346              throw new DuplicatedResultException("A result for
     this stage already exists");
347          }
348          // If arguments are valid, new StageResult is
     instantiated storing these checkpoints
349          // and is added to rider's results
350          StageResult stageResult = new StageResult(stage,
     checkpoints);
351          rider.addResult(stageResult);
352      }
353
354      @Override
355      public LocalTime[] getRiderResultsInStage(int stageId, int
      riderId) throws IDNotRecognisedException {
356          // Fetches the StageResult that corresponds to this
     rider and stage
357          StageResult result = getResultInStage(getRiderById(
     riderId), getStageById(stageId));
358          if (result == null) {
359              throw new IDNotRecognisedException("Rider "+
     riderId +" does not have any results in stage "+stageId);
360          } else {
361              // Returns the array of checkpoints for the result
362              return result.getCheckpoints();
363          }
364      }
365
366      @Override
367      public LocalTime getRiderAdjustedElapsedTimeInStage(int
     stageId, int riderId) throws IDNotRecognisedException {
368          Rider riderToFind = getRiderById(riderId);
369          Stage stage = getStageById(stageId);
370          // Retrieves a list of all riders in the stage
371          ArrayList<Rider> ridersInStage = getRidersInStage(
     stage);
372
373          // HashMap to associate riders with elapsed times
374          HashMap<Rider, LocalTime> ridersAndTimes = new HashMap
```

```
374 <>();
375         for (Rider rider : ridersInStage) {
376             // Adds each rider in the stage and their elapsed
    time in the stage to the HashMap
377             ridersAndTimes.put(rider, getElapsedTime(stage,
    rider));
378         }
379 
380         // HashMap is sorted by values (elapsed times)
381         HashMap<Rider, LocalTime> sortedMap =
    sortRidersByTimes(ridersAndTimes);
382 
383         // The HashMap is split into two arrays of riders and
    times
384         Rider[] sortedRidersInStage = sortedMap.keySet().
    toArray(new Rider[ridersInStage.size()]);
385         LocalTime[] sortedTimes = sortedMap.values().toArray(
    new LocalTime[ridersInStage.size()]);
386 
387         // streak represents the number of riders in a row who
     finished the stage with
388         // less than 1 second between them
389         int streak = 0;
390         for (int i=0; i<sortedRidersInStage.length; i++) {
391             // First rider is skipped as it has no previous
    rider
392             if (i > 0) {
393                 LocalTime elapsedTime = sortedTimes[i];
394                 LocalTime prevElapsedTime = sortedTimes[i-1];
395                 assert (elapsedTime.equals(sortedMap.get(
    sortedRidersInStage[i])));
396 
397                 // Calculates the time difference between two
    adjacent elapsed times
398                 long timeDifference = prevElapsedTime.until(
    elapsedTime, ChronoUnit.MILLIS);
399                 assert (timeDifference >= 0);
400 
401                 if (timeDifference < 1000) {
402                     // If the time difference is less than 1
    second (1000ms), streak is incremented
403                     streak++;
404                 } else {
405                     // Otherwise, the streak ends
406                     streak = 0;
407                 }
408             }
409 
```

```
410             if (sortedRidersInStage[i].equals(riderToFind)) {
411                 // When the rider we are looking for is found
    in the loop
412                 if (streak == 0) {
413                     // If there is no streak (previous rider
    was more than 1 second apart),
414                     // the rider's adjusted elapsed time is
    simply the rider's elapsed time
415                     return sortedTimes[i];
416                 }
417                 else {
418                     // If there is a streak, the elapsed time
    of the rider who began the streak
419                     // is returned
420                     return getElapsedTime(stage,
    sortedRidersInStage[i-streak]);
421                 }
422             }
423         }
424         // Return null if given rider does not exist in this
    stage
425         return null;
426     }
427 
428     @Override
429     public void deleteRiderResultsInStage(int stageId, int
    riderId) throws IDNotRecognisedException {
430         // Retrieves the Rider object and the StageResult that
     corresponds to it and this stage
431         Rider rider = getRiderById(riderId);
432         StageResult result = getResultInStage(rider,
    getStageById(stageId));
433         if (result == null) {
434             throw new IDNotRecognisedException("Rider "+
    riderId+" does not have any results in stage "+stageId);
435         } else {
436             // Removes the result if it exists in the stage
437             rider.removeResult(result);
438         }
439     }
440 
441     @Override
442     public int[] getRidersRankInStage(int stageId) throws
    IDNotRecognisedException {
443         // HashMap to associate riders with their adjusted
    elapsed times is initialised
444         HashMap<Rider, LocalTime> riderToResultMap = new
    HashMap<Rider, LocalTime>();
```

```
445         Stage stage = getStageById(stageId);
446 
447         for (Rider rider : getRidersInStage(stage)) {
448             // Finds the StageResult for each rider in this
    stage
449             StageResult result = getResultInStage(rider, stage
    );
450             if (result != null) {
451                 // If the rider is in this stage, their
    adjusted elapsed time is retrieved
452                 // And associated with them in the HashMap
453                 LocalTime elapsedTime =
    getRiderAdjustedElapsedTimeInStage(stageId, rider.getId());
454                 riderToResultMap.put(rider, elapsedTime);
455             }
456         }
457 
458 
459         // The hashmap is sorted so that riders will be order
    of their ranking
460         HashMap<Rider, LocalTime> sortedMap =
    sortRidersByTimes(riderToResultMap);
461 
462         // The method needs to return an int[] of IDs, so one
    is initialised
463         int[] rankedRiders = new int[sortedMap.size()];
464         int i = 0;
465         for (Rider rider : sortedMap.keySet()) {
466             // Each rider's ID from the sorted HashMap is
    added to the int[] in order
467             rankedRiders[i] = rider.getId();
468             i++;
469         }
470         return rankedRiders;
471     }
472 
473     @Override
474     public LocalTime[] getRankedAdjustedElapsedTimesInStage(
    int stageId) throws IDNotRecognisedException {
475         // Retrieves an int[] of ranked rider IDs
476         int[] rankedRiderIds = getRidersRankInStage(stageId);
477         LocalTime[] rankedTimes = new LocalTime[rankedRiderIds
    .length];
478         for (int i=0;i<rankedRiderIds.length;i++) {
479             // Retrieves the adjusted elapsed time for each
    rider in the stage and adds it
480             // to the ranked array of times
481             rankedTimes[i] =
```

```
481 getRiderAdjustedElapsedTimeInStage(stageId, rankedRiderIds[i
    ]);
482         }
483         return rankedTimes;
484     }
485 
486     @Override
487     public int[] getRidersPointsInStage(int stageId) throws
    IDNotRecognisedException {
488         Stage stage = getStageById(stageId);
489         // Retrieves a list of rider IDs in order of their
    rank
490         int[] rankedRiders = getRidersRankInStage(stageId);
491         StageType type = stage.getType();
492         // Initialises a new array to contain points for each
    rider
493         int[] points = new int[rankedRiders.length];
494         for (int i=0;i<points.length;i++) {
495             // i represents the current rider in the loop's
    ranking
496             if (i > 14) {
497                 // If the rider ranked 16th or more, they get
    no points
498                 points[i] = 0;
499             } else {
500                 // Looks up points table attribute to assign
    points
501                 switch (type) {
502                     case FLAT:
503                         points[i] = pointsTable[0][i];
504                         break;
505                     case MEDIUM_MOUNTAIN:
506                         points[i] = pointsTable[1][i];
507                         break;
508                     default: // HIGH_MOUNTAIN or TT
509                         points[i] = pointsTable[2][i];
510                         break;
511                 }
512             }
513 
514             // Calculates and adds the points aquired from
    immediate sprints in the stage
515             points[i] += getImmediateSprintPoints(getRiderById
    (rankedRiders[i]), stage);
516         }
517         return points;
518     }
519 
```

```
520     @Override
521     public int[] getRidersMountainPointsInStage(int stageId)
    throws IDNotRecognisedException {
522         Stage stage = getStageById(stageId);
523         Segment[] segments = stage.getSegments();
524         // Ranked list of rider IDs in the stage
525         int[] ridersRanks = getRidersRankInStage(stageId);
526         // Ranked list of Rider objects in the stage
527         Rider[] ridersInStage = new Rider[ridersRanks.length];
528         for (int i=0;i<ridersInStage.length;i++) {
529             // Adds the Rider object for each ID to the array
530             ridersInStage[i] = getRiderById(ridersRanks[i]);
531         }
532 
533         // Initalises an array of points for each rider in the
     stage, starting with 0 for all
534         int[] mountainPoints = new int[ridersInStage.length];
535         Arrays.fill(mountainPoints, 0);
536 
537         for (int i=0;i<segments.length;i++) {
538             SegmentType type = segments[i].getType();
539             if (type == SegmentType.SPRINT) {
540                 // Sprint segments are ignored when
    calculating points
541                 continue;
542             }
543 
544             // HashMap to associate riders with their segment
    time for this segment
545             HashMap<Rider, LocalTime> resultToTimeMap = new
    HashMap<Rider, LocalTime>();
546 
547             for (Rider rider : ridersInStage) {
548                 StageResult result = getResultInStage(rider,
    stage);
549                 if (result != null) {
550                     // Segment time is the time a rider
    reaches the segment
551                     LocalTime[] checkpoints = result.
    getCheckpoints();
552                     assert (checkpoints.length == segments.
    length + 2);
553                     LocalTime segmentTime = timeDifference(
    checkpoints[0], checkpoints[i+1]);
554                     resultToTimeMap.put(rider, segmentTime);
555                 }
556             }
557 
```

```
558             // Sorts this segments HashMap based on values (
    segment times)
559             HashMap<Rider, LocalTime> sortedMap =
    sortRidersByTimes(resultToTimeMap);
560 
561             // The rank of the current rider in this segment
    for each rider
562             int a = 0;
563             for (Rider rider : sortedMap.keySet()) {
564                 // Only checks the first 8 riders in the
    segment, as the others will
565                 // recieve no points for this segment
566                 if (a > 7) { break; }
567 
568                 for (int b=0;b<ridersInStage.length;b++) {
569                     // b represents the position of the rider
    in the final points array
570                     if (ridersInStage[b].equals(rider)) {
571                         // Where the rider in the segment
    matches up with the rider in the stage,
572                         // the rider's total points is
    incremented by the points for this segment
573                         // which is looked up in the points
    table, based on rank and segment type
574                         switch (type) {
575                             case C4:
576                                 mountainPoints[b] +=
    mountainPointsTable[a][0];
577                                 break;
578                             case C3:
579                                 mountainPoints[b] +=
    mountainPointsTable[a][1];
580                                 break;
581                             case C2:
582                                 mountainPoints[b] +=
    mountainPointsTable[a][2];
583                                 break;
584                             case C1:
585                                 mountainPoints[b] +=
    mountainPointsTable[a][3];
586                                 break;
587                             case HC:
588                                 mountainPoints[b] +=
    mountainPointsTable[a][4];
589                                 break;
590                             default:
591                                 assert (false);
592                         }
```

```
593                         break;
594                     }
595                 }
596                 a++;
597             }
598         }
599         return mountainPoints;
600     }
601 
602     @Override
603     public void eraseCyclingPortal() {
604         // Resets all static counter attributes so that IDs
    start from 0 again
605         Team.resetNoOfTeams();
606         Rider.resetNoOfRiders();
607         Race.resetNoOfRaces();
608         Stage.resetNoOfStages();
609         Segment.resetNoOfSegments();
610         StageResult.resetTotalResults();
611         // Clears list of teams and races in CyclingPortal
612         teams.clear();
613         races.clear();
614     }
615 
616     @Override
617     public void saveCyclingPortal(String filename) throws
    IOException {
618         // ObjectOutputStream can serialise an object and
    write it to a file
619         ObjectOutputStream oos = new ObjectOutputStream(new
    FileOutputStream(filename));
620         try {
621             // Serialises this CyclingPortal object
622             oos.writeObject(this);
623         } finally {
624             // ObjectOutputStream must close regardless of if
    write is successful
625             oos.close();
626         }
627     }
628 
629     @Override
630     public void loadCyclingPortal(String filename) throws
    IOException, ClassNotFoundException {
631         // ObjectInputStream can read a serialised file and
    deserialise the object
632         ObjectInputStream ois = new ObjectInputStream(new
    FileInputStream(filename));
```

```
633         try {
634             // Reads the serialised object and deserialises it
635             Object obj = ois.readObject();
636             CyclingPortal cyclingPortal;
637             if (obj instanceof CyclingPortal) {
638                 cyclingPortal = (CyclingPortal)obj;
639                 // Replaces this object attributes with those
    of loaded CyclingPortal
640                 teams = cyclingPortal.getTeamsList();
641                 races = cyclingPortal.getRacesList();
642             }
643         } finally {
644             // ObjectInputStream must close regardless of if
    read is successful
645             ois.close();
646         }
647     }
648 
649     @Override
650     public void removeRaceByName(String name) throws
    NameNotRecognisedException {
651         for (Race race : races) {
652             // Searches through each race until a matching
    name is found
653             if (race.getName().equals(name)) {
654                 // Removes this race from the list
655                 races.remove(race);
656                 // Exits the method so that the for loop does
    not continue
657                 return;
658             }
659         }
660         throw new NameNotRecognisedException("No race exists
    with name " + name);
661     }
662 
663     @Override
664     public LocalTime[] getGeneralClassificationTimesInRace(int
     raceId) throws IDNotRecognisedException {
665         Race race = getRaceById(raceId);
666         Stage[] stages = race.getStages();
667         // Retrieves all riders participating in this race
668         ArrayList<Rider> riders = getRidersInRace(race);
669         // Initialises a HashMap to associate riders with
    their elapsed times
670         HashMap<Rider, LocalTime> riderTimes = new HashMap<
    Rider, LocalTime>();
671         for (Rider rider : riders) {
```

```
672             // Every participating rider is added to the
    HashMap
673             riderTimes.put(rider, LocalTime.of(0, 0, 0));
674 
675             for (Stage stage : stages) {
676                 // The rider's elapsed time for each stage is
    found
677                 LocalTime t =
    getRiderAdjustedElapsedTimeInStage(stage.getId(),   rider.
    getId());
678                 // If this result exists (rider has finished
    the stage), HashMap
679                 // value for the rider is incremented by the
    elapsed time for this stage
680                 if (t != null) {
681                     riderTimes.replace(rider, riderTimes.get(
    rider).plusHours(t.getHour())
682                                                             .
    plusMinutes(t.getMinute())
683                                                             .
    plusSeconds(t.getSecond())
684                                                             .
    plusNanos(t.getNano()));
685                 }
686             }
687         }
688 
689         // The HashMap is then sorted by values (total elapsed
     times)
690         HashMap<Rider, LocalTime> sortedMap =
    sortRidersByTimes(riderTimes);
691 
692         // The method needs to return a LocalTime[], so one is
     initialised
693         LocalTime[] times = new LocalTime[riders.size()];
694         // Sorted classification times are added to this array
695         sortedMap.values().toArray(times);
696         return times;
697     }
698 
699     @Override
700     public int[] getRidersPointsInRace(int raceId) throws
    IDNotRecognisedException {
701         Race race = getRaceById(raceId);
702         Stage[] stages = race.getStages();
703         // Retrieves all riders participating in this race
704         ArrayList<Rider> riders = getRidersInRace(race);
705         // Initialises a HashMap to associate riders with
```

```
705 their points
706         Map<Rider, Integer> riderPoints = new HashMap<Rider,
    Integer>();
707 
708         for (Rider rider : riders) {
709             // Every participating rider is added to the
    HashMap
710             riderPoints.put(rider, 0);
711             for (Stage stage : stages) {
712                 // Loops through every stage in the race and
    retrieves an array of
713                 // ranked rider IDs
714                 int[] ranks = getRidersRankInStage(stage.getId
    ());
715 
716                 // Finds the index of the current rider in
    this array
717                 int indexOfRider = -1;
718                 for (int i=0; i<ranks.length; i++) {
719                     if (ranks[i] == rider.getId()) {
720                         indexOfRider = i;
721                     }
722                 }
723 
724                 if (indexOfRider != -1) {
725                     // Retrieves an array of ranked rider's
    points in the stage
726                     // Note: this is sorted by time and so
    items in this array and the
727                     //       ranks array will match up with
    each other
728                     int[] pointsArr = getRidersPointsInStage(
    stage.getId());
729                     int points = pointsArr[indexOfRider];
730 
731                     // Increments the rider's total points by
    the points in this stage
732                     riderPoints.replace(rider, riderPoints.get
    (rider) + points);
733                 }
734             }
735         }
736 
737         // The method needs to return an int[] sorted by
    elapsed time, so one is initialised
738         int[] sortedPoints = new int[riders.size()];
739         // Retrieves array of rider IDs sorted by elapsed time
740         int[] riderRanks = getRidersGeneralClassificationRank(
```

```
740 raceId);
741         for ( int i=0; i<riderRanks.length; i++ ) {
742             // Finds the rider for every ID in the array of
    ranks and adds its corresponding
743             // number of points in the HashMap to the array of
     sorted points
744             sortedPoints[i] = riderPoints.get(getRiderById(
    riderRanks[i]));
745         }
746         return sortedPoints;
747     }
748 
749     @Override
750     public int[] getRidersMountainPointsInRace(int raceId)
    throws IDNotRecognisedException {
751         Race race = getRaceById(raceId);
752         Stage[] stages = race.getStages();
753         ArrayList<Rider> riders = getRidersInRace(race);
754         // HashMap to associatie riders in the race and their
    total points in the race
755         HashMap<Rider, Integer> riderPoints = new HashMap<
    Rider, Integer>();
756 
757         for (Rider rider : riders) {
758             // Adds each rider and an inital value of 0 points
     to the HashMap
759             riderPoints.put(rider, 0);
760             for (Stage stage : stages) {
761                 if (getResultInStage(rider, stage) != null) {
762                     // For each stage in the race, retrieves
    the riders ranks if they are in the stage
763                     // (riders in a race should have results
    in all stages in the race)
764                     int[] ranks = getRidersRankInStage(stage.
    getId());
765 
766                     // The index of the current rider in the
    array of ranks for the stage
767                     int indexOfRider = -1;
768                     for (int i=0; i<ranks.length; i++) {
769                         if (ranks[i] == rider.getId()) {
770                             // When the ID in ranks that
    matches with this current rider is found,
771                             // the index is assigned and the
    loop is broken out of
772                             indexOfRider = i;
773                             break;
774                         }
```

```
775                     }
776
777                     if (indexOfRider != -1) {
778                         // Finds all the riders' mountain
    points in this stage
779                         int[] pointsArr =
    getRidersMountainPointsInStage(stage.getId());
780                         int points = 0;
781                         if (pointsArr.length > 0) {
782                             // The current rider's points in
    this stage are found
783                             points = pointsArr[indexOfRider];
784                         }
785                         // Rider's total points for the race
    is incremented by their points in this stage
786                         riderPoints.replace(rider, riderPoints
    .get(rider) + points);
787                     }
788                 }
789             }
790         }
791
792         // Method needs to return an int[] of points, so one
    is initialised
793         int[] sortedPoints = new int[riders.size()];
794         // Points need to be sorted by total elapsed times, so
     these are retrieved
795         int[] riderRanks = getRidersGeneralClassificationRank(
    raceId);
796         for ( int i=0; i<riderRanks.length; i++ ) {
797             // Adds the points of each rider to the points
    array,
798             // ordered by general classification rank
799             sortedPoints[i] = riderPoints.get(getRiderById(
    riderRanks[i]));
800         }
801         return sortedPoints;
802     }
803
804     @Override
805     public int[] getRidersGeneralClassificationRank(int raceId
    ) throws IDNotRecognisedException {
806         Race race = getRaceById(raceId);
807         Stage[] stages = race.getStages();
808         ArrayList<Rider> riders = getRidersInRace(race);
809         // HashMap to associate riders with their total
    adjusted elapsed times
810         HashMap<Rider, LocalTime> riderTimes = new HashMap<
```

```
Rider, LocalTime>();

        for (Rider rider : riders) {
            // Adds each rider in the race to the HashMap, initialising their time as 0:0:0
            riderTimes.put(rider, LocalTime.of(0, 0, 0));
            for (Stage stage : stages) {
                // Calculates the adjusted elapsed time for each stage for this rider
                LocalTime t = getRiderAdjustedElapsedTimeInStage(stage.getId(), rider.getId());
                if (t != null) {
                    // If the rider has a result in this stage, their total time is incremented
                    // by the adjusted elapsed time in this stage
                    riderTimes.replace(rider, riderTimes.get(rider).plusHours(t.getHour())
                                                                    .plusMinutes(t.getMinute())
                                                                    .plusSeconds(t.getSecond()));
                }
            }
        }

        // The HashMap is sorted by values (total elapsed times)
        HashMap<Rider, LocalTime> sortedMap = sortRidersByTimes(riderTimes);

        // The method needs to return an int[] of IDs, so one is initialised
        int[] riderIds = new int[riders.size()];
        int i = 0;
        for (Rider key : sortedMap.keySet()) {
            // The ID of each rider in the sorted HashMap is added to the int[] in order
            riderIds[i] = key.getId();
            i++;
        }
        return riderIds;
    }

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

```
844         // Retrieves race's ranking and points sorted by time
845         // This is so that the array of times aligns with the
    array of rider IDs
846         // Therefore, they can be paired up into a HashMap
847         int[] riderRanks = getRidersGeneralClassificationRank(
    raceId);
848         int[] riderPointsSortedByTime = getRidersPointsInRace(
    raceId);
849         assert (riderRanks.length == riderPointsSortedByTime.
    length);
850         HashMap<Rider, Integer> riderPoints = new HashMap<>();
851 
852         for (int i=0; i<riderRanks.length;i++) {
853             // Each rider and their corresponding points are
    put into the HashMap
854             riderPoints.put(getRiderById(riderRanks[i]),
855                     riderPointsSortedByTime[i]);
856         }
857 
858         // The HashMap is then sorted by number of points (
    sorted by value)
859         HashMap<Rider, Integer> sortedMap = sortRidersByPoints
    (riderPoints);
860         assert (sortedMap.size() == riderPoints.size());
861 
862         // The method needs to return an int[], so one is
    initialised
863         int[] riderIds = new int[riderRanks.length];
864         int i = 0;
865         for ( Rider key : sortedMap.keySet() ) {
866             // Finds each key in the sorted HashMap (the rider
    ), and adds its ID to the int[]
867             riderIds[i] = key.getId();
868             i++;
869         }
870 
871         // The array needs to be in descending order, so it is
     reversed
872         reverseArray(riderIds);
873         return riderIds;
874     }
875 
876     @Override
877     public int[] getRidersMountainPointClassificationRank(int
    raceId) throws IDNotRecognisedException {
878         // These two arrays associate rider IDs with their
    mountain points in the race
879         int[] riderRanks = getRidersGeneralClassificationRank(
```

```
879 raceId);
880         int[] riderPointsSortedByTime =
    getRidersMountainPointsInRace(raceId);
881         assert (riderRanks.length == riderPointsSortedByTime.
    length);
882 
883         // HashMap to associate riders with their total
    mountain points in the race
884         HashMap<Rider, Integer> riderPoints = new HashMap<>();
885 
886         for (int i=0; i<riderRanks.length;i++) {
887             // Adds each rider in the race and their
    associated mountain points to the HashMap
888             riderPoints.put(getRiderById(riderRanks[i]),
    riderPointsSortedByTime[i]);
889         }
890 
891         // The HashMap is sorted by values (total mountain
    points)
892         HashMap<Rider, Integer> sortedMap = sortRidersByPoints
    (riderPoints);
893 
894         // The method needs to return an int[] of rider IDs,
    so one is initialised
895         int[] riderIds = new int[riderRanks.length];
896         int i = 0;
897         for ( Rider key : sortedMap.keySet() ) {
898             // Finds each key in the sorted HashMap (the rider
    ), and adds its ID to the int[]
899             riderIds[i] = key.getId();
900             i++;
901         }
902 
903         // The array needs to be in descending order, so it is
     reversed
904         reverseArray(riderIds);
905         return riderIds;
906     }
907 
908     /**
909      * Public getter method to return a list of all teams
    stored in the
910      * 'teams' ArrayList.
911      *
912      * @return An ArrayList of team objects.
913      *
914      */
915     public ArrayList<Team> getTeamsList() {
```

```
916         return teams;
917     }
918 
919     /**
920      * Public getter method to return a list of all races
    stored in the
921      * 'races' ArrayList.
922      *
923      * @return An ArrayList of race objects.
924      *
925      */
926     public ArrayList<Race> getRacesList() {
927         return races;
928     }
929 
930     /**
931      * Private method to find the time difference between two
    local times,
932      * outputting the result as a LocalTime object.
933      *
934      * @param time1 LocalTime object.
935      * @param time2 LocalTime object.
936      * @return LocalTime object, in format HH:MM:SS:NN,
    representing
937      * difference between time1 and time2.
938      *
939      */
940     private LocalTime timeDifference(LocalTime time1,
    LocalTime time2) {
941         // Works out time difference in milliseconds
942         long elapsedTimeInMilliSecs = time1.until(time2,
    ChronoUnit.MILLIS);
943 
944         // Conversion of ms into hours, mins, secs, nanos
945         int hours = (int) elapsedTimeInMilliSecs / 3600000;
946         int mins = (int) (elapsedTimeInMilliSecs % 3600000) /
    60000;
947         int secs = (int) ((elapsedTimeInMilliSecs % 3600000
    ) % 60000) / 1000;
948         int nanos = (int) (((elapsedTimeInMilliSecs % 3600000
    ) % 60000) % 1000) * 1000000;
949 
950         // Returns a LocalTime object
951         return LocalTime.of(hours, mins, secs, nanos);
952     }
953 
954     /**
955      * Private method to sort a HashMap of Rider : LocalTime
```

```
955 by their
956      * adjusted elapsed times (LocalTime).
957      *
958      * @param initialMap The HashMap to be sorted.
959      * @return A HashMap object sorted by the LocalTime value
    (descending).
960      *
961      */
962     private HashMap<Rider, LocalTime> sortRidersByTimes(
    HashMap<Rider, LocalTime> initialMap) {
963         return initialMap.entrySet().stream()
964                 .sorted(Entry.comparingByValue())
965                 .collect(Collectors.toMap(Entry::getKey,
    Entry::getValue,
966                         (e1, e2) -> e1, LinkedHashMap::new));
967     }
968
969     /**
970      * Private method to sort a HashMap of Rider : Integer by
    the rider's
971      * points (integer).
972      *
973      * @param initialMap The HashMap to be sorted.
974      * @return A HashMap object sorted by the integer value (
    descending).
975      *
976      */
977     private HashMap<Rider, Integer> sortRidersByPoints(
    HashMap<Rider, Integer> initialMap) {
978         return initialMap.entrySet().stream()
979                 .sorted(Entry.comparingByValue())
980                 .collect(Collectors.toMap(Entry::getKey,
    Entry::getValue,
981                         (e1, e2) -> e1, LinkedHashMap::new));
982     }
983
984     /**
985      * Private method to reverse an int[], used to sort rider
    IDs by descending order.
986      *
987      * @param array The array to reverse.
988      */
989     private void reverseArray(int[] array) {
990         for (int i = 0; i < array.length / 2; i++) {
991             // Swaps first and last elements in the array and
    gradually moves inwards
992             // swapping elements until meeting at the middle
993             int temp = array[i];
```

```
994             array[i] = array[array.length - 1 - i];
995             array[array.length - 1 - i] = temp;
996         }}
997
998     /**
999      * Private method to find the StageResult object for a
    particular rider and stage.
1000     *
1001     * @param rider The rider to look for the result in.
1002     * @param stage The stage to look up the result for.
1003     * @return A StageResult object representing the rider's
    result in the stage, or
1004     *         null if the rider does not have a result in
    the stage.
1005     *
1006     */
1007    private StageResult getResultInStage(Rider rider, Stage
    stage) {
1008        for (StageResult result : rider.getResults()) {
1009            if (result.getStage().equals(stage)) {
1010                // A StageResult with the correct Stage was
    found, returns the StageResult
1011                return result;
1012            }
1013        }
1014        // A StageResult with the correct Stage was not found
    , returns null
1015        return null;
1016    }
1017
1018    /**
1019     * Private method to calculate the points a rider
    obtained from
1020     * immediate sprints in a stage.
1021     *
1022     * @param riderToFind The rider in question.
1023     * @param stage The stage in question.
1024     * @return An int representing the number of points
    obtained in the stage.
1025     */
1026    private int getImmediateSprintPoints(Rider riderToFind,
    Stage stage) {
1027        int totalPoints = 0;
1028        Segment[] segments = stage.getSegments();
1029        ArrayList<Rider> ridersInStage = getRidersInStage(
    stage);
1030
1031        for (int i=0; i<segments.length; i++) {
```

```
1032            if (!segments[i].getType().equals(SegmentType.
    SPRINT)) {
1033                // Ignores segments that aren't immediate
    sprints
1034                continue;
1035            }
1036
1037            // HashMap to associate riders with their times
    for this segment
1038            HashMap<Rider, LocalTime> riderTimesAtSegment =
    new HashMap<>();
1039
1040            for (Rider rider : ridersInStage) {
1041                StageResult result = getResultInStage(rider,
    stage);
1042                if (result != null) {
1043                    // Gets the rider's checkpoints in this
    stage
1044                    LocalTime[] checkpoints = result.
    getCheckpoints();
1045                    assert (checkpoints.length == segments.
    length + 2);
1046
1047                    // Calculates time this segment was
    reached and associates it with the rider in the HashMap
1048                    LocalTime timeAtSegment = timeDifference(
    checkpoints[0], checkpoints[i+1]);
1049                    riderTimesAtSegment.put(rider,
    timeAtSegment);
1050                }
1051            }
1052
1053            // HashMap is sorted by values (segment time)
1054            HashMap<Rider, LocalTime> sortedMap =
    sortRidersByTimes(riderTimesAtSegment);
1055
1056            // List to contain all riders in the stage,
    sorted by time they reached this segment
1057            ArrayList<Rider> riderRankingsAtSegment = new
    ArrayList<Rider>();
1058            for (Rider key : sortedMap.keySet()) {
1059                // Adds each rider in the HashMap to the list
     in order
1060                riderRankingsAtSegment.add(key);
1061            }
1062
1063            // The ranking of this rider is the index they
    have in the ranked list
```

```
1064            int riderRanking = riderRankingsAtSegment.indexOf
    (riderToFind);
1065
1066            if (riderRanking < 15) {
1067                // Looks up table to find points to add for
    the rider
1068                // Adds no points if the rider reaches the
    segment 16th or more
1069                totalPoints += pointsTable[2][riderRanking];
1070            }
1071
1072        }
1073        return totalPoints;
1074    }
1075
1076    /**
1077     * Private method to find a Race object based on its
    unique ID.
1078     * <p>
1079     *     Iterates through the 'races' ArrayList, checking
    each race's ID
1080     *     through the 'getId()' getter method, until the
    race matching the ID
1081     *     is found.
1082     * </p>
1083     *
1084     * @param id The ID of the race to be found.
1085     * @return A Race object, corresponding to the unique ID
    provided.
1086     * @throws IDNotRecognisedException If the ID does not
    match any Race in
1087     * the system.
1088     *
1089     */
1090    private Race getRaceById(int id) throws
    IDNotRecognisedException {
1091        for (Race race : races) {
1092            if (race.getId() == id) {
1093                return race;
1094            }
1095        }
1096        throw new IDNotRecognisedException("No race with an
    ID of " + id + " exists");
1097    }
1098
1099    /**
1100     * Private method to find a Stage object based on its
    unique ID.
```

```
1101      * <p>
1102      *     Iterates through the 'races' ArrayList, and again
     through the
1103      *     race's Stages (through the 'getStages()' getter
     method), until the
1104      *     Stage matching the ID is found.
1105      * </p>
1106      *
1107      * @param id The ID of the stage to be found.
1108      * @return A Stage object, corresponding to the unique ID
      provided.
1109      * @throws IDNotRecognisedException If the ID does not
     match any Stage in
1110      * the system.
1111      *
1112      */
1113     private Stage getStageById(int id) throws
     IDNotRecognisedException {
1114         for (Race race : races) {
1115             for (Stage stage : race.getStages()) {
1116                 // Loops through every Stage in the
     CyclingPortal
1117                 // If an ID match is found, returns this
     Stage
1118             if (stage.getId() == id) {
1119                 return stage;
1120             }
1121         }
1122         }
1123         throw new IDNotRecognisedException("No stage with an
     ID of " + id + " exists");
1124     }
1125
1126     /**
1127      * Private method to find a Segment object based on its
     unique ID.
1128      * <p>
1129      *     Iterates through the 'races' ArrayList, and again
     through the
1130      *     race's Stages (through the 'getStages()' getter
     method), and again
1131      *     through the stage's Segments (through the '
     getSegments()' getter
1132      *     method), until the Segment matching the ID is
     found.
1133      * </p>
1134      *
1135      * @param id The ID of the segment to be found.
```

```
1136      * @return A Segment object, corresponding to the unique
     ID provided.
1137      * @throws IDNotRecognisedException If the ID does not
     match any Segment
1138      * in the system.
1139      *
1140      */
1141     private Segment getSegmentById(int id) throws
     IDNotRecognisedException {
1142         for (Race race : races) {
1143             for (Stage stage : race.getStages()) {
1144                 for (Segment segment : stage.getSegments()) {
1145                     // Loops through every Segment in the
     CyclingPortal
1146                     // If an ID match is found, returns this
     Segment
1147                     if (segment.getId() == id) {
1148                         return segment;
1149                     }
1150                 }
1151             }
1152         }
1153         throw new IDNotRecognisedException("No segment with
     an ID of " + id + " exists");
1154     }
1155
1156     /**
1157      * Private method to find a Team object based on its
     unique ID.
1158      * <p>
1159      *     Iterates through the 'teams' ArrayList, until the
     Team matching
1160      *     the ID is found.
1161      * </p>
1162      *
1163      * @param id The ID of the team to be found.
1164      * @return A Team object, corresponding to the unique ID
     provided.
1165      * @throws IDNotRecognisedException If the ID does not
     match any Team in
1166      * the system.
1167      *
1168      */
1169     private Team getTeamById(int id) throws
     IDNotRecognisedException {
1170         for (Team team : teams) {
1171             // Loops through every Team in the CyclingPortal
1172             // If an ID match is found, returns this Team
```

```
1173                if (team.getId() == id) {
1174                    return team;
1175                }
1176            }
1177            throw new IDNotRecognisedException("No team with an
    ID of " + id + " exists");
1178        }
1179
1180        /**
1181         * Private method to find a Rider object based on its
    unique ID.
1182         * <p>
1183         *     Iterates through the 'teams' ArrayList, and again
    through the
1184         *     team's riders (through the 'getRiders()' getter
    method), until the
1185         *     Rider matching the ID is found.
1186         * </p>
1187         *
1188         * @param id The ID of the rider to be found.
1189         * @return A Rider object, corresponding to the unique ID
     provided.
1190         * @throws IDNotRecognisedException If the ID does not
    match any Rider in
1191         * the system.
1192         *
1193         */
1194        private Rider getRiderById(int id) throws
    IDNotRecognisedException {
1195            for (Team team : teams) {
1196                for (Rider rider : team.getRiders()) {
1197                    // Loops through every Rider in the
    CyclingPortal
1198                    // If an ID match is found, returns this
    Rider
1199                    if (rider.getId() == id) {
1200                        return rider;
1201                    }
1202                }
1203            }
1204            throw new IDNotRecognisedException("No rider with an
    ID of " + id + " exists");
1205        }
1206
1207        /**
1208         * Private method to find a Race object based on the ID
    of a Stage object
1209         * inside the race.
```

```
1210      * <p>
1211      *     Iterates through the 'races' ArrayList, and again
     through the
1212      *     race's stages (through the 'getStages()' getter
     method), until a
1213      *     Stage matching the ID is found.
1214      * </p>
1215      *
1216      * @param id The ID of the stage contained in the race to
      be found.
1217      * @return A Race object, corresponding to the Stage ID
     provided.
1218      * @throws IDNotRecognisedException If the ID does not
     match any Stages
1219      * in the system.
1220      *
1221      */
1222     private Race getRaceByStageId(int id) throws
     IDNotRecognisedException {
1223         for (Race race : races) {
1224             for (Stage stage : race.getStages()) {
1225                 // Loops through every Stage in the
     CyclingPortal
1226                 // If an ID match is found, returns the Race
     this Stage is in
1227             if (stage.getId() == id) {
1228                 return race;
1229             }
1230         }
1231         }
1232         throw new IDNotRecognisedException("No stage with an
     ID of " + id + " exists");
1233     }
1234
1235     /**
1236      * Private method to find a Stage object based on the ID
     of a Segment object
1237      * inside the Stage.
1238      * <p>
1239      *     Iterates through the 'races' ArrayList, and again
     through the
1240      *     race's stages (through the 'getStages()' getter
     method), and again
1241      *     through the stage's segments (through the '
     getSegments()' getter
1242      *     method) until a
1243      *     Segment matching the ID is found.
1244      * </p>
```

```
1245      *
1246      * @param id The ID of the segment contained in the stage
      to be found.
1247      * @return A Stage object, corresponding to the Segment
     ID provided.
1248      * @throws IDNotRecognisedException If the ID does not
     match any Segments
1249      * in the system.
1250      *
1251      */
1252     private Stage getStageBySegmentId(int id) throws
     IDNotRecognisedException {
1253         for (Race race : races) {
1254             for (Stage stage : race.getStages()) {
1255                 for (Segment segment : stage.getSegments()) {
1256                     // Loops through every Segment in the
     CyclingPortal
1257                     // If an ID match is found, returns the
     Stage this Segment is in
1258                     if (segment.getId() == id) {
1259                         return stage;
1260                     }
1261                 }
1262             }
1263         }
1264         throw new IDNotRecognisedException("No segment with
     an ID of " + id + " exists");
1265     }
1266
1267     /**
1268      * Private method to find a Team object based on the ID
     of a Rider object
1269      * inside the Team.
1270      * <p>
1271      *     Iterates through the 'teams' ArrayList, and again
     through the
1272      *     team's riders (through the 'getRiders()' getter
     method), until a
1273      *     Rider matching the ID is found.
1274      * </p>
1275      *
1276      * @param id The ID of the rider contained in the team to
      be found.
1277      * @return A Team object, corresponding to the Rider ID
     provided.
1278      * @throws IDNotRecognisedException If the ID does not
     match any Riders
1279      * in the system.
```

```
1280      *
1281      */
1282     private Team getTeamByRiderId(int id) throws
     IDNotRecognisedException {
1283         for (Team team : teams) {
1284             for (Rider rider : team.getRiders()) {
1285                 // Loops through every Rider in the
     CyclingPortal
1286                 // If an ID match is found, returns the Team
     this Rider is in
1287                 if (rider.getId() == id) {
1288                     return team;
1289                 }
1290             }
1291         }
1292         throw new IDNotRecognisedException("No rider with an
     ID of " + id + " exists");
1293     }
1294
1295     /**
1296      * Private method to calculate the elapsed time a rider
     took in a stage.
1297      *
1298      * @param stage The stage in question.
1299      * @param rider The particular rider who's elapsed time
     we want to find.
1300      * @return A LocalTime object in the form HH:MM:SS:nn
     which represents
1301      * the elapsed time the rider took to complete the stage.
1302      *
1303      */
1304     private LocalTime getElapsedTime(Stage stage, Rider rider
     ) {
1305         // Finds the appropriate StageResult object
1306         StageResult result = getResultInStage(rider, stage);
1307         // If this StageResult exists
1308         if (result != null) {
1309             LocalTime[] checkpoints = result.getCheckpoints
     ();
1310             // Returns the difference in time between the
     first and last checkpoint
1311             // i.e. between the start and finish of the stage
1312             return timeDifference(checkpoints[0], checkpoints
     [checkpoints.length-1]);
1313         } else {
1314             return null;
1315         }
1316     }
```

```
1317
1318     /**
1319      * Private method to return an ArrayList of riders in a
     particular stage.
1320      *
1321      * @param stage The stage in question.
1322      * @return An ArrayList of Rider objects, all of which
     have results for
1323      * the stage in question.
1324      *
1325      */
1326     private ArrayList<Rider> getRidersInStage(Stage stage) {
1327         // Initalises a list to hold the riders in the stage
1328         ArrayList<Rider> riders = new ArrayList<>();
1329         for (Team team : teams) {
1330             for (Rider rider : team.getRiders()) {
1331                 // Loops through every Rider in the
     CyclingPortal
1332                 // If the Rider has a StageResult for this
     Stage, it is added to the list
1333                 if (getResultInStage(rider, stage) != null) {
1334                     riders.add(rider);
1335                 }
1336             }
1337         }
1338         return riders;
1339     }
1340
1341     /**
1342      * Private method to return an ArrayList of riders in a
     particular race.
1343      *
1344      * @param race The race in question.
1345      * @return An ArrayList of Rider objects, all of which
     have results for
1346      * the race in question.
1347      *
1348      */
1349     private ArrayList<Rider> getRidersInRace(Race race) {
1350         // Initalises a list to hold the riders in the race
1351         ArrayList<Rider> ridersInRace = new ArrayList<Rider
     >();
1352         for (Stage stage : race.getStages()) {
1353             // Forms a list of riders in each stage
1354             ArrayList<Rider> ridersInStage = getRidersInStage
     (stage);
1355             for (Rider rider : ridersInStage) {
1356                 // Adds any new rider found into the final
```

```
1356 list
1357                 if (!ridersInRace.contains(rider)) {
1358                     ridersInRace.add(rider);
1359                 }
1360             }
1361         }
1362         return ridersInRace;
1363     }
1364 }
1365
```

File - Race.java

```java
package cycling;

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

/**
 * Race class.<br>
 * Represents a single race in the cycling competition.
 *
 * @author Joey Griffiths and Alexander Cairns
 *
 */
class Race implements Serializable {

    /**
     * The number of instances of Race, automatically incremented when the
     * constructor is called.<br>
     * Used for allocation of IDs.
     */
    private static int numberOfRaces = 0;

    /**
     * The ID of the Race.
     */
    private final int id;

    /**
     * The name of the race.
     */
    private final String name;

    /**
     * The race description.
     */
    private final String description;

    /**
     * The list of stages included in the race.
     */
    private ArrayList<Stage> stages = new ArrayList<>();

    /**
     * Race class constructor. Initialises a new race with a name and description,
     * and automatically assigns an ID using the number of instances of Race.
     *
```

```
46      * @param name The name of the race.
47      * @param description The race description.
48      *
49      */
50     Race(String name, String description) {
51         this.name = name;
52         this.description = description;
53         id = numberOfRaces++;
54     }
55 
56     /**
57      * Resets the static variable numberOfRaces.<br>
58      * Used to reset the CyclingPortal so that IDs start from 0
    again.
59      */
60     public static void resetNoOfRaces() {
61         numberOfRaces = 0;
62     }
63 
64     /**
65      * Adds a stage to the race.
66      * @param stage A Stage object to add to the race's stages.
67      */
68     public void addStage(Stage stage) {
69         stages.add(stage);
70         assert (stages.size() > 0);
71     }
72 
73     /**
74      * Removes a stage from the race.
75      * @param stage The Stage object to remove from the race's
   stages.
76      */
77     public void removeStage(Stage stage) {
78         stages.remove(stage);
79     }
80 
81     /**
82      * Get the details of this particular Race.
83      * @return A formatted string containing details about the
   race ID, name,
84      *         description, number of stages, and total length.
85      */
86     public String getDetails() {
87         double totalLength = getTotalLength();
88         return "ID: "+id+" | Name: "+name+" | Description: "+
   description
89                 +" | No. of Stages: "+stages.size()
```

```
90                  +" | Total Length: "+totalLength;
91      }
92
93      /**
94       * Method to return the ID of a race.
95       * @return The ID of the race.
96       */
97      public int getId() {
98          return id;
99      }
100
101     /**
102      * Method to return the name of a race.
103      * @return The name of the race.
104      */
105     public String getName() {
106         return name;
107     }
108
109     /**
110      * Method to return the description of a race.
111      * @return The race description.
112      */
113     public String getDesc() {
114         return description;
115     }
116
117     /**
118      * Method to return the number of stages in a race.
119      * @return The number of stages in the race.
120      */
121     public int getNoOfStages() {
122         return stages.size();
123     }
124
125     /**
126      * Method to return an array of Stage objects contained in
    the race.
127      * @return An array of every Stage objects in the race.
128      */
129     public Stage[] getStages() {
130         // Converts the stages ArrayList into an array
131         Stage[] stageArr = new Stage[stages.size()];
132         stageArr = stages.toArray(stageArr);
133         return stageArr;
134     }
135
136     /**
```

```
137      * Private method to compute the total length of the race
    , that is,
138      * the sum of all the lengths of each stage in the race.
139      * @return The total length of the race.
140      */
141     private double getTotalLength() {
142         double totalLength = 0;
143         double length;
144         for (Stage stage : stages) {
145             length = stage.getLength();
146             totalLength += length;
147         }
148         return totalLength;
149     }
150 }
151
```

File - Rider.java

```java
package cycling;

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

/**
 * Rider class.<br>
 * Represents a single rider in the cycling competition.
 *
 * @author Joey Griffiths and Alexander Cairns
 *
 */
class Rider implements Serializable {

    /**
     * The number of instances of Rider, automatically incremented when the
     * constructor is called.<br>
     * Used for allocation of IDs.
     */
    private static int noOfRiders = 0;

    /**
     * The ID of the Rider.
     */
    private final int id;

    /**
     * The name of the rider.
     */
    private final String name;

    /**
     * The year of birth of the rider.
     */
    private final int yearOfBirth;

    /**
     * The ArrayList of StageResult objects that the rider contains.
     */
    private ArrayList<StageResult> results = new ArrayList<>();

    /**
     * Rider class constructor.<br>
     * Assigns a name, year of birth and automatically assigns an ID using
     * the number of instances of rider.
```

```
46      *
47      * @param name The name of the rider.
48      * @param yearOfBirth The year of birth of the rider.
49      */
50     Rider(String name, int yearOfBirth) {
51         this.name = name;
52         this.yearOfBirth = yearOfBirth;
53         id = noOfRiders++;
54     }
55 
56     /**
57      * Method to reset the static variable noOfRiders.<br>
58      * Used to reset the CyclingPortal so that IDs start from 0
    again.
59      */
60     public static void resetNoOfRiders() {
61         noOfRiders = 0;
62     }
63 
64     /**
65      * Method to get the ID of the rider.
66      *
67      * @return The ID of the rider.
68      */
69     public int getId() {
70         return id;
71     }
72 
73     /**
74      * Method to get the name of the rider.
75      *
76      * @return The name of the rider.
77      */
78     public String getName() {
79         return name;
80     }
81 
82     /**
83      * Method to get the year of birth of the rider.
84      *
85      * @return The year of birth of the rider.
86      */
87     public int getYearOfBirth() {
88         return yearOfBirth;
89     }
90 
91     /**
92      * Method to add a StageResult object to the 'results'
```

```
92 ArrayList.
93      *
94      * @param result StageResult object representing the
   result that the
95      *               rider achieved in a stage.
96      */
97     public void addResult(StageResult result) {
98         results.add(result);
99     }
100
101    /**
102     * Method to remove a StageResult object from the 'results
   ' ArrayList.
103     *
104     * @param result StageResult object representing the
   result that the
105     *               rider achieved in a stage.
106     */
107    public void removeResult(StageResult result) {
108        results.remove(result);
109    }
110
111    /**
112     * Method to get an array of all the StageResult objects
   stored in the
113     * 'results' ArrayList.
114     *
115     * @return An array of StageResult objects stored in the '
   results'
116     * ArrayList.
117     */
118    public StageResult[] getResults() {
119        StageResult[] resultArr = new StageResult[results.size
   ()];
120        resultArr = results.toArray(resultArr);
121        return resultArr;
122    }
123 }
124
```

File - Segment.java

```java
package cycling;

import java.io.Serializable;

/**
 * Segment class.<br>
 * Represents a single segment in the cycling competition.
 *
 * @author Joey Griffiths and Alexander Cairns
 *
 */
class Segment implements Serializable {

    /**
     * The number of instances of Segment, automatically incremented when the
     * constructor is called.<br>
     * Used for allocation of IDs.
     */
    private static int noOfSegments = 0;

    /**
     * The ID of the segment.
     */
    private final int id;

    /**
     * The location (distance from the start of the stage) the segment is
     * found at.
     */
    private final double location;

    /**
     * The segment's type.<br>
     * Taken from the {@link SegmentType} enum.
     */
    private final SegmentType type;

    /**
     * The average gradient of the segment.
     */
    private double averageGradient;

    /**
     * The length of the segment, in km.
     */
    private double length;
```

```
47
48      /**
49       * Segment class constructor.<br>
50       * Assigns a location, a type, an average gradient, a
    length, and an
51       * automatic ID using the number of instances of Segment.
52       *
53       * @param location The location of the segment in the stage
    .
54       * @param type The type that this segment is.
55       * @param averageGradient The average gradient of the
    segment.
56       * @param length The length of the segment.
57       */
58      Segment(double location, SegmentType type, double
    averageGradient, double length) {
59          this.location = location;
60          this.type = type;
61          this.averageGradient = averageGradient;
62          this.length = length;
63          id = noOfSegments++;
64      }
65
66      /**
67       * Segment class constructor.<br>
68       * Assigns a location, a type, and an automatic ID using
    the number of
69       * instances of Segment.
70       *
71       * @param location The location of the segment in the stage
    .
72       * @param type The type that this segment is.
73       */
74      Segment(double location, SegmentType type) {
75          this.location = location;
76          this.type = type;
77          id = noOfSegments++;
78      }
79
80      /**
81       * Method to reset the static variable noOfSegments.<br>
82       * Used to reset the CyclingPortal so that IDs start from 0
     again.
83       */
84      public static void resetNoOfSegments() {
85          noOfSegments = 0;
86      }
87
```

```
88      /**
89       * Method to get the ID of the segment.
90       *
91       * @return The ID of the segment.
92       */
93      public int getId() {
94          return id;
95      }
96
97      /**
98       * Method to get the location of the segment in the stage.
99       *
100      * @return The location of the segment.
101      */
102     public double getLocation() {
103         return location;
104     }
105
106     /**
107      * Method to get the type that the segment is.
108      *
109      * @return The type of the segment.
110      */
111     public SegmentType getType() {
112         return type;
113     }
114
115     /**
116      * Method to get the average gradient of the segment.
117      *
118      * @return The averageGradient of the stage.
119      */
120     public double getAverageGradient() {
121         return averageGradient;
122     }
123
124     /**
125      * Method to get the length of the segment.
126      *
127      * @return The length of the segment.
128      */
129     public double getLength() {
130         return length;
131     }
132 }
133
```

File - Stage.java

```java
package cycling;

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

/**
 * Stage class.<br>
 * Represents a single stage in the cycling competition.
 *
 * @author Joey Griffiths and Alexander Cairns
 *
 */
class Stage implements Serializable {

    /**
     * The number of instances of Stage, automatically incremented when the
     * constructor is called.<br>
     * Used for allocation of IDs.
     */
    private static int noOfStages = 0;

    /**
     * The ID of the stage.
     */
    private final int id;

    /**
     * The name of the stage.
     */
    private final String name;

    /**
     * The description of the stage.
     */
    private final String description;

    /**
     * The length of the stage.
     */
    private final double length;

    /**
     * The stage's type.<br>
     * Taken from the {@link StageType} enum.
     */
    private final StageType type;
```

```
48
49      /**
50       * An ArrayList of Segment objects contained in the stage.
51       */
52      private ArrayList<Segment> segments = new ArrayList<>();
53
54      /**
55       * The start time of the race.
56       */
57      private final LocalDateTime startTime;
58
59      /**
60       * Whether the stage is prepared (has results associated
    with it) or not.
61       */
62      private boolean prepared = false;
63
64      /**
65       * Stage class constructor.<br>
66       * Assigns a name, a description, a length, a start time, a
    type and an
67       * automatic ID using the number of instances of stage.
68       *
69       * @param name The name of the stage.
70       * @param description The description of the stage.
71       * @param length The length of the stage.
72       * @param startTime The start time of the stage.
73       * @param type The stage's type.
74       */
75      Stage(String name, String description, double length,
76              LocalDateTime startTime, StageType type) {
77          this.name = name;
78          this.description = description;
79          this.length = length;
80          this.startTime = startTime;
81          this.type = type;
82          id = noOfStages++;
83      }
84
85      /**
86       * Resets the static variable noOfStages.<br>
87       * Used to reset the CyclingPortal so that IDs start from 0
    again.
88       */
89      public static void resetNoOfStages() {
90          noOfStages = 0;
91      }
92
```

```java
    /**
     * Method to get the ID of the stage.
     *
     * @return The ID of the stage.
     */
    public int getId() {
        return id;
    }

    /**
     * Method to get the name of the stage.
     *
     * @return The name of the stage.
     */
    public String getName() {
        return name;
    }

    /**
     * Method to get the description of the stage.
     *
     * @return The description of the stage.
     */
    public String getDescription() {
        return description;
    }

    /**
     * Method to get the length of the stage.
     *
     * @return The length of the stage.
     */
    public double getLength() {
        return length;
    }

    /**
     * Method to get the stage's type.
     *
     * @return The stage's type.
     */
    public StageType getType() {
        return type;
    }

    /**
     * Method to get the start time of the stage.
     *
```

```
141      * @return The start time of the stage.
142      */
143     public LocalDateTime getStartTime() {
144         return startTime;
145     }
146 
147     /**
148      * Method to get whether the stage is prepared or not.
149      *
150      * @return true / false (prepared / not prepared)
151      */
152     public boolean isPrepared() {
153         return prepared;
154     }
155 
156     /**
157      * Method to add a Segment object to the 'segments'
    ArrayList.
158      *
159      * @param segmentToAdd Segment object to be added to the
    stage.
160      */
161     public void addSegment(Segment segmentToAdd) {
162         // Finds the position to add the segment to in the
    list so that they are
163         // ordered by location in the stage
164         int i = 0;
165         for (Segment segment : segments) {
166             if (segmentToAdd.getLocation() <= segment.
    getLocation()) {
167                 // Checks all segments and if the segment to
    add is not further in the stage,
168                 // the segment is added behind this segment in
     the list
169                 segments.add(i, segmentToAdd);
170                 return;
171             }
172             i++;
173         }
174         // If the segment to add is the furthest in the stage
    , it is added to the end
175         segments.add(segmentToAdd);
176     }
177 
178     /**
179      * Method to remove a Segment object from the 'segments'
    ArrayList.
180      *
```

```
181      * @param segment Segment object to be removed from the
    stage.
182      */
183     public void removeSegment(Segment segment) {
184         segments.remove(segment);
185     }
186 
187     /**
188      * Method to get an array of all the segments in the race.
189      *
190      * @return An array of Segment objects stored in the stage
    .
191      */
192     public Segment[] getSegments() {
193         Segment[] segmentArr = new Segment[segments.size()];
194         segmentArr = segments.toArray(segmentArr);
195         return segmentArr;
196     }
197 
198     /**
199      * Method to set the status of the stage to prepared.
200      */
201     public void prepare() {
202         prepared = true;
203     }
204 }
205 
```

File - StageResult.java

```java
package cycling;

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

/**
 * StageResult class.<br>
 * Represents a single StageResult in the cycling competition.
 *
 * @author Joey Griffiths and Alexander Cairns
 *
 */
class StageResult implements Serializable {

    /**
     * The number of instances of StageResult, automatically incremented when
     * the constructor is called.<br>
     * Used for allocation of IDs.
     */
    private static int totalResults = 0;

    /**
     * The ID of the stage result.
     */
    private final int id;

    /**
     * The Stage object that this stage result is associated with.
     */
    private final Stage stage;

    /**
     * An array of checkpoints in the stage result (time at end of each
     * segment.)
     */
    private final LocalTime[] checkpoints;

    /**
     * StageResult class constructor.<br>
     * Assigns a stage, a list of checkpoints, and an automatic ID using the
     * number of instances of stageResult.
     *
     * @param stage The stage that this stage result is associated with.
```

```
44      * @param checkpoints The list of checkpoints in the stage
   result.
45      */
46     StageResult(Stage stage, LocalTime... checkpoints) {
47         this.stage = stage;
48         this.checkpoints = checkpoints;
49         this.id = totalResults++;
50     }
51 
52     /**
53      * Resets the static variable totalResults.<br>
54      * Used to reset the CyclingPortal so that IDs start from 0
    again.
55      */
56     public static void resetTotalResults() {
57         totalResults = 0;
58     }
59 
60     /**
61      * Method to get the ID of the stage result.
62      *
63      * @return The ID of the stage result.
64      */
65     public int getId() {
66         return id;
67     }
68 
69     /**
70      * Method to get the stage that this stage result is
   associated with.
71      *
72      * @return The stage result's stage.
73      */
74     public Stage getStage() {
75         return stage;
76     }
77 
78     /**
79      * Method to get an array of the checkpoints stored in the
   stage result.
80      *
81      * @return The array of checkpoints.
82      */
83     public LocalTime[] getCheckpoints() {
84         return checkpoints;
85     }
86 }
87 
```

File - Team.java

```java
package cycling;

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

/**
 * Team class.<br>
 * Represents a team in the cycling competition.
 *
 * @author Joey Griffiths and Alexander Cairns
 */
class Team implements Serializable {

    /**
     * The number of instances of Team, automatically incremented when
     * the constructor is called.<br>
     * Used for allocation of IDs.
     */
    private static int noOfTeams = 0;

    /**
     * The ID of the team.
     */
    private final int id;

    /**
     * The name of the team.
     */
    private final String name;

    /**
     * The team's description.
     */
    private final String description;

    /**
     * An ArrayList of Rider objects contained in the team.
     */
    private ArrayList<Rider> riders = new ArrayList<>();

    /**
     * Team class constructor.<br>
     * Assigns a name, a description and an automatic ID using the number of
     * isntances of team.
     *
     * @param name The team's name.
```

```
47      * @param description The team's description.
48      */
49     Team(String name, String description) {
50         this.name = name;
51         this.description = description;
52         id = noOfTeams++;
53     }
54 
55     /**
56      * Resets the static variable noOfTeams.<br>
57      * Used to reset the CyclingPortal so that IDs start from 0
    again.
58      */
59     public static void resetNoOfTeams() {
60         noOfTeams = 0;
61     }
62 
63     /**
64      * Method to add a rider to the team.
65      *
66      * @param rider The Rider object to be added to the 'riders
    ' ArrayList.
67      */
68     public void addRider(Rider rider) {
69         riders.add(rider);
70     }
71 
72     /**
73      * Method to remove a rider from the team.
74      *
75      * @param rider The Rider object to be removed from the '
   riders' ArrayList.
76      */
77     public void removeRider(Rider rider) {
78         riders.remove(rider);
79     }
80 
81     /**
82      * Method to get the ID of the team.
83      *
84       * @return The team's ID.
85      */
86     public int getId() {
87         return id;
88     }
89 
90     /**
91      * Method to get the name of the team.
```

File - Team.java

```java
     *
     * @return The team's name.
     */
    public String getName() {
        return name;
    }

    /**
     * Method to get the description of the team.
     *
     * @return The team's description.
     */
    public String getDescription() {
        return description;
    }

    /**
     * Method to get a list of all riders in the team.
     *
     * @return An array of Rider objects contained in the team
.
     */
    public Rider[] getRiders() {
        Rider[] riderArr = new Rider[riders.size()];
        riderArr = riders.toArray(riderArr);
        return riderArr;
    }
}

```