CyclingPortal Printout

123456789 & 987654321

Contents

1 CategorizedClimb.java

```java
package cycling;

public class CategorizedClimb extends Segment {
  private final Double averageGradient;
  private final Double length;

  public CategorizedClimb(
      Stage stage, Double location, SegmentType type, Double averageGradient, Double length)
      throws InvalidLocationException, InvalidStageStateException, InvalidStageTypeException {
    super(stage, type, location);
    this.averageGradient = averageGradient;
    this.length = length;
  }
}
```

2 CyclingPortal.java

```java
package cycling;

import java.io.*;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.ArrayList;
import java.util.List;

// TODO:
//    - Documentation/Comments

public class CyclingPortal implements CyclingPortalInterface {
  // ArrayLists for all of a cycling portal instances teams, riders, races, stages and segments.
  // Although HashMaps could have been used here to get riders by int ID, it would be slower in the
  // long run as we would need to constantly convert it back to arrays to output results.
  private ArrayList<Team> teams = new ArrayList<>();
  private ArrayList<Rider> riders = new ArrayList<>();
  private ArrayList<Race> races = new ArrayList<>();
  private ArrayList<Stage> stages = new ArrayList<>();
  private ArrayList<Segment> segments = new ArrayList<>();

  /**
   * Determine if a string contains any illegal whitespace characters.
   *
   * @param string The input string to be tested for whitespace.
   * @return A boolean, true if the input string contains whitespace, false if not.
   */
  public static boolean containsWhitespace(String string) {
    for (int i = 0; i < string.length(); ++i) {
      if (Character.isWhitespace(string.charAt(i))) {
        return true;
      }
    }
    return false;
  }

  /**
   * Get a Team object by a Team ID.
```

```
39      *
40      * @param ID The int ID of the Team to be looked up.
41      * @return The Team object of the team, if one is found.
42      * @throws IDNotRecognisedException Thrown if no team is found with the given Team ID.
43      */
44     public Team getTeamById(int ID) throws IDNotRecognisedException {
45       for (Team team : teams) {
46         if (team.getId() == ID) {
47           return team;
48         }
49       }
50       throw new IDNotRecognisedException("Team ID not found.");
51     }
52
53     /**
54      * Get a Rider object by a Rider ID.
55      *
56      * @param ID The int ID of the Rider to be looked up.
57      * @return The Rider object of the Rider, if one is found.
58      * @throws IDNotRecognisedException Thrown if no rider is found with the given Rider ID.
59      */
60     public Rider getRiderById(int ID) throws IDNotRecognisedException {
61       for (Rider rider : riders) {
62         if (rider.getId() == ID) {
63           return rider;
64         }
65       }
66       throw new IDNotRecognisedException("Rider ID not found.");
67     }
68
69     /**
70      * Get a Race object by a Race ID.
71      *
72      * @param ID The int ID of the Race to be looked up.
73      * @return The Race object of the race, if one is found.
74      * @throws IDNotRecognisedException Thrown if no race is found with the given Race ID.
75      */
76     public Race getRaceById(int ID) throws IDNotRecognisedException {
77       for (Race race : races) {
78         if (race.getId() == ID) {
79           return race;
80         }
81       }
82       throw new IDNotRecognisedException("Race ID not found.");
83     }
84
85     /**
86      * Get a Stage object by a Stage ID.
87      *
88      * @param ID The int ID of the Stage to be looked up.
89      * @return The Stage object of the stage, if one is found.
90      * @throws IDNotRecognisedException Thrown if no stage is found with the given Stage ID.
91      */
92     public Stage getStageById(int ID) throws IDNotRecognisedException {
93       for (Stage stage : stages) {
94         if (stage.getId() == ID) {
95           return stage;
96         }
```

```
97      }
98      throw new IDNotRecognisedException("Stage ID not found.");
99    }
100
101   /**
102    * Get a Segment object by a Segment ID.
103    *
104    * @param ID The int ID of the Segment to be looked up.
105    * @return The Segment object of the segment, if one is found.
106    * @throws IDNotRecognisedException Thrown if no segment is found with the given Segment ID.
107    */
108   public Segment getSegmentById(int ID) throws IDNotRecognisedException {
109     for (Segment segment : segments) {
110       if (segment.getId() == ID) {
111         return segment;
112       }
113     }
114     throw new IDNotRecognisedException("Segment ID not found.");
115   }
116
117   /**
118    * Loops over all races, stages and segments to remove all of a given riders results.
119    *
120    * @param rider The Rider object whose results will be removed from the Cycling Portal.
121    */
122   public void removeRiderResults(Rider rider) {
123     for (Race race : races) {
124       race.removeRiderResults(rider);
125     }
126     for (Stage stage : stages) {
127       stage.removeRiderResults(rider);
128     }
129     for (Segment segment : segments) {
130       segment.removeRiderResults(rider);
131     }
132   }
133
134   @Override
135   public int[] getRaceIds() {
136     int[] raceIDs = new int[races.size()];
137     for (int i = 0; i < races.size(); i++) {
138       Race race = races.get(i);
139       raceIDs[i] = race.getId();
140     }
141     return raceIDs;
142   }
143
144   @Override
145   public int createRace(String name, String description)
146       throws IllegalNameException, InvalidNameException {
147     // Check a race with this name does not already exist in the system.
148     for (Race race : races) {
149       if (race.getName().equals(name)) {
150         throw new IllegalNameException("A Race with the name " + name + " already exists.");
151       }
152     }
153     Race race = new Race(name, description);
154     races.add(race);
```

```
155     return race.getId();
156   }
157
158   @Override
159   public String viewRaceDetails(int raceId) throws IDNotRecognisedException {
160     Race race = getRaceById(raceId);
161     return race.getDetails();
162   }
163
164   @Override
165   public void removeRaceById(int raceId) throws IDNotRecognisedException {
166     Race race = getRaceById(raceId);
167     // Remove all the races stages from the CyclingPortal.
168     for (final Stage stage : race.getStages()) {
169       stages.remove(stage);
170     }
171     races.remove(race);
172   }
173
174   @Override
175   public int getNumberOfStages(int raceId) throws IDNotRecognisedException {
176     Race race = getRaceById(raceId);
177     return race.getStages().size();
178   }
179
180   @Override
181   public int addStageToRace(
182       int raceId,
183       String stageName,
184       String description,
185       double length,
186       LocalDateTime startTime,
187       StageType type)
188       throws IDNotRecognisedException, IllegalNameException, InvalidNameException,
189           InvalidLengthException {
190     Race race = getRaceById(raceId);
191     // Check a stage with this name does not already exist in the system.
192     for (final Stage stage : stages) {
193       if (stage.getName().equals(stageName)) {
194         throw new IllegalNameException("A stage with the name " + stageName + " already exists.");
195       }
196     }
197     Stage stage = new Stage(race, stageName, description, length, startTime, type);
198     stages.add(stage);
199     race.addStage(stage);
200     return stage.getId();
201   }
202
203   @Override
204   public int[] getRaceStages(int raceId) throws IDNotRecognisedException {
205     Race race = getRaceById(raceId);
206     ArrayList<Stage> raceStages = race.getStages();
207     int[] raceStagesId = new int[raceStages.size()];
208     // Gathers the Stage ID's of the Stages in the Race.
209     for (int i = 0; i < raceStages.size(); i++) {
210       Stage stage = race.getStages().get(i);
211       raceStagesId[i] = stage.getId();
212     }
```

```
213       return raceStagesId;
214     }
215
216     @Override
217     public double getStageLength(int stageId) throws IDNotRecognisedException {
218       Stage stage = getStageById(stageId);
219       return stage.getLength();
220     }
221
222     @Override
223     public void removeStageById(int stageId) throws IDNotRecognisedException {
224       Stage stage = getStageById(stageId);
225       Race race = stage.getRace();
226       // Removes stage from both the Races and Stages.
227       race.removeStage(stage);
228       stages.remove(stage);
229     }
230
231     @Override
232     public int addCategorizedClimbToStage(
233         int stageId, Double location, SegmentType type, Double averageGradient, Double length)
234         throws IDNotRecognisedException, InvalidLocationException, InvalidStageStateException,
235             InvalidStageTypeException {
236       Stage stage = getStageById(stageId);
237       CategorizedClimb climb = new CategorizedClimb(stage, location, type, averageGradient, length);
238       // Adds Categorized Climb to both the list of Segments and the Stage.
239       segments.add(climb);
240       stage.addSegment(climb);
241       return climb.getId();
242     }
243
244     @Override
245     public int addIntermediateSprintToStage(int stageId, double location)
246         throws IDNotRecognisedException, InvalidLocationException, InvalidStageStateException,
247             InvalidStageTypeException {
248       Stage stage = getStageById(stageId);
249       IntermediateSprint sprint = new IntermediateSprint(stage, location);
250       // Adds Intermediate Sprint to both the list of Segments and the Stage.
251       segments.add(sprint);
252       stage.addSegment(sprint);
253       return sprint.getId();
254     }
255
256     @Override
257     public void removeSegment(int segmentId)
258         throws IDNotRecognisedException, InvalidStageStateException {
259       Segment segment = getSegmentById(segmentId);
260       Stage stage = segment.getStage();
261       // Removes Segment from both the Stage and list of Segments.
262       stage.removeSegment(segment);
263       segments.remove(segment);
264     }
265
266     @Override
267     public void concludeStagePreparation(int stageId)
268         throws IDNotRecognisedException, InvalidStageStateException {
269       Stage stage = getStageById(stageId);
270       stage.concludePreparation();
```

```
271   }
272
273   @Override
274   public int[] getStageSegments(int stageId) throws IDNotRecognisedException {
275     Stage stage = getStageById(stageId);
276     ArrayList<Segment> stageSegments = stage.getSegments();
277     int[] stageSegmentsId = new int[stageSegments.size()];
278     // Gathers Segment ID's from the Segments in the Stage.
279     for (int i = 0; i < stageSegments.size(); i++) {
280       Segment segment = stageSegments.get(i);
281       stageSegmentsId[i] = segment.getId();
282     }
283     return stageSegmentsId;
284   }
285
286   @Override
287   public int createTeam(String name, String description)
288       throws IllegalNameException, InvalidNameException {
289     // Checks if the Team name already exists on the system.
290     for (final Team team : teams) {
291       if (team.getName().equals(name)) {
292         throw new IllegalNameException("A Team with the name " + name + " already exists.");
293       }
294     }
295     Team team = new Team(name, description);
296     teams.add(team);
297     return team.getId();
298   }
299
300   @Override
301   public void removeTeam(int teamId) throws IDNotRecognisedException {
302     Team team = getTeamById(teamId);
303     // Loops through and removes Team Riders and Team Rider Results.
304     for (final Rider rider : team.getRiders()) {
305       removeRiderResults(rider);
306       riders.remove(rider);
307     }
308     teams.remove(team);
309   }
310
311   @Override
312   public int[] getTeams() {
313     int[] teamIDs = new int[teams.size()];
314     for (int i = 0; i < teams.size(); i++) {
315       Team team = teams.get(i);
316       teamIDs[i] = team.getId();
317     }
318     return teamIDs;
319   }
320
321   @Override
322   public int[] getTeamRiders(int teamId) throws IDNotRecognisedException {
323     Team team = getTeamById(teamId);
324     ArrayList<Rider> teamRiders = team.getRiders();
325     int[] teamRiderIds = new int[teamRiders.size()];
326     // Gathers ID's of Riders in the Team.
327     for (int i = 0; i < teamRiderIds.length; i++) {
328       // Assert the rider is actually on the team.
```

```
329         assert teamRiders.get(i).getTeam().equals(team);
330         // Return the rider id.
331         teamRiderIds[i] = teamRiders.get(i).getId();
332       }
333       return teamRiderIds;
334     }
335
336     @Override
337     public int createRider(int teamID, String name, int yearOfBirth)
338         throws IDNotRecognisedException, IllegalArgumentException {
339       Team team = getTeamById(teamID);
340       Rider rider = new Rider(team, name, yearOfBirth);
341       // Adds Rider to both the Team and the list of Riders.
342       team.addRider(rider);
343       riders.add(rider);
344
345       // Assert at least one rider has been added
346       assert riders.size() > 0;
347
348       return rider.getId();
349     }
350
351     @Override
352     public void removeRider(int riderId) throws IDNotRecognisedException {
353       Rider rider = getRiderById(riderId);
354       removeRiderResults(rider);
355       // Removes Rider from both the Team and the list of Riders.
356       rider.getTeam().removeRider(rider);
357       riders.remove(rider);
358     }
359
360     @Override
361     public void registerRiderResultsInStage(int stageId, int riderId, LocalTime... checkpoints)
362         throws IDNotRecognisedException, DuplicatedResultException, InvalidCheckpointsException,
363             InvalidStageStateException {
364       Stage stage = getStageById(stageId);
365       Rider rider = getRiderById(riderId);
366       stage.registerResult(rider, checkpoints);
367     }
368
369     @Override
370     public LocalTime[] getRiderResultsInStage(int stageId, int riderId)
371         throws IDNotRecognisedException {
372       Stage stage = getStageById(stageId);
373       Rider rider = getRiderById(riderId);
374       StageResult result = stage.getRiderResult(rider);
375
376       if (result == null) {
377         // Returns an empty array if the Result is null.
378         return new LocalTime[] {};
379       } else {
380         LocalTime[] checkpoints = result.getCheckpoints();
381         // Rider Results will always be 1 shorter than the checkpoint length because
382         // the finish time checkpoint will be replaced with the Elapsed Time and the start time
383         // checkpoint will be ignored.
384         LocalTime[] resultsInStage = new LocalTime[checkpoints.length - 1];
385         LocalTime elapsedTime = LocalTime.MIDNIGHT.plus(result.getElapsedTime());
386         for (int i = 0; i < resultsInStage.length; i++) {
```

```
387         if (i == resultsInStage.length - 1) {
388           // Adds the Elapsed Time to the end of the array of Results.
389           resultsInStage[i] = elapsedTime;
390         } else {
391           // Adds each checkpoint to the array of Results until all have been added, skipping the
392           // Start time checkpoint.
393           resultsInStage[i] = checkpoints[i + 1];
394         }
395       }
396       return resultsInStage;
397     }
398   }
399
400   @Override
401   public LocalTime getRiderAdjustedElapsedTimeInStage(int stageId, int riderId)
402       throws IDNotRecognisedException {
403     Stage stage = getStageById(stageId);
404     Rider rider = getRiderById(riderId);
405     StageResult result = stage.getRiderResult(rider);
406     if (result == null) {
407       return null;
408     } else {
409       return result.getAdjustedElapsedLocalTime();
410     }
411   }
412
413   @Override
414   public void deleteRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException {
415     Stage stage = getStageById(stageId);
416     Rider rider = getRiderById(riderId);
417     stage.removeRiderResults(rider);
418   }
419
420   @Override
421   public int[] getRidersRankInStage(int stageId) throws IDNotRecognisedException {
422     Stage stage = getStageById(stageId);
423     // Gets a list of Riders from the Stage ordered by their Elapsed Times.
424     List<Rider> riders = stage.getRidersByElapsedTime();
425     int[] riderIds = new int[riders.size()];
426     // Gathers ID's from the ordered list of Riders.
427     for (int i = 0; i < riders.size(); i++) {
428       riderIds[i] = riders.get(i).getId();
429     }
430     return riderIds;
431   }
432
433   @Override
434   public LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId)
435       throws IDNotRecognisedException {
436     Stage stage = getStageById(stageId);
437     // Gets a list of Riders from the Stage ordered by their Elapsed Times.
438     List<Rider> riders = stage.getRidersByElapsedTime();
439     LocalTime[] riderAETs = new LocalTime[riders.size()];
440     // Gathers Riders' Adjusted Elapsed Times ordered by their Elapsed Times.
441     for (int i = 0; i < riders.size(); i++) {
442       Rider rider = riders.get(i);
443       riderAETs[i] = stage.getRiderResult(rider).getAdjustedElapsedLocalTime();
444     }
```

```
445     return riderAETs;
446   }
447
448   @Override
449   public int[] getRidersPointsInStage(int stageId) throws IDNotRecognisedException {
450     Stage stage = getStageById(stageId);
451     // Gets a list of Riders from the Stage ordered by their Elapsed Times.
452     List<Rider> riders = stage.getRidersByElapsedTime();
453     int[] riderSprinterPoints = new int[riders.size()];
454     // Gathers Sprinters' Points ordered by their Elapsed Times.
455     for (int i = 0; i < riders.size(); i++) {
456       Rider rider = riders.get(i);
457       riderSprinterPoints[i] = stage.getRiderResult(rider).getSprintersPoints();
458     }
459     return riderSprinterPoints;
460   }
461
462   @Override
463   public int[] getRidersMountainPointsInStage(int stageId) throws IDNotRecognisedException {
464     Stage stage = getStageById(stageId);
465     // Gets a list of Riders from the Stage ordered by their Elapsed Times.
466     List<Rider> riders = stage.getRidersByElapsedTime();
467     int[] riderMountainPoints = new int[riders.size()];
468     // Gathers Riders' Mountain Points ordered by their Elapsed Times.
469     for (int i = 0; i < riders.size(); i++) {
470       Rider rider = riders.get(i);
471       riderMountainPoints[i] = stage.getRiderResult(rider).getMountainPoints();
472     }
473     return riderMountainPoints;
474   }
475
476   @Override
477   public void eraseCyclingPortal() {
478     // Replaces teams, riders, races, stages and segments with empty ArrayLists.
479     teams = new ArrayList<>();
480     riders = new ArrayList<>();
481     races = new ArrayList<>();
482     stages = new ArrayList<>();
483     segments = new ArrayList<>();
484     // Sets the ID counters of the Rider, Team, Race, Stage and Segment objects back
485     // to 0.
486     Rider.resetIdCounter();
487     Team.resetIdCounter();
488     Race.resetIdCounter();
489     Stage.resetIdCounter();
490     Segment.resetIdCounter();
491
492     // Assert the portal is erased.
493     assert teams.size() == 0;
494     assert races.size() == 0;
495   }
496
497   @Override
498   public void saveCyclingPortal(String filename) throws IOException {
499     FileOutputStream file = new FileOutputStream(filename + ".ser");
500     ObjectOutputStream output = new ObjectOutputStream(file);
501     // Saves teams, riders, races, stages and segments ArrayLists.
502     // Saves ID counters of Team, Rider, Race, Stage and Segment objects.
```

```
503     SavedCyclingPortal savedCyclingPortal =
504         new SavedCyclingPortal(
505             teams,
506             riders,
507             races,
508             stages,
509             segments,
510             Team.getIdCounter(),
511             Rider.getIdCounter(),
512             Race.getIdCounter(),
513             Stage.getIdCounter(),
514             Segment.getIdCounter());
515     output.writeObject(savedCyclingPortal);
516     output.close();
517     file.close();
518   }
519
520   @Override
521   public void loadCyclingPortal(String filename) throws IOException, ClassNotFoundException {
522     eraseCyclingPortal();
523     FileInputStream file = new FileInputStream(filename + ".ser");
524     ObjectInputStream input = new ObjectInputStream(file);
525
526     SavedCyclingPortal savedCyclingPortal = (SavedCyclingPortal) input.readObject();
527     // Imports teams, riders, races, stages and segments ArrayLists from the last save.
528     teams = savedCyclingPortal.teams;
529     riders = savedCyclingPortal.riders;
530     races = savedCyclingPortal.races;
531     stages = savedCyclingPortal.stages;
532     segments = savedCyclingPortal.segments;
533
534     // Imports ID counters of Team, Rider, Race, Stage and Segment objects from the last save.
535     Team.setIdCounter(savedCyclingPortal.teamIdCount);
536     Rider.setIdCounter(savedCyclingPortal.riderIdCount);
537     Race.setIdCounter(savedCyclingPortal.raceIdCount);
538     Stage.setIdCounter(savedCyclingPortal.stageIdCount);
539     Segment.setIdCounter(savedCyclingPortal.segmentIdCount);
540
541     input.close();
542     file.close();
543   }
544
545   @Override
546   public void removeRaceByName(String name) throws NameNotRecognisedException {
547     for (final Race race : races) {
548       if (race.getName().equals(name)) {
549         races.remove(race);
550         return;
551       }
552     }
553     throw new NameNotRecognisedException("Race name is not in the system.");
554   }
555
556   @Override
557   public int[] getRidersGeneralClassificationRank(int raceId) throws IDNotRecognisedException {
558     Race race = getRaceById(raceId);
559     List<Rider> riders = race.getRidersByAdjustedElapsedTime();
560     int[] riderIds = new int[riders.size()];
```

```
561     // Gathers Rider ID's ordered by their Adjusted Elapsed Times.
562     for (int i = 0; i < riders.size(); i++) {
563       riderIds[i] = riders.get(i).getId();
564     }
565     return riderIds;
566   }
567
568   @Override
569   public LocalTime[] getGeneralClassificationTimesInRace(int raceId)
570       throws IDNotRecognisedException {
571     Race race = getRaceById(raceId);
572     // Gets a list of Riders from the Stage ordered by their Adjusted Elapsed Times.
573     List<Rider> riders = race.getRidersByAdjustedElapsedTime();
574     LocalTime[] riderTimes = new LocalTime[riders.size()];
575     // Gathers Riders' Cumulative Adjusted Elapsed LocalTimes ordered by their Adjusted Elapsed
576     // Times.
577     for (int i = 0; i < riders.size(); i++) {
578       riderTimes[i] = race.getRiderResults(riders.get(i)).getCumulativeAdjustedElapsedLocalTime();
579     }
580     return riderTimes;
581   }
582
583   @Override
584   public int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException {
585     Race race = getRaceById(raceId);
586     List<Rider> riders = race.getRidersByAdjustedElapsedTime();
587     int[] riderIds = new int[riders.size()];
588     // Gathers Riders' Cumulative Sprinters Points ordered by their Adjusted Elapsed Times.
589     for (int i = 0; i < riders.size(); i++) {
590       riderIds[i] = race.getRiderResults(riders.get(i)).getCumulativeSprintersPoints();
591     }
592     return riderIds;
593   }
594
595   @Override
596   public int[] getRidersMountainPointsInRace(int raceId) throws IDNotRecognisedException {
597     Race race = getRaceById(raceId);
598     List<Rider> riders = race.getRidersByAdjustedElapsedTime();
599     int[] riderIds = new int[riders.size()];
600     // Gathers Riders' Cumulative Mountain Points ordered by their Adjusted Elapsed Times.
601     for (int i = 0; i < riders.size(); i++) {
602       riderIds[i] = race.getRiderResults(riders.get(i)).getCumulativeMountainPoints();
603     }
604     return riderIds;
605   }
606
607   @Override
608   public int[] getRidersPointClassificationRank(int raceId) throws IDNotRecognisedException {
609     Race race = getRaceById(raceId);
610     List<Rider> riders = race.getRidersBySprintersPoints();
611     int[] riderIds = new int[riders.size()];
612     // Gathers Rider ID's ordered by their Sprinters Points.
613     for (int i = 0; i < riders.size(); i++) {
614       riderIds[i] = riders.get(i).getId();
615     }
616     return riderIds;
617   }
618
```

```
619   @Override
620   public int[] getRidersMountainPointClassificationRank(int raceId)
621       throws IDNotRecognisedException {
622     Race race = getRaceById(raceId);
623     List<Rider> riders = race.getRidersByMountainPoints();
624     int[] riderIds = new int[riders.size()];
625     // Gathers Rider ID's ordered by their Mountain Points.
626     for (int i = 0; i < riders.size(); i++) {
627       riderIds[i] = riders.get(i).getId();
628     }
629     return riderIds;
630   }
631 }
```

3 IntermediateSprint.java

```
1  package cycling;
2
3  public class IntermediateSprint extends Segment {
4    private final double location;
5
6    public IntermediateSprint(Stage stage, double location)
7        throws InvalidLocationException, InvalidStageTypeException, InvalidStageStateException {
8      super(stage, SegmentType.SPRINT, location);
9      this.location = location;
10   }
11 }
```

4 Race.java

```
1  package cycling;
2
3  import java.io.Serializable;
4  import java.time.LocalDateTime;
5  import java.util.*;
6  import java.util.stream.Collectors;
7
8  /**
9   * Race Class. This represents a Race that holds a Race's Stages, Riders Results, and also contains
10  * methods that deal with these.
11  */
12 public class Race implements Serializable {
13
14   private final String name;
15   private final String description;
16
17   private final ArrayList<Stage> stages = new ArrayList<>();
18
19   private HashMap<Rider, RaceResult> results = new HashMap<>();
20
21   private static int count = 0;
22   private final int id;
23
24   /**
25    * Constructor method that sets up Rider with a name and a description.
26    *
```

```
27      * @param name: Cannot be empty, null, have a length greater than 30 or contain any whitespace.
28      * @param description: A description of the race.
29      * @throws InvalidNameException Thrown if the Race name does not meet name requirements stated
30      *     above.
31      */
32     public Race(String name, String description) throws InvalidNameException {
33       if (name == null
34           || name.isEmpty()
35           || name.length() > 30
36           || CyclingPortal.containsWhitespace(name)) {
37         throw new InvalidNameException(
38             "The name cannot be null, empty, have more than 30 characters, or have white spaces.");
39       }
40       this.name = name;
41       this.description = description;
42       // ID counter represents the highest known ID at the current time to ensure there
43       // are no ID collisions.
44       this.id = Race.count++;
45     }
46
47     /** Method that resets the static ID counter of the Race. Used for erasing and loading. */
48     static void resetIdCounter() {
49       count = 0;
50     }
51
52     /**
53      * Method to get the current state of the static ID counter.
54      *
55      * @return the highest race ID stored currently.
56      */
57     static int getIdCounter() {
58       return count;
59     }
60
61     /**
62      * Method that sets the static ID counter to a given value. Used when loading to avoid ID
63      * collisions.
64      *
65      * @param newCount: new value of the static ID counter.
66      */
67     static void setIdCounter(int newCount) {
68       count = newCount;
69     }
70
71     /**
72      * Method to get the ID of the Race object.
73      *
74      * @return id: the Race's unique ID value.
75      */
76     public int getId() {
77       return id;
78     }
79
80     /**
81      * Method to get the name of the Race.
82      *
83      * @return name: the given name of the Race.
84      */
```

```
85    public String getName() {
86      return name;
87    }
88
89    /**
90     * Method that adds a Stage to the Race object's ordered list of Stages. It is added to the
91     * correct position based on its start time.
92     *
93     * @param stage: The stage to be added to the Race.
94     */
95    public void addStage(Stage stage) {
96      // Loops over stages in the race to insert the new stage in the correct place such that
97      // all of the stages are sorted by their start time.
98      for (int i = 0; i < stages.size(); i++) {
99        // Retrieves the start time of each Stage in the Race's current Stages one by one.
100       // These are already ordered by their start times.
101       LocalDateTime iStartTime = stages.get(i).getStartTime();
102       // Adds the new Stage to the list of stages in the correct position based on
103       // its start time.
104       if (stage.getStartTime().isBefore(iStartTime)) {
105         stages.add(i, stage);
106         return;
107       }
108     }
109     stages.add(stage);
110   }
111
112   /**
113    * Method to get the list of Stages in the Race ordered by their start times.
114    *
115    * @return stages: The ordered list of Stages.
116    */
117   public ArrayList<Stage> getStages() {
118     // stages is already sorted, so no sorting needs to be done.
119     return stages;
120   }
121
122   /**
123    * Method that removes a given Stage from the list of Stages.
124    *
125    * @param stage: the Stage to be deleted.
126    */
127   public void removeStage(Stage stage) {
128     stages.remove(stage);
129   }
130
131   /**
132    * Method to get then details of a Race including Race ID, name, description number of stages and
133    * total length.
134    *
135    * @return Concatenated paragraph of race details.
136    */
137   public String getDetails() {
138     double currentLength = 0;
139     for (final Stage stage : stages) {
140       currentLength = currentLength + stage.getLength();
141     }
142     return ("Race ID: "
```

```
143         + id
144         + System.lineSeparator()
145         + "Name: "
146         + name
147         + System.lineSeparator()
148         + "Description: "
149         + description
150         + System.lineSeparator()
151         + "Number of Stages: "
152         + stages.size()
153         + System.lineSeparator()
154         + "Total length: "
155         + currentLength);
156     }
157 
158     /**
159      * Method to get a list of Riders in the Race, sorted by their Adjusted Elapsed Time.
160      *
161      * @return The correctly sorted Riders.
162      */
163     public List<Rider> getRidersByAdjustedElapsedTime() {
164       // First generate the race result to calculate each riders Adjusted Elapsed Time.
165       calculateResults();
166       // Then return the riders sorted by their Adjusted Elapsed Time.
167       return sortRiderResultsBy(RaceResult.sortByAdjustedElapsedTime);
168     }
169 
170     /**
171      * Method to get a list of Riders in the Race, sorted by their Sprinters Points.
172      *
173      * @return The correctly sorted Riders.
174      */
175     public List<Rider> getRidersBySprintersPoints() {
176       // First generate the race result to calculate each riders Sprinters Points.
177       calculateResults();
178       // Then return the riders sorted by their sprinters points.
179       return sortRiderResultsBy(RaceResult.sortBySprintersPoints);
180     }
181 
182     /**
183      * Method to get a list of Riders in the Race, sorted by their Mountain Points.
184      *
185      * @return The correctly sorted Riders.
186      */
187     public List<Rider> getRidersByMountainPoints() {
188       // First generate the race result to calculate each riders Mountain Points.
189       calculateResults();
190       // Then return the riders sorted by their mountain points.
191       return sortRiderResultsBy(RaceResult.sortByMountainPoints);
192     }
193 
194     /**
195      * Method to get the results of a given Rider.
196      *
197      * @param rider: Rider to get the results of.
198      * @return RaceResult: Result of the Rider.
199      */
200     public RaceResult getRiderResults(Rider rider) {
```

```
201       // First generate the race result to calculate each riders results.
202       calculateResults();
203       // Then return the riders result object.
204       return results.get(rider);
205     }
206
207     /**
208      * Method to remove the Results of a given Rider.
209      *
210      * @param rider: Rider whose Results will be removed.
211      */
212     public void removeRiderResults(Rider rider) {
213       results.remove(rider);
214     }
215
216     /**
217      * Method to get a list of Riders sorted by a given comparator of their Results. Will only return
218      * riders who have results registered in their name.
219      *
220      * @param comparison: a comparator on the Riders' Results to sort the Riders by.
221      * @return List<Rider>: List of Riders (who posses recorded results) sorted by the comparator on
222      *     the Results.
223      */
224     private List<Rider> sortRiderResultsBy(Comparator<RaceResult> comparator) {
225       // convert the hashmap into a set
226       return results.entrySet().stream()
227           // Sort the set by the comparator on the results.
228           .sorted(Map.Entry.comparingByValue(comparator))
229           // Get the rider element of the set and ignore the results now they have been sorted.
230           .map(Map.Entry::getKey)
231           // Convert to a list of riders.
232           .collect(Collectors.toList());
233     }
234
235     /**
236      * Method to register the Rider's Result to the Stage.
237      *
238      * @param rider: Rider whose Result needs to be registered.
239      * @param stageResult: Stage that the Result will be added to.
240      */
241     private void registerRiderResults(Rider rider, StageResult stageResult) {
242       if (results.containsKey(rider)) {
243         // If results already exist for a given rider add the current stage results
244         // to the existing total race results.
245         results.get(rider).addStageResult(stageResult);
246       } else {
247         // If no race results exists, create a new RaceResult object based on the current
248         // stage results.
249         RaceResult raceResult = new RaceResult();
250         raceResult.addStageResult(stageResult);
251         results.put(rider, raceResult);
252       }
253     }
254
255     /** Private method that calculates the results for each Rider. */
256     private void calculateResults() {
257       // Clear existing results.
258       results = new HashMap<>();
```

```
259     // We must loop over all stages and collect their results for each rider as each riders results
260     // are dependent on their position in the race, and thus the results of the other riders.
261     for (Stage stage : stages) {
262       HashMap<Rider, StageResult> stageResults = stage.getStageResults();
263       for (Rider rider : stageResults.keySet()) {
264         registerRiderResults(rider, stageResults.get(rider));
265       }
266     }
267   }
268 }
```

5 RaceResult.java

```
1  package cycling;
2
3  import java.io.Serializable;
4  import java.time.Duration;
5  import java.time.LocalTime;
6  import java.util.Comparator;
7
8  /**
9   * This represents a given riders results in a race. The riders adjusted elapsed time, sprinters
10  * points and mountain points over all stages and segments are recorded here.
11  */
12 public class RaceResult implements Serializable {
13   private Duration cumulativeAdjustedElapsedTime = Duration.ZERO;
14   private int cumulativeSprintersPoints = 0;
15   private int cumulativeMountainPoints = 0;
16
17   // A comparator which sorts RaceResults based on Adjusted Elapsed Time in ascending order. The
18   // result with the shortest time will come first.
19   protected static final Comparator<RaceResult> sortByAdjustedElapsedTime =
20       Comparator.comparing(RaceResult::getCumulativeAdjustedElapsedTime);
21
22   // A comparator which sorts RaceResults based on Sprinters Points in descending order. The result
23   // with the most points will come first.
24   protected static final Comparator<RaceResult> sortBySprintersPoints =
25       (RaceResult result1, RaceResult result2) ->
26           Integer.compare(
27               result2.getCumulativeSprintersPoints(), result1.getCumulativeSprintersPoints());
28
29   // A comparator which sorts RaceResults based on Mountain Points in descending order. The result
30   // with the most points will come first.
31   protected static final Comparator<RaceResult> sortByMountainPoints =
32       (RaceResult result1, RaceResult result2) ->
33           Integer.compare(
34               result2.getCumulativeMountainPoints(), result1.getCumulativeMountainPoints());
35
36   /**
37    * A method to get the recorded Adjusted Elapsed Time over all stages.
38    *
39    * @return The cumulative adjusted elapsed time as a duration.
40    */
41   public Duration getCumulativeAdjustedElapsedTime() {
42     return this.cumulativeAdjustedElapsedTime;
43   }
44
```

```
45    /**
46     * A method to get the recorded Adjusted Elapsed Time over all stages as a LocalTime.
47     *
48     * @return The cumulative adjusted elapsed time as a Local Time
49     */
50    public LocalTime getCumulativeAdjustedElapsedLocalTime() {
51      // Calculated the AET as a Local time by adding the duration to midnight: 0:00 + Duration
52      return LocalTime.MIDNIGHT.plus(this.cumulativeAdjustedElapsedTime);
53    }
54
55    /**
56     * A method to get the recorded Mountain Points over all stages and segments.
57     *
58     * @return The cumulative mountain points.
59     */
60    public int getCumulativeMountainPoints() {
61      return this.cumulativeMountainPoints;
62    }
63
64    /**
65     * A method to get the recorded Sprinters Points over all stages and segments.
66     *
67     * @return The cumulative sprinters points.
68     */
69    public int getCumulativeSprintersPoints() {
70      return this.cumulativeSprintersPoints;
71    }
72
73    /**
74     * A method to add a stage result to the race result. This is useful as a riders results in a
      ↪  race
75     * is just a sum of their results in all a races stages. E.g. RaceResults = Stage1Result +
76     * Stage2Result + Stage3Result + ...
77     *
78     * @param stageResult the stage results which should be added to a race result.
79     */
80    public void addStageResult(StageResult stageResult) {
81      this.cumulativeAdjustedElapsedTime =
82          this.cumulativeAdjustedElapsedTime.plus(stageResult.getAdjustedElapsedTime());
83      this.cumulativeSprintersPoints += stageResult.getSprintersPoints();
84      this.cumulativeMountainPoints += stageResult.getMountainPoints();
85    }
86  }
```

6 Rider.java

```
1   package cycling;
2
3   import java.io.Serializable;
4
5   public class Rider implements Serializable {
6     private final Team team;
7     private final String name;
8     private final int yearOfBirth;
9
10    private static int count = 0;
11    private final int id;
```

```
12
13    public Rider(Team team, String name, int yearOfBirth) throws IllegalArgumentException {
14      if (name == null) {
15        throw new java.lang.IllegalArgumentException("The rider's name cannot be null.");
16      }
17      if (yearOfBirth < 1900) {
18        throw new java.lang.IllegalArgumentException(
19            "The rider's birth year is invalid, must be greater than 1900.");
20      }
21
22      this.team = team;
23      this.name = name;
24      this.yearOfBirth = yearOfBirth;
25      this.id = Rider.count++;
26    }
27
28    static void resetIdCounter() {
29      count = 0;
30    }
31
32    static int getIdCounter() {
33      return count;
34    }
35
36    static void setIdCounter(int newCount) {
37      count = newCount;
38    }
39
40    public int getId() {
41      return id;
42    }
43
44    public Team getTeam() {
45      return team;
46    }
47  }
```

7 SavedCyclingPortal.java

```
1   package cycling;
2
3   import java.io.Serializable;
4   import java.util.ArrayList;
5
6   public class SavedCyclingPortal implements Serializable {
7     final ArrayList<Team> teams;
8     final ArrayList<Rider> riders;
9     final ArrayList<Race> races;
10    final ArrayList<Stage> stages;
11    final ArrayList<Segment> segments;
12    final int teamIdCount;
13    final int riderIdCount;
14    final int raceIdCount;
15    final int stageIdCount;
16    final int segmentIdCount;
17
18    public SavedCyclingPortal(
```

```
19        ArrayList<Team> teams,
20        ArrayList<Rider> riders,
21        ArrayList<Race> races,
22        ArrayList<Stage> stages,
23        ArrayList<Segment> segments,
24        int teamIdCount,
25        int riderIdCount,
26        int raceIdCount,
27        int stageIdCount,
28        int segmentIdCount) {
29      this.teams = teams;
30      this.riders = riders;
31      this.races = races;
32      this.stages = stages;
33      this.segments = segments;
34      this.teamIdCount = teamIdCount;
35      this.riderIdCount = riderIdCount;
36      this.raceIdCount = raceIdCount;
37      this.stageIdCount = stageIdCount;
38      this.segmentIdCount = segmentIdCount;
39    }
40  }
```

8 Segment.java

```
1   package cycling;
2
3   import java.io.Serializable;
4   import java.time.LocalTime;
5   import java.util.HashMap;
6   import java.util.List;
7   import java.util.Map;
8   import java.util.stream.Collectors;
9
10  /**
11   * Segment Class. This represents a segment of a stage in a rice in the cycling portal. This deals
12   * with details about the segment as well as well as the segments results.
13   */
14  public class Segment implements Serializable {
15    private static int count = 0;
16    private final Stage stage;
17    private final int id;
18    private final SegmentType type;
19    private final double location;
20
21    private final HashMap<Rider, SegmentResult> results = new HashMap<>();
22
23    // Segment sprinters/mountain points .
24    private static final int[] SPRINT_POINTS = {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
25    private static final int[] HC_POINTS = {20, 15, 12, 10, 8, 6, 4, 2};
26    private static final int[] C1_POINTS = {10, 8, 6, 4, 2, 1};
27    private static final int[] C2_POINTS = {5, 3, 2, 1};
28    private static final int[] C3_POINTS = {2, 1};
29    private static final int[] C4_POINTS = {1};
30
31    /**
32     * Constructor method that creates a segment for a given stage, segment type and location.
```

```
33      *
34      * @param stage The stage object which this segment is in. The stage cannot be waiting for
    ↪  results
35      *     or be a time-trial stage.
36      * @param type The type of segment, can be either SPRINT, C4, C3, C2, C1, or HC.
37      * @param location The location of the segment in the stage in kilometers, cannot be longer than
38      *     the length of the stage.
39      * @throws InvalidLocationException
40      * @throws InvalidStageStateException
41      * @throws InvalidStageTypeException
42      */
43     public Segment(Stage stage, SegmentType type, double location)
44         throws InvalidLocationException, InvalidStageStateException, InvalidStageTypeException {
45       if (location > stage.getLength()) {
46         throw new InvalidLocationException("The location is out of bounds of the stage length.");
47       }
48       if (stage.isWaitingForResults()) {
49         throw new InvalidStageStateException("The stage is waiting for results.");
50       }
51       if (stage.getType().equals(StageType.TT)) {
52         throw new InvalidStageTypeException("Time-trial stages cannot contain any segments.");
53       }
54       this.stage = stage;
55       // ID counter represents the highest known ID at the current time to ensure
56       // there
57       // are no ID collisions.
58       this.id = Segment.count++;
59       this.type = type;
60       this.location = location;
61     }
62
63     /** Reset the static segment ID counter. Used for erasing/loading the CyclingPortal. */
64     static void resetIdCounter() {
65       count = 0;
66     }
67
68     /**
69      * Method to get the current state of the static ID counter.
70      *
71      * @return the highest segment ID stored currently.
72      */
73     static int getIdCounter() {
74       return count;
75     }
76
77     /**
78      * Method that sets the static ID counter to a given value. Used when loading to avoid ID
79      * collisions.
80      *
81      * @param newCount: new value of the static ID counter.
82      */
83     static void setIdCounter(int newCount) {
84       count = newCount;
85     }
86
87     /**
88      * Method to get the ID of the segment object.
89      *
```

```
90      * @return id: the Segments's unique ID value.
91      */
92     public int getId() {
93       return id;
94     }
95
96     /**
97      * Method to get the Stage which the segment exists in.
98      *
99      * @return The stage object.
100     */
101    public Stage getStage() {
102      return stage;
103    }
104
105    /**
106     * Method to get the location of the segment within the stage.
107     *
108     * @return the location in kilometers as a double.
109     */
110    public double getLocation() {
111      return location;
112    }
113
114    /**
115     * Method to register the time which a given rider completed the segment.
116     *
117     * @param rider The rider which finished the segment.
118     * @param finishTime The time which the rider finished the segment.
119     */
120    public void registerResults(Rider rider, LocalTime finishTime) {
121      // Create a segment result for the rider.
122      SegmentResult result = new SegmentResult(finishTime);
123      // Associate the result with the rider in the result HashMap.
124      results.put(rider, result);
125    }
126
127    /**
128     * Method to get a given riders results in this segment.
129     *
130     * @param rider The rider whose results will be returned.
131     * @return The results the rider received in the segment.
132     */
133    public SegmentResult getRiderResult(Rider rider) {
134      // First calculate the segments results, such as riders position and points.
135      calculateResults();
136      // Then return the results for the requested rider.
137      return results.get(rider);
138    }
139
140    /**
141     * Method to remove a given riders results from the segment.
142     *
143     * @param rider The rider object whose results should be removed.
144     */
145    public void removeRiderResults(Rider rider) {
146      results.remove(rider);
147    }
```

```java

  /**
   * Private function to sort all the riders who have results registered by their finish time.
   * Useful for getting each riders position.
   *
   * @return All riders who have a registered result sorted by their finish time.
   */
  private List<Rider> sortRiderResults() {
    // convert the hashmap into a set
    return results.entrySet().stream()
        // Sort the set by the finish time of the results
        .sorted(Map.Entry.comparingByValue(SegmentResult.sortByFinishTime))
        // Get the rider element of the set and ignore the results now they have been
        // sorted and convert to a list.
        .map(Map.Entry::getKey)
        .collect(Collectors.toList());
  }

  /** Private method to calculate the results for this segment. */
  private void calculateResults() {
    // First get a list of riders sorted by their finish time.
    List<Rider> riders = sortRiderResults();

    for (int i = 0; i < results.size(); i++) {
      Rider rider = riders.get(i);
      SegmentResult result = results.get(rider);
      int position = i + 1;
      // Position Calculation
      result.setPosition(position); // Set the riders position

      // Points Calculation
      int[] pointsDistribution =
          getPointsDistribution(); // Get the point distribution based on the segment type.
      if (position <= pointsDistribution.length) {
        // Get the riders points based on their position
        int points = pointsDistribution[i];
        if (this.type.equals(SegmentType.SPRINT)) {
          // If the segment is a sprint, set the riders points as sprinters points.
          result.setSprintersPoints(points);
          result.setMountainPoints(0);
        } else {
          // If the segment is not a sprint, set the riders points as mountain points.
          result.setSprintersPoints(0);
          result.setMountainPoints(points);
        }
      } else {
        // If the rider does not finish in a point-awarding position, reward 0 points.
        result.setMountainPoints(0);
        result.setSprintersPoints(0);
      }
    }
  }

  /**
   * Private method to get the point distribution of the segment based on the type of segment.
   *
   * @return an array of integers that represent the points that should be rewarded based on the
   *     segment type.
```

```
206     */
207   private int[] getPointsDistribution() {
208     return switch (type) {
209       case HC -> HC_POINTS;
210       case C1 -> C1_POINTS;
211       case C2 -> C2_POINTS;
212       case C3 -> C3_POINTS;
213       case C4 -> C4_POINTS;
214       case SPRINT -> SPRINT_POINTS;
215     };
216   }
217 }
```

9 SegmentResult.java

```
1  package cycling;
2
3  import java.io.Serializable;
4  import java.time.LocalTime;
5  import java.util.Comparator;
6
7  public class SegmentResult implements Serializable {
8    private final LocalTime finishTime;
9    private int position;
10   private int sprintersPoints;
11   private int mountainPoints;
12
13   protected static final Comparator<SegmentResult> sortByFinishTime =
14       Comparator.comparing(SegmentResult::getFinishTime);
15
16   public SegmentResult(LocalTime finishTime) {
17     this.finishTime = finishTime;
18   }
19
20   public LocalTime getFinishTime() {
21     return finishTime;
22   }
23
24   public void setPosition(int position) {
25     this.position = position;
26   }
27
28   public void setMountainPoints(int points) {
29     this.mountainPoints = points;
30   }
31
32   public void setSprintersPoints(int points) {
33     this.sprintersPoints = points;
34   }
35
36   public int getMountainPoints() {
37     return this.mountainPoints;
38   }
39
40   public int getSprintersPoints() {
41     return this.sprintersPoints;
42   }
```

```
}
```

10 Stage.java

```
package cycling;

import java.io.Serializable;
import java.time.Duration;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

/** Stage Class. */
public class Stage implements Serializable {
  private final Race race;
  private final String name;
  private final String description;
  private final double length;
  private final LocalDateTime startTime;
  private final StageType type;
  private final int id;
  private static int count = 0;
  private boolean waitingForResults = false;
  private final ArrayList<Segment> segments = new ArrayList<>();

  private final HashMap<Rider, StageResult> results = new HashMap<>();

  private static final int[] FLAT_POINTS = {50, 30, 20, 18, 16, 14, 12, 10, 8, 7, 6, 5, 4, 3, 2};
  private static final int[] MEDIUM_POINTS = {30, 25, 22, 19, 17, 15, 13, 11, 9, 7, 6, 5, 4, 3, 2};
  private static final int[] HIGH_POINTS = {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
  private static final int[] TT_POINTS = {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

  /**
   * Constructor method that sets a Stage up with a race, name, description, length startTime and
   * type.
   *
   * @param race: Race that the Stage is in.
   * @param name: name of the Stage.
   * @param description: description of the Stage.
   * @param length: length of the Stage.
   * @param startTime: start time of the Stage.
   * @param type: the type of Stage.
   * @throws InvalidNameException Thrown if the name is empty, null, longer than 30 characters or
   *     contains whitespace.
   * @throws InvalidLengthException Thrown if the length is less than 5km.
   */
  public Stage(
      Race race,
      String name,
      String description,
      double length,
      LocalDateTime startTime,
      StageType type)
```

```
54        throws InvalidNameException, InvalidLengthException {
55      if (name == null
56          || name.isEmpty()
57          || name.length() > 30
58          || CyclingPortal.containsWhitespace(name)) {
59        throw new InvalidNameException(
60            "Stage name cannot be null, empty, have more than 30 characters or have white spaces.");
61      }
62      if (length < 5) {
63        throw new InvalidLengthException("Length is invalid, cannot be less than 5km.");
64      }
65      this.name = name;
66      this.description = description;
67      this.race = race;
68      this.length = length;
69      this.startTime = startTime;
70      this.type = type;
71      // ID counter represents the highest known ID at the current time to ensure there
72      // are no ID collisions.
73      this.id = Stage.count++;
74    }
75
76    /** Method that resets the static ID counter of the Race. Used for erasing and loading. */
77    static void resetIdCounter() {
78      count = 0;
79    }
80
81    /**
82     * Method to get the current state of the static ID counter.
83     *
84     * @return the highest race ID stored currently.
85     */
86    static int getIdCounter() {
87      return count;
88    }
89
90    /**
91     * Method that sets the static ID counter to a given value. Used when loading to avoid ID
92     * collisions.
93     *
94     * @param newCount: new value of the static ID counter.
95     */
96    static void setIdCounter(int newCount) {
97      count = newCount;
98    }
99
100   /**
101    * Method to get the ID of the Race object.
102    *
103    * @return id: the Race's unique ID value.
104    */
105   public int getId() {
106     return id;
107   }
108
109   /**
110    * Method to get the name of the Stage.
111    *
```

```
112     * @return name: the given name of the Stage.
113     */
114    public String getName() {
115      return name;
116    }
117
118    /**
119     * Method to get the length of the Stage.
120     *
121     * @return length: the given length of the Stage.
122     */
123    public double getLength() {
124      return length;
125    }
126
127    /**
128     * Method to get the Stage's Race.
129     *
130     * @return race: the given Race that the Stage is in.
131     */
132    public Race getRace() {
133      return race;
134    }
135
136    /**
137     * Method to get the Stage's type.
138     *
139     * @return type: the given type of the Stage
140     */
141    public StageType getType() {
142      return type;
143    }
144
145    /**
146     * Method to get the Segments in the Stage.
147     *
148     * @return segments: a list of Segments in the Stage.
149     */
150    public ArrayList<Segment> getSegments() {
151      return segments;
152    }
153
154    /**
155     * Method to get the start time of the Stage.
156     *
157     * @return startTime: the given start time of the Stage.
158     */
159    public LocalDateTime getStartTime() {
160      return startTime;
161    }
162
163    /**
164     * Method that adds a Segment to the Stage. It is added to the list of Segments based on its
165     * location in the Stage.
166     *
167     * @param segment: Segment that will be added to the Stage.
168     */
169    public void addSegment(Segment segment) {
```

```
170     // Loops through the ordered list of segments to find the correct place for the new
171     // Segment to be added.
172     for (int i = 0; i < segments.size(); i++) {
173       // Compares the Segments based on their locations.
174       // The new Segment is inserted if its location is less than the location of the
175       // current Segment it is being compared to.
176       if (segment.getLocation() < segments.get(i).getLocation()) {
177         segments.add(i, segment);
178         return;
179       }
180     }
181     segments.add(segment);
182   }
183
184   /**
185    * Method that removes a given Segment from the Stage's Segments.
186    *
187    * @param segment: the Segment intended to be removed.
188    * @throws InvalidStageStateException Thrown if the Stage is waiting for results.
189    */
190   public void removeSegment(Segment segment) throws InvalidStageStateException {
191     if (waitingForResults) {
192       throw new InvalidStageStateException(
193           "The segment cannot be removed as it is waiting for results.");
194     }
195     segments.remove(segment);
196   }
197
198   /**
199    * Method that registers a Rider's result and adds it to the Stage.
200    *
201    * @param rider: the Rider whose results will be registered.
202    * @param checkpoints: the Rider's results.
203    * @throws InvalidStageStateException Thrown if the Stage is not waiting for results.
204    * @throws DuplicatedResultException Thrown if the Rider already has results registered in the
205    *     Stage.
206    * @throws InvalidCheckpointsException Thrown if the number checkpoints doesn't equal the number
207    *     of Segments in the Stage + 2
208    */
209   public void registerResult(Rider rider, LocalTime[] checkpoints)
210       throws InvalidStageStateException, DuplicatedResultException, InvalidCheckpointsException {
211     if (!waitingForResults) {
212       throw new InvalidStageStateException(
213           "Results can only be added to a stage while it is waiting for results.");
214     }
215     if (results.containsKey(rider)) {
216       throw new DuplicatedResultException("Each rider can only have one result per Stage.");
217     }
218     if (checkpoints.length != segments.size() + 2) {
219       throw new InvalidCheckpointsException(
220           "The length of the checkpoint must equal the number of Segments in the Stage + 2.");
221     }
222
223     StageResult result = new StageResult(checkpoints);
224     // Save Riders result for the Stage
225     results.put(rider, result);
226
227     // Propagate all the Riders results for each segment
```

```
228       for (int i = 0; i < segments.size(); i++) {
229         segments.get(i).registerResults(rider, checkpoints[i + 1]);
230       }
231     }
232
233     /**
234      * Method that concludes the Stage preparation and ensures that the Stage is now waiting for
235      * results.
236      *
237      * @throws InvalidStageStateException Thrown if the Stage is already waiting for results.
238      */
239     public void concludePreparation() throws InvalidStageStateException {
240       if (waitingForResults) {
241         throw new InvalidStageStateException("Stage is already waiting for results.");
242       }
243       waitingForResults = true;
244     }
245
246     /**
247      * Method to identify whether the Stage is waiting for results.
248      *
249      * @return A boolean, true if the Stage is waiting for results, false if it is not.
250      */
251     public boolean isWaitingForResults() {
252       return waitingForResults;
253     }
254
255     /**
256      * Method to calculate and return the results of a given Rider.
257      *
258      * @param rider: Rider whose results are desired.
259      * @return results of the Rider.
260      */
261     public StageResult getRiderResult(Rider rider) {
262       calculateResults();
263       return results.get(rider);
264     }
265
266     /**
267      * Method to remove the results of a Rider.
268      *
269      * @param rider whose results are to be removed.
270      */
271     public void removeRiderResults(Rider rider) {
272       results.remove(rider);
273     }
274
275     /**
276      * Method to
277      *
278      * @return
279      */
280     public List<Rider> getRidersByElapsedTime() {
281       calculateResults();
282       return sortRiderResults();
283     }
284
285     public HashMap<Rider, StageResult> getStageResults() {
```

```
286     calculateResults();
287     return results;
288   }
289
290   private List<Rider> sortRiderResults() {
291     return results.entrySet().stream()
292         .sorted(Map.Entry.comparingByValue(StageResult.sortByElapsedTime))
293         .map(Map.Entry::getKey)
294         .collect(Collectors.toList());
295   }
296
297   private void calculateResults() {
298     List<Rider> riders = sortRiderResults();
299
300     for (int i = 0; i < results.size(); i++) {
301       Rider rider = riders.get(i);
302       StageResult result = results.get(rider);
303       int position = i + 1;
304
305       // Position Calculation
306       result.setPosition(position);
307
308       // Adjusted Elapsed Time Calculations
309       if (i == 0) {
310         result.setAdjustedElapsedTime(result.getElapsedTime());
311       } else {
312         Rider prevRider = riders.get(i - 1);
313         Duration prevTime = results.get(prevRider).getElapsedTime();
314         Duration time = results.get(rider).getElapsedTime();
315
316         int timeDiff = time.minus(prevTime).compareTo(Duration.ofSeconds(1));
317         if (timeDiff <= 0) {
318           // Close Finish Condition
319           Duration prevAdjustedTime = results.get(prevRider).getAdjustedElapsedTime();
320           result.setAdjustedElapsedTime(prevAdjustedTime);
321         } else {
322           // Far Finish Condition
323           result.setAdjustedElapsedTime(time);
324         }
325       }
326
327       // Points Calculation
328       int sprintersPoints = 0;
329       int mountainPoints = 0;
330       for (Segment segment : segments) {
331         SegmentResult segmentResult = segment.getRiderResult(rider);
332         sprintersPoints += segmentResult.getSprintersPoints();
333         mountainPoints += segmentResult.getMountainPoints();
334       }
335       int[] pointsDistribution = getPointDistribution();
336       if (position <= pointsDistribution.length) {
337         sprintersPoints += pointsDistribution[i];
338       }
339       result.setSprintersPoints(sprintersPoints);
340       result.setMountainPoints(mountainPoints);
341     }
342   }
343
```

```
344   private int[] getPointDistribution() {
345     return switch (type) {
346       case FLAT -> FLAT_POINTS;
347       case MEDIUM_MOUNTAIN -> MEDIUM_POINTS;
348       case HIGH_MOUNTAIN -> HIGH_POINTS;
349       case TT -> TT_POINTS;
350     };
351   }
352 }
```

11 StageResult.java

```
1  package cycling;
2
3  import java.io.Serializable;
4  import java.time.Duration;
5  import java.time.LocalTime;
6  import java.util.Comparator;
7
8  public class StageResult implements Serializable {
9    private final LocalTime[] checkpoints;
10   private final Duration elapsedTime;
11   private Duration adjustedElapsedTime;
12   private int position;
13   private int sprintersPoints;
14   private int mountainPoints;
15
16   protected static final Comparator<StageResult> sortByElapsedTime =
17       Comparator.comparing(StageResult::getElapsedTime);
18
19   public StageResult(LocalTime[] checkpoints) {
20     this.checkpoints = checkpoints;
21     this.elapsedTime = Duration.between(checkpoints[0], checkpoints[checkpoints.length - 1]);
22   }
23
24   public LocalTime[] getCheckpoints() {
25     return this.checkpoints;
26   }
27
28   public Duration getElapsedTime() {
29     return elapsedTime;
30   }
31
32   public void setPosition(int position) {
33     this.position = position;
34   }
35
36   public void setAdjustedElapsedTime(Duration adjustedElapsedTime) {
37     this.adjustedElapsedTime = adjustedElapsedTime;
38   }
39
40   public Duration getAdjustedElapsedTime() {
41     return adjustedElapsedTime;
42   }
43
44   public LocalTime getAdjustedElapsedLocalTime() {
45     return checkpoints[0].plus(adjustedElapsedTime);
```

```
46    }
47
48    public void setMountainPoints(int points) {
49      this.mountainPoints = points;
50    }
51
52    public void setSprintersPoints(int points) {
53      this.sprintersPoints = points;
54    }
55
56    public int getMountainPoints() {
57      return mountainPoints;
58    }
59
60    public int getSprintersPoints() {
61      return sprintersPoints;
62    }
63
64    // --Commented out by Inspection START (28/03/2022, 3:31 pm):
65    //  public void add(StageResult res){
66    //    this.elapsedTime = this.elapsedTime.plus(res.getElapsedTime());
67    //    this.adjustedElapsedTime = this.adjustedElapsedTime.plus(res.getAdjustedElapsedTime());
68    //    this.sprintersPoints += res.getSprintersPoints();
69    //    this.mountainPoints += res.getMountainPoints();
70    //  }
71    // --Commented out by Inspection STOP (28/03/2022, 3:31 pm)
72  }
```

12 Team.java

```
1   package cycling;
2
3   import java.io.Serializable;
4   import java.util.ArrayList;
5
6   public class Team implements Serializable {
7     private final String name;
8     private final String description;
9
10    private final ArrayList<Rider> riders = new ArrayList<>();
11    private static int count = 0;
12    private final int id;
13
14    public Team(String name, String description) throws InvalidNameException {
15      if (name == null
16          || name.isEmpty()
17          || name.length() > 30
18          || CyclingPortal.containsWhitespace(name)) {
19        throw new InvalidNameException(
20            "Team name cannot be null, empty, have more than 30 characters or have white spaces.");
21      }
22      this.name = name;
23      this.description = description;
24      this.id = Team.count++;
25    }
26
27    static void resetIdCounter() {
```

```
28      count = 0;
29    }
30
31    static int getIdCounter() {
32      return count;
33    }
34
35    static void setIdCounter(int newCount) {
36      count = newCount;
37    }
38
39    public String getName() {
40      return name;
41    }
42
43    public int getId() {
44      return id;
45    }
46
47    public void removeRider(Rider rider) {
48      riders.remove(rider);
49    }
50
51    public ArrayList<Rider> getRiders() {
52      return riders;
53    }
54
55    public void addRider(Rider rider) {
56      riders.add(rider);
57    }
58  }
```