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:
//    - Asserts !!!!
//    - 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;
  }

  /**
```

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

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

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

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

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

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

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

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

```
503             Stage.getIdCounter(),
504             Segment.getIdCounter());
505     output.writeObject(savedCyclingPortal);
506     output.close();
507     file.close();
508   }
509
510   @Override
511   public void loadCyclingPortal(String filename) throws IOException, ClassNotFoundException {
512     eraseCyclingPortal();
513     FileInputStream file = new FileInputStream(filename + ".ser");
514     ObjectInputStream input = new ObjectInputStream(file);
515
516     SavedCyclingPortal savedCyclingPortal = (SavedCyclingPortal) input.readObject();
517     // Imports teams, riders, races, stages and segments ArrayLists from the last save.
518     teams = savedCyclingPortal.teams;
519     riders = savedCyclingPortal.riders;
520     races = savedCyclingPortal.races;
521     stages = savedCyclingPortal.stages;
522     segments = savedCyclingPortal.segments;
523
524     // Imports ID counters of Team, Rider, Race, Stage and Segment objects from the last save.
525     Team.setIdCounter(savedCyclingPortal.teamIdCount);
526     Rider.setIdCounter(savedCyclingPortal.riderIdCount);
527     Race.setIdCounter(savedCyclingPortal.raceIdCount);
528     Stage.setIdCounter(savedCyclingPortal.stageIdCount);
529     Segment.setIdCounter(savedCyclingPortal.segmentIdCount);
530
531     input.close();
532     file.close();
533   }
534
535   @Override
536   public void removeRaceByName(String name) throws NameNotRecognisedException {
537     for (final Race race : races) {
538       if (race.getName().equals(name)) {
539         races.remove(race);
540         return;
541       }
542     }
543     throw new NameNotRecognisedException("Race name is not in the system.");
544   }
545
546   @Override
547   public int[] getRidersGeneralClassificationRank(int raceId) throws IDNotRecognisedException {
548     Race race = getRaceById(raceId);
549     List<Rider> riders = race.getRidersByAdjustedElapsedTime();
550     int[] riderIds = new int[riders.size()];
551     // Gathers Rider ID's ordered by their Adjusted Elapsed Times.
552     for (int i = 0; i < riders.size(); i++) {
553       riderIds[i] = riders.get(i).getId();
554     }
555     return riderIds;
556   }
557
558   @Override
559   public LocalTime[] getGeneralClassificationTimesInRace(int raceId)
560       throws IDNotRecognisedException {
```

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

```
619     return riderIds;
620   }
621 }
```

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 int 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 String 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     */
```

```java
  public void addStage(Stage stage) {
    // Loops over stages in the race to insert the new stage in the correct place such that
    // all of the stages are sorted by their start time.
    for (int i = 0; i < stages.size(); i++) {
      // Retrieves the start time of each Stage in the Race's current Stages one by one.
      // These are already ordered by their start times.
      LocalDateTime iStartTime = stages.get(i).getStartTime();
      // Adds the new Stage to the list of stages in the correct position based on
      // its start time.
      if (stage.getStartTime().isBefore(iStartTime)) {
        stages.add(i, stage);
        return;
      }
    }
    stages.add(stage);
  }

  /**
   * Method to get the list of Stages in the Race ordered by their start times.
   *
   * @return Arraylist<Stages> stages: The ordered list of Stages.
   */
  public ArrayList<Stage> getStages() {
    // stages is already sorted, so no sorting needs to be done.
    return stages;
  }

  /**
   * Method that removes a given Stage from the list of Stages.
   *
   * @param stage: the Stage to be deleted.
   */
  public void removeStage(Stage stage) {
    stages.remove(stage);
  }

  /**
   * Method to get then details of a Race including Race ID, name, description number of stages and
   * total length.
   *
   * @return String: concatenated paragraph of race details.
   */
  public String getDetails() {
    double currentLength = 0;
    for (final Stage stage : stages) {
      currentLength = currentLength + stage.getLength();
    }
    return ("Race ID: "
        + id
        + System.lineSeparator()
        + "Name: "
        + name
        + System.lineSeparator()
        + "Description: "
        + description
        + System.lineSeparator()
        + "Number of Stages: "
        + 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 List<Rider>: 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 List<Rider>: 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 List<Rider>: 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

```java
package cycling;

import java.io.Serializable;
import java.time.Duration;
import java.time.LocalTime;
import java.util.Comparator;

public class RaceResult implements Serializable {
  private Duration cumulativeAdjustedElapsedTime = Duration.ZERO;
  private int cumulativeSprintersPoints = 0;
  private int cumulativeMountainPoints = 0;

  protected static final Comparator<RaceResult> sortByAdjustedElapsedTime =
      Comparator.comparing(RaceResult::getCumulativeAdjustedElapsedTime);

  protected static final Comparator<RaceResult> sortBySprintersPoints =
      (RaceResult result1, RaceResult result2) ->
          Integer.compare(
              result2.getCumulativeSprintersPoints(), result1.getCumulativeSprintersPoints());

  protected static final Comparator<RaceResult> sortByMountainPoints =
      (RaceResult result1, RaceResult result2) ->
          Integer.compare(
              result2.getCumulativeMountainPoints(), result1.getCumulativeMountainPoints());

  public Duration getCumulativeAdjustedElapsedTime() {
    return this.cumulativeAdjustedElapsedTime;
  }

  public LocalTime getCumulativeAdjustedElapsedLocalTime() {
    return LocalTime.MIDNIGHT.plus(this.cumulativeAdjustedElapsedTime);
  }

  public int getCumulativeMountainPoints() {
    return this.cumulativeMountainPoints;
  }

  public int getCumulativeSprintersPoints() {
    return this.cumulativeSprintersPoints;
  }

  public void addStageResult(StageResult stageResult) {
    this.cumulativeAdjustedElapsedTime =
        this.cumulativeAdjustedElapsedTime.plus(stageResult.getAdjustedElapsedTime());
    this.cumulativeSprintersPoints += stageResult.getSprintersPoints();
    this.cumulativeMountainPoints += stageResult.getMountainPoints();
  }
}
```

6 Rider.java

```java
package cycling;

import java.io.Serializable;

```

```java
public class Rider implements Serializable {
  private final Team team;
  private final String name;
  private final int yearOfBirth;

  private static int count = 0;
  private final int id;

  public Rider(Team team, String name, int yearOfBirth) throws IllegalArgumentException {
    if (name == null) {
      throw new java.lang.IllegalArgumentException("The rider's name cannot be null.");
    }
    if (yearOfBirth < 1900) {
      throw new java.lang.IllegalArgumentException(
          "The rider's birth year is invalid, must be greater than 1900.");
    }

    this.team = team;
    this.name = name;
    this.yearOfBirth = yearOfBirth;
    this.id = Rider.count++;
  }

  static void resetIdCounter() {
    count = 0;
  }

  static int getIdCounter() {
    return count;
  }

  static void setIdCounter(int newCount) {
    count = newCount;
  }

  public int getId() {
    return id;
  }

  public Team getTeam() {
    return team;
  }
}
```

7 SavedCyclingPortal.java

```java
package cycling;

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

public class SavedCyclingPortal implements Serializable {
  final ArrayList<Team> teams;
  final ArrayList<Rider> riders;
  final ArrayList<Race> races;
  final ArrayList<Stage> stages;
  final ArrayList<Segment> segments;
```

```
  final int teamIdCount;
  final int riderIdCount;
  final int raceIdCount;
  final int stageIdCount;
  final int segmentIdCount;

  public SavedCyclingPortal(
      ArrayList<Team> teams,
      ArrayList<Rider> riders,
      ArrayList<Race> races,
      ArrayList<Stage> stages,
      ArrayList<Segment> segments,
      int teamIdCount,
      int riderIdCount,
      int raceIdCount,
      int stageIdCount,
      int segmentIdCount) {
    this.teams = teams;
    this.riders = riders;
    this.races = races;
    this.stages = stages;
    this.segments = segments;
    this.teamIdCount = teamIdCount;
    this.riderIdCount = riderIdCount;
    this.raceIdCount = raceIdCount;
    this.stageIdCount = stageIdCount;
    this.segmentIdCount = segmentIdCount;
  }
}
```

8 Segment.java

```
package cycling;

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

public class Segment implements Serializable {
  private static int count = 0;
  private final Stage stage;
  private final int id;
  private final SegmentType type;
  private final double location;

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

  private static final int[] SPRINT_POINTS = {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
  private static final int[] HC_POINTS = {20, 15, 12, 10, 8, 6, 4, 2};
  private static final int[] C1_POINTS = {10, 8, 6, 4, 2, 1};
  private static final int[] C2_POINTS = {5, 3, 2, 1};
  private static final int[] C3_POINTS = {2, 1};
  private static final int[] C4_POINTS = {1};

```

```
26    public Segment(Stage stage, SegmentType type, double location)
27        throws InvalidLocationException, InvalidStageStateException, InvalidStageTypeException {
28      if (location > stage.getLength()) {
29        throw new InvalidLocationException("The location is out of bounds of the stage length.");
30      }
31      if (stage.isWaitingForResults()) {
32        throw new InvalidStageStateException("The stage is waiting for results.");
33      }
34      if (stage.getType().equals(StageType.TT)) {
35        throw new InvalidStageTypeException("Time-trial stages cannot contain any segments.");
36      }
37      this.stage = stage;
38      this.id = Segment.count++;
39      this.type = type;
40      this.location = location;
41    }
42
43    static void resetIdCounter() {
44      count = 0;
45    }
46
47    static int getIdCounter() {
48      return count;
49    }
50
51    static void setIdCounter(int newCount) {
52      count = newCount;
53    }
54
55    public int getId() {
56      return id;
57    }
58
59    public Stage getStage() {
60      return stage;
61    }
62
63    public double getLocation() {
64      return location;
65    }
66
67    public void registerResults(Rider rider, LocalTime finishTime) {
68      SegmentResult result = new SegmentResult(finishTime);
69      results.put(rider, result);
70    }
71
72    public SegmentResult getRiderResult(Rider rider) {
73      calculateResults();
74      return results.get(rider);
75    }
76
77    public void removeRiderResults(Rider rider) {
78      results.remove(rider);
79    }
80
81    private List<Rider> sortRiderResults() {
82      return results.entrySet().stream()
83          .sorted(Map.Entry.comparingByValue(SegmentResult.sortByFinishTime))
```

```
84          .map(Map.Entry::getKey)
85          .collect(Collectors.toList());
86    }
87
88    private void calculateResults() {
89      List<Rider> riders = sortRiderResults();
90
91      for (int i = 0; i < results.size(); i++) {
92        Rider rider = riders.get(i);
93        SegmentResult result = results.get(rider);
94        int position = i + 1;
95        // Position Calculation
96        result.setPosition(position);
97
98        // Points Calculation
99        int[] pointsDistribution = getPointsDistribution();
100       if (position <= pointsDistribution.length) {
101         int points = pointsDistribution[i];
102         if (this.type.equals(SegmentType.SPRINT)) {
103           result.setSprintersPoints(points);
104           result.setMountainPoints(0);
105         } else {
106           result.setSprintersPoints(0);
107           result.setMountainPoints(points);
108         }
109       } else {
110         result.setMountainPoints(0);
111         result.setSprintersPoints(0);
112       }
113     }
114   }
115
116   private int[] getPointsDistribution() {
117     return switch (type) {
118       case HC -> HC_POINTS;
119       case C1 -> C1_POINTS;
120       case C2 -> C2_POINTS;
121       case C3 -> C3_POINTS;
122       case C4 -> C4_POINTS;
123       case SPRINT -> SPRINT_POINTS;
124     };
125   }
126 }
```

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    }
43  }
```

10 Stage.java

```
1   package cycling;
2
3   import java.io.Serializable;
4   import java.time.Duration;
5   import java.time.LocalDateTime;
6   import java.time.LocalTime;
7   import java.util.ArrayList;
8   import java.util.HashMap;
9   import java.util.List;
10  import java.util.Map;
11  import java.util.stream.Collectors;
12
13  public class Stage implements Serializable {
14    private final Race race;
15    private final String name;
16    private final String description;
17    private final double length;
18    private final LocalDateTime startTime;
19    private final StageType type;
20    private final int id;
21    private static int count = 0;
22    private boolean waitingForResults = false;
```

```
23    private final ArrayList<Segment> segments = new ArrayList<>();
24
25    private final HashMap<Rider, StageResult> results = new HashMap<>();
26
27    private static final int[] FLAT_POINTS = {50, 30, 20, 18, 16, 14, 12, 10, 8, 7, 6, 5, 4, 3, 2};
28    private static final int[] MEDIUM_POINTS = {30, 25, 22, 19, 17, 15, 13, 11, 9, 7, 6, 5, 4, 3, 2};
29    private static final int[] HIGH_POINTS = {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
30    private static final int[] TT_POINTS = {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
31
32    public Stage(
33        Race race,
34        String name,
35        String description,
36        double length,
37        LocalDateTime startTime,
38        StageType type)
39        throws InvalidNameException, InvalidLengthException {
40      if (name == null
41          || name.isEmpty()
42          || name.length() > 30
43          || CyclingPortal.containsWhitespace(name)) {
44        throw new InvalidNameException(
45            "Stage name cannot be null, empty, have more than 30 characters or have white spaces.");
46      }
47      if (length < 5) {
48        throw new InvalidLengthException("Length is invalid, cannot be less than 5km.");
49      }
50      this.name = name;
51      this.description = description;
52      this.race = race;
53      this.length = length;
54      this.startTime = startTime;
55      this.type = type;
56      this.id = Stage.count++;
57    }
58
59    static void resetIdCounter() {
60      count = 0;
61    }
62
63    static int getIdCounter() {
64      return count;
65    }
66
67    static void setIdCounter(int newCount) {
68      count = newCount;
69    }
70
71    public int getId() {
72      return id;
73    }
74
75    public String getName() {
76      return name;
77    }
78
79    public double getLength() {
80      return length;
```

```
81    }
82
83    public Race getRace() {
84      return race;
85    }
86
87    public StageType getType() {
88      return type;
89    }
90
91    public ArrayList<Segment> getSegments() {
92      return segments;
93    }
94
95    public LocalDateTime getStartTime() {
96      return startTime;
97    }
98
99    public void addSegment(Segment segment) {
100     for (int i = 0; i < segments.size(); i++) {
101       if (segment.getLocation() < segments.get(i).getLocation()) {
102         segments.add(i, segment);
103         return;
104       }
105     }
106     segments.add(segment);
107   }
108
109   public void removeSegment(Segment segment) throws InvalidStageStateException {
110     if (waitingForResults) {
111       throw new InvalidStageStateException(
112           "The stage cannot be removed as it is waiting for results.");
113     }
114     segments.remove(segment);
115   }
116
117   public void registerResult(Rider rider, LocalTime[] checkpoints)
118       throws InvalidStageStateException, DuplicatedResultException, InvalidCheckpointsException {
119     if (!waitingForResults) {
120       throw new InvalidStageStateException(
121           "Results can only be added to a stage while it is waiting for results.");
122     }
123     if (results.containsKey(rider)) {
124       throw new DuplicatedResultException("Each rider can only have one result per Stage.");
125     }
126     if (checkpoints.length != segments.size() + 2) {
127       throw new InvalidCheckpointsException(
128           "The length of the checkpoint must equal number of Segments in the Stage + 2.");
129     }
130
131     StageResult result = new StageResult(checkpoints);
132     // Save Riders result for the Stage
133     results.put(rider, result);
134
135     // Propagate all the Riders results for each segment
136     for (int i = 0; i < segments.size(); i++) {
137       segments.get(i).registerResults(rider, checkpoints[i + 1]);
138     }
```

```
139   }
140
141   public void concludePreparation() throws InvalidStageStateException {
142     if (waitingForResults) {
143       throw new InvalidStageStateException("Stage is already waiting for results.");
144     }
145     waitingForResults = true;
146   }
147
148   public boolean isWaitingForResults() {
149     return waitingForResults;
150   }
151
152   public StageResult getRiderResult(Rider rider) {
153     calculateResults();
154     return results.get(rider);
155   }
156
157   public void removeRiderResults(Rider rider) {
158     results.remove(rider);
159   }
160
161   public List<Rider> getRidersByElapsedTime() {
162     calculateResults();
163     return sortRiderResults();
164   }
165
166   public HashMap<Rider, StageResult> getStageResults() {
167     calculateResults();
168     return results;
169   }
170
171   private List<Rider> sortRiderResults() {
172     return results.entrySet().stream()
173         .sorted(Map.Entry.comparingByValue(StageResult.sortByElapsedTime))
174         .map(Map.Entry::getKey)
175         .collect(Collectors.toList());
176   }
177
178   private void calculateResults() {
179     List<Rider> riders = sortRiderResults();
180
181     for (int i = 0; i < results.size(); i++) {
182       Rider rider = riders.get(i);
183       StageResult result = results.get(rider);
184       int position = i + 1;
185
186       // Position Calculation
187       result.setPosition(position);
188
189       // Adjusted Elapsed Time Calculations
190       if (i == 0) {
191         result.setAdjustedElapsedTime(result.getElapsedTime());
192       } else {
193         Rider prevRider = riders.get(i - 1);
194         Duration prevTime = results.get(prevRider).getElapsedTime();
195         Duration time = results.get(rider).getElapsedTime();
196
```

```
197           int timeDiff = time.minus(prevTime).compareTo(Duration.ofSeconds(1));
198           if (timeDiff <= 0) {
199             // Close Finish Condition
200             Duration prevAdjustedTime = results.get(prevRider).getAdjustedElapsedTime();
201             result.setAdjustedElapsedTime(prevAdjustedTime);
202           } else {
203             // Far Finish Condition
204             result.setAdjustedElapsedTime(time);
205           }
206         }
207
208         // Points Calculation
209         int sprintersPoints = 0;
210         int mountainPoints = 0;
211         for (Segment segment : segments) {
212           SegmentResult segmentResult = segment.getRiderResult(rider);
213           sprintersPoints += segmentResult.getSprintersPoints();
214           mountainPoints += segmentResult.getMountainPoints();
215         }
216         int[] pointsDistribution = getPointDistribution();
217         if (position <= pointsDistribution.length) {
218           sprintersPoints += pointsDistribution[i];
219         }
220         result.setSprintersPoints(sprintersPoints);
221         result.setMountainPoints(mountainPoints);
222       }
223     }
224
225     private int[] getPointDistribution() {
226       return switch (type) {
227         case FLAT -> FLAT_POINTS;
228         case MEDIUM_MOUNTAIN -> MEDIUM_POINTS;
229         case HIGH_MOUNTAIN -> HIGH_POINTS;
230         case TT -> TT_POINTS;
231       };
232     }
233   }
```

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

```java
package cycling;

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

public class Team implements Serializable {
  private final String name;
  private final String description;

  private final ArrayList<Rider> riders = new ArrayList<>();
  private static int count = 0;
  private final int id;

  public Team(String name, String description) throws InvalidNameException {
    if (name == null
        || name.isEmpty()
        || name.length() > 30
        || CyclingPortal.containsWhitespace(name)) {
      throw new InvalidNameException(
          "Team name cannot be null, empty, have more than 30 characters or have white spaces.");
    }
    this.name = name;
    this.description = description;
    this.id = Team.count++;
  }

  static void resetIdCounter() {
    count = 0;
  }

  static int getIdCounter() {
    return count;
  }

  static void setIdCounter(int newCount) {
    count = newCount;
  }

  public String getName() {
    return name;
  }

  public int getId() {
    return id;
  }

  public void removeRider(Rider rider) {
    riders.remove(rider);
  }

  public ArrayList<Rider> getRiders() {
    return riders;
  }

  public void addRider(Rider rider) {
    riders.add(rider);
```

```
57    }
58 }
```