

CyclingPortal Printout

123456789 & 987654321

Contents

1	CategorizedClimb.java	2
2	CyclingPortal.java	2
3	IntermediateSprint.java	13
4	Race.java	13
5	RaceResult.java	18
6	Rider.java	19
7	SavedCyclingPortal.java	19
8	Segment.java	20
9	SegmentResult.java	22
10	Stage.java	23
11	StageResult.java	27
12	Team.java	29

1 CategorizedClimb.java

```
1 package cycling;
2
3 public class CategorizedClimb extends Segment {
4     private final Double averageGradient;
5     private final Double length;
6
7     public CategorizedClimb(
8         Stage stage, Double location, SegmentType type, Double averageGradient, Double length)
9         throws InvalidLocationException, InvalidStageStateException, InvalidStageTypeException {
10        super(stage, type, location);
11        this.averageGradient = averageGradient;
12        this.length = length;
13    }
14 }
```

2 CyclingPortal.java

```
1 package cycling;
2
3 import java.io.*;
4 import java.time.LocalDateTime;
5 import java.time.LocalTime;
6 import java.util.ArrayList;
7 import java.util.List;
8
9 // TODO:
10 //     - Asserts !!!!
11 //     - Code Formatting
12 //     - Documentation/Comments
13 //     - Testing
14 //     - each function public/private/protected/default
15 //     - Optimise results?
16
17 public class CyclingPortal implements CyclingPortalInterface {
18     // ArrayLists for all of a cycling portal instances teams, riders, races, stages and segments.
19     // Although HashMaps could have been used here to get riders by int ID, it would be slower in the
20     // long run as we would need to constantly convert it back to arrays to output results.
21     private ArrayList<Team> teams = new ArrayList<>();
22     private ArrayList<Rider> riders = new ArrayList<>();
23     private ArrayList<Race> races = new ArrayList<>();
24     private ArrayList<Stage> stages = new ArrayList<>();
25     private ArrayList<Segment> segments = new ArrayList<>();
26
27     /**
28      * Determine if a string contains any illegal whitespace characters.
29      *
30      * @param string The input string to be tested for whitespace.
31      * @return A boolean, true if the input string contains whitespace, false if not.
32      */
33     public static boolean containsWhitespace(String string) {
34         for (int i = 0; i < string.length(); ++i) {
35             if (Character.isWhitespace(string.charAt(i))) {
36                 return true;
37             }
38         }
39     }
```

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

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

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

```
213     // Gathers the Stage ID's of the Stages in the Race.
214     for (int i = 0; i < raceStages.size(); i++) {
215         Stage stage = race.getStages().get(i);
216         raceStagesId[i] = stage.getId();
217     }
218     return raceStagesId;
219 }
220
221 @Override
222 public double getStageLength(int stageId) throws IDNotRecognisedException {
223     Stage stage = getStageById(stageId);
224     return stage.getLength();
225 }
226
227 @Override
228 public void removeStageById(int stageId) throws IDNotRecognisedException {
229     Stage stage = getStageById(stageId);
230     Race race = stage.getRace();
231     // Removes stage from both the Races and Stages.
232     race.removeStage(stage);
233     stages.remove(stage);
234 }
235
236 @Override
237 public int addCategorizedClimbToStage(
238     int stageId, Double location, SegmentType type, Double averageGradient, Double length)
239     throws IDNotRecognisedException, InvalidLocationException, InvalidStageStateException,
240     InvalidStageTypeException {
241     Stage stage = getStageById(stageId);
242     CategorizedClimb climb = new CategorizedClimb(stage, location, type, averageGradient, length);
243     // Adds Categorized Climb to both the list of Segments and the Stage.
244     segments.add(climb);
245     stage.addSegment(climb);
246     return climb.getId();
247 }
248
249 @Override
250 public int addIntermediateSprintToStage(int stageId, double location)
251     throws IDNotRecognisedException, InvalidLocationException, InvalidStageStateException,
252     InvalidStageTypeException {
253     Stage stage = getStageById(stageId);
254     IntermediateSprint sprint = new IntermediateSprint(stage, location);
255     // Adds Intermediate Sprint to both the list of Segments and the Stage.
256     segments.add(sprint);
257     stage.addSegment(sprint);
258     return sprint.getId();
259 }
260
261 @Override
262 public void removeSegment(int segmentId)
263     throws IDNotRecognisedException, InvalidStageStateException {
264     Segment segment = getSegmentById(segmentId);
265     Stage stage = segment.getStage();
266     // Removes Segment from both the Stage and list of Segments.
267     stage.removeSegment(segment);
268     segments.remove(segment);
269 }
270
```

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

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



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

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

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

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

```

619     // Gathers Rider ID's ordered by their Mountain Points.
620     for (int i = 0; i < riders.size(); i++) {
621         riderIds[i] = riders.get(i).getId();
622     }
623     return riderIds;
624 }
625 }

```

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     */
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 ArrayList<Stages> 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 String: 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 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  }
```

```
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.LocalDateTime;
6  import java.util.Comparator;
7
8  public class RaceResult implements Serializable {
9      private Duration cumulativeAdjustedElapsedTime = Duration.ZERO;
10     private int cumulativeSprintersPoints = 0;
11     private int cumulativeMountainPoints = 0;
12
13     protected static final Comparator<RaceResult> sortByAdjustedElapsedTime =
14         Comparator.comparing(RaceResult::getCumulativeAdjustedElapsedTime);
15
16     protected static final Comparator<RaceResult> sortBySprintersPoints =
17         (RaceResult result1, RaceResult result2) ->
18             Integer.compare(
19                 result2.getCumulativeSprintersPoints(), result1.getCumulativeSprintersPoints());
20
21     protected static final Comparator<RaceResult> sortByMountainPoints =
22         (RaceResult result1, RaceResult result2) ->
23             Integer.compare(
24                 result2.getCumulativeMountainPoints(), result1.getCumulativeMountainPoints());
25
26     public Duration getCumulativeAdjustedElapsedTime() {
27         return this.cumulativeAdjustedElapsedTime;
28     }
29
30     public LocalDateTime getCumulativeAdjustedElapsedLocalTime() {
31         return LocalDateTime.MIDNIGHT.plus(this.cumulativeAdjustedElapsedTime);
32     }
33
34     public int getCumulativeMountainPoints() {
35         return this.cumulativeMountainPoints;
36     }
37
38     public int getCumulativeSprintersPoints() {
39         return this.cumulativeSprintersPoints;
40     }
41
42     public void addStageResult(StageResult stageResult) {
43         this.cumulativeAdjustedElapsedTime =
44             this.cumulativeAdjustedElapsedTime.plus(stageResult.getAdjustedElapsedTime());
45         this.cumulativeSprintersPoints += stageResult.getSprintersPoints();
46         this.cumulativeMountainPoints += stageResult.getMountainPoints();
47     }
48 }
```

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.LocalDateTime;
5 import java.util.HashMap;
6 import java.util.List;
7 import java.util.Map;
8 import java.util.stream.Collectors;
9
10 public class Segment implements Serializable {
11     private static int count = 0;
12     private final Stage stage;
13     private final int id;
14     private final SegmentType type;
15     private final double location;
16
17     private final HashMap<Rider, SegmentResult> results = new HashMap<>();
18
19     private static final int[] SPRINT_POINTS = {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
```

```
20 private static final int[] HC_POINTS = {20, 15, 12, 10, 8, 6, 4, 2};
21 private static final int[] C1_POINTS = {10, 8, 6, 4, 2, 1};
22 private static final int[] C2_POINTS = {5, 3, 2, 1};
23 private static final int[] C3_POINTS = {2, 1};
24 private static final int[] C4_POINTS = {1};
25
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.LocalDateTime;
5 import java.util.Comparator;
```

```
6
7 public class SegmentResult implements Serializable {
8     private final LocalDateTime 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(LocalDateTime finishTime) {
17        this.finishTime = finishTime;
18    }
19
20    public LocalDateTime 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.LocalDateTime;
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.LocalDateTime;
6 import java.util.Comparator;
7
8 public class StageResult implements Serializable {
9     private final LocalDateTime[] 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 }
```