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 !!!!
//    - Code Formatting
//    - Documentation/Comments
//    - Testing
//    - each function public/private/protected/default
//    - Optimise results?

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<>();

  // // Record that will hold all the CyclingPortals teams, riders, races, stages & segments as
  ↪  well
  // as
  // // all the Id counts for each object.
  // private record SavedCyclingPortal(
  //     ArrayList<Team> teams,
  //     ArrayList<Rider> riders,
  //     ArrayList<Race> races,
  //     ArrayList<Stage> stages,
  //     ArrayList<Segment> segments,
  //     int teamIdCount,
  //     int riderIdCount,
```

```
38    //      int raceIdCount,
39    //      int stageIdCount,
40    //      int segmentIdCount) {}
41
42    /**
43     * Determine if a string contains any illegal whitespace characters.
44     *
45     * @param string The input string to be tested for whitespace.
46     * @return A boolean, true if the input string contains whitespace, false if not.
47     */
48    public static boolean containsWhitespace(String string) {
49      for (int i = 0; i < string.length(); ++i) {
50        if (Character.isWhitespace(string.charAt(i))) {
51          return true;
52        }
53      }
54      return false;
55    }
56
57    /**
58     * Get a Team object by a Team ID.
59     *
60     * @param ID The int ID of the Team to be looked up.
61     * @return The Team object of the team, if one is found.
62     * @throws IDNotRecognisedException Thrown if no team is found with the given Team ID.
63     */
64    public Team getTeamById(int ID) throws IDNotRecognisedException {
65      for (Team team : teams) {
66        if (team.getId() == ID) {
67          return team;
68        }
69      }
70      throw new IDNotRecognisedException("Team ID not found.");
71    }
72
73    /**
74     * Get a Rider object by a Rider ID.
75     *
76     * @param ID The int ID of the Rider to be looked up.
77     * @return The Rider object of the Rider, if one is found.
78     * @throws IDNotRecognisedException Thrown if no rider is found with the given Rider ID.
79     */
80    public Rider getRiderById(int ID) throws IDNotRecognisedException {
81      for (Rider rider : riders) {
82        if (rider.getId() == ID) {
83          return rider;
84        }
85      }
86      throw new IDNotRecognisedException("Rider ID not found.");
87    }
88
89    /**
90     * Get a Race object by a Race ID.
91     *
92     * @param ID The int ID of the Race to be looked up.
93     * @return The Race object of the race, if one is found.
94     * @throws IDNotRecognisedException Thrown if no race is found with the given Race ID.
95     */
```

```java
  public Race getRaceById(int ID) throws IDNotRecognisedException {
    for (Race race : races) {
      if (race.getId() == ID) {
        return race;
      }
    }
    throw new IDNotRecognisedException("Race ID not found.");
  }

  /**
   * Get a Stage object by a Stage ID.
   *
   * @param ID The int ID of the Stage to be looked up.
   * @return The Stage object of the stage, if one is found.
   * @throws IDNotRecognisedException Thrown if no stage is found with the given Stage ID.
   */
  public Stage getStageById(int ID) throws IDNotRecognisedException {
    for (Stage stage : stages) {
      if (stage.getId() == ID) {
        return stage;
      }
    }
    throw new IDNotRecognisedException("Stage ID not found.");
  }

  /**
   * Get a Segment object by a Segment ID.
   *
   * @param ID The int ID of the Segment to be looked up.
   * @return The Segment object of the segment, if one is found.
   * @throws IDNotRecognisedException Thrown if no segment is found with the given Segment ID.
   */
  public Segment getSegmentById(int ID) throws IDNotRecognisedException {
    for (Segment segment : segments) {
      if (segment.getId() == ID) {
        return segment;
      }
    }
    throw new IDNotRecognisedException("Segment ID not found.");
  }

  /**
   * Loops over all races, stages and segments to remove all of a given riders results.
   *
   * @param rider The Rider object whose results will be removed from the Cycling Portal.
   */
  public void removeRiderResults(Rider rider) {
    for (Race race : races) {
      race.removeRiderResults(rider);
    }
    for (Stage stage : stages) {
      stage.removeRiderResults(rider);
    }
    for (Segment segment : segments) {
      segment.removeRiderResults(rider);
    }
  }

```

```java
  @Override
  public int[] getRaceIds() {
    int[] raceIDs = new int[races.size()];
    for (int i = 0; i < races.size(); i++) {
      Race race = races.get(i);
      raceIDs[i] = race.getId();
    }
    return raceIDs;
  }

  @Override
  public int createRace(String name, String description)
      throws IllegalNameException, InvalidNameException {
    // Check a race with this name does not already exist in the system.
    for (Race race : races) {
      if (race.getName().equals(name)) {
        throw new IllegalNameException("A Race with the name " + name + " already exists.");
      }
    }
    Race race = new Race(name, description);
    races.add(race);
    return race.getId();
  }

  @Override
  public String viewRaceDetails(int raceId) throws IDNotRecognisedException {
    Race race = getRaceById(raceId);
    return race.getDetails();
  }

  @Override
  public void removeRaceById(int raceId) throws IDNotRecognisedException {
    Race race = getRaceById(raceId);
    // Remove all the races stages from the CyclingPortal.
    for (final Stage stage : race.getStages()) {
      stages.remove(stage);
    }
    races.remove(race);
  }

  @Override
  public int getNumberOfStages(int raceId) throws IDNotRecognisedException {
    Race race = getRaceById(raceId);
    return race.getStages().size();
  }

  @Override
  public int addStageToRace(
      int raceId,
      String stageName,
      String description,
      double length,
      LocalDateTime startTime,
      StageType type)
      throws IDNotRecognisedException, IllegalNameException, InvalidNameException,
          InvalidLengthException {
    Race race = getRaceById(raceId);
    // Check a stage with this name does not already exist in the system.
```

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

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

```
328       teams.remove(team);
329     }
330
331     @Override
332     public int[] getTeams() {
333       int[] teamIDs = new int[teams.size()];
334       for (int i = 0; i < teams.size(); i++) {
335         Team team = teams.get(i);
336         teamIDs[i] = team.getId();
337       }
338       return teamIDs;
339     }
340
341     @Override
342     public int[] getTeamRiders(int teamId) throws IDNotRecognisedException {
343       Team team = getTeamById(teamId);
344       ArrayList<Rider> teamRiders = team.getRiders();
345       int[] teamRiderIds = new int[teamRiders.size()];
346       // Gathers ID's of Riders in the Team.
347       for (int i = 0; i < teamRiderIds.length; i++) {
348         teamRiderIds[i] = teamRiders.get(i).getId();
349       }
350       return teamRiderIds;
351     }
352
353     @Override
354     public int createRider(int teamID, String name, int yearOfBirth)
355         throws IDNotRecognisedException, IllegalArgumentException {
356       Team team = getTeamById(teamID);
357       Rider rider = new Rider(team, name, yearOfBirth);
358       // Adds Rider to both the Team and the list of Riders.
359       team.addRider(rider);
360       riders.add(rider);
361       return rider.getId();
362     }
363
364     @Override
365     public void removeRider(int riderId) throws IDNotRecognisedException {
366       Rider rider = getRiderById(riderId);
367       removeRiderResults(rider);
368       // Removes Rider from both the Team and the list of Riders.
369       rider.getTeam().removeRider(rider);
370       riders.remove(rider);
371     }
372
373     @Override
374     public void registerRiderResultsInStage(int stageId, int riderId, LocalTime... checkpoints)
375         throws IDNotRecognisedException, DuplicatedResultException, InvalidCheckpointsException,
376             InvalidStageStateException {
377       Stage stage = getStageById(stageId);
378       Rider rider = getRiderById(riderId);
379       stage.registerResult(rider, checkpoints);
380     }
381
382     @Override
383     public LocalTime[] getRiderResultsInStage(int stageId, int riderId)
384         throws IDNotRecognisedException {
385       Stage stage = getStageById(stageId);
```

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

```
444   }
445
446   @Override
447   public LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId)
448       throws IDNotRecognisedException {
449     Stage stage = getStageById(stageId);
450     // Gets a list of Riders from the Stage ordered by their Elapsed Times.
451     List<Rider> riders = stage.getRidersByElapsedTime();
452     LocalTime[] riderAETs = new LocalTime[riders.size()];
453     // Gathers Riders' Adjusted Elapsed Times ordered by their Elapsed Times.
454     for (int i = 0; i < riders.size(); i++) {
455       Rider rider = riders.get(i);
456       riderAETs[i] = stage.getRiderResult(rider).getAdjustedElapsedLocalTime();
457     }
458     return riderAETs;
459   }
460
461   @Override
462   public int[] getRidersPointsInStage(int stageId) throws IDNotRecognisedException {
463     Stage stage = getStageById(stageId);
464     // Gets a list of Riders from the Stage ordered by their Elapsed Times.
465     List<Rider> riders = stage.getRidersByElapsedTime();
466     int[] riderSprinterPoints = new int[riders.size()];
467     // Gathers Sprinters' Points ordered by their Elapsed Times.
468     for (int i = 0; i < riders.size(); i++) {
469       Rider rider = riders.get(i);
470       riderSprinterPoints[i] = stage.getRiderResult(rider).getSprintersPoints();
471     }
472     return riderSprinterPoints;
473   }
474
475   @Override
476   public int[] getRidersMountainPointsInStage(int stageId) throws IDNotRecognisedException {
477     Stage stage = getStageById(stageId);
478     // Gets a list of Riders from the Stage ordered by their Elapsed Times.
479     List<Rider> riders = stage.getRidersByElapsedTime();
480     int[] riderMountainPoints = new int[riders.size()];
481     // Gathers Riders' Mountain Points ordered by their Elapsed Times.
482     for (int i = 0; i < riders.size(); i++) {
483       Rider rider = riders.get(i);
484       riderMountainPoints[i] = stage.getRiderResult(rider).getMountainPoints();
485     }
486     return riderMountainPoints;
487   }
488
489   @Override
490   public void eraseCyclingPortal() {
491     // Replaces teams, riders, races, stages and segments with empty ArrayLists.
492     teams = new ArrayList<>();
493     riders = new ArrayList<>();
494     races = new ArrayList<>();
495     stages = new ArrayList<>();
496     segments = new ArrayList<>();
497     // Sets the ID counters of the Rider, Team, Race, Stage and Segment objects back
498     // to 0.
499     Rider.resetIdCounter();
500     Team.resetIdCounter();
501     Race.resetIdCounter();
```

```
502     Stage.resetIdCounter();
503     Segment.resetIdCounter();
504   }
505
506   @Override
507   public void saveCyclingPortal(String filename) throws IOException {
508     FileOutputStream file = new FileOutputStream(filename + ".ser");
509     ObjectOutputStream output = new ObjectOutputStream(file);
510     // Saves teams, riders, races, stages and segments ArrayLists.
511     // Saves ID counters of Team, Rider, Race, Stage and Segment objects.
512     SavedCyclingPortal savedCyclingPortal =
513         new SavedCyclingPortal(
514             teams,
515             riders,
516             races,
517             stages,
518             segments,
519             Team.getIdCounter(),
520             Rider.getIdCounter(),
521             Race.getIdCounter(),
522             Stage.getIdCounter(),
523             Segment.getIdCounter());
524     output.writeObject(savedCyclingPortal);
525     output.close();
526     file.close();
527   }
528
529   @Override
530   public void loadCyclingPortal(String filename) throws IOException, ClassNotFoundException {
531     eraseCyclingPortal();
532     FileInputStream file = new FileInputStream(filename + ".ser");
533     ObjectInputStream input = new ObjectInputStream(file);
534
535     SavedCyclingPortal savedCyclingPortal = (SavedCyclingPortal) input.readObject();
536     // Imports teams, riders, races, stages and segments ArrayLists from the last save.
537     teams = savedCyclingPortal.teams;
538     riders = savedCyclingPortal.riders;
539     races = savedCyclingPortal.races;
540     stages = savedCyclingPortal.stages;
541     segments = savedCyclingPortal.segments;
542
543     // Imports ID counters of Team, Rider, Race, Stage and Segment objects from the last save.
544     Team.setIdCounter(savedCyclingPortal.teamIdCount);
545     Rider.setIdCounter(savedCyclingPortal.riderIdCount);
546     Race.setIdCounter(savedCyclingPortal.raceIdCount);
547     Stage.setIdCounter(savedCyclingPortal.stageIdCount);
548     Segment.setIdCounter(savedCyclingPortal.segmentIdCount);
549
550     input.close();
551     file.close();
552   }
553
554   @Override
555   public void removeRaceByName(String name) throws NameNotRecognisedException {
556     for (final Race race : races) {
557       if (race.getName().equals(name)) {
558         races.remove(race);
559         return;
```

```
560         }
561       }
562       throw new NameNotRecognisedException("Race name is not in the system.");
563     }
564
565     @Override
566     public int[] getRidersGeneralClassificationRank(int raceId) throws IDNotRecognisedException {
567       Race race = getRaceById(raceId);
568       List<Rider> riders = race.getRidersByAdjustedElapsedTime();
569       int[] riderIds = new int[riders.size()];
570       // Gathers Rider ID's ordered by their Adjusted Elapsed Times.
571       for (int i = 0; i < riders.size(); i++) {
572         riderIds[i] = riders.get(i).getId();
573       }
574       return riderIds;
575     }
576
577     @Override
578     public LocalTime[] getGeneralClassificationTimesInRace(int raceId)
579         throws IDNotRecognisedException {
580       Race race = getRaceById(raceId);
581       // Gets a list of Riders from the Stage ordered by their Adjusted Elapsed Times.
582       List<Rider> riders = race.getRidersByAdjustedElapsedTime();
583       LocalTime[] riderTimes = new LocalTime[riders.size()];
584       // Gathers Riders' Cumulative Adjusted Elapsed LocalTimes ordered by their Adjusted Elapsed
585       // Times.
586       for (int i = 0; i < riders.size(); i++) {
587         riderTimes[i] = race.getRiderResults(riders.get(i)).getCumulativeAdjustedElapsedLocalTime();
588       }
589       return riderTimes;
590     }
591
592     @Override
593     public int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException {
594       Race race = getRaceById(raceId);
595       List<Rider> riders = race.getRidersByAdjustedElapsedTime();
596       int[] riderIds = new int[riders.size()];
597       // Gathers Riders' Cumulative Sprinters Points ordered by their Adjusted Elapsed Times.
598       for (int i = 0; i < riders.size(); i++) {
599         riderIds[i] = race.getRiderResults(riders.get(i)).getCumulativeSprintersPoints();
600       }
601       return riderIds;
602     }
603
604     @Override
605     public int[] getRidersMountainPointsInRace(int raceId) throws IDNotRecognisedException {
606       Race race = getRaceById(raceId);
607       List<Rider> riders = race.getRidersByAdjustedElapsedTime();
608       int[] riderIds = new int[riders.size()];
609       // Gathers Riders' Cumulative Mountain Points ordered by their Adjusted Elapsed Times.
610       for (int i = 0; i < riders.size(); i++) {
611         riderIds[i] = race.getRiderResults(riders.get(i)).getCumulativeMountainPoints();
612       }
613       return riderIds;
614     }
615
616     @Override
617     public int[] getRidersPointClassificationRank(int raceId) throws IDNotRecognisedException {
```

```
618     Race race = getRaceById(raceId);
619     List<Rider> riders = race.getRidersBySprintersPoints();
620     int[] riderIds = new int[riders.size()];
621     // Gathers Rider ID's ordered by their Sprinters Points.
622     for (int i = 0; i < riders.size(); i++) {
623       riderIds[i] = riders.get(i).getId();
624     }
625     return riderIds;
626   }
627
628   @Override
629   public int[] getRidersMountainPointClassificationRank(int raceId)
630       throws IDNotRecognisedException {
631     Race race = getRaceById(raceId);
632     List<Rider> riders = race.getRidersByMountainPoints();
633     int[] riderIds = new int[riders.size()];
634     // Gathers Rider ID's ordered by their Mountain Points.
635     for (int i = 0; i < riders.size(); i++) {
636       riderIds[i] = riders.get(i).getId();
637     }
638     return riderIds;
639   }
640 }
```

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 and Riders, 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 final 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 is 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 an inputted value.
63     *
64     * @param newCount: new value of the static ID counter.
65     */
66    static void setIdCounter(int newCount) {
67      count = newCount;
68    }
69
70    /**
71     * Method to get the ID of the Race object.
72     *
73     * @return int id: the Race's unique ID value.
74     */
```

```java
  public int getId() {
    return id;
  }

  /**
   * Method to get the name of the Race.
   *
   * @return String name: the given name of the Race.
   */
  public String getName() {
    return name;
  }

  /**
   * Method that adds a Stage to the Race object's ordered list of Stages. It is added to the
   * correct position based on its start time.
   *
   * @param stage: The stage to be added to the Race.
   */
  public void addStage(Stage stage) {
    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() {
    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 details.
   */
```

```
133   public String getDetails() {
134     double currentLength = 0;
135     for (final Stage stage : stages) {
136       currentLength = currentLength + stage.getLength();
137     }
138     return ("Race ID: "
139         + id
140         + System.lineSeparator()
141         + "Name: "
142         + name
143         + System.lineSeparator()
144         + "Description: "
145         + description
146         + System.lineSeparator()
147         + "Number of Stages: "
148         + stages.size()
149         + System.lineSeparator()
150         + "Total length: "
151         + currentLength);
152   }
153
154   /**
155    * Method to get a list of Riders in the Race, sorted by their Adjusted Elapsed Time.
156    *
157    * @return List<Rider>: correctly sorted Riders.
158    */
159   public List<Rider> getRidersByAdjustedElapsedTime() {
160     calculateResults();
161     return sortRiderResultsBy(RaceResult.sortByAdjustedElapsedTime);
162   }
163
164   /**
165    * Method to get a list of Riders in the Race, sorted by their Sprinters Points.
166    *
167    * @return List<Rider>: correctly sorted Riders.
168    */
169   public List<Rider> getRidersBySprintersPoints() {
170     calculateResults();
171     return sortRiderResultsBy(RaceResult.sortBySprintersPoints);
172   }
173
174   /**
175    * Method to get a list of Riders in the Race, sorted by their Mountain Points.
176    *
177    * @return List<Rider>: correctly sorted Riders.
178    */
179   public List<Rider> getRidersByMountainPoints() {
180     calculateResults();
181     return sortRiderResultsBy(RaceResult.sortByMountainPoints);
182   }
183
184   /**
185    * Method to get the results of a given Rider.
186    *
187    * @param rider: Rider to get the results of.
188    * @return RaceResult: Result of the Rider.
189    */
190   public RaceResult getRiderResults(Rider rider) {
```

```
191       calculateResults();
192       return results.get(rider);
193     }
194
195     /**
196      * Method to remove the Results of a given Rider.
197      *
198      * @param rider: Rider whose Results will be removed.
199      */
200     public void removeRiderResults(Rider rider) {
201       results.remove(rider);
202     }
203
204     /**
205      * Method to get a list of Riders sorted by a given comparator of their Results.
206      *
207      * @param comparison: a comparator on the Riders' Results to sort the Riders by.
208      * @return List<Rider>: List of Riders sorted by the comparator on the Results.
209      */
210     private List<Rider> sortRiderResultsBy(Comparator<RaceResult> comparison) {
211       return results.entrySet().stream()
212           .sorted(Map.Entry.comparingByValue(comparison))
213           .map(Map.Entry::getKey)
214           .collect(Collectors.toList());
215     }
216
217     /**
218      * Method to register the Rider's Result to the Stage.
219      *
220      * @param rider: Rider whose Result needs to be registered.
221      * @param stageResult: Stage that the Result will be added to.
222      */
223     private void registerRiderResults(Rider rider, StageResult stageResult) {
224       if (results.containsKey(rider)) {
225         // When the hashmap of Results already contains the Results for the given Rider,
226         // results are not re-added.
227         results.get(rider).addStageResult(stageResult);
228       } else {
229         // If the hashmap of Results does not contain the Results for the given Rider,
230         // they then are added now.
231         RaceResult raceResult = new RaceResult();
232         raceResult.addStageResult(stageResult);
233         results.put(rider, raceResult);
234       }
235     }
236
237     /** Method that calculates the results for each Rider. */
238     private void calculateResults() {
239       for (Stage stage : stages) {
240         HashMap<Rider, StageResult> stageResults = stage.getStageResults();
241         for (Rider rider : stageResults.keySet()) {
242           registerRiderResults(rider, stageResults.get(rider));
243         }
244       }
245     }
246 }
```

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;

  // TODO: Test ordered Asc
  protected static final Comparator<RaceResult> sortByAdjustedElapsedTime =
      Comparator.comparing(RaceResult::getCumulativeAdjustedElapsedTime);

  // TODO: Test order Desc
  protected static final Comparator<RaceResult> sortBySprintersPoints =
      Comparator.comparing(RaceResult::getCumulativeSprintersPoints).reversed();
  // protected static final Comparator<RaceResult> sortBySprintersPoints = (RaceResult result1,
  //    RaceResult result2) -> Integer.compare(result2.getCumulativeSprintersPoints(),
  //        result1.getCumulativeSprintersPoints());
  protected static final Comparator<RaceResult> sortByMountainPoints =
      Comparator.comparing(RaceResult::getCumulativeMountainPoints).reversed();
  // 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;

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};
```

```
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 SegmentType getType() {
56      return type;
57    }
58
59    public int getId() {
60      return id;
61    }
62
63    public Stage getStage() {
64      return stage;
65    }
66
67    public double getLocation() {
68      return location;
69    }
70
71    public void registerResults(Rider rider, LocalTime finishTime) {
72      SegmentResult result = new SegmentResult(finishTime);
73      results.put(rider, result);
74    }
75
76    public SegmentResult getRiderResult(Rider rider) {
77      calculateResults();
```

```
78      return results.get(rider);
79    }
80
81    public void removeRiderResults(Rider rider) {
82      results.remove(rider);
83    }
84
85    private List<Rider> sortRiderResults() {
86      return results.entrySet().stream()
87          .sorted(Map.Entry.comparingByValue(SegmentResult.sortByFinishTime))
88          .map(Map.Entry::getKey)
89          .collect(Collectors.toList());
90    }
91
92    private void calculateResults() {
93      List<Rider> riders = sortRiderResults();
94
95      for (int i = 0; i < results.size(); i++) {
96        Rider rider = riders.get(i);
97        SegmentResult result = results.get(rider);
98        int position = i + 1;
99        // Position Calculation
100       result.setPosition(position);
101
102       // Points Calculation
103       int[] pointsDistribution = getPointsDistribution();
104       if (position <= pointsDistribution.length) {
105         int points = pointsDistribution[i];
106         if (this.type.equals(SegmentType.SPRINT)) {
107           result.setSprintersPoints(points);
108           result.setMountainPoints(0);
109         } else {
110           result.setSprintersPoints(0);
111           result.setMountainPoints(points);
112         }
113       } else {
114         result.setMountainPoints(0);
115         result.setSprintersPoints(0);
116       }
117     }
118   }
119
120   private int[] getPointsDistribution() {
121     return switch (type) {
122       case HC -> HC_POINTS;
123       case C1 -> C1_POINTS;
124       case C2 -> C2_POINTS;
125       case C3 -> C3_POINTS;
126       case C4 -> C4_POINTS;
127       case SPRINT -> SPRINT_POINTS;
128     };
129   }
130 }
```

9 SegmentResult.java

```java
package cycling;

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

public class SegmentResult implements Serializable {
  private final LocalTime finishTime;
  private int position;
  private int sprintersPoints;
  private int mountainPoints;

  protected static final Comparator<SegmentResult> sortByFinishTime =
      Comparator.comparing(SegmentResult::getFinishTime);

  public SegmentResult(LocalTime finishTime) {
    this.finishTime = finishTime;
  }

  public LocalTime getFinishTime() {
    return finishTime;
  }

  public void setPosition(int position) {
    this.position = position;
  }

  public int getPosition() {
    return position;
  }

  public void setMountainPoints(int points) {
    this.mountainPoints = points;
  }

  public void setSprintersPoints(int points) {
    this.sprintersPoints = points;
  }

  public int getMountainPoints() {
    return this.mountainPoints;
  }

  public int getSprintersPoints() {
    return this.sprintersPoints;
  }
}
```

10 Stage.java

```java
package cycling;

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

```java
import java.time.LocalTime;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

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

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

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

  public Stage(
      Race race,
      String name,
      String description,
      double length,
      LocalDateTime startTime,
      StageType type)
      throws InvalidNameException, InvalidLengthException {
    if (name == null
        || name.isEmpty()
        || name.length() > 30
        || CyclingPortal.containsWhitespace(name)) {
      throw new InvalidNameException(
          "Stage name cannot be null, empty, have more than 30 characters or have white spaces.");
    }
    if (length < 5) {
      throw new InvalidLengthException("Length is invalid, cannot be less than 5km.");
    }
    this.name = name;
    this.description = description;
    this.race = race;
    this.length = length;
    this.startTime = startTime;
    this.type = type;
    this.id = Stage.count++;
  }

  static void resetIdCounter() {
    count = 0;
  }

  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

```java
package cycling;

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

public class StageResult implements Serializable {
  private final LocalTime[] checkpoints;
  private final Duration elapsedTime;
  private Duration adjustedElapsedTime;
  private int position;
  private int sprintersPoints;
  private int mountainPoints;

  protected static final Comparator<StageResult> sortByElapsedTime =
      Comparator.comparing(StageResult::getElapsedTime);

  public StageResult(LocalTime[] checkpoints) {
    this.checkpoints = checkpoints;
    this.elapsedTime = Duration.between(checkpoints[0], checkpoints[checkpoints.length - 1]);
  }

  public LocalTime[] getCheckpoints() {
    return this.checkpoints;
  }

  public Duration getElapsedTime() {
    return elapsedTime;
  }

  public void setPosition(int position) {
    this.position = position;
  }

  public void setAdjustedElapsedTime(Duration adjustedElapsedTime) {
    this.adjustedElapsedTime = adjustedElapsedTime;
  }

  public int getPosition() {
    return position;
  }

  public Duration getAdjustedElapsedTime() {
    return adjustedElapsedTime;
  }

  public LocalTime getAdjustedElapsedLocalTime() {
    return checkpoints[0].plus(adjustedElapsedTime);
  }

  public void setMountainPoints(int points) {
    this.mountainPoints = points;
  }

  public void setSprintersPoints(int points) {
```

```
57      this.sprintersPoints = points;
58    }
59
60    public int getMountainPoints() {
61      return mountainPoints;
62    }
63
64    public int getSprintersPoints() {
65      return sprintersPoints;
66    }
67
68    // --Commented out by Inspection START (28/03/2022, 3:31 pm):
69    //  public void add(StageResult res){
70    //    this.elapsedTime = this.elapsedTime.plus(res.getElapsedTime());
71    //    this.adjustedElapsedTime = this.adjustedElapsedTime.plus(res.getAdjustedElapsedTime());
72    //    this.sprintersPoints += res.getSprintersPoints();
73    //    this.mountainPoints += res.getMountainPoints();
74    //  }
75    // --Commented out by Inspection STOP (28/03/2022, 3:31 pm)
76  }
```

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  }
```