CyclingPortal Printout

123456789 & 987654321

Contents

1 CategorizedClimb.java

```java
package cycling;

/**
 * Categorised Climb class. This represents a type of Segment that has a stage, location, type,
 * average gradient and a length.
 */
public class CategorizedClimb extends Segment {
  private final Double averageGradient;
  private final Double length;

  /**
   * Constructor method that sets up the Categorised Climb with a stage, location, type, average
   * gradient and length.
   *
   * @param stage that the Categorised Climb is in.
   * @param location of the Categorised Climb.
   * @param type of Categorised Climb.
   * @param averageGradient of the Categorised Climb.
   * @param length of the Categorised Climb.
   * @throws InvalidLocationException Thrown if the location is out of bounds of the Stage length.
   * @throws InvalidStageStateException Thrown if the Stage is waiting for results.
   * @throws InvalidStageTypeException Thrown if the type is a time trial.
   */
  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;

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

  /**
   * Determine if a string contains any illegal whitespace characters.
   *
```

```
22      * @param string The input string to be tested for whitespace.
23      * @return A boolean, true if the input string contains whitespace, false if not.
24      */
25     public static boolean containsWhitespace(String string) {
26       for (int i = 0; i < string.length(); ++i) {
27         if (Character.isWhitespace(string.charAt(i))) {
28           return true;
29         }
30       }
31       return false;
32     }
33
34     /**
35      * Get a Team object by a Team ID.
36      *
37      * @param ID The int ID of the Team to be looked up.
38      * @return The Team object of the team, if one is found.
39      * @throws IDNotRecognisedException Thrown if no team is found with the given Team ID.
40      */
41     public Team getTeamById(int ID) throws IDNotRecognisedException {
42       for (Team team : teams) {
43         if (team.getId() == ID) {
44           return team;
45         }
46       }
47       throw new IDNotRecognisedException("Team ID not found.");
48     }
49
50     /**
51      * Get a Rider object by a Rider ID.
52      *
53      * @param ID The int ID of the Rider to be looked up.
54      * @return The Rider object of the Rider, if one is found.
55      * @throws IDNotRecognisedException Thrown if no rider is found with the given Rider ID.
56      */
57     public Rider getRiderById(int ID) throws IDNotRecognisedException {
58       for (Rider rider : riders) {
59         if (rider.getId() == ID) {
60           return rider;
61         }
62       }
63       throw new IDNotRecognisedException("Rider ID not found.");
64     }
65
66     /**
67      * Get a Race object by a Race ID.
68      *
69      * @param ID The int ID of the Race to be looked up.
70      * @return The Race object of the race, if one is found.
71      * @throws IDNotRecognisedException Thrown if no race is found with the given Race ID.
72      */
73     public Race getRaceById(int ID) throws IDNotRecognisedException {
74       for (Race race : races) {
75         if (race.getId() == ID) {
76           return race;
77         }
78       }
79       throw new IDNotRecognisedException("Race ID not found.");
```

```
80    }
81
82    /**
83     * Get a Stage object by a Stage ID.
84     *
85     * @param ID The int ID of the Stage to be looked up.
86     * @return The Stage object of the stage, if one is found.
87     * @throws IDNotRecognisedException Thrown if no stage is found with the given Stage ID.
88     */
89    public Stage getStageById(int ID) throws IDNotRecognisedException {
90      for (Stage stage : stages) {
91        if (stage.getId() == ID) {
92          return stage;
93        }
94      }
95      throw new IDNotRecognisedException("Stage ID not found.");
96    }
97
98    /**
99     * Get a Segment object by a Segment ID.
100    *
101    * @param ID The int ID of the Segment to be looked up.
102    * @return The Segment object of the segment, if one is found.
103    * @throws IDNotRecognisedException Thrown if no segment is found with the given Segment ID.
104    */
105   public Segment getSegmentById(int ID) throws IDNotRecognisedException {
106     for (Segment segment : segments) {
107       if (segment.getId() == ID) {
108         return segment;
109       }
110     }
111     throw new IDNotRecognisedException("Segment ID not found.");
112   }
113
114   /**
115    * Loops over all races, stages and segments to remove all of a given riders results.
116    *
117    * @param rider The Rider object whose results will be removed from the Cycling Portal.
118    */
119   public void removeRiderResults(Rider rider) {
120     for (Race race : races) {
121       race.removeRiderResults(rider);
122     }
123     for (Stage stage : stages) {
124       stage.removeRiderResults(rider);
125     }
126     for (Segment segment : segments) {
127       segment.removeRiderResults(rider);
128     }
129   }
130
131   @Override
132   public int[] getRaceIds() {
133     int[] raceIDs = new int[races.size()];
134     for (int i = 0; i < races.size(); i++) {
135       Race race = races.get(i);
136       raceIDs[i] = race.getId();
137     }
```

```
138     return raceIDs;
139   }
140
141   @Override
142   public int createRace(String name, String description)
143       throws IllegalNameException, InvalidNameException {
144     // Check a race with this name does not already exist in the system.
145     for (Race race : races) {
146       if (race.getName().equals(name)) {
147         throw new IllegalNameException("A Race with the name " + name + " already exists.");
148       }
149     }
150     Race race = new Race(name, description);
151     races.add(race);
152     return race.getId();
153   }
154
155   @Override
156   public String viewRaceDetails(int raceId) throws IDNotRecognisedException {
157     Race race = getRaceById(raceId);
158     return race.getDetails();
159   }
160
161   @Override
162   public void removeRaceById(int raceId) throws IDNotRecognisedException {
163     Race race = getRaceById(raceId);
164     // Remove all the races stages from the CyclingPortal.
165     for (final Stage stage : race.getStages()) {
166       stages.remove(stage);
167     }
168     races.remove(race);
169   }
170
171   @Override
172   public int getNumberOfStages(int raceId) throws IDNotRecognisedException {
173     Race race = getRaceById(raceId);
174     return race.getStages().size();
175   }
176
177   @Override
178   public int addStageToRace(
179       int raceId,
180       String stageName,
181       String description,
182       double length,
183       LocalDateTime startTime,
184       StageType type)
185       throws IDNotRecognisedException, IllegalNameException, InvalidNameException,
186           InvalidLengthException {
187     Race race = getRaceById(raceId);
188     // Check a stage with this name does not already exist in the system.
189     for (final Stage stage : stages) {
190       if (stage.getName().equals(stageName)) {
191         throw new IllegalNameException("A stage with the name " + stageName + " already exists.");
192       }
193     }
194     Stage stage = new Stage(race, stageName, description, length, startTime, type);
195     stages.add(stage);
```

```
196       race.addStage(stage);
197       return stage.getId();
198     }
199
200     @Override
201     public int[] getRaceStages(int raceId) throws IDNotRecognisedException {
202       Race race = getRaceById(raceId);
203       ArrayList<Stage> raceStages = race.getStages();
204       int[] raceStagesId = new int[raceStages.size()];
205       // Gathers the Stage ID's of the Stages in the Race.
206       for (int i = 0; i < raceStages.size(); i++) {
207         Stage stage = race.getStages().get(i);
208         raceStagesId[i] = stage.getId();
209       }
210       return raceStagesId;
211     }
212
213     @Override
214     public double getStageLength(int stageId) throws IDNotRecognisedException {
215       Stage stage = getStageById(stageId);
216       return stage.getLength();
217     }
218
219     @Override
220     public void removeStageById(int stageId) throws IDNotRecognisedException {
221       Stage stage = getStageById(stageId);
222       Race race = stage.getRace();
223       // Removes stage from both the Races and Stages.
224       race.removeStage(stage);
225       stages.remove(stage);
226     }
227
228     @Override
229     public int addCategorizedClimbToStage(
230         int stageId, Double location, SegmentType type, Double averageGradient, Double length)
231         throws IDNotRecognisedException, InvalidLocationException, InvalidStageStateException,
232             InvalidStageTypeException {
233       Stage stage = getStageById(stageId);
234       CategorizedClimb climb = new CategorizedClimb(stage, location, type, averageGradient, length);
235       // Adds Categorized Climb to both the list of Segments and the Stage.
236       segments.add(climb);
237       stage.addSegment(climb);
238       return climb.getId();
239     }
240
241     @Override
242     public int addIntermediateSprintToStage(int stageId, double location)
243         throws IDNotRecognisedException, InvalidLocationException, InvalidStageStateException,
244             InvalidStageTypeException {
245       Stage stage = getStageById(stageId);
246       IntermediateSprint sprint = new IntermediateSprint(stage, location);
247       // Adds Intermediate Sprint to both the list of Segments and the Stage.
248       segments.add(sprint);
249       stage.addSegment(sprint);
250       return sprint.getId();
251     }
252
253     @Override
```

```
254   public void removeSegment(int segmentId)
255       throws IDNotRecognisedException, InvalidStageStateException {
256     Segment segment = getSegmentById(segmentId);
257     Stage stage = segment.getStage();
258     // Removes Segment from both the Stage and list of Segments.
259     stage.removeSegment(segment);
260     segments.remove(segment);
261   }
262
263   @Override
264   public void concludeStagePreparation(int stageId)
265       throws IDNotRecognisedException, InvalidStageStateException {
266     Stage stage = getStageById(stageId);
267     stage.concludePreparation();
268   }
269
270   @Override
271   public int[] getStageSegments(int stageId) throws IDNotRecognisedException {
272     Stage stage = getStageById(stageId);
273     ArrayList<Segment> stageSegments = stage.getSegments();
274     int[] stageSegmentsId = new int[stageSegments.size()];
275     // Gathers Segment ID's from the Segments in the Stage.
276     for (int i = 0; i < stageSegments.size(); i++) {
277       Segment segment = stageSegments.get(i);
278       stageSegmentsId[i] = segment.getId();
279     }
280     return stageSegmentsId;
281   }
282
283   @Override
284   public int createTeam(String name, String description)
285       throws IllegalNameException, InvalidNameException {
286     // Checks if the Team name already exists on the system.
287     for (final Team team : teams) {
288       if (team.getName().equals(name)) {
289         throw new IllegalNameException("A Team with the name " + name + " already exists.");
290       }
291     }
292     Team team = new Team(name, description);
293     teams.add(team);
294     return team.getId();
295   }
296
297   @Override
298   public void removeTeam(int teamId) throws IDNotRecognisedException {
299     Team team = getTeamById(teamId);
300     // Loops through and removes Team Riders and Team Rider Results.
301     for (final Rider rider : team.getRiders()) {
302       removeRiderResults(rider);
303       riders.remove(rider);
304     }
305     teams.remove(team);
306   }
307
308   @Override
309   public int[] getTeams() {
310     int[] teamIDs = new int[teams.size()];
311     for (int i = 0; i < teams.size(); i++) {
```

```
312       Team team = teams.get(i);
313       teamIDs[i] = team.getId();
314     }
315     return teamIDs;
316   }
317
318   @Override
319   public int[] getTeamRiders(int teamId) throws IDNotRecognisedException {
320     Team team = getTeamById(teamId);
321     ArrayList<Rider> teamRiders = team.getRiders();
322     int[] teamRiderIds = new int[teamRiders.size()];
323     // Gathers ID's of Riders in the Team.
324     for (int i = 0; i < teamRiderIds.length; i++) {
325       // Assert the rider is actually on the team.
326       assert teamRiders.get(i).getTeam().equals(team);
327       // Return the rider id.
328       teamRiderIds[i] = teamRiders.get(i).getId();
329     }
330     return teamRiderIds;
331   }
332
333   @Override
334   public int createRider(int teamID, String name, int yearOfBirth)
335       throws IDNotRecognisedException, IllegalArgumentException {
336     Team team = getTeamById(teamID);
337     Rider rider = new Rider(team, name, yearOfBirth);
338     // Adds Rider to both the Team and the list of Riders.
339     team.addRider(rider);
340     riders.add(rider);
341
342     // Assert at least one rider has been added
343     assert riders.size() > 0;
344
345     return rider.getId();
346   }
347
348   @Override
349   public void removeRider(int riderId) throws IDNotRecognisedException {
350     Rider rider = getRiderById(riderId);
351     removeRiderResults(rider);
352     // Removes Rider from both the Team and the list of Riders.
353     rider.getTeam().removeRider(rider);
354     riders.remove(rider);
355   }
356
357   @Override
358   public void registerRiderResultsInStage(int stageId, int riderId, LocalTime... checkpoints)
359       throws IDNotRecognisedException, DuplicatedResultException, InvalidCheckpointsException,
360           InvalidStageStateException {
361     Stage stage = getStageById(stageId);
362     Rider rider = getRiderById(riderId);
363     stage.registerResult(rider, checkpoints);
364   }
365
366   @Override
367   public LocalTime[] getRiderResultsInStage(int stageId, int riderId)
368       throws IDNotRecognisedException {
369     Stage stage = getStageById(stageId);
```

```
370     Rider rider = getRiderById(riderId);
371     StageResult result = stage.getRiderResult(rider);
372 
373     if (result == null) {
374       // Returns an empty array if the Result is null.
375       return new LocalTime[] {};
376     } else {
377       LocalTime[] checkpoints = result.getCheckpoints();
378       // Rider Results will always be 1 shorter than the checkpoint length because
379       // the finish time checkpoint will be replaced with the Elapsed Time and the start time
380       // checkpoint will be ignored.
381       LocalTime[] resultsInStage = new LocalTime[checkpoints.length - 1];
382       LocalTime elapsedTime = LocalTime.MIDNIGHT.plus(result.getElapsedTime());
383       for (int i = 0; i < resultsInStage.length; i++) {
384         if (i == resultsInStage.length - 1) {
385           // Adds the Elapsed Time to the end of the array of Results.
386           resultsInStage[i] = elapsedTime;
387         } else {
388           // Adds each checkpoint to the array of Results until all have been added, skipping the
389           // Start time checkpoint.
390           resultsInStage[i] = checkpoints[i + 1];
391         }
392       }
393       return resultsInStage;
394     }
395   }
396 
397   @Override
398   public LocalTime getRiderAdjustedElapsedTimeInStage(int stageId, int riderId)
399       throws IDNotRecognisedException {
400     Stage stage = getStageById(stageId);
401     Rider rider = getRiderById(riderId);
402     StageResult result = stage.getRiderResult(rider);
403     if (result == null) {
404       return null;
405     } else {
406       return result.getAdjustedElapsedLocalTime();
407     }
408   }
409 
410   @Override
411   public void deleteRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException {
412     Stage stage = getStageById(stageId);
413     Rider rider = getRiderById(riderId);
414     stage.removeRiderResults(rider);
415   }
416 
417   @Override
418   public int[] getRidersRankInStage(int stageId) throws IDNotRecognisedException {
419     Stage stage = getStageById(stageId);
420     // Gets a list of Riders from the Stage ordered by their Elapsed Times.
421     List<Rider> riders = stage.getRidersByElapsedTime();
422     int[] riderIds = new int[riders.size()];
423     // Gathers ID's from the ordered list of Riders.
424     for (int i = 0; i < riders.size(); i++) {
425       riderIds[i] = riders.get(i).getId();
426     }
427     return riderIds;
```

```
428   }
429
430   @Override
431   public LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId)
432       throws IDNotRecognisedException {
433     Stage stage = getStageById(stageId);
434     // Gets a list of Riders from the Stage ordered by their Elapsed Times.
435     List<Rider> riders = stage.getRidersByElapsedTime();
436     LocalTime[] riderAETs = new LocalTime[riders.size()];
437     // Gathers Riders' Adjusted Elapsed Times ordered by their Elapsed Times.
438     for (int i = 0; i < riders.size(); i++) {
439       Rider rider = riders.get(i);
440       riderAETs[i] = stage.getRiderResult(rider).getAdjustedElapsedLocalTime();
441     }
442     return riderAETs;
443   }
444
445   @Override
446   public int[] getRidersPointsInStage(int stageId) throws IDNotRecognisedException {
447     Stage stage = getStageById(stageId);
448     // Gets a list of Riders from the Stage ordered by their Elapsed Times.
449     List<Rider> riders = stage.getRidersByElapsedTime();
450     int[] riderSprinterPoints = new int[riders.size()];
451     // Gathers Sprinters' Points ordered by their Elapsed Times.
452     for (int i = 0; i < riders.size(); i++) {
453       Rider rider = riders.get(i);
454       riderSprinterPoints[i] = stage.getRiderResult(rider).getSprintersPoints();
455     }
456     return riderSprinterPoints;
457   }
458
459   @Override
460   public int[] getRidersMountainPointsInStage(int stageId) throws IDNotRecognisedException {
461     Stage stage = getStageById(stageId);
462     // Gets a list of Riders from the Stage ordered by their Elapsed Times.
463     List<Rider> riders = stage.getRidersByElapsedTime();
464     int[] riderMountainPoints = new int[riders.size()];
465     // Gathers Riders' Mountain Points ordered by their Elapsed Times.
466     for (int i = 0; i < riders.size(); i++) {
467       Rider rider = riders.get(i);
468       riderMountainPoints[i] = stage.getRiderResult(rider).getMountainPoints();
469     }
470     return riderMountainPoints;
471   }
472
473   @Override
474   public void eraseCyclingPortal() {
475     // Replaces teams, riders, races, stages and segments with empty ArrayLists.
476     teams = new ArrayList<>();
477     riders = new ArrayList<>();
478     races = new ArrayList<>();
479     stages = new ArrayList<>();
480     segments = new ArrayList<>();
481     // Sets the ID counters of the Rider, Team, Race, Stage and Segment objects back
482     // to 0.
483     Rider.resetIdCounter();
484     Team.resetIdCounter();
485     Race.resetIdCounter();
```

```
486       Stage.resetIdCounter();
487       Segment.resetIdCounter();
488
489       // Assert the portal is erased.
490       assert teams.size() == 0;
491       assert races.size() == 0;
492     }
493
494     @Override
495     public void saveCyclingPortal(String filename) throws IOException {
496       FileOutputStream file = new FileOutputStream(filename + ".ser");
497       ObjectOutputStream output = new ObjectOutputStream(file);
498       // Saves teams, riders, races, stages and segments ArrayLists.
499       // Saves ID counters of Team, Rider, Race, Stage and Segment objects.
500       SavedCyclingPortal savedCyclingPortal =
501           new SavedCyclingPortal(
502               teams,
503               riders,
504               races,
505               stages,
506               segments,
507               Team.getIdCounter(),
508               Rider.getIdCounter(),
509               Race.getIdCounter(),
510               Stage.getIdCounter(),
511               Segment.getIdCounter());
512       output.writeObject(savedCyclingPortal);
513       output.close();
514       file.close();
515     }
516
517     @Override
518     public void loadCyclingPortal(String filename) throws IOException, ClassNotFoundException {
519       eraseCyclingPortal();
520       FileInputStream file = new FileInputStream(filename + ".ser");
521       ObjectInputStream input = new ObjectInputStream(file);
522
523       SavedCyclingPortal savedCyclingPortal = (SavedCyclingPortal) input.readObject();
524       // Imports teams, riders, races, stages and segments ArrayLists from the last save.
525       teams = savedCyclingPortal.teams;
526       riders = savedCyclingPortal.riders;
527       races = savedCyclingPortal.races;
528       stages = savedCyclingPortal.stages;
529       segments = savedCyclingPortal.segments;
530
531       // Imports ID counters of Team, Rider, Race, Stage and Segment objects from the last save.
532       Team.setIdCounter(savedCyclingPortal.teamIdCount);
533       Rider.setIdCounter(savedCyclingPortal.riderIdCount);
534       Race.setIdCounter(savedCyclingPortal.raceIdCount);
535       Stage.setIdCounter(savedCyclingPortal.stageIdCount);
536       Segment.setIdCounter(savedCyclingPortal.segmentIdCount);
537
538       input.close();
539       file.close();
540     }
541
542     @Override
543     public void removeRaceByName(String name) throws NameNotRecognisedException {
```

```
544       for (final Race race : races) {
545         if (race.getName().equals(name)) {
546           races.remove(race);
547           return;
548         }
549       }
550       throw new NameNotRecognisedException("Race name is not in the system.");
551     }
552
553     @Override
554     public int[] getRidersGeneralClassificationRank(int raceId) throws IDNotRecognisedException {
555       Race race = getRaceById(raceId);
556       List<Rider> riders = race.getRidersByAdjustedElapsedTime();
557       int[] riderIds = new int[riders.size()];
558       // Gathers Rider ID's ordered by their Adjusted Elapsed Times.
559       for (int i = 0; i < riders.size(); i++) {
560         riderIds[i] = riders.get(i).getId();
561       }
562       return riderIds;
563     }
564
565     @Override
566     public LocalTime[] getGeneralClassificationTimesInRace(int raceId)
567         throws IDNotRecognisedException {
568       Race race = getRaceById(raceId);
569       // Gets a list of Riders from the Stage ordered by their Adjusted Elapsed Times.
570       List<Rider> riders = race.getRidersByAdjustedElapsedTime();
571       LocalTime[] riderTimes = new LocalTime[riders.size()];
572       // Gathers Riders' Cumulative Adjusted Elapsed LocalTimes ordered by their Adjusted Elapsed
573       // Times.
574       for (int i = 0; i < riders.size(); i++) {
575         riderTimes[i] = race.getRiderResults(riders.get(i)).getCumulativeAdjustedElapsedLocalTime();
576       }
577       return riderTimes;
578     }
579
580     @Override
581     public int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException {
582       Race race = getRaceById(raceId);
583       List<Rider> riders = race.getRidersByAdjustedElapsedTime();
584       int[] riderIds = new int[riders.size()];
585       // Gathers Riders' Cumulative Sprinters Points ordered by their Adjusted Elapsed Times.
586       for (int i = 0; i < riders.size(); i++) {
587         riderIds[i] = race.getRiderResults(riders.get(i)).getCumulativeSprintersPoints();
588       }
589       return riderIds;
590     }
591
592     @Override
593     public int[] getRidersMountainPointsInRace(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 Mountain Points ordered by their Adjusted Elapsed Times.
598       for (int i = 0; i < riders.size(); i++) {
599         riderIds[i] = race.getRiderResults(riders.get(i)).getCumulativeMountainPoints();
600       }
601       return riderIds;
```

```
602   }
603
604   @Override
605   public int[] getRidersPointClassificationRank(int raceId) throws IDNotRecognisedException {
606     Race race = getRaceById(raceId);
607     List<Rider> riders = race.getRidersBySprintersPoints();
608     int[] riderIds = new int[riders.size()];
609     // Gathers Rider ID's ordered by their Sprinters Points.
610     for (int i = 0; i < riders.size(); i++) {
611       riderIds[i] = riders.get(i).getId();
612     }
613     return riderIds;
614   }
615
616   @Override
617   public int[] getRidersMountainPointClassificationRank(int raceId)
618       throws IDNotRecognisedException {
619     Race race = getRaceById(raceId);
620     List<Rider> riders = race.getRidersByMountainPoints();
621     int[] riderIds = new int[riders.size()];
622     // Gathers Rider ID's ordered by their Mountain Points.
623     for (int i = 0; i < riders.size(); i++) {
624       riderIds[i] = riders.get(i).getId();
625     }
626     return riderIds;
627   }
628 }
```

3 IntermediateSprint.java

```
1  package cycling;
2
3  /** Intermediate Sprint class. This represents a type of Segment that has a stage and a location.
   ↪  */
4  public class IntermediateSprint extends Segment {
5    private final double location;
6
7    /**
8     * Constructor method that sets the Intermediate Sprint up with a stage and a location.
9     *
10    * @param stage of the Intermediate Sprint.
11    * @param location of the Intermediate Sprint
12    * @throws InvalidLocationException Thrown if the location is out of bounds of the Stage length.
13    * @throws InvalidStageStateException Thrown if the Stage is waiting for results.
14    * @throws InvalidStageTypeException Thrown if the type is a time trial.
15    */
16   public IntermediateSprint(Stage stage, double location)
17       throws InvalidLocationException, InvalidStageTypeException, InvalidStageStateException {
18     super(stage, SegmentType.SPRINT, location);
19     this.location = location;
20   }
21 }
```

4 Race.java

```java
package cycling;

import java.io.Serializable;
import java.time.LocalDateTime;
import java.util.*;
import java.util.stream.Collectors;

/**
 * Race Class. This represents a Race that holds a Race's Stages, Riders Results, and also contains
 * methods that deal with these.
 */
public class Race implements Serializable {

  private final String name;
  private final String description;

  private final ArrayList<Stage> stages = new ArrayList<>();

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

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

  /**
   * Constructor method that sets up Rider with a name and a description.
   *
   * @param name: Cannot be empty, null, have a length greater than 30 or contain any whitespace.
   * @param description: A description of the race.
   * @throws InvalidNameException Thrown if the Race name does not meet name requirements stated
   *     above.
   */
  public Race(String name, String description) throws InvalidNameException {
    if (name == null
        || name.isEmpty()
        || name.length() > 30
        || CyclingPortal.containsWhitespace(name)) {
      throw new InvalidNameException(
          "The name cannot be null, empty, have more than 30 characters, or have white spaces.");
    }
    this.name = name;
    this.description = description;
    // ID counter represents the highest known ID at the current time to ensure there
    // are no ID collisions.
    this.id = Race.count++;
  }

  /** Method that resets the static ID counter of the Race. Used for erasing and loading. */
  static void resetIdCounter() {
    count = 0;
  }

  /**
   * Method to get the current state of the static ID counter.
   *
   * @return the highest race ID stored currently.
   */
```

```
57    static int getIdCounter() {
58      return count;
59    }
60
61    /**
62     * Method that sets the static ID counter to a given value. Used when loading to avoid ID
63     * collisions.
64     *
65     * @param newCount: new value of the static ID counter.
66     */
67    static void setIdCounter(int newCount) {
68      count = newCount;
69    }
70
71    /**
72     * Method to get the ID of the Race object.
73     *
74     * @return id: the Race's unique ID value.
75     */
76    public int getId() {
77      return id;
78    }
79
80    /**
81     * Method to get the name of the Race.
82     *
83     * @return name: the given name of the Race.
84     */
85    public String getName() {
86      return name;
87    }
88
89    /**
90     * Method that adds a Stage to the Race object's ordered list of Stages. It is added to the
91     * correct position based on its start time.
92     *
93     * @param stage: The stage to be added to the Race.
94     */
95    public void addStage(Stage stage) {
96      // Loops over stages in the race to insert the new stage in the correct place such that
97      // all the stages are sorted by their start time.
98      for (int i = 0; i < stages.size(); i++) {
99        // Retrieves the start time of each Stage in the Race's current Stages one by one.
100       // These are already ordered by their start times.
101       LocalDateTime iStartTime = stages.get(i).getStartTime();
102       // Adds the new Stage to the list of stages in the correct position based on
103       // its start time.
104       if (stage.getStartTime().isBefore(iStartTime)) {
105         stages.add(i, stage);
106         return;
107       }
108     }
109     stages.add(stage);
110   }
111
112   /**
113    * Method to get the list of Stages in the Race ordered by their start times.
114    *
```

```
115     * @return stages: The ordered list of Stages.
116     */
117    public ArrayList<Stage> getStages() {
118      // stages is already sorted, so no sorting needs to be done.
119      return stages;
120    }
121
122    /**
123     * Method that removes a given Stage from the list of Stages.
124     *
125     * @param stage: the Stage to be deleted.
126     */
127    public void removeStage(Stage stage) {
128      stages.remove(stage);
129    }
130
131    /**
132     * Method to get then details of a Race including Race ID, name, description number of stages and
133     * total length.
134     *
135     * @return Concatenated paragraph of race details.
136     */
137    public String getDetails() {
138      double currentLength = 0;
139      for (final Stage stage : stages) {
140        currentLength = currentLength + stage.getLength();
141      }
142      return ("Race ID: "
143          + id
144          + System.lineSeparator()
145          + "Name: "
146          + name
147          + System.lineSeparator()
148          + "Description: "
149          + description
150          + System.lineSeparator()
151          + "Number of Stages: "
152          + stages.size()
153          + System.lineSeparator()
154          + "Total length: "
155          + currentLength);
156    }
157
158    /**
159     * Method to get a list of Riders in the Race, sorted by their Adjusted Elapsed Time.
160     *
161     * @return The correctly sorted Riders.
162     */
163    public List<Rider> getRidersByAdjustedElapsedTime() {
164      // First generate the race result to calculate each riders Adjusted Elapsed Time.
165      calculateResults();
166      // Then return the riders sorted by their Adjusted Elapsed Time.
167      return sortRiderResultsBy(RaceResult.sortByAdjustedElapsedTime);
168    }
169
170    /**
171     * Method to get a list of Riders in the Race, sorted by their Sprinters Points.
172     *
```

```
173     * @return The correctly sorted Riders.
174     */
175    public List<Rider> getRidersBySprintersPoints() {
176      // First generate the race result to calculate each riders Sprinters Points.
177      calculateResults();
178      // Then return the riders sorted by their sprinters points.
179      return sortRiderResultsBy(RaceResult.sortBySprintersPoints);
180    }
181
182    /**
183     * Method to get a list of Riders in the Race, sorted by their Mountain Points.
184     *
185     * @return The correctly sorted Riders.
186     */
187    public List<Rider> getRidersByMountainPoints() {
188      // First generate the race result to calculate each riders Mountain Points.
189      calculateResults();
190      // Then return the riders sorted by their mountain points.
191      return sortRiderResultsBy(RaceResult.sortByMountainPoints);
192    }
193
194    /**
195     * Method to get the results of a given Rider.
196     *
197     * @param rider: Rider to get the results of.
198     * @return RaceResult: Result of the Rider.
199     */
200    public RaceResult getRiderResults(Rider rider) {
201      // First generate the race result to calculate each riders results.
202      calculateResults();
203      // Then return the riders result object.
204      return results.get(rider);
205    }
206
207    /**
208     * Method to remove the Results of a given Rider.
209     *
210     * @param rider: Rider whose Results will be removed.
211     */
212    public void removeRiderResults(Rider rider) {
213      results.remove(rider);
214    }
215
216    /**
217     * Method to get a list of Riders sorted by a given comparator of their Results. Will only return
218     * riders who have results registered in their name.
219     *
220     * @param comparator comparator on the Riders' Results to sort the Riders by.
221     * @return List of Riders (who posses recorded results) sorted by the comparator on the Results.
222     */
223    private List<Rider> sortRiderResultsBy(Comparator<RaceResult> comparator) {
224      // convert the hashmap into a set
225      return results.entrySet().stream()
226          // Sort the set by the comparator on the results.
227          .sorted(Map.Entry.comparingByValue(comparator))
228          // Get the rider element of the set and ignore the results now they have been sorted.
229          .map(Map.Entry::getKey)
230          // Convert to a list of riders.
```

```
231         .collect(Collectors.toList());
232   }
233
234   /**
235    * Method to register the Rider's Result to the Stage.
236    *
237    * @param rider: Rider whose Result needs to be registered.
238    * @param stageResult: Stage that the Result will be added to.
239    */
240   private void registerRiderResults(Rider rider, StageResult stageResult) {
241     if (results.containsKey(rider)) {
242       // If results already exist for a given rider add the current stage results
243       // to the existing total race results.
244       results.get(rider).addStageResult(stageResult);
245     } else {
246       // If no race results exists, create a new RaceResult object based on the current
247       // stage results.
248       RaceResult raceResult = new RaceResult();
249       raceResult.addStageResult(stageResult);
250       results.put(rider, raceResult);
251     }
252   }
253
254   /** Private method that calculates the results for each Rider. */
255   private void calculateResults() {
256     // Clear existing results.
257     results = new HashMap<>();
258     // We must loop over all stages and collect their results for each rider as each riders results
259     // are dependent on their position in the race, and thus the results of the other riders.
260     for (Stage stage : stages) {
261       HashMap<Rider, StageResult> stageResults = stage.getStageResults();
262       for (Rider rider : stageResults.keySet()) {
263         registerRiderResults(rider, stageResults.get(rider));
264       }
265     }
266   }
267 }
```

5 RaceResult.java

```
1  package cycling;
2
3  import java.io.Serializable;
4  import java.time.Duration;
5  import java.time.LocalTime;
6  import java.util.Comparator;
7
8  /**
9   * This represents a given riders results in a race. The riders adjusted elapsed time, sprinters
10  * points and mountain points over all stages and segments are recorded here.
11  */
12 public class RaceResult implements Serializable {
13   private Duration cumulativeAdjustedElapsedTime = Duration.ZERO;
14   private int cumulativeSprintersPoints = 0;
15   private int cumulativeMountainPoints = 0;
16
17   // A comparator which sorts RaceResults based on Adjusted Elapsed Time in ascending order. The
```

```java
  // result with the shortest time will come first.
  protected static final Comparator<RaceResult> sortByAdjustedElapsedTime =
      Comparator.comparing(RaceResult::getCumulativeAdjustedElapsedTime);

  // A comparator which sorts RaceResults based on Sprinters Points in descending order. The result
  // with the most points will come first.
  protected static final Comparator<RaceResult> sortBySprintersPoints =
      (RaceResult result1, RaceResult result2) ->
          Integer.compare(
              result2.getCumulativeSprintersPoints(), result1.getCumulativeSprintersPoints());

  // A comparator which sorts RaceResults based on Mountain Points in descending order. The result
  // with the most points will come first.
  protected static final Comparator<RaceResult> sortByMountainPoints =
      (RaceResult result1, RaceResult result2) ->
          Integer.compare(
              result2.getCumulativeMountainPoints(), result1.getCumulativeMountainPoints());

  /**
   * A method to get the recorded Adjusted Elapsed Time over all stages.
   *
   * @return The cumulative adjusted elapsed time as a duration.
   */
  public Duration getCumulativeAdjustedElapsedTime() {
    return this.cumulativeAdjustedElapsedTime;
  }

  /**
   * A method to get the recorded Adjusted Elapsed Time over all stages as a LocalTime.
   *
   * @return The cumulative adjusted elapsed time as a Local Time
   */
  public LocalTime getCumulativeAdjustedElapsedLocalTime() {
    // Calculated the AET as a Local time by adding the duration to midnight: 0:00 + Duration
    return LocalTime.MIDNIGHT.plus(this.cumulativeAdjustedElapsedTime);
  }

  /**
   * A method to get the recorded Mountain Points over all stages and segments.
   *
   * @return The cumulative mountain points.
   */
  public int getCumulativeMountainPoints() {
    return this.cumulativeMountainPoints;
  }

  /**
   * A method to get the recorded Sprinters Points over all stages and segments.
   *
   * @return The cumulative sprinters points.
   */
  public int getCumulativeSprintersPoints() {
    return this.cumulativeSprintersPoints;
  }

  /**
   * A method to add a stage result to the race result. This is useful as a riders results in a
↪  race
```

```
75      * is just a sum of their results in all a races stages. E.g. RaceResults = Stage1Result +
76      * Stage2Result + Stage3Result + ...
77      *
78      * @param stageResult the stage results which should be added to a race result.
79      */
80     public void addStageResult(StageResult stageResult) {
81       this.cumulativeAdjustedElapsedTime =
82           this.cumulativeAdjustedElapsedTime.plus(stageResult.getAdjustedElapsedTime());
83       this.cumulativeSprintersPoints += stageResult.getSprintersPoints();
84       this.cumulativeMountainPoints += stageResult.getMountainPoints();
85     }
86   }
```

6 Rider.java

```
1   package cycling;
2
3   import java.io.Serializable;
4
5   /** Rider class which represents a rider in the CyclingPortal. */
6   public class Rider implements Serializable {
7     private final Team team;
8     private final String name;
9     private final int yearOfBirth;
10
11    // Highest used ID count, used in order to avoid ID clashes.
12    private static int count = 0;
13    private final int id;
14
15    /**
16     * Constructor for a Rider in a CyclingPortal
17     *
18     * @param team the team the rider races for.
19     * @param name the riders name, which cannot be null.
20     * @param yearOfBirth the riders year of birth, which must be greater than 1900.
21     * @throws IllegalArgumentException thrown if the riders name is null or the riders birth year is
22     *     not greater than 1900.
23     */
24    public Rider(Team team, String name, int yearOfBirth) throws IllegalArgumentException {
25      if (name == null) {
26        throw new java.lang.IllegalArgumentException("The rider's name cannot be null.");
27      }
28      if (yearOfBirth < 1900) {
29        throw new java.lang.IllegalArgumentException(
30            "The rider's birth year is invalid, must be greater than 1900.");
31      }
32
33      this.team = team;
34      this.name = name;
35      this.yearOfBirth = yearOfBirth;
36      this.id = Rider.count++;
37    }
38
39    /** Method to reset the static rider ID counter, used for loading and erasing. */
40    static void resetIdCounter() {
41      count = 0;
42    }
```

```
43
44    /**
45     * Method to get the static rider ID counter, used for saving.
46     *
47     * @return the lowest known available rider ID.
48     */
49    static int getIdCounter() {
50      return count;
51    }
52
53    /**
54     * Method to set the static rider ID counter to a specific value, used for loading and erasing.
55     *
56     * @param newCount the number the ID counter should be set to.
57     */
58    static void setIdCounter(int newCount) {
59      count = newCount;
60    }
61
62    /**
63     * Method to get the Riders ID.
64     *
65     * @return the Riders ID.
66     */
67    public int getId() {
68      return id;
69    }
70
71    /**
72     * Method to get the Riders Team.
73     *
74     * @return the Team the rider races for.
75     */
76    public Team getTeam() {
77      return team;
78    }
79  }
```

7 SavedCyclingPortal.java

```
1   package cycling;
2
3   import java.io.Serializable;
4   import java.util.ArrayList;
5
6   /**
7    * Class which is used to save the state of the CyclingPortal. Teams, Riders, Races, Stages, and
8    * Segments are all saved along with their respective ID counters.
9    */
10  public class SavedCyclingPortal implements Serializable {
11    final ArrayList<Team> teams;
12    final ArrayList<Rider> riders;
13    final ArrayList<Race> races;
14    final ArrayList<Stage> stages;
15    final ArrayList<Segment> segments;
16    final int teamIdCount;
17    final int riderIdCount;
```

```
18    final int raceIdCount;
19    final int stageIdCount;
20    final int segmentIdCount;
21
22    /**
23     * Constructor for a SavedCyclingPortal which is used in saving and loading.
24     *
25     * @param teams the teams to be saved.
26     * @param riders the riders to be saved.
27     * @param races the races to be saved.
28     * @param stages the stages to be saved.
29     * @param segments the segments to be saved.
30     * @param teamIdCount the highest known team ID, saved in order to avoid ID clashes.
31     * @param riderIdCount the highest known rider ID, saved in order to avoid ID clashes.
32     * @param raceIdCount the highest known race ID, saved in order to avoid ID clashes.
33     * @param stageIdCount the highest known stage ID, saved in order to avoid ID clashes.
34     * @param segmentIdCount the highest known segment ID, saved in order to avoid ID clashes.
35     */
36    public SavedCyclingPortal(
37        ArrayList<Team> teams,
38        ArrayList<Rider> riders,
39        ArrayList<Race> races,
40        ArrayList<Stage> stages,
41        ArrayList<Segment> segments,
42        int teamIdCount,
43        int riderIdCount,
44        int raceIdCount,
45        int stageIdCount,
46        int segmentIdCount) {
47      this.teams = teams;
48      this.riders = riders;
49      this.races = races;
50      this.stages = stages;
51      this.segments = segments;
52      this.teamIdCount = teamIdCount;
53      this.riderIdCount = riderIdCount;
54      this.raceIdCount = raceIdCount;
55      this.stageIdCount = stageIdCount;
56      this.segmentIdCount = segmentIdCount;
57    }
58  }
```

8 Segment.java

```
1   package cycling;
2
3   import java.io.Serializable;
4   import java.time.LocalTime;
5   import java.util.HashMap;
6   import java.util.List;
7   import java.util.Map;
8   import java.util.stream.Collectors;
9
10  /**
11   * Segment Class. This represents a segment of a stage in a rice in the cycling portal. This deals
12   * with details about the segment as well as the segments results.
13   */
```

```java
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<>();

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

  /**
   * Constructor method that creates a segment for a given stage, segment type and location.
   *
   * @param stage The stage object which this segment is in. The stage cannot be waiting for
   *     results
   *     or be a time-trial stage.
   * @param type The type of segment, can be either SPRINT, C4, C3, C2, C1, or HC.
   * @param location The location of the segment in the stage in kilometers, cannot be longer than
   *     the length of the stage.
   * @throws InvalidLocationException Thrown if the location is out of bounds of the stage length.
   * @throws InvalidStageStateException Thrown if the stage is waiting for results.
   * @throws InvalidStageTypeException Thrown if a segment is attempted to be added to a time trial
   *     stage.
   */
  public Segment(Stage stage, SegmentType type, double location)
      throws InvalidLocationException, InvalidStageStateException, InvalidStageTypeException {
    if (location > stage.getLength()) {
      throw new InvalidLocationException("The location is out of bounds of the stage length.");
    }
    if (stage.isWaitingForResults()) {
      throw new InvalidStageStateException("The stage is waiting for results.");
    }
    if (stage.getType().equals(StageType.TT)) {
      throw new InvalidStageTypeException("Time-trial stages cannot contain any segments.");
    }
    this.stage = stage;
    // ID counter represents the highest known ID at the current time to ensure
    // there
    // are no ID collisions.
    this.id = Segment.count++;
    this.type = type;
    this.location = location;
  }

  /** Reset the static segment ID counter. Used for erasing/loading the CyclingPortal. */
  static void resetIdCounter() {
    count = 0;
  }

  /**
   * Method to get the current state of the static ID counter.
```

```
71      *
72      * @return the highest segment ID stored currently.
73      */
74     static int getIdCounter() {
75       return count;
76     }
77
78     /**
79      * Method that sets the static ID counter to a given value. Used when loading to avoid ID
80      * collisions.
81      *
82      * @param newCount: new value of the static ID counter.
83      */
84     static void setIdCounter(int newCount) {
85       count = newCount;
86     }
87
88     /**
89      * Method to get the ID of the segment object.
90      *
91      * @return id: the Segments unique ID value.
92      */
93     public int getId() {
94       return id;
95     }
96
97     /**
98      * Method to get the Stage which the segment exists in.
99      *
100     * @return The stage object.
101     */
102    public Stage getStage() {
103      return stage;
104    }
105
106    /**
107     * Method to get the location of the segment within the stage.
108     *
109     * @return the location in kilometers as a double.
110     */
111    public double getLocation() {
112      return location;
113    }
114
115    /**
116     * Method to register the time which a given rider completed the segment.
117     *
118     * @param rider The rider which finished the segment.
119     * @param finishTime The time which the rider finished the segment.
120     */
121    public void registerResults(Rider rider, LocalTime finishTime) {
122      // Create a segment result for the rider.
123      SegmentResult result = new SegmentResult(finishTime);
124      // Associate the result with the rider in the result HashMap.
125      results.put(rider, result);
126    }
127
128    /**
```

```java
    * Method to get a given riders results in this segment.
    *
    * @param rider The rider whose results will be returned.
    * @return The results the rider received in the segment.
    */
  public SegmentResult getRiderResult(Rider rider) {
    // First calculate the segments results, such as riders position and points.
    calculateResults();
    // Then return the results for the requested rider.
    return results.get(rider);
  }

  /**
   * Method to remove a given riders results from the segment.
   *
   * @param rider The rider object whose results should be removed.
   */
  public void removeRiderResults(Rider rider) {
    results.remove(rider);
  }

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

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

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

      // Points Calculation
      int[] pointsDistribution =
          getPointsDistribution(); // Get the point distribution based on the segment type.
      if (position <= pointsDistribution.length) {
        // Get the riders points based on their position
        int points = pointsDistribution[i];
        if (this.type.equals(SegmentType.SPRINT)) {
          // If the segment is a sprint, set the riders points as sprinters points.
```

```
187             result.setSprintersPoints(points);
188             result.setMountainPoints(0);
189           } else {
190             // If the segment is not a sprint, set the riders points as mountain points.
191             result.setSprintersPoints(0);
192             result.setMountainPoints(points);
193           }
194         } else {
195           // If the rider does not finish in a point-awarding position, reward 0 points.
196           result.setMountainPoints(0);
197           result.setSprintersPoints(0);
198         }
199       }
200     }
201
202     /**
203      * Private method to get the point distribution of the segment based on the type of segment.
204      *
205      * @return an array of integers that represent the points that should be rewarded based on the
206      *     segment type.
207      */
208     private int[] getPointsDistribution() {
209       return switch (type) {
210         case HC -> HC_POINTS;
211         case C1 -> C1_POINTS;
212         case C2 -> C2_POINTS;
213         case C3 -> C3_POINTS;
214         case C4 -> C4_POINTS;
215         case SPRINT -> SPRINT_POINTS;
216       };
217     }
218   }
```

9 SegmentResult.java

```
1   package cycling;
2
3   import java.io.Serializable;
4   import java.time.LocalTime;
5   import java.util.Comparator;
6
7   /** This represents a given recorded result in a segment. */
8   public class SegmentResult implements Serializable {
9     private final LocalTime finishTime;
10    private int position;
11    private int sprintersPoints;
12    private int mountainPoints;
13
14    // A comparator which sorts SegmentResults based on Elapsed Time in ascending order. The
15    // result with the shortest time will come first.
16    protected static final Comparator<SegmentResult> sortByFinishTime =
17        Comparator.comparing(SegmentResult::getFinishTime);
18
19    /**
20     * Constructor for a given result in a stage.
21     *
22     * @param finishTime The time at which the segment was finished.
```

```
23      */
24     public SegmentResult(LocalTime finishTime) {
25       this.finishTime = finishTime;
26     }
27
28     /**
29      * A method to get the time at which the segment was finished.
30      *
31      * @return The LocalTime at which the segment was finished.
32      */
33     public LocalTime getFinishTime() {
34       return finishTime;
35     }
36
37     /**
38      * A method to set the position in a stage.
39      *
40      * @param position the position of the rider in the stage.
41      */
42     public void setPosition(int position) {
43       this.position = position;
44     }
45
46     /**
47      * A method to set the mountain points in a stage.
48      *
49      * @param points the mountain points received in the stage.
50      */
51     public void setMountainPoints(int points) {
52       this.mountainPoints = points;
53     }
54
55     /**
56      * A method to set the sprinters points in a stage.
57      *
58      * @param points the sprinters points received in the stage.
59      */
60     public void setSprintersPoints(int points) {
61       this.sprintersPoints = points;
62     }
63
64     /**
65      * A method to get the mountain points in a stage.
66      *
67      * @return the mountain points received in the stage
68      */
69     public int getMountainPoints() {
70       return this.mountainPoints;
71     }
72
73     /**
74      * A method to get the sprinters points in a stage.
75      *
76      * @return the sprinters points received in the stage.
77      */
78     public int getSprintersPoints() {
79       return this.sprintersPoints;
80     }
```

```
}
```

10 Stage.java

```
package cycling;

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

/** Stage Class. This represents a stage in a race of the CyclingPortal */
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<>();

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

  /**
   * Constructor method that sets a Stage up with a race, name, description, length startTime and
   * type.
   *
   * @param race: Race that the Stage is in.
   * @param name: name of the Stage, cannot be null, empty, have more than 30 characters or have
   *     white space.
   * @param description: description of the Stage.
   * @param length: length of the Stage in kilometers, cannot be less than 5km.
   * @param startTime: start time of the Stage.
   * @param type: the type of Stage, can be either FLAT, MEDIUM_MOUNTAIN, HIGH_MOUNTAIN, TT.
   * @throws InvalidNameException Thrown if the name is empty, null, longer than 30 characters or
   *     contains whitespace.
   * @throws InvalidLengthException Thrown if the length is less than 5km.
   */
  public Stage(
      Race race,
      String name,
      String description,
      double length,
```

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

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

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

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

```
286
287   /**
288    * Method to get the HashMap of Riders and their associated Results in the stage.
289    *
290    * @return The HashMap of Riders and their associated Results in the stage.
291    */
292   public HashMap<Rider, StageResult> getStageResults() {
293     calculateResults();
294     return results;
295   }
296
297   /**
298    * Sort all the riders with a registered result in the stage by their elapsed time.
299    *
300    * @return A list of riders sorted in ascending order of their elapsed time in the stage.
301    */
302   private List<Rider> sortRiderResults() {
303     return results.entrySet().stream()
304         .sorted(Map.Entry.comparingByValue(StageResult.sortByElapsedTime))
305         .map(Map.Entry::getKey)
306         .collect(Collectors.toList());
307   }
308
309   /** A private method to calculate all riders results in the stage. */
310   private void calculateResults() {
311     // Get a list of all riders with registered results sorted by their elapsed time.
312     List<Rider> riders = sortRiderResults();
313
314     for (int i = 0; i < results.size(); i++) {
315       Rider rider = riders.get(i);
316       StageResult result = results.get(rider);
317       int position = i + 1;
318
319       // Position Calculation
320       result.setPosition(position); // Assign the rider their position.
321
322       // Adjusted Elapsed Time Calculations
323       if (i == 0) {
324         // If the rider is the first in the race then their adjusted time = elapsed time.
325         result.setAdjustedElapsedTime(result.getElapsedTime());
326       } else {
327         // Get the previous riders & current riders times.
328         Rider prevRider = riders.get(i - 1);
329         Duration prevTime = results.get(prevRider).getElapsedTime();
330         Duration time = results.get(rider).getElapsedTime();
331
332         // If the difference between the current riders time and the previous time is less than 1
333         // second.
334         int timeDiff = time.minus(prevTime).compareTo(Duration.ofSeconds(1));
335         if (timeDiff <= 0) {
336           // Close Finish Condition
337           // Set the current riders adjusted time to be the same as the previous riders adjusted
338           // time.
339           Duration prevAdjustedTime = results.get(prevRider).getAdjustedElapsedTime();
340           result.setAdjustedElapsedTime(prevAdjustedTime);
341         } else {
342           // Far Finish Condition
343           // Set the current riders adjusted time = elapsed time.
```

```
344             result.setAdjustedElapsedTime(time);
345           }
346         }
347
348         // Points Calculation
349         int sprintersPoints = 0;
350         int mountainPoints = 0;
351         for (Segment segment : segments) {
352           // Sum the riders points in each segment.
353           SegmentResult segmentResult = segment.getRiderResult(rider);
354           sprintersPoints += segmentResult.getSprintersPoints();
355           mountainPoints += segmentResult.getMountainPoints();
356         }
357         int[] pointsDistribution = getPointDistribution();
358         if (position <= pointsDistribution.length) {
359           // Add any sprinters points the rider may have gained for the stage.
360           sprintersPoints += pointsDistribution[i];
361         }
362         result.setSprintersPoints(sprintersPoints);
363         result.setMountainPoints(mountainPoints);
364       }
365     }
366
367     /**
368      * Private method to get the point distribution based on the stage type.
369      *
370      * @return the distribution of points based on the stage type.
371      */
372     private int[] getPointDistribution() {
373       return switch (type) {
374         case FLAT -> FLAT_POINTS;
375         case MEDIUM_MOUNTAIN -> MEDIUM_POINTS;
376         case HIGH_MOUNTAIN -> HIGH_POINTS;
377         case TT -> TT_POINTS;
378       };
379     }
380 }
```

11 StageResult.java

```
1  package cycling;
2
3  import java.io.Serializable;
4  import java.time.Duration;
5  import java.time.LocalTime;
6  import java.util.Comparator;
7
8  /** This represents a given recorded result in a stage. */
9  public class StageResult implements Serializable {
10   private final LocalTime[] checkpoints;
11   private final Duration elapsedTime;
12   private Duration adjustedElapsedTime;
13   private int position;
14   private int sprintersPoints;
15   private int mountainPoints;
16
17   // A comparator which sorts StageResults based on Elapsed Time in ascending order. The
```

```java
  // result with the shortest time will come first.
  protected static final Comparator<StageResult> sortByElapsedTime =
      Comparator.comparing(StageResult::getElapsedTime);

  /**
   * Constructor for a given results in a stage.
   *
   * @param checkpoints The array of LocalTimes at which each checkpoint was crossed/
   */
  public StageResult(LocalTime[] checkpoints) {
    this.checkpoints = checkpoints;
    this.elapsedTime = Duration.between(checkpoints[0], checkpoints[checkpoints.length - 1]);
  }

  /**
   * A method to get the times at which each checkpoint was crossed.
   *
   * @return The array of LocalTimes at which each checkpoint was crossed.
   */
  public LocalTime[] getCheckpoints() {
    return this.checkpoints;
  }

  /**
   * A method to get the elapsed time since the start of the stage.
   *
   * @return The duration of time since the stage started.
   */
  public Duration getElapsedTime() {
    return elapsedTime;
  }

  /**
   * A method to set the position in a stage.
   *
   * @param position the position of the rider in the stage.
   */
  public void setPosition(int position) {
    this.position = position;
  }

  /**
   * A method to set the adjusted elapsed time in a stage.
   *
   * @param adjustedElapsedTime the adjusted elapsed time in the stage.
   */
  public void setAdjustedElapsedTime(Duration adjustedElapsedTime) {
    this.adjustedElapsedTime = adjustedElapsedTime;
  }

  /**
   * A method to get the adjusted elapsed time in a stage as a duration.
   *
   * @return the adjusted elapsed time as a duration.
   */
  public Duration getAdjustedElapsedTime() {
    return adjustedElapsedTime;
  }
```

```
76
77    /**
78     * A method to get the adjusted elapsed time in a stage as a duration.
79     *
80     * @return the adjusted elapsed time as a duration.
81     */
82    public LocalTime getAdjustedElapsedLocalTime() {
83      return checkpoints[0].plus(adjustedElapsedTime);
84    }
85
86    /**
87     * A method to set the mountain points in a stage.
88     *
89     * @param points the mountain points received in the stage.
90     */
91    public void setMountainPoints(int points) {
92      this.mountainPoints = points;
93    }
94
95    /**
96     * A method to set the sprinters points in a stage.
97     *
98     * @param points the sprinters points received in the stage.
99     */
100   public void setSprintersPoints(int points) {
101     this.sprintersPoints = points;
102   }
103
104   /**
105    * A method to get the mountain points in a stage.
106    *
107    * @return the mountain points received in the stage
108    */
109   public int getMountainPoints() {
110     return mountainPoints;
111   }
112
113   /**
114    * A method to get the sprinters points in a stage.
115    *
116    * @return the sprinters points received in the stage.
117    */
118   public int getSprintersPoints() {
119     return sprintersPoints;
120   }
121 }
```

12 Team.java

```
1  package cycling;
2
3  import java.io.Serializable;
4  import java.util.ArrayList;
5
6  /** Team class. This represents a team of riders. */
7  public class Team implements Serializable {
8    private final String name;
```

```
9   private final String description;
10
11  private final ArrayList<Rider> riders = new ArrayList<>();
12  private static int count = 0;
13  private final int id;
14
15  /**
16   * Constructor method that sets up the Team with a name and a description.
17   *
18   * @param name of the team.
19   * @param description of the team.
20   * @throws InvalidNameException Thrown if the team name is null, empty, has more than 30
21   *     characters or contains any whitespace.
22   */
23  public Team(String name, String description) throws InvalidNameException {
24    if (name == null
25        || name.isEmpty()
26        || name.length() > 30
27        || CyclingPortal.containsWhitespace(name)) {
28      throw new InvalidNameException(
29          "Team name cannot be null, empty, have more than 30 characters or have white spaces.");
30    }
31    this.name = name;
32    this.description = description;
33    this.id = Team.count++;
34  }
35
36  /** Method to reset the static ID counter. */
37  static void resetIdCounter() {
38    count = 0;
39  }
40
41  /**
42   * Method to get the current state of the static ID counter.
43   *
44   * @return the highest race ID stored currently.
45   */
46  static int getIdCounter() {
47    return count;
48  }
49
50  /**
51   * Method that sets the static ID counter to a given value. Used when loading to avoid ID
52   * collisions.
53   *
54   * @param newCount: new value of the static ID counter.
55   */
56  static void setIdCounter(int newCount) {
57    count = newCount;
58  }
59
60  /**
61   * Method that gets the name of the Team.
62   *
63   * @return name of the Team.
64   */
65  public String getName() {
66    return name;
```

```
67    }
68
69    /**
70     * Method that gets the ID of the Team.
71     *
72     * @return ID of the Team.
73     */
74    public int getId() {
75      return id;
76    }
77
78    /**
79     * Method that removes a Rider from the Team.
80     *
81     * @param rider to be removed.
82     */
83    public void removeRider(Rider rider) {
84      riders.remove(rider);
85    }
86
87    /**
88     * Method to get the Riders in the Team.
89     *
90     * @return A list of Riders in the Team.
91     */
92    public ArrayList<Rider> getRiders() {
93      return riders;
94    }
95
96    /**
97     * Method that adds a Rider to the Team.
98     *
99     * @param rider to be added to the Team.
100    */
101   public void addRider(Rider rider) {
102     riders.add(rider);
103   }
104 }
```