

CyclingPortal Printout

123456789 & 987654321

Contents

1	CategorizedClimb.java	2
2	CyclingPortal.java	2
3	IntermediateSprint.java	13
4	Race.java	14
5	RaceResult.java	18
6	Rider.java	20
7	SavedCyclingPortal.java	21
8	Segment.java	21
9	SegmentResult.java	25
10	Stage.java	27
11	StageResult.java	33
12	Team.java	36

1 CategorizedClimb.java

```

1  package cycling;
2
3  /**
4   * Categorised Climb class. This represents a type of Segment that has a stage, location, type,
5   * average gradient and a length.
6   */
7  public class CategorizedClimb extends Segment {
8      private final Double averageGradient;
9      private final Double length;
10
11     /**
12      * Constructor method that sets up the Categorised Climb with a stage, location, type, average
13      * gradient and length.
14      *
15      * @param stage that the Categorised Climb is in.
16      * @param location of the Categorised Climb.
17      * @param type of Categorised Climb.
18      * @param averageGradient of the Categorised Climb.
19      * @param length of the Categorised Climb.
20      * @throws InvalidLocationException Thrown if the location is out of bounds of the Stage length.
21      * @throws InvalidStageStateException Thrown if the Stage is waiting for results.
22      * @throws InvalidStageTypeException Thrown if the type is a time trial.
23      */
24     public CategorizedClimb(
25         Stage stage, Double location, SegmentType type, Double averageGradient, Double length)
26         throws InvalidLocationException, InvalidStageStateException, InvalidStageTypeException {
27         super(stage, type, location);
28         this.averageGradient = averageGradient;
29         this.length = length;
30     }
31 }

```

2 CyclingPortal.java

```

1  package cycling;
2
3  import java.io.*;
4  import java.time.LocalDateTime;
5  import java.time.LocalTime;
6  import java.util.ArrayList;
7  import java.util.List;
8
9  // TODO:
10 //      - Documentation/Comments
11
12 public class CyclingPortal implements CyclingPortalInterface {
13     // ArrayLists for all of a cycling portal instances teams, riders, races, stages and segments.
14     // Although HashMaps could have been used here to get riders by int ID, it would be slower in the
15     // long run as we would need to constantly convert it back to arrays to output results.
16     private ArrayList<Team> teams = new ArrayList<>();
17     private ArrayList<Rider> riders = new ArrayList<>();
18     private ArrayList<Race> races = new ArrayList<>();
19     private ArrayList<Stage> stages = new ArrayList<>();
20     private ArrayList<Segment> segments = new ArrayList<>();
21 }

```

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

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

```
138     Race race = races.get(i);
139     raceIDs[i] = race.getId();
140 }
141 return raceIDs;
142 }
143
144 @Override
145 public int createRace(String name, String description)
146     throws IllegalArgumentException, InvalidNameException {
147     // Check a race with this name does not already exist in the system.
148     for (Race race : races) {
149         if (race.getName().equals(name)) {
150             throw new IllegalArgumentException("A Race with the name " + name + " already exists.");
151         }
152     }
153     Race race = new Race(name, description);
154     races.add(race);
155     return race.getId();
156 }
157
158 @Override
159 public String viewRaceDetails(int raceId) throws IDNotRecognisedException {
160     Race race = getRaceById(raceId);
161     return race.getDetails();
162 }
163
164 @Override
165 public void removeRaceById(int raceId) throws IDNotRecognisedException {
166     Race race = getRaceById(raceId);
167     // Remove all the races stages from the CyclingPortal.
168     for (final Stage stage : race.getStages()) {
169         stages.remove(stage);
170     }
171     races.remove(race);
172 }
173
174 @Override
175 public int getNumberOfStages(int raceId) throws IDNotRecognisedException {
176     Race race = getRaceById(raceId);
177     return race.getStages().size();
178 }
179
180 @Override
181 public int addStageToRace(
182     int raceId,
183     String stageName,
184     String description,
185     double length,
186     LocalDateTime startTime,
187     StageType type)
188     throws IDNotRecognisedException, IllegalArgumentException, InvalidNameException,
189         InvalidLengthException {
190     Race race = getRaceById(raceId);
191     // Check a stage with this name does not already exist in the system.
192     for (final Stage stage : stages) {
193         if (stage.getName().equals(stageName)) {
194             throw new IllegalArgumentException("A stage with the name " + stageName + " already exists.");
195         }
196     }
197 }
```

```
196     }
197     Stage stage = new Stage(race, stageName, description, length, startTime, type);
198     stages.add(stage);
199     race.addStage(stage);
200     return stage.getId();
201 }
202
203 @Override
204 public int[] getRaceStages(int raceId) throws IDNotRecognisedException {
205     Race race = getRaceById(raceId);
206     ArrayList<Stage> raceStages = race.getStages();
207     int[] raceStagesId = new int[raceStages.size()];
208     // Gathers the Stage ID's of the Stages in the Race.
209     for (int i = 0; i < raceStages.size(); i++) {
210         Stage stage = race.getStages().get(i);
211         raceStagesId[i] = stage.getId();
212     }
213     return raceStagesId;
214 }
215
216 @Override
217 public double getStageLength(int stageId) throws IDNotRecognisedException {
218     Stage stage = getStageById(stageId);
219     return stage.getLength();
220 }
221
222 @Override
223 public void removeStageById(int stageId) throws IDNotRecognisedException {
224     Stage stage = getStageById(stageId);
225     Race race = stage.getRace();
226     // Removes stage from both the Races and Stages.
227     race.removeStage(stage);
228     stages.remove(stage);
229 }
230
231 @Override
232 public int addCategorizedClimbToStage(
233     int stageId, Double location, SegmentType type, Double averageGradient, Double length)
234     throws IDNotRecognisedException, InvalidLocationException, InvalidStageStateException,
235     InvalidStageTypeException {
236     Stage stage = getStageById(stageId);
237     CategorizedClimb climb = new CategorizedClimb(stage, location, type, averageGradient, length);
238     // Adds Categorized Climb to both the list of Segments and the Stage.
239     segments.add(climb);
240     stage.addSegment(climb);
241     return climb.getId();
242 }
243
244 @Override
245 public int addIntermediateSprintToStage(int stageId, double location)
246     throws IDNotRecognisedException, InvalidLocationException, InvalidStageStateException,
247     InvalidStageTypeException {
248     Stage stage = getStageById(stageId);
249     IntermediateSprint sprint = new IntermediateSprint(stage, location);
250     // Adds Intermediate Sprint to both the list of Segments and the Stage.
251     segments.add(sprint);
252     stage.addSegment(sprint);
253     return sprint.getId();
254 }
```

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

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



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

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

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

```
544
545 @Override
546 public void removeRaceByName(String name) throws NameNotRecognisedException {
547     for (final Race race : races) {
548         if (race.getName().equals(name)) {
549             races.remove(race);
550             return;
551         }
552     }
553     throw new NameNotRecognisedException("Race name is not in the system.");
554 }
555
556 @Override
557 public int[] getRidersGeneralClassificationRank(int raceId) throws IDNotRecognisedException {
558     Race race = getRaceById(raceId);
559     List<Rider> riders = race.getRidersByAdjustedElapsedTime();
560     int[] riderIds = new int[riders.size()];
561     // Gathers Rider ID's ordered by their Adjusted Elapsed Times.
562     for (int i = 0; i < riders.size(); i++) {
563         riderIds[i] = riders.get(i).getId();
564     }
565     return riderIds;
566 }
567
568 @Override
569 public LocalTime[] getGeneralClassificationTimesInRace(int raceId)
570     throws IDNotRecognisedException {
571     Race race = getRaceById(raceId);
572     // Gets a list of Riders from the Stage ordered by their Adjusted Elapsed Times.
573     List<Rider> riders = race.getRidersByAdjustedElapsedTime();
574     LocalTime[] riderTimes = new LocalTime[riders.size()];
575     // Gathers Riders' Cumulative Adjusted Elapsed LocalTimes ordered by their Adjusted Elapsed
576     // Times.
577     for (int i = 0; i < riders.size(); i++) {
578         riderTimes[i] = race.getRiderResults(riders.get(i)).getCumulativeAdjustedElapsedLocalTime();
579     }
580     return riderTimes;
581 }
582
583 @Override
584 public int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException {
585     Race race = getRaceById(raceId);
586     List<Rider> riders = race.getRidersByAdjustedElapsedTime();
587     int[] riderIds = new int[riders.size()];
588     // Gathers Riders' Cumulative Sprinters Points ordered by their Adjusted Elapsed Times.
589     for (int i = 0; i < riders.size(); i++) {
590         riderIds[i] = race.getRiderResults(riders.get(i)).getCumulativeSprintersPoints();
591     }
592     return riderIds;
593 }
594
595 @Override
596 public int[] getRidersMountainPointsInRace(int raceId) throws IDNotRecognisedException {
597     Race race = getRaceById(raceId);
598     List<Rider> riders = race.getRidersByAdjustedElapsedTime();
599     int[] riderIds = new int[riders.size()];
600     // Gathers Riders' Cumulative Mountain Points ordered by their Adjusted Elapsed Times.
601     for (int i = 0; i < riders.size(); i++) {
```

```

602     riderIds[i] = race.getRiderResults(riders.get(i)).getCumulativeMountainPoints();
603 }
604 return riderIds;
605 }
606
607 @Override
608 public int[] getRidersPointClassificationRank(int raceId) throws IDNotRecognisedException {
609     Race race = getRaceById(raceId);
610     List<Rider> riders = race.getRidersBySprintersPoints();
611     int[] riderIds = new int[riders.size()];
612     // Gathers Rider ID's ordered by their Sprinters Points.
613     for (int i = 0; i < riders.size(); i++) {
614         riderIds[i] = riders.get(i).getId();
615     }
616     return riderIds;
617 }
618
619 @Override
620 public int[] getRidersMountainPointClassificationRank(int raceId)
621     throws IDNotRecognisedException {
622     Race race = getRaceById(raceId);
623     List<Rider> riders = race.getRidersByMountainPoints();
624     int[] riderIds = new int[riders.size()];
625     // Gathers Rider ID's ordered by their Mountain Points.
626     for (int i = 0; i < riders.size(); i++) {
627         riderIds[i] = riders.get(i).getId();
628     }
629     return riderIds;
630 }
631 }

```

3 IntermediateSprint.java

```

1  package cycling;
2
3  /** 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

```

1  package cycling;
2
3  import java.io.Serializable;
4  import java.time.LocalDateTime;
5  import java.util.*;
6  import java.util.stream.Collectors;
7
8  /**
9   * Race Class. This represents a Race that holds a Race's Stages, Riders Results, and also contains
10  * methods that deal with these.
11  */
12  public class Race implements Serializable {
13
14      private final String name;
15      private final String description;
16
17      private final ArrayList<Stage> stages = new ArrayList<>();
18
19      private HashMap<Rider, RaceResult> results = new HashMap<>();
20
21      private static int count = 0;
22      private final int id;
23
24      /**
25       * Constructor method that sets up Rider with a name and a description.
26       *
27       * @param name: Cannot be empty, null, have a length greater than 30 or contain any whitespace.
28       * @param description: A description of the race.
29       * @throws InvalidNameException Thrown if the Race name does not meet name requirements stated
30       *         above.
31       */
32      public Race(String name, String description) throws InvalidNameException {
33          if (name == null
34              || name.isEmpty()
35              || name.length() > 30
36              || CyclingPortal.containsWhitespace(name)) {
37              throw new InvalidNameException(
38                  "The name cannot be null, empty, have more than 30 characters, or have white spaces.");
39          }
40          this.name = name;
41          this.description = description;
42          // ID counter represents the highest known ID at the current time to ensure there
43          // are no ID collisions.
44          this.id = Race.count++;
45      }
46
47      /** Method that resets the static ID counter of the Race. Used for erasing and loading. */
48      static void resetIdCounter() {
49          count = 0;
50      }
51
52      /**
53       * Method to get the current state of the static ID counter.
54       *
55       * @return the highest race ID stored currently.
56       */

```

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

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



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

```

231         // Convert to a list of riders.
232         .collect(Collectors.toList());
233     }
234
235     /**
236      * Method to register the Rider's Result to the Stage.
237      *
238      * @param rider: Rider whose Result needs to be registered.
239      * @param stageResult: Stage that the Result will be added to.
240      */
241     private void registerRiderResults(Rider rider, StageResult stageResult) {
242         if (results.containsKey(rider)) {
243             // If results already exist for a given rider add the current stage results
244             // to the existing total race results.
245             results.get(rider).addStageResult(stageResult);
246         } else {
247             // If no race results exists, create a new RaceResult object based on the current
248             // stage results.
249             RaceResult raceResult = new RaceResult();
250             raceResult.addStageResult(stageResult);
251             results.put(rider, raceResult);
252         }
253     }
254
255     /** Private method that calculates the results for each Rider. */
256     private void calculateResults() {
257         // Clear existing results.
258         results = new HashMap<>();
259         // We must loop over all stages and collect their results for each rider as each riders results
260         // are dependent on their position in the race, and thus the results of the other riders.
261         for (Stage stage : stages) {
262             HashMap<Rider, StageResult> stageResults = stage.getStageResults();
263             for (Rider rider : stageResults.keySet()) {
264                 registerRiderResults(rider, stageResults.get(rider));
265             }
266         }
267     }
268 }

```

5 RaceResult.java

```

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

```

```
17 // A comparator which sorts RaceResults based on Adjusted Elapsed Time in ascending order. The
18 // result with the shortest time will come first.
19 protected static final Comparator<RaceResult> sortByAdjustedElapsedTime =
20     Comparator.comparing(RaceResult::getCumulativeAdjustedElapsedTime);
21
22 // A comparator which sorts RaceResults based on Sprinters Points in descending order. The result
23 // with the most points will come first.
24 protected static final Comparator<RaceResult> sortBySprintersPoints =
25     (RaceResult result1, RaceResult result2) ->
26         Integer.compare(
27             result2.getCumulativeSprintersPoints(), result1.getCumulativeSprintersPoints());
28
29 // A comparator which sorts RaceResults based on Mountain Points in descending order. The result
30 // with the most points will come first.
31 protected static final Comparator<RaceResult> sortByMountainPoints =
32     (RaceResult result1, RaceResult result2) ->
33         Integer.compare(
34             result2.getCumulativeMountainPoints(), result1.getCumulativeMountainPoints());
35
36 /**
37  * A method to get the recorded Adjusted Elapsed Time over all stages.
38  *
39  * @return The cumulative adjusted elapsed time as a duration.
40  */
41 public Duration getCumulativeAdjustedElapsedTime() {
42     return this.cumulativeAdjustedElapsedTime;
43 }
44
45 /**
46  * A method to get the recorded Adjusted Elapsed Time over all stages as a LocalTime.
47  *
48  * @return The cumulative adjusted elapsed time as a Local Time
49  */
50 public LocalTime getCumulativeAdjustedElapsedLocalTime() {
51     // Calculated the AET as a Local time by adding the duration to midnight: 0:00 + Duration
52     return LocalTime.MIDNIGHT.plus(this.cumulativeAdjustedElapsedTime);
53 }
54
55 /**
56  * A method to get the recorded Mountain Points over all stages and segments.
57  *
58  * @return The cumulative mountain points.
59  */
60 public int getCumulativeMountainPoints() {
61     return this.cumulativeMountainPoints;
62 }
63
64 /**
65  * A method to get the recorded Sprinters Points over all stages and segments.
66  *
67  * @return The cumulative sprinters points.
68  */
69 public int getCumulativeSprintersPoints() {
70     return this.cumulativeSprintersPoints;
71 }
72
73 /**
```

```

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

```

6 Rider.java

```

1  package cycling;
2
3  import java.io.Serializable;
4
5  public class Rider implements Serializable {
6      private final Team team;
7      private final String name;
8      private final int yearOfBirth;
9
10     private static int count = 0;
11     private final int id;
12
13     public Rider(Team team, String name, int yearOfBirth) throws IllegalArgumentException {
14         if (name == null) {
15             throw new java.lang.IllegalArgumentException("The rider's name cannot be null.");
16         }
17         if (yearOfBirth < 1900) {
18             throw new java.lang.IllegalArgumentException(
19                 "The rider's birth year is invalid, must be greater than 1900.");
20         }
21
22         this.team = team;
23         this.name = name;
24         this.yearOfBirth = yearOfBirth;
25         this.id = Rider.count++;
26     }
27
28     static void resetIdCounter() {
29         count = 0;
30     }
31
32     static int getIdCounter() {
33         return count;
34     }
35
36     static void setIdCounter(int newCount) {
37         count = newCount;
38     }
39
40     public int getId() {

```

```
41     return id;
42 }
43
44 public Team getTeam() {
45     return team;
46 }
47 }
```

7 SavedCyclingPortal.java

```
1  package cycling;
2
3  import java.io.Serializable;
4  import java.util.ArrayList;
5
6  public class SavedCyclingPortal implements Serializable {
7      final ArrayList<Team> teams;
8      final ArrayList<Rider> riders;
9      final ArrayList<Race> races;
10     final ArrayList<Stage> stages;
11     final ArrayList<Segment> segments;
12     final int teamIdCount;
13     final int riderIdCount;
14     final int raceIdCount;
15     final int stageIdCount;
16     final int segmentIdCount;
17
18     public SavedCyclingPortal(
19         ArrayList<Team> teams,
20         ArrayList<Rider> riders,
21         ArrayList<Race> races,
22         ArrayList<Stage> stages,
23         ArrayList<Segment> segments,
24         int teamIdCount,
25         int riderIdCount,
26         int raceIdCount,
27         int stageIdCount,
28         int segmentIdCount) {
29         this.teams = teams;
30         this.riders = riders;
31         this.races = races;
32         this.stages = stages;
33         this.segments = segments;
34         this.teamIdCount = teamIdCount;
35         this.riderIdCount = riderIdCount;
36         this.raceIdCount = raceIdCount;
37         this.stageIdCount = stageIdCount;
38         this.segmentIdCount = segmentIdCount;
39     }
40 }
```

8 Segment.java

```
1  package cycling;
2
3  import java.io.Serializable;
```

```

4  import java.time.LocalDateTime;
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 race in the cycling portal. This deals
12  * with details about the segment as well as the segments results.
13  */
14 public class Segment implements Serializable {
15     private static int count = 0;
16     private final Stage stage;
17     private final int id;
18     private final SegmentType type;
19     private final double location;
20
21     private final HashMap<Rider, SegmentResult> results = new HashMap<>();
22
23     // Segment sprinters/mountain points.
24     private static final int[] SPRINT_POINTS = {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
25     private static final int[] HC_POINTS = {20, 15, 12, 10, 8, 6, 4, 2};
26     private static final int[] C1_POINTS = {10, 8, 6, 4, 2, 1};
27     private static final int[] C2_POINTS = {5, 3, 2, 1};
28     private static final int[] C3_POINTS = {2, 1};
29     private static final int[] C4_POINTS = {1};
30
31     /**
32      * Constructor method that creates a segment for a given stage, segment type and location.
33      *
34      * @param stage The stage object which this segment is in. The stage cannot be waiting for
35      * ↪ results
36      * or be a time-trial stage.
37      * @param type The type of segment, can be either SPRINT, C4, C3, C2, C1, or HC.
38      * @param location The location of the segment in the stage in kilometers, cannot be longer than
39      * the length of the stage.
40      * @throws InvalidLocationException
41      * @throws InvalidStageStateException
42      * @throws InvalidStageTypeException
43      */
44     public Segment(Stage stage, SegmentType type, double location)
45         throws InvalidLocationException, InvalidStageStateException, InvalidStageTypeException {
46         if (location > stage.getLength()) {
47             throw new InvalidLocationException("The location is out of bounds of the stage length.");
48         }
49         if (stage.isWaitingForResults()) {
50             throw new InvalidStageStateException("The stage is waiting for results.");
51         }
52         if (stage.getType().equals(StageType.TT)) {
53             throw new InvalidStageTypeException("Time-trial stages cannot contain any segments.");
54         }
55         this.stage = stage;
56         // ID counter represents the highest known ID at the current time to ensure
57         // there
58         // are no ID collisions.
59         this.id = Segment.count++;
60         this.type = type;
61         this.location = location;

```

```
61     }
62
63     /** Reset the static segment ID counter. Used for erasing/loading the CyclingPortal. */
64     static void resetIdCounter() {
65         count = 0;
66     }
67
68     /**
69      * Method to get the current state of the static ID counter.
70      *
71      * @return the highest segment ID stored currently.
72      */
73     static int getIdCounter() {
74         return count;
75     }
76
77     /**
78      * Method that sets the static ID counter to a given value. Used when loading to avoid ID
79      * collisions.
80      *
81      * @param newCount: new value of the static ID counter.
82      */
83     static void setIdCounter(int newCount) {
84         count = newCount;
85     }
86
87     /**
88      * Method to get the ID of the segment object.
89      *
90      * @return id: the Segments's unique ID value.
91      */
92     public int getId() {
93         return id;
94     }
95
96     /**
97      * Method to get the Stage which the segment exists in.
98      *
99      * @return The stage object.
100     */
101    public Stage getStage() {
102        return stage;
103    }
104
105    /**
106     * Method to get the location of the segment within the stage.
107     *
108     * @return the location in kilometers as a double.
109     */
110    public double getLocation() {
111        return location;
112    }
113
114    /**
115     * Method to register the time which a given rider completed the segment.
116     *
117     * @param rider The rider which finished the segment.
118     * @param finishTime The time which the rider finished the segment.
```

```
119     */
120 public void registerResults(Rider rider, LocalTime finishTime) {
121     // Create a segment result for the rider.
122     SegmentResult result = new SegmentResult(finishTime);
123     // Associate the result with the rider in the result HashMap.
124     results.put(rider, result);
125 }
126
127 /**
128  * Method to get a given riders results in this segment.
129  *
130  * @param rider The rider whose results will be returned.
131  * @return The results the rider received in the segment.
132  */
133 public SegmentResult getRiderResult(Rider rider) {
134     // First calculate the segments results, such as riders position and points.
135     calculateResults();
136     // Then return the results for the requested rider.
137     return results.get(rider);
138 }
139
140 /**
141  * Method to remove a given riders results from the segment.
142  *
143  * @param rider The rider object whose results should be removed.
144  */
145 public void removeRiderResults(Rider rider) {
146     results.remove(rider);
147 }
148
149 /**
150  * Private function to sort all the riders who have results registered by their finish time.
151  * Useful for getting each riders position.
152  *
153  * @return All riders who have a registered result sorted by their finish time.
154  */
155 private List<Rider> sortRiderResults() {
156     // convert the hashmap into a set
157     return results.entrySet().stream()
158         // Sort the set by the finish time of the results
159         .sorted(Map.Entry.comparingByValue(SegmentResult.sortByFinishTime))
160         // Get the rider element of the set and ignore the results now they have been
161         // sorted and convert to a list.
162         .map(Map.Entry::getKey)
163         .collect(Collectors.toList());
164 }
165
166 /** Private method to calculate the results for this segment. */
167 private void calculateResults() {
168     // First get a list of riders sorted by their finish time.
169     List<Rider> riders = sortRiderResults();
170
171     for (int i = 0; i < results.size(); i++) {
172         Rider rider = riders.get(i);
173         SegmentResult result = results.get(rider);
174         int position = i + 1;
175         // Position Calculation
176         result.setPosition(position); // Set the riders position
177     }
178 }
```



```

177
178 // Points Calculation
179 int[] pointsDistribution =
180     getPointsDistribution(); // Get the point distribution based on the segment type.
181 if (position <= pointsDistribution.length) {
182     // Get the riders points based on their position
183     int points = pointsDistribution[i];
184     if (this.type.equals(SegmentType.SPRINT)) {
185         // If the segment is a sprint, set the riders points as sprinters points.
186         result.setSprintersPoints(points);
187         result.setMountainPoints(0);
188     } else {
189         // If the segment is not a sprint, set the riders points as mountain points.
190         result.setSprintersPoints(0);
191         result.setMountainPoints(points);
192     }
193 } else {
194     // If the rider does not finish in a point-awarding position, reward 0 points.
195     result.setMountainPoints(0);
196     result.setSprintersPoints(0);
197 }
198 }
199 }
200
201 /**
202  * Private method to get the point distribution of the segment based on the type of segment.
203  *
204  * @return an array of integers that represent the points that should be rewarded based on the
205  *         segment type.
206  */
207 private int[] getPointsDistribution() {
208     return switch (type) {
209         case HC -> HC_POINTS;
210         case C1 -> C1_POINTS;
211         case C2 -> C2_POINTS;
212         case C3 -> C3_POINTS;
213         case C4 -> C4_POINTS;
214         case SPRINT -> SPRINT_POINTS;
215     };
216 }
217 }

```

9 SegmentResult.java

```

1 package cycling;
2
3 import java.io.Serializable;
4 import java.time.LocalDateTime;
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 LocalDateTime 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  }
81  }

```

10 Stage.java

```

1  package cycling;
2
3  import java.io.Serializable;
4  import java.time.Duration;
5  import java.time.LocalDate;
6  import java.time.LocalDateTime;
7  import java.time.LocalTime;
8  import java.util.ArrayList;
9  import java.util.HashMap;
10 import java.util.List;
11 import java.util.Map;
12 import java.util.stream.Collectors;
13
14 /** Stage Class. This represents a stage in a race of the CyclingPortal */
15 public class Stage implements Serializable {
16     private final Race race;
17     private final String name;
18     private final String description;
19     private final double length;
20     private final LocalDateTime startTime;
21     private final StageType type;
22     private final int id;
23     private static int count = 0;
24     private boolean waitingForResults = false;
25     private final ArrayList<Segment> segments = new ArrayList<>();
26
27     private final HashMap<Rider, StageResult> results = new HashMap<>();
28
29     // Segment sprinters/mountain points.
30     private static final int[] FLAT_POINTS = {50, 30, 20, 18, 16, 14, 12, 10, 8, 7, 6, 5, 4, 3, 2};
31     private static final int[] MEDIUM_POINTS = {30, 25, 22, 19, 17, 15, 13, 11, 9, 7, 6, 5, 4, 3, 2};
32     private static final int[] HIGH_POINTS = {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
33     private static final int[] TT_POINTS = {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
34
35     /**
36      * Constructor method that sets a Stage up with a race, name, description, length startTime and
37      * type.
38      *
39      * @param race: Race that the Stage is in.
40      * @param name: name of the Stage, cannot be null, empty, have more than 30 characters or have
41      *             white space.
42      * @param description: description of the Stage.
43      * @param length: length of the Stage in kilometers, cannot be less than 5km.
44      * @param startTime: start time of the Stage.
45      * @param type: the type of Stage, can be either FLAT, MEDIUM_MOUNTAIN, HIGH_MOUNTAIN, TT.

```

```
45  * @throws InvalidNameException Thrown if the name is empty, null, longer than 30 characters or
46  *     contains whitespace.
47  * @throws InvalidLengthException Thrown if the length is less than 5km.
48  */
49  public Stage(
50      Race race,
51      String name,
52      String description,
53      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 }
```

```
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.LocalDateTime;
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
18      // result with the shortest time will come first.
19      protected static final Comparator<StageResult> sortByElapsedTime =
20          Comparator.comparing(StageResult::getElapsedTime);
21
22      /**
23       * Constructor for a given results in a stage.
24       *
25       * @param checkpoints The array of LocalTimes at which each checkpoint was crossed/
26       */
27      public StageResult(LocalTime[] checkpoints) {
28          this.checkpoints = checkpoints;
29          this.elapsedTime = Duration.between(checkpoints[0], checkpoints[checkpoints.length - 1]);
30      }
31
32      /**
33       * A method to get the times at which each checkpoint was crossed.
34       *
35       * @return The array of LocalTimes at which each checkpoint was crossed.
36       */
37      public LocalTime[] getCheckpoints() {
38          return this.checkpoints;
39      }
40
41      /**
42       * A method to get the elapsed time since the start of the stage.
43       *
44       * @return The duration of time since the stage started.
45       */
46      public Duration getElapsedTime() {
47          return elapsedTime;
48      }
49
50      /**
51       * A method to set the position in a stage.
52       *
53       * @param position the position of the rider in the stage.
54       */
55      public void setPosition(int position) {
56          this.position = position;
57      }
58
59      /**
60       * A method to set the adjusted elapsed time in a stage.
61       *
62       * @param adjustedElapsedTime the adjusted elapsed time in the stage.
63       */
64      public void setAdjustedElapsedTime(Duration adjustedElapsedTime) {
65          this.adjustedElapsedTime = adjustedElapsedTime;
66      }
```

```
67
68 /**
69  * A method to get the adjusted elapsed time in a stage as a duration.
70  *
71  * @return the adjusted elapsed time as a duration.
72  */
73 public Duration getAdjustedElapsedTime() {
74     return adjustedElapsedTime;
75 }
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 }
```