```java
package cycling;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.Arrays;
import java.util.Comparator;
import java.util.LinkedList;
import java.util.List;

/**
 * CyclingPortal is an implementor of the CyclingPortalInterface interface.
 *
 * @author James Pilcher
 * @author Daniel Moulton
 * @version 1.0
 *
 */
public class CyclingPortal implements CyclingPortalInterface {

  //Array of segment points to be awarded depending on position (and type)
  private static final Integer[] SPRINT_SEGMENT_POINTS =
    {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
  private static final Integer[] HC_SEGMENT_POINTS =
    {20, 15, 12, 10, 8, 6, 4, 2};
  private static final Integer[] C1_SEGMENT_POINTS =
    {10, 8, 6, 4, 2, 1};
  private static final Integer[] C2_SEGMENT_POINTS =
    {5, 3, 2, 1};
  private static final Integer[] C3_SEGMENT_POINTS =
    {2, 1};
  private static final Integer[] C4_SEGMENT_POINTS =
    {1};

  //Array of stage points to be awarded depending on position (and type)

  private static final Integer[] FLAT_STAGE_POINTS =
    {50, 30, 20, 18, 16, 14, 12, 10, 8, 7, 6, 5, 4, 3, 2};
  private static final Integer[] MM_STAGE_POINTS =
    {30, 25, 22, 19, 17, 15, 13, 11, 9, 7, 6, 5, 4, 3, 2};
  private static final Integer[] HM_STAGE_POINTS =
    {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5,  4, 3, 2, 1};
  private static final Integer[] TT_STAGE_POINTS =
    {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

  private LinkedList<Rider> riderList = new LinkedList<Rider>(); // List of riders

  private LinkedList<Team> teamList = new LinkedList<Team>(); // List of teams

  private LinkedList<Segment> segmentList = new LinkedList<Segment>();  // List of
segments

  private LinkedList<Stage> stageList = new LinkedList<Stage>(); // List of stages

```

```
59   private LinkedList<Race> raceList = new LinkedList<Race>(); // List of races
60
61   private LinkedList<RiderStageResults> riderStageResultsList =
62       new LinkedList<RiderStageResults>(); // List of rider stage results
63
64
65   /**
66    * Given an ID to search for, and a list of objects (i.e. riders) searches through
     the list
67    * and either returns the object with that ID, or null if no such object exists.
68    *
69    * @param id ID of object to find
70    * @param objectList List of objects to search through
71    * @return The object with that ID, or null if no such object exists
72    */
73   private <T extends IdHaver> T correspondingObjectFinder(int id, LinkedList<T>
     objectList,
74       String objectType)  throws IDNotRecognisedException {
75     T correspondingObject = null;
76     for (T object : objectList) {
77       if (id == object.getId()) {
78         correspondingObject = object;
79         break;
80       }
81     }
82     if (correspondingObject == null) {
83       throw new IDNotRecognisedException(objectType + " ID " + id
84         + " not recognised in the system.");
85     }
86     return correspondingObject;
87   }
88
89   /**
90    * Will throw an InvalidNameException, if the object name is null, empty,
91    * has more than 30 characters, or has whitespaces.
92    *
93    * @param name Proposed name of a team/stage/race
94    * @param objectType Denotes whether it is a team/stage/race.
95    */
96   private void validNameChecker(String name, String objectType) throws
     InvalidNameException {
97     if (name == null || name == "" || name.length() > 30 || name.contains(" ")) {
98       throw new InvalidNameException(objectType
99         +  " name is null, empty, has more than 30 characters, or has whitespaces");
100    }
101  }
102
103  /**
104   * Will throw an InvalidStageStateException if the stage is waiting for results.
105   *
106   * @param stageState String of the current stage state
107   */
108  private void validStageStateChecker(String stageState) throws
     InvalidStageStateException {
109    if (stageState == "waiting for results") {
110      throw new InvalidStageStateException("Stage preparation has been concluded.");
111    }
112  }
113
114  /**
```

```
115     * Goes through each RiderStageResult object associated with the rider and passes
    this to
116     * deleteRiderResult which will delete the results.
117     *
118     * @param rider Object of the rider who's results are being deleted.
119     */
120    private void deleteAllRiderResults(Rider rider) {
121      assert rider != null;
122      LinkedList<RiderStageResults> riderResultsList =
123          new LinkedList<RiderStageResults>(rider.getRiderResultsList());
124      for (RiderStageResults riderStageResults : riderResultsList) {
125        deleteRiderResult(riderStageResults);
126      }
127    }
128
129    /**
130     * Deletes a riderStageResults object, removes all references to it.
131     *
132     * @param riderStageResults rider result object to be deleted.
133     */
134    private void deleteRiderResult(RiderStageResults riderStageResults) {
135      riderStageResults.getStage().getRiderResultsList().remove(riderStageResults);
136      riderStageResults.getRider().getRiderResultsList().remove(riderStageResults);
137      riderStageResultsList.remove(riderStageResults);
138    }
139
140    /**
141     * Deletes all the results in a given stage.
142     *
143     * @param stage Stage for results to be deleted within.
144     */
145    private void deleteAllStageResults(Stage stage) {
146      LinkedList<RiderStageResults> riderResultsList
147          = new LinkedList<RiderStageResults>(stage.getRiderResultsList());
148      for (RiderStageResults riderStageResults : riderResultsList) {
149        deleteRiderResult(riderStageResults);
150      }
151    }
152
153    /**
154     * Deletes a Team, all references to it, and all of its riders.
155     *
156     * @param team Team to be deleted.
157     */
158    private void deleteTeam(Team team) {
159      assert team != null;
160      teamList.remove(team.getId());
161      LinkedList<Rider> riders = new LinkedList<Rider>(team.getRiders());
162      for (Rider rider : riders) {
163        deleteRider(rider, team);
164      }
165      team = null;
166    }
167
168    /**
169     * Deletes a Rider, all references to them, and all of their corresponding
    results.
170     *
171     * @param rider Rider to be deleted.
172     * @param team Team the rider belongs to.
```

```
173     */
174   private void deleteRider(Rider rider, Team team) {
175     assert rider != null;
176     riderList.remove(rider);
177     deleteAllRiderResults(rider);
178     team.removeRider(rider);
179   }
180
181   /**
182    * Deletes a race, all references to it, and all of its corresponding results.
183    *
184    * @param race Race to be deleted.
185    */
186   private void deleteRace(Race race) {
187     assert race != null;
188     raceList.remove(race);
189     LinkedList<Stage> raceStages = new LinkedList<Stage>(race.getStages());
190     for (Stage stage : raceStages) {
191       deleteStage(stage, race);
192     }
193     race = null;
194   }
195
196   /**
197    * Deletes a stage, all references to it, and all of its corresponding results.
198    *
199    * @param stage Stage to be deleted.
200    * @param race Race the stage belongs to.
201    */
202   private void deleteStage(Stage stage, Race race) {
203     assert stage != null;
204     stageList.remove(stage);
205     race.removeStage(stage);
206     LinkedList<Segment> stageSegments = new LinkedList<Segment>
(stage.getSegments());
207     for (Segment segment : stageSegments) {
208       deleteSegment(segment, stage);
209     }
210     deleteAllStageResults(stage);
211     stage = null;
212   }
213
214   /**
215    * Removes a segment from a given stage.
216    *
217    * @param segment Segment to be deleted.
218    * @param stage Stage the segment belongs to.
219    */
220   private void deleteSegment(Segment segment, Stage stage) {
221     segmentList.remove(segment);
222     stage.removeSegment(segment);
223     segment = null;
224   }
225
226   /**
227    * Sorts a list of riders results by their elapsed time attribute.
228    *
229    * @param competingRiders List of ridersResults in a stage.
230    */
```

```
231   private void sortRidersByElapsedTime(LinkedList<RiderStageResults>
competingRiders) {
232     competingRiders.sort(Comparator.comparing((RiderStageResults rider)
233         -> rider.getElapsedTimeForStage()));
234   }
235
236   /**
237    * Adjusts all riders times in a stage. If one finishes within one second of
another,
238    * their time is bumped to the lowest of the two. This cascades all the way down.
239    *
240    * @param competingRiders List of rider results in the stage.
241    */
242   private void adjustRiderTimesInStage(LinkedList<RiderStageResults>
competingRiders) {
243     sortRidersByElapsedTime(competingRiders);
244     competingRiders.get(0).setAdjustedTimeForStage(
245         competingRiders.get(0).getElapsedTimeForStage());
246
247     for (int i = 0; i < competingRiders.size() - 1; i++) {
248       if (competingRiders.get(i + 1).getElapsedTimeForStage()
249           - competingRiders.get(i).getElapsedTimeForStage() < 1000_000_000L) {
250         competingRiders.get(i + 1).setAdjustedTimeForStage(
251             competingRiders.get(i).getAdjustedTimeForStage());
252       } else {
253         competingRiders.get(i + 1).setAdjustedTimeForStage(
254             competingRiders.get(i + 1).getElapsedTimeForStage());
255       }
256     }
257   }
258
259
260   /**
261    * This function appends the 0's to the pointsToBeAdded array, depending on the
number of
262    * riders in a given stage/segment.
263    *
264    * @param numRiders number of riders in the stage/segment.
265    * @param rankPoints Array of points that index's match the position in a
segment/race,
266       and the points match the points awarded to those positions.
267    * @return number of points to add.
268    */
269   private LinkedList<Integer> pointsToBeAddedFormatter(int numRiders, Integer[]
rankPoints) {
270     int rankPointsSize = rankPoints.length;
271     LinkedList<Integer> pointsToBeAdded
272         = new LinkedList<Integer>(Arrays.asList(rankPoints));
273     if (numRiders > rankPointsSize) {
274       int sizeDifference = numRiders - rankPointsSize;
275       for (int i = 0; i < sizeDifference; i++) {
276         pointsToBeAdded.add(0);
277       }
278     }
279     return pointsToBeAdded;
280   }
281
282   /**
283    * Awards each rider in a segment their segment points, given their position and
the
```

```
284     * type of segment.
285     * Mountain or sprint segments are decided by the boolean variable isSprintSegment
286     *
287     * @param competingRiders List of riders who competed in the segment.
288     * @param segment The segment we want to award points within.
289     * @param stageSegments List of all segments in the stage the segment is in.
290     * @param segmentPointsToBeAdded List of segment points to be added,
291         the index corresponds to position
292     * @param isSprintSegment Do we want to award a sprint segment or a mountain
    segment?
293     */
294    private void awardSegmentPoints(LinkedList<RiderStageResults> competingRiders,
    Segment segment,
295        LinkedList<Segment> stageSegments, LinkedList<Integer> segmentPointsToBeAdded,
296        boolean isSprintSegment) {
297      int indexForSegment = stageSegments.indexOf(segment);
298
299      LinkedList<RiderStageResults> ridersInSegment
300          = new LinkedList<RiderStageResults>(competingRiders);
301      ridersInSegment.sort(Comparator.comparing((RiderStageResults rider)
302          -> rider.getSegmentTime(indexForSegment)));
303
304      for (RiderStageResults rider : ridersInSegment) {
305        int indexForPoints = ridersInSegment.indexOf(rider);
306        int points = segmentPointsToBeAdded.get(indexForPoints);
307        if (isSprintSegment) {
308          rider.addPoints(points);
309        } else {
310          rider.addMountainPoints(points);
311        }
312      }
313
314    }
315
316    /**
317     * Awards all the riders in a given stage their (sprint) points. Points are
    awarded to their
318     * Corresponding riderStageResults object.
319     *
320     * @param stage The stage to award (sprint) points within.
321     */
322    private void awardPointsInStage(Stage stage) {
323      LinkedList<RiderStageResults> riderResultsList
324          = new LinkedList<RiderStageResults>(stage.getRiderResultsList());
325
326      LinkedList<Integer> pointsToBeAdded = new LinkedList<Integer>();
327      sortRidersByElapsedTime(riderResultsList);
328
329      for (RiderStageResults riderStageResults : riderResultsList) {
330        riderStageResults.resetPoints();
331      }
332
333      int riderResultsListSize = riderResultsList.size();
334      StageType stageType = stage.getType();
335      switch (stageType) {
336        case FLAT:
337          pointsToBeAdded = pointsToBeAddedFormatter(riderResultsListSize,
    FLAT_STAGE_POINTS);
338          break;
339        case MEDIUM_MOUNTAIN:
```

```
340         pointsToBeAdded = pointsToBeAddedFormatter(riderResultsListSize,
MM_STAGE_POINTS);
341         break;
342       case HIGH_MOUNTAIN:
343         pointsToBeAdded = pointsToBeAddedFormatter(riderResultsListSize,
HM_STAGE_POINTS);
344         break;
345       case TT:
346         pointsToBeAdded = pointsToBeAddedFormatter(riderResultsListSize,
TT_STAGE_POINTS);
347         break;
348       default: assert false;
349     }
350
351     for (RiderStageResults riderStageResults : riderResultsList) {
352       int indexForPoints = riderResultsList.indexOf(riderStageResults);
353       int points = pointsToBeAdded.get(indexForPoints);
354       riderStageResults.setPoints(points);
355     }
356     LinkedList<Segment> segments = new LinkedList<Segment>(stage.getSegments());
357     LinkedList<Integer> segmentPointsToBeAdded = new LinkedList<Integer>();
358     for (Segment segment : segments) {
359       if (segment.getSegmentType() == SegmentType.SPRINT) {
360         segmentPointsToBeAdded
361             = pointsToBeAddedFormatter(riderResultsList.size(),
SPRINT_SEGMENT_POINTS);
362         awardSegmentPoints(riderResultsList, segment, segments,
segmentPointsToBeAdded, true);
363       }
364     }
365   }
366
367
368   /**
369    * Awards all the riders in a given stage their mountain points. Points are
awarded to their
370    * Corresponding riderStageResults object.
371    *
372    * @param stage The stage to award mountain points within.
373    */
374   private void awardMountainPointsInStage(Stage stage) {
375     LinkedList<RiderStageResults> riderResultsList
376         = new LinkedList<RiderStageResults>(stage.getRiderResultsList());
377     LinkedList<Integer> pointsToBeAdded = new LinkedList<Integer>();
378     sortRidersByElapsedTime(riderResultsList);
379
380     for (RiderStageResults riderStageResults : riderResultsList) {
381       riderStageResults.resetMountainPoints();
382     }
383
384     LinkedList<Segment> segments = new LinkedList<Segment>(stage.getSegments());
385
386     for (Segment segment : segments) {
387       SegmentType segmentType = segment.getSegmentType();
388       int riderResultsListSize = riderResultsList.size();
389       if (!(segmentType == SegmentType.SPRINT)) {
390         switch (segmentType) {
391           case C1:
392             pointsToBeAdded = pointsToBeAddedFormatter(riderResultsListSize,
C1_SEGMENT_POINTS);
```

```
393             break;
394           case C2:
395             pointsToBeAdded = pointsToBeAddedFormatter(riderResultsListSize,
C2_SEGMENT_POINTS);
396             break;
397           case C3:
398             pointsToBeAdded = pointsToBeAddedFormatter(riderResultsListSize,
C3_SEGMENT_POINTS);
399             break;
400           case C4:
401             pointsToBeAdded = pointsToBeAddedFormatter(riderResultsListSize,
C4_SEGMENT_POINTS);
402             break;
403           case HC:
404             pointsToBeAdded = pointsToBeAddedFormatter(riderResultsListSize,
HC_SEGMENT_POINTS);
405             break;
406           default: assert false;
407         }
408         awardSegmentPoints(riderResultsList, segment, segments, pointsToBeAdded,
false);
409       }
410     }
411   }
412
413   /**
414    * Awards every rider in a given race their total Points classification points.
415    *
416    * @param race The specified race to sum total (sprint) points within
417    * @return Returns the list of riders in the given race
418        now with their awarded total (sprint) points.
419    */
420   private LinkedList<Rider> totalRidersPoints(Race race) {
421     LinkedList<Rider> riders = new LinkedList<Rider>();
422     for (Rider rider : riderList) {
423       rider.resetTotalElapsedTime();
424       rider.resetTotalPoints();
425     }
426
427     for (Stage stage : race.getStages()) {
428       awardPointsInStage(stage);
429       for (RiderStageResults riderStageResults : stage.getRiderResultsList()) {
430         Rider rider = riderStageResults.getRider();
431         rider.addTotalElapsedTime(riderStageResults.getElapsedTimeForStage());
432         rider.addTotalPoints(riderStageResults.getRiderPoints());
433         if (!riders.contains(rider)) {
434           riders.add(rider);
435         }
436       }
437     }
438     return riders;
439   }
440
441   /**
442    * Awards every rider in a given race their total Mountain classification points.
443    *
444    * @param race The specified race to sum total mountain points within
445    * @return Returns the list of riders in the given race,
446        now with their total mountain points awarded
447    */
```

```
448   private LinkedList<Rider> totalRidersMountainPoints(Race race) {
449     LinkedList<Rider> riders = new LinkedList<Rider>();
450     for (Rider rider : riderList) {
451       rider.resetTotalElapsedTime();
452       rider.resetTotalMountainPoints();
453     }
454     for (Stage stage : race.getStages()) {
455       awardMountainPointsInStage(stage);
456       for (RiderStageResults riderStageResults : stage.getRiderResultsList()) {
457         Rider rider = riderStageResults.getRider();
458         rider.addTotalElapsedTime(riderStageResults.getElapsedTimeForStage());
459         rider.addTotalMountainPoints(riderStageResults.getRiderMountainPoints());
460         if (!riders.contains(rider)) {
461           riders.add(rider);
462         }
463       }
464     }
465     return riders;
466   }
467
468
469   /**
470    * Sorts riders by their total elapsed time.
471    *
472    * @param riders List of riders to be sorted.
473    */
474   private void sortByTotalElapsedTime(LinkedList<Rider> riders) {
475     riders.sort(Comparator.comparing((Rider rider) -> rider.getTotalElapsedTime()));
476   }
477
478   /**
479    * Sorts riders by their total adjusted time.
480    *
481    * @param riders List of riders to be sorted.
482    */
483   private void sortByTotalAdjustedTime(LinkedList<Rider> riders) {
484     riders.sort(Comparator.comparing((Rider rider) ->
rider.getTotalAdjustedTime()));
485   }
486
487   /**
488    * Adjusts all rider times within a specified race, and returns them.
489    *
490    * @param race The specified race.
491    * @return A list of riders who competed in the race, sorted by
492      their total adjusted time.
493    */
494   private LinkedList<Rider> ridersTotalAdjustedTime(Race race) {
495
496     LinkedList<Rider> riders = new LinkedList<Rider>();
497     for (Rider rider : riderList) {
498       rider.resetTotalAdjustedTime();
499     }
500     for (Stage stage : race.getStages()) {
501       LinkedList<RiderStageResults> riderResultsList = stage.getRiderResultsList();
502       if (riderResultsList.size() == 0) {
503         break;
504       }
505       adjustRiderTimesInStage(riderResultsList);
506       for (RiderStageResults riderStageResults : riderResultsList) {
```

```
507         Rider rider = riderStageResults.getRider();
508         rider.addTotalAdjustedTime(riderStageResults.getAdjustedTimeForStage());
509         if (!riders.contains(rider)) {
510           riders.add(rider);
511         }
512       }
513     }
514     sortByTotalAdjustedTime(riders);
515     return riders;
516   }
517
518   @Override
519   public int[] getRaceIds() {
520     List<Integer> raceIds = new LinkedList<Integer>();
521     for (Race race : raceList) {
522       raceIds.add(race.getId());
523     }
524     return raceIds.stream().mapToInt(i -> i).toArray();
525   }
526
527   @Override
528   public int createRace(String name, String description)
529       throws IllegalNameException, InvalidNameException {
530     validNameChecker(name, "Race");
531     for (Race race : raceList) {
532       if (name == race.getName()) {
533         throw new IllegalNameException("Race name already exists in the platform.");
534       }
535     }
536     Race newRace = new Race(name, description);
537     raceList.add(newRace);
538     return newRace.getId();
539   }
540
541   @Override
542   public String viewRaceDetails(int raceId) throws IDNotRecognisedException {
543     Race race = correspondingObjectFinder(raceId, raceList, "Race");
544     String details;
545     String name = race.getName();
546     String description = race.getDescription();
547     Integer numberOfStages = race.getStages().size();
548     Double totalLength = race.totalLength();
549
550     details = "Race ID: " + raceId + ", Race Name: " + name + ", Race Description: "
551         + description + ", Number of stages: " + numberOfStages + ", Total Length: "
+ totalLength;
552     return details;
553   }
554
555   @Override
556   public void removeRaceById(int raceId) throws IDNotRecognisedException {
557     Race race = correspondingObjectFinder(raceId, raceList, "Race");
558     deleteRace(race);
559   }
560
561   @Override
562   public int getNumberOfStages(int raceId) throws IDNotRecognisedException {
563     Race race = correspondingObjectFinder(raceId, raceList, "Race");
564     return race.getStages().size();
565   }
```

```
566
567   @Override
568   public int addStageToRace(int raceId, String stageName, String description, double
      length,
569       LocalDateTime startTime, StageType type) throws IDNotRecognisedException,
570       IllegalNameException, InvalidNameException, InvalidLengthException {
571     validNameChecker(stageName, "Stage");
572     if (length < 5D) {
573       throw new InvalidLengthException("Length is less than 5km");
574     }
575     for (Stage stage : stageList) {
576       if (stageName == stage.getStageName()) {
577         throw new IllegalNameException("Stage name already exists in the
      platform.");
578       }
579     }
580     Race race = correspondingObjectFinder(raceId, raceList, "Race");
581     Stage newStage = new Stage(raceId, stageName, description, length, startTime,
      type);
582     race.addStage(newStage);
583     stageList.add(newStage);
584     return newStage.getId();
585   }
586
587   @Override
588   public int[] getRaceStages(int raceId) throws IDNotRecognisedException {
589     Race race = correspondingObjectFinder(raceId, raceList, "Race");
590     List<Integer> stageIds = new LinkedList<Integer>();
591     for (Stage stage : race.getStages()) {
592       stageIds.add(stage.getId());
593     }
594     return stageIds.stream().mapToInt(i -> i).toArray();
595   }
596
597   @Override
598   public double getStageLength(int stageId) throws IDNotRecognisedException {
599     Stage stage = correspondingObjectFinder(stageId, stageList, "Stage");
600     return stage.getLength();
601   }
602
603   @Override
604   public void removeStageById(int stageId) throws IDNotRecognisedException {
605     Stage stage = correspondingObjectFinder(stageId, stageList, "Stage");
606     Race raceContainingStage = correspondingObjectFinder(stage.getRaceId(),
      raceList, "Race");
607     deleteStage(stage, raceContainingStage);
608   }
609
610   @Override
611   public int addCategorizedClimbToStage(int stageId, Double location, SegmentType
      type,
612       Double averageGradient, Double length) throws IDNotRecognisedException,
613       InvalidLocationException, InvalidStageStateException,
      InvalidStageTypeException {
614     Stage stage = correspondingObjectFinder(stageId, stageList, "Stage");
615     validStageStateChecker(stage.getStageState());
616     if (stage.getType() == StageType.TT) {
617       throw new InvalidStageTypeException("Time-trial stages cannot contain any
      segment.");
618     }
```

```
619     if (location > stage.getLength()) {
620       throw new InvalidLocationException("Segment location is out of bounds of the
    stage length.");
621     }
622
623     Segment newClimb = new ClimbSegment(stageId, type, location, averageGradient,
    length);
624     stage.addSegment(newClimb);
625     segmentList.add(newClimb);
626     return newClimb.getId();
627   }
628
629   @Override
630   public int addIntermediateSprintToStage(int stageId, double location)
631       throws IDNotRecognisedException, InvalidLocationException,
632       InvalidStageStateException, InvalidStageTypeException {
633     Stage stage = correspondingObjectFinder(stageId, stageList, "Stage");
634     validStageStateChecker(stage.getStageState());
635     if (stage.getType() == StageType.TT) {
636       throw new InvalidStageTypeException("Time-trial stages cannot contain any
    segment.");
637     }
638     if (location > stage.getLength()) {
639       throw new InvalidLocationException("Segment location is out of bounds of the
    stage length.");
640     }
641
642     Segment newSprint = new Segment(stageId, SegmentType.SPRINT, location);
643     stage.addSegment(newSprint);
644     segmentList.add(newSprint);
645     return newSprint.getId();
646   }
647
648   @Override
649   public void removeSegment(int segmentId) throws IDNotRecognisedException,
650       InvalidStageStateException {
651     Segment segment = correspondingObjectFinder(segmentId, segmentList, "Segment");
652     Stage stageContainingSegment = correspondingObjectFinder(segment.getStageId(),
653           stageList, "Stage");
654     validStageStateChecker(stageContainingSegment.getStageState());
655     deleteSegment(segment, stageContainingSegment);
656   }
657
658   @Override
659   public void concludeStagePreparation(int stageId) throws
660       IDNotRecognisedException, InvalidStageStateException {
661     Stage stage = correspondingObjectFinder(stageId, stageList, "Stage");
662     validStageStateChecker(stage.getStageState());
663     stage.concludeStageState();
664   }
665
666   @Override
667   public int[] getStageSegments(int stageId) throws IDNotRecognisedException {
668     Stage stage = correspondingObjectFinder(stageId, stageList, "Stage");
669     List<Integer> segmentIds = new LinkedList<Integer>();
670     for (Segment segment : stage.getSegments()) {
671       segmentIds.add(segment.getId());
672     }
673     return segmentIds.stream().mapToInt(i -> i).toArray();
674   }
```

```
675
676   @Override
677   public int createTeam(String name, String description)
678       throws IllegalNameException, InvalidNameException {
679     validNameChecker(name, "Team");
680     for (Team team : teamList) {
681       if (name == team.getTeamName()) {
682         throw new IllegalNameException("Team name already exists in the platform");
683       }
684     }
685     Team newTeam = new Team(name, description);
686     teamList.add(newTeam);
687     return newTeam.getId();
688   }
689
690   @Override
691   public void removeTeam(int teamId) throws IDNotRecognisedException {
692     Team team = correspondingObjectFinder(teamId, teamList, "Team");
693     deleteTeam(team);
694   }
695
696   @Override
697   public int[] getTeams() {
698     List<Integer> teamIds = new LinkedList<Integer>();
699     for (Team team : teamList) {
700       teamIds.add(team.getId());
701     }
702     return teamIds.stream().mapToInt(i -> i).toArray();
703   }
704
705   @Override
706   public int[] getTeamRiders(int teamId) throws IDNotRecognisedException {
707     List<Rider> ridersInTeam = new LinkedList<Rider>();
708     List<Integer> teamRidersIds = new LinkedList<Integer>();
709     Team team = correspondingObjectFinder(teamId, teamList, "Team");
710     ridersInTeam = team.getRiders();
711     for (Rider rider : ridersInTeam) {
712       teamRidersIds.add(rider.getId());
713     }
714     return teamRidersIds.stream().mapToInt(i -> i).toArray();
715   }
716
717   @Override
718   public int createRider(int teamId, String name, int yearOfBirth)
719       throws IDNotRecognisedException, IllegalArgumentException {
720     if (yearOfBirth < 1900 || name == null) {
721       throw new IllegalArgumentException(
722           "Name of rider is null or year of birth is less than 1900");
723     }
724     Team team = correspondingObjectFinder(teamId, teamList, "Team");
725     Rider newRider = new Rider(yearOfBirth, name, teamId);
726     riderList.add(newRider);
727     team.addRider(newRider);
728     return newRider.getId();
729   }
730
731   @Override
732   public void removeRider(int riderId) throws IDNotRecognisedException {
733     Rider rider = correspondingObjectFinder(riderId, riderList, "Rider");
```

```
734     Team teamContainingRider = correspondingObjectFinder(rider.getTeamId(),
    teamList, "Team");
735     deleteRider(rider, teamContainingRider);
736   }
737
738   @Override
739   public void registerRiderResultsInStage(int stageId, int riderId, LocalTime...
    checkpoints)
740       throws IDNotRecognisedException, DuplicatedResultException,
    InvalidCheckpointsException,
741       InvalidStageStateException {
742     Stage stage = correspondingObjectFinder(stageId, stageList, "Stage");
743
744     if (!(stage.getStageState() == "waiting for results")) {
745       throw new InvalidStageStateException("Stage has not concluded preparation.");
746     }
747
748     for (RiderStageResults riderStageResults : stage.getRiderResultsList()) {
749       if (riderStageResults.getRider().getId() == riderId) {
750         throw new DuplicatedResultException(
751           "Rider has a result for the stage. A rider can have only one result per
    stage.");
752       }
753     }
754
755     if (!(checkpoints.length == stage.getSegments().size() + 2)) {
756       throw new InvalidCheckpointsException(
757         "The number checkpoint times don't match the number of segments (+2)");
758     }
759
760     Rider rider = correspondingObjectFinder(riderId, riderList, "Rider");
761     RiderStageResults riderStageResults = new RiderStageResults(rider, stage,
    checkpoints);
762     stage.addRiderResultToStage(riderStageResults);
763     riderStageResultsList.add(riderStageResults);
764     rider.addStageResults(riderStageResults);
765   }
766
767
768   /**
769    * Converts a time in nanoseconds into the h/m/s/nanoseconds LocalTime format.
770    *
771    * @param nanoseconds The time in nanoseconds to be converted.
772    */
773   private LocalTime nanoToLocalTime(Long nanoseconds) {
774     assert nanoseconds != null;
775     int second = (int) (nanoseconds / 1000_000_000);
776     int minute = (int) (second / 60);
777     int hour = (int) (minute / 60);
778
779     nanoseconds %= 1000_000_000;
780     hour %= 60;
781     minute %= 60;
782     second %= 60;
783     LocalTime time = LocalTime.of(hour, minute, second, nanoseconds.intValue());
784     return time;
785   }
786
787   @Override
788   public LocalTime[] getRiderResultsInStage(int stageId, int riderId)
```

```
789       throws IDNotRecognisedException {
790     Stage stage = correspondingObjectFinder(stageId, stageList, "Stage");
791     Rider rider = correspondingObjectFinder(riderId, riderList, "Rider");
792     LinkedList<Long> times = new LinkedList<Long>();
793     LinkedList<LocalTime> results = new LinkedList<LocalTime>();
794
795     for (RiderStageResults riderStageResults : rider.getRiderResultsList()) {
796       if (riderStageResults.getStage() == stage) {
797         for (Long segmentTime : riderStageResults.getSegmentTimes()) {
798           times.add(segmentTime + riderStageResults.getStartTime());
799         }
800         times.add(riderStageResults.getElapsedTimeForStage());
801         break;
802       }
803     }
804     for (Long time : times) {
805       results.add(nanoToLocalTime(time));
806     }
807     return results.toArray(new LocalTime[times.size()]);
808   }
809
810   @Override
811   public LocalTime getRiderAdjustedElapsedTimeInStage(int stageId, int riderId)
812       throws IDNotRecognisedException {
813     Stage stage = correspondingObjectFinder(stageId, stageList, "Stage");
814
815     Rider rider = correspondingObjectFinder(riderId, riderList, "Rider");
816
817     LinkedList<RiderStageResults> riderResultsList = stage.getRiderResultsList();
818     adjustRiderTimesInStage(riderResultsList);
819     LocalTime adjustedTime = null;
820
821     for (RiderStageResults riderStageResults : riderResultsList) {
822       if (riderStageResults.getRider() == rider) {
823         adjustedTime = nanoToLocalTime(riderStageResults.getAdjustedTimeForStage());
824         break;
825       }
826     }
827     return adjustedTime;
828   }
829
830   @Override
831   public void deleteRiderResultsInStage(int stageId, int riderId) throws
   IDNotRecognisedException {
832
833     Stage stage = correspondingObjectFinder(stageId, stageList, "Stage");
834     Rider rider = correspondingObjectFinder(riderId, riderList, "Rider");
835
836     LinkedList<RiderStageResults> riderResultsList
837         = new LinkedList<RiderStageResults>(stage.getRiderResultsList());
838
839     for (RiderStageResults riderStageResults : riderResultsList) {
840       if (riderStageResults.getRider() == rider) {
841         deleteRiderResult(riderStageResults);
842         break;
843       }
844     }
845   }
846
847   @Override
```

```
848   public int[] getRidersRankInStage(int stageId) throws IDNotRecognisedException {
849     Stage stage = correspondingObjectFinder(stageId, stageList, "Stage");
850
851     LinkedList<RiderStageResults> riderResultsList = stage.getRiderResultsList();
852     int[] riderIds = new int[riderResultsList.size()];
853     sortRidersByElapsedTime(riderResultsList);
854
855     for (int i = 0; i < riderResultsList.size(); i++) {
856       riderIds[i] = riderResultsList.get(i).getRider().getId();
857     }
858     return riderIds;
859   }
860
861   @Override
862   public LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId)
863       throws IDNotRecognisedException {
864     Stage stage = correspondingObjectFinder(stageId, stageList, "Stage");
865
866     LinkedList<RiderStageResults> riderResultsList = stage.getRiderResultsList();
867     LocalTime[] localTimes = new LocalTime[riderResultsList.size()];
868     adjustRiderTimesInStage(riderResultsList);
869     for (int i = 0; i < riderResultsList.size(); i++) {
870       localTimes[i] =
nanoToLocalTime(riderResultsList.get(i).getAdjustedTimeForStage());
871     }
872     return localTimes;
873   }
874
875   @Override
876   public int[] getRidersPointsInStage(int stageId) throws IDNotRecognisedException {
877     Stage stage = correspondingObjectFinder(stageId, stageList, "Stage");
878
879     awardPointsInStage(stage);
880     LinkedList<RiderStageResults> riderResultsList = stage.getRiderResultsList();
881
882     int[] riderPoints = new int[riderResultsList.size()];
883     sortRidersByElapsedTime(riderResultsList);
884
885     for (int i = 0; i < riderResultsList.size(); i++) {
886       riderPoints[i] = riderResultsList.get(i).getRiderPoints();
887     }
888     return riderPoints;
889   }
890
891   @Override
892   public int[] getRidersMountainPointsInStage(int stageId) throws
IDNotRecognisedException {
893     Stage stage = correspondingObjectFinder(stageId, stageList, "Stage");
894
895     awardMountainPointsInStage(stage);
896     LinkedList<RiderStageResults> riderResultsList = stage.getRiderResultsList();
897
898     int[] riderMountainPoints = new int[riderResultsList.size()];
899     sortRidersByElapsedTime(riderResultsList);
900
901     for (int i = 0; i < riderResultsList.size(); i++) {
902       riderMountainPoints[i] = riderResultsList.get(i).getRiderMountainPoints();
903     }
904     return riderMountainPoints;
905   }
```

```
906
907   @Override
908   public void eraseCyclingPortal() {
909     Rider.resetIdCounter();
910     riderList.clear();
911
912     Team.resetIdCounter();
913     teamList.clear();
914
915
916     Race.resetIdCounter();
917     raceList.clear();
918
919     Stage.resetIdCounter();
920     stageList.clear();
921
922     Segment.resetIdCounter();
923     segmentList.clear();
924
925     riderStageResultsList.clear();
926   }
927
928   @Override
929   public void saveCyclingPortal(String filename) throws IOException {
930     if (!filename.endsWith(".ser")) {
931       filename += ".ser";
932     }
933     File file = new File(filename);
934     if (file.exists() && !file.isDirectory()) {
935       file.delete();
936     }
937     try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(filename))) {
938       oos.writeObject(riderList);
939       oos.writeObject(teamList);
940       oos.writeObject(raceList);
941       oos.writeObject(stageList);
942       oos.writeObject(segmentList);
943       oos.writeObject(riderStageResultsList);
944       System.out.printf("Saved in %s%n", filename);
945       oos.close();
946     } catch (IOException e) {
947       throw new IOException("Failed to save contents to file");
948     }
949   }
950
951   @Override
952     public void loadCyclingPortal(String filename) throws IOException,
ClassNotFoundException {
953     eraseCyclingPortal();
954     if (!filename.endsWith(".ser")) {
955       filename += ".ser";
956     }
957     try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(filename))) {
958       Object obj = ois.readObject();
959       if (obj instanceof LinkedList<?>) {
960         if (((LinkedList<?>) obj).get(0) instanceof Rider) {
961           riderList = (LinkedList<Rider>) obj;
962         }
```

```
963         }
964         obj = ois.readObject();
965         if (obj instanceof LinkedList<?>) {
966           if (((LinkedList<?>) obj).get(0) instanceof Team) {
967             teamList = (LinkedList<Team>) obj;
968           }
969         }
970         obj = ois.readObject();
971         if (obj instanceof LinkedList<?>) {
972           if (((LinkedList<?>) obj).get(0) instanceof Race) {
973             raceList = (LinkedList<Race>) obj;
974           }
975         }
976         obj = ois.readObject();
977         if (obj instanceof LinkedList<?>) {
978           if (((LinkedList<?>) obj).get(0) instanceof Stage) {
979             stageList = (LinkedList<Stage>) obj;
980           }
981         }
982         obj = ois.readObject();
983         if (obj instanceof LinkedList<?>) {
984           if (((LinkedList<?>) obj).get(0) instanceof Segment) {
985             segmentList = (LinkedList<Segment>) obj;
986           }
987         }
988         obj = ois.readObject();
989         if (obj instanceof LinkedList<?>) {
990           if (((LinkedList<?>) obj).get(0) instanceof RiderStageResults) {
991             riderStageResultsList = (LinkedList<RiderStageResults>) obj;
992           }
993         }
994       } catch (IOException e) {
995         throw new IOException("Failed to load contents from file");
996       } catch (ClassNotFoundException e) {
997         throw new ClassNotFoundException("Required class files not found");
998       }
999     }
1000
1001
1002    @Override
1003    public void removeRaceByName(String name) throws NameNotRecognisedException {
1004      Race race = null;
1005      for (Race raceElement : raceList) {
1006        if (raceElement.getName().equals(name)) {
1007          race = raceElement;
1008          break;
1009        }
1010      }
1011      if (race == null) {
1012        throw new NameNotRecognisedException(
1013          "The given Race name does not match to any race in the system");
1014      }
1015      deleteRace(race);
1016    }
1017
1018    @Override
1019    public LocalTime[] getGeneralClassificationTimesInRace(int raceId)
1020        throws IDNotRecognisedException {
1021      Race race = correspondingObjectFinder(raceId, raceList, "Race");
1022      LinkedList<Rider> riders = ridersTotalAdjustedTime(race);
```

```
1023      LocalTime[] localTimes = new LocalTime[riders.size()];
1024      for (int i = 0; i < riders.size(); i++) {
1025        localTimes[i] = nanoToLocalTime(riders.get(i).getTotalAdjustedTime());
1026      }
1027      return localTimes;
1028    }
1029
1030    @Override
1031    public int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException {
1032      Race race = correspondingObjectFinder(raceId, raceList, "Race");
1033      LinkedList<Rider> riders = totalRidersPoints(race);
1034      sortByTotalElapsedTime(riders);
1035      int[] riderPoints = new int[riders.size()];
1036      for (int i = 0; i < riders.size(); i++) {
1037        riderPoints[i] = riders.get(i).getTotalPoints();
1038      }
1039      return riderPoints;
1040    }
1041
1042    @Override
1043    public int[] getRidersMountainPointsInRace(int raceId) throws
IDNotRecognisedException {
1044      Race race = correspondingObjectFinder(raceId, raceList, "Race");
1045      LinkedList<Rider> riders = totalRidersMountainPoints(race);
1046      sortByTotalElapsedTime(riders);
1047      int[] riderMountainPoints = new int[riders.size()];
1048      for (int i = 0; i < riders.size(); i++) {
1049        riderMountainPoints[i] = riders.get(i).getTotalMountainPoints();
1050      }
1051      return riderMountainPoints;
1052    }
1053
1054    @Override
1055    public int[] getRidersGeneralClassificationRank(int raceId) throws
IDNotRecognisedException {
1056      Race race = correspondingObjectFinder(raceId, raceList, "Race");
1057
1058      LinkedList<Rider> riders = ridersTotalAdjustedTime(race);
1059
1060      int[] riderIds = new int[riders.size()];
1061
1062      for (int i = 0; i < riders.size(); i++) {
1063        riderIds[i] = riders.get(i).getId();
1064      }
1065      return riderIds;
1066    }
1067
1068    /**
1069     * Sorts the list of riders by their total (sprint) points.
1070     *
1071     * @param riders List of riders to be sorted.
1072     */
1073    private void sortByTotalPoints(LinkedList<Rider> riders) {
1074      riders.sort(Comparator.comparing((Rider rider) -> (rider.getTotalPoints() *
-1)));
1075    }
1076
1077    @Override
1078    public int[] getRidersPointClassificationRank(int raceId) throws
IDNotRecognisedException {
```

```
1079
1080      Race race = correspondingObjectFinder(raceId, raceList, "Race");
1081      LinkedList<Rider> riders = totalRidersPoints(race);
1082      sortByTotalPoints(riders);
1083      int[] riderPointsId = new int[riders.size()];
1084      for (int i = 0; i < riders.size(); i++) {
1085        riderPointsId[i] = riders.get(i).getId();
1086      }
1087      return riderPointsId;
1088    }
1089
1090    /**
1091     * Sorts the list of riders by their total mountain points.
1092     *
1093     * @param riders List of riders to be sorted.
1094     */
1095    private void sortByTotalMountainPoints(LinkedList<Rider> riders) {
1096      riders.sort(Comparator.comparing((Rider rider) ->
    (rider.getTotalMountainPoints() * -1)));
1097    }
1098
1099    @Override
1100    public int[] getRidersMountainPointClassificationRank(int raceId)
1101        throws IDNotRecognisedException {
1102      Race race = correspondingObjectFinder(raceId, raceList, "Race");
1103      LinkedList<Rider> riders = totalRidersMountainPoints(race);
1104      sortByTotalMountainPoints(riders);
1105      int[] riderMountainPointsId = new int[riders.size()];
1106      for (int i = 0; i < riders.size(); i++) {
1107        riderMountainPointsId[i] = riders.get(i).getId();
1108      }
1109      return riderMountainPointsId;
1110    }
1111  }
```

```java
package cycling;

/**
 * Contains attributes and methods to do with IDs, each class that needs an ID inherits this.
 *
 * @author James Pilcher
 * @author Daniel Moulton
 * @version 1.0
 */
public class IdHaver {
  private int id;


  /**
   * Gets ID of object.
   *
   * @return int
   */
  public int getId() {
    return id;
  }


  /**
   * Sets ID of object.
   *
   * @param id value to set ID to
   */
  public void setId(int id) {
    this.id = id;
  }
}
```

```java
package cycling;

import java.io.Serializable;
import java.util.LinkedList;

/**
 * Represents a race.
 *
 * @author James Pilcher
 * @author Daniel Moulton
 * @version 1.0
 */
public class Race extends IdHaver implements Serializable {
  private int id;
  private String name;
  private String description;

  private LinkedList<Stage> stages = new LinkedList<Stage>(); // A Linked List of stages
  private static int numberOfRaces = 0; // The number of the races in existence.

  /**
   * Constructor for the Race class.
   *
   * @param name the name of the race
   * @param description a description of the race
   */
  public Race(String name, String description) {
    this.name = name;
    this.description = description;
    id = ++numberOfRaces;
    super.setId(id);
  }

  /**
   * Adds a stage to this race.
   *
   * @param stage The stage object that has been added to this race
   */
  public void addStage(Stage stage) {
    stages.add(stage);
  }


  /**
   * Removes a stage from this race.
   *
   * @param stage The stage object that has been removed from this race
   */
  public void removeStage(Stage stage) {
    stages.remove(stage);
  }

  /**
   * Calculates and returns the total length of the race.
   *
   * @return The total length of this race, calculated by summing the
   *         length of each stage in the race.
   */
```

```
59    public double totalLength() {
60      double totLength = 0D;
61      for (Stage stage : stages) {
62        totLength += stage.getLength();
63      }
64      return totLength;
65    }
66
67    /**
68     * Sorts and returns all stages ordered by their location in the race.
69     *
70     * @return An LinkedList of stages sorted by their location in the race.
71     */
72    public LinkedList<Stage> getStages() {
73      //sort them here
74      stages.sort((o1, o2)
75          -> o1.getStartTime().compareTo(
76          o2.getStartTime()));
77      return stages;
78    }
79
80    /**
81     * Gets the name of the race.
82     *
83     * @return the name of the race
84     */
85    public String getName() {
86      return name;
87    }
88
89    /**
90     * Gets the description of the race.
91     *
92     * @return the description of the race
93     */
94    public String getDescription() {
95      return description;
96    }
97
98    /**
99     * Resets the number of races, used when erasing the cycling portal to reset to an
    empty state.
100    */
101   public static void resetIdCounter() {
102     numberOfRaces = 0;
103   }
104 }
```

```java
package cycling;

import java.io.Serializable;
import java.time.LocalDateTime;
import java.util.LinkedList;

/**
 * Represents a stage within a race.
 *
 * @author James Pilcher
 * @author Daniel Moulton
 */
public class Stage extends IdHaver implements Serializable {
  private int id;
  private int raceId; // ID of race the stage belongs to
  private StageType type;
  private String stageName;
  private String description;
  private String stageState; // Stage state i.e., waiting for results
  private Double length;
  private LocalDateTime startTime;

  private LinkedList<Segment> segments = new LinkedList<Segment>(); /* Linked list of segments
  within this stage*/

  private LinkedList<RiderStageResults> riderResultsList =
      new LinkedList<RiderStageResults>(); // Linked list of rider results in stage

  private static int numberOfStages = 0; // Number of stages in portal.

  /**
   * Constructor for the Stage class.
   *
   * @param raceId ID of the race the stage belongs to.
   * @param stageName Name of the stage
   * @param description Description of the stage
   * @param length Length of the stage in kilometers
   * @param startTime Date and time of the start of the stage
   * @param type The type of the stage (determines how many points winners get)
   */
  public Stage(int raceId, String stageName, String description, double length,
      LocalDateTime startTime, StageType type) {
    this.raceId = raceId;
    this.stageName = stageName;
    this.description = description;
    this.length = length;
    this.startTime = startTime;
    this.type = type;
    id = ++numberOfStages;
    stageState = "in preparation";
    super.setId(id);
  }

  /**
   * Adds rider's results in stage to riderResultsList.
   *
   * @param rider Rider object of the corresponding rider
   */
```

```
59    public void addRiderResultToStage(RiderStageResults rider) {
60      riderResultsList.add(rider);
61    }
62
63    /**
64     * Adds a segment to the stage (adds segment object to stage's list of segments).
65     *
66     * @param segment Segment object to be added.
67     */
68    public void addSegment(Segment segment) {
69      segments.add(segment);
70    }
71
72
73    /**
74     * Removes specified segment object from stage's list of segments.
75     *
76     * @param segment Segment object to be removed.
77     */
78    public void removeSegment(Segment segment) {
79      segments.remove(segment);
80    }
81
82    /**
83     * Removes all segments from the stage's list of segments.
84     */
85    public void deleteSegments() {
86      segments.clear();
87    }
88
89    /**
90     * Sorts the list of segments by their position within the stage and returns the
   sorted list.
91     *
92     * @return List of segments ordered by their location/poistion in the stage.
93     */
94    public LinkedList<Segment> getSegments() {
95      segments.sort((o1, o2)
96          -> o1.getLocation().compareTo(
97          o2.getLocation()));
98      return segments;
99    }
100
101   /**
102    * Gets the ID of the race the stage belongs to.
103    *
104    * @return race ID
105    */
106   public int getRaceId() {
107     return raceId;
108   }
109
110   /**
111    * Gets the type of the stage i.e. FLAT.
112    *
113    * @return stage type
114    */
115   public StageType getType() {
116     return type;
117   }
```

```
118
119   /**
120    * Gets the name of the stage.
121    *
122    * @return stage name
123    */
124   public String getStageName() {
125     return stageName;
126   }
127
128   /**
129    * Gets the description of the stage.
130    *
131    * @return stage description
132    */
133   public String getDescription() {
134     return description;
135   }
136
137   /**
138    * Gets the length of the stage.
139    *
140    * @return stage length (in kilometers)
141    */
142   public double getLength() {
143     return length;
144   }
145
146   /**
147    * Gets the starting time of the stage.
148    *
149    * @return date and time the stage starts
150    */
151   public LocalDateTime getStartTime() {
152     return startTime;
153   }
154
155   /**
156    * Gets the current state of the stage i.e. waiting for results.
157    *
158    * @return stage state
159    */
160   public String getStageState() {
161     return stageState;
162   }
163
164   /**
165    * Gets list of riders results in this stage.
166    *
167    * @return riders results in this stage.
168    */
169   public LinkedList<RiderStageResults> getRiderResultsList() {
170     return riderResultsList;
171   }
172
173   /**
174    * Concludes the stage preparation.
175    */
176   public void concludeStageState() {
177     this.stageState = "waiting for results";
```

```
178   }
179
180   /**
181    * Reset the number of stages, used when erasing the cycling portal.
182    */
183   public static void resetIdCounter() {
184     numberOfStages = 0;
185   }
186 }
```

```java
package cycling;

import java.io.Serializable;

/**
 * Represents a segment.
 *
 * @author James Pilcher
 * @author Daniel Moulton
 */
public class Segment extends IdHaver implements Serializable {
  private int id;
  private int stageId;
  private Double location;
  private SegmentType type;

  private static int numberOfSegments = 0;

  /**
   * Constructor for segment class.
   *
   * @param stageId ID of the stage the segment is a part of
   * @param type The type of segment i.e. Sprint or HC
   * @param location Where in the stage the segment occurs (in kilometers)
   */
  public Segment(int stageId, SegmentType type, double location) {
    this.stageId = stageId;
    this.type = type;
    this.location = location;
    id = ++numberOfSegments;
    super.setId(id);
  }

  public Double getLocation() {
    return location;
  }

  public SegmentType getSegmentType() {
    return type;
  }

  public int getStageId() {
    return stageId;
  }

  /**
   * Resets the number of segments to 0, use when erasing cycling portal.
   */
  public static void resetIdCounter() {
    numberOfSegments = 0;
  }
}
```

```java
package cycling;

/**
 * Represents a categorised climb.
 *
 * @author James Pilcher
 * @author Daniel Moulton
 * @version 1.0
 */
public class ClimbSegment extends Segment {

  private double averageGradient;
  private double length;

  /**
   * Constructor of ClimbSegment class.
   *
   * @param stageId ID of the stage the segment belongs to.
   * @param type the type of segment i.e. Sprint, HC.
   * @param location where within the stage the climb is (in kilometers).
   * @param averageGradient Average gradient of the climb.
   * @param length How long the climb is (in kilometers).
   */
  public ClimbSegment(int stageId, SegmentType type, double location,
      double averageGradient, double length) {
    super(stageId, type, location);
    this.averageGradient = averageGradient;
    this.length = length;
  }
}
```

```java
package cycling;

import java.io.Serializable;
import java.util.LinkedList;

/**
 * Represents a team.
 *
 * @author James Pilcher
 * @author Daniel Moulton
 * @version 1.0
 */
public class Team extends IdHaver implements Serializable {

  private int id;
  private String teamName;
  private String teamDescription;

  private LinkedList<Rider> riders =
      new LinkedList<Rider>(); // Linked list of all riders belonging to the team

  private static int numberOfTeams = 0; // number of teams in the portal

  /**
   * Constructor for team class.
   *
   * @param teamName Name of the new team
   * @param teamDescription Description of the new team
   */
  public Team(String teamName, String teamDescription) {
    this.teamName = teamName;
    this.teamDescription = teamDescription;
    id = ++numberOfTeams;
    super.setId(id);
  }

  /**
   * Adds rider object to team's list of riders.
   *
   * @param rider The rider object to be removed.
   */
  public void addRider(Rider rider) {
    riders.add(rider);
  }

  /**
   * Removes a rider object from the team's list of riders.
   *
   * @param rider Rider object to be removed.
   */
  public void removeRider(Rider rider) {
    riders.remove(rider);
  }

  /**
   * Gets a list of all riders belonging to the team.
   *
   * @return list of riders
   */
```

```
60    public LinkedList<Rider> getRiders() {
61      return riders;
62    }
63
64    /**
65     * Gets the name of the team.
66     *
67     * @return the team name
68     */
69    public String getTeamName() {
70      return teamName;
71    }
72
73    /**
74     * Reset the number of teams, used when erasing the portal.
75     */
76    public static void resetIdCounter() {
77      numberOfTeams = 0;
78    }
79  }
```

```java
package cycling;

import java.io.Serializable;
import java.util.LinkedList;

/**
 * Represents a rider.
 *
 * @author James Pilcher
 * @author Daniel Moulton
 * @version 1.0
 */
public class Rider extends IdHaver implements Serializable {

  private int id;
  private String riderName;
  private int riderYearOfBirth;
  private int teamId;

  private Long totalAdjustedTime = 0L; /* Total time elapsed, adjusted for if a rider finishes
                                          within a second of the rider ahead */
  private Long totalElapsedTime = 0L; // Total time elapsed


  private int totalPoints = 0; // Total points rider has accumulated
  private int totalMountainPoints = 0; // Total Mountain points rider has accumulated

  private static int numberofRiders = 0; // Number of riders in the portal.

  private LinkedList<RiderStageResults> riderResultsList =
      new LinkedList<RiderStageResults>();   /* Linked list of rider's results in all stages */


  /**
   * Constructor for the rider class.
   *
   * @param riderYearOfBirth Year of birth of the rider
   * @param riderName Name of the rider
   * @param teamId ID of the team the rider belongs to
   */
  public Rider(int riderYearOfBirth, String riderName, int teamId) {
    this.riderYearOfBirth = riderYearOfBirth;
    this.riderName = riderName;
    this.teamId = teamId;
    id = ++numberofRiders;
    super.setId(id);
  }

  /**
   * Adds the rider's StageResult object for the corresponding stage to a LinkedList of all
   * the rider's stage results.
   *
   * @param rider Instace of the RiderStageResults object to be added to the LinkedList.
   */
  public void addStageResults(RiderStageResults rider) {
```

```
56      riderResultsList.add(rider);
57    }
58
59    /**
60     * Adds earnt points to the rider's total points.
61     *
62     * @param points The number of points to be added to the total points
63     */
64    public void addTotalPoints(int points) {
65      totalPoints += points;
66    }
67
68    /**
69     * Adds earnt mountain points to the rider's total mountain points.
70     *
71     * @param points The number of mountain points to be added to the total mountain
   points.
72     */
73    public void addTotalMountainPoints(int points) {
74      totalMountainPoints += points;
75    }
76
77    /**
78     * Adds adjusted time taken in specific stage for rider.
79     *
80     * @param time The time taken in specific stage for rider.
81     */
82    public void addTotalAdjustedTime(Long time) {
83      totalAdjustedTime += time;
84    }
85
86    /**
87     * Adds time taken in specific stage for rider.
88     *
89     * @param time The time taken in specific stage for rider.
90     */
91    public void addTotalElapsedTime(Long time) {
92      totalElapsedTime += time;
93    }
94
95
96    /**
97     * Gets ID of the team the rider belongs to.
98     *
99     * @return The ID of the team the rider belongs to.
100    */
101   public int getTeamId() {
102     return teamId;
103   }
104
105   /**
106    * Gets total time taken by rider.
107    *
108    * @return total time taken by rider
109    */
110   public Long getTotalElapsedTime() {
111     return totalElapsedTime;
112   }
113
114   /**
```

```
115     * Gets total adjusted time taken by rider.
116     *
117     * @return total adjusted time taken by rider
118     */
119    public Long getTotalAdjustedTime() {
120      return totalAdjustedTime;
121    }
122
123    /**
124     * Gets all the RiderStageResults objects belonging to the rider.
125     *
126     * @return LinkedList of all RiderStageResults objects belonging to the rider
127     */
128    public LinkedList<RiderStageResults> getRiderResultsList() {
129      return riderResultsList;
130    }
131
132    /**
133     * Resets the total adjusted time for the rider to 0.
134     */
135    public void resetTotalAdjustedTime() {
136      totalAdjustedTime = 0L;
137    }
138
139    /**
140     * Resets the total elapsed time for the rider to 0.
141     */
142    public void resetTotalElapsedTime() {
143      totalElapsedTime = 0L;
144    }
145
146    /**
147     * Reset the total points for the rider to 0.
148     */
149    public void resetTotalPoints() {
150      totalPoints = 0;
151    }
152
153    /**
154     * Reset the total mountain points for the rider to 0.
155     */
156    public void resetTotalMountainPoints() {
157      totalMountainPoints = 0;
158    }
159
160    /**
161     * Gets total points for rider.
162     *
163     * @return Total points for the rider
164     */
165    public int getTotalPoints() {
166      return totalPoints;
167    }
168
169    /**
170     * Gets total mountain points for rider.
171     *
172     * @return total mountain points for rider
173     */
174    public int getTotalMountainPoints() {
```

```
175     return totalMountainPoints;
176   }
177
178   /**
179    * Reset the number of riders, used when erasing the cycling portal to reset to an
   empty state.
180    */
181   public static void resetIdCounter() {
182     numberofRiders = 0;
183   }
184 }
```

```java
package cycling;

import java.io.Serializable;
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;
import java.util.LinkedList;


/**
 * Stores a single rider's results in a single stage.
 *
 * @author James Pilcher
 * @author Daniel Moulton
 * @version 1.0
 */
public class RiderStageResults implements Serializable {
  private Rider rider;
  private Stage stage;
  private Long startTime;

  private Long elapsedTimeForStage; // Time taken for the stage
  private Long adjustedTimeForStage; /* Time taken for the stage, adjusted for if a rider finishes
                              within a second of the rider ahead */
  private LinkedList<Long> segmentTimes
      = new LinkedList<Long>(); // List of rider's times in each segment

  private int riderPoints = 0;
  private int riderMountainPoints = 0;

  /**
   * Constructor for the RiderStageResults class.
   *
   * @param rider the rider the results relate to
   * @param stage the stage the results relate to
   * @param checkpoints time at each checkpoint in the stage for the rider
   */
  public RiderStageResults(Rider rider, Stage stage, LocalTime... checkpoints) {
    this.rider = rider;
    this.stage = stage;
    startTime = checkpoints[0].toNanoOfDay();
    elapsedTimeForStage
        = checkpoints[0].until(checkpoints[checkpoints.length - 1], ChronoUnit.NANOS);
    for (int i = 1; i < checkpoints.length - 1; i++) {
      segmentTimes.add(checkpoints[0].until(checkpoints[i], ChronoUnit.NANOS));
    }
  }

  /**
   *  Reset rider's points to 0.
   */
  public void resetPoints() {
    riderPoints = 0;
  }


  /**
   *  Reset rider's mountain points to 0.
```

```
58     */
59    public void resetMountainPoints() {
60      riderMountainPoints = 0;
61    }
62
63    /**
64     * Set rider's points to specified value.
65     *
66     * @param points value to set points to
67     */
68    public void setPoints(int points) {
69      riderPoints = points;
70    }
71
72    /**
73     * Increase rider's points by specified value.
74     *
75     * @param points value to increase rider's points by
76     */
77    public void addPoints(int points) {
78      riderPoints += points;
79    }
80
81    /**
82     * Increase rider's mountain points by specified value.
83     *
84     * @param points value to increase rider's mountain points by'
85     */
86    public void addMountainPoints(int points) {
87      riderMountainPoints += points;
88    }
89
90    /**
91     * Gets an arraylist of all the rider's segment times.
92     *
93     * @return arraylist of all the rider's segment times'
94     */
95    public LinkedList<Long> getSegmentTimes() {
96      return segmentTimes;
97    }
98
99    /**
100    * Gets the time of a rider in a certain segment.
101    *
102    * @param index index in the array of segment times
103    * @return rider's time in specified segment
104    */
105   public Long getSegmentTime(int index) {
106     return segmentTimes.get(index);
107   }
108
109   /**
110    * Gets the total time of the rider in the stage.
111    *
112    * @return total time of the rider in the stage
113    */
114   public Long getElapsedTimeForStage() {
115     return elapsedTimeForStage;
116   }
117
```

```
118   /**
119    * Gets rider's total points.
120    *
121    * @return rider's points
122    */
123   public int getRiderPoints() {
124     return riderPoints;
125   }
126
127   /**
128    * Gets rider's total mountain points.
129   *
130    * @return rider's mountain points
131    */
132   public int getRiderMountainPoints() {
133     return riderMountainPoints;
134   }
135
136
137   /**
138    * Gets the object of the stage.
139    *
140    * @return the stage object
141    */
142   public Stage getStage() {
143     return stage;
144   }
145
146   /**
147    * Gets the object of the rider.
148    *
149    * @return the rider object
150    */
151   public Rider getRider() {
152     return rider;
153   }
154
155   /**
156    * Gets the start time of the stage.
157    *
158    * @return stage's start time
159    */
160   public Long getStartTime() {
161     return startTime;
162   }
163
164   /**
165    * Gets rider's adjusted time for stage, adjusted for if
166    * a rider finishes within a second of the rider ahead.
167    *
168    * @return Rider's adjusted time
169    */
170   public Long getAdjustedTimeForStage() {
171     return adjustedTimeForStage;
172   }
173
174   /**
175    * Sets rider's adjusted time for this stage, adjusted for
176    * if a rider finishes within a second of the rider ahead.
177    *
```

```
178     * @param adjustedTimeForStage the new adjusted time for the stage
179     */
180   public void setAdjustedTimeForStage(Long adjustedTimeForStage) {
181     this.adjustedTimeForStage = adjustedTimeForStage;
182   }
183 }
```