

1 CyclingPortal.java

```
1 package cycling;
2
3 import java.io.FileInputStream;
4 import java.io.FileOutputStream;
5 import java.io.IOException;
6 import java.io.InvalidClassException;
7 import java.io.ObjectInputStream;
8 import java.io.ObjectOutputStream;
9 import java.time.LocalDateTime;
10 import java.time.LocalTime;
11 import java.util.ArrayList;
12 import java.util.Arrays;
13 import java.util.Comparator;
14 import java.util.HashMap;
15 import java.util.Map;
16
17 /**
18  * Cycling Portal implements CyclingPortalInterface class
19  *
20  * @author Ethan Hofton
21  * @author Jon Tao
22  * @version 1.0
23  */
24 public class CyclingPortal implements CyclingPortalInterface {
25
26     private ArrayList<Team> teams;
27     private ArrayList<Race> races;
28
29     /**
30      * CyclingPortal constructor initialises teams and races array list
31      */
32     public CyclingPortal() {
33         // constructor to init lists
34         teams = new ArrayList<>();
35         races = new ArrayList<>();
36     }
37
38     private Team findTeam(int teamID) throws IDNotRecognisedException {
39         // check if the list 'teams' has teamID
40         // O(n)
41
42         // loop througuh teams list and cheack the team class's id
43         // against the given id teamID
44         for (int i = 0; i < teams.size(); i++) {
45             if (teams.get(i).getTeamId() == teamID) {
46                 return teams.get(i);
47             }
48         }
49
50         // throw IDNotRecognisedException if not found
51         throw new IDNotRecognisedException("Team Id '" + teamID + "' not found");
52     }
53 }
```

```

53
54 private Rider findRider(int riderID) throws IDNotRecognisedException {
55     // check if the list 'teams' has teamID
56
57     // loop through each team and check if any of the riders on that team
58     // match the given rider id
59     for (int i = 0; i < teams.size(); i++) {
60         for (int j = 0; j < teams.get(i).getRiders().size(); j++) {
61             if (teams.get(i).getRiders().get(j).getRiderId() == riderID) {
62                 return teams.get(i).getRiders().get(j);
63             }
64         }
65     }
66
67     // throw IDNotRecognisedException if not found
68     throw new IDNotRecognisedException("Rider Id '" + riderID + "' not found");
69 }
70
71 private Race findRace(int raceID) throws IDNotRecognisedException {
72     // check if the list 'races' has raceID
73
74     // loop through races list and check given raceID
75     // against the race objects id
76     for (int i = 0; i < races.size(); i++) {
77         if (races.get(i).getRaceId() == raceID) {
78             return races.get(i);
79         }
80     }
81
82     // throw IDNotRecognisedException if not found
83     throw new IDNotRecognisedException("Race Id '" + raceID + "' not found");
84 }
85
86 private Stage findStage(int stageId) throws IDNotRecognisedException {
87     // check if the list 'races' has stageId
88
89     // loop though each race and loop through each races' stages
90     // if stage matches given id, return the stage
91     for (int i = 0; i < races.size(); i++) {
92         for (int j = 0; j < races.get(i).getStages().size(); j++) {
93             if (races.get(i).getStages().get(j).getStageId() == stageId) {
94                 return races.get(i).getStages().get(j);
95             }
96         }
97     }
98
99     throw new IDNotRecognisedException("Stage Id '" + stageId + "' not found");
100 }
101
102 private Segment findSegment(int segmentId) throws IDNotRecognisedException {
103     // check if the list 'races' has Segment with id segmentId
104
105     // loop through each races stages' segments
106     // if the segment id matches the given id, return that segment
107     for (int i = 0; i < races.size(); i++) {

```

```

108         Race currentRace = races.get(i);
109         for (int j = 0; j < currentRace.getStages().size(); j++) {
110             Stage currentStage = currentRace.getStages().get(j);
111             for (int m = 0; m < currentStage.getSegments().size(); m++) {
112                 Segment currentSegment = currentStage.getSegments().get(m);
113                 if (currentSegment.getSegmentId() == segmentId) {
114                     return currentSegment;
115                 }
116             }
117         }
118     }
119
120     throw new IDNotRecognisedException("Segment Id '" + segmentId + "' not found");
121 }
122
123 /**
124  * {@inheritDoc}
125  */
126 @Override
127 public int[] getRaceIds() {
128
129     // loop thorough each race in race list and add races id
130     // to a list of ids, return this list
131     int raceIds[] = new int[races.size()];
132     for (int i = 0; i < races.size(); i++) {
133         raceIds[i] = races.get(i).getRaceId();
134     }
135
136     return raceIds;
137 }
138
139 /**
140  * {@inheritDoc}
141  */
142 @Override
143 public int createRace(String name, String description) throws IllegalNameException,
144     InvalidNameException {
145
146     // errorus arguments checking
147     // check if the name is null, empty, contains whitespace or is longer the 30 charicters
148     if (name == null || name.equals("") || name.length() > 30 || name.contains(" ")) {
149         // throw an error if name does not meet these paramiters
150         throw new InvalidNameException("name cannot be null, empty, have more than 30 characters or
151             contain white spaces");
152     }
153
154     // check if the name allready exists in the platform
155     // loop through each race and check if the races name matches the given input name
156     for (int i = 0; i < races.size(); i++) {
157         if (name.equals(races.get(i).getName())) {
158             // theow exception if the name allreadt exists on platform
159             throw new IllegalNameException("name alrwdy exists in platform");
160         }
161     }
162 }

```

```

161     // create a new race
162     Race race = new Race(name, description);
163
164     // add the race to the cycling portals array list of races
165     races.add(race);
166
167     // return the race id
168     return race.getRaceId();
169 }
170
171 /**
172  * {@inheritDoc}
173  */
174 @Override
175 public String viewRaceDetails(int raceId) throws IDNotRecognisedException {
176
177     // find the race object in the system
178     // throws IDNotRecognisedException if the id does not exist on the platform
179     Race race = findRace(raceId);
180
181     // find the total length
182     // init total length to zero
183     double totalLen = 0.0;
184
185     // loop through each stage in the race and add the stage length to the total length
186     for (Stage stage : race.getStages()) {
187         totalLen += stage.getLength();
188     }
189
190     // stringify race details using race peramiters
191     String raceDetails = "raceID="+raceId;
192     raceDetails += ",name="+race.getName();
193     raceDetails += ",description="+race.getDescription();
194     raceDetails += ",numberOfStages="+race.getStages().size();
195     raceDetails += ",totalLength="+totalLen;
196
197     // return the stringified race detials
198     return raceDetails;
199 }
200
201 /**
202  * {@inheritDoc}
203  */
204 @Override
205 public void removeRaceById(int raceId) throws IDNotRecognisedException {
206     // find the race class in the portal
207     Race raceToRemove = findRace(raceId);
208
209     // removing race from the system also removes all related data
210     // since the race itself is the only thing that holds references to those
211     // related data classes
212     // remove the race class from the races array list
213     races.remove(raceToRemove);
214 }
215

```

```

216  /**
217   * {@inheritDoc}
218   */
219  @Override
220  public int getNumberOfStages(int raceId) throws IDNotRecognisedException {
221      // find the race within the portal
222      Race race = findRace(raceId);
223
224      // return the size of the array that stores the stages
225      return race.getStages().size();
226  }
227
228  /**
229   * {@inheritDoc}
230   */
231  @Override
232  public int addStageToRace(int raceId, String stageName, String description, double length,
233                          LocalDateTime startTime,
234                          StageType type)
235                          throws IDNotRecognisedException, IllegalNameException, InvalidNameException,
236                          InvalidLengthException {
237
238      // find race in portal
239      Race race = findRace(raceId);
240
241      // loop throug all the stages in the race
242      for (int i = 0; i < race.getStages().size(); i++) {
243          // check if the name allready exists in the race
244          // compare each stage name to the new stage name
245          if (race.getStages().get(i).getStageName().equals(stageName)) {
246              // if stage name allready excists throw an IllegalNameException
247              throw new IllegalNameException("name already exists on platform");
248          }
249      }
250
251      // check if the stage name is null, empty or grater than 30 charicters
252      if (stageName == null || stageName.equals("") || stageName.length() > 30) {
253          // throw InvalidNameException if paramaters are met
254          throw new InvalidNameException("Name cannot be null, empty or more than 30 characters");
255      }
256
257      // check if the stage length is less then 5km
258      if (length < 5) {
259          // throw InvalidLengthException
260          throw new InvalidLengthException("Length cannot be less than 5km");
261      }
262
263      // create the new stage
264      Stage stage = new Stage(race, stageName, description, length, startTime, type);
265
266      // add the stage to the race
267      race.addStage(stage);
268
269      // return the stage id
270      return stage.getStageId();

```

```

269 }
270
271 /**
272  * {@inheritDoc}
273  */
274 @Override
275 public int[] getRaceStages(int raceId) throws IDNotRecognisedException {
276     // find the race in the portal
277     Race race = findRace(raceId);
278
279     // initialise stage id list to return
280     // set array to the size of the number of stages for that stage
281     int stageIds[] = new int[race.getStages().size()];
282
283     // loop through all the stages in the race
284     for (int i = 0; i < stageIds.length; i++) {
285         // set each value of the array to the corresponding stage id
286         stageIds[i] = race.getStages().get(i).getStageId();
287     }
288
289     // return the list of stage ids
290     return stageIds;
291 }
292
293 /**
294  * {@inheritDoc}
295  */
296 @Override
297 public double getStageLength(int stageId) throws IDNotRecognisedException {
298     // find the stage in the system
299     Stage stage = findStage(stageId);
300
301     // return the length of the stage
302     return stage.getLength();
303 }
304
305 /**
306  * {@inheritDoc}
307  */
308 @Override
309 public void removeStageById(int stageId) throws IDNotRecognisedException {
310     // find the stage in the portal
311     Stage stage = findStage(stageId);
312
313     // removing the stage also removes all stage related data
314     // this is because the stage class is the only class that stores a reference
315     // to these classes
316     //
317     // remove the stage from the race
318     stage.getRace().removeStage(stage);
319 }
320
321 /**
322  * {@inheritDoc}
323  */

```

```

324 @Override
325 public int addCategorizedClimbToStage(int stageId, Double location, SegmentType type, Double
    averageGradient,
326     Double length) throws IDNotRecognisedException, InvalidLocationException,
    InvalidStageStateException,
327     InvalidStageTypeException {
328
329     // a climb segment cannot be a sprint
330     if (type == SegmentType.SPRINT) {
331         // throw an illegal argument exception if the given segment time is sprint
332         throw new IllegalArgumentException("Segment type is not valid.");
333     }
334
335     // find stage in portal
336     // throws IDNotRecognisedException
337     Stage stage = findStage(stageId);
338
339     // check if the segment location is out of bounds of the stage
340     if (stage.getLength() < location) {
341         // throw InvalidLocationException
342         throw new InvalidLocationException("location is out of bounds of the stage length");
343     }
344
345     // check if the stage stage is correct
346     // cannot add a new segment if the stage has concluded the stage preparation
347     if (stage.getStageState() == StageState.WAITING_FOR_RESULTS) {
348         // throw InvalidStageStateException
349         throw new InvalidStageStateException("Stage cannot be added while waiting for results");
350     }
351
352     // time trial stages cannot contain a segment
353     // check if the stage type is time trial
354     if (stage.getType() == StageType.TT) {
355         // if the type is a time trial, throw an InvalidStageTypeException
356         throw new InvalidStageTypeException("Time-trial stages cannot contain any segment");
357     }
358
359     // create new climb segment with the paramiters
360     ClimbSegment segment = new ClimbSegment(stage, location, type, averageGradient, length);
361
362     // add the segment to the stage
363     stage.addSegment(segment);
364
365     // return the id of the new segment
366     return segment.getSegmentId();
367 }
368
369 /**
370  * {@inheritDoc}
371  */
372 @Override
373 public int addIntermediateSprintToStage(int stageId, double location) throws IDNotRecognisedException,
    InvalidLocationException, InvalidStageStateException, InvalidStageTypeException {
374
375     // find stage in portal
376

```

```

377 // throws IDNotRecognisedException
378 Stage stage = findStage(stageId);
379
380 // check the location is in bounds of the stage
381 if (stage.getLength() < location) {
382     // throw InvalidLocationException if out of bounds
383     throw new InvalidLocationException("location is out of bounds of the stage length");
384 }
385
386 // cannot add segment if stage has finished stage preparation
387 // check the stage state is not waiting for results
388 if (stage.getState() == StageState.WAITING_FOR_RESULTS) {
389     // throw InvalidStageStateException
390     throw new InvalidStageStateException("Stage cannot be removed while waiting for results");
391 }
392
393 // time trial stages cannot have any segments
394 // check the stage type is not time trial
395 if (stage.getType() == StageType.TT) {
396     // if the stage type is time trial, throw InvalidStageTypeException
397     throw new InvalidStageTypeException("Time-trial stages cannot contain any segment");
398 }
399
400 // create a new sprint segment
401 SprintSegment segment = new SprintSegment(stage, location);
402
403 // add sprint segment to stage
404 stage.addSegment(segment);
405
406 // return the new segment id
407 return segment.getSegmentId();
408 }
409
410 /**
411  * {@inheritDoc}
412  */
413 @Override
414 public void removeSegment(int segmentId) throws IDNotRecognisedException, InvalidStageStateException {
415
416     // find segment in portal
417     // throws IDNotRecognisedException
418     Segment segmentToRemove = findSegment(segmentId);
419
420     // get the stage the segment belongs to
421     Stage stage = segmentToRemove.getStage();
422
423     // cannot remove segment if stage preparation has finished
424     // check the state of the stage is not waiting for results
425     if (stage.getState() == StageState.WAITING_FOR_RESULTS) {
426         // if stage state is waiting for results, throw InvalidStageStateException
427         throw new InvalidStageStateException("Stage cannot be removed while waiting for results");
428     }
429
430     // remove segment from stage
431     stage.removeSegment(segmentToRemove);

```



```

432 }
433
434 /**
435  * {@inheritDoc}
436  */
437 @Override
438 public void concludeStagePreparation(int stageId) throws IDNotRecognisedException,
    InvalidStageStateException {
439     // find the stage in the portal
440     // throws IDNotRecognisedException
441     Stage stage = findStage(stageId);
442
443     // conclude the stage preparation
444     // throws InvalidStageStateException
445     stage.concludeStagePreparation();
446 }
447
448 /**
449  * {@inheritDoc}
450  */
451 @Override
452 public int[] getStageSegments(int stageId) throws IDNotRecognisedException {
453
454     // find the stage in the portal
455     // throws IDNotRecognisedException
456     Stage stage = findStage(stageId);
457
458     // init new array the size of the number of segments in the stage
459     int[] stageSegmentIds = new int[stage.getSegments().size()];
460
461     // loop through each segment in the stage
462     for (int i = 0; i < stageSegmentIds.length; i++) {
463         // add the segments id to the respective index in the array
464         stageSegmentIds[i] = stage.getSegments().get(i).getSegmentId();
465     }
466
467     // return the segment ids
468     return stageSegmentIds;
469 }
470
471 /**
472  * {@inheritDoc}
473  */
474 @Override
475 public int createTeam(String name, String description) throws IllegalNameException,
    InvalidNameException {
476
477     // check if team name already exists
478     // loop through each time
479     for (Team team : teams) {
480         // check if the team name is equal to the new team name
481         if (name.equals(team.getTeamName())) {
482             // if equal, throw IllegalNameException
483             throw new IllegalNameException("Team name already exists");
484         }

```

```

485     }
486
487     // check the description
488     // the description has to be less then 30 chars, not null and not empty
489     if (name.length() > 30 || name.equals("") || name == null) {
490         // throw InvalidNameException if params are not met
491         throw new InvalidNameException("Name cannot be null, empty or longer then 30");
492     }
493
494     // create a new team and add it to the teams array list
495     Team newTeam = new Team(name, description);
496     teams.add(newTeam);
497
498     // return the new teams id
499     return newTeam.getTeamId();
500 }
501
502 /**
503  * {@inheritDoc}
504  */
505 @Override
506 public void removeTeam(int teamId) throws IDNotRecognisedException {
507
508     // find the team in the portal
509     // throws IDNotRecognisedException
510     Team teamToRemove = findTeam(teamId);
511
512     // remove the team referance from the teams array list
513     // the team is the only object that stores the team realted data
514     // threfore, deleting the team also deletes all its related data
515     teams.remove(teamToRemove);
516 }
517
518 /**
519  * {@inheritDoc}
520  */
521 @Override
522 public int[] getTeams() {
523     // return the ids as an array of all the teams
524     // init new array the size of the numnber of teams in the portal
525     int[] teamsToReturn = new int[teams.size()];
526
527     // loop through each value in the array
528     for (int i = 0; i < teams.size(); i++) {
529         // add the team id to the respective index in the array
530         teamsToReturn[i] = teams.get(i).getTeamId();
531     }
532
533     // return the array
534     return teamsToReturn;
535 }
536
537 /**
538  * {@inheritDoc}
539  */

```

```

540 @Override
541 public int[] getTeamRiders(int teamId) throws IDNotRecognisedException {
542     // find team in portal
543     // Throws IDNotRecognisedException
544     Team team = findTeam(teamId);
545
546     // create an array the size of all the riders there are in the given team
547     int teamRiders[] = new int[team.getRiders().size()];
548
549     // loop through each rider in the team
550     for (int i = 0; i < team.getRiders().size(); i++) {
551         // add there id to the array to there corrsponding index
552         teamRiders[i] = team.getRiders().get(i).getRiderId();
553     }
554
555     // return the array of rider ids
556     return teamRiders;
557 }
558
559 /**
560  * {@inheritDoc}
561  */
562 @Override
563 public int createRider(int teamID, String name, int yearOfBirth) throws IDNotRecognisedException,
564     IllegalArgumentException {
565     // check that the rider name is not null
566     // and the year of birth is not before 1900
567     if (name == null || yearOfBirth < 1900) {
568         // if the name or year of birth breaks these paramiters, throw IllegalArgumentException
569         throw new IllegalArgumentException("name cannot be null or year less then 1900");
570     }
571
572     // find the riders team in the portal
573     // throws IDNotRecognisedException
574     Team ridersTeam = findTeam(teamID);
575
576     // create a new rider
577     Rider newRider = new Rider(ridersTeam, name, yearOfBirth);
578
579     // add the rider to the tema
580     ridersTeam.addRider(newRider);
581
582     // return the new riders id
583     return newRider.getRiderId();
584 }
585
586 /**
587  * {@inheritDoc}
588  */
589 @Override
590 public void removeRider(int riderId) throws IDNotRecognisedException {
591
592     // find rider in portal
593     // throws IDNotRecognisedException

```

```

594     Rider rider = findRider(riderId);
595
596     // remove the rider from the team
597     rider.getTeam().removeRider(rider);
598
599     // remove rider race results
600     // loop through each race in the portal
601     for (int i = 0; i < races.size(); i++)
602     {
603         // store the race
604         Race race = races.get(i);
605         // loop through each races stages
606         for (int j = 0; j < race.getStages().size(); j++)
607         {
608             // store the stage
609             Stage stage = race.getStages().get(i);
610
611             // create a temporary array to store all the results that need to be
612             // removed from the stage as they reference rider
613             ArrayList<Results> resultsToRemove = new ArrayList<>();
614             for (int m = 0; m < stage.getResults().size(); m++)
615             {
616                 // store the result
617                 Results result = stage.getResults().get(m);
618                 // check if the riders id of the result matches the given rider id to remove
619                 if (result.getRider().getRiderId() == riderId)
620                 {
621                     // if the result needs to be removed, add it to the remove list
622                     resultsToRemove.add(result);
623                 }
624             }
625
626             // loop through each result to remove
627             for (Results result : resultsToRemove)
628             {
629                 // remove result from stage
630                 stage.removeResults(result);
631             }
632         }
633     }
634 }
635
636 /**
637  * {@inheritDoc}
638  */
639 @Override
640 public void registerRiderResultsInStage(int stageId, int riderId, LocalTime... checkpoints)
641     throws IDNotRecognisedException, DuplicatedResultException, InvalidCheckpointsException,
642     InvalidStageStateException {
643
644     // find rider in portal
645     // throws IDNotRecognisedException
646     Rider rider = findRider(riderId);
647
648     // find stage in portal

```

```

649 // throws IDNotRecognisedException
650 Stage stage = findStage(stageId);
651
652 // check rider does not have duplicate result
653 // loop through each result in stage
654 for (int i = 0; i < stage.getResults().size(); i++) {
655     // check the rider does not have a result by
656     // comparing the riders id with the stages riders id
657     if (stage.getResults().get(i).getRider() == rider) {
658         // duplicate found
659         // throw DuplicatedResultException
660         throw new DuplicatedResultException("Stage allready has results for rider");
661     }
662 }
663
664 // check length of checkpoints is equal to n+2
665 if (checkpoints.length != stage.getSegments().size() + 2) {
666     // throw InvalidCheckpointsException
667     throw new InvalidCheckpointsException("length of checkpoints is invalid");
668 }
669
670 // check if stage is "waiting for results"
671 if (stage.getStageState() != StageState.WAITING_FOR_RESULTS) {
672     // stage waiting for results, throwInvalidStageStateException
673     throw new InvalidStageStateException("Invalid stage state");
674 }
675
676 // create a new result
677 Results result = new Results(stage, rider, checkpoints);
678
679 // add result to stage
680 stage.addResults(result);
681 }
682
683 /**
684  * {@inheritDoc}
685  */
686 @Override
687 public LocalTime[] getRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException {
688
689     // find stage in portal
690     // throws IDNotRecognisedException
691     Stage stage = findStage(stageId);
692
693     // find rider in portal
694     // throws IDNotRecognisedException
695     Rider rider = findRider(riderId);
696
697     // init rider result to null
698     Results riderResult = null;
699
700     // find rider results
701     // loop through each stages result
702     for (int i = 0; i < stage.getResults().size(); i++) {
703         // if the target riders id matches the stages results rider id

```

```

704         // then rider result found
705         if (rider == stage.getResults().get(i).getRider()) {
706             // save the rider result
707             riderResult = stage.getResults().get(i);
708         }
709     }
710
711     // if the rider result is still null, the result has not been found
712     if (riderResult == null) {
713         // return an empty localtime array
714         return new LocalTime[0];
715     }
716
717     // initialise a rider results array that is the size of all of the riders results in the stage
718     // add one at the end to store the elapsed time
719     LocalTime[] riderResults = new LocalTime[riderResult.getTimes().length + 1];
720
721     // loop through all the results times
722     for (int i = 0; i < riderResult.getTimes().length; i++) {
723         // store the result times in the array
724         riderResults[i] = riderResult.getTimes()[i];
725     }
726
727     // store the elapsed time in the last spot of the array
728     // elapsed time calculated using result calculateElapsedTime() function
729     riderResults[riderResult.getTimes().length] = riderResult.calculateElapsedTime();
730
731     // return the results array
732     return riderResults;
733 }
734
735 /**
736  * {@inheritDoc}
737  */
738 @Override
739 public LocalTime getRiderAdjustedElapsedTimeInStage(int stageId, int riderId) throws
    IDNotRecognisedException {
740
741     // find stage in portal
742     // throws IDNotRecognisedException
743     Stage stage = findStage(stageId);
744
745     // find the rider in the portal
746     // throws IDNotRecognisedException
747     Rider rider = findRider(riderId);
748
749     // initialise rider result as null
750     Results riderResult = null;
751
752     // find rider results
753     // loop through all the results in the stage
754     for (int i = 0; i < stage.getResults().size(); i++) {
755         // check if the results rider matches the target rider
756         if (rider == stage.getResults().get(i).getRider()) {
757             // if the ids are the same, result is found

```

```

758         // store result in riderResult
759         riderResult = stage.getResults().get(i);
760     }
761 }
762
763 // if riderResult is still null, no result found
764 if (riderResult == null) {
765     // if not result found, return null
766     return null;
767 }
768
769 // otherwise, return the riders adjusted elapsed time
770 // calculated using results calculateAdjustedElapsedTime()
771 return riderResult.calculateAdjustedElapsedTime();
772 }
773
774 /**
775  * {@inheritDoc}
776  */
777 @Override
778 public void deleteRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException {
779
780     // find stage in portal
781     // throws IDNotRecognisedException
782     Stage stage = findStage(stageId);
783
784     // find rider in stage
785     // throws IDNotRecognisedException
786     Rider rider = findRider(riderId);
787
788     // initialise rider result as null
789     Results riderResult = null;
790
791     // find rider results
792     // loop through all the results in the stage
793     for (int i = 0; i < stage.getResults().size(); i++) {
794         // check the results rider id matches target rider id
795         if (rider == stage.getResults().get(i).getRider()) {
796             // if ids match, rider result found
797             riderResult = stage.getResults().get(i);
798         }
799     }
800
801     // if rider result still null, rider result does not exist
802     if (riderResult == null) {
803         // no results to be removed
804         // return
805         return;
806     }
807
808     // otherwise, remove results from stage
809     stage.removeResults(riderResult);
810 }
811
812 /**

```

```

813     * {@inheritDoc}
814     */
815     @Override
816     public int[] getRidersRankInStage(int stageId) throws IDNotRecognisedException {
817
818         // find stage in portal
819         // throws IDNotRecognisedException
820         Stage stage = findStage(stageId);
821
822         // create a list of results the size of all the results in the stage
823         Results[] rankedResults = new Results[stage.getResults().size()];
824
825         // loop throug all results in stage
826         for (int i = 0; i < rankedResults.length; i++) {
827             // add the results to the results array
828             rankedResults[i] = stage.getResults().get(i);
829         }
830
831         // sort array of results by there elapsed time
832         // do this using custom comparitor class, ResultsElapsedTimeComparator.
833         // this returns the differeance between the elapsed times of the results and orders by difference
834         Arrays.sort(rankedResults, new ResultsElapsedTimeComparator());
835
836         // create a new array to return the riders results
837         // array the size of all theriders in the stage
838         int[] riderRanks = new int[rankedResults.length];
839
840         // loop throug all the results
841         for (int i = 0; i < riderRanks.length; i++) {
842             // add the rider id to the array
843             // since rankedResults is ordered by elapsed time, so will riderRanks
844             riderRanks[i] = rankedResults[i].getRider().getRiderId();
845         }
846
847         // return the ranked list of riders
848         return riderRanks;
849     }
850
851     /**
852     * {@inheritDoc}
853     */
854     @Override
855     public LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId) throws IDNotRecognisedException {
856
857         // find stage in portal
858         // throws IDNotRecognisedException
859         Stage stage = findStage(stageId);
860
861         // get riders rank in stage
862         // throws IDNotRecognisedException
863         int[] ridersRanked = getRidersRankInStage(stageId);
864
865         // create a new array of localtimes to store the ranked adjusted elpased times
866         LocalTime[] riderAdjustedElapsedTimes = new LocalTime[ridersRanked.length];
867

```



```

868 // loop through all the riders from ridersRanked
869 for (int i = 0; i < riderAdjustedElapsedTimes.length; i++) {
870     // get the rider
871     Rider rider = findRider(ridersRanked[i]);
872
873     // loop throuh the stages results to find the rider result
874     for (int x = 0; x < stage.getResults().size(); x++) {
875         // cheack if the stage result belongs to the rider
876         if (stage.getResults().get(x).getRider() == rider) {
877             // use the found result to calculate the adjusted elapsed time
878             // append to array at index i. Sicne ridersRanked is ordered by elapsed time,
879             // so will riderAdjustedElapsedTimes
880             riderAdjustedElapsedTimes[i] = stage.getResults().get(x).calculateAdjustedElapsedTime();
881
882             // break out of inner loop to save time
883             continue;
884         }
885     }
886 }
887
888 // return the array or ranked adjusted elapsed times
889 return riderAdjustedElapsedTimes;
890 }
891
892 /**
893  * {@inheritDoc}
894  */
895 @Override
896 public int[] getRidersPointsInStage(int stageId) throws IDNotRecognisedException {
897
898     // find stage in portal
899     // throws IDNotRecognisedException
900     Stage stage = findStage(stageId);
901
902     // get a ranked list of rider ids
903     // throws IDNotRecognisedException
904     int[] ridersRanked = getRidersRankInStage(stageId);
905
906     // init an array to store the ranked riders points
907     // the size of the number of riders in the stage
908     int[] riderPoints = new int[ridersRanked.length];
909
910     // loop through all the riders
911     for (int i = 0; i < riderPoints.length; i++) {
912         // get the rider
913         Rider rider = findRider(ridersRanked[i]);
914
915         // add the rider points to the riderPoints array at index i
916         // since ridersRanked is ordered by elapsed time, so will riderPoints
917         // rider points calculated using Rider.getPointsInStage() function
918         riderPoints[i] = rider.getPointsInStage(stage, i+1);
919     }
920
921     // return the ordered array of rider points
922     return riderPoints;

```

```

923     }
924
925     /**
926     * {@inheritDoc}
927     */
928     @Override
929     public int[] getRidersMountainPointsInStage(int stageId) throws IDNotRecognisedException {
930
931         // find stage in portal
932         // throws IDNotRecognisedException
933         Stage stage = findStage(stageId);
934
935         // get the ranked list of riders
936         // throws IDNotRecognisedException
937         int[] ridersRanked = getRidersRankInStage(stageId);
938
939         // init a new array to store the mountain points for a rider in a stage
940         int[] riderPoints = new int[ridersRanked.length];
941
942         // loop through each rider in the stage
943         for (int i = 0; i < riderPoints.length; i++) {
944             // get the rider
945             Rider rider = findRider(ridersRanked[i]);
946
947             // set the points at index i in riderPoints for the mountain points that rider has aquired
948             // mountain points calculated using Stage.pointsForMountainClassification()
949             // since ridersRanked is ordered by elapsed time, so will riderPoints
950             riderPoints[i] = stage.pointsForMountainClassification(rider);
951         }
952
953         // return the ordered list of mountain points
954         return riderPoints;
955     }
956
957     /**
958     * {@inheritDoc}
959     */
960     @Override
961     public void eraseCyclingPortal() {
962
963         // clear cycling portal lists
964         // clear team
965         // clear races
966         teams.clear();
967         races.clear();
968
969         // reset counters
970         // reset race id counter
971         // reset rider id counter
972         // reset segment id counter
973         // reset stage id counter
974         // reset ream id counter
975         Race.resetCounter();
976         Rider.resetCounter();
977         Segment.resetCounter();

```

```

978         Stage.resetCounter();
979         Team.resetCounter();
980     }
981
982     /**
983     * {@inheritDoc}
984     */
985     @Override
986     public void saveCyclingPortal(String filename) throws IOException {
987
988         // create a new output file stream
989         ObjectOutputStream ostream = new ObjectOutputStream(new FileOutputStream(filename));
990
991         // write the cycling portal to the output stream
992         ostream.writeObject(this);
993
994         // close and commit the output stream
995         ostream.close();
996     }
997
998     /**
999     * {@inheritDoc}
1000     */
1001     @Override
1002     public void loadCyclingPortal(String filename) throws IOException, ClassNotFoundException {
1003
1004         // create a new input file stream
1005         ObjectInputStream istream = new ObjectInputStream(new FileInputStream(filename));
1006
1007         // create a new object and assign it to the value in the file
1008         Object portalObject = istream.readObject();
1009
1010         // check if the portal is an instance of cycling portal
1011         if (!(portalObject instanceof CyclingPortal)) {
1012             // close input file stream
1013             istream.close();
1014
1015             // throw exception
1016             throw new InvalidClassException("Object from file is not an instance of cycling portal");
1017         }
1018
1019         // otherwise, upcast the portal object to a cycling portal
1020         CyclingPortal portal = (CyclingPortal)portalObject;
1021
1022         // assign this cycling portals race list and team list to serialised portals
1023         this.races = portal.races;
1024         this.teams = portal.teams;
1025
1026         // close the input file stream
1027         istream.close();
1028     }
1029
1030     /**
1031     * {@inheritDoc}
1032     */

```

```

1033 @Override
1034 public void removeRaceByName(String name) throws NameNotRecognisedException {
1035
1036     // initialise a race as null
1037     Race race = null;
1038
1039     // loop through all the races
1040     for (int i = 0; i < races.size(); i++) {
1041         // cheack the target name and races name match
1042         if (races.get(i).getName() == name) {
1043             // if they match, assin race to the current race
1044             race = races.get(i);
1045
1046             // break out of loop as race allready found
1047             break;
1048         }
1049     }
1050
1051     // if race still null, race not found
1052     if (race == null) {
1053         // throw NameNotRecognisedException
1054         throw new NameNotRecognisedException("Race is not found with name " + name);
1055     }
1056
1057     // remove the race from the portal
1058     races.remove(race);
1059
1060     // since stage is stored in race and segments and results are stored in stage
1061     // deleting the race will also delete segments, results and stage
1062 }
1063
1064 /**
1065  * {@inheritDoc}
1066  */
1067 @Override
1068 public int[] getRidersGeneralClassificationRank(int raceId) throws IDNotRecognisedException {
1069     // find race in portal
1070     // throws IDNotRecognisedException
1071     Race race = findRace(raceId);
1072
1073     // if one of the stages does not have results, return an empty array
1074     // loop through each stage in race
1075     for (int i = 0; i < race.getStages().size(); i++) {
1076         // cheack if stage does not have results (results list empty)
1077         if (race.getStages().get(i).getResults().size() == 0) {
1078             // return empty array
1079             return new int[0];
1080         }
1081     }
1082
1083     // initialize an array list of results
1084     ArrayList<Results> results = new ArrayList<>();
1085
1086     // loop through each stage in the race
1087     for (int i = 0; i < race.getStages().size(); i++) {

```

```

1088     // loop through each result in the race
1089     for (int x = 0; x < race.getStages().get(i).getResults().size(); x++) {
1090         // add result to results array list
1091         results.add(race.getStages().get(i).getResults().get(x));
1092     }
1093 }
1094
1095 // init a hash map to pair up the rider with there result
1096 Map<Rider, LocalTime> timesMap = new HashMap<Rider, LocalTime>();
1097
1098 // loop through all results
1099 for (int i = 0; i < results.size(); i++) {
1100     // store the current rider
1101     Rider currentRider = results.get(i).getRider();
1102
1103     // check weather the rider has allready been entered into the hash map
1104     if (timesMap.containsKey(currentRider)) {
1105         // if allready added, add the result adjusted elapsed time to the value allready
1106         // in the hash map
1107
1108         // calculate the ammount of nannos the of the adjusted elapsed time
1109         long nanos = results.get(i).calculateAdjustedElapsedTime().toNanoOfDay();
1110
1111         // add the nannos to the old value in the hash map
1112         LocalTime newTime = timesMap.get(currentRider).plusNanos(nanos);
1113
1114         // relpace the old value in hash map with new value
1115         // (new value is old time + result time)
1116         timesMap.replace(currentRider, newTime);
1117     } else {
1118         // if the rider has not allready been entered into the hash map,
1119         // add the rider to the hash map paired with there adjusted elapsed time
1120         timesMap.put(currentRider, results.get(i).calculateAdjustedElapsedTime());
1121     }
1122 }
1123
1124 // create an array list of map entrys
1125 ArrayList<Map.Entry<Rider, LocalTime>> sorted = new ArrayList<>(timesMap.entrySet());
1126
1127 // sort the array of map entrys using the custom comparotor ResultsAdjustedElapsedTimeCompatiror
1128 // this comparotor comparse Map.Entry<Rider, LocalTime> by returning the differeance between
1129 // the local times
1130 sorted.sort(new ResultsAdjustedElapsedTimeCompatiror());
1131
1132 // init array of rider ids, the size of all the riders in the portal
1133 int orderedRiderIds[] = new int[sorted.size()];
1134
1135 // loop through all the riders in the portal
1136 for (int i = 0; i < orderedRiderIds.length; i++) {
1137     // add the riders id to the corrsponding index in the array
1138     // since the id is from the sorted array, orderedRiderIds will be ordered too
1139     orderedRiderIds[i] = sorted.get(i).getKey().getRiderId();
1140 }
1141
1142 // return the list of ordered rider ids

```

```

1143     return orderedRiderIds;
1144 }
1145
1146 /**
1147  * {@inheritDoc}
1148  */
1149 @Override
1150 public LocalTime[] getGeneralClassificationTimesInRace(int raceId) throws IDNotRecognisedException {
1151     // find race in portal
1152     // throws IDNotRecognisedException
1153     Race race = findRace(raceId);
1154
1155     // check if any of the results are empty
1156     // loop through all the stages in the race
1157     for (int i = 0; i < race.getStages().size(); i++) {
1158         // check if the results list for any stage is empty
1159         if (race.getStages().get(i).getResults().size() == 0) {
1160             // if empty, return an empty local time array
1161             return new LocalTime[0];
1162         }
1163     }
1164
1165     // initialize a new array list of results to store all the results for the race
1166     ArrayList<Results> results = new ArrayList<>();
1167
1168     // loop through each stage in the race
1169     for (int i = 0; i < race.getStages().size(); i++) {
1170         // loop through each result in the stage
1171         for (int x = 0; x < race.getStages().get(i).getResults().size(); x++) {
1172             // add the result to the results list
1173             results.add(race.getStages().get(i).getResults().get(x));
1174         }
1175     }
1176
1177     // initialize a hash map to pair together the riders and there times
1178     Map<Rider, LocalTime> timesMap = new HashMap<Rider, LocalTime>();
1179
1180     // loop through each result in the race
1181     for (int i = 0; i < results.size(); i++) {
1182         // store the current rider
1183         Rider currentRider = results.get(i).getRider();
1184
1185         // check if the hash map already contains an entry for the current rider
1186         if (timesMap.containsKey(currentRider)) {
1187             // if the hash map contains an entry for the rider,
1188             // add the current results adjusted elapsed time to the value for the rider
1189             // already in the hash map
1190
1191             // calculate the adjusted elapsed time for the current result in nano seconds
1192             long nanos = results.get(i).calculateAdjustedElapsedTime().toNanoOfDay();
1193
1194             // add the current results nano seconds to the riders current result
1195             LocalTime newTime = timesMap.get(currentRider).plusNanos(nanos);
1196
1197             // replace the old time with the new time

```

```

1198         timesMap.replace(currentRider, newTime);
1199     } else {
1200         // if rider does not allready have a map entry
1201         // add them into the hashmap paired with there time
1202         timesMap.put(currentRider, results.get(i).calculateAdjustedElapsedTime());
1203     }
1204 }
1205
1206 // create an array list of all the map entrys
1207 ArrayList<Map.Entry<Rider, LocalTime>> sorted = new ArrayList<>(timesMap.entrySet());
1208
1209 // sort the map using the custom comparitor whitch compares map entrys based of
1210 // the differance between there LocalTimes
1211 sorted.sort(new ResultsAdjustedElapsedTimeCompatiror());
1212
1213 // create an array of localTimes the size of all the riders in the race
1214 LocalTime orderedTimes[] = new LocalTime[sorted.size()];
1215
1216 // loop throguh all the riders
1217 for (int i = 0; i < orderedTimes.length; i++) {
1218     // set the array to the local time of the sorted lists value at the same index
1219     // this means the ordered time list will also be sorted the same way the soreded array list is
1220     orderedTimes[i] = sorted.get(i).getValue();
1221 }
1222
1223 // return the array of ordered times
1224 return orderedTimes;
1225 }
1226
1227 /**
1228  * {@inheritDoc}
1229  */
1230 @Override
1231 public int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException {
1232     // find race in portal
1233     // throws IDNotRecognisedException
1234     Race race = findRace(raceId);
1235
1236     // check if any of the results are empty
1237     // loop through all the stages in the race
1238     for (int i = 0; i < race.getStages().size(); i++) {
1239         // check if the results list for any stage is empty
1240         if (race.getStages().get(i).getResults().size() == 0) {
1241             // if empty, return an empty local time array
1242             return new int[0];
1243         }
1244     }
1245
1246     // initialize a new array list of results to store all the results for the race
1247     ArrayList<Results> results = new ArrayList<>();
1248
1249     // loop through each stage in the race
1250     for (int i = 0; i < race.getStages().size(); i++) {
1251         // loop through eahc reuslt in the stage
1252         for (int x = 0; x < race.getStages().get(i).getResults().size(); x++) {

```

```

1253         // add the result to the results list
1254         results.add(race.getStages().get(i).getResults().get(x));
1255     }
1256 }
1257
1258 // initialize a hash map to pair together the riders and there times
1259 Map<Rider, LocalTime> timesMap = new HashMap<Rider, LocalTime>();
1260
1261 // loop through each result in the race
1262 for (int i = 0; i < results.size(); i++) {
1263     // store the current rider
1264     Rider currentRider = results.get(i).getRider();
1265
1266     // check if the hash map allready contains an entry for the current rider
1267     if (timesMap.containsKey(currentRider)) {
1268         // if the hash map contains an entry for the rider,
1269         // add the current results adjusted elapsed time to the value for the rider
1270         // allready in the hash map
1271
1272         //calculate the adjusted elapsed time for the current result in nano seconds
1273         long nanos = results.get(i).calculateAdjustedElapsedTime().toNanoOfDay();
1274
1275         // add the current results nano seconds to the riders current result
1276         LocalTime newTime = timesMap.get(currentRider).plusNanos(nanos);
1277
1278         // replace the old time with the new time
1279         timesMap.replace(currentRider, newTime);
1280     } else {
1281         // if rider does not allreayd have a map entry
1282         // add them into the hashmap paired with there time
1283         timesMap.put(currentRider, results.get(i).calculateAdjustedElapsedTime());
1284     }
1285 }
1286
1287 // create an array list of all the map entrys
1288 ArrayList<Map.Entry<Rider, LocalTime>> sorted = new ArrayList<>(timesMap.entrySet());
1289
1290 // sort the map using the custom comparitor whitch compares map entrys based of
1291 // the differance between there LocalTimes
1292 sorted.sort(new ResultsAdjustedElapsedTimeCompatiror());
1293
1294 // init a new array the size of all the riders in the race
1295 int ridersPoints[] = new int[sorted.size()];
1296 // loop through all the riders in the race
1297 for (int i = 0; i < ridersPoints.length; i++) {
1298     // init there inital points to zero
1299     ridersPoints[i] = 0;
1300 }
1301
1302 // for each rider, find the total points in all stages
1303 // loop through each stage in the race
1304 for (int i = 0; i < race.getStages().size(); i++) {
1305     // store the current stage
1306     Stage currentStage = race.getStages().get(i);
1307

```



```

1308         // get a list of all the riders rank in that stage
1309         int ridersRanks[] = getRidersRankInStage(currentStage.getStageId());
1310
1311         // loop through each riders rank
1312         for (int x = 0; x < ridersRanks.length; x++) {
1313             // find the riders id at rank x
1314             int id = ridersRanks[x];
1315
1316             // add one to rank (so person with rank 0 is actually 1st)
1317             int rank = x + 1;
1318
1319             // loop through the ordered list of adjusted elapsed times and riders
1320             for (int y = 0; y < sorted.size(); y++) {
1321                 // check if the rider id in the sorted list matches the current rider
1322                 if (id == sorted.get(y).getKey().getRiderId()) {
1323                     // add the points for that stage and rider to the riders points list
1324                     ridersPoints[y] += sorted.get(y).getKey().getPointsInStage(currentStage, rank);
1325                 }
1326             }
1327         }
1328     }
1329
1330     // return the riders points
1331     return ridersPoints;
1332 }
1333
1334 /**
1335  * {@inheritDoc}
1336  */
1337 @Override
1338 public int[] getRidersMountainPointsInRace(int raceId) throws IDNotRecognisedException {
1339     // find race in portal
1340     // throws IDNotRecognisedException
1341     Race race = findRace(raceId);
1342
1343     // check if any of the results are empty
1344     // loop through all the stages in the race
1345     for (int i = 0; i < race.getStages().size(); i++) {
1346         // check if the results list for any stage is empty
1347         if (race.getStages().get(i).getResults().size() == 0) {
1348             // if empty, return an empty local time array
1349             return new int[0];
1350         }
1351     }
1352
1353     // initialize a new array list of results to store all the results for the race
1354     ArrayList<Results> results = new ArrayList<>();
1355
1356     // loop through each stage in the race
1357     for (int i = 0; i < race.getStages().size(); i++) {
1358         // loop through each result in the stage
1359         for (int x = 0; x < race.getStages().get(i).getResults().size(); x++) {
1360             // add the result to the results list
1361             results.add(race.getStages().get(i).getResults().get(x));
1362         }
1363     }

```

```

1363 }
1364
1365 // initialize a hash map to pair together the riders and there times
1366 Map<Rider, LocalTime> timesMap = new HashMap<Rider, LocalTime>();
1367
1368 // loop through each result in the race
1369 for (int i = 0; i < results.size(); i++) {
1370     // store the current rider
1371     Rider currentRider = results.get(i).getRider();
1372
1373     // check if the hash map allready contains an entry for the current rider
1374     if (timesMap.containsKey(currentRider)) {
1375         // if the hash map contains an enrty for the rider,
1376         // add the current results adjusted elapsed time to the value for the rider
1377         // allready in the hash map
1378
1379         //calculate the adjusted elapsed time for the current result in nano seconds
1380         long nanos = results.get(i).calculateAdjustedElapsedTime().toNanoOfDay();
1381
1382         // add the current results nano seconds to the riders current result
1383         LocalTime newTime = timesMap.get(currentRider).plusNanos(nanos);
1384
1385         // replace the old time with the new time
1386         timesMap.replace(currentRider, newTime);
1387     } else {
1388         // if rider does not allreayd have a map entry
1389         // add them into the hashmap paired with there time
1390         timesMap.put(currentRider, results.get(i).calculateAdjustedElapsedTime());
1391     }
1392 }
1393
1394 // create an array list of all the map entries
1395 ArrayList<Map.Entry<Rider, LocalTime>> sorted = new ArrayList<>(timesMap.entrySet());
1396
1397 // sort the map using the custom comparitor whitch compares map entrys based of
1398 // the difference between there LocalTimes
1399 sorted.sort(new ResultsAdjustedElapsedTimeCompatiror());
1400
1401 // init a new array the size of all the riders in the race
1402 int ridersPoints[] = new int[sorted.size()];
1403
1404 // loop through all the riders in the race
1405 for (int i = 0; i < ridersPoints.length; i++) {
1406     // init there inital points to zero
1407     ridersPoints[i] = 0;
1408 }
1409
1410 // for each rider, find the total points in all stages
1411 // loop through each stage in the race
1412 for (int i = 0; i < race.getStages().size(); i++) {
1413     // store the current stage
1414     Stage currentStage = race.getStages().get(i);
1415
1416     // loop through each rider in the sorted array
1417     for (int y = 0; y < sorted.size(); y++) {

```

```

1418         // add the rider at y's mountain points to the points array
1419         // points calculated using Rider.getMountainPointsInStage()
1420         ridersPoints[y] += sorted.get(y).getKey().getMountainPointsInStage(currentStage);
1421     }
1422 }
1423
1424 // return the riders mouinain points ordered by adjusted elapsed time
1425 return ridersPoints;
1426 }
1427
1428 /**
1429  * {@inheritDoc}
1430  */
1431 @Override
1432 public int[] getRidersPointClassificationRank(int raceId) throws IDNotRecognisedException {
1433     // get riders ranks for race
1434     int riderIds[] = getRidersGeneralClassificationRank(raceId);
1435
1436     // get riders points for race
1437     int riderPoints[] = getRidersPointsInRace(raceId);
1438
1439     // create a map mapping rider to there points
1440     Map<Rider, Integer> pointsMap = new HashMap<Rider, Integer>();
1441
1442     // loop through each rider in the race
1443     for (int i = 0; i < riderIds.length; i++) {
1444         // store the current rider
1445         Rider currentRider = findRider(riderIds[i]);
1446
1447         // add the current rider to the hash map with there points in race
1448         pointsMap.put(currentRider, riderPoints[i]);
1449     }
1450
1451     // create an array list of map entrys
1452     ArrayList<Map.Entry<Rider, Integer>> sorted = new ArrayList<>(pointsMap.entrySet());
1453
1454     // sort the array list by comparing the points
1455     // points are compared using a custom comparitor and annonamys function which returs the difference
1456     // between the current result and the result after
1457     // p2 - p1 in order to get reverce order (p1 - p2 for acending order)
1458     sorted.sort(Comparator.comparing(Map.Entry<Rider, Integer>::getValue, (p1, p2) -> {
1459         return p2 - p1;
1460     }));
1461
1462     // create an array to store the ids of all the riders
1463     int sortedIds[] = new int[riderIds.length];
1464
1465     // loop through all the riders in the sorted array
1466     for (int i = 0; i < sortedIds.length; i++) {
1467         // store the riders id in the array matching the index of the sorted array
1468         sortedIds[i] = sorted.get(i).getKey().getRiderId();
1469     }
1470
1471     // return the sorted list (by points aquired) of rider ids
1472     return sortedIds;

```

```

1473     }
1474
1475     /**
1476      * {@inheritDoc}
1477      */
1478     @Override
1479     public int[] getRidersMountainPointClassificationRank(int raceId) throws IDNotRecognisedException {
1480         // get riders ranks for race
1481         int riderIds[] = getRidersGeneralClassificationRank(raceId);
1482
1483         // get riders mountain points for race
1484         int riderPoints[] = getRidersMountainPointsInRace(raceId);
1485
1486         // create a map mapping rider to there points
1487         Map<Rider, Integer> pointsMap = new HashMap<Rider, Integer>();
1488
1489         // loop through each rider in the race
1490         for (int i = 0; i < riderIds.length; i++) {
1491             // store the current rider
1492             Rider currentRider = findRider(riderIds[i]);
1493
1494             // add the current rider to the hash map with there points in race
1495             pointsMap.put(currentRider, riderPoints[i]);
1496         }
1497
1498         // create an array list of map entrys
1499         ArrayList<Map.Entry<Rider, Integer>> sorted = new ArrayList<>(pointsMap.entrySet());
1500
1501         // sort the array list by comparing the points
1502         // points are compared using a custom comparitor and annonamys function which returs the difference
1503         // between the current result and the result after
1504         // p2 - p1 in order to get reverce order (p1 - p2 for acending order)
1505         sorted.sort(Comparator.comparing(Map.Entry<Rider, Integer>::getValue, (p1, p2) -> {
1506             return p2 - p1;
1507         }));
1508
1509         // create an array to store the ids of all the riders
1510         int sortedIds[] = new int[riderIds.length];
1511
1512         // loop through all the riders in the sorted array
1513         for (int i = 0; i < sortedIds.length; i++) {
1514             // store the riders id in the array matching the index of the sorted array
1515             sortedIds[i] = sorted.get(i).getKey().getRiderId();
1516         }
1517
1518         // return the sorted list (by points aquired) of rider ids
1519         return sortedIds;
1520     }
1521 }
1522 }

```

2 ClimbSegment.java

```

1 package cycling;

```

```

2
3 /**
4  * Class for ClimbSegemt extents {@link Segment}. Stores additional details requeted if the segment is a
5  * climbing segment.
6  *
7  * @author Ethan Hofton
8  * @author Jon Tao
9  * @version 1.0
10  *
11  */
12 public class ClimbSegment extends Segment {
13
14     private Double averageGradient;
15     private Double length;
16
17     /**
18      * The constructor for climb segment.
19      *
20      * @param stage the stage the segment is in
21      * @param location the location of the segment within the stage
22      * @param type the type of segment
23      * @param averageGradient average gradient of segment
24      * @param length length of segment
25      */
26     public ClimbSegment(Stage stage, double location, SegmentType type, Double averageGradient, Double
        length) {
27         // call Segment custructor
28         super(stage, location, type);
29
30         // set gradient and length
31         this.averageGradient = averageGradient;
32         this.length = length;
33     }
34
35     /**
36      * Getter for {@code this.averageGradient}
37      *
38      * @return the average gradient
39      */
40     public Double getAverageGradient() {
41         return this.averageGradient;
42     }
43
44     /**
45      * Getter for {@code this.length}
46      *
47      * @return the average gradient
48      */
49     public Double getLength() {
50         return this.length;
51     }
52
53     /**
54      * Returns if the segment is a climb segment.
55      * Overrides {@link cycling.Segment.isClimb}

```

```

56     *
57     * @return wether the segment is a climb or not
58     */
59     @Override
60     boolean isClimb() {
61         return true;
62     }
63
64     /**
65     * Returns if the segment is a sprint segment.
66     * Overrides {@link cycling.Segment.isSprint}
67     *
68     * @return wether the segment is a sprint or not
69     */
70     @Override
71     boolean isSprint() {
72         return false;
73     }
74
75     /**
76     * Calculates the points mountain points for the segment
77     * Data from Figure 2 in coursework spesification
78     *
79     * @param rank the rank of the rider
80     * @return the points the rider gets for the given rank
81     */
82     public int mountainPoints(int rank) {
83         // switch the segment type
84         // and return the points aquered for that type of segment given the riders rank
85         switch (type) {
86             case C1:
87                 // return points for C1
88                 return pointsFor1C(rank);
89             case C2:
90                 // return points for C2
91                 return pointsFor2C(rank);
92             case C3:
93                 // return points for C3
94                 return pointsFor3C(rank);
95             case C4:
96                 // return points for C4
97                 return pointsFor4C(rank);
98             case HC:
99                 // return points for HC
100                return pointsForHC(rank);
101            default:
102                // if segment type is not as above
103                // no points will be aquered so return 0
104                return 0;
105        }
106    }
107
108    /**
109    * Calculates the points for HC Mountain segment
110    * Data from Figure 2 in coursework spesification

```

```

111  *
112  * @param rank the rank of the rider
113  * @return the points the rider gets for the given rank
114  */
115  static public int pointsForHC(int rank) {
116      // swithc the rank and return the points aquered for HC
117      // data is from Figure 2 in coursework spec
118      switch (rank) {
119          case 1:
120              return 20;
121          case 2:
122              return 15;
123          case 3:
124              return 12;
125          case 4:
126              return 10;
127          case 5:
128              return 8;
129          case 6:
130              return 6;
131          case 7:
132              return 4;
133          case 8:
134              return 2;
135          default:
136              return 0;
137      }
138  }
139
140  /**
141   * Calculates the points for 1C Mountain segment
142   * Data from Figure 2 in coursework spesification
143   *
144   * @param rank the rank of the rider
145   * @return the points the rider gets for the given rank
146   */
147  static public int pointsFor1C(int rank) {
148      // swithc the rank and return the points aquered for C1
149      // data is from Figure 2 in coursework spec
150      switch (rank) {
151          case 1:
152              return 10;
153          case 2:
154              return 8;
155          case 3:
156              return 6;
157          case 4:
158              return 4;
159          case 5:
160              return 2;
161          case 6:
162              return 1;
163          default:
164              return 0;
165      }

```

```

166 }
167
168 /**
169  * Calculates the points for 2C Mountain segment
170  * Data from Figure 2 in coursework spesification
171  *
172  * @param rank the rank of the rider
173  * @return the points the rider gets for the given rank
174  */
175 static public int pointsFor2C(int rank) {
176     // swicth the rank and return the points aquered for C2
177     // data is from Figure 2 in coursework spec
178     switch (rank) {
179         case 1:
180             return 5;
181         case 2:
182             return 3;
183         case 3:
184             return 2;
185         case 4:
186             return 1;
187         default:
188             return 0;
189     }
190 }
191
192 /**
193  * Calculates the points for 3C Mountain segment
194  * Data from Figure 2 in coursework spesification
195  *
196  * @param rank the rank of the rider
197  * @return the points the rider gets for the given rank
198  */
199 static public int pointsFor3C(int rank) {
200     // swicth the rank and return the points aquered for C3
201     // data is from Figure 2 in coursework spec
202     switch (rank) {
203         case 1:
204             return 2;
205         case 2:
206             return 1;
207         default:
208             return 0;
209     }
210 }
211
212 /**
213  * Calculates the points for 4C Mountain segment
214  * Data from Figure 2 in coursework spesification
215  *
216  * @param rank the rank of the rider
217  * @return the points the rider gets for the given rank
218  */
219 static public int pointsFor4C(int rank) {
220     // swicth the rank and return the points aquered for C4

```



```

221     // data is from Figure 2 in coursework spec
222     switch (rank) {
223     case 1:
224         return 1;
225     default:
226         return 0;
227     }
228 }
229 }

```

3 Race.java

```

1  package cycling;
2
3  import java.io.Serializable;
4  import java.util.ArrayList;
5
6  /**
7   * Race class to store the race id and additional details relevant
8   * to the race
9   *
10  * @author Ethan Hofton
11  * @author Jon Tao
12  * @version 1.0
13  */
14  public class Race implements Serializable {
15      private static int raceCount = 0;
16
17      private int raceId;
18      private String name;
19      private String description;
20      private ArrayList<Stage> stages;
21
22      /**
23       * Race class constructor
24       *
25       * @param name the name of the race
26       * @param description the description of the race
27       */
28      public Race(String name, String description) {
29          // set the race id and increment the static race counter
30          this.raceId = raceCount++;
31
32          // set the rest of the class attributes
33          this.name = name;
34          this.description = description;
35
36          // initialize stages array list
37          this.stages = new ArrayList<>();
38      }
39
40      /**
41       * getter for {@code this.raceId}
42       *

```

```

43     * @return the id of the race
44     */
45     public int getRaceId() {
46         return raceId;
47     }
48
49     /**
50     * getter for {@code this.name}
51     *
52     * @return the name of the race
53     */
54     public String getName() {
55         return name;
56     }
57
58     /**
59     * getter for {@code this.description}
60     *
61     * @return the description of the race
62     */
63     public String getDescription() {
64         return description;
65     }
66
67     /**
68     * getter for {@code this.stages}
69     *
70     * @return the list of stages in the race
71     * @see cycling.Stage
72     */
73     public ArrayList<Stage> getStages() {
74         return stages;
75     }
76
77     /**
78     * adds a stage to the race
79     *
80     * @param stage the stage class to be added to the race
81     * @see cycling.Stage
82     */
83     public void addStage(Stage stage) {
84         // add stage to stages array list
85         stages.add(stage);
86     }
87
88     /**
89     * remove stage from race
90     *
91     * @param stage the stage class to be removed from the race
92     * @throws IDNotRecognisedException if the stage is not in the race
93     * @see cycling.Stage
94     */
95     public void removeStage(Stage stage) throws IDNotRecognisedException {
96         // check if stages contains stage to remove
97         if (!stages.contains(stage)) {

```

```

98         // if stages array list does not contain a stage, throw IDNotRecognisedException
99         throw new IDNotRecognisedException("stage does not exist in race with Id '"+raceId+"'");
100     }
101     // remove stage from stages array list
102     stages.remove(stage);
103 }
104
105 /**
106  * check if the race contains a given stage
107  *
108  * @param stage the stage to be checked
109  * @return boolean whether the race contains the stage
110  * @see cycling.Stage
111  */
112 public boolean containsStage(Stage stage) {
113     // return whether stages contains array list
114     return stages.contains(stage);
115 }
116
117 /**
118  * Reset the static counter to set the ids
119  */
120 public static void resetCounter() {
121     // reset static race count to zero
122     raceCount = 0;
123 }
124 }

```

4 ResultsAdjustedElapsedTimeComparator.java

```

1 package cycling;
2
3 import java.time.LocalDateTime;
4 import java.util.Comparator;
5 import java.util.Map;
6
7 /**
8  * comparator for results class compare by adjusted elapsed time
9  *
10  * @author Ethan Hofton
11  * @author Jon Tao
12  * @version 1.0
13  */
14 public class ResultsAdjustedElapsedTimeComparator implements Comparator<Map.Entry<Rider,LocalTime>> {
15     /**
16      * Compare 2 results using {@code LocalDateTime.compareTo}
17      *
18      * @param result1 first result to compare
19      * @param result2 second result to compare
20      * @return the value of result1 - result2
21      */
22     @Override
23     public int compare(Map.Entry<Rider,LocalTime> result1, Map.Entry<Rider,LocalTime> result2) {
24         // compare the value of each entry using LocalDateTime.compareTo

```

```

25     return result1.getValue().compareTo(result2.getValue());
26 }
27 }

```

5 ResultsElapsedTimeComparator.java

```

1  package cycling;
2
3  import java.util.Comparator;
4
5  /**
6   * compatoror for results class compare by elapsed time
7   *
8   * @author Ethan Hofton
9   * @author Jon Tao
10  * @version 1.0
11  */
12 public class ResultsElapsedTimeComparator implements Comparator<Results> {
13
14     /**
15      * Compare 2 reuslts using {@code LocalTime.compareTo}
16      *
17      * @param result1 first result to compare
18      * @param result2 second result to copmare
19      * @return the value of result1 - result2
20      */
21     @Override
22     public int compare(Results result1, Results result2) {
23         // compare 2 results by there adjusted elapsed time
24         // using LocalTime.compareTo and Reuslt.calculateElapsedTime
25         return result1.calculateElapsedTime().compareTo(result2.calculateElapsedTime());
26     }
27 }

```

6 ResultsMountainTimeCompatoror.java

```

1  package cycling;
2
3  import java.util.Comparator;
4
5  /**
6   * compatoror for results class compare by elapsed time
7   *
8   * @author Ethan Hofton
9   * @author Jon Tao
10  * @version 1.0
11  */
12 public class ResultsMountainTimeCompatoror implements Comparator<Results> {
13
14     private int pos;
15
16     /**
17      * Constructor for class
18      *

```

```

19     * @param pos the position the segment is in the checkpoint times
20     */
21     public ResultsMountainTimeCompatoror(int pos) {
22         // set class attrivutes
23         this.pos = pos;
24     }
25
26     /**
27     * Compare 2 reuslts using {@code LocalTime.compareTo}
28     *
29     * @param result1 first result to compare
30     * @param result2 second result to copmare
31     * @return the value of result1 - result2
32     */
33     @Override
34     public int compare(Results result1, Results result2) {
35         // compare 2 results at a cetrain position using LocalTime.compareTo
36         return result1.getTimes()[pos].compareTo(result2.getTimes()[pos]);
37     }
38 }

```

7 ResultsSegmentTimeCompatitor.java

```

1 package cycling;
2
3 import java.util.Comparator;
4
5 /**
6  * Results class compatotor.
7  * Used to compare 2 results based on the time to segment
8  *
9  * @author Ethan Hofton
10 * @author Jon Tao
11 * @version 1.0
12 */
13 public class ResultsSegmentTimeCompatitor implements Comparator<Results> {
14
15     private int pos;
16
17     /**
18     * Constructor for class
19     *
20     * @param pos the position the segment is in the checkpoint times
21     */
22     public ResultsSegmentTimeCompatitor(int pos) {
23         // set class attrivutes
24         this.pos = pos;
25     }
26
27     /**
28     * Compare 2 reuslts using {@code LocalTime.compareTo}
29     *
30     * @param result1 first result to compare
31     * @param result2 second result to copmare

```

```

32     * @return the value of result1 - result2
33     */
34     @Override
35     public int compare(Results result1, Results result2) {
36         // compare 2 results at a cetrain position using LocalTime.compareTo and
37         // Result.calculateTimeToSegment
38         return result1.calculateTimeToSegment(pos).compareTo(result2.calculateTimeToSegment(pos));
39     }

```

8 Rider.java

```

1  package cycling;
2
3  import java.io.Serializable;
4
5  /**
6   * The rider class. Stores rider id and other data relevent to the rider
7   *
8   * @author Ethan Hofton
9   * @author Jon Tao
10  * @version 1.0
11  */
12  public class Rider implements Serializable {
13
14      private static int riderCount = 0;
15
16      private int riderId;
17      private String riderName;
18      private int riderYearOfBirth;
19      private Team riderTeam;
20
21      /**
22       * The rider constructor
23       *
24       * @param team the team the rider belongs to
25       * @param riderName the name of the rider
26       * @param riderYearOfBirth the year of bith of the rider
27       * @see cycling.Team
28       */
29      public Rider(Team team, String riderName, int riderYearOfBirth) {
30          // set rider id and increment rider count
31          this.riderId = riderCount++;
32
33          // set rider class attributes
34          this.riderName = riderName;
35          this.riderYearOfBirth = riderYearOfBirth;
36          this.riderTeam = team;
37      }
38
39      /**
40       * Getter for {@code this.riderId}
41       *
42       * @return the id of the rider

```

```

43     */
44     public int getRiderId() {
45         return riderId;
46     }
47
48     /**
49      * Getter for {@code this.riderTeam}
50      *
51      * @return the team of the rider
52      * @see cycling.Team
53      */
54     public Team getTeam() {
55         return riderTeam;
56     }
57
58     /**
59      * Getter for {@code this.riderName}
60      *
61      * @return the name of the rider
62      */
63     public String getRiderName() {
64         return riderName;
65     }
66
67     /**
68      * Getter for {@code this.riderYearOfBirth}
69      *
70      * @return the year of birth of the rider
71      */
72     public int getRiderYearOfBirth() {
73         return riderYearOfBirth;
74     }
75
76     /**
77      * sums the rank points and sprint points for a rider and given stage
78      *
79      * @param stage the stage the rider accumulated points for
80      * @param rank the rank the rider got
81      * @return the total points accumulated for the given stage
82      */
83     public int getPointsInStage(Stage stage, int rank) {
84         // initialize the points
85         int points = 0;
86
87         // add the rank points
88         points += stage.pointsForRank(rank);
89
90         // add the intermediate sprint points
91         points += stage.pointsForIntermediateSprints(this);
92
93         return points;
94     }
95
96     /**
97      * returns the mountain points for that rider in the given stage

```

```

98     *
99     * @param stage the stage the rider accumulated points for
100    * @return the total points accumulated for the given stage
101    */
102    public int getMountainPointsInStage(Stage stage) {
103        // return the mountain points for this rider
104        return stage.pointsForMountainClassification(this);
105    }
106
107    /**
108     * Rest the static counter to set the ids
109     */
110    public static void resetCounter() {
111        // reset static rider counter
112        riderCount = 0;
113    }
114 }

```

9 Segment.java

```

1  package cycling;
2
3  import java.io.Serializable;
4
5  /**
6   * Segment class. Stores information common to both
7   * climb segments and sprint segments
8   *
9   * @author Ethan Hofton
10  * @author Jon Tao
11  * @version 1.0
12  */
13  public class Segment implements Serializable {
14      protected static int segmentCount;
15      protected int segmentId;
16      protected Stage stage;
17      protected double location;
18      protected SegmentType type;
19
20      /**
21       * Segment constructor
22       *
23       * @param stage the stage the segment belongs to
24       * @param location the location of the segment within the stage
25       * @param type the type of the segment
26       * @see cycling.Stage
27       * @see cycling.SegmentType
28       */
29      public Segment(Stage stage, double location, SegmentType type) {
30          // set segment id and increment segment count
31          this.segmentId = segmentCount++;
32
33          // set the class attributes
34          this.stage = stage;

```



```

35     this.location = location;
36     this.type = type;
37 }
38
39 /**
40  * Getter for {@code this.segmentId}
41  *
42  * @return the id for the segment
43  */
44 public int getSegmentId() {
45     return segmentId;
46 }
47
48 /**
49  * Getter for {@code this.stage}
50  *
51  * @return the stage the segment belongs to
52  * @see cycling.Stage
53  */
54 public Stage getStage() {
55     return stage;
56 }
57
58 /**
59  * Getter for {@code this.location}
60  *
61  * @return location of the segment within the stage
62  */
63 public double getLocation() {
64     return location;
65 }
66
67 /**
68  * Getter for {@code this.type}
69  *
70  * @return the type of segment
71  * @see cycling.SegmentType
72  */
73 public SegmentType getType() {
74     return type;
75 }
76
77 /**
78  * Check wither the segment is a climb or not
79  *
80  * @return boolean of wether the segment is a climb or not
81  */
82 boolean isClimb() {
83     return !isSprint();
84 }
85
86 /**
87  * Check wither the segment is a sprint or not
88  *
89  * @return boolean of wether the segment is a sprint or not

```

```

90     */
91     boolean isSprint() {
92         return type == SegmentType.SPRINT;
93     }
94
95     /**
96      * Rest the static counter to set the ids
97      */
98     public static void resetCounter() {
99         // reset the static segment counter
100         segmentCount = 0;
101     }
102 }

```

10 SprintSegment.java

```

1  package cycling;
2
3  /**
4   * extends {@link cycling.Segment}
5   * A special case of {@code Segment} where the type is {@code SegmentType.SPRINT}
6   *
7   * @author Ethan Hofton
8   * @author Jon Tao
9   * @version 1.0
10  * @see cycling.Segment
11  *
12  */
13  public class SprintSegment extends Segment {
14
15      /**
16       * SprintSegment Constructor. call super construoer explicitly passing {@code type} as {@code
17         SegmentType.SPRINT}
18       *
19       * @param stage the stage the segment belongs to
20       * @param location the location of the segment in the stage
21       * @see cycling.Stage
22       */
23      public SprintSegment(Stage stage, double location) {
24          // call segment constructor
25          super(stage, location, SegmentType.SPRINT);
26      }
27
28      /**
29       * Override of {@link cycling.Segment.isClimb} where the value is explicitly defined
30       *
31       * @return false
32       * @see cycling.Segment.isClimb
33       */
34      @Override
35      boolean isClimb() {
36          return false;
37      }
38  }

```

```

38  /**
39   * Override of {@link cycling.Segment.isSprint} where the value is explicitly defined
40   *
41   * @return true
42   * @see cycling.Segment.isSprint
43   */
44  @Override
45  boolean isSprint() {
46      return true;
47  }
48  }

```

11 Stage.java

```

1  package cycling;
2
3  import java.io.Serializable;
4  import java.time.LocalDateTime;
5  import java.util.ArrayList;
6  import java.util.Arrays;
7
8  /**
9   * Stage class to store stage id and data related to stage
10   *
11   * @author Ethan Hofton
12   * @author Jon Tao
13   * @version 1.0
14   */
15  public class Stage implements Serializable {
16      private static int stageCount = 0;
17      private int stageId;
18      private Race race;
19      private String stageName;
20      private String description;
21      private double length; // in KM
22      private LocalDateTime startTime;
23      private StageType type;
24      private StageState stageState;
25
26      private ArrayList<Segment> segments;
27      private ArrayList<Results> results;
28
29      /**
30       * Stage contrustor
31       *
32       * @param race the race the stage belongs to
33       * @param stageName the name of the stage
34       * @param description the stage description
35       * @param length the length of the stage
36       * @param startTime the time the stage will begin
37       * @param type the type of stage
38       * @see cycling.Race
39       * @see cycling.StageType
40       */

```

```

41 public Stage(Race race, String stageName, String description, double length, LocalDateTime startTime,
42             StageType type) {
43     // set the stage id and increment the static stage counter
44     this.stageId = stageCount++;
45
46     // set class attributes
47     this.race = race;
48     this.stageName = stageName;
49     this.description = description;
50     this.length = length;
51     this.startTime = startTime;
52     this.type = type;
53     this.stageState = StageState.STAGE_PREPERATION;
54
55     // initialize the class array lists
56     this.segments = new ArrayList<>();
57     this.results = new ArrayList<>();
58 }
59
60 /**
61  * Getter for {@code this.stageId}
62  *
63  * @return the id of the stage
64  */
65 public int getStageId() {
66     return stageId;
67 }
68
69 /**
70  * Getter for {@code this.race}
71  *
72  * @return the race the stage belongs to
73  * @see cycling.Race
74  */
75 public Race getRace() {
76     return race;
77 }
78
79 /**
80  * Getter for {@code this.stageName}
81  *
82  * @return the name of the stage
83  */
84 public String getStageName() {
85     return stageName;
86 }
87
88 /**
89  * Getter for {@code this.description}
90  *
91  * @return the description of the stage
92  */
93 public String getDescription() {
94     return description;
95 }

```

```

95
96 /**
97  * Getter for {@code this.length}
98  *
99  * @return the length of the stage
100  */
101 public double getLength() {
102     return length;
103 }
104
105 /**
106  * Getter for {@code this.startTime}
107  *
108  * @return the time the stage will begin
109  */
110 public LocalDateTime getStartTime() {
111     return startTime;
112 }
113
114 /**
115  * Getter for {@code this.type}
116  *
117  * @return the type of the stage
118  */
119 public StageType getType() {
120     return type;
121 }
122
123 /**
124  * Getter for {@code this.segments}
125  *
126  * @return a list of the segments the stage has
127  * @see cycling.Segment
128  */
129 public ArrayList<Segment> getSegments() {
130     return this.segments;
131 }
132
133 /**
134  * Add a segment to the stage
135  *
136  * @param segment the segment to be added to the stage
137  * @see cycling.Segment
138  */
139 public void addSegment(Segment segment) {
140     // add segment to segment array list
141     this.segments.add(segment);
142 }
143
144 /**
145  * Remove a segment from the stage
146  *
147  * @param segment the segment to be removed from the stage
148  * @see cycling.Segment
149  */

```

```

150 public void removeSegment(Segment segment) {
151     // remove segment from segment array list
152     this.segments.remove(segment);
153 }
154
155 /**
156  * Getter for {@code this.stageState}
157  *
158  * @return the state of the stage
159  * @see cycling.StageState
160  */
161 public StageState getStageState() {
162     return this.stageState;
163 }
164
165 /**
166  * Change the state of the stage to waiting for results.
167  * Function can only be called once
168  *
169  * @throws InvalidStageStateException if the function is called twice
170  */
171 public void concludeStagePreparation() throws InvalidStageStateException {
172     // conclude stage preparation
173     // if stage has already been concluded throw error
174     // check if stage type is already waiting for results
175     if (this.stageState == StageState.WAITING_FOR_RESULTS) {
176         // throw InvalidStageStateException if stage state is already waiting for results
177         throw new InvalidStageStateException("Stage is already waiting for results");
178     }
179
180     // set the stage state to waiting for results
181     this.stageState = StageState.WAITING_FOR_RESULTS;
182 }
183
184 /**
185  * add result to stage
186  *
187  * @param result the result to be added
188  * @see cycling.Results
189  */
190 public void addResults(Results result) {
191     // add result to results array list
192     results.add(result);
193 }
194
195 /**
196  * getter for {@code this.results}
197  *
198  * @return a list of results the stage contains
199  * @see cycling.Results
200  */
201 public ArrayList<Results> getResults() {
202     return results;
203 }
204

```

```

205  /**
206   * remove result from stage
207   *
208   * @param result result to be removed
209   * @throws IDNotRecognisedException if the result is not in the race
210   * @see cycling.Results
211   */
212  public void removeResults(Results result) throws IDNotRecognisedException {
213      // remove result from result array list
214      // check if result array contains result
215      if (!results.contains(result)) {
216          // if the result array does not contain result, throw an IDNotRecognisedException
217          throw new IDNotRecognisedException("result does not exist in race with Id '"+stageId+"'");
218      }
219      // remove result
220      results.remove(result);
221  }
222
223  /**
224   * Calculate the number of points for position in stage.
225   * Segments are not considered in this funciton
226   *
227   * @param rank position rider finished in segment
228   * @return points the rider gained for finishing position in stage
229   */
230  public int pointsForRank(int rank) {
231
232      // return the points aquired for a riders given rank
233      // switch the stage type and return the appriotate points based on the rank and stage type
234      switch (this.type) {
235          case FLAT:
236              return pointsForFlat(rank);
237          case HIGH_MOUNTAIN:
238              return pointsForHMTTIT(rank);
239          case MEDIUM_MOUNTAIN:
240              return pointsForMediumMountain(rank);
241          case TT:
242              return pointsForHMTTIT(rank);
243          default:
244              // if the stage type is not as above, no points were aquired and return zero
245              return 0;
246      }
247  }
248
249  /**
250   * calculate the points for the intermiedete sprints in stage for a given rider.
251   * Not including mountain points
252   *
253   * @param rider rider to calulcate points for
254   * @return the points the rider accumulated over the stage
255   */
256  public int pointsForIntermediateSprints(Rider rider) {
257      // initialize points to zero
258      int points = 0;
259

```

```

260 // loop through all the segments in the stage
261 for (int i = 0; i < segments.size(); i++) {
262     // check if the segment is a sprint segment
263     if (segments.get(i).isSprint()) {
264         // create an array for all the results
265         Results[] rankedResults = new Results[getResults().size()];
266
267         // loop through all the results in the stage
268         for (int x = 0; x < rankedResults.length; x++) {
269             // add the result to the results array
270             rankedResults[x] = getResults().get(x);
271         }
272
273         // sort the results array based on the elapsed time to the point
274         // sort using custom comparator ResultsSegmentTimeComparator
275         Arrays.sort(rankedResults, new ResultsSegmentTimeComparator(i+1));
276
277         // loop through all the RANKED results
278         for (int x = 0; x < rankedResults.length; x++) {
279             // if the result belongs to the rider
280             if (rankedResults[x].getRider() == rider) {
281                 // add the intermediat points to the sum
282                 points += pointsForHMTTIT(x+1);
283                 continue;
284             }
285         }
286     }
287 }
288
289 // return the points aquered
290 return points;
291 }
292
293 /**
294  * Calculate the points for the mountain segments
295  *
296  * @param rider the rider to calculate the points for
297  * @return the points the rider accumulated over the stage
298  */
299 public int pointsForMountainClassification(Rider rider) {
300
301     // initialize points to zero
302     int points = 0;
303
304     // loop through all the segments in the stage
305     for (int i = 0; i < segments.size(); i++) {
306         // check if the segment is a climb
307         if (segments.get(i).isClimb()) {
308             // if the segment is a climb, it is safe to upcase the segment to a climbsegment
309             ClimbSegment segment = (ClimbSegment)segments.get(i);
310
311             // create a new array to store all the results in the stage
312             Results[] rankedResults = new Results[getResults().size()];
313
314             // loop through each result in the stage

```



```

315         for (int x = 0; x < rankedResults.length; x++) {
316             // add the result to the list of results
317             rankedResults[x] = getResults().get(x);
318         }
319
320         // sore the ranked results using custom compatirot ResultsMountainTimeCompatoror to sort
321         // based of
322         // of the time at which the riders reached the segmenent finish
323         Arrays.sort(rankedResults, new ResultsMountainTimeCompatoror(i+1));
324
325         // loop through all the ranked results
326         for (int x = 0; x < rankedResults.length; x++) {
327             // if the result belongs to the rider
328             if (rankedResults[x].getRider() == rider) {
329                 // add the mountian points for that segment to the riders sum
330                 points += segment.mountainPoints(x+1);
331                 continue;
332             }
333         }
334     }
335
336     // return the points aquered
337     return points;
338 }
339
340 /**
341  * Calculates the points for flat finish stage
342  * Data from Figure 1 in coursework spesification
343  *
344  * @param rank the rank of the rider
345  * @return the points the rider gets for the given rank
346  */
347 static public int pointsForFlat(int rank) {
348     // return the points aquered for the rank and if the stage type is flat
349     // add data is taken from Figure 1 in coursework spec
350     switch (rank) {
351         case 1:
352             return 50;
353         case 2:
354             return 30;
355         case 3:
356             return 20;
357         case 4:
358             return 18;
359         case 5:
360             return 16;
361         case 6:
362             return 14;
363         case 7:
364             return 12;
365         case 8:
366             return 10;
367         case 9:
368             return 8;

```

```

369         case 10:
370             return 7;
371         case 11:
372             return 6;
373         case 12:
374             return 5;
375         case 13:
376             return 4;
377         case 14:
378             return 3;
379         case 15:
380             return 2;
381         default:
382             return 0;
383     }
384 }
385
386 /**
387  * Calculates the points for Medium Mountain finish stage
388  * Data from Figure 1 in coursework spesification
389  *
390  * @param rank the rank of the rider
391  * @return the points the rider gets for the given rank
392  */
393 static public int pointsForMediumMountain(int rank) {
394     // return the points aquered for the rank and if the stage type is medium
395     // add data is taken from Figure 1 in coursework spec
396     switch (rank) {
397         case 1:
398             return 30;
399         case 2:
400             return 25;
401         case 3:
402             return 22;
403         case 4:
404             return 19;
405         case 5:
406             return 17;
407         case 6:
408             return 15;
409         case 7:
410             return 13;
411         case 8:
412             return 11;
413         case 9:
414             return 9;
415         case 10:
416             return 7;
417         case 11:
418             return 6;
419         case 12:
420             return 5;
421         case 13:
422             return 4;
423         case 14:

```

```

424         return 3;
425     case 15:
426         return 2;
427     default:
428         return 0;
429     }
430 }
431
432 /**
433  * Calculates the points for High Mountain, Time Trail, Individual Trial stage
434  * Data from Figure 1 in coursework spesification
435  *
436  * @param rank the rank of the rider
437  * @return the points the rider gets for the given rank
438  */
439 static public int pointsForHMTTIT(int rank) {
440     // return the points aquered for the rank and if the stage type is high mountain, time trial or
441     // individual
442     // trail
443     // add data is taken from Figure 1 in coursework spec
444     switch (rank) {
445     case 1:
446         return 20;
447     case 2:
448         return 17;
449     case 3:
450         return 15;
451     case 4:
452         return 13;
453     case 5:
454         return 11;
455     case 6:
456         return 10;
457     case 7:
458         return 9;
459     case 8:
460         return 8;
461     case 9:
462         return 7;
463     case 10:
464         return 6;
465     case 11:
466         return 5;
467     case 12:
468         return 4;
469     case 13:
470         return 3;
471     case 14:
472         return 2;
473     case 15:
474         return 1;
475     default:
476         return 0;
477     }
}

```

```

478
479     /**
480      * Rest the static counter to set the ids
481      */
482     public static void resetCounter() {
483         // reset static counter of stage count
484         stageCount = 0;
485     }
486
487 }

```

12 StageState.java

```

1  package cycling;
2
3  /**
4   * This enum is used to represent the state of the stage.
5   *
6   * @author Ethan Hofton
7   * @author Jon Tao
8   * @version 1.0
9   *
10  */
11  public enum StageState {
12
13      /**
14       * Before the stage has concluded its preperation
15       */
16      STAGE_PREPERATION,
17
18      /**
19       * Stage is waiting for results to be entered
20       */
21      WAITING_FOR_RESULTS;
22  }

```

13 Team.java

```

1  package cycling;
2
3  import java.io.Serializable;
4  import java.util.ArrayList;
5
6
7  /**
8   * Team class stores team ID and data relavent to team
9   *
10  * @author Ethan Hofton
11  * @author Jon Tao
12  * @version 1.0
13  *
14  */
15  public class Team implements Serializable {
16

```

```

17 private static int teamCount = 0;
18
19 private ArrayList<Rider> teamRiders;
20
21 private int teamId;
22 private String teamName;
23 private String teamDescription;
24
25 /**
26  * Team construtor. initialises team ID
27  *
28  * @param teamName the name of the team
29  * @param teamDescription the team description
30  */
31 Team(String teamName, String teamDescription) {
32     // initialize team riders array list
33     this.teamRiders = new ArrayList<>();
34
35     // set team id and increment static team counter
36     this.teamId = teamCount++;
37
38     // set class attributes
39     this.teamName = teamName;
40     this.teamDescription = teamDescription;
41 }
42
43 /**
44  * Getter for {@code this.teamId}
45  *
46  * @return the id of the team
47  */
48 public int getTeamId() {
49     return teamId;
50 }
51
52 /**
53  * Getter for {@code this.teamName}
54  *
55  * @return the name of the team
56  */
57 public String getTeamName() {
58     return teamName;
59 }
60
61 /**
62  * Getter for {@code this.teamDescription}
63  *
64  * @return the description of the team
65  */
66 public String getTeamDescription() {
67     return teamDescription;
68 }
69
70 /**
71  * Getter for {@code this.teamRiders}

```

```

72     *
73     * @return an array of the riders on the team
74     * @see cycling.Rider
75     */
76     public ArrayList<Rider> getRiders() {
77         return teamRiders;
78     }
79
80     /**
81     * add rider to team
82     *
83     * @param newRider the rider to add to the team
84     * @see cycling.Rider
85     */
86     public void addRider(Rider newRider) {
87         // add rider to arraylist
88         teamRiders.add(newRider);
89     }
90
91     /**
92     * remove a rider from the team
93     *
94     * @param riderToRemove the rider to remove from the team
95     * @throws IDNotRecognisedException if the rider is not in the team
96     * @see cycling.Rider
97     */
98     public void removeRider(Rider riderToRemove) throws IDNotRecognisedException {
99         // findRider throws IDNotRecognisedException
100        // find rider position
101        int riderPosition = findRider(riderToRemove);
102
103        // remove rider at that index
104        teamRiders.remove(riderPosition);
105    }
106
107
108    /**
109    * return the index of the rider in {@code this.teamRiders}
110    *
111    * @param riderToFind the rider to find
112    * @return the index of the rider in the rider array
113    * @throws IDNotRecognisedException if the rider is not in the team
114    * @see cycling.Rider
115    */
116    public int findRider(Rider riderToFind) throws IDNotRecognisedException {
117
118        // loops through all team riders
119        // checks id against given rider id
120        // if ids match, return the position, id not throw exception
121        for (int i = 0; i < teamRiders.size(); i++) {
122            if (teamRiders.get(i).getRiderId() == riderToFind.getRiderId()) {
123                return i;
124            }
125        }
126    }

```

```

127         throw new IDNotRecognisedException("Rider id not found");
128     }
129
130     /**
131     * Check if the rider is in the team
132     *
133     * @param riderToFind the rider to find
134     * @return boolean whether the rider is in the team
135     * @see cycling.Rider
136     */
137     public boolean containsRider(Rider riderToFind) {
138         // try find the rider using findRider function
139         // if the function throws an IDNotRecognisedException exception,
140         // the rider does not exists and return false,
141         // otherwise return true
142         try {
143             findRider(riderToFind);
144         } catch (IDNotRecognisedException e) {
145             return false;
146         }
147
148         return true;
149     }
150
151     /**
152     * Reset the static counter to set the ids
153     */
154     public static void resetCounter() {
155         // reset team counter to zero
156         teamCount = 0;
157     }
158 }

```