

# 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.ObjectInputStream;
7 import java.io.ObjectOutputStream;
8 import java.time.LocalDateTime;
9 import java.time.LocalDate;
10 import java.util.ArrayList;
11 import java.util.Arrays;
12 import java.util.Comparator;
13 import java.util.HashMap;
14 import java.util.Map;
15
16 /**
17  * Cycling Portal implements CyclingPortalInterface class
18  *
19  * @author Ethan Hofton
20  * @author Jon Tao
21  * @version 1.0
22  */
23 public class CyclingPortal implements CyclingPortalInterface {
24
25     private ArrayList<Team> teams;
26     private ArrayList<Race> races;
27
28     /**
29      * CyclingPortal constructor initialises teams and races array list
30      *
31      * @return nothing
32      */
33     public CyclingPortal() {
34         // constructor to init lists
35         teams = new ArrayList<>();
36         races = new ArrayList<>();
37     }
38
39     private Team findTeam(int teamID) throws IDNotRecognisedException {
40         // check if the list 'teams' has teamID
41         // O(n)
42
43         // loop through teams list and check the team class's id
44         // against the given id teamID
45         for (int i = 0; i < teams.size(); i++) {
46             if (teams.get(i).getTeamId() == teamID) {
47                 return teams.get(i);
48             }
49         }
50
51         // throw IDNotRecognisedException if not found
52         throw new IDNotRecognisedException("Team Id '" + teamID + "' not found");
53     }
54 }
```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

540     */
541     @Override
542     public int[] getTeamRiders(int teamId) throws IDNotRecognisedException {
543         // find team in portal
544         // Throws IDNotRecognisedException
545         Team team = findTeam(teamId);
546         int teamRiders[] = new int[team.getRiders().size()];
547
548         for (int i = 0; i < team.getRiders().size(); i++) {
549             teamRiders[i] = team.getRiders().get(i).getRiderId();
550         }
551
552         return teamRiders;
553     }
554
555     /**
556     * {@inheritDoc}
557     */
558     @Override
559     public int createRider(int teamID, String name, int yearOfBirth) throws IDNotRecognisedException,
560         IllegalArgumentException {
561         // check year and name
562         if (name == null || yearOfBirth < 1900) {
563             throw new IllegalArgumentException("name cannot be null or year less then 1900");
564         }
565
566         // throws IDNotRecognisedException
567         Team ridersTeam = findTeam(teamID);
568         Rider newRider = new Rider(ridersTeam, name, yearOfBirth);
569
570         ridersTeam.addRider(newRider);
571
572         return newRider.getRiderId();
573     }
574
575     /**
576     * {@inheritDoc}
577     */
578     @Override
579     public void removeRider(int riderId) throws IDNotRecognisedException {
580         // throws IDNotRecognisedException
581         Rider rider = findRider(riderId);
582         rider.getTeam().removeRider(rider);
583         // TODO remove all race results (not implamented race)
584     }
585
586     /**
587     * {@inheritDoc}
588     */
589     @Override
590     public void registerRiderResultsInStage(int stageId, int riderId, LocalTime... checkpoints)
591         throws IDNotRecognisedException, DuplicatedResultException, InvalidCheckpointsException,
592         InvalidStageStateException {
593         // throws IDNotRecognisedException

```

```

594     Rider rider = findRider(riderId);
595
596     // throws IDNotRecognisedException
597     Stage stage = findStage(stageId);
598
599     // check rider does not have duplicate result
600     for (int i = 0; i < stage.getResults().size(); i++) {
601         if (stage.getResults().get(i).getRider() == rider) {
602             // duplicate found
603             throw new DuplicatedResultException("Stage already has results for rider");
604         }
605     }
606
607     // check length of checkpoints is equal to n+2
608     if (checkpoints.length != stage.getSegments().size() + 2) {
609         throw new InvalidCheckpointsException("length of checkpoints is invalid");
610     }
611
612     // check if stage is "waiting for results"
613     if (stage.getStageState() != StageState.WAITING_FOR_RESULTS) {
614         throw new InvalidStageStateException("Invalid stage state");
615     }
616
617     Results result = new Results(stage, rider, checkpoints);
618
619     stage.addResults(result);
620 }
621
622 /**
623  * {@inheritDoc}
624  */
625 @Override
626 public LocalTime[] getRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException {
627
628     // throws IDNotRecognisedException
629     Stage stage = findStage(stageId);
630
631     // throws IDNotRecognisedException
632     Rider rider = findRider(riderId);
633
634     Results riderResult = null;
635
636     // find rider results
637     for (int i = 0; i < stage.getResults().size(); i++) {
638         if (rider == stage.getResults().get(i).getRider()) {
639             riderResult = stage.getResults().get(i);
640         }
641     }
642
643     if (riderResult == null) {
644         return new LocalTime[0];
645     }
646
647     LocalTime[] riderResults = new LocalTime[riderResult.getTimes().length + 1];
648     for (int i = 0; i < riderResult.getTimes().length; i++) {

```

```

649         riderResults[i] = riderResult.getTimes()[i];
650     }
651
652     riderResults[riderResult.getTimes().length] = riderResult.calculateElapsedTime();
653
654     return riderResults;
655 }
656
657 /**
658  * {@inheritDoc}
659  */
660 @Override
661 public LocalTime getRiderAdjustedElapsedTimeInStage(int stageId, int riderId) throws
        IDNotRecognisedException {
662
663     // throws IDNotRecognisedException
664     Stage stage = findStage(stageId);
665
666     // throws IDNotRecognisedException
667     Rider rider = findRider(riderId);
668
669     Results riderResult = null;
670
671     // find rider results
672     for (int i = 0; i < stage.getResults().size(); i++) {
673         if (rider == stage.getResults().get(i).getRider()) {
674             riderResult = stage.getResults().get(i);
675         }
676     }
677
678     if (riderResult == null) {
679         return null;
680     }
681
682     return riderResult.calculateAdjustedElapsedTime();
683 }
684
685 /**
686  * {@inheritDoc}
687  */
688 @Override
689 public void deleteRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException {
690
691     // throws IDNotRecognisedException
692     Stage stage = findStage(stageId);
693
694     // throws IDNotRecognisedException
695     Rider rider = findRider(riderId);
696
697     Results riderResult = null;
698
699     // find rider results
700     for (int i = 0; i < stage.getResults().size(); i++) {
701         if (rider == stage.getResults().get(i).getRider()) {
702             riderResult = stage.getResults().get(i);

```

```

703     }
704 }
705
706 if (riderResult == null) {
707     return;
708 }
709
710 // remove rider result from stage
711 stage.removeResults(riderResult);
712 }
713
714 /**
715  * {@inheritDoc}
716  */
717 @Override
718 public int[] getRidersRankInStage(int stageId) throws IDNotRecognisedException {
719
720     // throws IDNotRecognisedException
721     Stage stage = findStage(stageId);
722
723     Results[] rankedResults = new Results[stage.getResults().size()];
724     for (int i = 0; i < rankedResults.length; i++) {
725         rankedResults[i] = stage.getResults().get(i);
726     }
727
728     Arrays.sort(rankedResults, new ResultsElapsedTimeComparator());
729
730     int[] riderRanks = new int[rankedResults.length];
731     for (int i = 0; i < riderRanks.length; i++) {
732         riderRanks[i] = rankedResults[i].getRider().getRiderId();
733     }
734
735     return riderRanks;
736 }
737
738 /**
739  * {@inheritDoc}
740  */
741 @Override
742 public LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId) throws IDNotRecognisedException {
743
744     // throws IDNotRecognisedException
745     Stage stage = findStage(stageId);
746
747     // throws IDNotRecognisedException
748     int[] ridersRanked = getRidersRankInStage(stageId);
749
750     LocalTime[] riderAdjustedElapsedTimes = new LocalTime[ridersRanked.length];
751
752     for (int i = 0; i < riderAdjustedElapsedTimes.length; i++) {
753         // get rider
754         Rider rider = findRider(ridersRanked[i]);
755         for (int x = 0; x < stage.getResults().size(); x++) {
756             if (stage.getResults().get(x).getRider() == rider) {
757                 riderAdjustedElapsedTimes[i] = stage.getResults().get(x).calculateAdjustedElapsedTime();

```

```

758         continue;
759     }
760 }
761 }
762
763     return riderAdjustedElapsedTimes;
764 }
765
766 /**
767  * {@inheritDoc}
768  */
769 @Override
770 public int[] getRidersPointsInStage(int stageId) throws IDNotRecognisedException {
771     // throws IDNotRecognisedException
772     Stage stage = findStage(stageId);
773
774     // throws IDNotRecognisedException
775     int[] ridersRanked = getRidersRankInStage(stageId);
776
777     int[] riderPoints = new int[ridersRanked.length];
778
779     for (int i = 0; i < riderPoints.length; i++) {
780         // get rider
781         Rider rider = findRider(ridersRanked[i]);
782
783         riderPoints[i] = rider.getPointsInStage(stage, i+1);
784     }
785
786     return riderPoints;
787 }
788
789 /**
790  * {@inheritDoc}
791  */
792 @Override
793 public int[] getRidersMountainPointsInStage(int stageId) throws IDNotRecognisedException {
794     // throws IDNotRecognisedException
795     Stage stage = findStage(stageId);
796
797     // throws IDNotRecognisedException
798     int[] ridersRanked = getRidersRankInStage(stageId);
799
800     int[] riderPoints = new int[ridersRanked.length];
801
802     for (int i = 0; i < riderPoints.length; i++) {
803         // get rider
804         Rider rider = findRider(ridersRanked[i]);
805
806         riderPoints[i] = stage.pointsForMountainClassification(rider);
807     }
808
809     return riderPoints;
810 }
811
812 /**

```

```

813     * {@inheritDoc}
814     */
815     @Override
816     public void eraseCyclingPortal() {
817         // clear cycling portal
818         teams.clear();
819         races.clear();
820
821         // reset counters
822         Race.resetCounter();
823         Rider.resetCounter();
824         Segment.resetCounter();
825         Stage.resetCounter();
826         Team.resetCounter();
827     }
828
829     /**
830     * {@inheritDoc}
831     */
832     @Override
833     public void saveCyclingPortal(String filename) throws IOException {
834         ObjectOutputStream ostream = new ObjectOutputStream(new FileOutputStream(filename));
835         ostream.writeObject(this);
836         ostream.close();
837     }
838
839     /**
840     * {@inheritDoc}
841     */
842     @Override
843     public void loadCyclingPortal(String filename) throws IOException, ClassNotFoundException {
844         ObjectInputStream istream = new ObjectInputStream(new FileInputStream(filename));
845         Object portalObject = istream.readObject();
846         if (!(portalObject instanceof CyclingPortal)) {
847             // throw exception
848         }
849         CyclingPortal portal = (CyclingPortal)portalObject;
850         this.races = portal.races;
851         this.teams = portal.teams;
852         istream.close();
853     }
854
855     /**
856     * {@inheritDoc}
857     */
858     @Override
859     public void removeRaceByName(String name) throws NameNotRecognisedException {
860         Race race = null;
861
862         for (int i = 0; i < races.size(); i++) {
863             if (races.get(i).getName() == name) {
864                 race = races.get(i);
865             }
866         }
867

```



```

868     if (race == null) {
869         // throw NameNotRecognisedException
870         throw new NameNotRecognisedException("Race is not found with name " + name);
871     }
872
873     races.remove(race);
874
875     // since stage is stored in race and segments and results are stored in stage
876     // deleting the race will also delete segments, results and stage
877 }
878
879 /**
880  * {@inheritDoc}
881  */
882 @Override
883 public int[] getRidersGeneralClassificationRank(int raceId) throws IDNotRecognisedException {
884     // throws IDNotRecognisedException
885     Race race = findRace(raceId);
886
887     for (int i = 0; i < race.getStages().size(); i++) {
888         if (race.getStages().get(i).getResults().size() == 0) {
889             return new int[0];
890         }
891     }
892
893     ArrayList<Results> results = new ArrayList<>();
894     for (int i = 0; i < race.getStages().size(); i++) {
895         for (int x = 0; x < race.getStages().get(i).getResults().size(); x++) {
896             results.add(race.getStages().get(i).getResults().get(x));
897         }
898     }
899
900     Map<Rider, LocalTime> timesMap = new HashMap<Rider, LocalTime>();
901     for (int i = 0; i < results.size(); i++) {
902         Rider currentRider = results.get(i).getRider();
903         if (timesMap.containsKey(currentRider)) {
904             long nanos = results.get(i).calculateAdjustedElapsedTime().toNanoOfDay();
905             LocalTime newTime = timesMap.get(currentRider).plusNanos(nanos);
906             timesMap.replace(currentRider, newTime);
907         } else {
908             timesMap.put(currentRider, results.get(i).calculateAdjustedElapsedTime());
909         }
910     }
911
912     ArrayList<Map.Entry<Rider, LocalTime>> sorted = new ArrayList<>(timesMap.entrySet());
913     sorted.sort(new ResultsAdjustedElapsedTimeComparator());
914
915     int orderedRiderIds[] = new int[sorted.size()];
916     for (int i = 0; i < orderedRiderIds.length; i++) {
917         orderedRiderIds[i] = sorted.get(i).getKey().getRiderId();
918     }
919
920     return orderedRiderIds;
921 }
922

```

```

923  /**
924   * {@inheritDoc}
925   */
926  @Override
927  public LocalTime[] getGeneralClassificationTimesInRace(int raceId) throws IDNotRecognisedException {
928      // throws IDNotRecognisedException
929      Race race = findRace(raceId);
930
931      for (int i = 0; i < race.getStages().size(); i++) {
932          if (race.getStages().get(i).getResults().size() == 0) {
933              return new LocalTime[0];
934          }
935      }
936
937      ArrayList<Results> results = new ArrayList<>();
938      for (int i = 0; i < race.getStages().size(); i++) {
939          for (int x = 0; x < race.getStages().get(i).getResults().size(); x++) {
940              results.add(race.getStages().get(i).getResults().get(x));
941          }
942      }
943
944      Map<Rider, LocalTime> timesMap = new HashMap<Rider, LocalTime>();
945      for (int i = 0; i < results.size(); i++) {
946          Rider currentRider = results.get(i).getRider();
947          if (timesMap.containsKey(currentRider)) {
948              long nanos = results.get(i).calculateAdjustedElapsedTime().toNanoOfDay();
949              LocalTime newTime = timesMap.get(currentRider).plusNanos(nanos);
950              timesMap.replace(currentRider, newTime);
951          } else {
952              timesMap.put(currentRider, results.get(i).calculateAdjustedElapsedTime());
953          }
954      }
955
956      ArrayList<Map.Entry<Rider, LocalTime>> sorted = new ArrayList<>(timesMap.entrySet());
957      sorted.sort(new ResultsAdjustedElapsedTimeComparator());
958
959      LocalTime orderedTimes[] = new LocalTime[sorted.size()];
960      for (int i = 0; i < orderedTimes.length; i++) {
961          orderedTimes[i] = sorted.get(i).getValue();
962      }
963
964      return orderedTimes;
965  }
966
967  /**
968   * {@inheritDoc}
969   */
970  @Override
971  public int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException {
972      // throws IDNotRecognisedException
973      Race race = findRace(raceId);
974
975      for (int i = 0; i < race.getStages().size(); i++) {
976          if (race.getStages().get(i).getResults().size() == 0) {
977              return new int[0];

```

```

978     }
979 }
980
981 ArrayList<Results> results = new ArrayList<>();
982 for (int i = 0; i < race.getStages().size(); i++) {
983     for (int x = 0; x < race.getStages().get(i).getResults().size(); x++) {
984         results.add(race.getStages().get(i).getResults().get(x));
985     }
986 }
987
988 Map<Rider, LocalTime> timesMap = new HashMap<Rider, LocalTime>();
989 for (int i = 0; i < results.size(); i++) {
990     Rider currentRider = results.get(i).getRider();
991     if (timesMap.containsKey(currentRider)) {
992         long nanos = results.get(i).calculateAdjustedElapsedTime().toNanoOfDay();
993         LocalTime newTime = timesMap.get(currentRider).plusNanos(nanos);
994         timesMap.replace(currentRider, newTime);
995     } else {
996         timesMap.put(currentRider, results.get(i).calculateAdjustedElapsedTime());
997     }
998 }
999
1000 ArrayList<Map.Entry<Rider, LocalTime>> sorted = new ArrayList<>(timesMap.entrySet());
1001 sorted.sort(new ResultsAdjustedElapsedTimeComparitor());
1002
1003 int ridersPoints[] = new int[sorted.size()];
1004 for (int i = 0; i < ridersPoints.length; i++) {
1005     ridersPoints[i] = 0;
1006 }
1007
1008 // for each rider, find the total points in all stages
1009 for (int i = 0; i < race.getStages().size(); i++) {
1010     Stage currentStage = race.getStages().get(i);
1011     int ridersRanks[] = getRidersRankInStage(currentStage.getStageId());
1012
1013     for (int x = 0; x < ridersRanks.length; x++) {
1014         int id = ridersRanks[x];
1015         int rank = x + 1;
1016
1017         for (int y = 0; y < sorted.size(); y++) {
1018             if (id == sorted.get(y).getKey().getRiderId()) {
1019                 ridersPoints[y] += sorted.get(y).getKey().getPointsInStage(currentStage, rank);
1020             }
1021         }
1022     }
1023 }
1024
1025 return ridersPoints;
1026 }
1027
1028 /**
1029  * {@inheritDoc}
1030  */
1031 @Override
1032 public int[] getRidersMountainPointsInRace(int raceId) throws IDNotRecognisedException {

```

```

1033 // throws IDNotRecognisedException
1034 Race race = findRace(raceId);
1035
1036 for (int i = 0; i < race.getStages().size(); i++) {
1037     if (race.getStages().get(i).getResults().size() == 0) {
1038         return new int[0];
1039     }
1040 }
1041
1042 ArrayList<Results> results = new ArrayList<>();
1043 for (int i = 0; i < race.getStages().size(); i++) {
1044     for (int x = 0; x < race.getStages().get(i).getResults().size(); x++) {
1045         results.add(race.getStages().get(i).getResults().get(x));
1046     }
1047 }
1048
1049 Map<Rider, LocalTime> timesMap = new HashMap<Rider, LocalTime>();
1050 for (int i = 0; i < results.size(); i++) {
1051     Rider currentRider = results.get(i).getRider();
1052     if (timesMap.containsKey(currentRider)) {
1053         long nanos = results.get(i).calculateAdjustedElapsedTime().toNanoOfDay();
1054         LocalTime newTime = timesMap.get(currentRider).plusNanos(nanos);
1055         timesMap.replace(currentRider, newTime);
1056     } else {
1057         timesMap.put(currentRider, results.get(i).calculateAdjustedElapsedTime());
1058     }
1059 }
1060
1061 ArrayList<Map.Entry<Rider, LocalTime>> sorted = new ArrayList<>(timesMap.entrySet());
1062 sorted.sort(new ResultsAdjustedElapsedTimeCompatiror());
1063
1064 int ridersPoints[] = new int[sorted.size()];
1065 for (int i = 0; i < ridersPoints.length; i++) {
1066     ridersPoints[i] = 0;
1067 }
1068
1069 // for each rider, find the total points in all stages
1070 for (int i = 0; i < race.getStages().size(); i++) {
1071     Stage currentStage = race.getStages().get(i);
1072
1073     for (int y = 0; y < sorted.size(); y++) {
1074         ridersPoints[y] += sorted.get(y).getKey().getMountainPointsInStage(currentStage);
1075     }
1076 }
1077
1078 return ridersPoints;
1079 }
1080
1081 /**
1082  * {@inheritDoc}
1083  */
1084 @Override
1085 public int[] getRidersPointClassificationRank(int raceId) throws IDNotRecognisedException {
1086
1087     int riderIds[] = getRidersGeneralClassificationRank(raceId);

```

```

1088     int riderPoints[] = getRidersPointsInRace(raceId);
1089
1090     Map<Rider, Integer> pointsMap = new HashMap<Rider, Integer>();
1091     for (int i = 0; i < riderIds.length; i++) {
1092         Rider currentRider = findRider(riderIds[i]);
1093         pointsMap.put(currentRider, riderPoints[i]);
1094     }
1095
1096     ArrayList<Map.Entry<Rider, Integer>> sorted = new ArrayList<>(pointsMap.entrySet());
1097
1098     sorted.sort(Comparator.comparing(Map.Entry<Rider, Integer>::getValue, (p1, p2) -> {
1099         return p2 - p1;
1100     }));
1101
1102     int sortedIds[] = new int[riderIds.length];
1103     for (int i = 0; i < sortedIds.length; i++) {
1104         sortedIds[i] = sorted.get(i).getKey().getRiderId();
1105     }
1106
1107     return sortedIds;
1108 }
1109
1110 /**
1111  * {@inheritDoc}
1112  */
1113 @Override
1114 public int[] getRidersMountainPointClassificationRank(int raceId) throws IDNotRecognisedException {
1115     int riderIds[] = getRidersGeneralClassificationRank(raceId);
1116     int riderPoints[] = getRidersMountainPointsInRace(raceId);
1117
1118     Map<Rider, Integer> pointsMap = new HashMap<Rider, Integer>();
1119     for (int i = 0; i < riderIds.length; i++) {
1120         Rider currentRider = findRider(riderIds[i]);
1121         pointsMap.put(currentRider, riderPoints[i]);
1122     }
1123
1124     ArrayList<Map.Entry<Rider, Integer>> sorted = new ArrayList<>(pointsMap.entrySet());
1125
1126     sorted.sort(Comparator.comparing(Map.Entry<Rider, Integer>::getValue, (p1, p2) -> {
1127         return p2 - p1;
1128     }));
1129
1130     int sortedIds[] = new int[riderIds.length];
1131     for (int i = 0; i < sortedIds.length; i++) {
1132         sortedIds[i] = sorted.get(i).getKey().getRiderId();
1133     }
1134
1135     return sortedIds;
1136 }
1137 }
1138

```

## 2 ClimbSegment.java

```

1 package cycling;
2
3 /**
4  * Class for ClimbSegment extends {@link Segment}. Stores additional details required 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         super(stage, location, type);
28         this.averageGradient = averageGradient;
29         this.length = length;
30     }
31
32     /**
33      * Getter for {@code this.averageGradient}
34      *
35      * @return the average gradient
36      */
37     public Double getAverageGradient() {
38         return this.averageGradient;
39     }
40
41     /**
42      * Getter for {@code this.length}
43      *
44      * @return the average gradient
45      */
46     public Double getLength() {
47         return this.length;
48     }
49
50     /**
51      * Returns if the segment is a climb segment.
52      * Overrides {@link cycling.Segment.isClimb}
53      *
54      * @return whether the segment is a climb or not

```

```

55     */
56     @Override
57     boolean isClimb() {
58         return true;
59     }
60
61     /**
62     * Returns if the segment is a sprint segment.
63     * Overrides {@link cycling.Segment.isSprint}
64     *
65     * @return whether the segment is a sprint or not
66     */
67     @Override
68     boolean isSprint() {
69         return false;
70     }
71
72     /**
73     * Calculates the points mountain points for the segment
74     * Data from Figure 2 in coursework specification
75     *
76     * @param rank the rank of the rider
77     * @return the points the rider gets for the given rank
78     */
79     public int mountainPoints(int rank) {
80         switch (type) {
81             case C1:
82                 return pointsFor1C(rank);
83             case C2:
84                 return pointsFor2C(rank);
85             case C3:
86                 return pointsFor3C(rank);
87             case C4:
88                 return pointsFor4C(rank);
89             case HC:
90                 return pointsForHC(rank);
91             default:
92                 return 0;
93         }
94     }
95
96     /**
97     * Calculates the points for HC Mountain segment
98     * Data from Figure 2 in coursework specification
99     *
100    * @param rank the rank of the rider
101    * @return the points the rider gets for the given rank
102    */
103    static public int pointsForHC(int rank) {
104        switch (rank) {
105            case 1:
106                return 20;
107            case 2:
108                return 15;
109            case 3:

```

```

110         return 12;
111     case 4:
112         return 10;
113     case 5:
114         return 8;
115     case 6:
116         return 6;
117     case 7:
118         return 4;
119     case 8:
120         return 2;
121     default:
122         return 0;
123     }
124 }
125
126 /**
127  * Calculates the points for 1C Mountain segment
128  * Data from Figure 2 in coursework spesification
129  *
130  * @param rank the rank of the rider
131  * @return the points the rider gets for the given rank
132  */
133 static public int pointsFor1C(int rank) {
134     switch (rank) {
135     case 1:
136         return 10;
137     case 2:
138         return 8;
139     case 3:
140         return 6;
141     case 4:
142         return 4;
143     case 5:
144         return 2;
145     case 6:
146         return 1;
147     default:
148         return 0;
149     }
150 }
151
152 /**
153  * Calculates the points for 2C Mountain segment
154  * Data from Figure 2 in coursework spesification
155  *
156  * @param rank the rank of the rider
157  * @return the points the rider gets for the given rank
158  */
159 static public int pointsFor2C(int rank) {
160     switch (rank) {
161     case 1:
162         return 5;
163     case 2:
164         return 3;

```



```

165         case 3:
166             return 2;
167         case 4:
168             return 1;
169         default:
170             return 0;
171     }
172 }
173
174 /**
175  * Calculates the points for 3C Mountain segment
176  * Data from Figure 2 in coursework spesification
177  *
178  * @param rank the rank of the rider
179  * @return the points the rider gets for the given rank
180  */
181 static public int pointsFor3C(int rank) {
182     switch (rank) {
183         case 1:
184             return 2;
185         case 2:
186             return 1;
187         default:
188             return 0;
189     }
190 }
191
192 /**
193  * Calculates the points for 4C 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 pointsFor4C(int rank) {
200     switch (rank) {
201         case 1:
202             return 1;
203         default:
204             return 0;
205     }
206 }
207
208 /**
209  * Class toString method
210  *
211  * @return segment details in formatted string
212  */
213 @Override
214 public String toString() {
215     return
216         "ClimbSegment[stage="+stage+",location="+location+",type="+type+",averageGradient="+averageGradient+",length="+length+"]";
217 }

```

### 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 relevent
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         this.raceId = raceCount++;
30         this.name = name;
31         this.description = description;
32         this.stages = new ArrayList<>();
33     }
34
35     /**
36      * getter for {@code this.raceId}
37      *
38      * @return the id of the race
39      */
40     public int getRaceId() {
41         return raceId;
42     }
43
44     /**
45      * getter for {@code this.name}
46      *
47      * @return the name of the race
48      */
49     public String getName() {
50         return name;
51     }
52 }
```

```

53  /**
54   * getter for {@code this.description}
55   *
56   * @return the description of the race
57   */
58  public String getDescription() {
59      return description;
60  }
61
62  /**
63   * getter for {@code this.stages}
64   *
65   * @return the list of stages in the race
66   * @see cycling.Stage
67   */
68  public ArrayList<Stage> getStages() {
69      return stages;
70  }
71
72  /**
73   * adds a stage to the race
74   *
75   * @param stage the stage class to be added to the race
76   * @see cycling.Stage
77   */
78  public void addStage(Stage stage) {
79      stages.add(stage);
80  }
81
82  /**
83   * remove stage from race
84   *
85   * @param stage the stage class to be removed from the race
86   * @throws IDNotRecognisedException if the stage is not in the race
87   * @see cycling.Stage
88   */
89  public void removeStage(Stage stage) throws IDNotRecognisedException {
90      if (!stages.contains(stage)) {
91          throw new IDNotRecognisedException("stage does not exist in race with Id '"+raceId+"'");
92      }
93      stages.remove(stage);
94  }
95
96  /**
97   * check if the race contains a given stage
98   *
99   * @param stage the stage to be checked
100   * @return boolean wether the race contains the stage
101   * @see cycling.Stage
102   */
103  public boolean containsStage(Stage stage) {
104      return stages.contains(stage);
105  }
106
107  /**

```

```

108     * Rest the static counter to set the ids
109     */
110     public static void resetCounter() {
111         raceCount = 0;
112     }
113
114     /**
115     * Class toString
116     *
117     * @return a formatted string with class detials
118     */
119     public String toString() {
120         return "Race[raceId="+raceId+"name="+name+",description="+description+"]";
121     }
122 }

```

## 4 ResultsAdjustedElapsedTimeCompatiror.java

```

1  package cycling;
2
3  import java.time.LocalDateTime;
4  import java.util.Comparator;
5  import java.util.Map;
6
7  /**
8   * compatoror 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 ResultsAdjustedElapsedTimeCompatiror implements Comparator<Map.Entry<Rider,LocalTime>> {
15      /**
16       * Compare 2 reuslts using {@code LocalDateTime.compareTo}
17       *
18       * @param result1 first result to compare
19       * @param result2 second result to copmare
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          return result1.getValue().compareTo(result2.getValue());
25      }
26  }

```

## 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 results using {@code LocalTime.compareTo}
16      *
17      * @param result1 first result to compare
18      * @param result2 second result to compare
19      * @return the value of result1 - result2
20      */
21     @Override
22     public int compare(Results result1, Results result2) {
23         return result1.calculateElapsedTime().compareTo(result2.calculateElapsedTime());
24     }
25 }

```

## 6 ResultsMountainTimeComparator.java

```

1  package cycling;
2
3  import java.util.Comparator;
4
5  /**
6   * comparator for results class compare by elapsed time
7   *
8   * @author Ethan Hofton
9   * @author Jon Tao
10  * @version 1.0
11  */
12 public class ResultsMountainTimeComparator 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 ResultsMountainTimeComparator(int pos) {
22         this.pos = pos;
23     }
24
25     /**
26      * Compare 2 results using {@code LocalTime.compareTo}
27      *
28      * @param result1 first result to compare
29      * @param result2 second result to compare
30      * @return the value of result1 - result2
31      */
32     @Override
33     public int compare(Results result1, Results result2) {

```

```

34     return result1.getTimes()[pos].compareTo(result2.getTimes()[pos]);
35 }
36 }

```

## 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          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          return result1.calculateTimeToSegment(pos).compareTo(result2.calculateTimeToSegment(pos));
36      }
37 }

```

## 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 birth of the rider
27      * @see cycling.Team
28      */
29     public Rider(Team team, String riderName, int riderYearOfBirth) {
30         this.riderId = riderCount++;
31
32         this.riderName = riderName;
33         this.riderYearOfBirth = riderYearOfBirth;
34         this.riderTeam = team;
35     }
36
37     /**
38      * Getter for {@code this.riderId}
39      *
40      * @return the id of the rider
41      */
42     public int getRiderId() {
43         return riderId;
44     }
45
46     /**
47      * Getter for {@code this.riderTeam}
48      *
49      * @return the team of the rider
50      * @see cycling.Team
51      */
52     public Team getTeam() {
53         return riderTeam;
54     }
55
56     /**
57      * Getter for {@code this.riderName}
58      *
59      * @return the name of the rider
60      */
61     public String getRiderName() {
62         return riderName;
63     }

```

```

64
65 /**
66  * Getter for {@code this.riderYearOfBirth}
67  *
68  * @return the year of birth of the rider
69  */
70 public int getRiderYearOfBirth() {
71     return riderYearOfBirth;
72 }
73
74 /**
75  * sums the rank points and sprint points for a rider and given stage
76  *
77  * @param stage the stage the rider accumulated points for
78  * @param rank the rank the rider got
79  * @return the total points accumulated for the given stage
80  */
81 public int getPointsInStage(Stage stage, int rank) {
82     int points = 0;
83
84     points += stage.pointsForRank(rank);
85     points += stage.pointsForIntermediateSprints(this);
86
87     return points;
88 }
89
90 /**
91  * returns the mountain points for that rider in the given stage
92  *
93  * @param stage the stage the rider accumulated points for
94  * @return the total points accumulated for the given stage
95  */
96 public int getMountainPointsInStage(Stage stage) {
97     return stage.pointsForMountainClassification(this);
98 }
99
100 /**
101  * Rest the static counter to set the ids
102  */
103 public static void resetCounter() {
104     riderCount = 0;
105 }
106
107 /**
108  * To string method for the rider
109  *
110  * @return formatted string with rider information
111  */
112 public String toString() {
113     return
114         "Rider[riderId="+riderId+",riderTeam="+riderTeam+",riderName="+riderName+",riderYearOfBirth="+riderYearOfB
115 }

```



## 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         this.segmentId = segmentCount++;
31         this.stage = stage;
32         this.location = location;
33         this.type = type;
34     }
35
36     /**
37      * Getter for {@code this.segmentId}
38      *
39      * @return the id for the segment
40      */
41     public int getSegmentId() {
42         return segmentId;
43     }
44
45     /**
46      * Getter for {@code this.stage}
47      *
48      * @return the stage the segment belongs to
49      * @see cycling.Stage
50      */
51     public Stage getStage() {
52         return stage;
```

```

53     }
54
55     /**
56      * Getter for {@code this.location}
57      *
58      * @return location of the segment within the stage
59      */
60     public double getLocation() {
61         return location;
62     }
63
64     /**
65      * Getter for {@code this.type}
66      *
67      * @return the type of segment
68      * @see cycling.SegmentType
69      */
70     public SegmentType getType() {
71         return type;
72     }
73
74     /**
75      * Check wither the segment is a climb or not
76      *
77      * @return boolean of wether the segment is a climb or not
78      */
79     boolean isClimb() {
80         return !isSprint();
81     }
82
83     /**
84      * Check wither the segment is a sprint or not
85      *
86      * @return boolean of wether the segment is a sprint or not
87      */
88     boolean isSprint() {
89         return type == SegmentType.SPRINT;
90     }
91
92     /**
93      * Rest the static counter to set the ids
94      */
95     public static void resetCounter() {
96         segmentCount = 0;
97     }
98
99     /**
100      * toString of Segment
101      *
102      * @return formatted string containg relavent segment data
103      */
104     public String toString() {
105         return "Segment[stage="+stage+",location="+location+",type="+type+"]";
106     }
107 }

```

## 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 construator 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         super(stage, location, SegmentType.SPRINT);
25     }
26
27     /**
28      * Override of {@link cycling.Segment.isClimb} where the value is explicitly defined
29      *
30      * @return false
31      * @see cycling.Segment.isClimb
32      */
33     @Override
34     boolean isClimb() {
35         return false;
36     }
37
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
49     /**
50      * SprintSegment toString
51      *
52      * @return formatted string with relevent class data
```

```

52     */
53     @Override
54     public String toString() {
55         return "SprintSegment[stage="+stage+",location="+location+",type="+type+"]";
56     }
57 }

```

## 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          this.stageId = stageCount++;
44          this.race = race;
45          this.stageName = stageName;

```

```

45     this.description = description;
46     this.length = length;
47     this.startTime = startTime;
48     this.type = type;
49     this.stageState = StageState.STAGE_PREPERATION;
50
51     segments = new ArrayList<>();
52     this.results = new ArrayList<>();
53 }
54
55 /**
56  * Getter for {@code this.stageId}
57  *
58  * @return the id of the stage
59  */
60 public int getStageId() {
61     return stageId;
62 }
63
64 /**
65  * Getter for {@code this.race}
66  *
67  * @return the race the stage belongs to
68  * @see cycling.Race
69  */
70 public Race getRace() {
71     return race;
72 }
73
74 /**
75  * Getter for {@code this.stageName}
76  *
77  * @return the name of the stage
78  */
79 public String getStageName() {
80     return stageName;
81 }
82
83 /**
84  * Getter for {@code this.description}
85  *
86  * @return the description of the stage
87  */
88 public String getDescription() {
89     return description;
90 }
91
92 /**
93  * Getter for {@code this.length}
94  *
95  * @return the length of the stage
96  */
97 public double getLength() {
98     return length;
99 }

```

```

100
101 /**
102  * Getter for {@code this.startTime}
103  *
104  * @return the time the stage will begin
105  */
106 public LocalDateTime getStartTime() {
107     return startTime;
108 }
109
110 /**
111  * Getter for {@code this.type}
112  *
113  * @return the type of the stage
114  */
115 public StageType getType() {
116     return type;
117 }
118
119 /**
120  * Getter for {@code this.segments}
121  *
122  * @return a list of the segments the stage has
123  * @see cycling.Segment
124  */
125 public ArrayList<Segment> getSegments() {
126     return this.segments;
127 }
128
129 /**
130  * Add a segment to the stage
131  *
132  * @param segment the segment to be added to the stage
133  * @see cycling.Segment
134  */
135 public void addSegment(Segment segment) {
136     this.segments.add(segment);
137 }
138
139 /**
140  * Remove a segment from the stage
141  *
142  * @param segment the segment to be removed from the stage
143  * @see cycling.Segment
144  */
145 public void removeSegment(Segment segment) {
146     this.segments.remove(segment);
147 }
148
149 /**
150  * Getter for {@code this.stageState}
151  *
152  * @return the state of the stage
153  * @see cycling.StageState
154  */

```

```

155 public StageState getStageState() {
156     return this.stageState;
157 }
158
159 /**
160  * Chage the state of the stage to waiting for results.
161  * Function can only be called once
162  *
163  * @throws InvalidStageStateException if the function is called twice
164  */
165 public void concludeStagePreparation() throws InvalidStageStateException {
166     if (this.stageState == StageState.WAITING_FOR_RESULTS) {
167         throw new InvalidStageStateException("Stage is allready waiting for results");
168     }
169
170     this.stageState = StageState.WAITING_FOR_RESULTS;
171 }
172
173 /**
174  * add result to stage
175  *
176  * @param result the result to be added
177  * @see cycling.Results
178  */
179 public void addResults(Results result) {
180     results.add(result);
181 }
182
183 /**
184  * getter for {@code this.results}
185  *
186  * @return a list of results the stage contains
187  * @see cycling.Results
188  */
189 public ArrayList<Results> getResults() {
190     return results;
191 }
192
193 /**
194  * remove result from stage
195  *
196  * @param result result to be removed
197  * @throws IDNotRecognisedException if the result is not in the race
198  * @see cycling.Results
199  */
200 public void removeResults(Results result) throws IDNotRecognisedException {
201     if (!results.contains(result)) {
202         throw new IDNotRecognisedException("result does not exist in race with Id '"+stageId+"'");
203     }
204     results.remove(result);
205 }
206
207 /**
208  * Calculate the number of points for position in stage.
209  * Segments are not considered in this funciton

```

```

210  *
211  * @param rank position rider finished in segment
212  * @return points the rider gained for finishing position in stage
213  */
214  public int pointsForRank(int rank) {
215
216      switch (this.type) {
217          case FLAT:
218              return pointsForFlat(rank);
219          case HIGH_MOUNTAIN:
220              return pointsForHMTTIT(rank);
221          case MEDIUM_MOUNTAIN:
222              return pointsForMediumMountain(rank);
223          case TT:
224              return pointsForHMTTIT(rank);
225          default:
226              return 0;
227      }
228  }
229
230  /**
231   * calculate the points for the intermiedete sprints in stage for a given rider.
232   * Not including mountain points
233   *
234   * @param rider rider to calulcate points for
235   * @return the points the rider accumulated over the stage
236   */
237  public int pointsForIntermediateSprints(Rider rider) {
238      int points = 0;
239
240      for (int i = 0; i < segments.size(); i++) {
241          if (segments.get(i).isSprint()) {
242
243              Results[] rankedResults = new Results[getResults().size()];
244              for (int x = 0; x < rankedResults.length; x++) {
245                  rankedResults[x] = getResults().get(x);
246              }
247
248              Arrays.sort(rankedResults, new ResultsSegmentTimeCompatitor(i+1));
249
250              for (int x = 0; x < rankedResults.length; x++) {
251                  if (rankedResults[x].getRider() == rider) {
252                      points += pointsForHMTTIT(x+1);
253                      continue;
254                  }
255              }
256          }
257      }
258
259      return points;
260  }
261
262  /**
263   * Calculate the points for the mountain segments
264   *

```



```

265  * @param rider the rider to calculate the points for
266  * @return the points the rider accumulated over the stage
267  */
268  public int pointsForMountainClassification(Rider rider) {
269
270      int points = 0;
271
272      for (int i = 0; i < segments.size(); i++) {
273          if (segments.get(i).isClimb()) {
274              ClimbSegment segment = (ClimbSegment)segments.get(i);
275
276              // throws IDNotRecognisedException
277              Results[] rankedResults = new Results[getResults().size()];
278              for (int x = 0; x < rankedResults.length; x++) {
279                  rankedResults[x] = getResults().get(x);
280              }
281
282              Arrays.sort(rankedResults, new ResultsMountainTimeComparator(i+1));
283
284              for (int x = 0; x < rankedResults.length; x++) {
285                  if (rankedResults[x].getRider() == rider) {
286                      points += segment.mountainPoints(x+1);
287                      continue;
288                  }
289              }
290          }
291      }
292
293      return points;
294  }
295
296  /**
297   * Calculates the points for flat finish stage
298   * Data from Figure 1 in coursework specification
299   *
300   * @param rank the rank of the rider
301   * @return the points the rider gets for the given rank
302   */
303  static public int pointsForFlat(int rank) {
304      switch (rank) {
305          case 1:
306              return 50;
307          case 2:
308              return 30;
309          case 3:
310              return 20;
311          case 4:
312              return 18;
313          case 5:
314              return 16;
315          case 6:
316              return 14;
317          case 7:
318              return 12;
319          case 8:

```

```

320         return 10;
321     case 9:
322         return 8;
323     case 10:
324         return 7;
325     case 11:
326         return 6;
327     case 12:
328         return 5;
329     case 13:
330         return 4;
331     case 14:
332         return 3;
333     case 15:
334         return 2;
335     default:
336         return 0;
337     }
338 }
339
340 /**
341  * Calculates the points for Medium Mountain finish stage
342  * Data from Figure 1 in coursework specification
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 pointsForMediumMountain(int rank) {
348     switch (rank) {
349     case 1:
350         return 30;
351     case 2:
352         return 25;
353     case 3:
354         return 22;
355     case 4:
356         return 19;
357     case 5:
358         return 17;
359     case 6:
360         return 15;
361     case 7:
362         return 13;
363     case 8:
364         return 11;
365     case 9:
366         return 9;
367     case 10:
368         return 7;
369     case 11:
370         return 6;
371     case 12:
372         return 5;
373     case 13:
374         return 4;

```

```

375         case 14:
376             return 3;
377         case 15:
378             return 2;
379         default:
380             return 0;
381     }
382 }
383
384 /**
385  * Calculates the points for High Mountain, Time Trail, Individual Trial stage
386  * Data from Figure 1 in coursework specification
387  *
388  * @param rank the rank of the rider
389  * @return the points the rider gets for the given rank
390  */
391 static public int pointsForHMTTIT(int rank) {
392     switch (rank) {
393         case 1:
394             return 20;
395         case 2:
396             return 17;
397         case 3:
398             return 15;
399         case 4:
400             return 13;
401         case 5:
402             return 11;
403         case 6:
404             return 10;
405         case 7:
406             return 9;
407         case 8:
408             return 8;
409         case 9:
410             return 7;
411         case 10:
412             return 6;
413         case 11:
414             return 5;
415         case 12:
416             return 4;
417         case 13:
418             return 3;
419         case 14:
420             return 2;
421         case 15:
422             return 1;
423         default:
424             return 0;
425     }
426 }
427
428 /**
429  * Rest the static counter to set the ids

```

```

430     */
431     public static void resetCounter() {
432         stageCount = 0;
433     }
434 }
435 }

```

## 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 preparation
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     this.teamRiders = new ArrayList<>();
33     this.teamId = teamCount++;
34
35     this.teamName = teamName;
36     this.teamDescription = teamDescription;
37 }
38
39 /**
40  * Getter for {@code this.teamId}
41  *
42  * @return the id of the team
43  */
44 public int getTeamId() {
45     return teamId;
46 }
47
48 /**
49  * Getter for {@code this.teamName}
50  *
51  * @return the name of the team
52  */
53 public String getTeamName() {
54     return teamName;
55 }
56
57 /**
58  * Getter for {@code this.teamDescription}
59  *
60  * @return the description of the team
61  */
62 public String getTeamDescription() {
63     return teamDescription;
64 }
65
66 /**
67  * Getter for {@code this.teamRiders}
68  *
69  * @return an array of the riders on the team
70  * @see cycling.Rider
71  */
72 public ArrayList<Rider> getRiders() {
73     return teamRiders;
74 }
75

```

```

76  /**
77   * add rider to team
78   *
79   * @param newRider the rider to add to the team
80   * @see cycling.Rider
81   */
82  public void addRider(Rider newRider) {
83      // add rider to arraylist
84      teamRiders.add(newRider);
85  }
86
87  /**
88   * remove a rider from the team
89   *
90   * @param riderToRemove the rider to remove from the team
91   * @throws IDNotRecognisedException if the rider is not in the team
92   * @see cycling.Rider
93   */
94  public void removeRider(Rider riderToRemove) throws IDNotRecognisedException {
95      // findRider throws IDNotRecognisedException
96      int riderPosition = findRider(riderToRemove);
97      teamRiders.remove(riderPosition);
98  }
99
100
101  /**
102   * return the index of the rider in {@code this.teamRiders}
103   *
104   * @param riderToFind the rider to find
105   * @return the index of the rider in the rider array
106   * @throws IDNotRecognisedException if the rider is not in the team
107   * @see cycling.Rider
108   */
109  public int findRider(Rider riderToFind) throws IDNotRecognisedException {
110
111      // loops through all team riders
112      // checks id against given rider id
113      // if ids match, return the position, id not throw exception
114      for (int i = 0; i < teamRiders.size(); i++) {
115          if (teamRiders.get(i).getRiderId() == riderToFind.getRiderId()) {
116              return i;
117          }
118      }
119
120      throw new IDNotRecognisedException("Rider id not found");
121  }
122
123  /**
124   * Check if the rider is in the team
125   *
126   * @param riderToFind the rider to find
127   * @return boolean whether the rider is in the team
128   * @see cycling.Rider
129   */
130  public boolean containsRider(Rider riderToFind) {

```

```

131     // try find the rider using findRider function
132     // if the function throws an IDNotRecognisedException exception,
133     // the rider does not exists and reutrn false,
134     // otherwise return ture
135     try {
136         findRider(riderToFind);
137     } catch (IDNotRecognisedException e) {
138         return false;
139     }
140
141     return true;
142 }
143
144 /**
145  * Rest the static counter to set the ids
146  */
147 public static void resetCounter() {
148     teamCount = 0;
149 }
150
151 /**
152  * Rider toString
153  *
154  * @return a formatted string with relevent rider data
155  */
156 public String toString() {
157     return "Team[";
158 }
159 }

```