

1 CyclingPortal.java

```
1 package cycling;
2
3 import java.util.Arrays;
4 import java.util.Comparator;
5 import java.util.HashMap;
6 import java.io.IOException;
7 import java.time.LocalDateTime;
8 import java.time.LocalTime;
9 import java.util.ArrayList;
10 import java.io.ObjectOutputStream;
11 import java.io.FileOutputStream;
12 import java.io.ObjectInputStream;
13 import java.io.FileInputStream;
14
15
16
17 /**
18  * CyclingPortal implements CyclingPortalInterface; contains
19  * methods for
20  * handling the following classes: Race, Stage, Segment,
21  * RiderManager (and in
22  * turn Rider and Team), and Result.
23  * These classes are used manage races and their subdivisions,
24  * teams and their
25  * riders, and to calculate and assign points.
26  * Also contains methods for saving and loading
27  * MiniCyclingPortalInterface to
28  * and from a file.
29  *
30  * @author Ethan Ray & Thomas Newbold
31  * @version 1.0
32  */
33 public class CyclingPortal implements CyclingPortalInterface {
34     public RiderManager riderManager = new RiderManager();
35
36     @Override
37     public int[] getRaceIds() {
38         return Race.getAllRaceIds();
39     }
40
41     @Override
42     public int createRace(String name, String description)
43         throws IllegalArgumentException, InvalidNameException {
44         Race r = new Race(name, description);
45         return r.getRaceId();
46     }
47 }
```

```

43
44     @Override
45     public String viewRaceDetails(int raceId) throws
        IDNotRecognisedException {
46         double sum = 0.0;
47         for(int id : Race.getStages(raceId)) {
48             sum += Stage.getStageLength(id);
49         }
50         return Race.toString(raceId)+Double.toString(sum)+" ";
51     }
52
53     @Override
54     public void removeRaceById(int raceId) throws
        IDNotRecognisedException {
55         Race.removeRace(raceId);
56     }
57
58     @Override
59     public int getNumberOfStages(int raceId) throws
        IDNotRecognisedException {
60         int[] stageIds = Race.getStages(raceId);
61         return stageIds.length;
62     }
63
64     @Override
65     public int addStageToRace(int raceId, String stageName,
        String description, double length, LocalDateTime
        startTime,
66         StageType type)
67         throws IDNotRecognisedException,
        IllegalNameException, InvalidNameException,
        InvalidLengthException {
68         return Race.addStageToRace(raceId, stageName,
        description, length, startTime, type);
69     }
70
71     @Override
72     public int[] getRaceStages(int raceId) throws
        IDNotRecognisedException {
73         return Race.getStages(raceId);
74     }
75
76     @Override
77     public double getStageLength(int stageId) throws
        IDNotRecognisedException {
78         return Stage.getStageLength(stageId);
79     }
80
81     @Override
82     public void removeStageById(int stageId) throws

```

```

            IDNotRecognisedException {
83         Race.removeStage(stageId);
84     }
85
86     @Override
87     public int addCategorizedClimbToStage(int stageId, Double
        location, SegmentType type, Double averageGradient,
88         Double length) throws IDNotRecognisedException,
            InvalidLocationException,
            InvalidStageStateException,
            InvalidStageTypeException {
89         return Stage.addSegmentToStage(stageId, location, type,
90             averageGradient, length);
91     }
92
93     @Override
94     public int addIntermediateSprintToStage(int stageId, double
        location) throws IDNotRecognisedException,
95         InvalidLocationException,
            InvalidStageStateException,
            InvalidStageTypeException {
96         return Stage.addSegmentToStage(stageId, location,
97             SegmentType.SPRINT, 0.0, 0.0);
98     }
99
100    @Override
101    public void removeSegment(int segmentId) throws
        IDNotRecognisedException, InvalidStageStateException {
102        Stage.removeSegment(segmentId);
103    }
104
105    @Override
106    public void concludeStagePreparation(int stageId) throws
        IDNotRecognisedException, InvalidStageStateException {
107        Stage.updateStageState(stageId);
108    }
109
110    @Override
111    public int[] getStageSegments(int stageId) throws
        IDNotRecognisedException {
112        return Stage.getSegments(stageId);
113    }
114
115    @Override
116    public int createTeam(String name, String description)
        throws IllegalNameException, InvalidNameException {
117        return riderManager.createTeam(name, description);
118    }
119
120    @Override

```

```

120     public void removeTeam(int teamId) throws
        IDNotRecognisedException {
121         riderManager.removeTeam(teamId);
122     }
123
124     @Override
125     public int[] getTeams() {
126         return riderManager.getTeams();
127     }
128
129     @Override
130     public int[] getTeamRiders(int teamId) throws
        IDNotRecognisedException {
131         return riderManager.getTeamRiders(teamId);
132     }
133
134     @Override
135     public int createRider(int teamID, String name, int
        yearOfBirth) throws IDNotRecognisedException,
        IllegalArgumentException {
136         return riderManager.createRider(teamID, name,
            yearOfBirth);
137     }
138
139     @Override
140     public void removeRider(int riderId) throws
        IDNotRecognisedException {
141         riderManager.removeRider(riderId);
142     }
143
144     @Override
145     public void registerRiderResultsInStage(int stageId, int
        riderId, LocalDateTime... checkpoints)
        throws IDNotRecognisedException,
        DuplicatedResultException,
        InvalidCheckpointsException,
        InvalidStageStateException {
146         if (Stage.getStageState(stageId).equals(StageState.
            BUILDING)) {
147             throw new InvalidStageStateException("stage is not
                waiting for results");
148         } else if (Stage.getSegments(stageId).length+2 !=
            checkpoints.length) {
149             throw new InvalidCheckpointsException("checkpoint
                count mismatch");
150         }
151     }
152     try {
153

```

```

157         Result.getResult(stageId, riderId);
158         throw new DuplicatedResultException();
159     } catch(IDNotRecognisedException ex) {
160         Stage.getStage(stageId);
161         riderManager.getRider(riderId);
162         // above should throw exceptions if IDs are not in
            system
163         new Result(stageId, riderId, checkpoints);
164     }
165 }
166
167 @Override
168 public LocalTime[] getRiderResultsInStage(int stageId, int
riderId) throws IDNotRecognisedException {
169     Stage.getStage(stageId);
170     riderManager.getRider(riderId);
171     // above should throw exceptions if IDs are not in
        system
172     Result result = Result.getResult(stageId, riderId);
173     LocalTime[] checkpointTimes = result.getCheckpoints();
174     LocalTime[] out = new LocalTime[checkpointTimes.length
+1];
175     for(int i=0; i<checkpointTimes.length; i++) {
176         out[i] = checkpointTimes[i];
177     }
178     out[-1] = result.getTotalElapsed();
179     return out;
180 }
181
182 @Override
183 public LocalTime getRiderAdjustedElapsedTimeInStage(int
stageId, int riderId) throws IDNotRecognisedException {
184     Stage.getStage(stageId);
185     riderManager.getRider(riderId);
186     // above should throw exceptions if IDs are not in
        system
187     LocalTime[] adjustedTimes = Result.getResult(stageId,
riderId).adjustedCheckpoints();
188     LocalTime elapsedTime = adjustedTimes[0];
189     for(int i=1; i<adjustedTimes.length; i++) {
190         LocalTime t = adjustedTimes[i];
191         //elapsedTime.plusHours(t.getHour()).plusMinutes(t.
getMinute()).plusSeconds(t.getSecond()).
plusNanos(t.getNano());
192         elapsedTime = elapsedTime.plusHours(t.getHour());
193         elapsedTime = elapsedTime.plusMinutes(t.getMinute()
);
194         elapsedTime = elapsedTime.plusSeconds(t.getSecond()
);
195     }

```

```

196         return elapsedTime;
197     }
198
199     @Override
200     public void deleteRiderResultsInStage(int stageId, int
        riderId) throws IDNotRecognisedException {
201         Stage.getStage(stageId);
202         riderManager.getRider(riderId);
203         // above should throw exceptions if IDs are not in
            system
204         Result.removeResult(stageId, riderId);
205     }
206
207     @Override
208     public int[] getRidersRankInStage(int stageId) throws
        IDNotRecognisedException {
209         Result[] results = Result.getResultsInStage(stageId);
210         int[] riderRanks = new int[results.length];
211         Arrays.fill(riderRanks, -1);
212         for(Result r : results) {
213             for(int i=0; i<riderRanks.length; i++) {
214                 if(riderRanks[i] == -1) {
215                     riderRanks[i] = r.getRiderId();
216                     break;
217                 } else if(r.getTotalElapsed().isBefore(Result.
                    getResult(stageId, riderRanks[i]).
                    getTotalElapsed())) {
218                     int temp;
219                     int prev = r.getRiderId();
220                     for(int j=i; j<riderRanks.length; j++) {
221                         temp = riderRanks[j];
222                         riderRanks[j] = prev;
223                         prev = temp;
224                         if(prev == -1) {
225                             break;
226                         }
227                     }
228                     break;
229                 }
230             }
231         }
232         return riderRanks;
233     }
234
235     @Override
236     public LocalTime[] getRankedAdjustedElapsedTimesInStage(int
        stageId) throws IDNotRecognisedException {
237         int[] riderRanks = this.getRidersRankInStage(stageId);
238         LocalTime[] out = new LocalTime[riderRanks.length];
239         for(int i=0; i<out.length; i++) {

```

```

240         Result r = Result.getResult(stageId, riderRanks[i])
                ;
241         LocalTime[] checkpoints = r.getCheckpoints();
242         LocalTime[] adjustedTimes = r.adjustedCheckpoints()
                ;
243         out[i] = adjustedTimes[0];
244         LocalTime adjustedSplit;
245         for(int j=0; j<adjustedTimes.length; j++) {
246             adjustedSplit = Result.getElapsed(adjustedTimes
                [j], checkpoints[j]);
247             out[i] = out[i].plusHours(adjustedSplit.getHour
                ());
248             out[i] = out[i].plusMinutes(adjustedSplit.
                getMinute());
249             out[i] = out[i].plusSeconds(adjustedSplit.
                getSecond());
250         }
251     }
252     return out;
253 }
254
255 @Override
256 public int[] getRidersPointsInStage(int stageId) throws
    IDNotRecognisedException {
257     StageType type = Stage.getStageType(stageId);
258     int[] points = new int[Result.getResultsInStage(stageId
        ).length];
259     int[] distribution = new int[15];
260     // distributions from https://en.wikipedia.org/wiki/
        Points_classification_in_the_Tour_de_France
261     switch(type) {
262         case FLAT:
263             distribution = new int
                []{50,30,20,18,16,14,12,10,8,7,6,5,4,3,2};
264             break;
265         case MEDIUM_MOUNTAIN:
266             distribution = new int
                []{30,25,22,19,17,15,13,11,9,7,6,5,4,3,2};
267             break;
268         case HIGH_MOUNTAIN:
269             distribution = new int
                []{20,17,15,13,11,10,9,8,7,6,5,4,3,2,1};
270             break;
271         case TT:
272             distribution = new int
                []{20,17,15,13,11,10,9,8,7,6,5,4,3,2,1};
273             break;
274     }
275     for(int i=0; i<Math.min(points.length, distribution.
        length); i++) {

```

```

276         points[i] = distribution[i];
277     }
278     return points;
279 }
280
281 @Override
282 public int[] getRidersMountainPointsInStage(int stageId)
    throws IDNotRecognisedException {
283     Result[] results = Result.getResultsInStage(stageId);
284     // All results referring to the stage with id *stageId*
285     int[] riders = getRidersRankInStage(stageId);
286     // An int array of rider ids, from first to last
287     int[] segments = Stage.getSegments(stageId);
288     // An int array of the segment ids in the stage
289     int[] points = new int[riders.length];
290     // The int in position i is the number of points to be
        awarded to the rider with id riders[i]
291     for(int s=0; s<segments.length; s++) {
292         SegmentType type = Segment.getSegmentType(segments[
            s]);
293         int[] distribution = new int[1];
294         // The points to be awarded in order for the
            segment
295         switch(type) {
296             case C4:
297                 distribution = new int[]{1};
298                 break;
299             case C3:
300                 distribution = new int[]{2,1};
301                 break;
302             case C2:
303                 distribution = new int[]{5,3,2,1};
304                 break;
305             case C1:
306                 distribution = new int[]{10,8,6,4,2,1};
307                 break;
308             case HC:
309                 distribution = new int
                    []{20,15,12,10,8,6,4,2};
310                 break;
311             case SPRINT:
312         }
313         // get ranks for segment
314         int[] riderRanks = new int[results.length];
315         Arrays.fill(riderRanks, -1);
316         for(Result r : results) {
317             for(int i=0; i<riderRanks.length; i++) {
318                 if(riderRanks[i] == -1) {
319                     riderRanks[i] = r.getRiderId();
320                     break;

```



```

321         } else if(r.getCheckpoints()[s].isBefore(
                Result.getResult(stageId, riderRanks[i])
                .getCheckpoints()[s])) {
322             int temp;
323             int prev = r.getRiderId();
324             for(int j=i; j<riderRanks.length; j++)
                {
325                 temp = riderRanks[j];
326                 riderRanks[j] = prev;
327                 prev = temp;
328                 if(prev == -1) {
329                     break;
330                 }
331             }
332             break;
333         }
334     }
335 }
336 ArrayList<Integer> ridersArray = new ArrayList<
    Integer>();
337 for(int r : riders) { ridersArray.add(r); }
338 for(int i=0; i<Math.min(points.length, distribution
    .length); i++) {
339     int overallPos = ridersArray.indexOf(riderRanks
        [i]);
340     if(overallPos<points.length && overallPos!=-1)
        {
341         points[overallPos] += distribution[i];
342     }
343 }
344 }
345 return points;
346 }
347
348 @Override
349 public void eraseCyclingPortal() {
350
351     Team.teamNames.clear();
352     Team.teamTopId = 0;
353     Rider.ridersTopId = 0;
354
355     RiderManager.allRiders.clear();
356     RiderManager.allTeams.clear();
357
358
359     Race.allRaces.clear();
360     Race.removedIds.clear();
361     Race.loadId();
362
363     Segment.allSegments.clear();

```

```

364         Segment.removedIds.clear();
365         Segment.loadId();
366
367         Stage.allStages.clear();
368         Stage.removedIds.clear();
369         Stage.loadId();
370
371         Result.allResults.clear();
372
373
374     }
375
376     @Override
377     public void saveCyclingPortal(String filename) throws
378         IOException {
379         try {
380             FileOutputStream fos = new FileOutputStream(
381                 filename);
382             ObjectOutputStream oos = new ObjectOutputStream(fos
383                 );
384             ArrayList<ArrayList> allObj = new ArrayList<>();
385             allObj.add(RiderManager.allTeams);
386             allObj.add(RiderManager.allRiders);
387             allObj.add(Stage.allStages);
388             allObj.add(Stage.removedIds);
389             allObj.add(Race.allRaces);
390             allObj.add(Race.removedIds);
391             allObj.add(Result.allResults);
392             allObj.add(Segment.allSegments);
393             allObj.add(Segment.removedIds);
394
395             oos.writeObject(allObj);
396
397             oos.flush();
398             oos.close();
399
400         } catch (IOException ex) {
401             ex.printStackTrace();
402         }
403
404     @Override
405     public void loadCyclingPortal(String filename) throws
406         IOException, ClassNotFoundException {
407         try {
408             FileInputStream fis = new FileInputStream(filename)
409                 ;
410             ObjectInputStream ois = new ObjectInputStream(fis);

```

```

409     ArrayList<Object> allObjects = new ArrayList<>();
410     ArrayList<Team> allTeams = new ArrayList<>();
411     ArrayList<Rider> allRiders = new ArrayList<>();
412     ArrayList<Result> allResults = new ArrayList<Result
        >();
413     ArrayList<Race> allRaces = new ArrayList<Race>();
414     ArrayList<Stage> allStages = new ArrayList<Stage>()
        ;
415     ArrayList<Segment> allSegments = new ArrayList<
        Segment>();
416     ArrayList<Integer> removedIds = new ArrayList<>();
417
418     Class<?> classFlag = null;
419
420     allObjects = (ArrayList) ois.readObject();
421     for (Object tempObj : allObjects){
422         ArrayList Objects = (ArrayList) tempObj;
423         for (Object obj : Objects){
424             if (classFlag != null){
425                 if (obj.getClass() != classFlag && obj.
                    getClass() != Integer.class){
426                     if (classFlag == Race.class){
427                         Race.removedIds = removedIds;
428                     }
429                     if (classFlag == Segment.class){
430                         Segment.removedIds = removedIds;
431                     }
432                     if (classFlag == Stage.class){
433                         Stage.removedIds = removedIds;
434                     }
435                     classFlag = null;
436                     removedIds.clear();
437
438                 }
439             }
440             else{
441                 Integer removedId = (Integer) obj;
442                 removedIds.add(removedId);
443             }
444         }
445     }
446     String objClass = obj.getClass().getName();
447     System.out.println(objClass);
448     if (obj.getClass() == Rider.class){
449         Rider newRider = (Rider) obj;
450         allRiders.add(newRider);
451         System.out.println("NEW RIDER");
452     }
453     if (obj.getClass() == Team.class){
454         Team newTeam = (Team) obj;

```

```

455         allTeams.add(newTeam);
456         System.out.println("NEW TEAM");
457     }
458     if (obj.getClass() == Result.class){
459         Result newResult = (Result) obj;
460         allResults.add(newResult);
461         System.out.println("NEW RESULT");
462     }
463     if (obj.getClass() == Stage.class){
464         Stage newStage = (Stage) obj;
465         allStages.add(newStage);
466         System.out.println("NEW STAGE");
467         classFlag = Stage.class;
468     }
469     if (obj.getClass() == Race.class){
470         Race newRace = (Race) obj;
471         allRaces.add(newRace);
472         System.out.println("NEW Race");
473         classFlag = Race.class;
474     }
475     if (obj.getClass() == Segment.class){
476         Segment newSeg = (Segment) obj;
477         allSegments.add(newSeg);
478         System.out.println("NEW SEGMENT");
479         classFlag = Segment.class;
480     }
481
482
483     System.out.println(obj.getClass());
484 }
485
486 if (classFlag == Race.class){
487     Race.removedIds = removedIds;
488 }
489 if (classFlag == Segment.class){
490     Segment.removedIds = removedIds;
491 }
492 if (classFlag == Stage.class){
493     Stage.removedIds = removedIds;
494 }
495
496 this.riderManager.setAllTeams(allTeams);
497 this.riderManager.setAllRiders(allRiders);
498 Race.allRaces = allRaces;
499 Race.loadId();
500 Stage.allStages = allStages;
501 Stage.loadId();
502 Segment.allSegments = allSegments;
503 Segment.loadId();
504 Result.allResults = allResults;

```

```

505         ois.close();
506     }
507     catch (Exception ex) {
508         ex.printStackTrace();
509     }
510 }
511
512 }
513
514 @Override
515 public void removeRaceByName(String name) throws
    NameNotRecognisedException {
516     boolean found = false;
517     for (int raceId : Race.getAllRaceIds()){
518         try {
519             if (name == Race.getRaceName(raceId)){
520                 Race.removeRace(raceId);
521             }
522         }
523         catch(Exception c){
524             assert(false); // Exception will not throw by
                    // for each condition
525             // This try catch is easier than moving
                    // exceptions to CyclingPortal level
526         }
527     }
528     if (!found){ throw new NameNotRecognisedException("Name
                    not in System.");}
529 }
530
531 }
532
533 @Override
534 public LocalTime[] getGeneralClassificationTimesInRace(int
    raceId) throws IDNotRecognisedException {
535     Race currentRace = Race.getRace(raceId);
536     int[] stageIds = currentRace.getStages();
537     int[] riderIds = this.riderManager.getRiderIds();
538     HashMap<Integer,Long> riderElaspedTime = new HashMap<
        Integer,Long>(); //Rider Id -> totalTime (long)
539     for (int riderId : riderIds){
540         riderElaspedTime.put(riderId,0L);
541     }
542     for (int stageId : stageIds){
543         Result[] temp = Result.getResultsInStage(stageId);
544         for(Result result: temp){
545             int riderId = result.getRiderId();
546             LocalTime getTotalElasped = result.
                    getTotalElasped();
547             long timeTaken = getTotalElasped.toNanoOfDay();

```

```

548         Long newTime = (Long)riderElaspedTime.get(
549             riderId)+timeTaken;
550         riderElaspedTime.put(riderId,newTime);
551     }
552 }
553 long[][] riderTimePos = new long[riderIds.length][2];
554 int count = 0;
555 for (int riderId : riderIds){
556     Long finalRiderTime = riderElaspedTime.get(riderId)
557         ;// ## -> [[time,riderId],...] sort by time!
558     riderTimePos[count][0] = riderId;
559     riderTimePos[count][1] = finalRiderTime;
560     count++;
561 }
562 Arrays.sort(riderTimePos, Comparator.comparingDouble(o
563     -> o[1]));
564 LocalTime[] finalTimes = new LocalTime[riderIds.length
565     ];
566 count = 0;
567 for (long[] items : riderTimePos){
568     finalTimes[count]= LocalTime.ofNanoOfDay(items[1]);
569     count++;
570 }
571 return finalTimes;
572 }
573
574 @Override
575 public int[] getRidersPointsInRace(int raceId) throws
576     IDNotRecognisedException {
577     ArrayList<Integer> order = new ArrayList<Integer>();
578     for(int riderId : getRidersGeneralClassificationRank(
579         raceId)) {
580         order.add(riderId);
581     }
582 int[] out = new int[order.size()];
583 int[] stageRank, stagePoints;
584 for(int stageId : Race.getStages(raceId)) {
585     stageRank = getRidersRankInStage(stageId);
586     stagePoints = getRidersPointsInStage(stageId);
587     for(int i=0; i<stageRank.length; i++) {
588         out[order.indexOf(stageRank[i])] += stagePoints
589             [i];
590     }
591 }
592 return out;
593 }

```

```

591
592     @Override
593     public int[] getRidersMountainPointsInRace(int raceId)
594         throws IDNotRecognisedException {
595         ArrayList<Integer> order = new ArrayList<Integer>();
596         for(int riderId : getRidersGeneralClassificationRank(
597             raceId)) {
598             order.add(riderId);
599         }
600         int[] out = new int[order.size()];
601         int[] stageRank, stagePoints;
602         for(int stageId : Race.getStages(raceId)) {
603             stageRank = getRidersRankInStage(stageId);
604             stagePoints = getRidersMountainPointsInStage(
605                 stageId);
606             for(int i=0; i<stageRank.length; i++) {
607                 out[order.indexOf(stageRank[i])] += stagePoints
608                     [i];
609             }
610         }
611         return out;
612     }
613
614     @Override
615     public int[] getRidersGeneralClassificationRank(int raceId)
616         throws IDNotRecognisedException {
617         Race currentRace = Race.getRace(raceId);
618         int[] stageIds = currentRace.getStages();
619         int[] riderIds = this.riderManager.getRiderIds();
620         HashMap<Integer,Long> riderElaspedTime = new HashMap<
621             Integer,Long>(); //Rider Id -> totalTime (long)
622         for (int riderId : riderIds){
623             riderElaspedTime.put(riderId,0L);
624         }
625         for (int stageId : stageIds){
626             Result[] temp = Result.getResultsInStage(stageId);
627             for(Result result: temp){
628                 int riderId = result.getRiderId();
629                 LocalTime getTotalElasped = result.
630                     getTotalElasped();
631                 long timeTaken = getTotalElasped.toNanoOfDay();
632                 Long newTime = (Long)riderElaspedTime.get(
633                     riderId)+timeTaken;
634                 riderElaspedTime.put(riderId,newTime);
635             }
636         }
637         long[][] riderTimePos = new long[riderIds.length][2];
638         int count = 0;
639         for (int riderId : riderIds){

```

```

633         Long finalRiderTime = riderElapsedTime.get(riderId)
           ;// ## -> [[time,riderId],...] sort by time!
634         riderTimePos[count][0] = riderId;
635         riderTimePos[count][1] = finalRiderTime;
636         count++;
637     }
638     Arrays.sort(riderTimePos, Comparator.comparingDouble(o
        -> o[1]));
639     int[] finalPos = new int[riderIds.length];
640     count = 0;
641     for (long[] items : riderTimePos){
642         finalPos[count]= (int)items[0];
643         count++;
644     }
645
646     return finalPos;
647 }
648
649 @Override
650 public int[] getRidersPointClassificationRank(int raceId)
        throws IDNotRecognisedException {
651     int[] order = getRidersGeneralClassificationRank(raceId
        );
652     int[] points = getRidersPointsInRace(raceId);
653     int[] out = new int[order.length];
654     for(int i=0; i<out.length; i++) {
655         int maxPoints = -1;
656         int nextId = -1;
657         for(int j=0; j<order.length; j++) {
658             int id = order[j];
659             if(id<0) { continue; }
660             if(points[id] > maxPoints) {
661                 maxPoints = points[j];
662                 nextId = id;
663             }
664         }
665         if(maxPoints < 0) {
666             break;
667         } else {
668             System.out.print(nextId);
669             out[i] = nextId;
670             order[nextId] = -1;
671         }
672     }
673     return out;
674 }
675
676 @Override
677 public int[] getRidersMountainPointClassificationRank(int
        raceId) throws IDNotRecognisedException {

```



```

678         // effectively a clone of the method above
679         int[] order = getRidersGeneralClassificationRank(raceId
        );
680         int[] points = getRidersMountainPointsInRace(raceId);
681         int[] out = new int[order.length];
682         for(int i=0; i<out.length; i++) {
683             int maxPoints = -1;
684             int nextId = -1;
685             for(int j=0; j<order.length; j++) {
686                 int id = order[j];
687                 if(id<0) { continue; }
688                 if(points[id] > maxPoints) {
689                     maxPoints = points[j];
690                     nextId = id;
691                 }
692             }
693             if(maxPoints < 0) {
694                 break;
695             } else {
696                 out[i] = nextId;
697                 order[nextId] = -1;
698             }
699         }
700         return out;
701     }
702 }

```

2 Race.java

```

1 package cycling;
2
3 import java.util.ArrayList;
4 import java.io.Serializable;
5 import java.time.LocalDateTime;
6
7 /**
8  * Race encapsulates tour races, each of which has a number of
9  * associated
10  * Stages.
11  *
12  * @author Thomas Newbold
13  * @version 2.0
14  */
15 public class Race implements Serializable {
16     // Static class attributes
17     private static int idMax = 0;
18     public static ArrayList<Integer> removedIds = new ArrayList
        <Integer>();

```

```

19     public static ArrayList<Race> allRaces = new ArrayList<Race
20         >();
21
22     /**
23      * Loads the value of idMax.
24      */
25     public static void loadId() {
26         if(Race.allRaces.size()!=0) {
27             Race.idMax = Race.allRaces.get(Race.allRaces.size()
28                 -1).getRaceId() + 1;
29         } else {
30             Race.idMax = 0;
31         }
32     }
33
34     /**
35      * @param raceId The ID of the race instance to fetch
36      * @return The race instance with the associated ID
37      * @throws IDNotRecognisedException If no race exists with
38      *         the requested ID
39      */
40     public static Race getRace(int raceId) throws
41         IDNotRecognisedException {
42         boolean removed = Race.removedIds.contains(raceId);
43         if(raceId<Race.idMax && raceId >= 0 && !removed) {
44             int index = raceId;
45             for(int j=0; j<Race.removedIds.size(); j++) {
46                 if(Race.removedIds.get(j) < raceId) {
47                     index--;
48                 }
49             }
50             return allRaces.get(index);
51         } else if (removed) {
52             throw new IDNotRecognisedException("no race
53                 instance for raceID");
54         } else {
55             throw new IDNotRecognisedException("raceID out of
56                 range");
57         }
58     }
59
60     /**
61      * @return An integer array of the race IDs of all races
62      */
63     public static int[] getAllRaceIds() {
64         int length = Race.allRaces.size();
65         int[] raceIdsArray = new int[length];
66         int i = 0;
67         for(Race race : allRaces) {
68             raceIdsArray[i] = race.getRaceId();

```

```

63         i++;
64     }
65     return raceIdsArray;
66 }
67
68 /**
69  * @param raceId The ID of the race instance to remove
70  * @throws IDNotRecognisedException If no race exists with
71  * the requested ID
72  */
73 public static void removeRace(int raceId) throws
74     IDNotRecognisedException {
75     boolean removed = Race.removedIds.contains(raceId);
76     if(raceId<Race.idMax && raceId >= 0 && !removed) {
77         Race r = getRace(raceId);
78         for(int id : r.getStages()) {
79             r.removeStageFromRace(id);
80         }
81         allRaces.remove(r);
82         removedIds.add(raceId);
83     } else if (removed) {
84         throw new IDNotRecognisedException("no race
85             instance for raceID");
86     } else {
87         throw new IDNotRecognisedException("raceID out of
88             range");
89     }
90 }
91
92 // Instance attributes
93 private int raceId;
94 private String raceName;
95 private String raceDescription;
96 private ArrayList<Integer> stageIds;
97
98 /**
99  * @param name String to be checked
100  * @return true if name is valid for the system
101  */
102 private static boolean validName(String name) {
103     if(name==null || name.equals("")) {
104         return false;
105     } else if(name.length()>30) {
106         return false;
107     } else if(name.contains(" ")) {
108         return false;
109     } else {
110         return true;
111     }
112 }

```

```

109
110      /**
111       * Race constructor; creates new race and adds to allRaces
112         array.
113       *
114       * @param name The name of the new race
115       * @param description The description for the new race
116       * @throws IllegalArgumentException If name already exists in
117         the system
118       * @throws InvalidNameException If name is empty/null,
119         contains whitespace,
120         or is longer than 30
121         characters
122       */
123     public Race(String name, String description) throws
124       IllegalArgumentException,
125       InvalidNameException {
126       for(Race race : allRaces) {
127         if(race.getRaceName().equals(name)) {
128           throw new IllegalArgumentException("name already
129             exists");
130         }
131       }
132       if(!validName(name)) {
133         throw new InvalidNameException("invalid name");
134       }
135       if(Race.removedIds.size() > 0) {
136         this.raceId = Race.removedIds.get(0);
137         Race.removedIds.remove(0);
138       } else {
139         this.raceId = idMax++;
140       }
141       this.raceName = name;
142       this.raceDescription = description;
143       this.stageIds = new ArrayList<Integer>();
144       Race.allRaces.add(this);
145     }
146
147     /**
148      * @return A string representation of the race instance
149     */
150     public String toString() {
151       String id = Integer.toString(this.raceId);
152       String name = this.raceName;
153       String description = this.raceDescription;
154       String list = this.stageIds.toString();
155       return String.format("Race[%s]: %s; %s; StageIds=%s;",
156         id, name,
157         description, list);
158     }

```

```

152
153     /**
154      * @param id The ID of the race
155      * @return A string representation of the race instance
156      * @throws IDNotRecognisedException If no race exists with
157      *         the requested ID
158      */
159     public static String toString(int id) throws
160         IDNotRecognisedException {
161         return getRace(id).toString();
162     }
163
164     /**
165      * @return The integer raceId for the race instance
166      */
167     public int getRaceId() { return this.raceId; }
168
169     /**
170      * @return The string raceName for the race instance
171      */
172     public String getRaceName() { return this.raceName; }
173
174     /**
175      * @param id The ID of the race
176      * @return The string raceName for the race with the
177      *         associated id
178      * @throws IDNotRecognisedException If no race exists with
179      *         the requested ID
180      */
181     public static String getRaceName(int id) throws
182         IDNotRecognisedException {
183         return getRace(id).raceName;
184     }
185
186     /**
187      * @return The string raceDescription for the race instance
188      */
189     public String getRaceDescription() { return this.
190         raceDescription; }
191
192     /**
193      * @param id The ID of the race
194      * @return The string raceDescription for the race with the
195      *         associated id
196      * @throws IDNotRecognisedException If no race exists with
197      *         the requested ID
198      */
199     public static String getRaceDescription(int id) throws
200         IDNotRecognisedException
201     {

```

```

193         return getRace(id).raceDescription;
194     }
195
196     /**
197     * @return An integer array of stage IDs for the race
198     *         instance
199     */
200     public int[] getStages() {
201         int length = this.stageIds.size();
202         int[] stageIdsArray = new int[length];
203         for(int i=0; i<length; i++) {
204             stageIdsArray[i] = this.stageIds.get(i);
205         }
206         return stageIdsArray;
207     }
208
209     /**
210     * @param id The ID of the race
211     * @return An integer array of stage IDs for the race
212     *         instance
213     * @throws IDNotRecognisedException If no race exists with
214     *         the requested ID
215     */
216     public static int[] getStages(int id) throws
217         IDNotRecognisedException {
218         Race race = getRace(id);
219         int length = race.stageIds.size();
220         int[] stageIdsArray = new int[length];
221         for(int i=0; i<length; i++) {
222             stageIdsArray[i] = race.stageIds.get(i);
223         }
224         return stageIdsArray;
225     }
226
227     /**
228     * @param name The new name for the race instance
229     */
230     public void setRaceName(String name) {
231         this.raceName = name;
232     }
233
234     /**
235     * @param id The ID of the race to be updated
236     * @param name The new name for the race instance
237     * @throws IDNotRecognisedException If no race exists with
238     *         the requested ID
239     */
240     public static void setRaceName(int id, String name) throws
241         IDNotRecognisedException {
242         getRace(id).setRaceName(name);
243     }

```

```

238     }
239
240     /**
241     * @param description The new description for the race
242     * instance
243     */
244     public void setRaceDescription(String description) {
245         this.raceDescription = description;
246     }
247
248     /**
249     * @param id The ID of the race to be updated
250     * @param description The new description for the race
251     * instance
252     * @throws IDNotRecognisedException If no race exists with
253     * the requested ID
254     */
255     public static void setRaceDescription(int id, String
256         description) throws
257         IDNotRecognisedException
258     {
259         getRace(id).setRaceDescription(description);
260     }
261
262     /**
263     * Creates a new stage and adds the ID to the stageIds
264     * array.
265     *
266     * @param name The name of the new stage
267     * @param description The description of the new stage
268     * @param length The length of the new stage (in km)
269     * @param startTime The date and time at which the stage
270     * will be held
271     * @param type The StageType, used to determine the point
272     * distribution
273     * @return The ID of the new stage
274     */
275     public int addStageToRace(String name, String description,
276         double length,
277         LocalDateTime startTime,
278         StageType type) throws
279         IllegalArgumentException,
280         InvalidNameException,
281         InvalidLengthException {
282         Stage newStage = new Stage(name, description, length,
283             startTime, type);
284         this.stageIds.add(newStage.getStageId());
285         return newStage.getStageId();
286     }

```

```

276  /**
277   * Creates a new stage and adds the ID to the stageIds
      array.
278   *
279   * @param id The ID of the race to which the stage will be
      added
280   * @param name The name of the new stage
281   * @param description The description of the new stage
282   * @param length The length of the new stage (in km)
283   * @param startTime The date and time at which the stage
      will be held
284   * @param type The StageType, used to determine the point
      distribution
285   * @return The ID of the new stage
286   * @throws IDNotRecognisedException If no race exists with
      the requested ID
287   */
288  public static int addStageToRace(int id, String name,
      String description,
289                                  double length,
                                  LocalDateTime startTime
                                  ,
290                                  StageType type) throws
      IDNotRecognisedException,
291      IllegalNameException,
292      InvalidNameException,
      InvalidLengthException {
293      return getRace(id).addStageToRace(name, description,
294      length, startTime, type);
295  }
296
297  /**
298   * Removes a stageId from the array of stageIds for a race
      instance,
299   * as well as from the static array of all stages in the
      Stage class.
300   *
301   * @param stageId The ID of the stage to be removed
302   * @throws IDNotRecognisedException If no stage exists with
      the requested ID
303   */
304  private void removeStageFromRace(int stageId) throws
      IDNotRecognisedException {
305      if(this.stageIds.contains(stageId)) {
306          this.stageIds.remove(stageId);
307          Stage.removeStage(stageId);
308      } else {
309          throw new IDNotRecognisedException("stageID not
      found in race");
310      }

```



```

311     }
312
313     /**
314     * Removes a stageId from the array of stageIds for a race
315     * instance,
316     * as well as from the static array of all stages in the
317     * Stage class.
318     *
319     * @param id The ID of the race to which the stage will be
320     * removed
321     * @param stageId The ID of the stage to be removed
322     * @throws IDNotRecognisedException If no stage exists with
323     * the requested ID
324     */
325     public static void removeStageFromRace(int id, int stageId)
326     throws
327         IDNotRecognisedException
328     {
329         getRace(id).removeStageFromRace(stageId);
330     }
331
332     /**
333     * Removes a stageId from the array of stageIds for a race
334     * instance,
335     * as well as from the static array of all stages in the
336     * Stage class.
337     *
338     * @param stageId The ID of the stage to be removed
339     * @throws IDNotRecognisedException If no stage exists with
340     * the requested ID
341     */
342     public static void removeStage(int stageId) throws
343         IDNotRecognisedException {
344         for(Race race : allRaces) {
345             if(race.stageIds.contains(stageId)) {
346                 race.removeStageFromRace(stageId);
347                 break;
348             }
349         }
350     }
351 }

```

3 Stage.java

```

1 package cycling;
2
3 import java.util.ArrayList;
4 import java.io.Serializable;
5 import java.time.LocalDateTime;

```

```

6 import java.time.format.DateTimeFormatter;
7
8 /**
9  * Stage encapsulates race stages, each of which has a number
10  * of associated
11  * Segments.
12  *
13  * @author Thomas Newbold
14  * @version 2.0
15  */
16 public class Stage implements Serializable {
17     // Static class attributes
18     private static int idMax = 0;
19     public static ArrayList<Integer> removedIds = new ArrayList<
20         <Integer>();
21     public static ArrayList<Stage> allStages = new ArrayList<
22         Stage>();
23
24     /**
25      * Loads the value of idMax.
26      */
27     public static void loadId(){
28         if(Stage.allStages.size()!=0) {
29             Stage.idMax = Stage.allStages.get(Stage.allStages.
30                 size()-1).getStageId() + 1;
31         } else {
32             Stage.idMax = 0;
33         }
34     }
35
36     /**
37      * @param stageId The ID of the stage instance to fetch
38      * @return The stage instance with the associated ID
39      * @throws IDNotRecognisedException If no stage exists with
40      * the requested ID
41      */
42     public static Stage getStage(int stageId) throws
43         IDNotRecognisedException {
44         boolean removed = Stage.removedIds.contains(stageId);
45         if(stageId<Stage.idMax && stageId >= 0 && !removed) {
46             int index = stageId;
47             for(int j=0; j<Stage.removedIds.size(); j++) {
48                 if(Stage.removedIds.get(j) < stageId) {
49                     index--;
50                 }
51             }
52             return allStages.get(index);
53         } else if (removed) {
54             throw new IDNotRecognisedException("no stage

```

```

        instance for stageID");
50     } else {
51         throw new IDNotRecognisedException("stageId out of
            range");
52     }
53 }
54
55 /**
56  * @return An integer array of the stage IDs of all stage
57  */
58 public static int[] getAllStageIds() {
59     int length = Stage.allStages.size();
60     int[] stageIdsArray = new int[length];
61     int i = 0;
62     for(Stage stage : allStages) {
63         stageIdsArray[i] = stage.getStageId();
64         i++;
65     }
66     return stageIdsArray;
67 }
68
69 /**
70  * @param stageId The ID of the stage instance to remove
71  * @throws IDNotRecognisedException If no stage exists with
72  *         the requested ID
73  */
74 public static void removeStage(int stageId) throws
    IDNotRecognisedException {
75     boolean removed = Stage.removedIds.contains(stageId);
76     if(stageId < Stage.idMax && stageId >= 0 && !removed) {
77         Stage s = getStage(stageId);
78         for(int id : s.getSegments()) {
79             s.removeSegmentFromStage(id);
80         }
81         allStages.remove(s);
82         removedIds.add(stageId);
83     } else if (removed) {
84         throw new IDNotRecognisedException("no stage
            instance for stageID");
85     } else {
86         throw new IDNotRecognisedException("stageId out of
            range");
87     }
88 }
89
90 // Instance attributes
91 private int stageId;
92 private StageState stageState;
93 private String stageName;
94 private String stageDescription;

```

```

94     private double stageLength;
95     private LocalDateTime stageStartTime;
96     private StageType stageType;
97     private ArrayList<Integer> segmentIds;
98
99     /**
100      * @param name String to be checked
101      * @return true if name is valid for the system
102      */
103     private static boolean validName(String name) {
104         if(name==null || name.equals("")) {
105             return false;
106         } else if(name.length()>30) {
107             return false;
108         } else if(name.contains(" ")) {
109             return false;
110         } else {
111             return true;
112         }
113     }
114
115     /**
116      * Stage constructor; creates a new stage and adds to
117      * allStages array.
118      *
119      * @param name The name of the new stage
120      * @param description The description of the new stage
121      * @param length The total length of the new stage
122      * @param startTime The start time for the new stage
123      * @param type The type of the new stage
124      * @throws IllegalArgumentException If name already exists in
125      * the system
126      * @throws InvalidNameException If name is empty/null,
127      * contains whitespace,
128      * or is longer than 30
129      * characters
130      * @throws InvalidLengthException If the length is less
131      * than 5km
132      */
133     public Stage(String name, String description, double length
134                 ,
135                 LocalDateTime startTime, StageType type)
136         throws
137         IllegalArgumentException, InvalidNameException,
138         InvalidLengthException {
139         for(Stage stage : allStages) {
140             if(stage.getStageName().equals(name)) {
141                 throw new IllegalArgumentException("name already
142                     exists");
143             }
144         }
145     }

```

```

136     }
137     if(!validName(name)) {
138         throw new InvalidNameException("invalid name");
139     }
140     if(length<5) {
141         throw new InvalidLengthException("length less than
142             5km");
143     }
144     if(Stage.removedIds.size() > 0) {
145         this.stageId = Stage.removedIds.get(0);
146         Stage.removedIds.remove(0);
147     } else {
148         this.stageId = idMax++;
149     }
150     this.stageState = StageState.BUILDING;
151     this.stageName = name;
152     this.stageDescription = description;
153     this.stageLength = length;
154     this.stageStartTime = startTime;
155     this.stageType = type;
156     this.segmentIds = new ArrayList<Integer>();
157     Stage.allStages.add(this);
158 }
159 /**
160  * @return A string representation of the stage instance
161  */
162 public String toString() {
163     String id = Integer.toString(this.stageId);
164     String state;
165     switch (this.stageState) {
166         case BUILDING:
167             state = "In preperation";
168             break;
169         case WAITING:
170             state = "Waiting for results";
171             break;
172         default:
173             state = "null state";
174     }
175     String name = this.stageName;
176     String description = this.stageDescription;
177     String length = Double.toString(this.stageLength);
178     DateTimeFormatter formatter = DateTimeFormatter.
179         ofPattern("HH:hh dd-MM-yyyy");
180     String startTime = this.stageStartTime.format(formatter
181         );
182     String list = this.segmentIds.toString();
183     String type;
184     switch (this.stageType) {

```

```

183         case FLAT:
184             type = "Flat";
185             break;
186         case MEDIUM_MOUNTAIN:
187             type = "Medium Mountain";
188             break;
189         case HIGH_MOUNTAIN:
190             type = "High Mountain";
191             break;
192         case TT:
193             type = "Time Trial";
194             break;
195         default:
196             type = "null type";
197     }
198     return String.format("Stage[%s] (%s): %s (%s); %s; %skm;
199                          %s; SegmentIds=%s;",
200                          id, state, name, type, description
201                          , length,
202                          startTime, list);
203 }
204
205 /**
206  * @param id The ID of the stage
207  * @return A string representation of the stage instance
208  * @throws IDNotRecognisedException If no stage exists with
209  *         the requested ID
210  */
211 public static String toString(int id) throws
212     IDNotRecognisedException {
213     return getStage(id).toString();
214 }
215
216 /**
217  * @return The integer stageId for the stage instance
218  */
219 public int getStageId() { return this.stageId; }
220
221 /**
222  * @return The state of the stage instance
223  */
224 public StageState getStageState() { return this.stageState;
225 }
226
227 /**
228  * @param id The ID of the stage
229  * @return The state of the stage instance
230  * @throws IDNotRecognisedException If no stage exists with
231  *         the requested ID
232  */

```

```

227     public static StageState getStageState(int id) throws
228                                     IDNotRecognisedException
229                                     {
230         return getStage(id).getStageState();
231     }
232     /**
233     * @return The string raceName for the stage instance
234     */
235     public String getStageName() { return this.stageName; }
236
237     /**
238     * @param id The ID of the stage
239     * @return The string stageName for the stage with the
240     *         associated id
241     * @throws IDNotRecognisedException If no stage exists with
242     *         the requested ID
243     */
244     public static String getStageName(int id) throws
245                                     IDNotRecognisedException {
246         return getStage(id).stageName;
247     }
248
249     /**
250     * @return The string stageDescription for the stage
251     *         instance
252     */
253     public String getStageDescription() { return this.
254         stageDescription; }
255
256     /**
257     * @param id The ID of the stage
258     * @return The string stageDescription for the stage with
259     *         the associated id
260     * @throws IDNotRecognisedException If no stage exists with
261     *         the requested ID
262     */
263     public static String getStageDescription(int id) throws
264                                     IDNotRecognisedException
265                                     {
266         return getStage(id).stageDescription;
267     }
268
269     /**
270     * @return The length of the stage instance
271     */
272     public double getStageLength() { return this.stageLength; }
273
274     /**
275     * @param id The ID of the stage
276     * @return The length of the stage instance

```

```

268      * @throws IDNotRecognisedException If no stage exists with
269      the requested ID
270      */
271      public static double getStageLength(int id) throws
272      IDNotRecognisedException {
273          return getStage(id).stageLength;
274      }
275      /**
276      * @return The start time for the stage instance
277      */
278      public LocalDateTime getStageStartTime() { return this.
279      stageStartTime; }
280      /**
281      * @param id The ID of the stage
282      * @return The start time for the stage instance
283      * @throws IDNotRecognisedException If no stage exists with
284      the requested ID
285      */
286      public static LocalDateTime getStageStartTime(int id)
287      throws
288      IDNotRecognisedException
289      {
290          return getStage(id).stageStartTime;
291      }
292      /**
293      * @return The type of the stage instance
294      */
295      public StageType getStageType() { return this.stageType; }
296      /**
297      * @param id The ID of the stage
298      * @return The type of the stage instance
299      * @throws IDNotRecognisedException If no stage exists with
300      the requested ID
301      */
302      public static StageType getStageType(int id) throws
303      IDNotRecognisedException {
304          return getStage(id).getStageType();
305      }
306      /**
307      * @return An integer array of segment IDs for the stage
308      instance
309      */
310      public int[] getSegments() {
311          int length = this.segmentIds.size();
312          int[] segmentIdsArray = new int[length];

```



```

309         for(int i=0; i<length; i++) {
310             segmentIdsArray[i] = this.segmentIds.get(i);
311         }
312         return segmentIdsArray;
313     }
314
315     /**
316      * @param id The ID of the stage
317      * @return An integer array of segment IDs for the stage
318      *         instance
319      * @throws IDNotRecognisedException If no stage exists with
320      *         the requested ID
321      */
322     public static int[] getSegments(int id) throws
323         IDNotRecognisedException {
324         Stage stage = getStage(id);
325         int length = stage.segmentIds.size();
326         int[] segmentIdsArray = new int[length];
327         for(int i=0; i<length; i++) {
328             segmentIdsArray[i] = stage.segmentIds.get(i);
329         }
330         return segmentIdsArray;
331     }
332
333     /**
334      * Updates the stage state from building to waiting for
335      * results.
336      *
337      * @throws InvalidStageStateException If the stage is
338      *         already waiting for results
339      */
340     public void updateStageState() throws
341         InvalidStageStateException {
342         InvalidStageStateException {
343             if(this.stageState.equals(StageState.WAITING)) {
344                 throw new InvalidStageStateException("stage is
345                     already waiting for results");
346             } else if(this.stageState.equals(StageState.BUILDING))
347             {
348                 this.stageState = StageState.WAITING;
349             }
350         }
351     }
352
353     /**
354      * Updates the stage state from building to waiting for
355      * results.
356      *
357      * @param id The ID of the stage to be updated
358      * @throws IDNotRecognisedException If no stage exists with
359      *         the requested ID
360      * @throws InvalidStageStateException If the stage is

```

```

349         already waiting for results
350     */
351     public static void updateStageState(int id) throws
352         IDNotRecognisedException,
353         InvalidStageStateException
354     {
355         getStage(id).updateStageState();
356     }
357
358     /**
359     * @param name The new name for the stage instance
360     */
361     public void setStageName(String name) {
362         this.stageName = name;
363     }
364
365     /**
366     * @param id The ID of the stage to be updated
367     * @param name The new name for the stage instance
368     * @throws IDNotRecognisedException If no stage exists with
369     *         the requested ID
370     */
371     public static void setStageName(int id, String name) throws
372         IDNotRecognisedException {
373         getStage(id).setStageName(name);
374     }
375
376     /**
377     * @param description The new description for the stage
378     *         instance
379     */
380     public void setStageDescription(String description) {
381         this.stageDescription = description;
382     }
383
384     /**
385     * @param id The ID of the stage to be updated
386     * @param description The new description for the stage
387     *         instance
388     * @throws IDNotRecognisedException If no stage exists with
389     *         the requested ID
390     */
391     public static void setStageDescription(int id, String
392         description) throws
393         IDNotRecognisedException
394     {
395         getStage(id).setStageDescription(description);
396     }
397
398     /**

```

```

390     * @param length The new length for the stage instance
391     */
392     public void setStageLength(double length) {
393         this.stageLength = length;
394     }
395
396     /**
397     * @param id The ID of the stage to be updated
398     * @param length The new length for the stage instance
399     * @throws IDNotRecognisedException If no stage exists with
400     *         the requested ID
401     */
402     public static void setStageLength(int id, double length)
403         throws
404             IDNotRecognisedException
405         {
406         getStage(id).stageLength = length;
407     }
408
409     /**
410     * @param startTime The new start time for the stage
411     *         instance
412     */
413     public void setStageStartTime(LocalDateTime startTime) {
414         this.stageStartTime = startTime;
415     }
416
417     /**
418     * @param id The ID of the stage to be updated
419     * @param startTime The new start time for the stage
420     *         instance
421     * @throws IDNotRecognisedException If no stage exists with
422     *         the requested ID
423     */
424     public static void setStageStartTime(int id, LocalDateTime
425         startTime)
426         throws
427             IDNotRecognisedException
428         {
429         getStage(id).stageStartTime = startTime;
430     }
431
432     /**
433     * Creates a new stage and adds the ID to the stageIds
434     *         array.
435     *
436     * @param location The location of the new segment
437     * @param type The type of the new segment
438     * @param averageGradient The average gradient of the new
439     *         segment

```

```

429      * @param length The length (in km) of the new segment
430      * @throws InvalidLocationException If the segment finishes
         outside of the
431      *
         bounds of the stage
432      * @throws InvalidStageStateException If the segment state
         is waiting for
433      *
         results
434      * @throws InvalidStageTypeException If the stage type is a
         time-trial
435      *
         (cannot contain
         segments)
436     */
437     public int addSegmentToStage(double location, SegmentType
         type,
438                                double averageGradient, double
         length) throws
439                                InvalidLocationException,
440                                InvalidStageStateException,
441                                InvalidStageTypeException {
442         if(location > this.getStageLength()) {
443             throw new InvalidLocationException("segment
         finishes outside of stage bounds");
444         }
445         if(this.getStageState().equals(StageState.WAITING)) {
446             throw new InvalidStageStateException("stage is
         waiting for results");
447         }
448         if(this.getStageType().equals(StageType.TT)) {
449             throw new InvalidStageTypeException("time trial
         stages cannot contain segments");
450         }
451         Segment newSegment = new Segment(location, type,
         averageGradient, length);
452         this.segmentIds.add(newSegment.getSegmentId());
453         return newSegment.getSegmentId();
454     }
455
456     /**
457      * Creates a new stage and adds the ID to the stageIds
         array.
458      *
459      * @param id The ID of the stage to which the segment will
         be added
460      * @param location The location of the new segment
461      * @param type The type of the new segment
462      * @param averageGradient The average gradient of the new
         segment
463      * @param length The length (in km) of the new segment
464      * @throws IDNotRecognisedException If no stage exists with
         the requested ID

```

```

465      * @throws InvalidLocationException If the segment finishes
         outside of the
466      *
         bounds of the stage
467      * @throws InvalidStageStateException If the segment state
         is waiting for
468      *
         results
469      * @throws InvalidStageTypeException If the stage type is a
         time-trial
470      *
         (cannot contain
         segments)
471      */
472      public static int addSegmentToStage(int id, double location
         , SegmentType type,
473
         double averageGradient,
         double length)
         throws
474         IDNotRecognisedException
         ,
475         InvalidLocationException
         ,
476         InvalidStageStateException
         ,
477         InvalidStageTypeException
         {
478         return getStage(id).addSegmentToStage(location, type,
         averageGradient, length);
479     }
480
481     /**
482     * Removes a segmentId from the array of segmentIds for a
         stage instance,
483     * as well as from the static array of all segments in the
         Segment class.
484     *
485     * @param segmentId The ID of the segment to be removed
486     * @throws IDNotRecognisedException If no segment exists
         with the requested
487     *
         ID
488     */
489     private void removeSegmentFromStage(int segmentId) throws
         IDNotRecognisedException
490     {
491         if(this.segmentIds.contains(segmentId)) {
492             this.segmentIds.remove(segmentId);
493             Segment.removeSegment(segmentId);
494         } else {
495             throw new IDNotRecognisedException("segmentID not
         found in race");
496         }
497     }

```

```

498
499     /**
500      * Removes a segmentId from the array of segmentIds for a
501      * stage instance,
502      * as well as from the static array of all segments in the
503      * Segment class.
504      *
505      * @param id The ID of the stage to which the segment will
506      *           be removed
507      * @param segmentId The ID of the segment to be removed
508      * @throws IDNotRecognisedException If no segment exists
509      *           with the requested
510      *           ID
511      */
512     public static void removeSegmentFromStage(int id, int
513         segmentId) throws
514         IDNotRecognisedException
515     {
516         getStage(id).removeSegmentFromStage(segmentId);
517     }
518
519     /**
520      * Removes a segmentId from the array of segmentIds for a
521      * stage instance,
522      * as well as from the static array of all segments in the
523      * Segment class.
524      *
525      * @param segmentId The ID of the segment to be removed
526      * @throws IDNotRecognisedException If no segment exists
527      *           with the requested
528      *           ID
529      */
530     public static void removeSegment(int segmentId) throws
531         IDNotRecognisedException {
532         for(Stage stage : allStages) {
533             if(stage.segmentIds.contains(segmentId)) {
534                 stage.removeSegmentFromStage(segmentId);
535                 break;
536             }
537         }
538     }
539 }

```

4 StageState.java

```

1 package cycling;
2
3 /**
4  * This enum is used to represent the state of a stage.

```

```

5  *
6  * @author Thomas Newbold
7  * @version 1.0
8  *
9  */
10 public enum StageState {
11
12     /**
13      * Used for stages still in preperation - i.e. segments are
14      * still being
15      * added.
16      */
17     BUILDING,
18
19     /**
20      * Used for stages waiting for results
21      */
22     WAITING;
23 }

```

5 Segment.java

```

1  package cycling;
2
3  import java.io.Serializable;
4  import java.util.ArrayList;
5
6  /**
7   * Segment encapsulates race segments
8   *
9   * @author Thomas Newbold
10  * @version 2.0
11  *
12  */
13 public class Segment implements Serializable {
14     // Static class attributes
15     private static int idMax = 0;
16     public static ArrayList<Integer> removedIds = new ArrayList
17         <Integer>();
18     public static ArrayList<Segment> allSegments = new
19         ArrayList<Segment>();
20
21     /**
22      * Loads the value of idMax.
23      */
24     public static void loadId() {
25         if (Segment.allSegments.size() != 0) {
26             Segment.idMax = Segment.allSegments.get(-1).
27                 getSegmentId() + 1;
28         }
29     }
30 }

```

```

25         } else {
26             Segment.idMax = 0;
27         }
28     }
29
30     /**
31     * @param segmentId The ID of the segment instance to fetch
32     * @return The segment instance with the associated ID
33     * @throws IDNotRecognisedException If no segment exists
34     *                                     with the requested ID
35     */
36     public static Segment getSegment(int segmentId) throws
37                                     IDNotRecognisedException {
38         boolean removed = Segment.removedIds.contains(segmentId
39         );
40         if(segmentId < Segment.idMax && segmentId >= 0 && !
41             removed) {
42             int index = segmentId;
43             for(int j=0; j<Segment.removedIds.size(); j++) {
44                 if(Segment.removedIds.get(j) < segmentId) {
45                     index--;
46                 }
47             }
48             return allSegments.get(index);
49         } else if (removed) {
50             throw new IDNotRecognisedException("no segment
51             instance for "+
52             "segmentId");
53         } else {
54             throw new IDNotRecognisedException("segmentId out
55             of range");
56         }
57     }
58
59     /**
60     * @return An integer array of the segment IDs of all
61     *         segment
62     */
63     public static int[] getAllSegmentIds() {
64         int length = Segment.allSegments.size();
65         int[] segmentIdsArray = new int[length];
66         int i = 0;
67         for(Segment segment : allSegments) {
68             segmentIdsArray[i] = segment.getSegmentId();
69             i++;
70         }
71         return segmentIdsArray;
72     }

```



```

69  /**
70   * @param segmentId The ID of the segment instance to
      remove
71   * @throws IDNotRecognisedException If no segment exists
      with the requested
72   *                                     ID
73   */
74  public static void removeSegment(int segmentId) throws
75                                     IDNotRecognisedException {
76      boolean removed = Segment.removedIds.contains(segmentId
77      );
78      if(segmentId < Segment.idMax && segmentId >= 0 && !
79          removed) {
80          Segment s = getSegment(segmentId);
81          allSegments.remove(s);
82          removedIds.add(segmentId);
83      } else if (removed) {
84          throw new IDNotRecognisedException("no segment
85          instance for "+
86                                             "segmentId");
87      } else {
88          throw new IDNotRecognisedException("segmentId out
89          of range");
90      }
91  }
92
93  // Instance attributes
94  private int segmentId;
95  private double segmentLocation;
96  private SegmentType segmentType;
97  private double segmentAverageGradient;
98  private double segmentLength;
99
100  /**
101   * Segment constructor; creates a new segment and adds to
      allSegment array.
102   *
103   * @param location The location of the finish of the new
      segment in the stage
104   * @param type The type of the new segment
105   * @param averageGradient The average gradient of the new
      segment
106   * @param length The length of the new segment
107   */
108  public Segment(double location, SegmentType type, double
      averageGradient,
109                  double length) {
110      if(Segment.removedIds.size() > 0) {
111          this.segmentId = Segment.removedIds.get(0);
112          Segment.removedIds.remove(0);

```

```

109         } else {
110             this.segmentId = idMax++;
111         }
112         this.segmentLocation = location;
113         this.segmentType = type;
114         this.segmentAverageGradient = averageGradient;
115         this.segmentLength = length;
116         Segment.allSegments.add(this);
117     }
118
119     /**
120     * @return A string representation of the segment instance
121     */
122     public String toString() {
123         String id = Integer.toString(this.segmentId);
124         String location = Double.toString(this.segmentLocation)
125             ;
126         String type;
127         switch (this.segmentType) {
128             case SPRINT:
129                 type = "Sprint";
130                 break;
131             case C4:
132                 type = "Category 4 Climb";
133                 break;
134             case C3:
135                 type = "Category 3 Climb";
136                 break;
137             case C2:
138                 type = "Category 2 Climb";
139                 break;
140             case C1:
141                 type = "Category 1 Climb";
142                 break;
143             case HC:
144                 type = "Hors Categorie";
145                 break;
146             default:
147                 type = "null category";
148         }
149         String averageGrad = Double.toString(this.
150             segmentAverageGradient);
151         String length = Double.toString(this.segmentLength);
152         return String.format("Segment[%s]: %s; %skm; Location=%
153             s; Gradient=%s;",
154             id, type, length, location,
155             averageGrad);
156     }
157
158     /**

```

```

155     * @param id The ID of the segment
156     * @return A string representation of the segment instance
157     * @throws IDNotRecognisedException If no segment exists
158         with the requested
159         ID
160     */
161     public static String toString(int id) throws
162         IDNotRecognisedException {
163         return getSegment(id).toString();
164     }
165
166     /**
167     * @return The integer segmentId for the segment instance
168     */
169     public int getSegmentId() { return this.segmentId; }
170
171     /**
172     * @return The integer representing the location of the
173     segment instance
174     */
175     public double getSegmentLocation() { return this.
176         segmentLocation; }
177
178     /**
179     * @param id The ID of the segment
180     * @return The integer representing the location of the
181     segment instance
182     * @throws IDNotRecognisedException If no segment exists
183     with the requested
184     ID
185     */
186     public static double getSegmentLocation(int id) throws
187         IDNotRecognisedException
188     {
189         return getSegment(id).segmentLocation;
190     }
191
192     /**
193     * @return The type of the segment instance
194     */
195     public SegmentType getSegmentType() { return this.
196         segmentType; }
197
198     /**
199     * @param id The ID of the segment
200     * @return The type of the segment instance
201     * @throws IDNotRecognisedException If no segment exists
202     with the requested
203     ID
204     */

```

```

196     public static SegmentType getSegmentType(int id) throws
197                                     IDNotRecognisedException
198     {
199         return getSegment(id).segmentType;
200     }
201     /**
202     * @return The average gradient of the segment instance
203     */
204     public double getSegmentAverageGradient() {
205         return this.segmentAverageGradient;
206     }
207     /**
208     * @param id The ID of the segment
209     * @return The average gradient of the segment instance
210     * @throws IDNotRecognisedException If no segment exists
211     *         with the requested
212     *             ID
213     */
214     public static double getSegmentAverageGradient(int id)
215                                     throws
216                                     IDNotRecognisedException
217     {
218         return getSegment(id).segmentAverageGradient;
219     }
220     /**
221     * @return The length of the segment instance
222     */
223     public double getSegmentLength() { return this.
224         segmentLength; }
225     /**
226     * @param id The ID of the segment
227     * @return The length of the segment instance
228     * @throws IDNotRecognisedException If no segment exists
229     *         with the requested
230     *             ID
231     */
232     public static double getSegmentLength(int id) throws
233                                     IDNotRecognisedException {
234         return getSegment(id).segmentLength;
235     }
236     /**
237     * @param location The new location for the segment
238     *         instance
239     */
240     public void setSegmentLocation(double location) {

```

```

238         this.segmentLocation = location;
239     }
240
241     /**
242     * @param id The ID of the segment to be updated
243     * @param location The new location for the segment
244     *           instance
245     * @throws IDNotRecognisedException If no segment exists
246     *           with the requested
247     *                               ID
248     */
249     public static void setSegmentLocation(int id, double
250         location) throws
251         IDNotRecognisedException
252     {
253         getSegment(id).setSegmentLocation(location);
254     }
255
256     /**
257     * @param type The new type for the segment instance
258     */
259     public void setSegmentType(SegmentType type) {
260         this.segmentType = type;
261     }
262
263     /**
264     * @param id The ID of the segment to be updated
265     * @param type The new type for the segment instance
266     * @throws IDNotRecognisedException If no segment exists
267     *           with the requested
268     *                               ID
269     */
270     public static void setSegmentType(int id, SegmentType type)
271     throws
272         IDNotRecognisedException
273     {
274         getSegment(id).setSegmentType(type);
275     }
276
277     /**
278     * @param averageGradient The new average gradient for the
279     *           segment instance
280     */
281     public void setSegmentAverageGradient(double
282         averageGradient) {
283         this.segmentAverageGradient = averageGradient;
284     }
285
286     /**
287     * @param id The ID of the segment to be updated

```

```

279      * @param averageGradient The new average gradient for the
      *       segment instance
280      * @throws IDNotRecognisedException If no segment exists
      *       with the requested
281      *                               ID
282      */
283      public static void setSegmentAverageGradient(int id, double
      averageGradient)
284
      throws
      IDNotRecognisedException
      {
285          getSegment(id).setSegmentAverageGradient(
      averageGradient);
286      }
287
288      /**
289      * @param length The new length for the segment instance
290      */
291      public void setSegmentLength(double length) {
292          this.segmentLength = length;
293      }
294
295      /**
296      * @param id The ID of the segment to be updated
297      * @param length The new length for the segment instance
298      * @throws IDNotRecognisedException If no segment exists
      *       with the requested
299      *                               ID
300      */
301      public static void setSegmentLength(int id, double length)
      throws
302
      IDNotRecognisedException
      {
303          getSegment(id).setSegmentLength(length);
304      }
305  }

```

6 Result.java

```

1  package cycling;
2
3  import java.util.ArrayList;
4  import java.util.Arrays;
5  import java.io.Serializable;
6  import java.time.LocalDateTime;
7  import java.time.format.DateTimeFormatter;
8  import java.time.temporal.ChronoUnit;
9
10 /**

```

```

11  * Result encapsulates rider results per stage, and handles
    time adjustments and
12  * rankings (scoring is done externally based on points
    distributions defined in
13  * Cycling Portal)
14  *
15  * @author Thomas Newbold
16  * @version 1.1
17  */
18  public class Result implements Serializable {
19      // Static class attributes
20      public static ArrayList<Result> allResults = new ArrayList<
        Result>();
21
22      /**
23       * @param stageId The ID of the stage
24       * @return An array of all results for a stage
25       */
26      public static Result[] getResultsInStage(int stageId) {
27          ArrayList<Result> stage = new ArrayList<Result>();
28          for(Result r : allResults) {
29              stage.add(r);
30          }
31          stage.removeIf(r -> r.getStageId() != stageId);
32          Result[] resultsForStage = new Result[stage.size()];
33          for(int i=0; i<stage.size(); i++) {
34              resultsForStage[i] = stage.get(i);
35          }
36          return resultsForStage;
37      }
38
39      /**
40       * @param riderId The ID of the driver
41       * @return An array of all results for a driver
42       */
43      public static Result[] getResultsForRider(int riderId) {
44          ArrayList<Result> rider = new ArrayList<Result>(
            allResults);
45          rider.removeIf(r -> r.getRiderId() != riderId);
46          Result[] resultsForRider = new Result[rider.size()];
47          for(int i=0; i<rider.size(); i++) {
48              resultsForRider[i] = rider.get(i);
49          }
50          return resultsForRider;
51      }
52
53      // Instance attributes
54      private int stageId;
55      private int riderId;
56      private LocalTime[] checkpoints;

```

```

57
58 /**
59  * Result constructor; creates a new result entry and adds
60  * to the
61  * allResults array.
62  * @param sId The ID of the stage the result refers to
63  * @param rId The ID of the rider who achieved the result
64  * @param check An array of times at which the rider
65  *             checkpoint (including start and finish)
66  */
67 public Result(int sId, int rId, LocalTime... check) {
68     this.stageId = sId;
69     this.riderId = rId;
70     this.checkpoints = check;
71     Result.allResults.add(this);
72 }
73
74 /**
75  * @return A string representation of the Result instance
76  */
77 public String toString() {
78     String sId = Integer.toString(this.stageId);
79     String rId = Integer.toString(this.riderId);
80     int l = this.getCheckpoints().length;
81     String times[] = new String[l];
82     DateTimeFormatter formatter = DateTimeFormatter.
83         ofPattern("HH:mm:ss");
84     for(int i=0; i<l; i++) {
85         times[i] = this.getCheckpoints()[i].format(
86             formatter);
87     }
88     return String.format("Stage[%s]-Rider[%s]: SplitTimes=%s; Total=%s",
89         sId, rId, Arrays.toString(times),
90         getTotalElapsed().format(formatter));
91 }
92
93 /**
94  * @param sId The ID of the stage of the result instance
95  * @param rId The ID of the associated rider to the result
96  *             instance
97  * @return The Result instance
98  * @throws IDNotRecognisedException If an instance for the
99  *             rider/stage
100  *             combination is not
101  *             found in the
102  *             allResults array

```



```

98      */
99      public static Result getResult(int sId, int rId) throws
        IDNotRecognisedException {
100          for(Result r : allResults) {
101              if(r.getRiderId()==rId && r.getStageId()==sId) {
102                  return r;
103              }
104          }
105          throw new IDNotRecognisedException("results not found
            for rider in stage");
106      }
107
108      /**
109       * @param sId The ID of the stage of the result instance to
            remove
110       * @param rId The ID of the associated rider to the result
            instance to remove
111       * @throws IDNotRecognisedException If an instance for the
            rider/stage
112       *                                     combination is not
            found in the
113       *                                     allResults array
114       */
115      public static void removeResult(int sId, int rId) throws
        IDNotRecognisedException {
116          for(Result r : allResults) {
117              if(r.getRiderId()==rId && r.getStageId()==sId) {
118                  allResults.remove(r);
119                  break;
120              }
121          }
122          throw new IDNotRecognisedException("results not found
            for rider in stage");
123      }
124
125      /**
126       * @return The stageId of the stage the result refers to
127       */
128      public int getStageId() { return this.stageId; }
129
130      /**
131       * @return The riderId of the rider associated with the
            result
132       */
133      public int getRiderId() { return this.riderId; }
134
135      /**
136       * @return An array of the split times between each
            checkpoint
137       */

```

```

138     public LocalTime[] getCheckpoints() {
139         LocalTime[] out = new LocalTime[this.checkpoints.length
140             -1];
141         for(int n=0;n<this.checkpoints.length-1; n++) {
142             out[n] = getElapsed(checkpoints[n],checkpoints[n
143                 +1]);
144         }
145         return out;
146     }
147     /**
148      * @return The total time elapsed between the start and end
149      * checkpoints
150      */
151     public LocalTime getTotalElapsed() {
152         LocalTime[] times = this.checkpoints;
153         return Result.getElapsed(times[0], times[times.length
154             -1]);
155     }
156     /**
157      * @param a Start time
158      * @param b End time
159      * @return The time difference between two times, a and b
160      */
161     public static LocalTime getElapsed(LocalTime a, LocalTime b
162     ) {
163         int hours = (int)a.until(b, ChronoUnit.HOURS);
164         int minutes = (int)a.until(b, ChronoUnit.MINUTES);
165         int seconds = (int)a.until(b, ChronoUnit.SECONDS);
166         return LocalTime.of(hours%24, minutes%60, seconds%60);
167     }
168     /**
169      * @return An array of the checkpoint times, adjusted to a
170      * threshold of
171      * one second
172      */
173     public LocalTime[] adjustedCheckpoints() {
174         LocalTime[] adjusted = this.getCheckpoints();
175         for(int n=0; n<adjusted.length; n++) {
176             adjusted[n] = adjustedCheckpoint(n);
177         }
178         return adjusted;
179     }
180     /**
181      * Recursive adjuster, used in {@link #adjustedCheckpoints
182      * ()}.
183      *

```

```

181     * @param n The index of the checkpoint to adjust
182     * @return The adjusted time for checkpoint n
183     */
184     public LocalTime adjustedCheckpoint(int n) {
185         for(int i=0; i<allResults.size(); i++) {
186             Result r = allResults.get(i);
187             if(r.getRiderId()==this.getRiderId() && r.
               getStageId()==this.getStageId()) {
188                 continue;
189             }
190             LocalTime selfTime = this.getCheckpoints()[n];
191             LocalTime rTime = r.getCheckpoints()[n];
192             if(selfTime.until(rTime, ChronoUnit.SECONDS)<1) {
193                 return r.adjustedCheckpoint(n);
194             } else {
195                 return selfTime;
196             }
197         }
198         return null;
199     }
200 }

```

7 Team.java

```

1 package cycling;
2 import java.io.Serializable;
3 import java.util.ArrayList;
4 /**
5  * Team Class holds the teamId,name,description and riderIds
   belonging to that team.
6  *
7  *
8  * @author Ethan Ray
9  * @version 1.0
10 *
11 */
12
13 public class Team implements Serializable {
14     public static ArrayList<String> teamNames = new ArrayList
        <>();
15     public static int teamTopId = 0;
16
17     private int teamID;
18     private String name;
19     private String description;
20     private ArrayList<Integer> riderIds = new ArrayList<>();
21
22
23     /**

```

```

24      * @param name String - A name for the team, , If the name
      is null, empty, has more than 30 characters, or has
      white spaces will throw InvaildNameException.
25      * @param description String - A description for the team.
26      * @throws IllegalNameException name String - Is a
      duplicate name of any other Team, IllegalNameException
      will be thrown.
27      * @throws InvaildNameException name String - If the name is
      null, empty, has more than 30 characters, or has white
      spaces will throw InvaildNameException.
28      */
29      public Team(String name, String description) throws
      IllegalNameException, InvalidNameException
30      {
31          if (name == "" || name.length()>30 || name.contains(" ")
32              ){
33              throw new InvalidNameException("Team name cannot be
34              empty, longer than 30 characters , or has white
35              spaces.");
36          }
37          for (int i = 0;i<teamNames.size();i++){
38              if (teamNames.get(i) == name){
39                  throw new IllegalNameException("That team name
40                  already exists!");
41              }
42          }
43          teamNames.add(name);
44          this.teamID = teamTopId++;
45          this.name = name;
46          this.description = description;
47      }
48      /**
49      * @param rider Rider - A rider to add to the team.
50      */
51      public void addRider(Rider rider){
52          this.riderIds.add(rider.getRiderId());
53      }
54      /**
55      * @param riderId int - A riderId to be removed from the
56      team.
57      */
58      public void removeRiderId(int riderId){
59          for (int i =0;i<this.riderIds.size();i++){
60              if (this.riderIds.get(i)==riderId){
61                  this.riderIds.remove(i);
62                  break;
63              }
64          }
65      }

```

```

62     }
63     /**
64      * @return An Array of integers - which are the riderIds in
        that team.
65      */
66     public int[] getRiderIds() {
67         int [] currentRiderIds = new int[this.riderIds.size()];
68         for (int i=0; i<this.riderIds.size();i++){
69             currentRiderIds[i]=this.riderIds.get(i);
70         }
71         return currentRiderIds;
72     }
73     /**
74      * @return A Integer - teamId of the team.
75      */
76     public int getId() {
77         return this.teamID;
78     }
79     /**
80      * @return A String - Name of the team.
81      */
82     public String getTeamName() {
83         return this.name;
84     }
85     /**
86      * @return A String - The description of the team.
87      */
88     public String getDescription() {
89         return this.description;
90     }
91 }

```

8 Rider.java

```

1  package cycling;
2
3  import java.io.Serializable;
4
5  /**
6   * Rider Class holds the riders teamId,riderId,name and
        yearOfBirth
7   *
8   *
9   * @author Ethan Ray
10  * @version 1.0
11  *
12  */
13
14

```

```

15 public class Rider implements Serializable {
16     public static int ridersTopId;
17     private int riderId;
18     private int teamID;
19     private String name;
20     private int yearOfBirth;
21
22
23     /**
24      * @param teamID int - A team Id that the rider will belong
25      *    too
26      * @param name String - A name for the rider, Has to be non
27      *    -null or IllegalArgumentException is thrown.
28      * @param yearOfBirth int - A year that the rider was born
29      *    in. Has to be above 1900 or IllegalArgumentException is
30      *    thrown.
31      * @throws IllegalArgumentException name String - Has to be
32      *    non-null or IllegalArgumentException is thrown.
33      * @throws IllegalArgumentException yearOfBirth int - A
34      *    year that the rider was born in. Has to be above 1900
35      *    or IllegalArgumentException is thrown.
36     */
37     public Rider(int teamID, String name, int yearOfBirth)
38         throws IllegalArgumentException
39     {
40         this.riderId = ridersTopId++;
41         this.teamID = teamID;
42         if (name == "" || name == null){
43             throw new IllegalArgumentException("Illegal name
44             entered for rider");
45         }
46         this.name = name;
47         if (yearOfBirth < 1900){
48             throw new IllegalArgumentException("Illegal value
49             for yearOfBirth given please enter a value above
50             1900.");
51         }
52         this.yearOfBirth = yearOfBirth;
53     }
54
55     /**
56      * @return The RiderId of the rider.
57     */
58     public int getRiderId(){
59         return this.riderId;
60     }
61
62     /**
63      * @return The team Id that the rider belongs to/
64     */
65     public int getRiderTeamId(){
66         return this.teamID;
67     }

```

```

54     }
55     /**
56      * @return The rider's name.
57      */
58     public String getRiderName() {
59         return this.name;
60     }
61     /**
62      * @return The the year of birth of the rider.
63      */
64     public int getRiderYOB() {
65         return this.yearOfBirth;
66     }
67
68 }

```

9 RiderManager.java

```

1  package cycling;
2
3  import java.io.Serializable;
4  import java.util.ArrayList;
5
6  public class RiderManager implements Serializable{
7      public static ArrayList<Rider> allRiders = new ArrayList
8          <>();
9      public static ArrayList<Team> allTeams = new ArrayList<>();
10
11     /**
12      * @param teamID int - A team Id that the rider will belong
13      * too. If the ID doesn't exist IDNotRecognisedException
14      * is thrown.
15      * @param name String - A name for the rider, Has to be non
16      * -null or IllegalArgumentException is thrown.
17      * @param yearOfBirth int - A year that the rider was born
18      * in. Has to be above 1900 or IllegalArgumentException is
19      * thrown.
20      * @return riderId of the rider created.
21      * @throws IDNotRecognisedException teamId int - If the ID
22      * doesn't exist IDNotRecognisedException is thrown.
23      * @throws IllegalArgumentException yearOfBirth int - A
24      * year that the rider was born in. Has to be above 1900
25      * or IllegalArgumentException is thrown.
26      */
27     int createRider(int teamID, String name, int yearOfBirth)
28         throws IDNotRecognisedException, IllegalArgumentException
29     {
30         int teamIndex = getIndexForTeamId(teamID);

```

```

21         Rider newRider = new Rider(teamID,name,yearOfBirth);
22         allRiders.add(newRider);
23         Team ridersTeam = allTeams.get(teamIndex);
24         ridersTeam.addRider(newRider);
25         return newRider.getRiderId();
26     }
27     /**
28      * @param riderId int - A riderId of a rider to be removed.
29      * If the ID doesn't exist IDNotRecognisedException is
30      * thrown.
31      * @throws IDNotRecognisedException riderId int - If the ID
32      * doesn't exist IDNotRecognisedException is thrown.
33      */
34     void removeRider(int riderId) throws
35         IDNotRecognisedException
36     {
37         int riderIndex = getIndexForRiderId(riderId);
38         int teamId = allRiders.get(riderIndex).getRiderTeamId()
39         ;
40         int teamIndex = getIndexForTeamId(teamId);
41         Team riderTeam = allTeams.get(teamIndex);
42         riderTeam.removeRiderId(riderId);
43         allRiders.remove(riderIndex);
44     }
45     /**
46      * @param riderId int - A riderId of a rider to be searched
47      * for. If the ID doesn't exist IDNotRecognisedException
48      * is thrown.
49      * @throws IDNotRecognisedException riderId int - If the ID
50      * doesn't exist IDNotRecognisedException is thrown.
51      * @return An int which is the index that maps to the
52      * riderId.
53      */
54     int getIndexForRiderId(int riderId) throws
55         IDNotRecognisedException{
56         int index =-1;
57         if (allRiders.size() == 0){
58             throw new IDNotRecognisedException("No rider exists
59             with that ID");
60         }
61         for (int i=0; i<allRiders.size();i++){
62             if (allRiders.get(i).getRiderId()==riderId){
63                 index = i;
64                 break;
65             }
66         }
67         if (index == -1){
68             throw new IDNotRecognisedException("No rider exists
69             with that ID");
70         }
71     }

```



```

59         return index;
60     }
61     /**
62     * @param name String - A name for the team, , If the name
        is null, empty, has more than 30 characters, or has
        white spaces will throw InvaildNameException.
63     * @param description String - A description for the team.
64     * @throws IllegalNameException name String - Is a
        duplicate name of any other Team, IllegalNameException
        will be thrown.
65     * @throws InvailNameException name String - If the name is
        null, empty, has more than 30 characters, or has white
        spaces will throw InvaildNameException.
66     */
67     int createTeam(String name, String description) throws
        IllegalNameException, InvalidNameException{
68         Team newTeam = new Team(name,description);
69         allTeams.add(newTeam);
70         return newTeam.getId();
71     }
72     /**
73     * @param teamId int - A teamId of a rider to be removed.
        If the ID doesn't exist IDNotRecognisedException is
        thrown.
74     * @throws IDNotRecognisedException riderId int - If the ID
        doesn't exist IDNotRecognisedException is thrown.
75     */
76     void removeTeam(int teamId) throws IDNotRecognisedException
        { // Delete team and all riders in that team
77         int teamIndex = getIndexForTeamId(teamId);
78         Team currentTeam = allTeams.get(teamIndex);
79         for (Integer riderId : currentTeam.getRiderIds()) {
80             removeRider(riderId);
81         }
82         allTeams.remove(teamIndex);
83     }
84 }
85 /**
86 * @return All the teamId's that are currently in the
        system as an int[]
87 */
88
89 int[] getTeams(){
90     int [] allTeamIds = new int[allTeams.size()];
91     for (int i=0; i<allTeams.size();i++){
92         allTeamIds[i]=allTeams.get(i).getId();
93     }
94     return allTeamIds;
95 }
96 /**

```

```

97      * @param teamId int - A teamId to get RidersId in that
      team. If the ID doesn't exist IDNotRecognisedException
      is thrown.
98      * @throws IDNotRecognisedException teamId int - If the ID
      doesn't exist IDNotRecognisedException is thrown.
99      * @return All the riderId's in a team as an int[]
100     */
101     int[] getTeamRiders(int teamId) throws
        IDNotRecognisedException{
102         Team currentTeam = getTeam(teamId);
103         return currentTeam.getRiderIds();
104     }
105
106     /**
107      * @return All team names in the system as an String[]
108      */
109     String[] getTeamsNames(){
110         String [] allTeamNames = new String[allTeams.size()];
111         for (int i=0; i<allTeams.size();i++){
112             allTeamNames[i] = allTeams.get(i).getTeamName();
113         }
114         return allTeamNames;
115     }
116
117     /**
118      * @return All rider names in the system as an String[]
119      */
120     String[] getRidersNames(){
121         String [] allRiderNames = new String[allRiders.size()];
122         for (int i=0; i<allRiders.size();i++){
123             allRiderNames[i] = allRiders.get(i).getRiderName();
124         }
125         return allRiderNames;
126     }
127
128     /**
129      * @param teamId int - A teamId of a team to search for its
      index. If the ID doesn't exist
      IDNotRecognisedException is thrown.
130      * @throws IDNotRecognisedException teamId int - If the ID
      doesn't exist IDNotRecognisedException is thrown.
131      * @return An int which is the index that maps to the
      teamId.
132      */
133     int getIndexForTeamId(int teamId) throws
        IDNotRecognisedException{
134         int index = -1;
135         if (allTeams.size() == 0){
136             throw new IDNotRecognisedException("No Team exists
            with that ID");
137         }
138         for (int i=0; i<allTeams.size();i++){

```

```

137         if (allTeams.get(i).getId()==teamId) {
138             index = i;
139             break;
140         }
141     }
142     if (index == -1){
143         throw new IDNotRecognisedException("No rider exists
144             with that ID");
145     }
146     return index;
147 }
148 /**
149  * @param teamId int - A teamId of a team to search for its
150  * object. If the ID doesn't exist
151  * IDNotRecognisedException is thrown.
152  * @throws IDNotRecognisedException teamId int - If the ID
153  * doesn't exist IDNotRecognisedException is thrown.
154  * @return A Team object with the teamId parsed.
155  */
156 Team getTeam(int teamId) throws IDNotRecognisedException{
157     int teamIndex = getIndexForTeamId(teamId);
158     return allTeams.get(teamIndex);
159 }
160 /**
161  * @param riderId int - A riderId of a team to search for
162  * its object. If the ID doesn't exist
163  * IDNotRecognisedException is thrown.
164  * @throws IDNotRecognisedException riderId int - If the ID
165  * doesn't exist IDNotRecognisedException is thrown.
166  * @return A Rider object with the riderId parsed.
167  */
168 Rider getRider(int riderId) throws IDNotRecognisedException
169 {
170     int riderIndex = getIndexForRiderId(riderId);
171     return allRiders.get(riderIndex);
172 }
173 void setAllTeams(ArrayList<Team> allTeams){
174
175     RiderManager.allTeams = allTeams;
176     if (allTeams.size() != 0){
177         Team lastTeam = allTeams.get(allTeams.size()-1);
178         Team.teamTopId = lastTeam.getId()+1;
179     }
180 }
181 void setAllRiders(ArrayList<Rider> allRiders){
182     RiderManager.allRiders = allRiders;
183     if (allRiders.size() != 0){
184         Rider lastRider = allRiders.get(allRiders.size()-1)
185             ;
186         Rider.ridersTopId = lastRider.getRiderId()+1;

```

```
178         }
179     }
180     int [] getRiderIds(){
181         int[] riderIdArray = new int[allRiders.size()];
182         int count = 0;
183         for (Rider rider : RiderManager.allRiders){
184             riderIdArray[count] = rider.getRiderId();
185             count++;
186         }
187     }
188     return riderIdArray;
189 }
190
191 }
```