```
1  package cycling;
2
3  import java.io.*;
4  import java.io.IOException;
5  import java.time.LocalDateTime;
6  import java.time.LocalTime;
7  import java.util.ArrayList;
8  import java.util.List;
9
10 /**
11  * CyclingPortal is a minimally compiling, but non-functioning implementor
12  * of the CyclingPortalInterface interface.
13  *
14  * @author Jude Goulding & Bethany Whiting
15  * @version 1.13.6
16  *
17  */
18 public class CyclingPortal implements CyclingPortalInterface{
19         private ArrayList<Rider> ridersInternal = new ArrayList<Rider>();
20         private ArrayList<Team> teamsInternal = new ArrayList<Team>();
21         private ArrayList<Race> racesInternal = new ArrayList<Race>();
22         private ArrayList<Result> resultsInternal = new ArrayList<Result>();
23         private int currentStageID = 0;
24         private int currentSegmentID = 0;
25         private int currentRaceID = 0;
26         private int currentTeamID = 0;
27         private int currentRiderID = 0;
28
29         public CyclingPortal(){
30                 ridersInternal = new ArrayList<Rider>();
31                 teamsInternal = new ArrayList<Team>();
```

```
32                racesInternal = new ArrayList<Race>();
33                resultsInternal = new ArrayList<Result>();
34                currentStageID = 0;
35                currentSegmentID = 0;
36                currentRaceID = 0;
37                currentTeamID = 0;
38                currentRiderID = 0;
39        }
40        /**
41        * This method gets the races that are currently in the system
42        *
43        * @return An array of raceIds in the system or a empty array if none are in the system
44        */
45        @Override
46        public int[] getRaceIds() {
47                int[] races = new int[racesInternal.size()];
48                for(int i = 0; i < racesInternal.size(); i++ ){
49                        races[i] = racesInternal.get(i).getRaceID();
50                }
51                return races;
52        }
53
54        /**
55        * This method checks to see if the name of the race exists
56        *
57        * @param name : Race's name
58        * @param arraySet race : List of races within the system
59        * @return boolean value depending on if the name is already in the system
60        */
61        public boolean nameRaceExists(String name, List<Race> arraySet){
62                for (Race a : arraySet){
```

```
63                              if (name == a.getName()){
64                                      return true;
65                              }
66                      }
67                      return false;
68              }
69
70              /**
71              * This method checks to see if the name of the race exists
72              *
73              * @param name : Team's name
74              * @param arraySet race : List of teams within the system
75              * @return boolean value depending on if the name is already in the system
76              */
77              public boolean nameTeamExists(String name, List<Team> arraySet){
78                      for (Team a : arraySet){
79                              if (name == a.getName()){
80                                      return true;
81                              }
82                      }
83                      return false;
84              }
85
86              /**
87              * Checks to see if the name given for a race is valid, not longer then 30 characters and is not
88      blank
89              *
90              * @param name : Race's name
91              * @return boolean value depending on if the name is invalid, true if invalid and false if valid
92              */
93              public boolean nameInValid(String name){
```

```
94                      if (name.length() > 30 || name.contains("\s") || name.contains(" ") ||
95  name.equals(null) || name.equals("")){
96                              return true;
97                      }
98                      return false;
99              }
100
101             /**
102             * This method creates the race within the system with the name and description given
103             *
104             * @param name : Race's name
105             * @param description : Race's description
106             * @throws IllegalNameException : If the name given is already within the system
107             * @throws InvalidNameException : If the name given is blank or is more then 30 characters
108 long
109             * @return A unique Id for the race created
110             */
111             @Override
112             public int createRace(String name, String description) throws IllegalNameException,
113 InvalidNameException {
114                     if (nameRaceExists(name, racesInternal)){
115                             throw new IllegalNameException("That name is already taken");
116                     }
117                     if (nameInValid(name)){
118                             throw new InvalidNameException("That name is invalid");
119                     }
120                     Race newRace = new Race(currentRaceID++, name, description);
121                     racesInternal.add(newRace);
122                     return racesInternal.get(racesInternal.size()-1).getRaceID();
123             }
124
125             /**
```

```
126         * Getting the details about a race from its ID
127         *
128         * @param raceId : The ID of the race wishing to find details on
129         * @throws IDNotRecognisedException : If the ID is not within the system
130         * @return A formatted string in the form raceID,name,description,stageLength
131         */
132         @Override
133         public String viewRaceDetails(int raceId) throws IDNotRecognisedException {
134                 for (Race a : racesInternal){
135                         if (a.getRaceID() == raceId){
136                                 return a.toString();
137                         }
138                 }
139                 throw new IDNotRecognisedException();
140         }
141
142         /**
143         * This method removes the race and all the details about it
144         *
145         * @param raceId : The Id is the race wanted to be removed
146         * @throws IDNotRecgonisedException : If the Id is not within the system
147         */
148         @Override
149         public void removeRaceById(int raceId) throws IDNotRecognisedException {
150                 for (int index = 0; index < racesInternal.size(); index++){
151                         if (racesInternal.get(index).getRaceID() == raceId){
152                                 racesInternal.remove(index);
153                         }
154                 }
155                 throw new IDNotRecognisedException();
156         }
```

```
157
158             /**
159             * The method gets the number of stages within a race
160             *
161             * @param raceId : The ID of the race that is being queried
162             * @throws IDNotRecognisedException : If the Id is not within the system
163             * @return The number of stages that have been created for the race
164             */
165             @Override
166             public int getNumberOfStages(int raceId) throws IDNotRecognisedException {
167                     for (Race a : racesInternal){
168                             if (a.getRaceID() == raceId){
169                                     return a.getStages().size();
170                             }
171                     }
172                     throw new IDNotRecognisedException();
173             }
174
175             /**
176             * This creates a new stage within a race
177             *
178             * @param raceId : The Id of the race that stage wants to be added to
179             * @param stageName : The name for the stage
180             * @param description : A description for the stage
181             * @param length : The length of the stage in kilometers
182             * @param startTime : The data and time at which the stage will be raced
183             * @throws IDNotRecognisedException : If the Id is not within the system
184             * @throws IllegalNameException : If the name given is blank or is more then 30 characters
185     long
186             * @throws InvalidLengthException : If the length is less than 5km
187             * @return A unique Id for the stage created
```

```
188             */
189             @Override
190             public int addStageToRace(int raceId, String stageName, String description, double length,
191     LocalDateTime startTime,
192                                     StageType type)
193                                     throws IDNotRecognisedException, IllegalNameException,
194     InvalidNameException, InvalidLengthException {
195                     if (nameInValid(stageName)){
196                             throw new InvalidNameException("Invalid name");
197                     }
198                     if (length < 5.0){
199                             throw new InvalidLengthException();
200                     }
201                     for (Race a : racesInternal){
202                             if (a.getRaceID() == raceId){
203                                     if ( a.getStages() != null){
204                                             for (Stage b : a.getStages()){
205                                                     if (b.getName() == stageName){
206                                                             throw new IllegalNameException();
207                                                             }
208                                                     }
209                                             }
210                                     return a.addStage(currentStageID++, stageName, description,
211     length, startTime, type);
212                                     }
213                             }
214                     throw new IDNotRecognisedException();
215             }
216
217             /**
218             * Gets a list of the id of the stages that are in a race
219             *
```

```
220         * @param raceId : The Id of the race that is wanted
221         * @throws IdNotRecognisedException : If the Id is not within the system
222         * @return The list of stage IDs within the wanted race, ordered from first to last.
223         */
224         @Override
225         public int[] getRaceStages(int raceId) throws IDNotRecognisedException {
226                 for (Race a : racesInternal){
227                         if (a.getRaceID() == raceId){
228                                 return a.getStageIDs();
229                         }
230                 }
231                 throw new IDNotRecognisedException();
232         }
233
234         /**
235         * This method gets the length of a stage in km
236         *
237         * @param stageId : The ID of the stage needed
238         * @throws IdNotRecognisedException : If the Id is not within the system
239         * @return The length of the stage
240         */
241         @Override
242         public double getStageLength(int stageId) throws IDNotRecognisedException {
243                 for (Race a : racesInternal){
244                         for (Stage b : a.getStages()){
245                                 if(b.getStageId() == stageId){
246                                         return b.getStageLength();
247                                 }
248                         }
249                 }
250                 throw new IDNotRecognisedException();
```

```
251             }
252
253             /**
254             * The method removes a stage and all the details relating to it
255             *
256             * @param stageId : The ID of the stage wanted
257             * @throws IdNotRecognisedException : If the Id is not within the system
258             */
259             @Override
260             public void removeStageById(int stageId) throws IDNotRecognisedException {
261                     for (Race a : racesInternal){
262                             for (int b : a.getStageIDs()){
263                                     if(b == stageId){
264                                             a.removeStage(b);
265                                     }
266                             }
267                     }
268                     throw new IDNotRecognisedException();
269
270             }
271
272             /**
273             * Adds a climb segment to a stage
274             *
275             * @param stageId : The ID of the stage that you want to add a climb to
276             * @param location : The location in km where the climb finishes within the stage
277             * @param type :  The type of the climb - {@link SegmentType#C4}, {@link
278     SegmentType#C3}, {@link SegmentType#C2}, {@link SegmentType#C1}, or {@link
279     SegmentType#HC}.
280             * @param averagerGradient : The average gradient of the climb
281             * @param length : The length of the climb in km
282             * @throws IDNotRecognisedException : If the Id is not within the system
```

```
283             * @throws InvalidLocationException : If the location is not within the length of the stage
284             * @throws InvalidStageStateException : The stage is waiting for results
285             * @throws InvalidStageTypeException : Time-trial stages cannot contain any segments
286             * @return The ID of the segment created
287             */
288             @Override
289             public int addCategorizedClimbToStage(int stageId, Double location, SegmentType type,
290     Double averageGradient,
291                                     Double length) throws IDNotRecognisedException, InvalidLocationException,
292     InvalidStageStateException,
293                                     InvalidStageTypeException {
294
295                     for (Race a : racesInternal){
296                             for (Stage b : a.getStages()){
297                                     if(b.getStageId() == stageId){
298                                             if (location < 0 || location >= b.getStageLength()){
299                                                     throw new InvalidLocationException();
300                                             }
301                                             else if (b.getStageType() == StageType.TT){
302                                                     throw new InvalidStageTypeException();
303                                             }
304                                             else if (b.isWaiting()){
305                                                     throw new InvalidStageStateException();
306                                             }
307                                             b.addCatClimbSegment(currentSegmentID++, location, type,
308     averageGradient, length);
309                                     }
310                             }
311                     }
312                     throw new IDNotRecognisedException();
313             }
314
```

```
315         /**
316         * Adds a sprint segment to a stage
317         *
318         * @param stageId : The Id of the stage you wish to put the sprint into
319         * @param location : The location in km where the sprint finshed within the stage
320         * @throws IDNotRecognisedException : If the Id is not within the system
321         * @throws InvalidLocationException : If the location is not within the length of the stage
322         * @throws InvalidStageStateException : The stage is waiting for results
323         * @throws InvalidStageTypeException : Time-trial stages cannot contain any segments
324         * @return The ID of the segment created
325         */
326         @Override
327         public int addIntermediateSprintToStage(int stageId, double location) throws
328 IDNotRecognisedException,
329                         InvalidLocationException, InvalidStageStateException,
330 InvalidStageTypeException {
331                 for (Race a : racesInternal){
332                         for (Stage b : a.getStages()){
333                                 if(b.getStageId() == stageId){
334                                         if (location < 0 || location >= b.getStageLength()){
335                                                 throw new InvalidLocationException();
336                                         }
337                                         else if (b.getStageType() == StageType.TT){
338                                                 throw new InvalidStageTypeException();
339                                         }
340                                         else if (b.isWaiting()){
341                                                 throw new InvalidStageStateException();
342                                         }
343                                         b.addSprintSegment(currentSegmentID++, location);
344                                 }
345                         }
346                 }
```

```
347                     throw new IDNotRecognisedException();
348             }
349
350             /**
351             * Removes a segment from a stage
352             *
353             * @param segmentId : The ID of the segment wishing to be you removed
354             * @throws IDNotRecognisedException : If the Id is not within the system
355             * @throws InvalidStageStateException : The stage is waiting for results
356             */
357             @Override
358             public void removeSegment(int segmentId) throws IDNotRecognisedException,
359     InvalidStageStateException {
360                     for (Race a : racesInternal){
361                             for (Stage b : a.getStages()){
362                                     for (int c = 0; c < b.getSegments().size(); c++){
363                                             if (b.getSegments().get(c).getSegmentID() == segmentId){
364                                                     if(b.isWaiting()){
365                                                             throw new InvalidStageStateException();
366                                                     }
367                                                     b.removeSeg(c);
368                                             }
369                                     }
370                             }
371                     }
372                     throw new IDNotRecognisedException();
373             }
374
375             /**
376             * This concludes the preparation of a stage and will make the state of the stage "waiting for
377     results"
378             *
```

```
379             * @param stageId : The ID of the stage to be concluded
380             * @throws IDNotRecognisedException : If the Id is not within the system
381             * @throws InvalidStageStateException : The stage is waiting for results
382             */
383             @Override
384             public void concludeStagePreparation(int stageId) throws IDNotRecognisedException,
385     InvalidStageStateException {
386                     for (Race a : racesInternal){
387                             for (Stage b : a.getStages()){
388                                     if (b.getStageId() == stageId){
389                                             if(b.isWaiting()){
390                                                     throw new InvalidStageStateException();
391                                             }
392                                             b.setWaiting();
393                                     }
394                             }
395                     }
396                     throw new IDNotRecognisedException();
397             }
398
399             /**
400             * Gets all the segments within a stage
401             *
402             * @param stageId : The ID of the stage wanted
403             * @throws IDNotRecognisedException : If the Id is not within the system
404             * @return A list of segment IDs ordered by there location within the stage
405             */
406             @Override
407             public int[] getStageSegments(int stageId) throws IDNotRecognisedException {
408                     for (Race a : racesInternal){
409                             for (Stage b : a.getStages()){
```

```
410                                     return b.getSegmentIDs();
411                             }
412                     }
413                     throw new IDNotRecognisedException();
414             }
415
416             /**
417             * Creates a team using a name and desciption
418             *
419             * @param name : The name of the team
420             * @param description : The desciption of the team
421             * @throws IllegalNameException : If the name given is already within the system
422             * @throws InvalidNameException : If the name given is blank or is more then 30 characters
423     long
424             * @return A ID for the team
425             */
426             @Override
427             public int createTeam(String name, String description) throws IllegalNameException,
428     InvalidNameException {
429                     if (nameTeamExists(name, teamsInternal)){
430                             throw new IllegalNameException();
431                     }
432                     else if (nameInValid(name)){
433                             throw new InvalidNameException();
434                     }
435                     Team team = new Team(currentTeamID++, name, description);
436                     teamsInternal.add(team);
437                     return teamsInternal.get(teamsInternal.size()-1).getTeamID();
438             }
439
440             /**
441             * Removes a team
```

```
442         *
443         * @param teamId : The ID of the team to be removed
444         * @throws IDNotRecognisedException : If the Id is not within the system
445         */
446         @Override
447         public void removeTeam(int teamId) throws IDNotRecognisedException {
448                 for(int a = 0; a < teamsInternal.size(); a++){
449                         if (teamsInternal.get(a).getTeamID() == teamId){
450                                 teamsInternal.remove(a);
451                         }
452                 }
453         }
454
455         /**
456         * Gets all the team IDs in the system
457         *
458         * @return The list of IDs of teams
459         */
460         @Override
461         public int[] getTeams() {
462                 int teamsSize = teamsInternal.size();
463                 int[] teamIDs = new int[teamsSize];
464                 for (int a = 0; a < teamsSize; a++){
465                         teamIDs[a] = teamsInternal.get(a).getTeamID();
466                 }
467                 return teamIDs;
468         }
469
470         /**
471         * Get the riders within a team
472         *
```

```
473             * @param teamId : The Id of the team wanted
474             * @throws IDNotRecognisedException : If the Id is not within the system
475             * @return A list of hte riders' ID
476             */
477             @Override
478             public int[] getTeamRiders(int teamId) throws IDNotRecognisedException {
479                     for(Team a : teamsInternal){
480                             if(a.getTeamID() == teamId){
481                                     return a.getRiders();
482                             }
483                     }
484                     throw new IDNotRecognisedException();
485             }
486
487             /**
488             * Creates a rider
489             *
490             * @param teamID : The ID rider's team
491             * @param name : The name of the rider
492             * @param yearOfBirth : The year of birth of the rider
493             * @throws IDNotRecognisedException : If the Id is not within the system
494             * @throws IllegalArgumentException If the name of the rider is null or the year of birth is
495     less than 1900.
496             * @return The ID of the rider created
497             */
498             @Override
499             public int createRider(int teamID, String name, int yearOfBirth)
500                             throws IDNotRecognisedException, IllegalArgumentException {
501                     for(Team a : teamsInternal){
502                             if(a.getTeamID() == teamID){
503                                     if (name == null || yearOfBirth <= 1900){
```

```
504                                         throw new IllegalArgumentException();
505                                 }
506
507                                 Rider temp = new Rider(currentRiderID++, name, yearOfBirth,
508 teamID);
509                                 ridersInternal.add(temp);
510                                 return a.addRider(temp.getRiderID());
511                         }
512                 }
513                 throw new IDNotRecognisedException();
514
515         }
516
517         /**
518         * Removes a rider from the system. When a rider is removed from the platform, all of its
519 results should be also removed. Race results must be updated.
520         *
521         * @param riderId : The ID of the rider to be removed.
522         * @throws IDNotRecognisedException : If the ID does not match to any rider in the system.
523         */
524         @Override
525         public void removeRider(int riderId) throws IDNotRecognisedException {
526                 for (int a = 0; a < ridersInternal.size(); a++){
527         if(ridersInternal.get(a).getRiderID() == riderId){
528
529                                 //remove from system
530                                 Rider riderToRemove = ridersInternal.get(a);
531                                 ridersInternal.remove(a);
532
533                                 //remove from team
534                                 for(Team b : teamsInternal){
535                                         if(b.getTeamID() == riderToRemove.getTeamID()){
```

```
536                                                     b.removeRider(riderToRemove.getRiderID());
537
538                                             }
539                                     }
540
541                                     //remove results
542                                     for (int c = 0; c < resultsInternal.size(); c++){
543                                             Result currentResult = resultsInternal.get(c);
544                                             if (currentResult.getRiderID() ==
545     riderToRemove.getRiderID()){
546                                                     resultsInternal.remove(c);
547                                             }
548                                     }
549                             }
550         }
551                     throw new IDNotRecognisedException();
552
553             }
554
555             /**
556             * Checks to see if the name of a rider is already in the system
557             *
558             * @param riderID : ID of the rider checked
559             * @returns A boolean value depending on if the name is in the system
560             */
561             public boolean riderIDInvalid(int riderID){
562                     for (Rider a : ridersInternal){
563                             if(a.getRiderID() == riderID){
564                                     return false;
565                             }
566                     }
567                     return true;
```

```
568             }
569
570             /**
571             * Checks to see if the of a stage is already in the system
572             *
573             * @param stageID : The id of the stage in the system
574             * @return A boolean value depending on if the name is in the system
575             */
576             public boolean stageIDInvalid(int stageID){
577                     for (Race a : racesInternal){
578                             for(int b : a.getStageIDs()){
579                                     if (b == stageID){
580                                             return false;
581                                     }
582                             }
583                     }
584                     return true;
585             }
586
587             /**
588             * Record the times of a rider in a stage.
589             *
590             * @param stageId  : The ID of the stage the result refers to.
591             * @param riderId  : The ID of the rider.
592             * @param checkpoints : An array of times at which the rider reached each of the segments
593     of the stage, including the start time and the finish line.
594             * @throws IDNotRecognisedException    If the ID does not match to any rider or stage in the
595     system.
596             * @throws DuplicatedResultException   Thrown if the rider has already a result for the stage.
597     Each rider can have only one result per stage.
598             * @throws InvalidCheckpointsException Thrown if the length of checkpoints is not equal to
599     n+2, where n is the number of segments in the stage; +2 represents the start time and the finish
600     time of the stage.
```

```
601         * @throws InvalidStageStateException  Thrown if the stage is not "waiting for results".
602 Results can only be added to a stage while it is "waiting for results".
603         */
604         @Override
605         public void registerRiderResultsInStage(int stageId, int riderId, LocalTime... checkpoints)
606                         throws IDNotRecognisedException, DuplicatedResultException,
607 InvalidCheckpointsException,
608                         InvalidStageStateException {
609                 if (riderIDInvalid(riderId) || stageIDInvalid(stageId)){
610                         throw new IDNotRecognisedException();
611                 }
612                 LocalTime temp = checkpoints[0];
613                 for(LocalTime a : checkpoints){
614                         if (a.isBefore(temp)){
615                                 throw new InvalidCheckpointsException();
616                         }
617                         temp = a;
618                 }
619                 Result newResult = new Result(stageId, riderId, checkpoints);
620                 resultsInternal.add(newResult);
621         }
622
623         /**
624          * Get the times of a rider in a stage.
625          *
626          * @param stageId : The ID of the stage the result refers to.
627          * @param riderId : The ID of the rider.
628          * @throws IDNotRecognisedException If the ID does not match to any rider or stage in the
629 system.
630          * @return The array of times at which the rider reached each of the segments of the stage
631 and the total elapsed time. The elapsed time is the difference between the finish time and the start
632 time. Return an empty array if there is no result registered for the rider in the stage.
633          */
```

```
634         @Override
635         public LocalTime[] getRiderResultsInStage(int stageId, int riderId) throws
636 IDNotRecognisedException {
637                 boolean foundRider = false;
638                 for (Rider a : ridersInternal){
639                         if (a.getRiderID() == riderId){
640                                 foundRider = true;
641                         }
642                 }
643
644                 boolean foundStage = false;
645                 for (Race b : racesInternal){
646                         for (int bb : b.getStageIDs()){
647                                 if (bb == stageId) {
648                                         foundStage = true;
649                                 }
650                         }
651                 }
652
653                 if (foundRider == false || foundStage == false){
654                         throw new IDNotRecognisedException();
655                 }
656                 for (Result c : resultsInternal){
657                         if (c.getRiderID() == riderId && c.getStageID() == stageId){
658                                 return c.getTimes();
659                         }
660                 }
661                 return new LocalTime[0];
662         }
663
664         /**
```

```
665             * Sorting the times of the riders
666             */
667             public void sortResultsByTime(){
668                     int n = resultsInternal.size();
669         for (int j = 1; j < n; j++) {
670            Result key = resultsInternal.get(j);
671            int i = j-1;
672            while ( (i > -1) && ( resultsInternal.get(i).getTimeTotal() > key.getTimeTotal() ) ) {
673               resultsInternal.set(i+1,resultsInternal.get(i));
674               i--;
675            }
676            resultsInternal.set(i+1, key);
677         }
678              }
679
680             /**
681              * Creates adjusted elapsed time for each rider
682              *
683              * @param stageId : The ID of the stage the result refers to.
684              * @param riderId : The ID of the rider.
685              * @throws IDNotRecognisedException   If the ID does not match to any rider or stage in the
686 system.
687              * @return The adjusted elapsed time for the rider in the stage. Return an empty array if
688 there is no result registered for the rider in the stage.
689              */
690             @Override
691             public LocalTime getRiderAdjustedElapsedTimeInStage(int stageId, int riderId) throws
692 IDNotRecognisedException {
693                     int desiredRiderIndex = -1;
694                     ArrayList<Integer> riderTimes = new ArrayList<Integer>();
695                     int index = 0;
696                     sortResultsByTime();
```

```
697
698                 //step through to find the result needed
699                 for (Result a : resultsInternal){
700                         if(a.getStageID() == stageId){
701                                 if(a.getRiderID() == riderId){
702                                         desiredRiderIndex = index;
703                                 }
704
705                                 int currentRiderTime = a.getTimeTotal();
706                                 riderTimes.add(currentRiderTime);
707                         }
708                 }
709
710                 if (desiredRiderIndex < 0){
711                         throw new IDNotRecognisedException();
712                 }
713
714                 //step backwards through the results to adjust the times
715
716                 Integer prevCheckedTime = riderTimes.get(riderTimes.size()-1);
717                 Integer smallestInSectionIndex = riderTimes.size() - 1;
718
719                 for (int i = riderTimes.size()-1; i >= 0; i--){
720                         int currentTime = riderTimes.get(i);
721                         int difference = currentTime - prevCheckedTime;
722
723                         if (difference > 1 || (difference <= 1 && i == 0)){
724                                 for(int a = i+1; a <= smallestInSectionIndex; a++){
725                                         riderTimes.set(a,prevCheckedTime);
726                                 }
727                                 smallestInSectionIndex = i;
```

```
728                             }
729                     }
730
731                     int desiredRiderTime = riderTimes.get(desiredRiderIndex);
732                     int hours = desiredRiderTime / 3600;
733                     int minutes = (desiredRiderTime % 3600) / 60;
734                     int seconds = desiredRiderTime % 60;
735                     LocalTime desiredRiderAdjustedTime = LocalTime.of(hours, minutes, seconds);
736                     return desiredRiderAdjustedTime;
737             }
738
739             /**
740              * Removes the stage results from the rider.
741              *
742              * @param stageId : The ID of the stage the result refers to.
743              * @param riderId : The ID of the rider.
744              * @throws IDNotRecognisedException If the ID does not match to any rider or
745              *                          stage in the system.
746              */
747             @Override
748             public void deleteRiderResultsInStage(int stageId, int riderId) throws
749     IDNotRecognisedException {
750                     for (int a = 0; a < resultsInternal.size(); a ++){
751                             if (resultsInternal.get(a).getRiderID() == riderId &&
752     resultsInternal.get(a).getStageID() == stageId){
753                                     resultsInternal.remove(a);
754                             }
755                     }
756                     throw new IDNotRecognisedException();
757             }
758
759             /**
```

```
760             * Get the riders finished position in a a stage.
761             *
762             * @param stageId : The ID of the stage being queried.
763             * @throws IDNotRecognisedException If the ID does not match any stage in the system.
764             * @return A list of riders ID sorted by their elapsed time. An empty list if there is no result
765     for the stage.
766             */
767             @Override
768             public int[] getRidersRankInStage(int stageId) throws IDNotRecognisedException {
769                     ArrayList<Integer> riderIDsInOrder = new ArrayList<Integer>();
770                     sortResultsByTime();
771                     for(Result res : resultsInternal){
772                             if (res.getStageID() == stageId){
773                                     riderIDsInOrder.add(res.getRiderID());
774                             }
775                     }
776                     if (riderIDsInOrder.size() == 0){
777                             throw new IDNotRecognisedException();
778                     }
779                     int[] ridersIntArray = new int[riderIDsInOrder.size()];
780                     for (int i = 0; i < ridersIntArray.length; i++){
781                             ridersIntArray[i] = riderIDsInOrder.get(i);
782                     }
783                     return ridersIntArray;
784             }
785
786             /**
787              * Get the adjusted elapsed times of riders in a stage.
788              *
789              * @param stageId : The ID of the stage being queried.
790              * @throws IDNotRecognisedException If the ID does not match any stage in the system.
```

```
791         * @return The ranked list of adjusted elapsed times sorted by their finish time. An empty
792 list if there is no result for the stage. These times should match the riders returned by {@link
793 #getRidersRankInStage(int)}.
794         */
795         @Override
796         public LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId) throws
797 IDNotRecognisedException {
798                 int desiredRiderIndex = -1;
799                 ArrayList<Integer> riderTimes = new ArrayList<Integer>();
800                 sortResultsByTime();
801
802                 //step through to find the result needed
803                 for (Result a : resultsInternal){
804                         if(a.getStageID() == stageId){
805                                 int currentRiderTime = a.getTimeTotal();
806                                 riderTimes.add(currentRiderTime);
807                         }
808                 }
809
810                 if (desiredRiderIndex < 0){
811                         throw new IDNotRecognisedException();
812                 }
813
814                 //step backwards through the results to adjust the times
815
816                 Integer prevCheckedTime = riderTimes.get(riderTimes.size()-1);
817                 Integer smallestInSectionIndex = riderTimes.size() - 1;
818                 LocalTime[] riderAdjustedTimesArray = new LocalTime[riderTimes.size()];
819
820                 for (int i = riderTimes.size()-1; i >= 0; i--){
821                         int currentTime = riderTimes.get(i);
822                         int difference = currentTime - prevCheckedTime;
```

```
823
824                             if (difference > 1 || (difference <= 1 && i == 0)){
825                                     for(int a = i+1; a <= smallestInSectionIndex; a++){
826                                             riderTimes.set(a,prevCheckedTime);
827                                     }
828                                     smallestInSectionIndex = i;
829                             }
830
831                     }
832                     for (int i = 0; i < riderTimes.size()-1; i++){
833                             int timeSeconds = riderTimes.get(i);
834                             int hours = timeSeconds / 3600;
835                             int minutes = (timeSeconds % 3600) / 60;
836                             int seconds = timeSeconds % 60;
837                             riderAdjustedTimesArray[i] = LocalTime.of(hours, minutes, seconds);
838                     }
839                     return riderAdjustedTimesArray;
840             }
841
842             /**
843              * Get the number of points obtained by each rider in a stage.
844              *
845              * @param stageId : The ID of the stage being queried.
846              * @throws IDNotRecognisedException If the ID does not match any stage in the system.
847              * @return The ranked list of points each riders received in the stage, sorted by their elapsed
848     time. An empty list if there is no result for the stage. These points should match the riders returned
849     by {@link #getRidersRankInStage(int)}.
850              */
851             @Override
852             public int[] getRidersPointsInStage(int stageId) throws IDNotRecognisedException {
853                     final int[] POINTS_FLAT_STAGE = {50,30,20,18,16,14,12,10,8,7,6,5,4,3,2};
854                     final int QUALIFYING = 15;
```

```
855             //get rider IDs in order
856             ArrayList<Integer> riderIDsInOrder = new ArrayList<Integer>();
857             sortResultsByTime();
858             for(Result res : resultsInternal){
859                     if (res.getStageID() == stageId){
860                             riderIDsInOrder.add(res.getRiderID());
861                     }
862             }
863             if (riderIDsInOrder.size() == 0){
864                     throw new IDNotRecognisedException();
865             }
866
867             //find stage type - points are allocated differently
868             StageType desiredStageType = null;
869             for (Race race : racesInternal){
870                     for (Stage stage : race.getStages()){
871                             if (stage.getStageId() == stageId){
872                                     desiredStageType = stage.getStageType();
873                             }
874                     }
875             }
876             //Assume that the desired stage is already a flat stage
877
878             //replace IDs with points for each entry
879             for (int i = 0; i < riderIDsInOrder.size(); i++){
880                     if (i <= QUALIFYING){
881                             riderIDsInOrder.set(i,0);
882                     }
883                     switch (desiredStageType){
884                             case HIGH_MOUNTAIN:
885                                     riderIDsInOrder.set(i,POINTS_FLAT_STAGE [i]);
```

```
886                                         break;
887                                 default:
888                                         break;
889
890                         }
891                 }
892
893                 //swap from arraylist to int[]
894                 int[] ridersIntArray = new int[riderIDsInOrder.size()];
895                 for (int i = 0; i < ridersIntArray.length; i++){
896                         ridersIntArray[i] = riderIDsInOrder.get(i);
897                 }
898                 return ridersIntArray;
899         }
900
901         /**
902          * Get the number of mountain points obtained by each rider in a stage.
903          *
904          * @param stageId : The ID of the stage being queried.
905          * @throws IDNotRecognisedException If the ID does not match any stage in the system.
906          * @return The ranked list of mountain points each riders received in the stage, sorted by
907 their finish time. An empty list if there is no result for the stage. These points should match the
908 riders returned by {@link #getRidersRankInStage(int)}.
909          */
910         @Override
911         public int[] getRidersMountainPointsInStage(int stageId) throws IDNotRecognisedException {
912                 final int[] POINTS_MID_MNTN_STAGE = {30,25,22,19,17,15,13,11,9,7,6,5,4,3,2};
913                 final int[] POINTS_HIGH_MNTN_STAGE = {20,17,15,13,11,10,9,8,7,6,5,4,3,2,1};
914                 final int QUALIFYING = 15;
915                 //get rider IDs in order
916                 ArrayList<Integer> riderIDsInOrder = new ArrayList<Integer>();
917                 sortResultsByTime();
```

```
918             for(Result res : resultsInternal){
919                 if (res.getStageID() == stageId){
920                     riderIDsInOrder.add(res.getRiderID());
921                 }
922             }
923             if (riderIDsInOrder.size() == 0){
924                 throw new IDNotRecognisedException();
925             }
926
927             //find stage type - points are allocated differently
928             StageType desiredStageType = null;
929             for (Race race : racesInternal){
930                 for (Stage stage : race.getStages()){
931                     if (stage.getStageId() == stageId){
932                         desiredStageType = stage.getStageType();
933                     }
934                 }
935             }
936             //Assume that the desired stage is already a flat stage
937
938             //replace IDs with points for each entry
939             for (int i = 0; i < riderIDsInOrder.size(); i++){
940                 if (i <= QUALIFYING){
941                     riderIDsInOrder.set(i,0);
942                 }
943                 switch (desiredStageType){
944                     case HIGH_MOUNTAIN:
945                         riderIDsInOrder.set(i,POINTS_HIGH_MNTN_STAGE[i]);
946                         break;
947                     case MEDIUM_MOUNTAIN:
948                         riderIDsInOrder.set(i,POINTS_MID_MNTN_STAGE[i]);
```

```
949                                     break;
950                             default:
951                                     break;
952                     }
953             }
954
955             //swap from arraylist to int[]
956             int[] ridersIntArray = new int[riderIDsInOrder.size()];
957             for (int i = 0; i < ridersIntArray.length; i++){
958                     ridersIntArray[i] = riderIDsInOrder.get(i);
959             }
960             return ridersIntArray;
961     }
962
963     /**
964      * Method empties this MiniCyclingPortalInterface of its contents and resets all
965      * internal counters.
966      */
967     @Override
968     public void eraseCyclingPortal() {
969             ridersInternal = new ArrayList<Rider>();
970             teamsInternal = new ArrayList<Team>();
971             racesInternal = new ArrayList<Race>();
972             resultsInternal = new ArrayList<Result>();
973             currentStageID = 0;
974             currentSegmentID = 0;
975             currentRaceID = 0;
976             currentTeamID = 0;
977             currentRiderID = 0;
978     }
979
```

```
980         /**
981          * Method saves this MiniCyclingPortalInterface contents into a serialised file, with the
982 filename given in the argument.
983          *
984          * @param filename Location of the file to be saved.
985          * @throws IOException If there is a problem experienced when trying to save the store
986 contents to the file.
987          */
988         @Override
989         public void saveCyclingPortal(String filename) throws IOException {
990                 try (ObjectOutputStream out = new ObjectOutputStream(new
991 FileOutputStream(filename+".ser"))) {
992                         out.writeObject(ridersInternal);
993                         out.writeObject(teamsInternal);
994                         out.writeObject(racesInternal);
995                         out.writeObject(resultsInternal);
996                         out.writeObject(currentStageID);
997                         out.writeObject(currentSegmentID);
998                         out.writeObject(currentRaceID);
999                         out.writeObject(currentTeamID);
1000                        out.writeObject(currentRiderID);
1001                }
1002        }
1003
1004        /**
1005         * Method should load and replace this MiniCyclingPortalInterface contents with the
1006         * serialised contents stored in the file given in the argument.
1007         *
1008         * @param filename : Location of the file to be loaded.
1009         * @throws IOException If there is a problem experienced when trying to load the store
1010 contents from the file.
1011         * @throws ClassNotFoundException If required class files cannot be found when loading.
```

```
1012            */
1013            @Override
1014            @SuppressWarnings("unchecked")
1015            public void loadCyclingPortal(String filename) throws IOException, ClassNotFoundException
1016    {
1017                    try (ObjectInputStream in = new ObjectInputStream(new
1018    FileInputStream(filename+".ser"))) {
1019                            ridersInternal = (ArrayList<Rider>) in.readObject();
1020                            teamsInternal =  (ArrayList<Team>) in.readObject();
1021                            racesInternal =  (ArrayList<Race>) in.readObject();
1022                            resultsInternal = (ArrayList<Result>)  in.readObject();
1023                            currentStageID =  (Integer) in.readObject();
1024                            currentSegmentID = (Integer)  in.readObject();
1025                            currentRaceID =  (Integer) in.readObject();
1026                            currentTeamID = (Integer) in.readObject();
1027                            currentRiderID= (Integer) in.readObject();
1028                    }
1029            }
1030
1031    //The end of MiniCyclingPortalInterface
1032    //The beginning of CyclingPortalInterface
1033
1034            /**
1035             * The method removes the race and all its related information, i.e., stages,
1036             * segments, and results.
1037             *
1038             * @param name The name of the race to be removed.
1039             * @throws NameNotRecognisedException If the name does not match to any race in the
1040    system.
1041             */
1042            @Override
1043            public void removeRaceByName(String name) throws NameNotRecognisedException {
```

```
1044                    for (int a = 0; a < racesInternal.size(); a++){
1045                            if (racesInternal.get(a).getName().equals(name)){
1046                                    racesInternal.remove(a);
1047                            }
1048                    }
1049                    throw new NameNotRecognisedException();
1050
1051            }
1052
1053            /**
1054             * Get the general classification times of riders in a race.
1055             *
1056             * @param raceId : The ID of the race being queried.
1057             * @throws IDNotRecognisedException If the ID does not match any race in the system.
1058             * @return A list of riders' times sorted by the sum of their adjusted elapsed times in all
1059     stages of the race. An empty list if there is no result for any stage in the race. These times should
1060     match the riders returned by {@link #getRidersGeneralClassificationRank(int)}.
1061             */
1062            @Override
1063            public LocalTime[] getGeneralClassificationTimesInRace(int raceId) throws
1064     IDNotRecognisedException {
1065                    // TODO Auto-generated method stub
1066                    return null;
1067            }
1068
1069            /**
1070             * Get the overall points of riders in a race.
1071             *
1072             * @param raceId : The ID of the race being queried.
1073             * @throws IDNotRecognisedException If the ID does not match any race in the system.
1074             * @return A list of riders' points (i.e., the sum of their points in all stages of the race), sorted
1075     by the total elapsed time. An empty list if there is no result for any stage in the race. These points
1076     should match the riders returned by {@link #getRidersGeneralClassificationRank(int)}.
```

```
1077            */
1078            @Override
1079            public int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException {
1080
1081                    ////-----rethink after getRidersGeneralClassificationRank-----////
1082                    /*
1083
1084                    int[] requiredStagesFromRace = {};
1085                    Integer amountOfStages = null;
1086                    for (Race a : racesInternal){
1087                            if(a.getRaceID() == raceId){
1088                                    requiredStagesFromRace = a.getStageIDs();
1089                                    amountOfStages = requiredStagesFromRace.length;
1090                            }
1091                    }
1092                    sortResultsByTime();
1093                    //get race results per stage
1094                    for (Result a : resultsInternal){
1095                            //search through results
1096                            //find results in order
1097                            //find riderIDs in order of times
1098                            //find times in order
1099
1100                    }
1101                    //sum times
1102                    //sum points
1103                    //sort both in parallel
1104                    //
1105
1106
1107                    //get rider order per stage
```

```
1108                    //find total eadjusted time for each stage
1109                    //total points
1110                    //sort both lists so that points are in order of time
1111
1112                    */
1113                    return null;
1114            }
1115
1116            /**
1117             * Get the overall mountain points of riders in a race.
1118             *
1119             * @param raceId The ID of the race being queried.
1120             * @throws IDNotRecognisedException If the ID does not match any race in the system.
1121             * @return A list of riders' mountain points, sorted by the total elapsed time. An empty list if
1122     there is no result for any stage in the race. These points should match the riders returned by {@link
1123     #getRidersGeneralClassificationRank(int)}.
1124             */
1125            @Override
1126            public int[] getRidersMountainPointsInRace(int raceId) throws IDNotRecognisedException {
1127                    // TODO Auto-generated method stub
1128                    return null;
1129            }
1130
1131            /**
1132            * Returns a list of the results in a stage
1133            *
1134            * @param stageID : The id of the stage being used
1135            * @return A list of the results of the stage
1136            */
1137            private ArrayList<Result> getResultListInStage(int stageID){
1138                    ArrayList<Result> resultsNeeded = new ArrayList<Result>();
1139                    for (Result res : resultsInternal){
```

```
1140                            if (res.getStageID() == stageID){
1141                                    resultsNeeded.add(res);
1142                            }
1143                    }
1144                    return resultsNeeded;
1145            }
1146
1147            /**
1148             * Get the general classification rank of riders in a race.
1149             *
1150             * @param raceId : The ID of the race being queried.
1151             * @throws IDNotRecognisedException If the ID does not match any race in the system.
1152             * @return A ranked list of riders' IDs sorted ascending by the sum of their adjusted elapsed
1153     times in all stages of the race. That is, the first in this list is the winner (least time). An empty list if
1154     there is no result for any stage in the race.
1155             */
1156            @Override
1157            public int[] getRidersGeneralClassificationRank(int raceId) throws IDNotRecognisedException
1158     {
1159                    //find the race in racesInternal
1160                    Race raceInQuestion = null;
1161                    for (Race race : racesInternal){
1162                            if (race.getRaceID() == raceId){
1163                                    raceInQuestion = race;
1164                            }
1165                    }
1166
1167                    if (raceInQuestion == null){
1168                            throw new IDNotRecognisedException();
1169                    }
1170
1171                    //iterate through stages in the race
```

```
1172                    sortResultsByTime();
1173                    ArrayList<Result> resultsInStage = new ArrayList<Result>();
1174                    for (Stage stage : raceInQuestion.getStages()){
1175                            //find a list of results for that stage
1176                            resultsInStage = getResultListInStage(stage.getStageId());
1177
1178
1179
1180
1181                    }
1182                    return null;
1183            }
1184
1185            /**
1186             * Get the ranked list of riders based on the points classification in a race.
1187             *
1188             * @param raceId : The ID of the race being queried.
1189             * @throws IDNotRecognisedException If the ID does not match any race in the system.
1190             * @return A ranked list of riders' IDs sorted descending by the sum of their points in all
1191     stages of the race. That is, the first in this list is the winner (more points). An empty list if there is no
1192     result for any stage in the race.
1193             */
1194            @Override
1195            public int[] getRidersPointClassificationRank(int raceId) throws IDNotRecognisedException {
1196                    // TODO Auto-generated method stub
1197                    return null;
1198            }
1199
1200            /**
1201             * Get the ranked list of riders based on the mountain classification in a race.
1202             *
1203             * @param raceId The ID of the race being queried.
```

```
1204            * @throws IDNotRecognisedException If the ID does not match any race in the system.
1205            * @return A ranked list of riders' IDs sorted descending by the sum of their mountain points
1206    in all stages of the race. That is, the first in this list is the winner (more points). An empty list if there
1207    is no result for any stage in the race.
1208            */
1209            @Override
1210            public int[] getRidersMountainPointClassificationRank(int raceId) throws
1211    IDNotRecognisedException {
1212                    // TODO Auto-generated method stub
1213                    return null;
1214            }
1215
1216            /**
1217            * Converts the array of teams to a string
1218            *
1219            * @return A string of the teams in the system
1220            */
1221            public String[] debugTeamsToString(){
1222                    int size = teamsInternal.size();
1223                    String[] teamsAsStrings = new String[size];
1224                    for (int i = 0; i < size; i++){
1225                            teamsAsStrings[i] = teamsInternal.get(i).toString();
1226                    }
1227                    return teamsAsStrings;
1228            }
1229
1230            /**
1231            * Converts the array of rider to a string
1232            *
1233            * @return A string of the rider in the system
1234            */
1235            public String[] debugRidersToString(){
```

```
1236                    int size = ridersInternal.size();
1237                    String[] ridersAsStrings = new String[size];
1238                    for (int i = 0; i < size; i++){
1239                            ridersAsStrings[i] = ridersInternal.get(i).toString();
1240                    }
1241                    return ridersAsStrings;
1242            }
1243
1244            /**
1245            * Converts the array of races to a string with the length
1246            *
1247            * @return A string of the races in the system
1248            */
1249            public String[] debugRacesToStringLen(){
1250                    int size = racesInternal.size();
1251                    String[] racesAsStrings = new String[size];
1252                    for (int i = 0; i < size; i++){
1253                            racesAsStrings[i] = racesInternal.get(i).toString();
1254                    }
1255                    return racesAsStrings;
1256            }
1257
1258            /**
1259            * Converts the array of races to a string with the stages
1260            *
1261            * @return A string of the races in the system
1262            */
1263            public String[] debugRacesToStringStage(){
1264                    int size = racesInternal.size();
1265                    String[] racesAsStrings = new String[size];
1266                    for (int i = 0; i < size; i++){
```

```
1267                            racesAsStrings[i] = racesInternal.get(i).toStringStage();
1268                    }
1269                    return racesAsStrings;
1270            }
1271
1272            /**
1273            * Converts the array of results to a string
1274            *
1275            * @return A string of the results in the system
1276            */
1277            public String[] debugResultsToString(){
1278                    int size = resultsInternal.size();
1279                    String[] resultsAsStrings = new String[size];
1280                    for (int i = 0; i < size; i++){
1281                            resultsAsStrings[i] = resultsInternal.get(i).toString();
1282                    }
1283                    return resultsAsStrings;
1284            }
1285    }
1286
```

```
1  package cycling;
2
3  public @interface Override {
4
5  }
```

```
1   package cycling;
2
3   import java.io.Serializable;
4   import java.util.ArrayList;
5   import java.time.LocalDateTime;
6
7   /**
8    * Race is a class that creates a race object.
9    *
10   * @author Jude Goulding & Bethany Whiting
11   *
12   */
13
14  public class Race implements Serializable{
15      private int raceID;
16      private ArrayList<Stage> stages;
17      private String name;
18      private String description;
19
20       /**
21       * Constructs an instant of a race
22       *
23       * @param stageIdIn
24       * @param stageName
25       * @param stageDescription
26       * @param stageLength
27       * @param stageStartTime
28       * @param sType
29       */
30      public Race(int index, String n, String d){
31          name = n;
```

```
32          description = d;
33          raceID = index;
34          stages = new ArrayList<Stage>();
35      }
36
37      /**
38      * Returns the ID of a race
39      *
40      * @return raceID
41      */
42      public int getRaceID(){
43          return raceID;
44      }
45
46      /**
47      * Returns of the name of a race
48      *
49      * @param name
50      */
51      public String getName(){
52          return name;
53      }
54
55      /**
56      * Returns the id of a stage created
57      *
58      * @param stageID
59      * @param stageName
60      * @param stageDescription
61      * @param stageLength
62      * @param stageStartTime
```

```
63      * @param sType
64      * @return a Id of the stage
65      */
66      public int addStage(int stageID, String stageName, String stageDescription, double stageLength,
67  LocalDateTime stageStartTime,
68      StageType sType){
69          Stage stage = new Stage(stageID, stageName, stageDescription, stageLength, stageStartTime,
70  sType);
71          stages.add(stage);
72          return stage.getStageId();
73      }
74
75      /**
76      * Returns a list of the stages in a race
77      *
78      * @return stages
79      */
80      public ArrayList<Stage> getStages(){
81          return stages;
82      }
83
84      /**
85      * Returns an array of the stage Id from a race
86      *
87      * @return stageIDs
88      */
89      public int[] getStageIDs(){
90          int[] stageIDs = new int[stages.size()];
91          for (int a = 0; a < stages.size(); ++a){
92              stageIDs[a] = stages.get(a).getStageId();
93          }
94          return stageIDs;
```

```
95      }
96
97      /**
98      * Returns a string with all the details for a race
99      *
100     * @return The details in the format id, name, description and total length
101     */
102     public String toString(){
103        double stageLength = 0.0;
104        for (Stage a : stages){
105           stageLength += a.getStageLength();
106        }
107        return String.format("id=%d,name=%s,description=%s,total length=%f
108 km",raceID,name,description,stageLength);
109     }
110
111     /**
112     * Returns a string of the deatils about a race and the stages in it
113     *
114     * @return The details in the format id, name, description and stages
115     */
116     public String toStringStage(){
117        String stagesStr = "";
118        for (Stage a : stages){
119           stagesStr += a.toString();
120        }
121        return
122 String.format("id=%d,name=%s,description=%s,stages=[%s]",raceID,name,description,stagesStr);
123     }
124
125     /**
126     * Removes a stage from a race
```

```
127         *
128         * @param stageID
129         */
130         public void removeStage(int stageID){
131             for (int index = 0; index <= stages.size(); index++){
132                 if (stages.get(index).getStageId() == stageID){
133                     stages.remove(index);
134                 }
135             }
136         }
137     }
```

```
1   package cycling;
2
3   import java.time.LocalTime;
4   import java.io.Serializable;
5   import java.time.temporal.ChronoUnit;
6
7   /**
8    * Resuls class holds the rider ID, stage ID and checkpoint times in one place
9    */
10  public class Result implements Serializable{
11      private int stageID;
12      private int riderID;
13      private LocalTime[] checkpointTimes;
14
15      /**
16       * Contructs a result.
17       *
18       * @param sID   id of the stage the times were reecorded in
19       * @param rID   id of the rider who rode
20       * @param cpTimes the list of times through each checkpoint
21       */
22      public Result(int sID, int rID, LocalTime[] cpTimes){
23          stageID = sID;
24          riderID = rID;
25          checkpointTimes = cpTimes;
26      }
27
28      /**
29       * returns the ID of the rider who achieved the result
30       * @return riderID
31       */
```

```
32      public int getRiderID(){
33          return riderID;
34      }
35
36      /**
37       * returns the id of the stage the result was recorded in
38       * @return stageID
39       */
40      public int getStageID(){
41          return stageID;
42      }
43
44
45      /**
46       * returns an array of times recorded in the stage by the rider
47       * @return checkpointTimes
48       */
49      public LocalTime[] getTimes(){
50          return checkpointTimes;
51      }
52
53      /**
54       * returns the total amount of time the stage took to complete, in seconds
55       * @return defInt
56       */
57      public int getTimeTotal(){
58          LocalTime start = checkpointTimes[0];
59          LocalTime finish = checkpointTimes[checkpointTimes.length-1];
60          long difference = start.until(finish, ChronoUnit.SECONDS);
61          Integer defInt = Math.toIntExact(difference);
62          return defInt;
```

```
63        }
64
65        /**
66         * returns the details of the result as a singe string
67         * @return details
68         */
69        public String toString(){
70            return String.format("rider=%d,stage=%d,time=%dseconds",riderID,stageID,this.getTimeTotal());
71        }
72    }
```

```
1  package cycling;
2
3  import java.io.Serializable;
4
5  public class Rider implements Serializable{
6      /**
7       * The rider class
8       *
9       *
10      */
11
12     private String name;
13     private int riderID;
14     private int yearOfBirth;
15     private int teamID;
16
17     /**
18      * Rider contructor
19      *
20      * @param nameIN
21      * @param birthYr
22      * @param team
23      */
24     public Rider(int rID, String nameIN, int birthYr, int team){
25         riderID = rID;
26         name = nameIN;
27         yearOfBirth = birthYr;
28         teamID = team;
29     }
30     /**
31      * Returns the name of the rider
```

```
32          *
33          * @return name
34          */
35         public String getName(){
36             return name;
37         }
38
39         /**
40          * Sets the name of the rider to the string given
41          *
42          * @param nameIN
43          */
44         public void setName(String nameIN){
45             name = nameIN;
46         }
47
48         /**
49          * Returns the year that rider was born in
50          * @return yearOfBirth
51          */
52         public int getYear(){
53             return yearOfBirth;
54         }
55
56         /**
57          * Sets the riders year of birth to be the inputted integer
58          * @param year
59          */
60         public void setYear(int year){
61             yearOfBirth = year;
62         }
```

```
63
64      /**
65       * returns thhe ID of the rider
66       * @return riderID
67       */
68      public int getRiderID(){
69          return riderID;
70      }
71
72      /**
73       * returns the ID of the team that the rider is in
74       * @return teamID
75       */
76      public int getTeamID(){
77          return teamID;
78      }
79
80      /**
81       * Sets the rider's team to the inputted team. To be used with addRider or removeRider to change
82  the rider's team in the system
83       * @param newTeamID
84       */
85      public void changeTeam(int newTeamID){
86          teamID = newTeamID;
87      }
88
89      /**
90       * returns a string of the details of the rider
91       * @return details
92       */
93      public String toString(){
```

```
94              return String.format("id=%d,name=%s,yob=%d,team=%d",riderID,name,yearOfBirth,teamID);
95          }
96      }
```

```
1  package cycling;
2
3  import java.io.Serializable;
4
5  /**
6   * Segment is a class that creates a Segment object.
7   *
8   * @author Jude Goulding & Bethany Whiting
9   *
10  */
11
12 public class Segment implements Serializable{
13
14    private SegmentType segmentType;
15    private int segmentID;
16    private double averageGradient;
17    private double length;
18    private double location;
19
20    /**
21     * Constructs an instant of a segment
22     *
23     * @param sID
24     * @param fin
25     * @param sType
26     * @param aGradient
27     * @param len
28     */
29    public Segment(int sID, double fin, SegmentType sType, double aGradient, double len){
30       segmentType = sType;
31       averageGradient = aGradient;
```

```
32          segmentID = sID;
33          length = len;
34          location = fin;
35      }
36
37      /**
38      * Adds to the instant of a segment
39      *
40      * @param sID
41      * @param fin
42      */
43      public Segment(int sID, double fin){
44          segmentID = sID;
45          location = fin;
46          segmentType = SegmentType.SPRINT;
47          averageGradient = 0.0;
48          length = 0.0;
49      }
50
51      /**
52      * Creates string of detail of a segment
53      *
54      * @return A string in the format id, type, gradient, location, length
55      */
56      public String toString(){
57          return String.format("[id=%d, type=%s, average gradient=%s, start location=%d, length=%d]",
58  segmentID, segmentType, averageGradient, location, length);
59      }
60
61      /**
62      * Returns the ID of a segment
```

```
63          *
64          * @return segmentID
65          */
66          public int getSegmentID(){
67              return segmentID;
68          }
69      }
```

```
1   package cycling;
2
3   import java.io.Serializable;
4   import java.time.LocalDateTime;
5   import java.util.ArrayList;
6
7   /**
8    * Stage is a class that creates a stage object as part of a race.
9    *
10   * @author Jude Goulding & Bethany Whiting
11   *
12   */
13
14  public class Stage implements Serializable{
15
16      private StageType stageType;
17      private int stageID;
18      private String name;
19      private String description;
20      private double length;
21      private LocalDateTime startTime;
22      private ArrayList<Segment> segments;
23      private boolean waitingForResults = false;
24
25      /**
26       * Constructs an instant of a stage
27       *
28       * @param stageIdIn
29       * @param stageName
30       * @param stageDescription
31       * @param stageLength
```

```
32          * @param stageStartTime
33          * @param sType
34          */
35         public Stage(int stageIdIn, String stageName, String stageDescription, double stageLength,
36     LocalDateTime stageStartTime,
37         StageType sType){
38             stageID = stageIdIn;
39             stageType = sType;
40             name = stageName;
41             description = stageDescription;
42             length = stageLength;
43             startTime = stageStartTime;
44             segments = new ArrayList<Segment>();
45         }
46
47         /**
48         * Creates a string of all the information about a stage
49         *
50         * @return A string with the id, name, desciption, start time, type, length and segments
51         */
52         public String toString(){
53             String segmentList = "";
54             for (Segment a : segments){
55                 segmentList += a.toString() + ",";
56             }
57             return
58     String.format("[id=%d,name=%s,description=%s,starttime=%s,type=%s,length=%f,segments=[%s]]",
59     stageID, name, description, startTime, stageType, length, segmentList);
60         }
61
62         /**
63         * Returns the ids of the stage
```

```
64      *
65      * @return stageID
66      */
67      public int getStageId(){
68          return stageID;
69      }
70
71      /**
72      * Returns the name of the stage
73      *
74      * @return name
75      */
76      public String getName(){
77          return name;
78      }
79
80      /**
81      * Returns the decription of the stage
82      *
83      * @return description
84      */
85      public String getDescription(){
86          return description;
87      }
88
89      /**
90      * Returns the length of the stage
91      *
92      * @return length
93      */
94      public double getStageLength(){
```

```
95          return length;
96      }
97
98      /**
99      * Gets a new segment object and returns the ID for the climb segment
100     *
101     * @return The ID of the new segment
102     */
103     public int addCatClimbSegment(int segmentId, Double location, SegmentType type, Double
104 averageGradient, Double length){
105         Segment temp = new Segment(segmentId, location, type, averageGradient, length);
106         segments.add(temp);
107         return temp.getSegmentID();
108     }
109
110     /**
111     * Gets a new segment object and returns the ID fof the sprint segment
112     *
113     * @return The ID of the new segment
114     */
115     public int addSprintSegment(int segmentId, Double location){
116         Segment temp = new Segment(segmentId, location);
117         segments.add(temp);
118         return temp.getSegmentID();
119     }
120
121     /**
122     * Returns the type of stage it is
123     *
124     * @return stageType
125     */
```

```
126     public StageType getStageType(){
127         return stageType;
128     }
129 
130     /**
131     * Checks if the stage is waiting for results
132     *
133     * @return waitingForResults
134     */
135     public boolean isWaiting(){
136         return waitingForResults;
137     }
138 
139     /**
140     * Returns a list of segments in a stage
141     *
142     * @return segments
143     */
144     public ArrayList<Segment> getSegments(){
145         return segments;
146     }
147 
148     /**
149     * Removes a segment within the index given
150     *
151     * @param index
152     */
153     public void removeSeg(int index){
154         segments.remove(index);
155     }
156 
```

```
157         /**
158         * Sets waitingForResults to true
159         */
160         public void setWaiting(){
161             waitingForResults = true;
162         }
163 
164         /**
165         * Returns an array of id of the segments within a stage
166         */
167         public int[] getSegmentIDs(){
168             int[] stageIDs = new int[segments.size()];
169             for (int a = 0; a < segments.size(); ++a){
170                 stageIDs[a] = segments.get(a).getSegmentID();
171             }
172             return stageIDs;
173         }
174     }
```

```
1  package cycling;
2
3  import java.io.Serializable;
4  import java.util.ArrayList;
5  /**
6   * The team object. Represents a cycling team if cyclists.
7   *
8   * @author Jude Goulding & Bethany Whiting
9   *
10  */
11 public class Team implements Serializable{
12    private String name;
13    private String description;
14    private int teamID;
15    private ArrayList<Integer> riders;
16
17    /**
18     * Constructs an instant of a team
19     *
20     * @param tID
21     * @param n
22     * @param desc
23     */
24    public Team(int tID, String n, String desc){
25       name = n;
26       description = desc;
27       teamID = tID;
28       riders = new ArrayList<Integer>();
29    }
30
31    /**
```

```
32          * Returns the name of the team
33          * @return name
34          */
35         public String getName(){
36             return name;
37         }
38
39         /**
40          * Returns the description of the team
41          * @return description
42          */
43         public String getDesc(){
44             return description;
45         }
46
47         /**
48          * Returns the ID of the team
49          * @return team ID
50          */
51         public int getTeamID(){
52             return teamID;
53         }
54
55         /**
56          * return a list of the IDs of the riders in the team
57          * @return list of rider IDs
58          */
59         public int[] getRiders(){
60             int[] riderIDs = new int[riders.size()];
61             for (int a = 0; a < riders.size(); a++){
62                 riderIDs[a] = riders.get(a);
```

```
63          }
64          return riderIDs;
65      }
66
67      /**
68       * Adds a rider's ID to the list in the team
69       * @param riderID
70       * @return the ID of the rider that was added
71       */
72      public int addRider(int riderID){
73          riders.add(riderID);
74          return riderID;
75      }
76
77      /**
78       * removes a rider from the list to be used in tangent with adjacent methods in the Rider class that
79  modify the team it belongs to
80       * @param riderID
81       */
82      public void removeRider(int riderID){
83          for (int a = 0; a < riders.size(); a++){
84              if (riders.get(a) == riderID){
85                  riders.remove(a);
86              }
87          }
88      }
89
90      /**
91       * returns a string containing th details of the team
92       * @return string representation of the team
93       */
```

```
94      public String toString(){
95          String ridersStr = "";
96          for (Integer a : riders){
97              ridersStr += Integer.toString(a) + ",";
98          }
99          return
100 String.format("id=%d,name=%s,description=%s,riders=[%s]",teamID,name,description,ridersStr);
101     }
102 }
```