

```
1 package cycling;
2
3 import java.io.FileInputStream;
4 import java.io.FileOutputStream;
5 import java.io.IOException;
6 import java.io.ObjectInputStream;
7 import java.io.ObjectOutputStream;
8 import java.lang.ProcessBuilder.Redirect.Type;
9 import java.time.Duration;
10 import java.time.LocalDateTime;
11 import java.time.LocalTime;
12 import java.time.temporal.ChronoUnit;
13 import java.util.*;
14 import java.util.Map.Entry;
15 import java.util.stream.Collectors;
16
17 import java.util.HashMap;
18 import java.util.ArrayList;
19
20
21 /**
22  * CyclingPortal is a minimally compiling, but non-functioning implementor
23  * of the CyclingPortalInterface interface.
24  *
25  * @author Kaloyan Gaydarov
26  * @author Taariq Fadhill
27  * @version 4.20
28  * @since 14/02/2022
29  *
30  */
31 public class CyclingPortal implements CyclingPortalInterface {
32     //----- Initialization of ArrayLists used to store the objects ----
33     -----//
34     private ArrayList<Race> raceArray = new ArrayList<>();
35     private ArrayList<Team> teamArray = new ArrayList<>();
36     private ArrayList<Rider> riderArray = new ArrayList<>();
37
38     //-----RACE METHODS-----
39     --//
40
41     /**
42      * This method is used to initialise a new session of the cycling portal
43      *
44      */
45     public CyclingPortal() {
46         teamArray = new ArrayList<>();
47         raceArray = new ArrayList<>();
48         riderArray = new ArrayList<>();
49     }
50
51     /**
52      * This method is used to find a race using its ID
53      *
54      * @param id The ID of the race that is being searched up.
55      * @return The race object of the given ID.
```

```

54  * @return Else it returns an empty race object.
55  */
56  private Race getRaceById(int id) {
57      for (Race race : raceArray) {
58          if (race.getRaceID() == id) {
59              return race;
60          }
61      }
62      return new Race();
63  }
64
65  /**
66   * This method is used to find a race using the ID of a given stage.
67   *
68   * @param id The ID of the stage that is being searched up.
69   * @return The race object of the given stage ID.
70   * @return Else it returns an empty race object.
71   */
72  private Race getRaceByStageId(int id) {
73      for (Race race : raceArray) {
74          for (Stage stage : race.getStages()) {
75              if (stage.getStageId() == id) {
76                  return race;
77              }
78          }
79      }
80      return new Race();
81  }
82  /**
83   * This method is used to get a stage from the ID of a given segment.
84   *
85   * @param id The ID of the segment that is being searched up.
86   * @return The stage object of the given segment ID.
87   * @return Empty stage object if Null.
88   */
89  private Stage getStageBySegmentId(int id) {
90      for (Race race: raceArray){
91          for (Stage stage : race.getStages()) {
92              for (Segment segment : stage.getStageSegments()) {
93                  if (segment.getSegmentID() == id) {
94                      return stage;
95                  }
96              }
97          }
98      }
99      return new Stage();
100 }
101
102 /**
103  * This method is used to find a stage using its ID
104  *
105  * @param id The ID of the stage that is being searched up.
106  * @return The stage object of the given ID.
107  * @return Else it returns an empty stage object.
108  */

```

```
109 private Stage getStageByID(int id) {
110     for (Race race : raceArray){
111         for (Stage stage : race.getStages()) {
112             if (stage.getStageId() == id) {
113                 return stage;
114             }
115         }
116     }
117     return new Stage();
118 }
119
120 /**
121  * This method is used to find a segment using its ID
122  *
123  * @param id The ID of the segment that is being searched up.
124  * @return The segment object of the given ID.
125  * @return Else it returns an empty segment object.
126  */
127 private Segment getSegmentByID(int id) {
128     for (Race race : raceArray) {
129         for (Stage stage : race.getStages()) {
130             for (Segment segment : stage.getStageSegments()) {
131                 if (segment.getSegmentID() == id) {
132                     return segment;
133                 }
134             }
135         }
136     }
137     return new Segment();
138 }
139
140 /**
141  * This method is used to find a team using its ID
142  *
143  * @param id The ID of the team that is being searched up.
144  * @return The team object of the given ID.
145  * @return Else it returns an empty team object.
146  */
147 private Team getTeamByID(int id) {
148     for (Team team : teamArray) {
149         if (team.getTeamID() == id) {
150             return team;
151         }
152     }
153     return new Team();
154 }
155
156 /**
157  * This method is used to find a rider using its ID
158  *
159  * @param id The ID of the rider that is being searched up.
160  * @return The rider object of the given ID.
161  * @return Else it returns an empty rider object.
162  */
163 private Rider getRiderByID(int id) {
```

```

163     private Rider getRiderById(int id) {
164         for (Rider rider: riderArray) {
165             if (rider.getRiderID() == id) {
166                 return rider;
167             }
168         }
169         return new Rider();
170     }
171
172     //-----PORTAL METHODS-----
173     ----//
174     /**
175      * This method is used to get all race IDs on the platform
176      *
177      * @return Array of int containing the race IDs
178      */
179     @Override
180     public int[] getRaceIds() {
181         //Temp array of integers to hold all IDs
182         int[] raceIDs = new int[raceArray.size()];
183         //Add each race ID to the array of integers and then return the array
184         for (int i=0; i<raceArray.size(); i++) {
185             raceIDs[i] = raceArray.get(i).getRaceID();
186         }
187         return raceIDs;
188     }
189
190     /**
191      * This method is used to create a staged race with a given name and
192      * description.
193      *
194      * @param name The name of the race.
195      * @param description The description of the race (Can be null).
196      * @return The unique ID of the created race.
197      * @exception IllegalArgumentException Thrown when attempting to assign a
198      * race name already in use in the system.
199      * @exception InvalidNameException If the name is null, empty, has more
200      * than 30 characters, or has white spaces.
201      */
202     @Override
203     public int createRace(String name, String description) throws
204     IllegalArgumentException, InvalidNameException {
205         // Check name matches the requirements needed
206         if((name == null) || (name.length() > 30) || (name.contains(" ") ||
207         (name == ""))) {
208             throw new InvalidNameException("Race name doesn't match the
209             requirements");
210         }
211         // Checks race Name doesn't exist already
212         for(int i = 0; i < raceArray.size(); i++) {
213             if (name.equals(raceArray.get(i).getRaceName())) {
214                 throw new IllegalArgumentException("Race name " + name + " already
215                 exists");
216             }
217         }
218     }

```

```

210     }
211     //Create a new race object
212     Race race = new Race(name , description);
213     raceArray.add(race);
214     return race.getRaceID();
215 }
216
217 /**
218  * This method is used to view the race details and get them returned.
219  *
220  * @param raceId The unique id of the race to see its details.
221  * @return String of all details of the race concatenated together.
222  * @exception IDNotRecognisedException The ID of the race is not existint
in the platform.
223  */
224 @Override
225 public String viewRaceDetails(int raceId) throws IDNotRecognisedException
{
226     // If raceId doesnt relate to a stage Name, throw exception
227     if(getRaceByID(raceId).getRaceName() == "Null"){
228         throw new IDNotRecognisedException("ID "+ raceId + " doesnt exist.");
229     }else{
230         // Else return a string with all details for the race
231         return getRaceByID(raceId).getRaceDetails();
232     }
233 }
234
235 /**
236  * This method is used to remoove a race by its ID.
237  *
238  * @param raceId The unique id of the race to be deleted.
239  * @exception IDNotRecognisedException The ID of the race to be deleted
does not exist.
240  */
241 @Override
242 public void removeRaceById(int raceId) throws IDNotRecognisedException {
243     // Checks if race exists with the given ID then removes it
244     if(getRaceByID(raceId).getRaceName() == "Null"){
245         throw new IDNotRecognisedException("ID " + raceId + " does not exist't
exist");
246     }else{
247         Race race = getRaceByID(raceId);
248         Stage[] stages = race.getStagesV2();
249         for(Stage stage: stages){
250             Segment[] segments = stage.getStageSegments();
251             for(Segment segment: segments){
252                 try{
253                     removeSegment(segment.getSegmentID());
254                 } catch (Exception e){
255                     System.out.println(e);
256                 }
257             }
258             removeStageById(stage.getStageId());
259         }
260         raceArray.remove(raceId-1);

```

```
261     }
262 }
263
264 /**
265  * This method is used to get the number of stages in a race.
266  *
267  * @param raceId The unique id of the race to see the number of stages.
268  * @return Integer of number of stages.
269  * @exception IDNotRecognisedException The ID of the race is not existint
in the platform.
270  */
271 @Override
272 public int getNumberOfStages(int raceId) throws IDNotRecognisedException {
273     // If raceId doesnt relate to a stage Name, throw exception
274     if(getRaceByID(raceId).getRaceName() == "Null"){
275         throw new IDNotRecognisedException("ID "+ raceId + " doesnt exist.");
276     }else{
277         // Else returns the number of stages
278         assert(getRaceByID(raceId).getNumberOfStages() < 0):"Number of stages
is invalid";
279         return getRaceByID(raceId).getNumberOfStages();
280     }
281 }
282
283 /**
284  * This method is used to add a stage to a race.
285  *
286  * @param raceId The unique id of the race to see its details.
287  * @param stageName The name of the stage to be added to the race.
288  * @param description The description of the stage to be added to the
race.
289  * @param length The length in kilometres to be added to the race.
290  * @param stageTime The start time of the stage to be added to the race.
291  * @param type The type of the stage to be added to the race.
292  * @return The ID of the stage that is added to the race.
293  * @exception IDNotRecognisedException The ID of the race is not existint
in the platform.
294  * @exception IllegalNameException The name of the stage already exists in
the platform.
295  * @exception InvalidNameException The name of the stage does not meet the
requirements.
296  * @exception InvalidLengthException The length of the stage must be
larger than 5 kilometres.
297  */
298 @Override
299 public int addStageToRace(int raceId, String stageName, String
description, double length, LocalDateTime startTime,
300     StageType type)
301     throws IDNotRecognisedException, IllegalNameException,
InvalidNameException, InvalidLengthException {
302     // Check parameters
303     if ((stageName == null) || (stageName.isEmpty()) ||
(stageName.length() > 30)){
304         throw new InvalidNameException("Stage name does not meet
requirements");
305     }
```

```

305     }
306     // Check length
307     if (length<5) {
308         throw new InvalidLengthException("Stage length cannot be less than
5 km");
309     }
310     // Check stage Name doesnt exist already
311     for (Race race : raceArray) {
312         for (Stage stage : race.getStages()) {
313             if (stage.getStageName().equals(stageName)) {
314                 throw new IllegalArgumentException("Stage name " + stageName + "
already exists");
315             }
316         }
317     }
318     // Temp holding the race object with the raceId given
319     Race raceTemp = getRaceByID(raceId);
320     // Check raceId exists
321     if (getRaceByID(raceId).getRaceName() == "Null"){
322         throw new IDNotRecognisedException("ID "+ raceId + " doesnt
exist.");
323     } else {
324         // Add stage to the race object
325         Stage stage = new Stage(stageName, description, length,
startTime,type);
326         raceTemp.addStage(stage);
327         return stage.getStageId();
328     }
329 }
330
331 /**
332  * This method is used to get a race stage.
333  *
334  * @param raceId The unique id of the race to see its details.
335  * @return The ID of the stage that is added to the race.
336  */
337 @Override
338 public int[] getRaceStages(int raceId) throws IDNotRecognisedException {
339     // Gets the race objects from the given race id
340     Race race = getRaceByID(raceId);
341     // Uses race object to get all the stages
342     Stage[] stages = race.getStagesV2();
343     int[] stageIds = new int[stages.length];
344     // For each stage get its ID and adds it to an array of integers
345     assert(stages.length < 0):"Invalid amount of stages";
346     for (int i=0; i<stages.length; i++) {
347         stageIds[i] = stages[i].getStageId();
348     }
349     return stageIds;
350 }
351
352 /**
353  * This method is used to get a race stage.
354  *
355  * @param raceId The unique id of the race to see its details.

```

```

356     * @return The ID of the stage that is added to the race.
357     */
358     @Override
359     public double getStageLength(int stageId) throws IDNotRecognisedException
360     {
361         // If stageId doesnt relate to a stage Name, throw exception
362         if(getStageById(stageId).getStageName() == "Null"){
363             throw new IDNotRecognisedException("ID "+ stageId + " doesnt exist.");
364         } else {
365             // Else returns the length of the stage
366             assert(getStageById(stageId).getStageLength() < 0):"Invalid length of
stage";
367             return getStageById(stageId).getStageLength();
368         }
369     }
370     /**
371     * This method is used to remove a race by its ID.
372     *
373     * @param raceId The unique id of the race to be deleted.
374     * @exception IDNotRecognisedException The ID of the race to be deleted
375     does not exist.
376     */
377     @Override
378     public void removeStageById(int stageId) throws IDNotRecognisedException {
379         // If stageId doesnt relate to a stage Name, throw exception
380         if(getStageById(stageId).getStageName() == "Null"){
381             throw new IDNotRecognisedException("ID "+ stageId + " doesnt exist.");
382         } else {
383             // Else removes the stage
384             getRaceByStageId(stageId).removeStage(getStageById(stageId));
385         }
386     }
387     /**
388     * This method is used to add a climb segment to a stage.
389     *
390     * @param stageId The stage ID for the the segment to be added to the
391     stage.
392     * @param location The kilometre location of the segment finishes to be
393     added to the stage.
394     * @param type The category of the climb to be added to the stage.
395     * @param averageGradient The average gradient of the segment to be added
396     to the stage.
397     * @param length The length of the segment in kilometres to be added to
398     the stage.
399     * @return The ID of the segment which was added to the stage.
400     * @exception IDNotRecognisedException The ID of the stage is not existsnt
401     in the platform.
402     * @exception InvalidLocationException The location of the segment is out
403     of the bounds of the stage length.
404     * @exception InvalidStageStateException The stage is in the process of
405     receiving a result and can't receive this results.
406     * @exception InvalidStageTypeException A time-trial stage cannot contain
407     any segments.
408     */

```



```

400  @Override
401  public int addCategorizedClimbToStage(int stageId, Double location,
SegmentType type, Double averageGradient,
402      Double length) throws IDNotRecognisedException,
InvalidLocationException, InvalidStageStateException,
403      InvalidStageTypeException {
404      //Check if stage exists
405      if (getStageByID(stageId).getStageName() == "Null"){
406          throw new IDNotRecognisedException("ID " + stageId + " doesnt exist.");
407      } else {
408          //Define the stage to be added to
409          Stage stage = getStageByID(stageId);
410          //Check if stage location allows the length to be added and is large
than zero
411          if (stage.getStageLength() < location){
412              throw new InvalidLocationException("Location is out of bounds of the
stage length by " + (location-(stage.getStageLength())));
413          }
414          if (location < 0){
415              throw new InvalidLocationException("Location must be more than
zero.");
416          }
417          //Check if stage is time-trialed
418          if (stage.getStageType() == StageType.TT) {
419              throw new InvalidStageTypeException("Time-trial stages cannot
contain any segment.");
420          }
421          //Check if stage is prepared to get results added to
422          if (!stage.isPrepared()) {
423              throw new InvalidStageStateException("Stage cannot be added as its
not prepared.");
424          }
425          //Add segment to the objects
426          Segment segment = new Segment(location,type,averageGradient,length);
427          stage.addStageSegment(segment);
428          return segment.getSegmentID();
429      }
430  }
431
432  /**
433   * This method is used to add a intermediate sprint segment to a stage.
434   *
435   * @param stageId The stage ID for the the segment to be added to the
stage.
436   * @param location The kilometre location of the segment finishes to be
added to the stage.
437   * @return The ID of the segment which was added to the stage.
438   * @exception IDNotRecognisedException The ID of the stage is not existsnt
in the platform.
439   * @exception InvalidLocationException The location of the segment is out
of the bounds of the stage length.
440   * @exception InvalidStageStateException The stage is in the process of
receiving a result and can't receive this results.
441   * @exception InvalidStageTypeException A time-trial stage cannot contain
any segments.

```

```

442     */
443     @Override
444     public int addIntermediateSprintToStage(int stageId, double location)
throws IDNotRecognisedException,
445         InvalidLocationException, InvalidStageStateException,
InvalidStageTypeException {
446         //Check if stage exists
447         if (getStageByID(stageId).getStageName() == "Null"){
448             throw new IDNotRecognisedException("ID " + stageId + " doesnt exist.");
449         } else {
450             //Define the stage to be added to
451             Stage stage = getStageByID(stageId);
452             //Check if stage location allows the length to be added and is large
than zero
453             if (stage.getStageLength() < location){
454                 throw new InvalidLocationException("Location is out of bounds of the
stage length by " + (location-(stage.getStageLength())));
455             }
456             if (location < 0){
457                 throw new InvalidLocationException("Location must be more than
zero.");
458             }
459             //Check if stage is not time-trialed
460             if (stage.getStageType() == StageType.TT) {
461                 throw new InvalidStageTypeException("Time-trial stages cannot
contain any segment.");
462             }
463             //Check if stage is prepared to get results added to
464             if (!stage.isPrepared()) {
465                 throw new InvalidStageStateException("Stage cannot be added as its
not prepaired.");
466             }
467
468             //Add segment to the objects
469             Segment segment = new Segment(location,SegmentType.SPRINT);
470             stage.addStageSegment(segment);
471             return segment.getSegmentID();
472         }
473     }
474
475     /**
476      * This method is used to remove a segement using its ID.
477      *
478      * @param segmentId The unique id of the segment to be deleted.
479      * @exception IDNotRecognisedException The ID of the segment to be deleted
does not exist.
480      * @exception InvalidStageStateException The stage is in the process of
receiving a result and can't remove a stage.
481      */
482     @Override
483     public void removeSegment(int segmentId) throws IDNotRecognisedException,
InvalidStageStateException {
484         //Check if segment exists
485         if (getSegmentByID(segmentId).getSegmentID() == 0){
486             throw new IDNotRecognisedException("ID " + segmentId + " doesnt

```

```

486         throw new IDNotRecognisedException("ID " + segmentId + " doesn't
exist.");
487     } else {
488         //Find the stage the segment belongs to.
489         Stage stage = getStageBySegmentId(segmentId);
490         //Check if the stage is waiting for a result.
491         if (!stage.isPrepared()){
492             throw new InvalidStageStateException("Stage cannot be removed as its
not prepaired.");
493         }
494         //Then remove the segment from the stage.
495         stage.removeStageSegment(getSegmentByID(segmentId));
496     }
497 }
498
499 /**
500  * This method is used to prepare a stage.
501  *
502  * @param stageId The unique id of the stage to be preapred.
503  * @exception IDNotRecognisedException The ID of the stage to be preapred
does not exist.
504  * @exception InvalidStageStateException The stage is in the process of
receiving a result and can't be prepared.
505  */
506 @Override
507 public void concludeStagePreparation(int stageId) throws
IDNotRecognisedException, InvalidStageStateException {
508     //Check if stage exists
509     if (getStageByID(stageId).getStageName() == "Null"){
510         throw new IDNotRecognisedException("ID " + stageId + " doesnt exist.");
511     } else {
512         //Check if stage is waiting for a result
513         if (!getStageByID(stageId).isPrepared()) {
514             throw new InvalidStageStateException("Stage is processing a
result.");
515         } else {
516             //Set stage as prepared.
517             getStageByID(stageId).prepare();
518         }
519     }
520 }
521
522 /**
523  * This method is used to get the IDs of the segments in a given stage by
its ID.
524  *
525  * @param stageId The unique id of the stage which its segments are
wanted.
526  * @return A integer array of the IDs of the segments in the stage.
527  * @exception IDNotRecognisedException The ID of the stage which its
segments are wanted does not exist in the system.
528  */
529 @Override
530 public int[] getStageSegments(int stageId) throws IDNotRecognisedException
{
531     //Check if stage exists

```

```

531 //check if stage exists
532 if (getStageByID(stageId).getStageName() == "Null"){
533     throw new IDNotRecognisedException("ID " + stageId + " doesnt exist.");
534 } else {
535     //Get segments from the stage and store in array of segments
536     Segment[] segments = getStageByID(stageId).getStageSegments();
537     assert(segments.length < 0):"Invalid amount of segments";
538     //Set a temp array of integers which will hold the IDs of the segments
    which is the length of the array of segments
539     int[] segmentIDs = new int[segments.length];
540     // For each segment add its ID to array of IDs
541     for (int i = 0; i < segmentIDs.length; i++){
542         segmentIDs[i] = segments[i].getSegmentID();
543     }
544     //Return the integer array of segment IDs.
545     return segmentIDs;
546 }
547 }
548
549 /**
550  * This method is used to create a new team.
551  *
552  * @param name The name of the team.
553  * @param description The description of the team.
554  * @return The unique ID of the created team.
555  * @exception IllegalNameException Thrown when attempting to assign a
    team name already in use in the system.
556  * @exception InvalidNameException If the name is null, empty, has more
    than 30 characters, or has white spaces.
557  */
558 @Override
559 public int createTeam(String name, String description) throws
    IllegalNameException, InvalidNameException {
560     //Check if the team name already exists
561     for(int i = 0; i < teamArray.size(); i++) {
562         if (teamArray.get(i).getTeamName() == name) {
563             throw new IllegalNameException("Team name already exists in the
    system");
564         }
565     }
566     //Check if team name meets the requirements
567     if((name == null) || (name.length() > 30) || (name.contains(" ") ||
    (name == ""))) {
568         throw new InvalidNameException("Team name cannot be null, empty,
    longer than 30 characters or contain a white space");
569     }
570     //Create a new team object
571     Team team = new Team(name, description);
572     teamArray.add(team);
573     return team.getTeamID();
574 }
575 }
576
577 /**
578  * This method is used to remove a team.
579  */

```

```

579     ^
580     * @param teamId The team ID to be removed.
581     * @exception IDNotRecognisedException The ID of the team to be removed
does not exist.
582     */
583     @Override
584     public void removeTeam(int teamId) throws IDNotRecognisedException {
585         //Check if the teamId exists
586         if (getTeamByID(teamId).getTeamName() == "Null"){
587             throw new IDNotRecognisedException("ID "+ teamId + " doesnt exist.");
588         } else {
589             //Remove team from array
590             teamArray.remove(getTeamByID(teamId));
591         }
592     }
593
594     /**
595     * This method is used to get all team IDs on the platform.
596     *
597     * @return Array of integers containing the team IDs.
598     */
599     @Override
600     public int[] getTeams() {
601         //Temp array of integers to hold all IDs
602         int[] teamIDs = new int[teamArray.size()];
603         //Add each team ID to the array of integers and then return the array
604         for (int i=0; i<teamArray.size(); i++) {
605             teamIDs[i] = teamArray.get(i).getTeamID();
606         }
607         return teamIDs;
608     }
609
610     /**
611     * This method is used to get all rider IDs in the team.
612     *
613     * @param teamId The team ID which all riders that are a part of are
wanted.
614     * @return Array of integers containing the rider IDs that are a part of
this team.
615     * @exception IDNotRecognisedException The ID of the team to find the
riders does not exists.
616     */
617     @Override
618     public int[] getTeamRiders(int teamId) throws IDNotRecognisedException {
619         //Check if the teamId exists
620         if (getTeamByID(teamId).getTeamName() == "Null"){
621             throw new IDNotRecognisedException("ID "+ teamId + " doesnt exist.");
622         } else {
623             //Create a temp integer array of size of number of riders in the team
624             Rider[] riders = getTeamByID(teamId).getRiders();
625             int[] riderIds = new int[riders.length];
626             //For each rider in team add there ID to the array
627             for (int i=0; i<riderIds.length; i++) {
628                 riderIds[i] = riders[i].getRiderID();
629             }
630             //Return the integer array of rider IDs in the team

```

```

630         //Return the integer array of rider IDs in the team
631         return riderIds;
632     }
633 }
634
635 /**
636  * This method is used to create a new rider.
637  *
638  * @param teamId The team which the rider belongs to.
639  * @param name The name of the rider.
640  * @param yearOfBirth The year of birth of the rider.
641  * @return The unique ID of the created rider.
642  * @exception IDNotRecognisedException The teamID does not match a team in
the system.
643  * @exception IllegalArgumentException The name of the rider is null or
the year of birth is less than 1900.
644  */
645 @Override
646 public int createRider(int teamID, String name, int yearOfBirth) throws
IDNotRecognisedException, IllegalArgumentException {
647     //Check if the teamID exists
648     if (getTeamByID(teamID).getTeamName() == "Null"){
649         throw new IDNotRecognisedException("ID " + teamID + " doesnt exist.");
650     } else {
651         //Check if name and year of birth match the requirements
652         if((name == null) || (yearOfBirth < 1900)){
653             throw new IllegalArgumentException("Name or Year of Birth do not
match the requirements.");
654         } else {
655             //Create a new rider object
656             assert(yearOfBirth>1900 && yearOfBirth<2010):"Invalid rider year of
birth";
657             Rider rider = new Rider(teamID, name , yearOfBirth);
658             riderArray.add(rider);
659             getTeamByID(teamID).addRider(rider);
660             return rider.getRiderID();
661         }
662     }
663 }
664
665 /**
666  * This method is used to remove a rider.
667  *
668  * @param teamId The rider ID to be removed.
669  * @exception IDNotRecognisedException The ID of the rider to be removed
does not exist.
670  */
671 @Override
672 public void removeRider(int riderId) throws IDNotRecognisedException {
673     for (Team team : teamArray) {
674         for (Rider rider : team.getRiders()) {
675             if (rider.getRiderID() == riderId) {
676                 team.removeRider(getRiderByID(riderId));
677                 Results[] results = rider.getRiderResults();
678                 for(Results result: results){

```

```

679         stage stage = result.getResultStage();
680         deleteRiderResultsInStage(stage.getStageId(), riderId);
681     }
682     rider.setRiderName("Null");
683 }
684 }
685 }
686 assert (getRiderByID(riderId).getRiderName() != null) :
687     new IDNotRecognisedException("ID " + riderId + " does not match any
riders in system.");
688 }
689
690 /**
691  * This method is used to register a result for a rider.
692  *
693  * @param stageId The stage ID the result refers to being added.
694  * @param riderId The ID of the rider that the result is being added for.
695  * @param checkpoints An array of times at which the rider reached each of
the segments of the stage (including start and finish time).
696  * @exception IDNotRecognisedException The ID of the rider or stage does
not exist.
697  * @exception DuplicatedResultException The rider has already had a result
added for this specific stage.
698  * @exception InvalidCheckpointsException The length of the checkpoints
must be equal to n+2 of the number of segments in the stage
699  * (+2 indicates the start and finish times being
includes aswell)
700  * @exception InvalidStageStateException the stage is waiting for a result
and is not prepared to register another result yet.
701  */
702 @Override
703 public void registerRiderResultsInStage(int stageId, int riderId,
LocalTime... checkpoints)
704     throws IDNotRecognisedException, DuplicatedResultException,
InvalidCheckpointsException,
705     InvalidStageStateException {
706     //Check rider exists
707     if(getRiderByID(riderId).getRiderName() == "Null"){
708         throw new IDNotRecognisedException("ID " + riderId + "does not
exist");
709     }
710     //Check stage exists
711     if(getStageByID(stageId).getStageName() == "Null"){
712         throw new IDNotRecognisedException("ID " + stageId + "does not
exist");
713     }
714     //Get the objects of the rider and stage IDs relate to
715     Rider rider = getRiderByID(riderId);
716     Stage stage = getStageByID(stageId);
717     //Check stage is prepared for registering a result
718     if(!stage.isPrepared()){
719         throw new InvalidStageStateException("Stage is waiting for a result");
720     }
721     //Check length of checkpoints is number of segments + 2
722     if(checkpoints.length != stage.getStageSegments().length + 2){

```



```

723     throw new InvalidCheckpointsException("Number of checkpoints must be
number of segments + 2");
724 }
725 //Check the rider doesnt have a result for this stage already
726 for(int i = 0; i < stage.getStageResults().size(); i++){
727     if (stage.getStageResults().get(i).getResultRider() == rider){
728         throw new DuplicatedResultException("Stage already has a result for
this rider");
729     }
730 }
731 //Register the results
732 Results result = new Results(stage , rider, checkpoints);
733 stage.addStageResults(result);
734 rider.addRiderResult(result);
735 }
736
737 /**
738  * This method gets the all the results of a given rider in a given stage,
segments included.
739  * @param stageId The ID of the stage where the results will come from.
740  * @param riderId The ID of the rider which will be used to get the
results.
741  * @exception IDNotRecognisedException The ID of the rider or stage does
not exist.
742  * @return Result times based on the rider's ID and Stage ID.
743  */
744 @Override
745 public LocalTime[] getRiderResultsInStage(int stageId, int riderId) throws
IDNotRecognisedException {
746     boolean exist = false;
747     //Check rider exists
748     if(getRiderByID(riderId).getRiderName() == null){
749         throw new IDNotRecognisedException("ID " + riderId + "does not
exist");
750     }
751     //Check stage exists
752     if(getStageByID(stageId).getStageName() == null){
753         throw new IDNotRecognisedException("ID " + stageId + "does not
exist");
754     }
755     //Get the objects of the rider and stage object related to their IDs
756     Rider rider = getRiderByID(riderId);
757     Stage stage = getStageByID(stageId);
758
759     //Check rider does not have a result in that stage
760     for (Results result : rider.getRiderResults()) {
761         if (result.getResultStage().equals(stage)) {
762             exist = true;
763             return result.getResultTimes();
764         }
765     }
766
767     //Check rider doesnt have result
768     if(!exist){
769         throw new IDNotRecognisedException("Rider " + riderId + " doesnt have

```



```

a result");
770     }
771     return null;
772 }
773 /**
774  * This method finds the elapsed time the rider was in the stage for.
775  * @param stageId The ID of the stage where the results will come from.
776  * @param riderId The ID of the rider which will be used to get the
777  results.
778  * @exception IDNotRecognisedException The ID of the rider or stage does
not exist.
779  * @return The finish time - start time to give you the duration between
the start and finish.
780  */
781 @Override
782 public LocalTime getRiderAdjustedElapsedTimeInStage(int stageId, int
riderId) throws IDNotRecognisedException {
783     //Check rider exists
784     if(getRiderByID(riderId).getRiderName() == null){
785         throw new IDNotRecognisedException("ID " + riderId + "does not
exist");
786     }
787     //Check stage exists
788     if(getStageByID(stageId).getStageName() == null){
789         throw new IDNotRecognisedException("ID " + stageId + "does not
exist");
790     }
791     //Get rider results as an array of times
792     LocalTime[] temp = getRiderResultsInStage(stageId, riderId);
793     //Find last time
794     int lastTime = temp.length - 1;
795     //If rider has at least 2 times registered
796     if(lastTime < 1) {
797         return null;
798     } else {
799         //assert (temp[lastTime] == temp[0]) : return null;
800         //Find the duration between the start time and end time for hours,mins
and secs
801         int timeHor = (int) Duration.between(temp[0],
temp[lastTime]).toHoursPart();
802         int timeMin = (int) Duration.between(temp[0],
temp[lastTime]).toMinutesPart();
803         int timeSec = (int) Duration.between(temp[0],
temp[lastTime]).toSecondsPart();
804         //Set a LocalTime for the durations between the start and finish
805         LocalTime timeOvr = LocalTime.of(timeHor, timeMin, timeSec);
806         return timeOvr;
807     }
808 }
809 /**
810  * This method deletes the results of a given rider in a given stage,
based on the rider and stage ID.
811  * @param stageId The ID of the stage where the results will come from.
812  * @param riderId The ID of the rider which will be used to get the

```

```
results.
813  * @exception IDNotRecognisedException The ID of the rider or stage does
not exist.
814  */
815  @Override
816  public void deleteRiderResultsInStage(int stageId, int riderId) throws
IDNotRecognisedException {
817      boolean found = false;
818      //Check rider exists
819      if(getRiderByID(riderId).getRiderName() == null){
820          throw new IDNotRecognisedException("ID " + riderId + "does not
exist");
821      }
822      //Check stage exists
823      if(getStageByID(stageId).getStageName() == null){
824          throw new IDNotRecognisedException("ID " + stageId + "does not
exist");
825      }
826      //Set rider and stage objects from there IDs
827      Stage stage = getStageByID(stageId);
828      Rider rider = getRiderByID(riderId);
829      //Get riders results as an array
830      Results[] results = rider.getRiderResults();
831      //Check that for the stage the rider has a result
832      for (Results results2 : results) {
833          if(results2.getResultStage().equals(stage)){
834              if(results2.getResultRider().equals(rider)){
835                  //Remove result from both stage and rider
836                  stage.removeResults(results2);
837                  rider.removeResults(results2);
838                  found = true;
839              }
840          }
841      }
842      //If not found
843      if(!found){
844          throw new IDNotRecognisedException("Stage doesnt have a result for
this rider");
845      }
846  }
847
848  /**
849   * This gets a list of rider IDs sorted by there finishing time.
850   * @param stageId The ID of the stage where the results will come from.
851   * @param riderId The ID of the rider which will be used to get the
results.
852   * @exception IDNotRecognisedException The ID of the rider or stage does
not exist.
853   */
854  @Override
855  public int[] getRidersRankInStage(int stageId) throws
IDNotRecognisedException {
856      //Get sorted times of riders
857      LocalTime[] timesSorted = getRankedAdjustedElapsedTimesInStage(stageId);
858      //Set stage and result objects
```

```

859     Stage stage = getStageByID(stageId);
860     //If no stages return empty list
861     int[] leader = new int[0];
862     if(stage.getStageLength() == 0){
863         return leader;
864     }
865     ArrayList<Results> results = stage.getStageResults();
866     //Get a temporal leaderboard
867     int[] temporal= new int[riderArray.size()];
868     //Check that for the stage the rider has a result
869     int i = 1;
870     for (Results results2 : results) {
871         if(results2.getResultStage().equals(stage)){
872             temporal[i] = results2.getResultRider().getRiderID();
873             i++;
874         }
875     }
876     //Get the length the leaderboard needs to be
877     int count = 0;
878     for(int x = 0; x<riderArray.size();x++){
879         if(temporal[x] != 0){
880             count++;
881         }
882     }
883     //Set leaderboard of times to right size so theres no overhang
884     int[] leaderboard = new int[count];
885     //For each index increment so its added to right leaderboard spot
886     int flagging = 0;
887     for(int y = 0; y<riderArray.size(); y++){
888         if(temporal[y] != 0){
889             leaderboard[flagging] = temporal[y];
890             flagging++;
891         }
892     }
893     //Set leaderboard of IDs to the right size
894     int[] sortedLeaderboard = new int[count];
895     int a = 0;
896     for(int b = 0; b<count; b++){
897         //Set the rider ID for its corresponding position in the sorted time
898         array //Similar to linear search
899         if(getRiderAdjustedElapsedTimeInStage(stageId,
900         leaderboard[b]).equals(timesSorted[a])){
901             sortedLeaderboard[a] = leaderboard[b];
902             a++;
903             if(a==count){
904                 break;
905             }
906             //Resets search
907             b=-1;
908         }
909     }
910     return sortedLeaderboard;
911 }
912 /**

```

```
912     * This method gets the adjusted alapsed times of the riders in the stage,
and ranks them based on fastest elapsed time to slowest.
913     * @param stageId The ID of the stage where the results will come from.
914     * @exception IDNotRecognisedException The ID of the rider or stage does
not exist.
915     * @return Result times based on Stage ID and ranked with fastest elapsed
time to slowest.
916     */
917     @Override
918     public LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId)
throws IDNotRecognisedException {
919         //Check stage exists
920         if(getStageByID(stageId).getStageName() == null){
921             throw new IDNotRecognisedException("Stage " + stageId + "doesnt
exist");
922         }
923         //Set stage and result objects
924         Stage stage = getStageByID(stageId);
925         ArrayList<Results> results = stage.getStageResults();
926         //Get a temporal leaderboard
927         int[] temporal= new int[riderArray.size()];
928         //Check that for the stage the rider has a result
929         int i = 1;
930         for (Results results2 : results) {
931             if(results2.getResultStage().equals(stage)){
932                 temporal[i] = results2.getResultRider().getRiderID();
933                 i++;
934             }
935         }
936         //Get the length the leaderboard needs to be
937         int count = 0;
938         for(int x = 0; x<riderArray.size();x++){
939             if(temporal[x] != 0){
940                 count++;
941             }
942         }
943         //Set leaderboard to right size so theres no overhang
944         int[] leaderboard = new int[count];
945         //For each index increment so its added to right leaderboard spot
946         int flagging = 0;
947         for(int y = 0; y<riderArray.size(); y++){
948             if(temporal[y] != 0){
949                 leaderboard[flagging] = temporal[y];
950                 flagging++;
951             }
952         }
953         //Get elapsed time
954         LocalTime[] times = new LocalTime[count];
955         for(int j = 0; j<count; j++){
956             if(leaderboard[j] == 0){
957                 break;
958             } else {
959                 times[j] = getRiderAdjustedElapsedTimeInStage(stageId,
leaderboard[j]);
960             }
961         }
```

```

961     }
962     //Sort times
963     Arrays.sort(times);
964     return times;
965 }
966
967 /**
968  * This gets a list of rider points in the stage by there finishing time.
969  * @param stageId The ID of the stage where the results will come from.
970  * @return an array of rider points in a specific stage.
971  * @exception IDNotRecognisedException The ID of the stage does not exist.
972  */
973 @Override
974 public int[] getRidersPointsInStage(int stageId) throws
IDNotRecognisedException {
975     //Check stage exists
976     if(getStageByID(stageId).getStageName() == null){
977         throw new IDNotRecognisedException("Stage ID "+stageId+" doesnt
exist.");
978     }
979     //Get objects for stage, array of sorted riders & stage type
980     Stage stage = getStageByID(stageId);
981     int[] sortedRiders = getRidersRankInStage(stageId);
982     //Checks if no results exists return empty list
983     int[] leader = new int[0];
984     if(sortedRiders.length == 0){
985         return leader;
986     }
987     StageType stageType = stage.getStageType();
988     //Create arrays for points
989     int[] stagePoints = new int[sortedRiders.length];
990     //For each rider in relative position
991     for(int i = 0; i<sortedRiders.length; i++){
992         //Get rider to then set there points in object class
993         Rider rider = getRiderByID(sortedRiders[i]);
994         //If position is more than 15 no points
995         if(i>15){
996             stagePoints[i] = 0;
997         } else {
998             //Switch for each type of stage
999             switch (stageType) {
1000                 case FLAT:
1001                     //Switch for each position to reward points
1002                     switch (i+1) {
1003                         case 1:
1004                             stagePoints[i] = 50;
1005                             break;
1006                         case 2:
1007                             stagePoints[i] = 30;
1008                             break;
1009                         case 3:
1010                             stagePoints[i] = 20;
1011                             break;
1012                         case 4:
1013                             stagePoints[i] = 10;

```

```
1013         stagePoints[i] = 10;
1014         break;
1015     case 5:
1016         stagePoints[i] = 16;
1017         break;
1018     case 6:
1019         stagePoints[i] = 14;
1020         break;
1021     case 7:
1022         stagePoints[i] = 12;
1023         break;
1024     case 8:
1025         stagePoints[i] = 10;
1026         break;
1027     case 9:
1028         stagePoints[i] = 8;
1029         break;
1030     case 10:
1031         stagePoints[i] = 7;
1032         break;
1033     case 11:
1034         stagePoints[i] = 6;
1035         break;
1036     case 12:
1037         stagePoints[i] = 5;
1038         break;
1039     case 13:
1040         stagePoints[i] = 4;
1041         break;
1042     case 14:
1043         stagePoints[i] = 3;
1044         break;
1045     case 15:
1046         stagePoints[i] = 2;
1047         break;
1048     default:
1049         //Incase didnt catch position
1050         stagePoints[i] = 0;
1051         break;
1052     }
1053     break;
1054 case MEDIUM_MOUNTAIN:
1055     //Switch for each position to reward points
1056     switch (i+1) {
1057     case 1:
1058         stagePoints[i] = 30;
1059         break;
1060     case 2:
1061         stagePoints[i] = 25;
1062         break;
1063     case 3:
1064         stagePoints[i] = 22;
1065         break;
1066     case 4:
1067         stagePoints[i] = 19;
1068         break;
```

```
1068         break;
1069     case 5:
1070         stagePoints[i] = 17;
1071         break;
1072     case 6:
1073         stagePoints[i] = 15;
1074         break;
1075     case 7:
1076         stagePoints[i] = 13;
1077         break;
1078     case 8:
1079         stagePoints[i] = 11;
1080         break;
1081     case 9:
1082         stagePoints[i] = 9;
1083         break;
1084     case 10:
1085         stagePoints[i] = 7;
1086         break;
1087     case 11:
1088         stagePoints[i] = 6;
1089         break;
1090     case 12:
1091         stagePoints[i] = 5;
1092         break;
1093     case 13:
1094         stagePoints[i] = 4;
1095         break;
1096     case 14:
1097         stagePoints[i] = 3;
1098         break;
1099     case 15:
1100         stagePoints[i] = 2;
1101         break;
1102     default:
1103         //Incase didnt catch position
1104         stagePoints[i] = 0;
1105         break;
1106     }
1107     break;
1108 case HIGH_MOUNTAIN:
1109     //Switch for each position to reward points
1110     switch (i+1) {
1111     case 1:
1112         stagePoints[i] = 20;
1113         break;
1114     case 2:
1115         stagePoints[i] = 17;
1116         break;
1117     case 3:
1118         stagePoints[i] = 15;
1119         break;
1120     case 4:
1121         stagePoints[i] = 13;
1122         break;
1123     case 5:
1124         stagePoints[i] = 11;
1125         break;
1126     case 6:
1127         stagePoints[i] = 9;
1128         break;
1129     case 7:
1130         stagePoints[i] = 7;
1131         break;
1132     case 8:
1133         stagePoints[i] = 6;
1134         break;
1135     case 9:
1136         stagePoints[i] = 5;
1137         break;
1138     case 10:
1139         stagePoints[i] = 4;
1140         break;
1141     case 11:
1142         stagePoints[i] = 3;
1143         break;
1144     case 12:
1145         stagePoints[i] = 2;
1146         break;
1147     case 13:
1148         stagePoints[i] = 1;
1149         break;
1150     case 14:
1151         stagePoints[i] = 0;
1152         break;
1153     default:
1154         stagePoints[i] = 0;
1155         break;
1156     }
1157     break;
1158 }
```

```
1123         case 5:
1124             stagePoints[i] = 11;
1125             break;
1126         case 6:
1127             stagePoints[i] = 10;
1128             break;
1129         case 7:
1130             stagePoints[i] = 9;
1131             break;
1132         case 8:
1133             stagePoints[i] = 8;
1134             break;
1135         case 9:
1136             stagePoints[i] = 7;
1137             break;
1138         case 10:
1139             stagePoints[i] = 6;
1140             break;
1141         case 11:
1142             stagePoints[i] = 5;
1143             break;
1144         case 12:
1145             stagePoints[i] = 4;
1146             break;
1147         case 13:
1148             stagePoints[i] = 3;
1149             break;
1150         case 14:
1151             stagePoints[i] = 2;
1152             break;
1153         case 15:
1154             stagePoints[i] = 1;
1155             break;
1156         default:
1157             //Incase didnt catch position
1158             stagePoints[i] = 0;
1159             break;
1160     }
1161     default:
1162         //Incase didnt catch position
1163         stagePoints[i] = 0;
1164         break;
1165 }
1166 //Set rider points in object
1167 rider.addRiderPoints(stagePoints[i]);
1168 }
1169 }
1170 return stagePoints;
1171 }
1172 }
1173
1174 /**
1175  * This gets a list of rider points in the mountain stage by getting the
1176  * time of each segment they crossed.
1177  * @param stageId The ID of the stage where the results will come from.
```



```

1177 * @return an array of rider points in a specific stage where the segments
1178 are mountains.
1179 * @exception IDNotRecognisedException The ID of the stage does not exist.
1180 */
1181 @Override
1182 public int[] getRidersMountainPointsInStage(int stageId) throws
1183 IDNotRecognisedException {
1184     //Check stage exists
1185     if(getStageByID(stageId).getStageName() == null){
1186         throw new IDNotRecognisedException("Stage ID "+stageId+" doesnt
1187 exist.");
1188     }
1189     //Get objects for stage, array of sorted riders & stage segments
1190     Stage stage = getStageByID(stageId);
1191     int[] sortedRiders = getRidersRankInStage(stageId);
1192     //Checks if no results exists return empty list
1193     int[] leader = new int[0];
1194     if(sortedRiders.length == 0){
1195         return leader;
1196     }
1197     Segment[] segmentType = stage.getStageSegments();
1198     //Create arrays for riders and points
1199     Rider[] riders = new Rider[sortedRiders.length];
1200     int[] pointStage = new int[sortedRiders.length];
1201     //Set a HashMap that maps rider objects to times for each segment
1202     HashMap<Rider, LocalTime> resultMap = new HashMap<Rider, LocalTime>();
1203     //Sets the rider IDs into rider Objects in a array
1204     for (int z=0;z<sortedRiders.length;z++) {
1205         riders[z] = getRiderByID(sortedRiders[z]);
1206     }
1207     //For each segment type
1208     for(int j = 0; j<segmentType.length; j++){
1209         SegmentType type = segmentType[j].getSegmentType();
1210         //Check that segment type is not sprint
1211         if(type != SegmentType.SPRINT){
1212             //Loop through results
1213             for(int i = 0; i<segmentType.length; i++){
1214                 //Check for each rider if they exists
1215                 for (Rider rider : riders) {
1216                     if(rider.getRiderName() == null){
1217                         continue;
1218                     }
1219                     //Get the riders segment times for each segment starting from
1220                     first segment to last one
1221                     LocalTime[] results = getRiderResultsInStage(stageId,
1222 rider.getRiderID());
1223                     if (results != null) {
1224                         assert (results.length == segmentType.length):"Cant set a
1225 result for a segment that doesnt exist";
1226                         int timeHor = (int) Duration.between(results[0],
1227 results[i+1]).toHoursPart();
1228                         int timeMin = (int) Duration.between(results[0],
1229 results[i+1]).toMinutesPart();
1230                         int timeSec = (int) Duration.between(results[0],
1231 results[i+1]).toSecondsPart();

```

```
1223         LocalTime segmentOvr = LocalTime.of(timeHor, timeMin,
timeSec);
1224         resultMap.put(rider, segmentOvr);
1225     }
1226 }
1227 //Sort HashMap by segment crossing
1228 Map<Rider, LocalTime> sortedMap =
1229     resultMap.entrySet().stream()
1230         .sorted(Entry.comparingByValue())
1231         .collect(Collectors.toMap(Entry::getKey, Entry::getValue, (e1,
e2) -> e1, LinkedHashMap::new));
1232 //Set position position of rider
1233 int pos = 1;
1234 //For each rider in HashMap
1235 for (Rider rider : sortedMap.keySet()) {
1236     //If there position is more than 8 they dont get points
1237     if (pos > 8) {
1238         break;
1239     }
1240     //Check each rider ID relates to the rider in the HashMap
1241     for (int b=0;b<riders.length;b++) {
1242         if (riders[b].equals(rider)){
1243             //Switch for each type of segment
1244             switch (type) {
1245                 case C4:
1246                     //Switch for each position to reward points
1247                     switch(pos){
1248                         case 1:
1249                             pointStage[b] = 1;
1250                             pos++;
1251                             break;
1252                         default:
1253                             pointStage[b] = 0;
1254                             pos++;
1255                             break;
1256                     }
1257                     break;
1258                 case C3:
1259                     //Switch for each position to reward points
1260                     switch(pos){
1261                         case 1:
1262                             pointStage[b] = 2;
1263                             pos++;
1264                             break;
1265                         case 2:
1266                             pointStage[b] = 1;
1267                             pos++;
1268                             break;
1269                         default:
1270                             pointStage[b] = 0;
1271                             pos++;
1272                             break;
1273                     }
1274                     break;
1275                 case C2:
```

```
1276 //Switch for each position to reward points
1277 switch(pos){
1278     case 1:
1279         pointStage[b] = 5;
1280         pos++;
1281         break;
1282     case 2:
1283         pointStage[b] = 3;
1284         pos++;
1285         break;
1286     case 3:
1287         pointStage[b] = 2;
1288         pos++;
1289         break;
1290     default:
1291         pointStage[b] = 0;
1292         pos++;
1293         break;
1294 }
1295 break;
1296 case C1:
1297     //Switch for each position to reward points
1298     switch(pos){
1299         case 1:
1300             pointStage[b] = 10;
1301             pos++;
1302             break;
1303         case 2:
1304             pointStage[b] = 8;
1305             pos++;
1306             break;
1307         case 3:
1308             pointStage[b] = 6;
1309             pos++;
1310             break;
1311         case 4:
1312             pointStage[b] = 4;
1313             pos++;
1314             break;
1315         case 5:
1316             pointStage[b] = 2;
1317             pos++;
1318             break;
1319         case 6:
1320             pointStage[b] = 1;
1321             pos++;
1322             break;
1323         default:
1324             pointStage[b] = 0;
1325             pos++;
1326             break;
1327     }
1328     break;
1329 case HC:
1330     //Switch for each position to reward points
```

```
1331         switch(pos){
1332             case 1:
1333                 pointStage[b] = 20;
1334                 pos++;
1335                 break;
1336             case 2:
1337                 pointStage[b] = 15;
1338                 pos++;
1339                 break;
1340             case 3:
1341                 pointStage[b] = 12;
1342                 pos++;
1343                 break;
1344             case 4:
1345                 pointStage[b] = 10;
1346                 pos++;
1347                 break;
1348             case 5:
1349                 pointStage[b] = 8;
1350                 pos++;
1351                 break;
1352             case 6:
1353                 pointStage[b] = 6;
1354                 pos++;
1355                 break;
1356             case 7:
1357                 pointStage[b] = 4;
1358                 pos++;
1359                 break;
1360             case 8:
1361                 pointStage[b] = 2;
1362                 pos++;
1363                 break;
1364             default:
1365                 pointStage[b] = 0;
1366                 pos++;
1367                 break;
1368         }
1369         break;
1370     default:
1371         pointStage[b] = 0;
1372         pos++;
1373         break;
1374     }
1375 }
1376     rider.addRiderPoints(pointStage[b]);
1377 }
1378 }
1379 }
1380 }
1381 }
1382
1383     return pointStage;
1384 }
1385 }
```

```
1386 /**
1387  * Erases all the records from the system as if it was new.
1388  */
1389 @Override
1390 public void eraseCyclingPortal() {
1391     // Clear Arrays
1392     teamArray.clear();
1393     raceArray.clear();
1394     riderArray.clear();
1395
1396     // Reset counts of everything
1397     Race.resetNumberOfRaces();
1398     Race.resetNumberOfStages();
1399     Rider.resetNumberOfRiders();
1400     Segment.resetNumberOfSegments();
1401     Stage.resetNumberOfSegments();
1402     Stage.resetNumberOfStages();
1403     Team.resetNumberOfRiders();
1404     Team.resetNumberOfTeams();
1405 }
1406
1407 /**
1408  * Saves the system data onto a file of a given name.
1409  * @param filename the name of the file to be saved.
1410  * @exception IOException failed or interrupted I/O operation.
1411  */
1412 @Override
1413 public void saveCyclingPortal(String filename) throws IOException {
1414     ObjectOutputStream saveFile = new ObjectOutputStream(new
1415     FileOutputStream(filename));
1416     try{
1417         saveFile.writeObject(this);
1418     } finally {
1419         saveFile.close();
1420     }
1421 }
1422
1423 /**
1424  * Loads the data from a given file name onto the system from where it was
1425  * saved.
1426  * @param filename the name of the file to be loaded.
1427  * @exception IOException failed or interrupted I/O operation.
1428  * @exception ClassNotFoundException class with name could not be found.
1429  */
1430 @Override
1431 public void loadCyclingPortal(String filename) throws IOException,
1432 ClassNotFoundException {
1433     ObjectInputStream loadFile = new ObjectInputStream(new
1434     FileInputStream(filename));
1435     try {
1436         CyclingPortal obj = (CyclingPortal) loadFile.readObject();
1437         this.raceArray = obj.raceArray;
1438         this.teamArray = obj.teamArray;
1439         this.riderArray = obj.riderArray;
1440     }
```

```
1437     finally {
1438         loadFile.close();
1439     }
1440
1441 }
1442
1443 /**
1444  * This method is used to remove a race by its name.
1445  *
1446  * @param name The name of the race to be deleted.
1447  * @exception NameNotRecognisedException when using a name that does not
1448  exist.
1449  */
1450 @Override
1451 public void removeRaceByName(String name) throws
1452 NameNotRecognisedException {
1453     //For each race in the race Array
1454     //Linear Search Big O: O(N) Space complexity: O(1)
1455     for (Race race : raceArray) {
1456         //If the race object name is equalivant
1457         if (race.getRaceName() == name) {
1458             //Remove that race object from the array
1459             raceArray.remove(race);
1460             return;
1461         }
1462     }
1463     //If no race is found throw error
1464     throw new NameNotRecognisedException("No race with name " + name + "
1465 exists.");
1466 }
1467
1468 /**
1469  * This method returns a list of race elapsed times based on sorted
1470 elapsed overall times.
1471  * @param raceId the ID of the race.
1472  * @return an array of finish times.
1473  * @exception IDNotRecognisedException ID does not exist in the system.
1474  */
1475 @Override
1476 public LocalTime[] getGeneralClassificationTimesInRace(int raceId) throws
1477 IDNotRecognisedException {
1478     //Check race exists
1479     if (getRaceByID(raceId).getRaceName() == null){
1480         throw new IDNotRecognisedException("Race with ID " + raceId + " doesnt
1481 exist in the system");
1482     }
1483     //Get the race by its ID
1484     Race race = getRaceByID(raceId);
1485     //Get stages in a array by the race ID
1486     Stage[] stages = race.getStagesV2();
1487     //If no stages in race return empty array
1488     LocalTime[] leader = new LocalTime[0];
1489     if (stages.length == 0){
1490         return leader;
1491     }
1492 }
```

```

1486 //Creates a new array list of riders
1487 ArrayList<Rider> riders = new ArrayList<>();
1488 //Setting all riders in race in the array using the stages
1489 for(Stage stageFindRider : stages){
1490     int[] tempHoldID = getRidersRankInStage(stageFindRider.getStageId());
1491     for(int tempIDs : tempHoldID){
1492         if(riders.contains(getRiderByID(tempIDs))){
1493             break;
1494         } else {
1495             riders.add(getRiderByID(tempIDs));
1496         }
1497     }
1498 }
1499 //Create a HashMap to calculate elapsed race time through elapsed stage
time
1500 HashMap<Rider, LocalTime> riderRaceElapsed = new HashMap<Rider,
LocalTime>();
1501 for (Rider rider : riders) {
1502     riderRaceElapsed.put(rider, LocalTime.of(0, 0, 0));
1503     for (Stage stage : stages) {
1504         LocalTime tempTimes =
getRiderAdjustedElapsedTimeInStage(stage.getStageId(),rider.getRiderID());
1505         if (tempTimes != null) {
1506             riderRaceElapsed.replace(rider,
riderRaceElapsed.get(rider).plusHours(tempTimes.getHour())
1507             .plusMinutes(tempTimes.getMinute()).plusSeconds(tempTimes.getSecond()));
1508         }
1509     }
1510 }
1511 //Sort by elapsed race time
1512 Map<Rider, LocalTime> sortedRiders =
1513     riderRaceElapsed.entrySet().stream()
1514     .sorted(Entry.comparingByValue())
1515     .collect(Collectors.toMap(Entry::getKey, Entry::getValue,(e1,
e2) -> e1, LinkedHashMap::new));
1516 //Transform HashMap to an array
1517 LocalTime[] timesSorted = new LocalTime[riders.size()];
1518 sortedRiders.values().toArray(timesSorted);
1519 //Return array of race finish times
1520 return timesSorted;
1521 }
1522
1523 /**
1524  * This method returns a list of riders' points, sorted by the total
elapsed time.
1525  * @param raceId ID of the race.
1526  * @return an array of riders' points in the race.
1527  * @exception IDNotRecognisedException ID does not exist in the system.
1528  */
1529 @Override
1530 public int[] getRidersPointsInRace(int raceId) throws
IDNotRecognisedException {
1531     //Check race exists

```

```

1532 //CHECK RACE EXISTS
1533 if(getRaceByID(raceId).getRaceName() == null){
1534     throw new IDNotRecognisedException("Race with ID " + raceId + "
doesn't exist in the system");
1535 }
1536 //Get the race by its ID
1537 Race race = getRaceByID(raceId);
1538 //Get stages in a array by the race ID
1539 Stage[] stages = race.getStagesV2();
1540 //If no stages in race return empty array
1541 int[] leader = new int[0];
1542 if(stages.length == 0){
1543     return leader;
1544 }
1545 //Creates a new array list of riders
1546 ArrayList<Rider> riders = new ArrayList<>();
1547 //Setting all riders in race in the array using the stages
1548 for(Stage stageFindRider : stages){
1549     int[] tempHoldID = getRidersRankInStage(stageFindRider.getStageId());
1550     for(int tempIDs : tempHoldID){
1551         if(riders.contains(getRiderByID(tempIDs))){
1552             break;
1553         } else {
1554             riders.add(getRiderByID(tempIDs));
1555         }
1556     }
1557 }
1558
1559 Map<Rider, Integer> riderRacePoints = new HashMap<Rider, Integer>();
1560 for (Rider rider : riders) {
1561     // Riders are added to HashMap
1562     riderRacePoints.put(rider, 0);
1563     for (Stage stage : stages) {
1564         // retrieves an array of ranked rider IDs in the stages
1565         int[] ranks = getRidersRankInStage(stage.getStageId());
1566         // Finds the index of the current rider in the array
1567         int indexOfRider = -1;
1568         for (int i=0; i<ranks.length; i++) {
1569             if (ranks[i] == rider.getRiderID()) {
1570                 indexOfRider = i;
1571             }
1572         }
1573         if (indexOfRider != -1) {
1574             int[] pointsArr = getRidersPointsInStage(stage.getStageId());
1575             int points = pointsArr[indexOfRider];
1576
1577             // Adds stage points to existing race points
1578             riderRacePoints.replace(rider, riderRacePoints.get(rider) +
points);
1579         }
1580     }
1581 }
1582
1583 // Creates array to store points
1584 int[] sortedPoints = new int[riders.size()];
1585 // Creates array of rider IDs sorted by elapsed time

```



```

1585 // Creates array of rider IDs sorted by elapsed time
1586 int[] riderRanks = getRidersGeneralClassificationRank(raceId);
1587 for ( int i=0; i<riderRanks.length; i++ ) {
1588     sortedPoints[i] = riderRacePoints.get(getRiderByID(riderRanks[i]));
1589 }
1590 return sortedPoints;
1591 }
1592
1593 /**
1594  * This method returns a list of riders' points for the mountain segments,
sorted by the total elapsed time.
1595  * @param raceId ID of the race.
1596  * @return an array of riders' points in the mountain segments race.
1597  * @exception IDNotRecognisedException ID does not exist in the system.
1598  */
1599 @Override
1600 public int[] getRidersMountainPointsInRace(int raceId) throws
IDNotRecognisedException {
1601     //Check race exists
1602     if(getRaceByID(raceId).getRaceName() == null){
1603         throw new IDNotRecognisedException("Race with ID " + raceId + " doesnt
exist in the system");
1604     }
1605     //Get the race by its ID
1606     Race race = getRaceByID(raceId);
1607     //Get stages in a array by the race ID
1608     Stage[] stages = race.getStagesV2();
1609     //If no stages in race return empty array
1610     int[] leader = new int[0];
1611     if(stages.length == 0){
1612         return leader;
1613     }
1614     //Creates a new array list of riders
1615     ArrayList<Rider> riders = new ArrayList<>();
1616     //Setting all riders in race in the array using the stages
1617     for(Stage stageFindRider : stages){
1618         int[] tempHoldID = getRidersRankInStage(stageFindRider.getStageId());
1619         for(int tempIDs : tempHoldID){
1620             if(riders.contains(getRiderByID(tempIDs))){
1621                 break;
1622             } else {
1623                 riders.add(getRiderByID(tempIDs));
1624             }
1625         }
1626     }
1627
1628     Map<Rider, Integer> riderRacePoints = new HashMap<Rider, Integer>();
1629     for (Rider rider : riders) {
1630         // Riders are added to HashMap
1631         riderRacePoints.put(rider, 0);
1632         for (Stage stage : stages) {
1633             // retrieves an array of ranked rider IDs in the stages
1634             int[] ranks = getRidersRankInStage(stage.getStageId());
1635             // Finds the index of the current rider in the array
1636             int indexOfRider = -1;

```

```

1637         for (int i=0; i<ranks.length; i++) {
1638             if (ranks[i] == rider.getRiderID()) {
1639                 indexOfRider = i;
1640             }
1641         }
1642         if (indexOfRider != -1) {
1643             int[] pointsArr =
getRidersMountainPointsInStage(stage.getStageId());
1644             int points = pointsArr[indexOfRider];
1645
1646             // Adds stage points to existing race points
1647             riderRacePoints.replace(rider, riderRacePoints.get(rider) +
points);
1648         }
1649     }
1650 }
1651
1652 // Creates array to store points
1653 int[] sortedPoints = new int[riders.size()];
1654 // Creates array of rider IDs sorted by elapsed time
1655 int[] riderRanks = getRidersGeneralClassificationRank(raceId);
1656 for ( int i=0; i<riderRanks.length; i++ ) {
1657     sortedPoints[i] = riderRacePoints.get(getRiderByID(riderRanks[i]));
1658 }
1659 return sortedPoints;
1660 }
1661
1662 /**
1663  * This method returns a list of rider IDs from the race, sorted by the
elapsed time.
1664  * @param raceId ID of the race.
1665  * @return an array of riders' points in the segments race.
1666  * @exception IDNotRecognisedException ID does not exist in the system.
1667  */
1668 @Override
1669 public int[] getRidersGeneralClassificationRank(int raceId) throws
IDNotRecognisedException {
1670     //Check race exists
1671     if(getRaceByID(raceId).getRaceName() == null){
1672         throw new IDNotRecognisedException("Race with ID " + raceId + " doesnt
exist in the system");
1673     }
1674     //Get the race by its ID
1675     Race race = getRaceByID(raceId);
1676     //Get stages in a array by the race ID
1677     Stage[] stages = race.getStagesV2();
1678     //If no stages in race return empty array
1679     int[] leader = new int[0];
1680     if(stages.length == 0){
1681         return leader;
1682     }
1683     //Creates a new array list of riders
1684     ArrayList<Rider> riders = new ArrayList<>();
1685     //Setting all riders in race in the array using the stages
1686     for(Stage stageFindRider : stages){

```

```

1687         int[] tempHoldID = getRidersRankInStage(stageFindRider.getStageId());
1688         for(int tempIDs : tempHoldID){
1689             if(riders.contains(getRiderByID(tempIDs))){
1690                 break;
1691             } else {
1692                 riders.add(getRiderByID(tempIDs));
1693             }
1694         }
1695     }
1696     //Create a HashMap to calculate elapsed race time through elapsed stage
time
1697     HashMap<Integer, LocalTime> riderRaceElapsed = new HashMap<Integer,
LocalTime>();
1698     for (Rider rider : riders) {
1699         riderRaceElapsed.put(rider.getRiderID(), LocalTime.of(0, 0, 0));
1700         for (Stage stage : stages) {
1701             LocalTime tempTimes =
getRiderAdjustedElapsedTimeInStage(stage.getStageId(), rider.getRiderID());
1702             if (tempTimes != null) {
1703                 riderRaceElapsed.replace(rider.getRiderID(),
riderRaceElapsed.get(rider.getRiderID()).plusHours(tempTimes.getHour())
1704                 .plusMinutes(tempTimes.getMinute()).plusSeconds(tempTimes.getSecond()));
1705             }
1706         }
1707     }
1708     //Sort by elapsed race time
1709     Map<Integer, LocalTime> sortedRiders =
1710         riderRaceElapsed.entrySet().stream()
1711         .sorted(Entry.comparingByValue())
1712         .collect(Collectors.toMap(Entry::getKey, Entry::getValue, (e1,
e2) -> e1, LinkedHashMap::new));
1713     //Transform HashMap to an array
1714     int[] ridersSorted = new int[riders.size()];
1715     int index = 0;
1716     for(Map.Entry<Integer, LocalTime> mapEntry : sortedRiders.entrySet()){
1717         ridersSorted[index] = mapEntry.getKey();
1718         index++;
1719     }
1720     //Return array of race IDs
1721     return ridersSorted;
1722 }
1723
1724 /**
1725  * This method returns a list of rider IDs for the race, sorted by there
points.
1726  * @param raceId ID of the race.
1727  * @return an array of rider IDs in the segments race ordered by points.
1728  * @exception IDNotRecognisedException ID does not exist in the system.
1729  */
1730
1731 @Override
1732 public int[] getRidersPointClassificationRank(int raceId) throws
IDNotRecognisedException {
1733     //Check race exists

```

```
1734     if(getRaceByID(raceId).getRaceName() == null){
1735         throw new IDNotRecognisedException("Race with ID " + raceId + " doesnt
exist in the system");
1736     }
1737     //Get the race by its ID
1738     Race race = getRaceByID(raceId);
1739     //Get stages in a array by the race ID
1740     Stage[] stages = race.getStagesV2();
1741     //If no stages in race return empty array
1742     int[] leader = new int[0];
1743     if(stages.length == 0){
1744         return leader;
1745     }
1746     //Creates a new array list of riders
1747     ArrayList<Rider> riders = new ArrayList<>();
1748     //Setting all riders in race in the array using the stages
1749     for(Stage stageFindRider : stages){
1750         int[] tempHoldID = getRidersRankInStage(stageFindRider.getStageId());
1751         for(int tempIDs : tempHoldID){
1752             if(riders.contains(getRiderByID(tempIDs))){
1753                 break;
1754             } else {
1755                 riders.add(getRiderByID(tempIDs));
1756             }
1757         }
1758     }
1759
1760     Map<Rider, Integer> riderRacePoints = new HashMap<Rider, Integer>();
1761     for (Rider rider : riders) {
1762         // Riders are added to HashMap
1763         riderRacePoints.put(rider, 0);
1764         for (Stage stage : stages) {
1765             // retrieves an array of ranked rider IDs in the stages
1766             int[] ranks = getRidersRankInStage(stage.getStageId());
1767             // Finds the index of the current rider in the array
1768             int indexOfRider = -1;
1769             for (int i=0; i<ranks.length; i++) {
1770                 if (ranks[i] == rider.getRiderID()) {
1771                     indexOfRider = i;
1772                 }
1773             }
1774             if (indexOfRider != -1) {
1775                 int[] pointsArr = getRidersPointsInStage(stage.getStageId());
1776                 int points = pointsArr[indexOfRider];
1777
1778                 // Adds stage points to existing race points
1779                 riderRacePoints.replace(rider, riderRacePoints.get(rider) +
points);
1780             }
1781         }
1782     }
1783     Map<Rider, Integer> sortedRiders =
1784         riderRacePoints.entrySet().stream()
1785         .sorted(Entry.comparingByValue())
1786         .collect(Collectors.toMap(Entry::getKey, Entry::getValue, (e1,
```

```
e2) -> e1, LinkedHashMap::new));
1787 //Transform HashMap to an array
1788 int[] ridersSorted = new int[riders.size()];
1789 int index = 0;
1790 for(Map.Entry<Rider, Integer> mapEntry : sortedRiders.entrySet()){
1791     ridersSorted[index] = mapEntry.getKey().getRiderID();
1792     index++;
1793 }
1794 //Return array of race IDs
1795 return ridersSorted;
1796 }
1797
1798 /**
1799  * This method returns a list of rider IDs for the race, sorted by there
1800 points in the mountain segments.
1801  * @param raceId ID of the race.
1802  * @return an array of rider IDs in the segments race ordered by points in
1803 mountain segment.
1804  * @exception IDNotRecognisedException ID does not exist in the system.
1805  */
1806 @Override
1807 public int[] getRidersMountainPointClassificationRank(int raceId) throws
IDNotRecognisedException {
1808     //Check race exists
1809     if(getRaceByID(raceId).getRaceName() == null){
1810         throw new IDNotRecognisedException("Race with ID " + raceId + " doesnt
exist in the system");
1811     }
1812     //Get the race by its ID
1813     Race race = getRaceByID(raceId);
1814     //Get stages in a array by the race ID
1815     Stage[] stages = race.getStagesV2();
1816     //If no stages in race return empty array
1817     int[] leader = new int[0];
1818     if(stages.length == 0){
1819         return leader;
1820     }
1821     //Creates a new array list of riders
1822     ArrayList<Rider> riders= new ArrayList<>();
1823     //Setting all riders in race in the array using the stages
1824     for(Stage stageFindRider : stages){
1825         int[] tempHoldID = getRidersRankInStage(stageFindRider.getStageId());
1826         for(int tempIDs : tempHoldID){
1827             if(riders.contains(getRiderByID(tempIDs))){
1828                 break;
1829             } else {
1830                 riders.add(getRiderByID(tempIDs));
1831             }
1832         }
1833     }
1834     Map<Rider, Integer> riderRacePoints = new HashMap<Rider, Integer>();
1835     for (Rider rider : riders) {
1836         // Riders are added to HashMap
```

```
1837     riderRacePoints.put(rider, 0);
1838     for (Stage stage : stages) {
1839         // retrieves an array of ranked rider IDs in the stages
1840         int[] ranks = getRidersRankInStage(stage.getStageId());
1841         // Finds the index of the current rider in the array
1842         int indexOfRider = -1;
1843         for (int i=0; i<ranks.length; i++) {
1844             if (ranks[i] == rider.getRiderID()) {
1845                 indexOfRider = i;
1846             }
1847         }
1848         if (indexOfRider != -1) {
1849             int[] pointsArr =
1850             getRidersMountainPointsInStage(stage.getStageId());
1851             int points = pointsArr[indexOfRider];
1852             // Adds stage points to existing race points
1853             riderRacePoints.replace(rider, riderRacePoints.get(rider) +
1854             points);
1855         }
1856     }
1857
1858     Map<Rider, Integer> sortedRiders =
1859         riderRacePoints.entrySet().stream()
1860         .sorted(Entry.comparingByValue())
1861         .collect(Collectors.toMap(Entry::getKey, Entry::getValue, (e1,
1862         e2) -> e1, LinkedHashMap::new));
1863     //Transform HashMap to an array
1864     int[] ridersSorted = new int[riders.size()];
1865     int index = 0;
1866     for(Map.Entry<Rider, Integer> mapEntry : sortedRiders.entrySet()){
1867         ridersSorted[index] = mapEntry.getKey().getRiderID();
1868         index++;
1869     }
1870     //Return array of race IDs
1871     return ridersSorted;
1872 }
1873 }
1874 }
```

```

1 package cycling;
2 import java.io.Serializable;
3 import java.util.ArrayList;
4 /**
5  * CyclingPortal is a minimally compiling, but non-functioning implementor
6  * of the CyclingPortalInterface interface.
7  *
8  * @author Taariq Fadhill
9  * @author Kaloyan Gaydarov
10 * @version 4.20
11 * @since 14/02/2022
12 *
13 */
14
15 public class Race implements Serializable{
16     //-----Private Initial Varriables-----
17     //-----//
18     // Initializes number of races to 0.
19     private static int numberOfRaces = 0;
20     // Initializes raceID.
21     private int raceID;
22     // Initializes raceName.
23     private String raceName;
24     // Initializes raceDesc.
25     private String raceDesc;
26     // Initializes number of stages to 0.
27     private static int numberOfStages = 0;
28     // Initializes array of objects of stages.
29     private ArrayList<Stage> stages;
30
31     //-----Setter methods-----
32     //-----//
33     /**
34     * Sets the race name according to the raceName and raceDesc objects.
35     * @param raceName name of the race.
36     *
37     * Sets the race description according to the raceName and raceDesc
38     objects.
39     * @param raceDesc description of the race.
40     *
41     * Resets number of races, setting to 0.
42     * Resets number of stages in a race, setting to 0.
43     * Adds a stage from the objects stage.
44     * @param stage where object stages are stored.
45     */
46     public void setRaceName(String raceName){this.raceName= raceName;}
47     public void setraceDesc(String raceDesc){this.raceDesc= raceDesc;}
48     public static void resetNumberOfRaces() {numberOfRaces = 0;}
49     public static void resetNumberOfStages() {numberOfStages = 0;}
50     public void addStage(Stage stage) {
51         stages.add(stage);
52         numberOfStages++;
53     }
54 }

```

```

53 //-----Getter methods-----
54 -----//
55 /**
56  * Gets the race ID of the object.
57  * @return the race ID.
58  * Gets the race name of the object.
59  * @return the race name.
60  * Gets the race description.
61  * @return the race description.
62  * Gets the array list of object: stages.
63  * @return array list of object: stages.
64  * Gets the number of stages.
65  * @return the number of stages in the object.
66  *
67  * Gets the details of the race.
68  * @param raceId the ID of the race.
69  * @param raceName the name of the race.
70  * @param raceDesc the description of the race.
71  * @param numberOfStages the number of stages in the race.
72  * @param totalLength the total length of the race in km.
73 */
74 public int getRaceID() {return raceID;}
75 public String getRaceName(){return raceName;}
76 public String getRaceDesc(){return raceDesc;}
77 public ArrayList<Stage> getStages() {return stages;}
78 public Stage[] getStagesV2() {
79     Stage[] stageArray = new Stage[stages.size()];
80     stageArray = stages.toArray(stageArray);
81     return stageArray;
82 }
83 public int getNumberOfStages() {return numberOfStages;}
84 double getTotalLength() {
85     double totalLength = 0;
86     double length;
87     for (Stage stage : stages) {
88         length = stage.getStageLength();
89         assert (length >= 0);
90         totalLength += length;
91     }
92     return totalLength;
93 }
94 public String getRaceDetails() {
95     double totalLength = getTotalLength();
96     return "ID: "+raceID+" | Name: "+raceName+" | Description:
97 "+raceDesc+" | Number of Stages: "+numberOfStages+" | Total Length:
98 "+totalLength;
99 }
100 //-----Constructor methods-----
101 -----//
102 /**
103  * Creates Object 'Race' with initialized values of 'Null', 'Null' and '0'
104  * respectively, and an empty array.
105 */

```



```
102  ~/  
103  public Race(){  
104      this.raceName = "Null";  
105      this.raceDesc = "Null";  
106      this.raceID = 0;  
107      this.stages = new ArrayList<>();  
108  }  
109  
110  /**  
111  * Creates Object 'Race' with initialized values.  
112  * @param raceName the name of a given race.  
113  * @param raceDesc the description of a given race.  
114  */  
115  public Race(String raceName, String raceDesc){  
116      this.raceName = raceName;  
117      this.raceDesc = raceDesc;  
118      this.raceID = ++numberOfRaces;  
119      this.stages = new ArrayList<>();  
120  }  
121  
122  //-----Remover methods-----  
-----//  
123  /**  
124  * Creates method for removing a stage.  
125  * @param stage object 'stage' that will be removed.  
126  * @exception IDNotRecognisedException The ID of the stage does not  
127  exist.  
128  */  
129  public void removeStage(Stage stage) throws IDNotRecognisedException {  
130      if (!stages.contains(stage)) {  
131          throw new IDNotRecognisedException("stage does not exist in race  
with Id '"+raceID+"'");  
132      }  
133      stages.remove(stage);  
134      numberOfStages--;  
135  }  
136 }  
137  
138
```

```

1 package cycling;
2 import java.io.Serializable;
3 import java.time.LocalDateTime;
4 /**
5  * CyclingPortal is a minimally compiling, but non-functioning implementor
6  * of the CyclingPortalInterface interface.
7  *
8  * @author Kaloyan Gaydarov
9  * @author Taariq Fadhill
10 * @version 4.20
11 * @since 14/02/2022
12 *
13 */
14
15 public class Results implements Serializable{
16     //-----Private Initial Varriables-----
17     -----//
18     //Gets the number of results for the stage, initializing at 0.
19     private static int numberOfResults = 0;
20     //initializes resultID.
21     private int resultID;
22     //initializes stage.
23     private Stage stage;
24     //initializes rider.
25     private Rider rider;
26     //initializes array of local times.
27     private LocalDateTime[] times;
28
29     //-----Setter methods-----
30     -----//
31     /**
32      * Sets the results in a stage.
33      * @param stage object stage.
34      * Sets the results to a rider.
35      * @param rider object rider.
36      * Sets the result time based on local time array.
37      * @param times collected from local time array.
38      * Resets number of results in a stage, setting to 0.
39      */
40
41     public void setResultStage(Stage stage) {this.stage = stage;}
42     public void setResultRider(Rider rider) {this.rider = rider;}
43     public void setResultTime(LocalDateTime[] times) {this.times = times;}
44     public static void resetNumberOfResults() {numberOfResults = 0;}
45
46     //-----Getter methods-----
47     -----//
48     /**
49      * Gets the results ID of the object.
50      * @return the result ID.
51      * Gets the stage object of where the given result is.
52      * @return the stage object.
53      * Gets the rider based on their result in the stage.

```

```
52     * Gets the rider based on their result in the stage.
53     * @return the rider object.
54     * Gets the times of a given result.
55     * @return the result times as an array.
56     */
57
58
59     public int getResultsId() {return resultID;}
60     public Stage getResultStage() {return stage;}
61     public Rider getResultRider() {return rider;}
62     public LocalTime[] getResultTimes() {return times;}
63
64
65     //-----Constructor methods-----
66     -----//
67     /**
68      * Creates object 'Results' with initialized values.
69      * @param stage the stage the results will be stored in.
70      * @param rider the rider associated with the result.
71      * @param times the time of the given result.
72      */
73     public Results(Stage stage, Rider rider , LocalTime... times) {
74         this.stage = stage;
75         this.rider = rider;
76         this.times = times;
77         this.resultID = ++numberOfResults;
78     }
79     /**
80      * Creates empty object 'Results' and sets the result ID to 0.
81      */
82     public Results(){
83         this.stage = new Stage();
84         this.resultID = 0;
85     }
86 }
87
```

```

1 package cycling;
2
3 import java.io.Serializable;
4 import java.util.ArrayList;
5 /**
6  * CyclingPortal is a minimally compiling, but non-functioning implementor
7  * of the CyclingPortalInterface interface.
8  *
9  * @author Kaloyan Gaydarov
10 * @author Taariq Fadhill
11 * @version 4.20
12 * @since 14/02/2022
13 *
14 */
15
16 public class Rider implements Serializable{
17     //-----Private Initial Varriables-----
18     //-----//
19     // Initializes the number of riders to 0.
20     private static int numberOfRiders = 0;
21     // Initializes teamID.
22     private int teamID;
23     // Initializes riderName.
24     private String riderName;
25     // Initializes yearOfBirth.
26     private int yearOfBirth;
27     // Initializes riderID.
28     private int riderID;
29     // Initializes points.
30     private int points;
31     // Initializes array of objects of results.
32     private ArrayList<Results> resultsArray = new ArrayList<Results>();
33
34     //-----Setter methods-----
35     //-----//
36     /**
37      * Sets the rider name.
38      * @param riderName name of the rider.
39      *
40      * Sets the team ID.
41      * @param teamID ID of the team the rider is in.
42      *
43      * Sets the year of birth.
44      * @param yearOfBirth year that the rider was born, must be 2010 <
45      yearOfBirth < 1900.
46      *
47      * Resets the number of riders in a team, setting to 0 .
48      *
49      * Sets points for the rider.
50      * @param points points that will be stored in the rider object.
51      *
52      * Adds points to the rider object.
53      * @param points points that will be stored in the rider object.
54      *
55      * Adds the result of the rider to the results array.

```

```

52      // Adds the result of the rider to the results array.
53      * @param results result of the rider that will be stored in the results
array.
54      */
55      public void setRiderName(String riderName){this.riderName= riderName;}
56      public void setTeamID(int teamID){this.teamID= teamID;}
57      public void setYearOfBirth(int yearOfBirth){this.yearOfBirth=
yearOfBirth; }
58      public static void resetNumberOfRiders() {numberOfRiders = 0;}
59      public void setRiderPoints(int points){this.points = points;}
60      public void addRiderPoints(int points){this.points = this.points +
points;}
61      public void addRiderResult(Results results){resultsArray.add(results);}
62
63      //-----Getter methods-----
-----//
64      /**
65       * Gets the team ID of the object.
66       * @return the team ID.
67       * Gets the name of the rider.
68       * @return name of the rider.
69       * Gets the year of birth of the rider.
70       * @return year that the rider was born, must be 2010 < yearOfBirth >
1900.
71       * Gets the rider ID of the object.
72       * @return the rider ID.
73       * Gets the points for the given rider.
74       * @return points assigned to the specific rider.
75       * Gets the array list of object: results.
76       * @return array list of object: results.
77       *
78       *
79       * Gets the details of the rider.
80       * @param riderID the ID of the rider.
81       * @param riderName the name of the rider.
82       * @param teamID the ID of the team the rider is assigned to.
83       * @param yearOfBirth year that the rider was born, must be 2010 <
yearOfBirth > 1900.
84       * @param points points the rider as acquired.
85       */
86      public int getTeamID() {return teamID;}
87      public String getRiderName() {return riderName;}
88      public int getYearOfBirth() {return yearOfBirth;}
89      public int getRiderID() {return riderID;}
90      public int getRiderPoints() {return points;}
91      public Results[] getRiderResults() {
92          Results[] resultArr = new Results[resultsArray.size()];
93          resultArr = resultsArray.toArray(resultArr);
94          return resultArr;
95      };
96      public String getRiderDetails() {
97          return "Rider ID: "+riderID+" | Name: "+riderName+" | Team ID:
"+teamID+" | Year Of Birth: "+yearOfBirth+" | Points: "+points;
98      }
99
100      //-----Constructor methods-----

```

```

100 // -----//
101 /**
102  * Creates object 'Rider' with initialized values of 0, 'Null', 0, 0 and
103  * 0 respectively.
104  */
105 public Rider() {
106     teamID = 0;
107     riderName = "Null";
108     yearOfBirth = 0;
109     riderID = 0;
110     points = 0;
111 }
112 /**
113  * Creates object 'Rider' with initialized values.
114  * @param teamID ID of the team the rider is assigned to.
115  * @param riderName name of the rider.
116  * @param yearOfBirth year that the rider was born, must be 2010 <
117  * yearOfBirth > 1900.
118  * @param riderID ID of the rider.
119  * @param points points acquired by the rider.
120  */
121 public Rider(int teamID, String riderName , int yearOfBirth) {
122     this.teamID = teamID;
123     this.riderName = riderName;
124     this.yearOfBirth = yearOfBirth;
125     this.riderID = ++numberOfRiders;
126     this.points = 0;
127 }
128 //-----Remover methods-----
129 //-----//
130 /**
131  * Creates method for removing a rider.
132  * @param result object result that will be removed.
133  * @throws IDNotRecognisedException The ID of the rider does not exist.
134  */
135 public void removeResults(Results result) throws IDNotRecognisedException
136 {
137     resultsArray.remove(result);
138 }
139 }
140

```

```

1 package cycling;
2
3 import java.io.Serializable;
4 /**
5  * CyclingPortal is a minimally compiling, but non-functioning implementor
6  * of the CyclingPortalInterface interface.
7  *
8  * @author Kaloyan Gaydarov
9  * @author Taariq Fadhill
10 * @version 4.20
11 * @since 14/02/2022
12 *
13 */
14
15 public class Segment implements Serializable{
16     //-----Private Initial Varriables-----
17     //-----//
18     // Initializes the number of segments to 0.
19     private static int numberOfSegments = 0;
20     // Initializes segmentID.
21     private int segmentID;
22     // Initializes location.
23     private double location;
24     // Initializes segment types.
25     private SegmentType type;
26     // Initializes average gradient.
27     private double averageGradient;
28     // Initializes length.
29     private double length;
30     // Initializes stage.
31     private Stage stage;
32
33     //-----Setter methods-----
34     //-----//
35     /**
36      * Sets the segment location.
37      * @param location where the segments will be.
38      *
39      * Sets the average gradient of a given segment.
40      * @param averageGradient the average gradient of the segment as a
41      * percentage.
42      *
43      * Sets the length of the segment.
44      * @param length length of the segment in km.
45      *
46      * Sets the stage the segment will be held.
47      * @param stage the type of stage.
48      *
49      * Resets the number of segments to 0.
50      * @param numberOfSegments the number of segments in a given stage.
51      */
52     public void setSegementLocation(double location) {this.location =
53 location;}
54     public void setSegmentAverageGradient(double averageGradient)
55 {this.averageGradient = averageGradient;}

```

```

51     public void setSegmentLength(double length) {this.length = length;}
52     public void setSegmentStage(Stage stage) {this.stage = stage;}
53     public static void resetNumberOfSegments() {numberOfSegments = 0;}
54
55
56     //-----Getter methods-----
57     -----//
58     /**
59      * Gets the segment ID of the object.
60      * @return the segment ID.
61      * Gets the location of the Segment.
62      * @return location of the segment.
63      * Gets the type of segment.
64      * @return type of segment in a given stage.
65      * Gets the average gradient of a given segment.
66      * @return the average gradient as a percentage.
67      * Gets the length of a given segment.
68      * @return length of the segment in km.
69      * Gets the stage a given segment is in.
70      * @return stage of a given segment.
71      *
72      * Gets the details of the segment.
73      * @param segmentID the ID of the segment.
74      * @param location location of the segment.
75      * @param type type of segment.
76      * @param averageGradient the average gradient of the segment as a
77      percentage.
78      * @param length length of the segment in km.
79      * @param stage stage that the segment is in.
80      */
81     public int getSegmentID() {return segmentID;}
82     public double getSegmentLocation() {return location;}
83     public SegmentType getSegmentType() {return type;}
84     public double getSegmentAverageGradient() {return averageGradient;}
85     public double getSegmentLength() {return length;}
86     public Stage getSegmentStage() {return stage;}
87     public String getSegementDetails() {
88         return "ID: "+segmentID+" | Location: "+location+" | Type:: "+type+"
89 | Average Gradient: "+averageGradient+" | Length: "+length+" | Stage:
90 "+stage;
91     }
92
93     //-----Constructor methods-----
94     -----//
95     /**
96      * Creates object 'Segment' with initialized values of 0, 0 and 0
97      respectively.
98      */
99     public Segment() {
100         this.location = 0;
101         this.segmentID = 0;
102         this.averageGradient = 0.

```



```
99         this.averageGradient = 0,
100     }
101
102     /**
103      * Creates object 'Rider' with initialized values.
104      * @param location location of the segment.
105      * @param type type of segment.
106      * @param averageGradient average gradient of the segment as a
percentage.
107      * @param length length of the segment in km.
108      */
109     public Segment(double location, SegmentType type, double averageGradient,
double length) {
110         this.location = location;
111         this.type = type;
112         this.averageGradient = averageGradient;
113         this.length = length;
114         this.segmentID = ++numberOfSegments;
115     }
116
117     Segment(double location, SegmentType type) {
118         this.location = location;
119         this.type = type;
120         this.segmentID = ++numberOfSegments;
121     }
122
123 }
```

```

1 package cycling;
2 import java.io.Serializable;
3 import java.time.LocalDateTime;
4 import java.util.ArrayList;
5 /**
6  * CyclingPortal is a minimally compiling, but non-functioning implementor
7  * of the CyclingPortalInterface interface.
8  *
9  * @author Taariq Fadhill
10 * @author Kaloyan Gaydarov
11 * @version 4.20
12 * @since 14/02/2022
13 *
14 */
15
16 public class Stage implements Serializable{
17     //-----Private Initial Varriables-----
18     //-----//
19     // Initializes number of stages to 0.
20     private static int numberOfStages = 0;
21     // Initializes stageID.
22     private int stageID;
23     //Initializes stageName.
24     private String stageName;
25     // Initializes stageDesc.
26     private String stageDesc;
27     // Initializes length.
28     private double length;
29     // Initializes type of stage.
30     private StageType type;
31     // Creates array list 'Segment'.
32     private ArrayList<Segment> segmentsArray = new ArrayList<Segment>();
33     // Creates array list 'Results'
34     private ArrayList<Results> resultsArray = new ArrayList<Results>();
35     // Initializes number of segments to 0.
36     private static int numberOfSegments = 0;
37     // Initializes start time.
38     private LocalDateTime startTime;
39     // Initializes prepared object as 'true'.
40     private boolean prepared= true;
41
42     //-----Prepare methods-----
43     //-----//
44     /**
45      * Prepares function isPrepared()
46      * @return state of is/isn't prepared
47      */
48     public boolean isPrepared() {return prepared;}
49     public void prepare() {prepared = true;}
50
51     //-----Setter methods-----
52     //-----//

```

```

53  /**
54  * Sets the start time of the stage.
55  * @param startTime start time of the stage.
56  * Sets the name of the stage.
57  * @param stageName name of the stage.
58  * Sets the description of the stage.
59  * @param stageDesc description of the stage.
60  * Sets the length of the stage.
61  * @param length of the stage.
62  * Sets the type of stage.
63  * @param type of stage
64  *
65  * Resets the number of stages to 0.
66  * Resets the number of segments in a given stage to 0.
67  */
68  public void setStartTime(LocalDateTime startTime){this.startTime =
startTime;}
69  public void setStageName(String stageName){this.stageName = stageName;}
70  public void setStageDesc(String stageDesc){this.stageDesc = stageDesc;}
71  public void setStageLength(int length){this.length = length;}
72  public void setType(StageType type){this.type = type;}
73  public static void resetNumberOfStages() {numberOfStages = 0;}
74  public static void resetNumberOfSegments() {numberOfSegments = 0;}
75
76  //-----Getter methods-----
77  -----//
78  /**
79  * Gets the stage ID of the object.
80  * @return the stage ID.
81  * Gets the name of the stage.
82  * @return the name of the stage.
83  * Gets the description of the stage.
84  * @return description of the stage.
85  * Gets the length of the stage.
86  * @return the length of the stage in km.
87  * Gets the type of stage.
88  * @return type of stage.
89  * Gets the stage results from the array list 'Results'.
90  * @return results in the stage as an array.
91  * Gets the start time in the stage.
92  * @return the start time of the races in that stage.
93  * Gets the segments which are in that stage.
94  * @return segments as an array.
95  *
96  *
97  * Gets the details of the stage.
98  * @param stageID the ID of the stage.
99  * @param stageName the name of the stage.
100  * @param stageDesc the description of the stage.
101  * @param numberOfSegments the number of segments in the stage.
102  * @param stageLength the length of the stage in km.
103  */
104  public int getStageId() {return stageID;}
105  public String getStageName() {return stageName;}

```

```

105 public String getStageName() {return stageName;}
106 public String getStageDescription() {return stageDesc;}
107 public double getStageLength() {return length;}
108 public StageType getStageType() {return type;}
109 public ArrayList<Results> getStageResults() {return resultsArray;}
110 public LocalDateTime getStartTime(){return startTime;}
111 public Segment[] getStageSegments() {
112     Segment[] segmentArray = new Segment[segmentsArray.size()];
113     segmentArray = segmentsArray.toArray(segmentArray);
114     return segmentArray;
115 }
116 public String getStageDetails() {
117     double stageLength = getStageLength();
118     return "ID: "+stageID+" | Name: "+stageName+" | Description:
"+stageDesc+" | Number of Segements: "+numberOfSegments+" | Stage Length:
"+stageLength;
119 }
120
121
122 //-----Constructor methods-----
-----//
123 /**
124  * Creates object 'Stage' with initialized values of 'Null', 'Null', 0
and 0 respectively.
125  */
126 public Stage() {
127     this.stageName = "Null";
128     this.stageDesc = "Null";
129     this.length = 0;
130     this.stageID = 0;
131 }
132
133 /**
134  * Creates object 'Stage' with initialized values.
135  * @param stageName the name of the stage.
136  * @param stageDesc the description of a given stage.
137  * @param length length of the stage in km.
138  * @param startTime start time of races in the stage.
139  * @param type type of stage.
140  */
141 public Stage(String stageName, String stageDesc, double
length,LocalDateTime startTime, StageType type) {
142     this.stageName = stageName;
143     this.stageDesc = stageDesc;
144     this.length = length;
145     this.startTime = startTime;
146     this.type = type;
147     this.stageID = ++numberOfStages;
148 }
149
150
151 //-----Remover & Adder methods-----
-----//
152 /**
153  * Creates method for removing a segment from a given stage.
154  * @param segment object 'segment' that will be removed

```

```
154      * @param segment object 'segment' that will be removed.
155      *
156      * Creates method for adding a segment to a given stage.
157      * @param segment object 'segment' that will be added.
158      *
159      * Creates method for adding results to a stage.
160      * @param result object 'result' that will be added.
161      *
162      * Creates method for removing results from a stage.
163      * @param result object 'result' that will be removed.
164      */
165      public void removeStageSegment(Segment segment) {
166          segmentsArray.remove(segment);
167          --numberOfSegments;
168      }
169      public void addStageSegment(Segment segment) {
170          segmentsArray.add(segment);
171          ++numberOfSegments;
172      }
173      public void addStageResults(Results result) {
174          resultsArray.add(result);
175      }
176      public void removeResults(Results result) throws IDNotRecognisedException
177      {
178          resultsArray.remove(result);
179      }
180  }
```

```
1 package cycling;
2 import java.io.Serializable;
3 import java.util.ArrayList;
4 /**
5  * CyclingPortal is a minimally compiling, but non-functioning implementor
6  * of the CyclingPortalInterface interface.
7  *
8  * @author Taariq Fadhill
9  * @author Kaloyan Gaydarov
10 * @version 4.20
11 * @since 14/02/2022
12 *
13 */
14
15 public class Team implements Serializable{
16     //-----Private Initial Varriables-----
17     //-----//
18     // Initializes number of teams to 0.
19     private static int numberOfTeams = 0;
20     // Initializes teamID.
21     private int teamID;
22     // Initializes teamName.
23     private String teamName;
24     // Initializes teamDesc.
25     private String teamDesc;
26     // Initializes points.
27     private int points;
28     // Creates array list 'Rider'.
29     private ArrayList<Rider> teamRiders = new ArrayList<Rider>();
30     // Initializes number of riders to 0.
31     private static int numberOfRiders = 0;
32
33     //-----Setter methods-----
34     //-----//
35     /**
36      * Sets the name of the team.
37      * @param teamName name of the team.
38      * Sets the description of the team.
39      * @param teamDesc description of the team.
40      * Sets the total points for the team.
41      * @param points
42      * Adds points to the total points for the team.
43      * @param points
44      * Resets the number of teams to 0.
45      * Resets the number of riders in a team to 0.
46      */
47     public void setTeamName(String teamName){this.teamName = teamName;}
48     public void setTeamDesc(String teamDesc){this.teamDesc = teamDesc;}
49     public void setTeamPoints(int points){this.points = points;}
50     public void addTeamPoints(int points){this.points = this.points +
51     points;}
52     public static void resetNumberOfTeams() {numberOfTeams = 0;}
53     public static void resetNumberOfRiders() {numberOfRiders= 0;}
54
55     //-----Getter methods-----
```

```

53  -----//
54  /**
55   * Gets the team ID of the object.
56   * @return the team ID.
57   * Gets the name of the team.
58   * @return the name of the team.
59   * Gets the description of the team.
60   * @return description of the team.
61   * Gets the total points for a team.
62   * @return total points.
63   * Gets the number of riders in a team.
64   * @return number of riders.
65   * Gets riders which are in a team.
66   * @return riders in a given team as an array.
67   * Gets riders based on their riderID.
68   * @return riders as an object.
69   *
70   *
71   * Gets the details of the team.
72   * @param teamID the ID of the team.
73   * @param teamName the name of the team.
74   * @param teamDesc description of the team.
75   * @param points total points for the team.
76   */
77  public int getTeamID() {return teamID;}
78  public String getTeamName() {return teamName;}
79  public String getTeamDesc() {return teamDesc;}
80  public int getTeamPoints(){return points;}
81  public int getNumberOfTeamRiders(){ return numberOfRiders;}
82  public Rider[] getRiders() {
83      Rider[] riderArray = new Rider[teamRiders.size()];
84      riderArray = teamRiders.toArray(riderArray);
85      return riderArray;
86  }
87  public Rider getRider(int riderID) {
88      for (Rider rider : teamRiders) {
89          if (rider.getRiderID() == riderID) {
90              return rider;
91          }
92      }
93      return new Rider();
94  }
95  public String getTeamDetails() {
96      return "ID: "+teamID+" | Name: "+teamName+" | Description:
97      "+teamDesc+" | Points: "+points;
98  }
99  //-----Constructor methods-----
100  -----//
101  /**
102   * Creates object 'Team' with initialized values of 'Null', 'Null', 0 and
103   * 0 respectively.
104   */
105  public Team() {
106      this.teamName = "Null";

```

```
103     this.teamName = null ;
104     this.teamDesc = "Null";
105     this.teamID = 0;
106     this.points = 0;
107 }
108
109 /**
110  * Creates object 'Team' with initialized values.
111  * @param teamName the name of the team.
112  * @param teamDesc description of a given team.
113  * @param teamID ID of the team.
114  * @param points points of the team.
115  */
116 public Team(String teamName, String teamDesc) {
117     this.teamName = teamName;
118     this.teamDesc = teamDesc;
119     this.teamID = ++numberOfTeams;
120     this.points = 0;
121 }
122
123 //-----Remover & Adder methods-----
124 -----//
125 /**
126  * Creates method for removing a rider from a given team.
127  * @param rider object 'rider' that will be removed.
128  *
129  * Creates method for adding a rider to a given team.
130  * @param rider object 'rider' that will be added.
131  */
132 public void removeRider(Rider rider) {
133     teamRiders.remove(rider);
134     --numberOfRiders;
135 }
136 public void addRider(Rider newRider) {
137     teamRiders.add(newRider);
138     numberOfRiders++;
139 }
```


