```java
package cycling;

import java.io.FileInputStream;

@SuppressWarnings("unchecked")

/**
 * CyclingPortal - A compiling and fucntioning implementor
 * of the CycllingPortalInterface Interface.
 */
public class CyclingPortal implements CyclingPortalInterface {
    private ArrayList<Team> teamObjects;
    private ArrayList<Race> raceObjects;
    private CounterStates counterStates;

    public CyclingPortal() {
        teamObjects = new ArrayList<>();
        raceObjects = new ArrayList<>();
        counterStates = new CounterStates();
    }

    @Override
    public int[] getRaceIds() {
        int raceCount = raceObjects.size();
        int[] idArray = new int[raceCount];
        // storing raceIds in an array which is returned later
        for (int i = 0; i < raceCount; i++) {
            idArray[i] = raceObjects.get(i).getId();
        }
        return idArray;
    }

    @Override
    public int createRace(String name, String description) throws IllegalNameException,
  InvalidNameException {
        if (doesRaceNameExist(name)) {
            throw new IllegalNameException("Race name already exists.");
        } else if (name == null || name == "" || name.length() > 30 || name.matches(".*\\s
  +.*")) {
            throw new InvalidNameException(
                    "Invalid name, must not be null, empty, not longer than 30 characters and
  not contain white spaces.");
        }
        // creating a race, adding it to the platform and then incrementing the counter
        int raceid = counterStates.getRaceCounter();
        Race race = new Race(raceid, name, description);
        registerRace(race);
        counterStates.incrementRaceCounter();
        return race.getId();
    }

    @Override
    public String viewRaceDetails(int raceId) throws IDNotRecognisedException {
        /*
         * checking if the race Id exists then returning
         * a string with the race details
         */
```

```
62          if (doesRaceIdExist(raceId)) {
63              int raceCount = raceObjects.size();
64              for (int i = 0; i < raceCount; i++) {
65                  if (raceObjects.get(i).getId() == raceId) {
66                      Race selectedRace = raceObjects.get(i);
67                      String returnString = "Race ID: " + selectedRace.getId()
68                              + "\nRace Name: " + selectedRace.getName()
69                              + "\nRace Description: " + selectedRace.getDescription()
70                              + "\nNumber of Stages: " + selectedRace.getNumberOfStages()
71                              + "\nTotal Length: " + selectedRace.getTotalLength();
72                      return returnString;
73                  }
74              }
75              return "";
76          } else {
77              throw new IDNotRecognisedException("The ID does not match to any race in the
    system.");
78          }
79      }
80
81      @Override
82      public void removeRaceById(int raceId) throws IDNotRecognisedException {
83          /*
84           * checking if the race Id exists then removing the
85           * race from the system
86           */
87          if (doesRaceIdExist(raceId)) {
88              int raceCount = raceObjects.size();
89              for (int i = 0; i < raceCount; i++) {
90                  if (raceObjects.get(i).getId() == raceId) {
91                      raceObjects.remove(i);
92                      break;
93                  }
94              }
95          } else {
96              throw new IDNotRecognisedException("The ID does not match to any race in the
    system.");
97          }
98      }
99
100     @Override
101     public int getNumberOfStages(int raceId) throws IDNotRecognisedException {
102         if (doesRaceIdExist(raceId)) {
103             int raceCount = raceObjects.size();
104             for (int i = 0; i < raceCount; i++) {
105                 if (raceObjects.get(i).getId() == raceId) {
106                     return raceObjects.get(i).getNumberOfStages();
107                 }
108             }
109             return 0;
110         } else {
111             throw new IDNotRecognisedException("The ID does not match to any race in the
    system.");
112         }
113     }
114
115     @Override
```

```
116     public int addStageToRace(int raceId, String stageName, String description, double
    length, LocalDateTime startTime,
117             StageType type)
118             throws IDNotRecognisedException, IllegalNameException, InvalidNameException,
    InvalidLengthException {
119         if (!doesRaceIdExist(raceId)) {
120             throw new IDNotRecognisedException("The ID does not match to any race in the
    system.");
121         } else if (doesStageNameExist(raceId, stageName)) {
122             throw new IllegalNameException("Stage name already exists in the system.");
123         } else if (stageName == null || stageName == "" || stageName.length() > 30) {
124             throw new InvalidNameException(
125                     "Invalid stage name, must not be null, empty and not longer than 30
    characters.");
126         } else if (length < 5) {
127             throw new InvalidLengthException("Stage length can not be less than 5km.");
128         }
129         int stageId = counterStates.getStageCounter();
130         Stage stage = new Stage(stageId, raceId, stageName, description, length, startTime,
    type);
131         addStageToRaceObject(raceId, stage);
132         counterStates.incrementStageCounter();
133         return stage.getId();
134     }
135 
136     @Override
137     public int[] getRaceStages(int raceId) throws IDNotRecognisedException {
138         if (!doesRaceIdExist(raceId)) {
139             throw new IDNotRecognisedException("Race id does not match to any race id in the
    system.");
140         }
141         ArrayList<Stage> stageObjects = getStages(raceId);
142         int stageCount = stageObjects.size();
143         int[] idArray = new int[stageCount];
144         for (int i = 0; i < stageCount; i++) {
145             idArray[i] = stageObjects.get(i).getId();
146         }
147         return idArray;
148     }
149 
150     @Override
151     public double getStageLength(int stageId) throws IDNotRecognisedException {
152         if (!doesStageIdExist(stageId)) {
153             throw new IDNotRecognisedException("Stage id does not match to any stage id in
    the system.");
154         }
155         int[] raceIds = getRaceIds();
156         int raceCount = raceIds.length;
157         for (int i = 0; i < raceCount; i++) {
158             ArrayList<Stage> stageObjects = getStages(raceIds[i]);
159             int stageCount = stageObjects.size();
160             for (int j = 0; j < stageCount; j++) {
161                 if (stageObjects.get(j).getId() == stageId) {
162                     return stageObjects.get(j).getLength();
163                 }
164             }
165         }
```

```
166         return 1.0;
167     }
168 
169     @Override
170     public void removeStageById(int stageId) throws IDNotRecognisedException {
171         if (!doesStageIdExist(stageId)) {
172             throw new IDNotRecognisedException("Stage id does not match to any stage id in
    the system.");
173         }
174         int[] raceIds = getRaceIds();
175         int raceCount = raceIds.length;
176         for (int i = 0; i < raceCount; i++) {
177             ArrayList<Stage> stageObjects = getStages(raceIds[i]);
178             int stageCount = stageObjects.size();
179             for (int j = 0; j < stageCount; j++) {
180                 if (stageObjects.get(j).getId() == stageId) {
181                     int raceIndex = i;
182                     int stageIndex = j;
183                     removeStage(raceIndex, stageIndex);
184                     break;
185                 }
186             }
187         }
188     }
189 
190     @Override
191     public int addCategorizedClimbToStage(int stageId, Double location, SegmentType type,
    Double averageGradient,
192             Double length) throws IDNotRecognisedException, InvalidLocationException,
    InvalidStageStateException,
193             InvalidStageTypeException {
194         if (!doesStageIdExist(stageId)) {
195             throw new IDNotRecognisedException("Stage ID does not match to any stage ID in
    the system.");
196         } else if (location > getStageLength(stageId) || location < 0) {
197             throw new InvalidLocationException("The segment location is out of bounds of the
    stage length.");
198         } else if (getStageStateByStageId(stageId).equals("waiting for results")) {
199             throw new InvalidStageStateException("The stage is currently `waiting for
    results`.");
200         } else if (getStageType(stageId).equals(StageType.TT)) {
201             throw new InvalidStageTypeException("Time trial stages cannot contain any
    segments.");
202         }
203         int segmentId = counterStates.getSegmentCounter();
204         Segment segment = new Segment(segmentId, stageId, location, type, averageGradient,
    length);
205         addSegmentToStage(stageId, segment);
206         counterStates.incrementSegmentCounter();
207         return segment.getId();
208     }
209 
210     @Override
211     public int addIntermediateSprintToStage(int stageId, double location) throws
    IDNotRecognisedException,
212             InvalidLocationException, InvalidStageStateException, InvalidStageTypeException {
213         if (!doesStageIdExist(stageId)) {
```

```
214             throw new IDNotRecognisedException("Stage ID does not match to any stage ID in
    the system.");
215         } else if (location > getStageLength(stageId) || location < 0) {
216             throw new InvalidLocationException("The segment location is out of bounds of the
    stage length.");
217         } else if (getStageStateByStageId(stageId).equals("waiting for results")) {
218             throw new InvalidStageStateException("The stage is currently `waiting for
    results`.");
219         } else if (getStageType(stageId).equals(StageType.TT)) {
220             throw new InvalidStageTypeException("Time trial stages cannot contain any
    segments.");
221         }
222         int segmentId = counterStates.getSegmentCounter();
223         Segment segment = new Segment(segmentId, stageId, location);
224         addSegmentToStage(stageId, segment);
225         counterStates.incrementSegmentCounter();
226         return segment.getId();
227     }
228
229     @Override
230     public void removeSegment(int segmentId) throws IDNotRecognisedException,
    InvalidStageStateException {
231         if (!doesSegmentIdExist(segmentId)) {
232             throw new IDNotRecognisedException("Segment ID was not found in the system.");
233         } else if (getStageStateBySegmentId(segmentId).equals("waiting for results")) {
234             throw new InvalidStageStateException("The stage is currently `waiting for
    results`.");
235         }
236         int[] raceIds = getRaceIds();
237         int raceCount = raceIds.length;
238         for (int i = 0; i < raceCount; i++) {
239             ArrayList<Stage> stageObjects = getStages(raceIds[i]);
240             int stageCount = stageObjects.size();
241             for (int j = 0; j < stageCount; j++) {
242                 int[] segmentIds = stageObjects.get(j).getSegmentIds();
243                 for (int k = 0; k < segmentIds.length; k++) {
244                     if (segmentIds[k] == segmentId) {
245                         stageObjects.get(j).removeSegment(k);
246                     }
247                 }
248             }
249         }
250     }
251
252     @Override
253     public void concludeStagePreparation(int stageId) throws IDNotRecognisedException,
    InvalidStageStateException {
254         if (!doesStageIdExist(stageId)) {
255             throw new IDNotRecognisedException("Stage ID was not found in the system.");
256         } else if (getStageStateByStageId(stageId).equals("waiting for results")) {
257             throw new InvalidStageStateException("The stage is currently `waiting for
    results`.");
258         }
259         int[] raceIds = getRaceIds();
260         int raceCount = raceIds.length;
261         for (int i = 0; i < raceCount; i++) {
262             ArrayList<Stage> stageObjects = getStages(raceIds[i]);
```

```
263             int stageCount = stageObjects.size();
264             for (int j = 0; j < stageCount; j++) {
265                 if (stageObjects.get(j).getId() == stageId) {
266                     stageObjects.get(j).conclude();
267                 }
268             }
269         }
270     }
271
272     @Override
273     public int[] getStageSegments(int stageId) throws IDNotRecognisedException {
274         if (!doesStageIdExist(stageId)) {
275             throw new IDNotRecognisedException("Stage ID was not found in the system.");
276         }
277         int[] raceIds = getRaceIds();
278         int raceCount = raceIds.length;
279         for (int i = 0; i < raceCount; i++) {
280             ArrayList<Stage> stageObjects = getStages(raceIds[i]);
281             int stageCount = stageObjects.size();
282             for (int j = 0; j < stageCount; j++) {
283                 if (stageObjects.get(j).getId() == stageId) {
284                     return stageObjects.get(j).getSegmentIds();
285                 }
286             }
287         }
288         return new int[] {};
289     }
290
291     @Override
292     public int createTeam(String name, String description) throws IllegalNameException,
    InvalidNameException {
293         if (doesTeamNameExist(name)) {
294             throw new IllegalNameException("Team name already exists.");
295         } else if (name == null || name == "" || name.length() > 30) {
296             throw new InvalidNameException("Invalid name, must not be null, empty and not
    longer than 30 characters.");
297         } else {
298             int teamId = counterStates.getTeamCounter();
299             Team team = new Team(teamId, name, description);
300             registerTeam(team);
301             counterStates.incrementTeamCounter();
302             return team.getId();
303         }
304     }
305
306     @Override
307     public void removeTeam(int teamId) throws IDNotRecognisedException {
308         if (doesTeamIdExist(teamId)) {
309             removeTeamById(teamId);
310         } else {
311             throw new IDNotRecognisedException("The ID does not match to any team in the
    system.");
312         }
313     }
314
315     @Override
316     public int[] getTeams() {
```

```
317         int teamCount = teamObjects.size();
318         int[] idArray = new int[teamCount];
319         for (int i = 0; i < teamCount; i++) {
320             idArray[i] = teamObjects.get(i).getId();
321         }
322         return idArray;
323     }
324 
325     @Override
326     public int[] getTeamRiders(int teamId) throws IDNotRecognisedException {
327         if (!doesTeamIdExist(teamId)) {
328             throw new IDNotRecognisedException("Team could not be found.");
329         }
330         int teamCount = teamObjects.size();
331         for (int i = 0; i < teamCount; i++) {
332             if (teamObjects.get(i).getId() == teamId) {
333                 Team foundTeam = teamObjects.get(i);
334 
335                 int riderCount = foundTeam.getRiderCount();
336                 int[] idArray = new int[riderCount];
337                 for (int j = 0; j < riderCount; j++) {
338                     idArray[j] = foundTeam.getRiderIdAtIndex(j);
339                 }
340                 return idArray;
341             }
342         }
343         return new int[] {};
344     }
345 
346     @Override
347     public int createRider(int teamID, String name, int yearOfBirth)
348             throws IDNotRecognisedException, IllegalArgumentException {
349         if (!doesTeamIdExist(teamID)) {
350             throw new IDNotRecognisedException("Team could not be found.");
351         } else if (name == null || name == "" || yearOfBirth < 1900) {
352             throw new IllegalArgumentException(
353                     "Invalid name must not be null nor empty. Or invalid year of birth, must
    be greater than or equal to 1900.");
354         } else {
355             int riderId = counterStates.getRiderCounter();
356             Rider rider = new Rider(riderId, teamID, name, yearOfBirth);
357             addRiderToTeam(teamID, rider);
358             counterStates.incrementRiderCounter();
359             return rider.getId();
360         }
361     }
362 
363     @Override
364     public void removeRider(int riderId) throws IDNotRecognisedException {
365         if (!doesRiderIdExist(riderId)) {
366             throw new IDNotRecognisedException("Rider ID does not match to any rider in the
    system.");
367         }
368         removeAllResultsForRider(riderId);
369         ArrayList<Team> teamObjects = getTeamObjects();
370         int teamCount = teamObjects.size();
371         for (int i = 0; i < teamCount; i++) {
```

```
372             if (teamObjects.get(i).doesRiderExist(riderId)) {
373                 teamObjects.get(i).removeRider(riderId);
374             }
375         }
376     }
377
378     @Override
379     public void registerRiderResultsInStage(int stageId, int riderId, LocalTime...
    checkpoints)
380             throws IDNotRecognisedException, DuplicatedResultException,
    InvalidCheckpointsException,
381             InvalidStageStateException {
382         if (!doesRiderIdExist(riderId)) {
383             throw new IDNotRecognisedException("Rider ID does not match to any rider in the
    system.");
384         } else if (!doesStageIdExist(stageId)) {
385             throw new IDNotRecognisedException("Stage ID does not match to any stage in the
    system.");
386         } else if (doesRiderHaveResult(stageId, riderId)) {
387             throw new DuplicatedResultException("Rider already has a result in this stage.");
388         } else if (getSegmentsCount(stageId) + 2 != checkpoints.length) {
389             throw new InvalidCheckpointsException(
390                     "The length of the checkpoints is invalid, must be equal to the number of
    segment + 2 (start and finish).");
391         } else if (getStageStateByStageId(stageId).equals("waiting for results")) {
392             throw new InvalidStageStateException("The stage is currently `waiting for
    results`.");
393         } else {
394             Result result = new Result(riderId, checkpoints);
395             linkRiderResultsInStage(stageId, result);
396         }
397     }
398
399     @Override
400     public LocalTime[] getRiderResultsInStage(int stageId, int riderId) throws
    IDNotRecognisedException {
401         if (!doesRiderIdExist(riderId)) {
402             throw new IDNotRecognisedException("Rider ID does not match to any rider in the
    system.");
403         } else if (!doesStageIdExist(stageId)) {
404             throw new IDNotRecognisedException("Stage ID does not match to any stage in the
    system.");
405         }
406         int[] raceIds = getRaceIds();
407         int raceCount = raceIds.length;
408         for (int i = 0; i < raceCount; i++) {
409             ArrayList<Stage> stageObjects = getStages(raceIds[i]);
410             int stageCount = stageObjects.size();
411             for (int j = 0; j < stageCount; j++) {
412                 if (stageObjects.get(j).getId() == stageId) {
413                     return stageObjects.get(j).getRiderResultsInStage(riderId);
414                 }
415             }
416         }
417         return new LocalTime[] {};
418     }
419
```

```
420     @Override
421     public LocalTime getRiderAdjustedElapsedTimeInStage(int stageId, int riderId) throws
    IDNotRecognisedException {
422         if (!doesRiderIdExist(riderId)) {
423             throw new IDNotRecognisedException("Rider ID does not match to any rider in the
    system.");
424         } else if (!doesStageIdExist(stageId)) {
425             throw new IDNotRecognisedException("Stage ID does not match to any stage in the
    system.");
426         }
427         int[] raceIds = getRaceIds();
428         int raceCount = raceIds.length;
429         for (int i = 0; i < raceCount; i++) {
430             ArrayList<Stage> stageObjects = getStages(raceIds[i]);
431             int stageCount = stageObjects.size();
432             for (int j = 0; j < stageCount; j++) {
433                 if (stageObjects.get(j).getId() == stageId) {
434                     return stageObjects.get(j).getRiderAdjustedElapsedTimeInStage(riderId);
435                 }
436             }
437         }
438         return null;
439     }
440
441     @Override
442     public void deleteRiderResultsInStage(int stageId, int riderId) throws
    IDNotRecognisedException {
443         if (!doesRiderIdExist(riderId)) {
444             throw new IDNotRecognisedException("Rider ID does not match to any rider in the
    system.");
445         } else if (!doesStageIdExist(stageId)) {
446             throw new IDNotRecognisedException("Stage ID does not match to any stage in the
    system.");
447         }
448         int[] raceIds = getRaceIds();
449         int raceCount = raceIds.length;
450         for (int i = 0; i < raceCount; i++) {
451             ArrayList<Stage> stageObjects = getStages(raceIds[i]);
452             int stageCount = stageObjects.size();
453             for (int j = 0; j < stageCount; j++) {
454                 if (stageObjects.get(j).getId() == stageId) {
455                     stageObjects.get(j).removeAllRiderResults(riderId);
456                 }
457             }
458         }
459     }
460
461     @Override
462     public int[] getRidersRankInStage(int stageId) throws IDNotRecognisedException {
463         if (!doesStageIdExist(stageId)) {
464             throw new IDNotRecognisedException("Stage ID does not match to any stage in the
    system.");
465         }
466         if (getStageResultsCount(stageId) > 0) {
467             int[] raceIds = getRaceIds();
468             int raceCount = raceIds.length;
469             for (int i = 0; i < raceCount; i++) {
```

```
470                 ArrayList<Stage> stageObjects = getStages(raceIds[i]);
471                 int stageCount = stageObjects.size();
472                 for (int j = 0; j < stageCount; j++) {
473                     if (stageObjects.get(j).getId() == stageId) {
474                         return stageObjects.get(j).getRidersRankInStage();
475                     }
476                 }
477             }
478             return new int[] {};
479         }
480         return new int[] {};
481     }
482 
483     @Override
484     public LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId) throws
    IDNotRecognisedException {
485         if (!doesStageIdExist(stageId)) {
486             throw new IDNotRecognisedException("Stage ID does not match to any stage in the
    system.");
487         }
488         int[] raceIds = getRaceIds();
489         int raceCount = raceIds.length;
490         for (int i = 0; i < raceCount; i++) {
491             ArrayList<Stage> stageObjects = getStages(raceIds[i]);
492             int stageCount = stageObjects.size();
493             for (int j = 0; j < stageCount; j++) {
494                 if (stageObjects.get(j).getId() == stageId) {
495                     return stageObjects.get(j).getRankedAdjustedElapsedTimesInStage();
496                 }
497             }
498         }
499         return new LocalTime[] {};
500     }
501 
502     @Override
503     public int[] getRidersPointsInStage(int stageId) throws IDNotRecognisedException {
504         if (!doesStageIdExist(stageId)) {
505             throw new IDNotRecognisedException("Stage ID does not match to any stage in the
    system.");
506         }
507         if (getStageResultsCount(stageId) > 0) {
508             int[] raceIds = getRaceIds();
509             int raceCount = raceIds.length;
510             for (int i = 0; i < raceCount; i++) {
511                 ArrayList<Stage> stageObjects = getStages(raceIds[i]);
512                 int stageCount = stageObjects.size();
513                 for (int j = 0; j < stageCount; j++) {
514                     if (stageObjects.get(j).getId() == stageId) {
515                         return stageObjects.get(j).getRidersPointsInStage();
516                     }
517                 }
518             }
519             return new int[] {};
520         }
521         return new int[] {};
522     }
523 
```

```
524     @Override
525     public int[] getRidersMountainPointsInStage(int stageId) throws IDNotRecognisedException
    {
526         if (!doesStageIdExist(stageId)) {
527             throw new IDNotRecognisedException("Stage ID does not match to any stage in the
    system.");
528         }
529         if (getStageResultsCount(stageId) > 0) {
530             int[] raceIds = getRaceIds();
531             int raceCount = raceIds.length;
532             for (int i = 0; i < raceCount; i++) {
533                 ArrayList<Stage> stageObjects = getStages(raceIds[i]);
534                 int stageCount = stageObjects.size();
535                 for (int j = 0; j < stageCount; j++) {
536                     if (stageObjects.get(j).getId() == stageId) {
537                         return stageObjects.get(j).getRidersMountainPointsInStage();
538                     }
539                 }
540             }
541             return new int[] {};
542         }
543         return new int[] {};
544     }
545
546     @Override
547     public void eraseCyclingPortal() {
548         raceObjects = new ArrayList<Race>();
549         teamObjects = new ArrayList<Team>();
550         counterStates.resetAllCounts();
551     }
552
553     @Override
554     public void saveCyclingPortal(String filename) throws IOException {
555         FileOutputStream file = new FileOutputStream(filename);
556         ObjectOutputStream out = new ObjectOutputStream(file);
557         out.writeObject(teamObjects);
558         out.writeObject(raceObjects);
559         out.writeObject(counterStates);
560         out.close();
561         file.close();
562     }
563
564     @Override
565     public void loadCyclingPortal(String filename) throws IOException, ClassNotFoundException
    {
566         FileInputStream file = new FileInputStream(filename);
567         ObjectInputStream in = new ObjectInputStream(file);
568         teamObjects = (ArrayList<Team>) in.readObject();
569         raceObjects = (ArrayList<Race>) in.readObject();
570         counterStates = (CounterStates) in.readObject();
571         in.close();
572         file.close();
573     }
574
575     @Override
576     public void removeRaceByName(String name) throws NameNotRecognisedException {
577         if (doesRaceNameExist(name)) {
```

```
578             int raceCount = raceObjects.size();
579             for (int i = 0; i < raceCount; i++) {
580                 if (raceObjects.get(i).getName().equals(name)) {
581                     raceObjects.remove(i);
582                     break;
583                 }
584             }
585         } else {
586             throw new NameNotRecognisedException("The name does not match to any race in the system.");
587         }
588     }
589 
590     @Override
591     public LocalTime[] getGeneralClassificationTimesInRace(int raceId) throws IDNotRecognisedException {
592         if (!doesRaceIdExist(raceId)) {
593             throw new IDNotRecognisedException("Race ID was not found in the system.");
594         }
595         int raceCount = raceObjects.size();
596         for (int i = 0; i < raceCount; i++) {
597             if (raceObjects.get(i).getId() == raceId) {
598                 return raceObjects.get(i).getGeneralClassificationTimesInRace();
599             }
600         }
601         return new LocalTime[] {};
602     }
603 
604     @Override
605     public int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException {
606         if (!doesRaceIdExist(raceId)) {
607             throw new IDNotRecognisedException("Race ID was not found in the system.");
608         }
609         // Finds the relevant race object and gets the rider points in the race.
610         int raceCount = raceObjects.size();
611         for (int i = 0; i < raceCount; i++) {
612             if (raceObjects.get(i).getId() == raceId) {
613                 return raceObjects.get(i).getRidersPointsInRace();
614             }
615         }
616         return new int[] {};
617     }
618 
619     @Override
620     public int[] getRidersMountainPointsInRace(int raceId) throws IDNotRecognisedException {
621         if (!doesRaceIdExist(raceId)) {
622             throw new IDNotRecognisedException("Race ID was not found in the system.");
623         }
624         int raceCount = raceObjects.size();
625         for (int i = 0; i < raceCount; i++) {
626             if (raceObjects.get(i).getId() == raceId) {
627                 return raceObjects.get(i).getRidersMountainPointsInRace();
628             }
629         }
630         return new int[] {};
631     }
632 
```

```
633     @Override
634     public int[] getRidersGeneralClassificationRank(int raceId) throws
    IDNotRecognisedException {
635         if (!doesRaceIdExist(raceId)) {
636             throw new IDNotRecognisedException("Race ID was not found in the system.");
637         }
638         int raceCount = raceObjects.size();
639         for (int i = 0; i < raceCount; i++) {
640             if (raceObjects.get(i).getId() == raceId) {
641                 return raceObjects.get(i).getRidersGeneralClassificationRank();
642             }
643         }
644         return new int[] {};
645     }
646 
647     @Override
648     public int[] getRidersPointClassificationRank(int raceId) throws IDNotRecognisedException
    {
649         if (!doesRaceIdExist(raceId)) {
650             throw new IDNotRecognisedException("Race ID was not found in the system.");
651         }
652         int raceCount = raceObjects.size();
653         for (int i = 0; i < raceCount; i++) {
654             if (raceObjects.get(i).getId() == raceId) {
655                 return raceObjects.get(i).getRidersPointClassificationRank();
656             }
657         }
658         return new int[] {};
659     }
660 
661     @Override
662     public int[] getRidersMountainPointClassificationRank(int raceId) throws
    IDNotRecognisedException {
663         if (!doesRaceIdExist(raceId)) {
664             throw new IDNotRecognisedException("Race ID was not found in the system.");
665         }
666         int raceCount = raceObjects.size();
667         for (int i = 0; i < raceCount; i++) {
668             if (raceObjects.get(i).getId() == raceId) {
669                 return raceObjects.get(i).getRidersMountainPointClassificationRank();
670             }
671         }
672         return new int[] {};
673     }
674 
675     // Segment Handler Functions
676     /**
677      * adds a segment to the stage
678      *
679      * @param stageId the Id of the stage
680      * @param segment the segment object
681      */
682     public void addSegmentToStage(int stageId, Segment segment) {
683         int[] raceIds = getRaceIds();
684         int raceCount = raceIds.length;
685         for (int i = 0; i < raceCount; i++) {
686             ArrayList<Stage> stageObjects = getStages(raceIds[i]);
```

```
687             int stageCount = stageObjects.size();
688             for (int j = 0; j < stageCount; j++) {
689                 if (stageObjects.get(j).getId() == stageId) {
690                     stageObjects.get(j).addSegment(segment);
691                 }
692             }
693         }
694     }
695 
696     /**
697      * gets the current state of the stage by the segment Id
698      *
699      * @param segmentId the Id of the segment
700      * @return whether the stage is waiting for results or not
701      */
702     public String getStageStateBySegmentId(int segmentId) {
703         int[] raceIds = getRaceIds();
704         int raceCount = raceIds.length;
705         for (int i = 0; i < raceCount; i++) {
706             ArrayList<Stage> stageObjects = getStages(raceIds[i]);
707             int stageCount = stageObjects.size();
708             for (int j = 0; j < stageCount; j++) {
709                 int[] segmentIds = stageObjects.get(j).getSegmentIds();
710                 for (int k = 0; k < segmentIds.length; k++) {
711                     if (segmentIds[k] == segmentId) {
712                         return stageObjects.get(j).getState();
713                     }
714                 }
715             }
716         }
717         return "";
718     }
719 
720     /**
721      * checks if the segment Id exsits
722      *
723      * @param segmentId the Id of the segment
724      * @return true if the Id exists, false if it doesn't
725      */
726     public boolean doesSegmentIdExist(int segmentId) {
727         int[] raceIds = getRaceIds();
728         int raceCount = raceIds.length;
729         for (int i = 0; i < raceCount; i++) {
730             ArrayList<Stage> stageObjects = getStages(raceIds[i]);
731             int stageCount = stageObjects.size();
732             for (int j = 0; j < stageCount; j++) {
733                 int[] segmentIds = stageObjects.get(j).getSegmentIds();
734                 for (int k = 0; k < segmentIds.length; k++) {
735                     if (segmentIds[k] == segmentId) {
736                         return true;
737                     }
738                 }
739             }
740         }
741         return false;
742     }
743 
```

```java
    // Rider Handler Functions
    /**
     * checks if the rider Id exists
     *
     * @param riderId the Id of he rider
     * @return true if the riderId exists, false if it doesn't
     */
    public boolean doesRiderIdExist(int riderId) {
        ArrayList<Team> teamObjects = getTeamObjects();
        int teamCount = teamObjects.size();
        for (int i = 0; i < teamCount; i++) {
            if (teamObjects.get(i).doesRiderExist(riderId)) {
                return true;
            }
        }
        return false;
    }

    // Team Handler Functions
    /**
     * registers a Team
     *
     * @param team team object
     */
    public void registerTeam(Team team) {
        teamObjects.add(team);
    }

    /**
     * adds a rider to the team
     *
     * @param teamId the Id of the team
     * @param rider the rider Object to be added
     */
    public void addRiderToTeam(int teamId, Rider rider) {
        int teamCount = teamObjects.size();
        for (int i = 0; i < teamCount; i++) {
            if (teamObjects.get(i).getId() == teamId) {
                teamObjects.get(i).addRider(rider);
            }
        }
    }

    /**
     * checks if a team name exists
     *
     * @param nameSearch Team's name
     * @return true if the team's name exists, false if it doesn't
     */
    public boolean doesTeamNameExist(String nameSearch) {
        int teamCount = teamObjects.size();
        for (int i = 0; i < teamCount; i++) {
            if (teamObjects.get(i).getName().equals(nameSearch)) {
                return true;
            }
        }
        return false;
```

```
801     }
802
803     /**
804      * checks if the team Id exists
805      *
806      * @param idSearch the Id of the team
807      * @return True if the team Id exists, false if it doesn't
808      */
809     public boolean doesTeamIdExist(int idSearch) {
810         int teamCount = teamObjects.size();
811         for (int i = 0; i < teamCount; i++) {
812             if (teamObjects.get(i).getId() == idSearch) {
813                 return true;
814             }
815         }
816         return false;
817     }
818
819     /**
820      * removes a team from the system and
821      * the riders associated with the team
822      * alongside their results
823      *
824      * @param teamId the id of the team
825      */
826     public void removeTeamById(int teamId) {
827         int teamCount = teamObjects.size();
828         for (int i = 0; i < teamCount; i++) {
829             if (teamObjects.get(i).getId() == teamId) {
830                 Team teamTBD = teamObjects.get(i);
831                 int riderCount = teamTBD.getRiderCount();
832                 for (int j = 0; j < riderCount; j++) {
833                     int riderId = teamTBD.getRiderIdAtIndex(j);
834                     removeAllResultsForRider(riderId);
835                 }
836                 teamObjects.remove(i);
837                 break;
838             }
839         }
840     }
841
842     /**
843      * gets the teamObjects
844      *
845      * @return Arraylist of teamObjects
846      */
847     public ArrayList<Team> getTeamObjects() {
848         return teamObjects;
849     }
850
851     /**
852      * remove the rider's results from the system
853      *
854      * @param riderId the id of the rider
855      */
856     public void removeAllResultsForRider(int riderId) {
857         int[] raceIds = getRaceIds();
```

```
858         int raceCount = raceIds.length;
859         for (int i = 0; i < raceCount; i++) {
860             ArrayList<Stage> stageObjects = getStages(raceIds[i]);
861             int stageCount = stageObjects.size();
862             for (int j = 0; j < stageCount; j++) {
863                 stageObjects.get(j).removeAllRiderResults(riderId);
864             }
865         }
866     }
867 
868     // Stage Handler Functions
869     /**
870      * checks if the stage name exists
871      *
872      * @param raceId the Id of the race
873      * @param name   the name of the stage
874      * @return true if the stage name exists, false if it doesn't
875      */
876     public boolean doesStageNameExist(int raceId, String name) {
877         ArrayList<Stage> stageObjects = getStages(raceId);
878         int stageCount = stageObjects.size();
879         for (int i = 0; i < stageCount; i++) {
880             if (stageObjects.get(i).getName().equals(name)) {
881                 return true;
882             }
883         }
884         return false;
885     }
886 
887     /**
888      * gets the current state of the stage by the stage Id
889      *
890      * @param stageId the Id of the stage
891      * @return whether the stage is waiting for results or not
892      */
893     public String getStageStateByStageId(int stageId) {
894         int[] raceIds = getRaceIds();
895         int raceCount = raceIds.length;
896         for (int i = 0; i < raceCount; i++) {
897             ArrayList<Stage> stageObjects = getStages(raceIds[i]);
898             int stageCount = stageObjects.size();
899             for (int j = 0; j < stageCount; j++) {
900                 if (stageObjects.get(j).getId() == stageId) {
901                     return stageObjects.get(j).getState();
902                 }
903             }
904         }
905         return "";
906     }
907 
908     /**
909      * gets the type of the stage
910      *
911      * @param stageId the Id of the stage
912      * @return the type of the stage
913      */
914     public StageType getStageType(int stageId) {
```

```
915         int[] raceIds = getRaceIds();
916         int raceCount = raceIds.length;
917         for (int i = 0; i < raceCount; i++) {
918             ArrayList<Stage> stageObjects = getStages(raceIds[i]);
919             int stageCount = stageObjects.size();
920             for (int j = 0; j < stageCount; j++) {
921                 if (stageObjects.get(j).getId() == stageId) {
922                     return stageObjects.get(j).getType();
923                 }
924             }
925         }
926         return null;
927     }
928 
929     /**
930      * checks if the Id of the stage exists
931      *
932      * @param stageId the Id of the stage
933      * @return true if the stage Id exists, false if it doesn't
934      */
935     public boolean doesStageIdExist(int stageId) {
936         int[] raceIds = getRaceIds();
937         int raceCount = raceIds.length;
938         for (int i = 0; i < raceCount; i++) {
939             ArrayList<Stage> stageObjects = getStages(raceIds[i]);
940             int stageCount = stageObjects.size();
941             for (int j = 0; j < stageCount; j++) {
942                 if (stageObjects.get(j).getId() == stageId) {
943                     return true;
944                 }
945             }
946         }
947         return false;
948     }
949 
950     /**
951      * checks if the rider has a result
952      *
953      * @param stageId the Id of the Stage
954      * @param riderId the Id of the Rider
955      * @return True if the rider has a result, false if he doesn't
956      */
957     public boolean doesRiderHaveResult(int stageId, int riderId) {
958         int[] raceIds = getRaceIds();
959         int raceCount = raceIds.length;
960         for (int i = 0; i < raceCount; i++) {
961             ArrayList<Stage> stageObjects = getStages(raceIds[i]);
962             int stageCount = stageObjects.size();
963             for (int j = 0; j < stageCount; j++) {
964                 if (stageObjects.get(j).getId() == stageId) {
965                     return stageObjects.get(j).doesRiderHaveResult(riderId);
966                 }
967             }
968         }
969         return false;
970     }
971 
```

```
    /**
     * gets the number of segments for a specific stage
     *
     * @param stageId the Id of the stage
     * @return the number of segments in the stage
     */
    public int getSegmentsCount(int stageId) {
        int[] raceIds = getRaceIds();
        int raceCount = raceIds.length;
        for (int i = 0; i < raceCount; i++) {
            ArrayList<Stage> stageObjects = getStages(raceIds[i]);
            int stageCount = stageObjects.size();
            for (int j = 0; j < stageCount; j++) {
                if (stageObjects.get(j).getId() == stageId) {
                    return stageObjects.get(j).getSegmentsCount();
                }
            }
        }
        return 0;
    }

    /**
     * adds results to stages
     *
     * @param stageId stage's Id
     * @param result  result Object
     */
    public void linkRiderResultsInStage(int stageId, Result result) {
        int[] raceIds = getRaceIds();
        int raceCount = raceIds.length;
        for (int i = 0; i < raceCount; i++) {
            ArrayList<Stage> stageObjects = getStages(raceIds[i]);
            int stageCount = stageObjects.size();
            for (int j = 0; j < stageCount; j++) {
                if (stageObjects.get(j).getId() == stageId) {
                    stageObjects.get(j).addResult(result);
                }
            }
        }
    }

    /**
     * gets the number of results in a stage
     *
     * @param stageId the stage's Id
     * @return the number of results in the stage
     */
    public int getStageResultsCount(int stageId) {
        int[] raceIds = getRaceIds();
        int raceCount = raceIds.length;
        for (int i = 0; i < raceCount; i++) {
            ArrayList<Stage> stageObjects = getStages(raceIds[i]);
            int stageCount = stageObjects.size();
            for (int j = 0; j < stageCount; j++) {
                if (stageObjects.get(j).getId() == stageId) {
                    return stageObjects.get(j).getResultsCount();
                }
```

```
1029            }
1030        }
1031        return 0;
1032    }
1033
1034    // Race Handler Functions
1035    /**
1036     * registers a race
1037     *
1038     * @param race the race object
1039     */
1040    public void registerRace(Race race) {
1041        raceObjects.add(race);
1042    }
1043
1044    /**
1045     * checks if the race name exists
1046     *
1047     * @param nameSearch the name of the race
1048     * @return true if the race name exists, false if it doesn't
1049     */
1050    public boolean doesRaceNameExist(String nameSearch) {
1051        int raceCount = raceObjects.size();
1052        for (int i = 0; i < raceCount; i++) {
1053            if (raceObjects.get(i).getName().equals(nameSearch)) {
1054                return true;
1055            }
1056        }
1057        return false;
1058    }
1059
1060    /**
1061     * checks if the Race Id exsits
1062     *
1063     * @param idSearch the Race's Id
1064     * @return true if the race id exists, false if it doesn't
1065     */
1066    public boolean doesRaceIdExist(int idSearch) {
1067        int raceCount = raceObjects.size();
1068        for (int i = 0; i < raceCount; i++) {
1069            if (raceObjects.get(i).getId() == idSearch) {
1070                return true;
1071            }
1072        }
1073        return false;
1074    }
1075
1076    /**
1077     * adds a stage to a race
1078     *
1079     * @param raceId race's Id
1080     * @param stage  stage's Id
1081     */
1082    public void addStageToRaceObject(int raceId, Stage stage) {
1083        int raceCount = raceObjects.size();
1084        for (int i = 0; i < raceCount; i++) {
1085            if (raceObjects.get(i).getId() == raceId) {
```

```
1086                raceObjects.get(i).addStage(stage);
1087                break;
1088            }
1089        }
1090    }
1091
1092    /**
1093     * gets the stages in a race
1094     *
1095     * @param raceId race's Id
1096     * @return a list of the stages in the race
1097     */
1098    public ArrayList<Stage> getStages(int raceId) {
1099        int raceCount = raceObjects.size();
1100        for (int i = 0; i < raceCount; i++) {
1101            if (raceObjects.get(i).getId() == raceId) {
1102                return raceObjects.get(i).getStages();
1103            }
1104        }
1105        return new ArrayList<Stage>();
1106    }
1107
1108    /**
1109     * removes a stage from a race
1110     *
1111     * @param raceIndex  race object's position
1112     * @param stageIndex stage object's position
1113     */
1114    public void removeStage(int raceIndex, int stageIndex) {
1115        raceObjects.get(raceIndex).removeStage(stageIndex);
1116    }
1117}
1118
```

```java
package cycling;

import java.io.Serializable;

/**
 * Race- A class which stores information about races
 * and handles some of the operations related to races.
 */
public class Race implements Serializable {
    private int id;
    private String name;
    private String description;
    private ArrayList<Stage> stageObjects = new ArrayList<Stage>();

    /**
     * Constructor for the Objects of Race class
     *
     * @param id          the Id of the race
     * @param name        the name of the race
     * @param description the description of the race
     */
    public Race(int id, String name, String description) {
        this.id = id;
        this.name = name;
        this.description = description;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public void addStage(Stage stage) {
        stageObjects.add(stage);
    }

    public int getNumberOfStages() {
        return stageObjects.size();
    }

    /**
     * calculates the total length of all stages in a race
```

```
63      *
64      * @return the total length in kms
65      */
66     public double getTotalLength() {
67         int stage_count = stageObjects.size();
68         double totalLength = 0;
69         for (int i = 0; i < stage_count; i++) {
70             totalLength += stageObjects.get(i).getLength();
71         }
72         return totalLength;
73     }
74 
75     public ArrayList<Stage> getStages() {
76         return stageObjects;
77     }
78 
79     /**
80      * Removes a stage from the race
81      *
82      * @param stageIndex position of the stage in the arraylist
83      */
84     public void removeStage(int stageIndex) {
85         stageObjects.remove(stageIndex);
86     }
87 
88     public int[] getRidersGeneralClassificationRank() {
89         /*
90          * A loop to check if the stage has results
91          * if not an empty array is returned
92          */
93         int stage_count = stageObjects.size();
94         for (int j = 0; j < stage_count; j++) {
95             if (stageObjects.get(j).getResultsCount() == 0) {
96                 return new int[] {};
97             }
98         }
99 
100        // A sample of the results of a stage are stored in an arraylist
101        ArrayList<Result> sampleResultObjects = stageObjects.get(0).getResultObjects();
102        int result_count = sampleResultObjects.size();
103 
104        /*
105         * A Hashmap to store the riderId along with its corresponding
106         * score is created and is initialized by storing the riderIds
107         * from the array of the sample results
108         */
109        HashMap<Integer, LocalTime> riderHashMap = new HashMap<Integer, LocalTime>();
110        for (int i = 0; i < result_count; i++) {
111            riderHashMap.put(sampleResultObjects.get(i).getRiderId(), LocalTime.ofNanoOfDay
   (0));
112        }
113 
114        LocalTime currentAdjElapsedTime;
115        int rider_count = riderHashMap.size();
116        /*
117         * the key part of the hashmap, which consists of the riderIds
118         * is converted to an arrray which will store the ridersIds
```

```
119          */
120         int[] rider_ids = riderHashMap.keySet().stream().mapToInt(Number::intValue).toArray();
121 
122         /*
123          * A loop to check if the rider has taken part in all of the stages
124          * of a specific race, if not his riderId is removed from the Hashmap
125          * as it will provide an unfair advantage
126          */
127         for (int j = 0; j < stage_count; j++) {
128             for (int i = 0; i < rider_count; i++) {
129                 if (!stageObjects.get(j).doesRiderHaveResult(rider_ids[i])) {
130                     riderHashMap.remove(rider_ids[i]);
131                     rider_count = riderHashMap.size();
132                     rider_ids = riderHashMap.keySet().stream().mapToInt
    (Number::intValue).toArray();
133                 }
134             }
135         }
136 
137         for (int i = 0; i < rider_count; i++) {
138             /*
139              * a loop that loops for the number of stages in a race
140              * it gets current elapsed time of a rider by the function
141              * getRiderAdjustedElapsedTimeInStage and updates the Hashmap
142              * after adding it to the previous sum of elapsed times
143              */
144             for (int j = 0; j < stage_count; j++) {
145                 currentAdjElapsedTime = stageObjects.get(j).getRiderAdjustedElapsedTimeInStage
    (rider_ids[i]);
146                 LocalTime elapsedTimeSum = riderHashMap.get(rider_ids[i]);
147                 riderHashMap.replace(rider_ids[i],
148                         elapsedTimeSum.plus(currentAdjElapsedTime.toNanoOfDay(),
    ChronoUnit.NANOS));
149             }
150         }
151 
152         ArrayList<Result> resultsArray = new ArrayList<Result>();
153         /*
154          * A loop that creates a result object for each rider to store
155          * the riderID along with the result, then each object is
156          * stored in an Arraylist named resultsArray
157          */
158         for (int i = 0; i < rider_count; i++) {
159             Result result = new Result(rider_ids[i], riderHashMap.get(rider_ids[i]));
160             resultsArray.add(result);
161         }
162 
163         // the resultsArray is sorted by the elapsed time
164         Collections.sort(resultsArray);
165 
166         int[] finalIdsArray = new int[resultsArray.size()];
167         /*
168          * A loop that takes the riderId part of the Hashmap after it
169          * has been sorted by elapsed time to store it an Array named
170          * finalIds which is returned later
171          */
172         for (int i = 0; i < resultsArray.size(); i++) {
```

```
173             finalIdsArray[i] = resultsArray.get(i).getRiderId();
174         }
175         return finalIdsArray;
176     }
177 
178     public LocalTime[] getGeneralClassificationTimesInRace() {
179 
180         int stage_count = stageObjects.size();
181         /*
182          * A loop to check that the each Stage of the Race has results
183          * otherwise an empty array of localtime is returned
184          */
185         for (int j = 0; j < stage_count; j++) {
186             if (stageObjects.get(j).getResultsCount() == 0) {
187                 return new LocalTime[] {};
188             }
189         }
190 
191         ArrayList<Result> sampleResultObjects = stageObjects.get(0).getResultObjects();
192         int result_count = sampleResultObjects.size();
193 
194         HashMap<Integer, LocalTime> riderHashMap = new HashMap<Integer, LocalTime>();
195         // the riderHashmap is initialized by using the riderIds of a specific stage
196         for (int i = 0; i < result_count; i++) {
197             riderHashMap.put(sampleResultObjects.get(i).getRiderId(), LocalTime.ofNanoOfDay
    (0));
198         }
199 
200         LocalTime currentAdjElapsedTime;
201         int rider_count = riderHashMap.size();
202         int[] rider_ids = riderHashMap.keySet().stream().mapToInt(Number::intValue).toArray();
203 
204         /*
205          * A loop to check if the rider has participated in all stages of
206          * the race, if not the riderId is removed from the Hashmap
207          */
208         for (int j = 0; j < stage_count; j++) {
209             for (int i = 0; i < rider_count; i++) {
210                 if (!stageObjects.get(j).doesRiderHaveResult(rider_ids[i])) {
211                     riderHashMap.remove(rider_ids[i]);
212                     rider_count = riderHashMap.size();
213                     rider_ids = riderHashMap.keySet().stream().mapToInt
    (Number::intValue).toArray();
214                 }
215             }
216         }
217 
218         /*
219          * A loop to update the Elapsed time part of the Hashmap after
220          * going through all the results of the rider in all stages
221          */
222         for (int i = 0; i < rider_count; i++) {
223             for (int j = 0; j < stage_count; j++) {
224                 currentAdjElapsedTime = stageObjects.get(j).getRiderAdjustedElapsedTimeInStage
    (rider_ids[i]);
225                 LocalTime elapsedTimeSum = riderHashMap.get(rider_ids[i]);
226                 riderHashMap.replace(rider_ids[i],
```

```
227                         elapsedTimeSum.plus(currentAdjElapsedTime.toNanoOfDay(),
   ChronoUnit.NANOS));
228             }
229         }
230 
231         ArrayList<Result> resultsArray = new ArrayList<Result>();
232         /*
233          * using a loop to make a result object for each rider along with the
234          * elapsed time which is stored in an array named resultsArray
235          */
236         for (int i = 0; i < rider_count; i++) {
237             Result result = new Result(rider_ids[i], riderHashMap.get(rider_ids[i]));
238             resultsArray.add(result);
239         }
240 
241         // resultsArray is sorted by Elapsed time
242         Collections.sort(resultsArray);
243 
244         LocalTime[] finalElapsedTimesArray = new LocalTime[resultsArray.size()];
245         /*
246          * after the result Array has been sorted, the elapsed time
247          * of each rider is stored in an array named finalElapsedTimesArray
248          * by the following loop
249          */
250         for (int i = 0; i < resultsArray.size(); i++) {
251             finalElapsedTimesArray[i] = resultsArray.get(i).getCheckPoint(0);
252         }
253 
254         return finalElapsedTimesArray;
255     }
256 
257     public int[] getRidersPointsInRace() {
258         /*
259          * a list of rider Ids sorted by the sum of elapsed time
260          * will be stored in an array named riderIds
261          */
262         int[] riderIds = getRidersGeneralClassificationRank();
263         // If there are no riderIds an empty array will be returned
264         if (riderIds.length == 0) {
265             return new int[] {};
266         }
267         // the riderHashmap is initialized
268         HashMap<Integer, Integer> riderHashMap = new HashMap<Integer, Integer>();
269         for (int i = 0; i < riderIds.length; i++) {
270             riderHashMap.put(riderIds[i], 0);
271         }
272         int stage_count = stageObjects.size();
273         /*
274          * a loop that loops for the number of stages in the race to make sure that
275          * the points of all stages are added up
276          */
277         for (int i = 0; i < stage_count; i++) {
278             /*
279              * two arrrays to store the riderIds and their corresponding points
280              * sorted by their elapsed time
281              */
282             int[] stageRiderIds = stageObjects.get(i).getRidersRankInStage();
```

```
283             int[] stageRiderPoints = stageObjects.get(i).getRidersPointsInStage();
284             // a loop which will update the hashmap with the points for each rider
285             for (int j = 0; j < stageRiderIds.length; j++) {
286                 if (riderHashMap.containsKey(stageRiderIds[j])) {
287                     int previousPoints = riderHashMap.get(stageRiderIds[j]);
288                     riderHashMap.replace(stageRiderIds[j], previousPoints + stageRiderPoints
    [j]);
289                 }
290             }
291         }
292         int[] finalPointsArray = new int[riderIds.length];
293         /*
294          * a loop to store the points part of the hashmap into an array
295          * named finalpointsArray
296          */
297         for (int i = 0; i < riderIds.length; i++) {
298             finalPointsArray[i] = riderHashMap.get(riderIds[i]);
299         }
300         return finalPointsArray;
301     }
302 
303     public int[] getRidersMountainPointsInRace() {
304         /*
305          * the function getRidersGenenralClassificationRank will return a list of
306          * riders' IDs sorted ascending by the sum of their adjusted elasped times
307          * which will be stored in an array named riderIds
308          */
309         int[] riderIds = getRidersGeneralClassificationRank();
310         // if the array has no rider Ids an empty array will be returned
311         if (riderIds.length == 0) {
312             return new int[] {};
313         }
314         // A hashmap with the riderIds and their respective point is initialized
315         HashMap<Integer, Integer> riderHashMap = new HashMap<Integer, Integer>();
316         for (int i = 0; i < riderIds.length; i++) {
317             riderHashMap.put(riderIds[i], 0);
318         }
319         int stage_count = stageObjects.size();
320 
321         for (int i = 0; i < stage_count; i++) {
322             /*
323              * the riderIds and their points are retrieved by calling the functions
324              * getRidersRankInstage and getRidersMountainPointsInStage and are
325              * stored in two separate arrays
326              */
327             int[] stageRiderIds = stageObjects.get(i).getRidersRankInStage();
328             int[] stageRiderPoints = stageObjects.get(i).getRidersMountainPointsInStage();
329             // the following loop for the number of riders in that stage
330             // and will update the hasmap with the points of each rider
331             for (int j = 0; j < stageRiderIds.length; j++) {
332                 if (riderHashMap.containsKey(stageRiderIds[j])) {
333                     int previousPoints = riderHashMap.get(stageRiderIds[j]);
334                     riderHashMap.replace(stageRiderIds[j], previousPoints + stageRiderPoints
    [j]);
335                 }
336             }
337         }
```

```
338
339         int[] finalPointsArray = new int[riderIds.length];
340         /*
341          * the values part of the hashmap which contains the points of each
342          * rider is called and its values are stored in an array
343          * named finalPointsArray which is returned later
344          */
345         for (int i = 0; i < riderIds.length; i++) {
346             finalPointsArray[i] = riderHashMap.get(riderIds[i]);
347         }
348         return finalPointsArray;
349     }
350
351     public int[] getRidersPointClassificationRank() {
352         /*
353          * the riderIds and their points are retrieved by calling the functions
354          * getRidersGeneralClassificationRank and getRidersPointsInRace and are
355          * stored in two separate arrays
356          */
357         int[] riderIds = getRidersGeneralClassificationRank();
358         int[] riderPoints = getRidersPointsInRace();
359
360         // the bubblesort function is callled to sort by the riderPoints
361         bubbleSort(riderPoints, riderIds);
362
363         return riderIds;
364     }
365
366     public int[] getRidersMountainPointClassificationRank() {
367         /*
368          * the riderIds and their points are retrieved by calling the functions
369          * getRidersGeneralClassificationRank and getRidersMountainPointsInRace and are
370          * stored in two separate arrays
371          */
372         int[] riderIds = getRidersGeneralClassificationRank();
373         int[] riderPoints = getRidersMountainPointsInRace();
374
375         // the bubblesort function is called to sort by the riderPoints
376         bubbleSort(riderPoints, riderIds);
377
378         return riderIds;
379     }
380
381     /**
382      * An algorithm for sorting by riderPoints
383      */
384     public void bubbleSort(int[] pointsArray, int[] idsArray) {
385         boolean sorted = false;
386         int temp;
387         while (!sorted) {
388             sorted = true;
389             for (int i = 0; i < pointsArray.length - 1; i++) {
390                 if (pointsArray[i] < pointsArray[i + 1]) {
391
392                     temp = pointsArray[i];
393                     pointsArray[i] = pointsArray[i + 1];
394                     pointsArray[i + 1] = temp;
```

```
395
396                     temp = idsArray[i];
397                     idsArray[i] = idsArray[i + 1];
398                     idsArray[i + 1] = temp;
399
400                     sorted = false;
401                 }
402             }
403         }
404     }
405
406 }
407
```

```java
package cycling;

import java.io.Serializable;

/**
 * Stage- A class which stores information about stages
 * and handles some of the operations related to stages.
 */
public class Stage implements Serializable {

    private int id;
    private double raceId;
    private String stageName;
    private String description;
    private double length;
    private LocalDateTime startTime;
    private StageType type;
    private String state = "not waiting for results";

    private ArrayList<Segment> segmentObjects = new ArrayList<Segment>();
    private ArrayList<Result> resultObjects = new ArrayList<Result>();

    /**
     * Constructor for the Objects of Stage Class
     *
     * @param stageId     the Id of the stage
     * @param raceId      the Id of the race
     * @param stageName   An identifier name for the stage.
     * @param description A descriptive text for the stage.
     * @param length      Stage length in kilometres.
     * @param startTime   The date and time in which the stage will be raced. It
     *                    cannot be null.
     * @param type        The type of the stage. This is used to determine the
     *                    amount of points given to the winner.
     */
    public Stage(int stageId, int raceId, String stageName, String description, double length,
  LocalDateTime startTime,
            StageType type) {
        this.id = stageId;
        this.setRaceId(raceId);
        this.stageName = stageName;
        this.setDescription(description);
        this.length = length;
        this.setStartTime(startTime);
        this.type = type;
    }

    public LocalDateTime getStartTime() {
        return startTime;
    }

    public void setStartTime(LocalDateTime startTime) {
        this.startTime = startTime;
    }

    public String getDescription() {
        return description;
```

```
 63     }
 64
 65     public void setDescription(String description) {
 66         this.description = description;
 67     }
 68
 69     public double getRaceId() {
 70         return raceId;
 71     }
 72
 73     public void setRaceId(double raceId) {
 74         this.raceId = raceId;
 75     }
 76
 77     public int getId() {
 78         return id;
 79     }
 80
 81     public double getLength() {
 82         return length;
 83     }
 84
 85     public String getName() {
 86         return stageName;
 87     }
 88
 89     public String getState() {
 90         return state;
 91     }
 92
 93     public StageType getType() {
 94         return type;
 95     }
 96
 97     public void conclude() {
 98         state = "waiting for results";
 99     }
100
101     public void addSegment(Segment segment) {
102         segmentObjects.add(segment);
103     }
104
105     /**
106      * Get the segmentIds of a stage
107      *
108      * @return an array of SegmentIds of a stage
109      */
110     public int[] getSegmentIds() {
111         int segment_count = segmentObjects.size();
112         int[] id_array = new int[segment_count];
113         for (int i = 0; i < segment_count; i++) {
114             id_array[i] = segmentObjects.get(i).getId();
115         }
116         return id_array;
117     }
118
119     /**
```

```
120      * removes a segment from the stage
121      *
122      * @param segmentIndex
123      */
124     public void removeSegment(int segmentIndex) {
125         segmentObjects.remove(segmentIndex);
126     }
127 
128     /**
129      * checks if the rider wih the passed Id has a result
130      * in the current Stage
131      *
132      * @param riderId
133      * @return true if the rider has a result, false if he doesn't
134      */
135     public boolean doesRiderHaveResult(int riderId) {
136         int result_count = resultObjects.size();
137         // loop to check if the riderId exists
138         for (int i = 0; i < result_count; i++) {
139             if (resultObjects.get(i).getRiderId() == riderId) {
140                 return true;
141             }
142         }
143         return false;
144     }
145 
146     /**
147      * gets the number of segments in the stage
148      *
149      * @return the number of segment in the stage
150      */
151     public int getSegmentsCount() {
152         return segmentObjects.size();
153     }
154 
155     public void addResult(Result result) {
156         resultObjects.add(result);
157     }
158 
159     /**
160      * Removes all the rider's results by the riderId
161      *
162      * @param riderId the Id of the rider
163      */
164     public void removeAllRiderResults(int riderId) {
165         int result_count = resultObjects.size();
166         // loop to find the meant riderId
167         for (int i = 0; i < result_count; i++) {
168             if (resultObjects.get(i).getRiderId() == riderId) {
169                 resultObjects.remove(i);
170                 break;
171             }
172         }
173     }
174 
175     public int getResultsCount() {
176         return resultObjects.size();
```

```
177     }
178
179     public int[] getRidersRankInStage() {
180         // sorts the resultObjects by elapsed time
181         Collections.sort(resultObjects);
182         int result_count = resultObjects.size();
183         int[] id_array = new int[result_count];
184         // loops to store the sorted Ids in a new array which is returned later
185         for (int i = 0; i < result_count; i++) {
186             id_array[i] = resultObjects.get(i).getRiderId();
187         }
188         return id_array;
189     }
190
191     public int[] getRidersPointsInStage() {
192         HashMap<Integer, Integer> riderHashMap = new HashMap<Integer, Integer>();
193         // resultObjects being sorted by Elapsed time
194         Collections.sort(resultObjects);
195         int rider_count = resultObjects.size();
196         // riderHashmap initialization
197         for (int i = 0; i < rider_count; i++) {
198             riderHashMap.put(resultObjects.get(i).getRiderId(), 0);
199         }
200
201         int segment_count = segmentObjects.size();
202         for (int i = 0; i < segment_count; i++) {
203
204             SegmentType segmentType = segmentObjects.get(i).getSegmentType();
205
206             if (segmentType.equals(SegmentType.SPRINT)) {
207
208                 int result_count = resultObjects.size();
209                 ArrayList<Result> segmentResults = new ArrayList<Result>();
210                 /*
211                  * a loop that creates result objects for the number of riders
212                  * if the segment is of type Sprint
213                  */
214                 for (int j = 0; j < result_count; j++) {
215                     int riderId = resultObjects.get(j).getRiderId();
216                     LocalTime checkpoint_start = resultObjects.get(j).getCheckPoint(0);
217                     LocalTime checkpoint_finish = resultObjects.get(j).getCheckPoint(i);
218                     segmentResults.add(new Result(riderId, checkpoint_start,
    checkpoint_finish));
219                 }
220                 Collections.sort(segmentResults);
221
222                 rider_count = segmentResults.size();
223                 /*
224                  * a loop to update the hashmap with the rider points
225                  * for the sprint segment
226                  */
227
228                 for (int j = 0; j < rider_count; j++) {
229                     int finishedAt = j;
230                     int points = ResultHandler.getSegmentPoints(segmentType, finishedAt);
231                     int previousValue = riderHashMap.get(segmentResults.get(j).getRiderId());
232                     riderHashMap.replace(segmentResults.get(j).getRiderId(), previousValue +
```

```
   points);
233                 }
234             }
235 
236         }
237 
238         int result_count = resultObjects.size();
239         int[] points_array = new int[result_count];
240         // a loop to calculate the rider points in the stage
241         for (int i = 0; i < result_count; i++) {
242             int finishedAt = i;
243             int points = ResultHandler.getStagePoints(this.type, finishedAt);
244             int previousValue = riderHashMap.get(resultObjects.get(i).getRiderId());
245             riderHashMap.replace(resultObjects.get(i).getRiderId(), previousValue + points);
246 
247             points_array[i] = riderHashMap.get(resultObjects.get(i).getRiderId());
248         }
249         return points_array;
250     }
251 
252     public int[] getRidersMountainPointsInStage() {
253         HashMap<Integer, Integer> riderHashMap = new HashMap<Integer, Integer>();
254         Collections.sort(resultObjects);
255         int rider_count = resultObjects.size();
256         // Hashmap Initialization
257         for (int i = 0; i < rider_count; i++) {
258             riderHashMap.put(resultObjects.get(i).getRiderId(), 0);
259         }
260 
261         int segment_count = segmentObjects.size();
262         for (int i = 0; i < segment_count; i++) {
263 
264             SegmentType segmentType = segmentObjects.get(i).getSegmentType();
265 
266             if (!segmentType.equals(SegmentType.SPRINT)) {
267 
268                 int result_count = resultObjects.size();
269                 ArrayList<Result> segmentResults = new ArrayList<Result>();
270                 /*
271                  * a loop that creates result objects for the number of riders
272                  * if the segment is not of type Sprint
273                  */
274                 for (int j = 0; j < result_count; j++) {
275                     int riderId = resultObjects.get(j).getRiderId();
276                     LocalTime checkpoint_start = resultObjects.get(j).getCheckPoint(0);
277                     LocalTime checkpoint_finish = resultObjects.get(j).getCheckPoint(i);
278                     segmentResults.add(new Result(riderId, checkpoint_start,
   checkpoint_finish));
279                 }
280                 Collections.sort(segmentResults);
281 
282                 rider_count = segmentResults.size();
283                 /*
284                  * a loop to update the hashmap with the rider points
285                  * for the segments not of type sprint
286                  */
287                 for (int j = 0; j < rider_count; j++) {
```

```
288                     int finishedAt = j;
289                     int points = ResultHandler.getSegmentPoints(segmentType, finishedAt);
290                     int previousValue = riderHashMap.get(segmentResults.get(j).getRiderId());
291                     riderHashMap.replace(segmentResults.get(j).getRiderId(), previousValue +
   points);
292                 }
293             }
294 
295         }
296 
297         int result_count = resultObjects.size();
298         int[] points_array = new int[result_count];
299         // a loop to calculate the rider points in the stage
300         for (int i = 0; i < result_count; i++) {
301             points_array[i] = riderHashMap.get(resultObjects.get(i).getRiderId());
302         }
303         return points_array;
304     }
305 
306     public LocalTime[] getRiderResultsInStage(int riderId) {
307         int result_count = resultObjects.size();
308         // searching for the meant rider id to get the results in stage
309         for (int i = 0; i < result_count; i++) {
310             if (resultObjects.get(i).getRiderId() == riderId) {
311                 return resultObjects.get(i).getRiderResultsInStage();
312             }
313         }
314 
315         return new LocalTime[] {};
316     }
317 
318     public LocalTime getRiderAdjustedElapsedTimeInStage(int riderId) {
319         Collections.sort(resultObjects);
320         /*
321          * if the stage is of type TT, the Elapsed time is directly
322          * returned, otherwise a function to adjust the elapsed time is
323          * called then the adjusted elapsed time is returned
324          */
325         if (type.equals(StageType.TT)) {
326             int result_count = resultObjects.size();
327             for (int i = 0; i < result_count; i++) {
328                 if (resultObjects.get(i).getRiderId() == riderId) {
329                     return resultObjects.get(i).getElapsedTime();
330                 }
331             }
332         } else {
333             int result_count = resultObjects.size();
334             int riderIndex = -1;
335             LocalTime riderElapsedTime = null;
336             for (int i = 0; i < result_count; i++) {
337                 if (resultObjects.get(i).getRiderId() == riderId) {
338                     riderIndex = i;
339                     riderElapsedTime = resultObjects.get(i).getElapsedTime();
340                 }
341             }
342             return getRiderAdjustedElapsed(riderIndex, riderElapsedTime);
343         }
```

```
344         return null;
345     }
346 
347     /**
348      * gets the adjusted elapsed times of the riders
349      *
350      * @param riderIndex       the index of the rider in resultObjects
351      * @param riderElapsedTime the rider's elapsed time
352      * @return the adjusted elapsed time
353      */
354     private LocalTime getRiderAdjustedElapsed(int riderIndex, LocalTime riderElapsedTime) {
355         /*
356          * if the difference between the elapsed times is less than
357          * one second, they are adjusted
358          */
359         for (int i = riderIndex - 1; i >= 0; i--) {
360             LocalTime tempTime = resultObjects.get(i).getElapsedTime();
361             long differenceInNanoSeconds = tempTime.until(riderElapsedTime, ChronoUnit.NANOS);
362             if (differenceInNanoSeconds < 1000000000) { // 1000000000 nanoseconds is
    equivalent to 1 Second
363                 riderElapsedTime = tempTime;
364             }
365         }
366         return riderElapsedTime;
367     }
368 
369     public LocalTime[] getRankedAdjustedElapsedTimesInStage() {
370         Collections.sort(resultObjects);
371         int result_count = resultObjects.size();
372         LocalTime[] localTimeArray = new LocalTime[result_count];
373         // the elapsed times are stored in an array named localTimeArray
374         for (int i = 0; i < result_count; i++) {
375             localTimeArray[i] = resultObjects.get(i).getElapsedTime();
376         }
377 
378         /*
379          * if the stage time is of type TT the sorted arrray with elapsed
380          * times is directly, otherwirse the elapsed times are adjusted
381          * then returned
382          */
383         if (type.equals(StageType.TT)) {
384             return localTimeArray;
385         } else {
386             for (int i = 0; i < result_count; i++) {
387                 int riderIndex = i;
388                 LocalTime riderElapsedTime = resultObjects.get(i).getElapsedTime();
389                 localTimeArray[i] = getRiderAdjustedElapsed(riderIndex, riderElapsedTime);
390             }
391             return localTimeArray;
392         }
393 
394     }
395 
396     public ArrayList<Result> getResultObjects() {
397         return resultObjects;
398     }
399 
```

```
400}
401
```

```java
package cycling;

import java.io.Serializable;

/**
 * Segment-A class which stores information about segments
 * and handles some of the operations related to segments.
 */
public class Segment implements Serializable {
    private int id;

    private int stageId;
    private Double location;
    private SegmentType type;
    private Double averageGradient;
    private Double length;

    /**
     * Constructor for the Objects of Segment class
     *
     * @param segmentId       the Id of the segment
     * @param stageId         The Id of the stage which the segment is a part of
     * @param location        the place at which the segment takes place
     * @param type            type of the segment
     * @param averageGradient the average gradient for the segment
     * @param length          length of the segment
     */
    public Segment(int segmentId, int stageId, Double location, SegmentType type, Double averageGradient,
            Double length) {
        this.id = segmentId;
        this.setStageId(stageId);
        this.setLocation(location);
        this.type = type;
        this.setAverageGradient(averageGradient);
        this.setLength(length);
    }

    public Double getLength() {
        return length;
    }

    public void setLength(Double length) {
        this.length = length;
    }

    public Double getAverageGradient() {
        return averageGradient;
    }

    public void setAverageGradient(Double averageGradient) {
        this.averageGradient = averageGradient;
    }

    public Double getLocation() {
        return location;
    }
```

```
57
58      public void setLocation(Double location) {
59          this.location = location;
60      }
61
62      public int getStageId() {
63          return stageId;
64      }
65
66      public void setStageId(int stageId) {
67          this.stageId = stageId;
68      }
69
70      /**
71       * A 2nd constructor
72       *
73       * @param segmentId the Id of the segemnt
74       * @param stageId   the Id of the stage
75       * @param location  the place at which the segment takes place
76       */
77      public Segment(int segmentId, int stageId, double location) {
78          this.id = segmentId;
79          this.setStageId(stageId);
80          this.setLocation(location);
81          this.type = SegmentType.SPRINT;
82      }
83
84      public int getId() {
85          return id;
86      }
87
88      public SegmentType getSegmentType() {
89          return type;
90      }
91  }
92
```

```java
package cycling;

import java.io.Serializable;

/**
 * Team-A class which stores information about teams and
 * handles some of the operations related to teams.
 */
public class Team implements Serializable {
    private int id;
    private String name;
    private String description;
    private ArrayList<Rider> riderObjects = new ArrayList<Rider>();

    /**
     * Constructor for the Objects of Team class
     *
     * @param teamId      the Id of the team
     * @param name        The identifier name of the team
     * @param description Team's description
     */
    public Team(int teamId, String name, String description) {
        this.id = teamId;
        this.name = name;
        this.description = description;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public void addRider(Rider rider) {
        riderObjects.add(rider);
    }

    public int getRiderCount() {
        return riderObjects.size();
    }

    public int getRiderIdAtIndex(int index) {
        return riderObjects.get(index).getId();
```

```
59      }
60
61      /**
62       * Checks the existence of a rider by the Id
63       *
64       * @param riderId the Id of the rider
65       * @return true if the rider exist, false if not
66       */
67      public boolean doesRiderExist(int riderId) {
68          int rider_count = riderObjects.size();
69          // loop to find if the riderId exists
70          for (int i = 0; i < rider_count; i++) {
71              if (riderObjects.get(i).getId() == riderId) {
72                  return true;
73              }
74          }
75          return false;
76      }
77
78      /**
79       * removes a rider by the Id
80       *
81       * @param riderId the Id of the rider
82       */
83      public void removeRider(int riderId) {
84          int rider_count = riderObjects.size();
85          // loop to find the meant riderId
86          for (int i = 0; i < rider_count; i++) {
87              if (riderObjects.get(i).getId() == riderId) {
88                  riderObjects.remove(i);
89                  break;
90              }
91          }
92      }
93  }
94
```

```java
package cycling;

import java.io.Serializable;

/**
 * Rider- A class which stores information about riders
 * and handles some of the operations related ot riders.
 */
public class Rider implements Serializable {
    private String name;
    private int yearOfBirth;
    private int id;
    private int teamId;

    /**
     * Constructor for the Objects of Rider class
     *
     * @param riderId     the Id of the Rider
     * @param teamId      the Id of the team which the rider is a part of
     * @param name        the name of rider
     * @param yearOfBirth the year of birth of the rider
     */
    public Rider(int riderId, int teamId, String name, int yearOfBirth) {
        this.id = riderId;
        this.name = name;
        this.yearOfBirth = yearOfBirth;
        this.setTeamId(teamId);
    }

    public int getId() {
        return this.id;
    }

    public int getTeamId() {
        return teamId;
    }

    public void setTeamId(int teamId) {
        this.teamId = teamId;
    }

    public String getName() {
        return this.name;
    }

    public int getYearOfBirth() {
        return this.yearOfBirth;
    }

}

```

```java
package cycling;

import java.io.Serializable;

/**
 * Result- A class which stores information about the results
 * and handles some of the operations related to results
 */
public class Result implements Comparable<Result>, Serializable {

    private int riderId;
    private LocalTime[] checkpoints;

    /**
     * Constructor for the Objects of Reuslt class
     *
     * @param riderId     the Id of the rider
     * @param checkpoints An array of times at which the rider reached each of the
     *                    segments of the stage, including the start time and the
     *                    finish line.
     */
    public Result(int riderId, LocalTime... checkpoints) {
        this.riderId = riderId;
        this.checkpoints = new LocalTime[checkpoints.length];
        for (int i = 0; i < checkpoints.length; i++) {
            this.checkpoints[i] = checkpoints[i];
        }
    }

    public int getRiderId() {
        return riderId;
    }

    /**
     * get the elapsed time of the rider in the stage
     *
     * @return the difference between the start time and finish time ; elapsed time
     */
    public LocalTime getElapsedTime() {
        if (checkpoints.length > 1) {
            return LocalTime.ofNanoOfDay(checkpoints[0].until(checkpoints[checkpoints.length -
1], ChronoUnit.NANOS));
        } else {
            return checkpoints[0];
        }
    }

    @Override
    public int compareTo(Result result) {
        if (getElapsedTime() == null || result.getElapsedTime() == null) {
            return 0;
        }
        return getElapsedTime().compareTo(result.getElapsedTime());
    }

    public int getRiderPoints(int finishedAt, StageType type) {
        return 0;
```

```
59      }
60
61      /**
62       * gets the time at which a specific segment was finished
63       *
64       * @param index the order of the segment finished in the stage
65       * @return the time at which the segment was completed
66       */
67      public LocalTime getCheckPoint(int index) {
68          return checkpoints[index];
69      }
70
71      public LocalTime[] getRiderResultsInStage() {
72          LocalTime[] resultsArray = new LocalTime[checkpoints.length - 1];
73          /*
74           * loop to store the time the segments where reached in
75           * the resultsArray
76           */
77          for (int i = 1; i < checkpoints.length - 1; i++) {
78              resultsArray[i - 1] = checkpoints[i];
79          }
80          // the elapsed time is stored as the last element in the resultsArray
81          resultsArray[resultsArray.length - 1] = LocalTime
82                  .ofNanoOfDay(checkpoints[0].until(checkpoints[checkpoints.length - 1],
  ChronoUnit.NANOS));
83          return resultsArray;
84      }
85  }
86
```

```java
package cycling;

import java.util.ArrayList;

/**
 * ResultHandler- A class that handles the operations related
 * to calculating the points of the riders based on the stages
 * and segments
 */
public class ResultHandler {

    /**
     * gets the rider's points in the stage
     *
     * @param type       the stage type
     * @param finishedAt the place at which the rider finished the stage
     * @return rider's points in the stage based on his place
     *         and the type of the stage.
     */
    public static int getStagePoints(StageType type, int finishedAt) {

        if (finishedAt > 14 || finishedAt < 0)
            return 0;

        HashMap<StageType, List<Integer>> pointsMap = new HashMap<StageType, List<Integer>>();

        pointsMap.put(StageType.FLAT, new ArrayList<Integer>());
        pointsMap.put(StageType.MEDIUM_MOUNTAIN, new ArrayList<Integer>());
        pointsMap.put(StageType.HIGH_MOUNTAIN, new ArrayList<Integer>());
        pointsMap.put(StageType.TT, new ArrayList<Integer>());
        pointsMap.get(StageType.TT).add(20);
        pointsMap.get(StageType.TT).add(17);
        pointsMap.get(StageType.TT).add(15);
        pointsMap.get(StageType.TT).add(13);
        pointsMap.get(StageType.TT).add(11);
        pointsMap.get(StageType.TT).add(10);
        pointsMap.get(StageType.TT).add(9);
        pointsMap.get(StageType.TT).add(8);
        pointsMap.get(StageType.TT).add(7);
        pointsMap.get(StageType.TT).add(6);
        pointsMap.get(StageType.TT).add(5);
        pointsMap.get(StageType.TT).add(4);
        pointsMap.get(StageType.TT).add(3);
        pointsMap.get(StageType.TT).add(2);
        pointsMap.get(StageType.TT).add(1);

        pointsMap.get(StageType.HIGH_MOUNTAIN).add(20);
        pointsMap.get(StageType.HIGH_MOUNTAIN).add(17);
        pointsMap.get(StageType.HIGH_MOUNTAIN).add(15);
        pointsMap.get(StageType.HIGH_MOUNTAIN).add(13);
        pointsMap.get(StageType.HIGH_MOUNTAIN).add(11);
        pointsMap.get(StageType.HIGH_MOUNTAIN).add(10);
        pointsMap.get(StageType.HIGH_MOUNTAIN).add(9);
        pointsMap.get(StageType.HIGH_MOUNTAIN).add(8);
        pointsMap.get(StageType.HIGH_MOUNTAIN).add(7);
        pointsMap.get(StageType.HIGH_MOUNTAIN).add(6);
        pointsMap.get(StageType.HIGH_MOUNTAIN).add(5);
```

```
60          pointsMap.get(StageType.HIGH_MOUNTAIN).add(4);
61          pointsMap.get(StageType.HIGH_MOUNTAIN).add(3);
62          pointsMap.get(StageType.HIGH_MOUNTAIN).add(2);
63          pointsMap.get(StageType.HIGH_MOUNTAIN).add(1);
64
65          pointsMap.get(StageType.MEDIUM_MOUNTAIN).add(30);
66          pointsMap.get(StageType.MEDIUM_MOUNTAIN).add(25);
67          pointsMap.get(StageType.MEDIUM_MOUNTAIN).add(22);
68          pointsMap.get(StageType.MEDIUM_MOUNTAIN).add(19);
69          pointsMap.get(StageType.MEDIUM_MOUNTAIN).add(17);
70          pointsMap.get(StageType.MEDIUM_MOUNTAIN).add(15);
71          pointsMap.get(StageType.MEDIUM_MOUNTAIN).add(13);
72          pointsMap.get(StageType.MEDIUM_MOUNTAIN).add(11);
73          pointsMap.get(StageType.MEDIUM_MOUNTAIN).add(9);
74          pointsMap.get(StageType.MEDIUM_MOUNTAIN).add(7);
75          pointsMap.get(StageType.MEDIUM_MOUNTAIN).add(6);
76          pointsMap.get(StageType.MEDIUM_MOUNTAIN).add(5);
77          pointsMap.get(StageType.MEDIUM_MOUNTAIN).add(4);
78          pointsMap.get(StageType.MEDIUM_MOUNTAIN).add(3);
79          pointsMap.get(StageType.MEDIUM_MOUNTAIN).add(2);
80
81          pointsMap.get(StageType.FLAT).add(50);
82          pointsMap.get(StageType.FLAT).add(30);
83          pointsMap.get(StageType.FLAT).add(20);
84          pointsMap.get(StageType.FLAT).add(18);
85          pointsMap.get(StageType.FLAT).add(16);
86          pointsMap.get(StageType.FLAT).add(14);
87          pointsMap.get(StageType.FLAT).add(12);
88          pointsMap.get(StageType.FLAT).add(10);
89          pointsMap.get(StageType.FLAT).add(8);
90          pointsMap.get(StageType.FLAT).add(7);
91          pointsMap.get(StageType.FLAT).add(6);
92          pointsMap.get(StageType.FLAT).add(5);
93          pointsMap.get(StageType.FLAT).add(4);
94          pointsMap.get(StageType.FLAT).add(3);
95          pointsMap.get(StageType.FLAT).add(2);
96
97          return pointsMap.get(type).get(finishedAt);
98      }
99
100     /**
101      * gets the rider's points in the segment
102      *
103      * @param segmentType the type of the segment
104      * @param finishedAt  the place at which the rider finished the segment
105      * @return the rider's points in the segment based on his place
106      *         in the segment and the type of the segment.
107      */
108     public static int getSegmentPoints(SegmentType segmentType, int finishedAt) {
109
110         if (finishedAt > 14 || finishedAt < 0)
111             return 0;
112
113         HashMap<SegmentType, List<Integer>> pointsMap = new HashMap<SegmentType,
    List<Integer>>();
114
115         pointsMap.put(SegmentType.C4, new ArrayList<Integer>());
```

```
116         pointsMap.put(SegmentType.C3, new ArrayList<Integer>());
117         pointsMap.put(SegmentType.C2, new ArrayList<Integer>());
118         pointsMap.put(SegmentType.C1, new ArrayList<Integer>());
119         pointsMap.put(SegmentType.HC, new ArrayList<Integer>());
120         pointsMap.put(SegmentType.SPRINT, new ArrayList<Integer>());
121         pointsMap.get(SegmentType.SPRINT).add(20);
122         pointsMap.get(SegmentType.SPRINT).add(17);
123         pointsMap.get(SegmentType.SPRINT).add(15);
124         pointsMap.get(SegmentType.SPRINT).add(13);
125         pointsMap.get(SegmentType.SPRINT).add(11);
126         pointsMap.get(SegmentType.SPRINT).add(10);
127         pointsMap.get(SegmentType.SPRINT).add(9);
128         pointsMap.get(SegmentType.SPRINT).add(8);
129         pointsMap.get(SegmentType.SPRINT).add(7);
130         pointsMap.get(SegmentType.SPRINT).add(6);
131         pointsMap.get(SegmentType.SPRINT).add(5);
132         pointsMap.get(SegmentType.SPRINT).add(4);
133         pointsMap.get(SegmentType.SPRINT).add(3);
134         pointsMap.get(SegmentType.SPRINT).add(2);
135         pointsMap.get(SegmentType.SPRINT).add(1);
136
137         pointsMap.get(SegmentType.HC).add(20);
138         pointsMap.get(SegmentType.HC).add(15);
139         pointsMap.get(SegmentType.HC).add(12);
140         pointsMap.get(SegmentType.HC).add(10);
141         pointsMap.get(SegmentType.HC).add(8);
142         pointsMap.get(SegmentType.HC).add(6);
143         pointsMap.get(SegmentType.HC).add(4);
144         pointsMap.get(SegmentType.HC).add(2);
145         pointsMap.get(SegmentType.HC).add(0);
146         pointsMap.get(SegmentType.HC).add(0);
147         pointsMap.get(SegmentType.HC).add(0);
148         pointsMap.get(SegmentType.HC).add(0);
149         pointsMap.get(SegmentType.HC).add(0);
150         pointsMap.get(SegmentType.HC).add(0);
151         pointsMap.get(SegmentType.HC).add(0);
152
153         pointsMap.get(SegmentType.C1).add(10);
154         pointsMap.get(SegmentType.C1).add(8);
155         pointsMap.get(SegmentType.C1).add(6);
156         pointsMap.get(SegmentType.C1).add(4);
157         pointsMap.get(SegmentType.C1).add(2);
158         pointsMap.get(SegmentType.C1).add(1);
159         pointsMap.get(SegmentType.C1).add(0);
160         pointsMap.get(SegmentType.C1).add(0);
161         pointsMap.get(SegmentType.C1).add(0);
162         pointsMap.get(SegmentType.C1).add(0);
163         pointsMap.get(SegmentType.C1).add(0);
164         pointsMap.get(SegmentType.C1).add(0);
165         pointsMap.get(SegmentType.C1).add(0);
166         pointsMap.get(SegmentType.C1).add(0);
167         pointsMap.get(SegmentType.C1).add(0);
168
169         pointsMap.get(SegmentType.C2).add(5);
170         pointsMap.get(SegmentType.C2).add(3);
171         pointsMap.get(SegmentType.C2).add(2);
172         pointsMap.get(SegmentType.C2).add(1);
```

```
173         pointsMap.get(SegmentType.C2).add(0);
174         pointsMap.get(SegmentType.C2).add(0);
175         pointsMap.get(SegmentType.C2).add(0);
176         pointsMap.get(SegmentType.C2).add(0);
177         pointsMap.get(SegmentType.C2).add(0);
178         pointsMap.get(SegmentType.C2).add(0);
179         pointsMap.get(SegmentType.C2).add(0);
180         pointsMap.get(SegmentType.C2).add(0);
181         pointsMap.get(SegmentType.C2).add(0);
182         pointsMap.get(SegmentType.C2).add(0);
183         pointsMap.get(SegmentType.C2).add(0);
184
185         pointsMap.get(SegmentType.C3).add(2);
186         pointsMap.get(SegmentType.C3).add(1);
187         pointsMap.get(SegmentType.C3).add(0);
188         pointsMap.get(SegmentType.C3).add(0);
189         pointsMap.get(SegmentType.C3).add(0);
190         pointsMap.get(SegmentType.C3).add(0);
191         pointsMap.get(SegmentType.C3).add(0);
192         pointsMap.get(SegmentType.C3).add(0);
193         pointsMap.get(SegmentType.C3).add(0);
194         pointsMap.get(SegmentType.C3).add(0);
195         pointsMap.get(SegmentType.C3).add(0);
196         pointsMap.get(SegmentType.C3).add(0);
197         pointsMap.get(SegmentType.C3).add(0);
198         pointsMap.get(SegmentType.C3).add(0);
199         pointsMap.get(SegmentType.C3).add(0);
200
201         pointsMap.get(SegmentType.C4).add(1);
202         pointsMap.get(SegmentType.C4).add(0);
203         pointsMap.get(SegmentType.C4).add(0);
204         pointsMap.get(SegmentType.C4).add(0);
205         pointsMap.get(SegmentType.C4).add(0);
206         pointsMap.get(SegmentType.C4).add(0);
207         pointsMap.get(SegmentType.C4).add(0);
208         pointsMap.get(SegmentType.C4).add(0);
209         pointsMap.get(SegmentType.C4).add(0);
210         pointsMap.get(SegmentType.C4).add(0);
211         pointsMap.get(SegmentType.C4).add(0);
212         pointsMap.get(SegmentType.C4).add(0);
213         pointsMap.get(SegmentType.C4).add(0);
214         pointsMap.get(SegmentType.C4).add(0);
215         pointsMap.get(SegmentType.C4).add(0);
216
217         return pointsMap.get(segmentType).get(finishedAt);
218     }
219
220 }
221
```

```
 1import java.io.IOException;
18
19
20public class CyclingPortalInterfaceTestApp {
21
22    /**
23     * Test method.
24     *
25     * @param args not used
26     */
27    public static void main2(String[] args) {
28        System.out.println("The system compiled and started the execution...");
29
30        CyclingPortal portal = new CyclingPortal();
31        // CyclingPortalInterface portal = new BadCyclingPortal();
32
33        assert (portal.getRaceIds().length == 0)
34                : "Innitial SocialMediaPlatform not empty as required or not returning an
  empty array.";
35
36    }
37
38    public static void main(String[] args) {
39        CyclingPortal portal = new CyclingPortal();
40        try {
41            portal.createRace("Race01", "First Race");
42            portal.createRace("Race02", "Second Race");
43            portal.removeRaceById(2);
44            portal.createRace("Race03", "Third Race");
45            portal.removeRaceByName("Race03");
46            portal.createRace("Race04", "Fourth Race");
47            System.out.println("-> Race Ids");
48            for (int ret : portal.getRaceIds()) {
49                System.out.println(ret);
50            }
51            System.out.println("-> Race Details");
52            System.out.println(portal.viewRaceDetails(1));
53            portal.createTeam("Team01", "First Team");
54            portal.createTeam("Team02", "Second Team");
55            portal.removeTeam(2);
56            portal.createTeam("Team03", "Third Team");
57            System.out.println("-> Team Ids");
58            for (int ret : portal.getTeams()) {
59                System.out.println(ret);
60            }
61            portal.createRider(1, "Name1", 1999);
62            portal.createRider(1, "Name2", 1999);
63            portal.createRider(1, "Name3", 1999);
64            portal.createRider(1, "Name4", 1999);
65            portal.removeRider(4);
66            portal.createRider(1, "Name5", 1999);
67            System.out.println("-> Team Rider Ids");
68            for (int ret : portal.getTeamRiders(1)) {
69                System.out.println(ret);
70            }
71            portal.addStageToRace(1, "FLAT", "Description1", 6, LocalDateTime.now(),
  StageType.FLAT);
```

```
 72             portal.addStageToRace(1, "HIGH_MOUNTAIN", "Description1", 8, LocalDateTime.now(),
   StageType.FLAT);
 73             portal.addStageToRace(1, "FLAT2", "Description1", 6, LocalDateTime.now(),
   StageType.FLAT);
 74             portal.addStageToRace(1, "HIGH_MOUNTAIN2", "Description1", 8, LocalDateTime.now(),
   StageType.HIGH_MOUNTAIN);
 75             portal.removeStageById(3);
 76             portal.removeStageById(4);
 77             System.out.println("-> Race Number of Stages");
 78             System.out.println(portal.getNumberOfStages(1));
 79             System.out.println("-> Race Stage Ids");
 80             for (int ret : portal.getRaceStages(1)) {
 81                 System.out.println(ret);
 82             }
 83             System.out.println("-> Stage Length");
 84             System.out.println(portal.getStageLength(1));
 85             portal.addCategorizedClimbToStage(1, 2.0, SegmentType.HC, 1.0, 1.0);
 86             portal.addIntermediateSprintToStage(1, 2);
 87             portal.addCategorizedClimbToStage(1, 2.0, SegmentType.HC, 1.0, 1.0);
 88             portal.addIntermediateSprintToStage(1, 2);
 89             portal.addCategorizedClimbToStage(2, 2.0, SegmentType.HC, 1.0, 1.0);
 90             portal.addIntermediateSprintToStage(2, 2);
 91             portal.removeSegment(3);
 92             portal.removeSegment(4);
 93             System.out.println("-> Stage Segment Ids");
 94             for (int ret : portal.getStageSegments(1)) {
 95                 System.out.println(ret);
 96             }
 97             portal.registerRiderResultsInStage(1, 1, LocalTime.of(1, 0),
 98                     LocalTime.of(1, 30),
 99                     LocalTime.of(1, 45),
100                     LocalTime.of(2, 30, 35));
101             portal.registerRiderResultsInStage(1, 2, LocalTime.of(1, 0),
102                     LocalTime.of(1, 25),
103                     LocalTime.of(1, 40),
104                     LocalTime.of(2, 30, 30));
105             portal.registerRiderResultsInStage(1, 3, LocalTime.of(1, 0),
106                     LocalTime.of(1, 28),
107                     LocalTime.of(1, 43),
108                     LocalTime.of(2, 30, 34, 700));
109             // portal.registerRiderResultsInStage(1, 5, LocalTime.of(1, 0),
110             //         LocalTime.of(1, 26),
111             //         LocalTime.of(1, 41),
112             //         LocalTime.of(2, 30, 33, 800));
113             portal.registerRiderResultsInStage(2, 1, LocalTime.of(1, 0),
114                     LocalTime.of(1, 30),
115                     LocalTime.of(1, 45),
116                     LocalTime.of(1, 30, 35));
117             portal.registerRiderResultsInStage(2, 2, LocalTime.of(1, 0),
118                     LocalTime.of(1, 25),
119                     LocalTime.of(1, 40),
120                     LocalTime.of(2, 30, 30));
121             portal.registerRiderResultsInStage(2, 3, LocalTime.of(1, 0),
122                     LocalTime.of(1, 27),
123                     LocalTime.of(1, 43),
124                     LocalTime.of(2, 30, 32));
125             portal.registerRiderResultsInStage(2, 5, LocalTime.of(1, 0),
```

```
126                     LocalTime.of(1, 26),
127                     LocalTime.of(1, 41),
128                     LocalTime.of(2, 30, 31));
129             portal.deleteRiderResultsInStage(2, 5);
130             // portal.concludeStagePreparation(2);
131             portal.removeRider(2);
132             portal.registerRiderResultsInStage(2, 5, LocalTime.of(1, 0),
133                     LocalTime.of(1, 26),
134                     LocalTime.of(1, 41),
135                     LocalTime.of(2, 30, 31));
136             System.out.println("-> Rider Result In Stage");
137             for (LocalTime ret : portal.getRiderResultsInStage(1, 1)) {
138                 System.out.println(ret);
139             }
140             System.out.println("-> Rider Adjusted Elapsed Time");
141             System.out.println(portal.getRiderAdjustedElapsedTimeInStage(1, 1));
142             System.out.println("-> Ranked Adjusted Elapsed Times In Stage");
143             for (LocalTime ret : portal.getRankedAdjustedElapsedTimesInStage(1)) {
144                 System.out.println(ret);
145             }
146             System.out.println("-> Riders Ranks in Stage");
147             for (int ret : portal.getRidersRankInStage(1)) {
148                 System.out.println(ret);
149             }
150             System.out.println("-> Riders Points in Stage");
151             for (int ret : portal.getRidersPointsInStage(1)) {
152                 System.out.println(ret);
153             }
154             System.out.println("-> Riders Mountain in Stage");
155             for (int ret : portal.getRidersMountainPointsInStage(1)) {
156                 System.out.println(ret);
157             }
158 
159             System.out.println("-> Riders Points in Race");
160             for (int ret : portal.getRidersPointsInRace(1)) {
161                 System.out.println(ret);
162             }
163             System.out.println("-> Riders Mountain Points in Race");
164             for (int ret : portal.getRidersMountainPointsInRace(1)) {
165                 System.out.println(ret);
166             }
167             System.out.println("-> Riders General Classification in Race");
168             for (int ret : portal.getRidersGeneralClassificationRank(1)) {
169                 System.out.println(ret);
170             }
171             System.out.println("-> Riders Points Classification in Race");
172             for (int ret : portal.getRidersPointClassificationRank(1)) {
173                 System.out.println(ret);
174             }
175             System.out.println("-> Riders Mountain Classification in Race");
176             for (int ret : portal.getRidersMountainPointClassificationRank(1)) {
177                 System.out.println(ret);
178             }
179             System.out.println("-> Riders Classification Times in Race");
180             for (LocalTime ret : portal.getGeneralClassificationTimesInRace(1)) {
181                 System.out.println(ret);
182             }
```

```
183
184             // portal.eraseCyclingPortal();
185             // portal.createRace("Race05", "First Race");
186             // portal.createRace("Race06", "Second Race");
187             // portal.removeRaceById(2);
188             // portal.createRace("Race03", "Third Race");
189             // portal.removeRaceByName("Race03");
190             // portal.createRace("Race04", "Fourth Race");
191             // System.out.println("-> Race Ids");
192             // for (int ret : portal.getRaceIds()) {
193             // System.out.println(ret);
194             // }
195             // System.out.println("-> Riders Classification Times in Race");
196             // for (LocalTime ret : portal.getGeneralClassificationTimesInRace(1)) {
197             // System.out.println(ret);
198             // }
199
200             portal.saveCyclingPortal("filename.txt");
201             portal.eraseCyclingPortal();
202             System.out.println("-> Race Ids");
203             for (int ret : portal.getRaceIds()) {
204                 System.out.println(ret);
205             }
206             portal.loadCyclingPortal("filename.txt");
207             System.out.println("-> Race Ids");
208             for (int ret : portal.getRaceIds()) {
209                 System.out.println(ret);
210             }
211
212         } catch (IllegalNameException e) {
213             e.printStackTrace();
214         } catch (InvalidNameException e) {
215             e.printStackTrace();
216         } catch (IDNotRecognisedException e) {
217             e.printStackTrace();
218         } catch (InvalidLengthException e) {
219             e.printStackTrace();
220         } catch (InvalidLocationException e) {
221             e.printStackTrace();
222         } catch (InvalidStageStateException e) {
223             e.printStackTrace();
224         } catch (InvalidStageTypeException e) {
225             e.printStackTrace();
226         } catch (DuplicatedResultException e) {
227             e.printStackTrace();
228         } catch (InvalidCheckpointsException e) {
229             e.printStackTrace();
230         } catch (NameNotRecognisedException e) {
231             e.printStackTrace();
232         } catch (IOException e) {
233             e.printStackTrace();
234         } catch (ClassNotFoundException e) {
235             e.printStackTrace();
236         }
237
238     }
239
```

```
240}
241
```