```java
package cycling;

import java.io.*;
import java.lang.reflect.Array;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.*;

public class CyclingPortal implements CyclingPortalInterface , Serializable, Sorts {
    //Array of all the used id's in the cycling portal, these are unique ids for all needed identifiers.
    ArrayList<Integer> usedIds = new ArrayList<>();
    //Hashmap of all the teams in the cycling portal, each value is the object Team
    //Hashmap<teamID, Team object>
    HashMap<Integer, Team> teamHashMap = new HashMap<>();
    //Hashmap of all the races in the cycling portal, each value is a race object
    //Hashmap<RaceID, Race Object>
    HashMap<Integer, Race> raceHashMap = new HashMap<>();

    private int createNewID(){
        //Function to make a new unique ID
        Random random = new Random();
        int id = random.nextInt(10000);
        if(usedIds.contains(id)){
            createNewID();
        }
        return id;
    }
    private void removeId(int id){
        //Function to remove an ID from usedIds
        usedIds.remove(usedIds.indexOf(id));
    }
    private boolean idExists(int id) {
        //Function to see if an id is already in use
        //Return True if it exists and False if not
        if (usedIds.contains(id)){
            return true;
        }
        return false;
    }
    private boolean nameIsInvalid(String name){
        //Function to check a name against a set if rules to see if its valid
        if(name.equals(null) || name.equals("") || name.length() >= 30){
```

```
43              return true;
44          }
45          return false;
46      }
47      private boolean stageNameIsIllegal(Integer raceId, String
    stageName){
48          //Function to check if a stage name os valid or not
49          for(Stage stage : raceHashMap.get(raceId).getStages().
    values()){
50              if(stage.getStageName().equals(stageName)){
51                  return true;
52              }
53          }
54          return false;
55      }
56      private boolean raceNameIsIllegal(String name){
57          //Function to check if a race name is valid or not
58          for(Race race : raceHashMap.values()){
59              if(race.getRaceName().equals(name)){
60                  return true;
61              }
62          }
63          return false;
64      }
65      private boolean teamNameIsIllegal(String name) {
66          //Function to check if a team name is valid or not
67          for (Team team : teamHashMap.values()) {
68              if (team.getTeamName() == name) {
69                  return true;
70              }
71          }
72          return false;
73      }
74      private boolean locationIsInvalid(Stage stage, Double
    location){
75          //Function to check if a location is a valid length for
     a select stage
76          if(location < stage.getStageLength()){
77              return false;
78          }
79          return true;
80      }
81      private Stage getStageFromStageID(Integer stageId){
82          //Function to get a stage object given a stage ID
83          for(Race race : raceHashMap.values()){
84              for(Integer stageIds : race.getStages().keySet()){
85                  if(stageIds.equals(stageId)){
86                      return race.getStages().get(stageId);
```

```
87                  }
88              }
89          }
90          return null;
91      }
92      private boolean stageStateUnfinished(Integer stageId){
93          //Function to return if a stage is finished or not
94          Stage stage = getStageFromStageID(stageId);
95          if(stage.getStageState().equals("Unfinished")){
96              return true;
97          }
98          return false;
99      }
100     private boolean stageStateWaitingForResults(Integer
    stageId){
101         //Function to check if a stage is waiting for some
    results from some riders
102         Stage stage = getStageFromStageID(stageId);
103         if(stage.getStageState().equals("waiting for results"
    )){
104             return true;
105         }
106         return false;
107     }
108     private Integer getStageFromSegmentID(Integer segmentId){
109         //A function to return a stage object given a segment
    ID
110         for(Race race: raceHashMap.values()){
111             for(Stage stage : race.getStages().values()){
112                 for(Integer segmentIds : stage.getSegments().
    keySet()){
113                     if(segmentIds.equals(segmentId)){
114                         return stage.getStageId();
115                     }
116                 }
117             }
118         }
119         return null;
120
121     }
122
123
124         @Override
125         public int[] getRaceIds() {
126             //Function to return all the raceID's in the
    raceHashMap
127             ArrayList<Integer> intKeys = new ArrayList<Integer
    >(raceHashMap.keySet());
```

```
128             return intKeys.stream().mapToInt(Integer::intValue
    ).toArray();
129         }
130 
131         @Override
132         public int createRace(String name, String description
    ) throws IllegalNameException, InvalidNameException {
133             //Function to create a race object and add it to
    the RaceHashMap
134             if(nameIsInvalid(name)){
135                 throw new InvalidNameException("This name does
     not meet system standards");
136             }
137             if(raceNameIsIllegal(name)) {
138                 throw new IllegalNameException("race name : "
     + name + " already exists in the system");
139             }
140             int raceId = createNewID();
141             Race newRace = new Race( name , raceId ,
    description);
142             usedIds.add(raceId);
143             raceHashMap.put(raceId , newRace);
144             return newRace.getRaceId();
145         }
146 
147         @Override
148         public String viewRaceDetails(int raceId) throws
    IDNotRecognisedException {
149             //Function to view the details of a race with its
    raceId
150             if(raceHashMap.containsKey(raceId)){
151                 return raceHashMap.get(raceId).toString();
152             }else{
153                 throw new IDNotRecognisedException("Race is
    not present");
154             }
155         }
156 
157         @Override
158         public void removeRaceById(int raceId) throws
    IDNotRecognisedException {
159             //Function to remove a race from the raceHashMap
    given its ID
160             if(raceHashMap.containsKey(raceId)){
161                 raceHashMap.remove(raceId);
162                 removeId(raceId);
163                 assert !(raceHashMap.containsKey(raceId));
164             }else{
```

```
165                 throw new IDNotRecognisedException("Race is
    not present");
166             }
167         }
168 
169         @Override//Test written
170         public int getNumberOfStages(int raceId) throws
    IDNotRecognisedException {
171             //Function to get the number of stages a race has
    given ist raceId
172             if (raceHashMap.containsKey(raceId)) {
173                 return raceHashMap.get(raceId).getStages().
    size();
174             } else {
175                 throw new IDNotRecognisedException("RaceId is
    not recognised");
176             }
177         }
178 
179         @Override//Test written
180         public int addStageToRace(int raceId, String stageName
    , String description, double length, LocalDateTime startTime,
    StageType type) throws IDNotRecognisedException,
    IllegalNameException, InvalidNameException,
    InvalidLengthException {
181             //Function to add a stage to a given race,
    providing its stage name, description, length, start time and
    type
182             if(!idExists(raceId)){
183                 throw new IDNotRecognisedException("Race Id
    not recognised ");
184             }else if(stageNameIsIllegal(raceId, stageName)) {
185                 throw new IllegalNameException("Stage name
    already exists in platform");
186             }else if(nameIsInvalid(stageName)){
187                 throw new InvalidNameException("Stage name is
    invalid");
188             }else if(length < 5){
189                 throw new InvalidLengthException("Length of
    stage is less than 5km");
190             }
191             int stageId = createNewID();
192             raceHashMap.get(raceId).addStageToStages(new Stage
    (stageId, stageName, length , type, startTime, description));
193             usedIds.add(stageId);
194 
195             assert usedIds.contains(stageId);
196             assert raceHashMap.get(raceId).getStages().
```

```
196 containsKey(stageId);
197
198             return stageId;
199         }
200
201         @Override//Test written
202         public int[] getRaceStages(int raceId) throws
    IDNotRecognisedException {
203             //Function to return to the
204             if(raceHashMap.containsKey(raceId)){
205                 ArrayList<Integer> returnIds = new ArrayList
    <>();
206                 ArrayList<LocalDateTime> datesOfStages = new
    ArrayList<>();
207                 for (Stage stage : raceHashMap.get(raceId).
    getStages().values()){
208                     datesOfStages.add(stage.getStartTime());
209                 }
210                 datesOfStages.sort(Comparator.naturalOrder());
211
212                 for (LocalDateTime dateTime : datesOfStages){
213                     for (Stage stage : raceHashMap.get(raceId
    ).getStages().values()){
214                         if(stage.getStartTime().equals(
    dateTime)){
215                             returnIds.add(stage.getStageId());
216                         }
217                     }
218                 }
219                 return returnIds.stream().mapToInt(Integer::
    intValue).toArray();
220             }else{
221                 throw new IDNotRecognisedException("raceID not
     recognised ");
222             }
223         }
224
225         @Override
226         public double getStageLength(int stageId) throws
    IDNotRecognisedException {
227             for (Race race: raceHashMap.values()) {
228                 if(race.getStages().containsKey(stageId)){
229                     return race.getStages().get(stageId).
    getStageLength();
230                 }
231             }
232             throw new IDNotRecognisedException("Stage not
    present");
```

```
233         }
234 
235         @Override//Test written
236         public void removeStageById(int stageId) throws
    IDNotRecognisedException {
237             if(idExists(stageId)){
238                 for (Race race : raceHashMap.values()) {
239                     (race.getStages().values()).removeIf(stage
     -> stage.getStageId() == stageId);
240                     removeId(stageId);
241                     assert !(usedIds.contains(stageId));
242                     assert !(race.getStages().containsKey(
    stageId));
243                     return;
244                 }
245             }else{
246                 throw new IDNotRecognisedException("Stage Id
    not recognised");
247             }
248         }
249 
250         @Override //Tested
251         public int addCategorizedClimbToStage(int stageId,
    Double location, SegmentType type, Double averageGradient,
    Double length) throws IDNotRecognisedException,
    InvalidLocationException, InvalidStageStateException,
    InvalidStageTypeException {
252             if(!idExists(stageId)){
253                 throw new IDNotRecognisedException("Stage Id
    does not exist");
254             }
255             if(locationIsInvalid(getStageFromStageID(stageId
    ) , location)){
256                 throw new InvalidLocationException("Location
    is invalid for segment");
257             }
258             if(!stageStateUnfinished(stageId)){
259                 throw new InvalidStageStateException("Stage
    state is not acceptable");
260             }
261             if(getStageFromStageID(stageId).getType().equals(
    StageType.TT)){
262                 throw new InvalidStageTypeException("Not a
    valid stage type to add a climb too");
263             }
264 
265             Stage stage = getStageFromStageID(stageId);
266             int segmentId = createNewID();
```

```
267             stage.addSegmentToSegmentHashmap(segmentId, new
    CategorisedClimb(segmentId , location , type , new ArrayList
    <>() , averageGradient , length));
268 
269             usedIds.add(segmentId);
270             return segmentId;
271         }
272 
273         @Override//Test written
274         public int addIntermediateSprintToStage(int stageId,
    double location) throws IDNotRecognisedException,
    InvalidLocationException, InvalidStageStateException,
    InvalidStageTypeException {
275             if(!idExists(stageId)){
276                 throw new IDNotRecognisedException("Stage Id
    does not exist");
277             }
278             if(locationIsInvalid(getStageFromStageID(stageId
    ) , location)){
279                 throw new InvalidLocationException("Location
    is invalid for segment");
280             }
281             if(!stageStateUnfinished(stageId)){
282                 throw new InvalidStageStateException("Stage
    state is not acceptable");
283             }
284             if(getStageFromStageID(stageId).getType().equals(
    StageType.TT)){
285                 throw new InvalidStageTypeException("Not a
    valid stage type to add a climb too");
286             }
287 
288 
289             Stage stage = getStageFromStageID(stageId);
290             int segmentId = createNewID();
291             stage.addSegmentToSegmentHashmap(segmentId, new
    IntermediateSprint(segmentId,location, new ArrayList<>()));
292 
293             usedIds.add(segmentId);
294             return segmentId;
295         }
296 
297         @Override//tested
298         public void removeSegment(int segmentId) throws
    IDNotRecognisedException, InvalidStageStateException {
299             if(!idExists(segmentId)){
300                 throw new IDNotRecognisedException("This ID
    does not exist");
```

```
301             }
302             if(!stageStateUnfinished(getStageFromSegmentID(
    segmentId))){
303                 throw new InvalidStageStateException("Stage
    preparation has already been concluded");
304             }
305 
306             for (Race race:raceHashMap.values()) {
307                 for(Stage stage:race.getStages().values()){
308                     if(stage.getSegments().containsKey(
    segmentId)){
309                         stage.getSegments().remove(segmentId);
310                         removeId(segmentId);
311                         break;
312                     }
313                 }
314 
315             }
316 
317         }
318 
319         @Override //tested
320         public void concludeStagePreparation(int stageId)
    throws IDNotRecognisedException, InvalidStageStateException {
321             if(!idExists(stageId)){
322                 throw new IDNotRecognisedException();
323             }
324             if(!stageStateUnfinished(stageId)){
325                 throw new InvalidStageStateException("Stage
    preparation has already been concluded");
326             }
327             getStageFromStageID(stageId).setStageState("
    waiting for results");
328 
329         }
330 
331         @Override //tested
332         public int[] getStageSegments(int stageId) throws
    IDNotRecognisedException {
333             if(!idExists(stageId)){
334                 throw new IDNotRecognisedException();
335             }
336             ArrayList<Segment> segments = new ArrayList<>(
    getStageFromStageID(stageId).getSegments().values());
337 
338             return segmentInsertionSort(segments);
339 
340         }
```

```
341
342         @Override //test written
343         public int createTeam(String name, String description
    ) throws IllegalNameException, InvalidNameException {
344             if(teamNameIsIllegal(name)){
345                 throw new IllegalNameException("Team name is
    already taken");
346             }
347             if(nameIsInvalid(name)){
348                 throw new InvalidNameException();
349             }
350
351             int teamID = createNewID();
352             Team newTeam = new Team(name,teamID,description);
353             teamHashMap.put(teamID, newTeam);
354             usedIds.add(teamID);
355
356             return teamID;
357         }
358
359         @Override //tested
360         public void removeTeam(int teamId) throws
    IDNotRecognisedException {
361             if(!idExists(teamId)){
362                 throw new IDNotRecognisedException("Team ID
    does not exist");
363             }
364             teamHashMap.remove(teamId);
365             removeId(teamId);
366
367         }
368
369         @Override //tested
370         public int[] getTeams() {
371                 return teamHashMap.keySet().stream().mapToInt(
    Integer::intValue).toArray();
372         }
373
374         @Override //tested
375         public int[] getTeamRiders(int teamId) throws
    IDNotRecognisedException {
376             try{
377                 return teamHashMap.get(teamId).getTeamRiders
    ().keySet().stream().mapToInt(Integer::intValue).toArray();
378             }catch (Exception e){
379                 throw  new IDNotRecognisedException("Team does
     not exist");
380             }
```

```
381
382         }
383
384         @Override //tested
385         public int createRider(int teamID, String name, int
    yearOfBirth) throws IDNotRecognisedException,
    IllegalArgumentException {
386             if (name.equals("") || name == null || yearOfBirth
     < 1900){
387                 throw new IllegalArgumentException();
388             }
389             try {
390                 int riderId = createNewID();
391                 teamHashMap.get(teamID).addRiderToTeam(riderId
    , new Rider(name, yearOfBirth, riderId));
392                 usedIds.add(riderId);
393                 return riderId;
394             }catch (Exception e){
395                 throw new IDNotRecognisedException("The team
    does not exist");
396             }
397         }
398
399
400
401
402
403
404
405         //Needs to add to function so that all rider results
    are removed from other data in the system
406         @Override //tested
407         public void removeRider(int riderId) throws
    IDNotRecognisedException {
408             try {
409                 for (Team team : teamHashMap.values()) {
410                     if (team.getTeamRiders().containsKey(
    riderId)) {
411                         team.removeRiderFromTeam(riderId);
412                     }
413                 }
414                 for (Race race:raceHashMap.values()){
415                     for (Stage stage:race.getStages().values
    ()){
416                         for (Segment segment:stage.getSegments
    ().values()) {
417                             ArrayList<RiderResult>
    riderResults =  segment.getRidersResultsInSegment();
```

```
418                             for (RiderResult rider:
    riderResults) {
419                                 if(rider.getRiderId() ==
    riderId){
420                                     segment.
    getRidersResultsInSegment().remove(rider);
421                                     break;
422                                 }
423                             }
424                         }
425                     }
426                 }
427                 removeId(riderId);
428 
429             }catch(Exception e){
430                 throw new IDNotRecognisedException("The rider
    ID does not exist in a team");
431             }
432         }
433 
434 
435 
436 
437 
438 
439 
440 
441         @Override //tested
442         public void registerRiderResultsInStage(int stageId,
    int riderId, LocalTime... checkpoints) throws
    IDNotRecognisedException, DuplicatedResultException,
    InvalidCheckpointsException, InvalidStageStateException {
443             //DuplicateResultException is thrown in the
    stageResults class
444             if(! idExists(stageId)||! idExists(riderId)){
445                 throw new IDNotRecognisedException("An Id was
    not recognised");
446             }
447             Stage stage = getStageFromStageID(stageId);
448             if(stage.getSegments().size() + 2 != checkpoints.
    length){
449                 System.out.println("Stage has " + stage.
    getSegments().size() + " segments" );
450                 throw new InvalidCheckpointsException("
    Checkpoints length does not match the number of segments");
451             }
452             if(!stageStateWaitingForResults(stageId)){
453                 throw new InvalidStageStateException();
```

```
454             }
455             StageResults results =  stage.getStageResults();
456             if(results.getStageTimesObtained().containsKey(
    riderId)){
457                 throw new DuplicatedResultException();
458             }
459 
460             stage.GetSegmentAndAddRiderResults(riderId,
    checkpoints);
461             results.addResultToTimesObtained(riderId,
    checkpoints);
462 
463         }
464 
465         @Override //tested
466         public LocalTime[] getRiderResultsInStage(int stageId
    , int riderId) throws IDNotRecognisedException {
467             if(idExists(stageId) && idExists(riderId)){
468                 return  getStageFromStageID(stageId).
    getStageResults().returnTimeResults(riderId);
469             }else{
470                 throw new IDNotRecognisedException();
471             }
472         }
473 
474         @Override //tested
475         public LocalTime getRiderAdjustedElapsedTimeInStage(
    int stageId, int riderId) throws IDNotRecognisedException {
476             if (!idExists(stageId) || !idExists(riderId)) {
477                 throw new IDNotRecognisedException("Id not
    recognised");
478             }
479 
480             return getStageFromStageID(stageId).
    getStageResults().returnRiderIdAndAdjustedElapsedTime(riderId
    ).getFinishTime();
481 
482         }
483 
484         @Override
485         public void deleteRiderResultsInStage(int stageId, int
     riderId) throws IDNotRecognisedException {
486             try {
487                 getStageFromStageID(stageId).getStageResults
    ().removeRiderResult(riderId);
488             }catch (Exception e){
489                 throw new IDNotRecognisedException();
490             }
```

```
491         }
492 
493         ////////
494         @Override //tested
495         public int[] getRidersRankInStage(int stageId) throws
    IDNotRecognisedException {
496         if(!idExists(stageId)){
497             throw new IDNotRecognisedException();
498         }
499           return getStageFromStageID(stageId).getStageResults
    ().returnRankedRiderIdbyTime();
500         }
501 
502         @Override//need test
503         public LocalTime[]
    getRankedAdjustedElapsedTimesInStage(int stageId) throws
    IDNotRecognisedException {
504         if(!idExists(stageId)){
505             throw new IDNotRecognisedException();
506         }
507 
508             ArrayList<LocalTime> allRiderAdjustedElapsedTimes
     = new ArrayList<>();
509             int[] riderIdsInStage = getStageFromStageID(
    stageId).getStageResults().getStageTimesObtained().keySet().
    stream().mapToInt(Integer::intValue).toArray();
510             for( int riderId : riderIdsInStage) {
511                 allRiderAdjustedElapsedTimes.add(
    getStageFromStageID(stageId).getStageResults().
    returnRiderIdAndAdjustedElapsedTime(riderId).getFinishTime());
512             }
513             Collections.sort(allRiderAdjustedElapsedTimes);
514 
515             if(allRiderAdjustedElapsedTimes.size() > 0){
516                 return allRiderAdjustedElapsedTimes.toArray(
    LocalTime[]::new);
517             }else{
518                 return new LocalTime[0];
519             }
520 
521         }
522 
523         @Override//need test
524         public int[] getRidersPointsInStage(int stageId)
    throws IDNotRecognisedException {
525             if(!idExists(stageId)){
526                 throw new IDNotRecognisedException();
527             }
```

```
528             Stage stage = getStageFromStageID(stageId);
529             try{
530                 stage.getStageResults().clearAllPointsObtained
    ();
531                 stage.pushTotalSegmentPointsToStageResults();
532                 stage.pushFinishPointsToStageResults();
533                 return stage.getStageResults().
    getRankedRidersPointsInStage(stage.getStageResults().
    getStagePointsObtained());
534             }catch(Exception e) {
535                 return new int[0];
536             }
537         }
538 
539         @Override //need test
540         public int[] getRidersMountainPointsInStage(int
    stageId) throws IDNotRecognisedException {
541             if(!idExists(stageId)){
542                 throw new IDNotRecognisedException();
543             }
544             Stage stage = getStageFromStageID(stageId);
545             try{
546                 stage.getStageResults().clearAllPointsObtained
    ();
547                 stage.pushTotalSegmentPointsToStageResults();
548                 return stage.getStageResults().
    getRankedRidersPointsInStage(stage.getStageResults().
    getStageMountainPointsObtained());
549             }catch(Exception e) {
550                 return new int[0];
551             }
552         }
553 
554         @Override
555         public void eraseCyclingPortal() {
556             this.usedIds = new ArrayList<>();
557             this.teamHashMap = new HashMap<>();
558             this.raceHashMap = new HashMap<>();
559 
560         }
561 
562         @Override
563         public void saveCyclingPortal(String filename) throws
    IOException {
564                 try(ObjectOutputStream out = new
    ObjectOutputStream(new FileOutputStream(filename+".txt"))){
565                     out.writeObject(this);
566                 }catch (IOException ex){
```

```
567                     throw new IOException();
568                 }
569         }
570 
571         @Override
572         public void loadCyclingPortal(String filename) throws
    IOException, ClassNotFoundException {
573                 try(ObjectInputStream in = new
    ObjectInputStream(new FileInputStream(filename+".txt"))){
574                     Object obj = in.readObject();
575                     if(obj instanceof CyclingPortal){
576                         this.raceHashMap = ((CyclingPortal)
    obj).raceHashMap;
577                         this.teamHashMap = ((CyclingPortal)
    obj).teamHashMap;
578                         this.usedIds = ((CyclingPortal) obj).
    usedIds;
579                     }
580                 }catch(IOException exception){
581                     throw new IOException();
582                 }catch (ClassNotFoundException exception){
583                     throw new ClassNotFoundException();
584                 }
585         }
586 
587         @Override
588         public void removeRaceByName(String name) throws
    NameNotRecognisedException {
589         boolean found = false;
590             for (Race race:raceHashMap.values()) {
591                 if (race.getRaceName().equals(name)){
592                     raceHashMap.remove(race.getRaceId());
593                     found = true;
594                     break;
595                 }
596             }
597             if(!found){
598                 throw new NameNotRecognisedException("A race
    with that name does not exist");
599             }
600         }
601 
602         @Override
603         public LocalTime[] getGeneralClassificationTimesInRace
    (int raceId) throws IDNotRecognisedException {
604             if(!idExists(raceId)){
605                 throw new IDNotRecognisedException();
606             }
```

```
607
608             Race race = raceHashMap.get(raceId);
609             ArrayList<RiderResult> riderRankedArray = race.
    getGeneralClassificationInRace();
610
611             ArrayList<LocalTime> returnArray = new ArrayList
    <>();
612             for (RiderResult rider : riderRankedArray){
613                 returnArray.add(rider.getFinishTime());
614             }
615
616             try{
617                 return returnArray.toArray(LocalTime[]::new);
618             }catch(Exception e){
619                 return new LocalTime[0];
620             }
621
622
623         }
624
625         @Override
626         public int[] getRidersPointsInRace(int raceId) throws
    IDNotRecognisedException{
627             if(!idExists(raceId)){
628                 throw new IDNotRecognisedException();
629             }
630             Race race = raceHashMap.get(raceId);
631             return bubbleSortRiderResultsByTime(race.
    getRidersPointsOrMountainPointsInRace(false));
632         }
633
634         @Override
635         public int[] getRidersMountainPointsInRace(int raceId
    ) throws IDNotRecognisedException {
636             if(!idExists(raceId)){
637                 throw new IDNotRecognisedException();
638             }
639             Race race = raceHashMap.get(raceId);
640             try {
641                 return bubbleSortRiderResultsByTime(race.
    getRidersPointsOrMountainPointsInRace(true));
642             }catch(Exception e){
643                 return new int[0];
644             }
645
646         }
647
648         @Override
```

```
649         public int[] getRidersGeneralClassificationRank(int
    raceId) throws IDNotRecognisedException {
650             if(!idExists(raceId)){
651                 throw new IDNotRecognisedException();
652             }
653 
654             Race race = raceHashMap.get(raceId);
655             ArrayList<RiderResult> riderRankedArray = race.
    getGeneralClassificationInRace();
656 
657             ArrayList<Integer> returnArray = new ArrayList
    <>();
658             for (RiderResult rider : riderRankedArray){
659                 returnArray.add(rider.getRiderId());
660             }
661 
662             try{
663                 return returnArray.stream().mapToInt(Integer::
    intValue).toArray();
664             }catch(Exception e){
665                 return new int[0];
666             }
667 
668         }
669 
670         @Override
671         public int[] getRidersPointClassificationRank(int
    raceId) throws IDNotRecognisedException {
672             if (!idExists(raceId)) {
673                 throw new IDNotRecognisedException();
674             }
675             ArrayList<RiderResult> riderResultsArray = new
    ArrayList<>();
676             Race race = raceHashMap.get(raceId);
677             for (Stage stage : race.getStages().values()){
678                 StageResults results = stage.getStageResults
    ();
679                 int [] ridersIDs = results.
    returnRankedRiderIdbyTime();
680                 for (int riderId: ridersIDs){
681                     int riderPoints = results.
    returnRiderIdAndAdjustedElapsedTime(riderId).getPoints();
682                     boolean inArray = false;
683                     int points = results.
    getStagePointsObtained().get(riderId);
684                     for (RiderResult riderResult :
    riderResultsArray){
685                         if(riderResult.getRiderId() == riderId
```

```
685 ){
686                             riderResult.setPoints(riderResult.
    getPoints() + points);
687                             inArray = true;
688                             break;
689                         }
690                     }
691                     if(!inArray){
692                         riderResultsArray.add(new RiderResult(
    riderId,riderPoints));
693                     }
694                 }
695             }
696 
697             try{
698                 return bubbleSortRiderResultsByPoints(
    riderResultsArray);
699             }catch (Exception e){
700                 return new int[0];
701             }
702         }
703 
704         @Override
705         public int[] getRidersMountainPointClassificationRank(
    int raceId) throws IDNotRecognisedException {
706             if (!idExists(raceId)) {
707                 throw new IDNotRecognisedException();
708             }
709             ArrayList<RiderResult> riderResultsArray = new
    ArrayList<>();
710             Race race = raceHashMap.get(raceId);
711             for (Stage stage : race.getStages().values()){
712                 StageResults results = stage.getStageResults
    ();
713                 int [] ridersIDs = results.
    returnRankedRiderIdbyTime();
714                 boolean inArray = false;
715                 for (int riderId: ridersIDs){
716                     int riderPoints = results.
    returnRiderIdAndAdjustedElapsedTime(riderId).getPoints();
717                     int points = results.
    getStageMountainPointsObtained().get(riderId);
718                     for (RiderResult riderResult :
    riderResultsArray){
719                         if(riderResult.getRiderId() == riderId
    ){
720                             riderResult.setPoints(riderResult.
    getPoints() + points);
```

```
721                                 inArray = true;
722                                 break;
723                             }
724                         }
725                         if(!inArray){
726                             riderResultsArray.add(new RiderResult(
    riderId,riderPoints));
727                         }
728                     }
729                 }
730 
731                 try{
732                   return bubbleSortRiderResultsByPoints(
    riderResultsArray);
733                 }catch (Exception e){
734                     return new int[0];
735                 }
736             }
737 
738             //TEST METHODS
739             public int getRiderPointsInStage(int stageId , int
    riderId){
740             Stage stage = getStageFromStageID(stageId);
741                    return stage.getStageResults().
    getStagePointsObtained().get(riderId);
742             }
743             public int getRiderMountainPointsInStage(int stageId
     , int riderId){
744                 Stage stage = getStageFromStageID(stageId);
745                 return stage.getStageResults().
    getStageMountainPointsObtained().get(riderId);
746             }
747 
748 
749 }
750 
```

```java
package cycling;

import java.lang.reflect.Type;
import java.util.ArrayList;
import java.util.HashMap;

public class CategorisedClimb extends Segment{
    /**
     * A class to for if Categorised climbs
     *
     */
    // The segment type
    private SegmentType type;
    // The average Gradient of the climb
    private double averageGradient;
    // The length of the climb
    private double length;

    public CategorisedClimb(int segmentId, double location, SegmentType type, ArrayList<RiderResult> ridersResultsInSegment, double averageGradient, double length) {
        /**
         * A constructor make a new instance of a categorised climb
         * @param segmentId The segments ID
         * @param location The length of the segment
         * @param type The type that the segment is
         * @param ridersResultsInSegment The riders result's for the segment
         * @param averageGradient The average gradient for the climb
         * @param length The length of the climb
         */
        super(segmentId, location, ridersResultsInSegment);
        this.type = type;
        this.averageGradient = averageGradient;
        this.length = length;
    }

    public SegmentType getType() {
        /**
         * A function to get the type of the segment
         * @return type
         */
        return type;
    }

    public void setType(SegmentType type) {
```

```
44          /**
45           * A method to set the segment type
46           * @param type The type
47           */
48          this.type = type;
49      }
50
51      public double getAverageGradient() {
52          /**
53           * A function to get the average gradient of the climb
54           * @return averageGradient
55           */
56          return averageGradient;
57      }
58
59      public void setAverageGradient(double averageGradient) {
60          /**
61           * A function to set an average gradient
62           * @param averageGradient The average gradient
63           */
64          this.averageGradient = averageGradient;
65      }
66
67      public double getLength() {
68          /**
69           * A method to get the length of the climb
70           * @return length
71           */
72          return length;
73      }
74
75      public void setLength(double length) {
76          /**
77           * A method to set the length of the climb
78           * @param length The length to be set
79           */
80          this.length = length;
81      }
82
83      public HashMap<Integer , Integer>
    calculateCategorisedClimbPoints(){
84          /**A method to calculate the points of each rider in
    the segment in the climb
85           * @return returnHash, a hashmap of all riderId's and
    points achieved
86           */
87          int[] hardPointsForPosition = {20, 15, 12, 10, 8 , 6 ,4
     ,2};
```

```
88          int[] easyPointsFordPosition = {5 , 3 , 2 , 1};
89
90          HashMap<Integer, Integer> returnHash = new HashMap
    <>();
91          int count = this.type.equals(SegmentType.C1) ? 3 :
    this.type.equals(SegmentType.C3) ?  2 : this.type.equals(
    SegmentType.C4) ? 3 :  0;
92          try {
93
94              for (RiderResult riderResult :
    getRidersResultsInSegment()) {
95                  if (this.type.equals(SegmentType.HC)) {
96                      if (count < 8) {
97                          returnHash.put(riderResult.getRiderId
    (), hardPointsForPosition[count]);
98                          count++;
99                      }
100                 } else if (this.type.equals(SegmentType.C1)) {
101                     if (count < 8) {
102                         returnHash.put(riderResult.getRiderId
    (), hardPointsForPosition[count]);
103                         count++;
104                     }
105                 } else if (this.type.equals(SegmentType.C2)) {
106                     if (count < 4) {
107                         returnHash.put(riderResult.getRiderId
    (), easyPointsFordPosition[count]);
108                         count++;
109                     }
110                 } else if (this.type.equals(SegmentType.C3)) {
111                     if (count < 4) {
112                         returnHash.put(riderResult.getRiderId
    (), easyPointsFordPosition[count]);
113                         count++;
114                     }
115                 } else if (this.type.equals(SegmentType.C4)) {
116                     if (count < 4) {
117                         returnHash.put(riderResult.getRiderId
    (), easyPointsFordPosition[count]);
118                         count++;
119                     }
120                 }
121             }
122         }catch (Exception e){
123             System.out.println("Something went wrong with
    segment point collection in mountain race");
124             System.out.println(e);
125         }
```

```
126
127         return returnHash;
128     }
129
130
131 }
132
```

```java
package cycling;

import java.util.ArrayList;
import java.util.HashMap;

public class IntermediateSprint extends Segment{
    /**
     *A class for the intermediate sprints in a stage
     */
    public IntermediateSprint(int segmentId, double location, ArrayList<RiderResult> ridersResultsInSegment) {
        /**
         * A constructor for the class, to make a new instance of an intermediate sprint
         * @param segmentId The segments ID
         * @param location The distance of the race
         * @param ridersResultsInSegment The riders result's in the segment
         */
        super(segmentId, location, ridersResultsInSegment);

    }

    public HashMap<Integer , Integer> calculatePointsIntermediateSprint(){
        /**
         * A function to calculate the points obtained by each rider in the segment
         * @return returnHash, a hashmap of riderId and the points they obtained in the segment
         */
        int[] PointsForPosition = {20, 17, 15, 13, 11, 10 , 9 , 8 , 7 , 6 , 5 , 4 , 3 , 2 , 1};

        HashMap<Integer, Integer> returnHash = new HashMap<>();
        int count = 0;
        try {

            for (RiderResult riderSegResult : getRidersResultsInSegment()) {
                if(count < 14){
                    returnHash.put(riderSegResult.getRiderId() , PointsForPosition[count]);
                    count++;
                }
            }
        }catch (Exception e){
            System.out.println("Something went wrong with point
```

```
 collection in mountain race");
                System.out.println(e);
        }

        return returnHash;
    }
}

```

```java
package cycling;

import java.io.Serializable;
import java.time.LocalTime;
import java.util.ArrayList;
import java.util.HashMap;

public class Race implements Serializable {
    /**
     * Class to make the object Race, handles all the race attributes and methods.
     *Containing all stages in a race and the results for the race.
     */
    //Hashmap that contains all the riders present in a race, riderID and Rider Object
    HashMap<Integer,Rider> riders = new HashMap<>();
    //Hashmap that contains the stage objects and their stage IDs that are in the race
    HashMap<Integer, Stage> stages = new HashMap<>();
    //Race Name
    private String raceName;
    //Race ID
    private int raceId;
    //Race Description
    private String description;
    //Race results object

    //Need build constructors and getters and setters
    public Race(String raceName, int raceId, String description) {
        /**
         * Creates a race object.
         * @param raceName The name of the race.
         * @param raceId The ID of the race.
         * @param description The description of the race.
         */
        this.riders = new HashMap<Integer,Rider>();
        this.stages = new HashMap<Integer,Stage>();
        this.raceName = raceName;
        this.raceId = raceId;
        this.description = description;
    }

    @Override
    public String toString() {
        /**
         * Function to return the race objects attributes.
```

```
44          * @return A string that contains all the attributes of
    the race object.
45          */
46         double totalLength = 0;
47         for (Stage stage : stages.values()){
48             totalLength += stage.getStageLength();
49         }
50         return "\nRace:" +
51                 "  \nraceId =" + raceId +
52                 ", \nraceName ='" + raceName + '\'' +
53                 ", \ndescription ='" + description + '\'' +
54                 ", \nnumber of stages =" + stages.size() +
55                 ", \ntotal Length =" + totalLength ;
56     }
57
58     public String getRaceName(){
59         /**
60          * Function to return the races name.
61          * @return The race objects name attribute.
62          */
63         return this.raceName;
64     }
65     public int getRaceId(){
66         /**
67          * Function to return the races ID.
68          * @return The races ID.
69          */
70         return this.raceId;
71     }
72     public String getDescription(){
73         /**
74          * Function to return the race's description.
75          * @return The race description.
76          */
77         return this.description;
78     }
79     public void setRaceId(Integer RaceID){
80         /**
81          * Function to set the race ID of the race object.
82          * @param RaceID new race ID.
83          */
84         this.raceId = RaceID;
85     }
86     public void setDescription(String description){
87         /**
88          * Function to set the description of the race object.
89          * @param description The new description.
90          */
```

```
91          this.description =description;
92      }
93      public HashMap<Integer,Stage> getStages(){
94          /**
95           * Function to return the stages of a race.
96           * @return A hashmap of the stages contained in the
    race.
97           */
98          return this.stages;
99      }
100     public void addStageToStages(Stage stage){
101         /**
102          * Function to add a stage object to the hashmap of
    stages in the object.
103          * @param stage A new stage to be added to the stage
    attribute.
104          */
105         this.stages.put(stage.getStageId(), stage);
106     }
107     public ArrayList<RiderResult>
    getRidersPointsOrMountainPointsInRace(boolean isMountainPoints
    ){
108         /**
109          * A function to get all riders in the race, points or
     mountain points.
110          * @param isMountainPoints Boolean to whether to
    return the mountain points or just normal points.
111          * @return riderResultsArray, an array of objects and
    containing riderId, and races points.
112          */
113         ArrayList<RiderResult> riderResultsArray = new
    ArrayList<>();
114
115         for (Stage stage : this.getStages().values()){
116             stage.getStageResults().clearAllPointsObtained();
117             stage.pushTotalSegmentPointsToStageResults();
118             stage.pushFinishPointsToStageResults();
119
120             StageResults results = stage.getStageResults();
121             int [] ridersIDs = results.
    returnRankedRiderIdbyTime();
122             for (int riderId: ridersIDs){
123
124                 ArrayList<LocalTime> timeArrayList = results.
    getStageTimesObtained().get(riderId);
125                 LocalTime finishTime = timeArrayList.get(
    timeArrayList.size() -1);
126                 int points = 0;
```

```
127
128                 if(!isMountainPoints){
129                     points = results.getStagePointsObtained().
    get(riderId);
130                 }else if(isMountainPoints){
131                     if(results.getStageMountainPointsObtained
    ().containsKey(riderId)){
132                         points = results.
    getStageMountainPointsObtained().get(riderId);
133                     }else{
134                         points = 0;
135                     }
136                 }
137                 boolean inArray = false;
138                 for (RiderResult riderResult :
    riderResultsArray){
139                     if(riderResult.getRiderId() == riderId){
140                         //Adding points
141                         riderResult.setPoints(riderResult.
    getPoints() + points);
142                         //creating the new time
143                         LocalTime originalTime = riderResult.
    getFinishTime();
144                         LocalTime newTime = originalTime.
    plusHours(finishTime.getHour())
145                                 .plusMinutes(finishTime.
    getMinute()).plusSeconds(finishTime.getSecond());
146                         riderResult.setFinishTime(newTime);
147                         inArray = true;
148                         break;
149                     }
150                 }
151                 if(!inArray){
152                     riderResultsArray.add(new RiderResult(
    riderId,finishTime,points));
153                 }
154                 System.out.println(riderId + " has "+ points
     + " mountain points at the end of this stage");
155             }
156         }
157         return riderResultsArray;
158     }
159     public ArrayList<RiderResult>
    getGeneralClassificationInRace(){
160         /**
161          * Function that creates an ArrayList of RiderResult
    objects and accumulates each riders elapsed times from all
    stages
```

```
162          * @return RiderResult arraylist sorted by time
163          */
164
165         ArrayList<RiderResult> riderRankedArray = new
    ArrayList<>();
166
167         for (Stage stage : this.getStages().values()){
168             StageResults stageResults = stage.getStageResults
    ();
169             int [] ridersIDs = stageResults.
    returnRankedRiderIdbyTime();
170             for (int riderId: ridersIDs){
171                 boolean inArray = false;
172                 LocalTime elapsedTime = stageResults.
    returnRiderIdAndAdjustedElapsedTime(riderId).getFinishTime();
173                 for (RiderResult riderResult :
    riderRankedArray){
174                     if(riderResult.getRiderId() == riderId){
175                         //creating the new time
176                         LocalTime originalTime = riderResult.
    getFinishTime();
177                         LocalTime newTime = originalTime.
    plusHours(elapsedTime.getHour())
178                                 .plusMinutes(elapsedTime.
    getMinute()).plusSeconds(elapsedTime.getSecond());
179                         riderResult.setFinishTime(newTime);
180                         inArray = true;
181                         break;
182                     }
183                 }
184                 if(!inArray){
185                     riderRankedArray.add(new RiderResult(
    riderId, elapsedTime));
186                 }
187             }
188         }
189
190         try{
191             return bubbleSortRiderResultsByTime(
    riderRankedArray);
192         }catch(Exception e){
193             return new ArrayList<RiderResult>();
194         }
195     }
196
197     private ArrayList<RiderResult>
    bubbleSortRiderResultsByTime(ArrayList<RiderResult> arrayList
    ){
```

```
198         /**
199          * Function to perform a bubble sort on an arraylist of RiderResult objects based on the finish time within the object
200          * @return A sorted arraylist of RiderResult
201          * @param A RiderResult arraylist
202          */
203         for (int i = 0; i < arrayList.size(); i++) {
204             for (int j = i + 1; j < arrayList.size(); j++) {
205                 if (arrayList.get(i).getFinishTime().isAfter(arrayList.get(j).getFinishTime())) {
206                     RiderResult R1 = arrayList.get(j);
207                     arrayList.set(j, arrayList.get(i));
208                     arrayList.set(i, R1);
209                 }
210             }
211         }
212 
213         return arrayList;
214 
215     }
216 
217 
218 }
219 
```

```java
package cycling;

import java.io.Serializable;

public class Rider implements Serializable {
    /**
     * Class to create the object Rider that
     *contains each rider's information in the cycling portal.
     */
    private String name;    //Riders name
    private int yearOfBirth; //Riders year of birth
    private int riderId; //Riders unique ID

    public Rider(String name, int yearOfBirth, int riderId) {
        /**
         * A method to create a new Rider instance.
         * @param name The name of the new Rider.
         * @param yearOfBirth The date of birth for the new rider.
         * @param riderId The new ID of the rider.
         */
        this.name = name;
        this.yearOfBirth = yearOfBirth;
        this.riderId = riderId;
    }

    public String getName() {
        /**
         * A function to return the name of the rider.
         * @return The name of the rider.
         */
        return name;
    }

    public void setName(String name) {
        /**
         * A method to set the name of the rider.
         * @param name The name of the rider.
         */
        this.name = name;
    }

    public int getYearOfBirth() {
        /**
         * A function to return a rider's year of birth.
         * @return The rider's year of birth.
         */
        return yearOfBirth;
```

```
48      }
49
50      public void setYearOfBirth(int yearOfBirth) {
51          /**
52           * A method to set the year of birth for the rider.
53           * @param yearOfBirth The rider's year of birth.
54           */
55          this.yearOfBirth = yearOfBirth;
56      }
57
58      public int getRiderId() {
59          /**
60           * A function to return the rider ID of the rider.
61           * @return The riders ID.
62           */
63          return riderId;
64      }
65
66      public void setRiderId(int riderId) {
67          /**
68           * A method to set the rider ID of the rider.
69           * @param riderId The id to be set.
70           */
71          this.riderId = riderId;
72      }
73  }
74
```

```java
package cycling;

import java.io.Serializable;
import java.time.LocalTime;

public class RiderResult implements Serializable {
    /**
     * A class to create the rider result object.
     *This object is used to hold all data for a riders results in a race and its stages
     */
    // The rider ID
    private int riderId;
    // The riders finish time
    private LocalTime FinishTime;
    // The riders point's
    private int points;

    public RiderResult(int riderId, LocalTime segmentFinishTime) {
        /**
         * A method to create an instance of the RiderResult class.
         * @param riderId The ID of the rider.
         * @param segmentFinishTime The segment finish time of the rider.
         */
        this.riderId = riderId;
        this.FinishTime = segmentFinishTime;
    }
    public RiderResult(int riderId, LocalTime segmentFinishTime, int points) {
        /**A method to create an instance of the RiderResult class.
         * @param riderId The ID of the rider.
         * @param segmentFinishTime The segment finish time of the rider.
         * @param points The points obtained per segment.
         */
        this.riderId = riderId;
        this.FinishTime = segmentFinishTime;
        this.points = points;
    }
    public RiderResult(int riderId, int points) {
        /**A method to create an instance of the RiderResult class.
         * @param riderId The ID of the rider.
         * @param points The points obtained per segment.
```

```
41          */
42         this.riderId = riderId;
43         this.points = points;
44     }
45
46
47     public void setRiderId(int riderId) {
48         /**
49          * A method to set the rider ID.
50          * @param The rider ID.
51          */
52         this.riderId = riderId;
53     }
54
55     public void setPoints(int points) {
56         /**
57          * A method to set the points obtained by the rider.
58          * @param points The points achieved.
59          */
60         this.points = points;
61     }
62
63     public void setFinishTime(LocalTime finishTime) {
64         /**
65          * A method to set the finish time of a rider.
66          * @param finishTime The time the rider finished in.
67          */
68         FinishTime = finishTime;
69     }
70
71     public int getPoints() {
72         /**
73          * A function to get the points achieved by the rider.
74          * @return The points of the rider.
75          */
76         return points;
77     }
78
79     public int getRiderId() {
80         /**
81          * A function to return the riders ID.
82          * @return The riders ID.
83          */
84         return riderId;
85     }
86
87     public LocalTime getFinishTime() {
88         /**
```

```
89          * A function to return the finish time of the rider.
90          * @return The riders finish time.
91          */
92         return FinishTime;
93     }
94 }
95
```

```java
package cycling;
import java.io.Serializable;
import java.util.ArrayList;

public class Segment implements Serializable {
    /**
     * A class for the segments in each stage.
     *This class also contains each result for each rider in the segment
     */
    //The segment ID
    private int segmentId;
    //The segment location
    private double location;
    //An array list of all the riders results in the segment
    private ArrayList<RiderResult> ridersResultsInSegment = new ArrayList<>();

    public void sortRiderResultsInSegment(){
        /**
         * A method that will sort the riders results in the segment in order of time.
         *
         */
        for (int i = 0; i < ridersResultsInSegment.size(); i++) {
            for (int j = i + 1; j < ridersResultsInSegment.size(); j++) {
                if (ridersResultsInSegment.get(i).getFinishTime().isAfter(ridersResultsInSegment.get(j).getFinishTime())) {
                    RiderResult temp = ridersResultsInSegment.get(j);
                    ridersResultsInSegment.set(j, ridersResultsInSegment.get(i));
                    ridersResultsInSegment.set(i, temp);

                }
            }
        }
    }

    public Segment(int segmentId, double location, ArrayList<RiderResult> ridersResultsInSegment) {
        /**
         * A constructor to create an instance of a segment
         * @param segmentId The segment ID
         * @param location The location of the segment
         * @param ridersResultsInSegment The rider results in
```

```
39 the segment
40          */
41         this.segmentId = segmentId;
42         this.location = location;
43         this.ridersResultsInSegment = ridersResultsInSegment;
44     }
45 
46     public ArrayList<RiderResult> getRidersResultsInSegment() {
47         /**
48          * A function to return the riders results in the
   segment
49          * @return ridersResultsInSegment
50          */
51         return ridersResultsInSegment;
52     }
53 
54     public int getSegmentId() {
55         /**
56          * A function to return the segment ID
57          * @return segmentId
58          */
59         return segmentId;
60     }
61 
62     public double getLocation() {
63         /**
64          * A function to return the location of the segment
65          * @return location
66          */
67         return location;
68     }
69 
70     public void setLocation(double location) {
71         /**
72          * A method to set the location of the segment
73          * @param location The location of the segment
74          */
75         this.location = location;
76     }
77 
78 
79 }
80 
```

```java
package cycling;

/**
 * This enum is used to represent the segment types within stages on road races.
 *
 * @author Diogo Pacheco
 * @version 1.0
 *
 */
public enum SegmentType {
    /**
     *An enum class for the types of segment
     *
     */

    /**
     * An intermediate sprint.
     */
    SPRINT,

    /**
     * A categorised 4 climb. The easiest categorised climbs of all, under 2km long
     * with an average grade of around 5% or 2-3% up to 5km long.
     */
    C4,

    /**
     * A categorised 3 climb. This could be a climb as short as 1km with a steep
     * gradient of about 10% or a mellower climb up to 10km long with up to a 5%
     * gradient.
     */
    C3,

    /**
     * A categorised 2 climb. Category 2 could be a short climb, for example 5km at
     * 8 percent, or as long as 15km at 4. percent
     */
    C2,

    /**
     * A categorised 1 climb. Still a very significant climb, it could be a big
```

```
42      * mountain climb with a lesser gradient or a shorter climb
    with a steep pitch,
43      * for example 8km at 8% through to 20km at 5%.
44      */
45     C1,
46
47     /**
48      * From the French term "Hors Categorie" (HC) meaning
   beyond categorisation. The
49      * toughest of the tough. The longest or steepest climbs,
   often both combined.
50      */
51     HC;
52 }
53
```

```java
package cycling;

import cycling.RiderResult;
import java.util.ArrayList;

public interface Sorts {
    /**
     *A class that performs sorts on data in the cycling portal
     *Sorting times and scores
     * @param arrayList
     * @return
     */
    default int[] bubbleSortRiderResultsByTime(ArrayList<RiderResult> arrayList){
        //Function to perform a bubble sort on the bases of time. On an array list of RiderResult objects
        for (int i = 0; i < arrayList.size(); i++) {
            for (int j = i + 1; j < arrayList.size(); j++) {
                if (arrayList.get(i).getFinishTime().isAfter(arrayList.get(j).getFinishTime())) {
                    RiderResult R1 = arrayList.get(j);
                    arrayList.set(j, arrayList.get(i));
                    arrayList.set(i, R1);
                }
            }
        }
        ArrayList<Integer> returnArray = new ArrayList<>();
        for (RiderResult rider : arrayList){
            returnArray.add(rider.getPoints());
        }
        return returnArray.stream().mapToInt(Integer::intValue).toArray();
    }
    default int[] bubbleSortRiderResultsByPoints(ArrayList<RiderResult> arrayList) {
        //Function to perform a bubble sort on the bases of points. On an array list of RiderResult objects
        for (int i = 0; i < arrayList.size(); i++) {
            for (int j = i + 1; j < arrayList.size(); j++) {
                if (arrayList.get(i).getPoints() > (arrayList.get(j).getPoints())) {
                    RiderResult R1 = arrayList.get(j);
                    arrayList.set(j, arrayList.get(i));
                    arrayList.set(i, R1);
                }
            }
        }
        ArrayList<Integer> returnArray = new ArrayList<>();
```

```
42          for (RiderResult rider : arrayList){
43              returnArray.add(rider.getRiderId());
44          }
45          return returnArray.stream().mapToInt(Integer::intValue
    ).toArray();
46      }
47      default int[] segmentInsertionSort(ArrayList<Segment>
    arrayList){
48          //A function to perform an insertion sort
49          for (int i = 1; i < arrayList.size(); i++) {
50              double key = arrayList.get(i).getLocation();
51              int j = i - 1;
52              while(j >= 0 && key < arrayList.get(j).getLocation
    ()) {
53                  Segment temp = arrayList.get(j);
54                  arrayList.set(j, arrayList.get(j + 1));
55                  arrayList.set(j + 1, temp);
56                  j--;
57              }
58          }
59          ArrayList<Integer> returnArrayList = new ArrayList<>();
60          for (int i = 0; i < arrayList.size(); i++) {
61              returnArrayList.add(arrayList.get(i).getSegmentId
    ());
62          }
63          return returnArrayList.stream().mapToInt(Integer::
    intValue).toArray();
64      }
65  }
66
```

```java
package cycling;

import java.io.Serializable;
import java.time.Duration;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.HashMap;

public class Stage implements Serializable, Sorts {
    /**
     *A class to create a stage object.
     *cContaining all the segments in the stage and methods to calculate the results for this stage.
     */
    // The stages unique ID
    private int stageId;
    //The stages name
    private String stageName;
    //The stages distance
    private double stageLength;
    //The type of stage the stage is
    private StageType type;
    //The start time of the stage in the race
    private LocalDateTime startTime;
    //The segment objects in the stage
    private HashMap<Integer, Segment> segments;
    //A description of the stage
    private String description;
    //The Stages state
    private String stageState;
    //The results of the stage contained in an object
    private StageResults stageResults;

    //public Stage(int stageId, String stageName, double stageLength, StageType type, LocalDateTime startTime, HashMap<Integer, Segment> segments, String description, String stageState, StageResults stageResults) {
    public Stage(int stageId, String stageName, double stageLength, StageType type, LocalDateTime startTime, String description) {
        /**
         * A constructor to make a new stage instance
         * @param stageId The id of the stage
         * @param stageName The name of the stage
```

```
43          * @param stageLength The distance of the stage in the
    race
44          * @param type The stages type
45          * @param startTime The start time of the stage in the
    race
46          * @param Description A description of the stage
47          */
48         this.stageId = stageId;
49         this.stageName = stageName;
50         this.stageLength = stageLength;
51         this.type = type;
52         this.startTime = startTime;
53         this.segments = new HashMap<Integer, Segment>();
54         this.description = description;
55         this.stageState = "Unfinished";
56         this.stageResults = new StageResults();
57     }
58 
59     public String getStageName(){
60         /**
61          * A function to get the stages name
62          * @return stageName
63          */
64         return stageName;
65     }
66 
67     public int getStageId() {
68         /**
69          * A function to get the ID of the stage
70          * @return stageId
71          */
72         return stageId;
73     }
74 
75     public void setStageId(int stageId) {
76         /**
77          * A method to set the id of the stage
78          * @param stageId The stage ID
79          */
80         this.stageId = stageId;
81     }
82 
83     public double getStageLength() {
84         /**
85          * A function to return the length of the stage
86          * @return stageLength
87          */
88         return stageLength;
```

```
89      }
90
91      public void setStageLength(int stageLength) {
92          /**
93           * A method to set the length of the stage
94           * @param stageLength The stage length
95           */
96          this.stageLength = stageLength;
97      }
98
99      public StageType getType() {
100         /**
101          * A function to return the stage type
102          * @return type
103          */
104         return type;
105     }
106
107     public void setType(StageType type) {
108         /**
109          * A method to set the stages type
110          * @param type The stages type
111          */
112         this.type = type;
113     }
114
115     public LocalDateTime getStartTime() {
116         /**
117          * A function to return the start time of the stage
118          * @return startTime
119          */
120         return startTime;
121     }
122
123     public void setStartTime(LocalDateTime startTime) {
124         /**
125          * A method to set the start time of the stage
126          * @param startTime The start time of the stage in the
    race
127          */
128         this.startTime = startTime;
129     }
130
131     public HashMap<Integer, Segment> getSegments() {
132         /**
133          * A function to return the segments in the stage
134          * @return segments
135          */
```

```
136         return segments;
137     }
138 
139     public void setSegments(HashMap<Integer,Segment> segments
    ) {
140         /**
141          * A method to set the segments of the stage
142          * @param segments The segments of the stage
143          */
144         this.segments = segments;
145     }
146 
147     public String getDescription() {
148         /**
149          * A function to return the description of the stage
150          * @return description
151          */
152         return description;
153     }
154 
155     public void setDescription(String description) {
156         /**
157          * A method to set the description of the stage
158          * @param description The stages description
159          */
160         this.description = description;
161     }
162 
163     public String getStageState() {
164         /**
165          * A function to return the stages state
166          * @return stageState
167          */
168         return stageState;
169     }
170 
171     public void setStageState(String stageState) {
172         /**
173          * A method to set the state of the stage
174          * @param stageState The new state of the stage
175          */
176         this.stageState = stageState;
177     }
178 
179     public StageResults getStageResults() {
180         /**
181          * A function to get the results of the stage
182          * @return stageResults
```

```
183          */
184         return stageResults;
185     }
186 
187     public void setStageResults(StageResults stageResults) {
188         /**
189          * A method to set the results of the stage
190          * @param stageResults The results of the stage
191          */
192         this.stageResults = stageResults;
193     }
194 
195     public void addSegmentToSegmentHashmap(Integer segmentId
     , Segment segment){
196         /**
197          * A method to add a new segment to the segment
    hashmap
198          * @param segmentId The segments ID
199          * @param segment The segment object
200          */
201         this.segments.put(segmentId, segment);
202     }
203 
204     //Used to obtain segment points
205     public void GetSegmentAndAddRiderResults(int riderId,
    LocalTime[] checkpoints){
206         /**
207          * A function to get the results for each segment
    object
208          * @param riderId The riders id
209          * @param checkpoints The checkpoints wanted
210          */
211         Collection<Segment> collection = segments.values();
212         ArrayList<Segment> segmentArrayList = new ArrayList<>(
    collection);
213         ArrayList<LocalTime> checkpointsArrayList= new
    ArrayList<>(Arrays.asList(checkpoints));
214 
215         checkpointsArrayList.remove(0);
216 
217             for (int i = 0; i < checkpointsArrayList.size() ;
    i++) {
218                 if (i + 1 < checkpointsArrayList.size()) {
219                     LocalTime segmentTime =
    checkpointsArrayList.get(i);
220                     segmentArrayList.get(i).
    getRidersResultsInSegment().add(new RiderResult(riderId,
    segmentTime));
```

```
221                 }
222             }
223     }
224 
225     public void pushTotalSegmentPointsToStageResults(){
226         /**
227          * A method to push the segment points to the stage
    results attribute
228          *
229          */
230         for(Segment segment : segments.values()){
231             segment.sortRiderResultsInSegment();
232             if(segment instanceof CategorisedClimb){
233                 HashMap<Integer, Integer> riderIdAndPoints
     = ((CategorisedClimb) segment).
    calculateCategorisedClimbPoints(); //Hashmap of the  rider Id
    and points earned in the segment
234                 riderIdAndPoints.forEach((key , value) -> {
235                     if(this.stageResults.
    getStageMountainPointsObtained().containsKey(key)){
236                         stageResults.
    getStageMountainPointsObtained().put(key, stageResults.
    getStageMountainPointsObtained().get(key) + value);
237                     }else{
238                         stageResults.
    getStageMountainPointsObtained().put(key, value);
239                     }
240                 });
241             }else if(segment instanceof  IntermediateSprint){
242                 HashMap<Integer, Integer> riderIdAndPoints
     = ((IntermediateSprint) segment).
    calculatePointsIntermediateSprint();
243                 riderIdAndPoints.forEach((key , value) -> {
244                     if(this.stageResults.
    getStagePointsObtained().containsKey(key)){
245                         stageResults.getStagePointsObtained().
    put(key, stageResults.getStagePointsObtained().get(key) +
    value);
246                     }else{
247                         stageResults.getStagePointsObtained().
    put(key, value);
248                     }
249                 });
250             }
251 
252         }
253     }
254 
```

```
255     public void pushFinishPointsToStageResults() {
256         /**
257          * A method to calculate the riders points based of
    their finishing position in the stage and
258          * put each rider's finish points in the stage results
259          */
260         int[] flatFinish = {50,30,20,18,16,14,12,10,8,7,6,5,4,
    3,2};
261         int[] hillyFinishOrMediumMountainFinish = {30,25,22,19
    ,17,15,13,11,9,7,6,5,4,3,2};
262         int[] highMountainOrIndividualTimeTrialFinish = {20,17
    ,15,13,11,10,9,8,7,6,5,4,3,2,1};
263 
264         HashMap<Integer, Integer> riderIdAndPoints = new
    HashMap<>();
265         int count = 0;
266         int[] ridersRankByTime = this.stageResults.
    returnRankedRiderIdbyTime();
267 
268         try {
269             for (int rider : ridersRankByTime) {
270                 if (this.type.equals(StageType.FLAT)) {
271                     if (count < 14) {
272                         riderIdAndPoints.put(rider, flatFinish
    [count]);
273                         count++;
274                     }
275                 } else if (this.type.equals(StageType.
    MEDIUM_MOUNTAIN)) {
276                     if (count < 14) {
277                         riderIdAndPoints.put(rider,
    hillyFinishOrMediumMountainFinish[count]);
278                         count++;
279                     }
280                 } else if (this.type.equals(StageType.
    HIGH_MOUNTAIN) || this.type.equals(StageType.TT)) {
281                     if (count < 14) {
282                         riderIdAndPoints.put(rider,
    highMountainOrIndividualTimeTrialFinish[count]);
283                         count++;
284                     }
285                 }
286             }
287         }catch(Exception e){
288             System.out.println("Something went wrong
    collecting stage points");
289             System.out.println(e);
290         }
```

```
291
292         riderIdAndPoints.forEach((key , value) -> {
293             if(this.stageResults.getStagePointsObtained().
    containsKey(key)){
294                 stageResults.getStagePointsObtained().put(key
    , stageResults.getStagePointsObtained().get(key) + value);
295             }else{
296                 stageResults.getStagePointsObtained().put(key
    , value);
297             }
298         });
299
300     }
301
302     @Override
303     public String toString() {
304         /**
305          * A function to return the attributes of a stage
    instance in a string form
306          * @return All the classes attributes
307          */
308         return "Stage{" +
309                 "stageId=" + stageId +
310                 ", stageName='" + stageName + '\'' +
311                 ", stageLength=" + stageLength +
312                 ", type=" + type +
313                 ", startTime=" + startTime +
314                 ", segments=" + segments +
315                 ", description='" + description + '\'' +
316                 ", stageState='" + stageState + '\'' +
317                 ", stageResults=" + stageResults +
318                 '}';
319     }
320
321
322 }
323
324
```

```java
package cycling;

import java.io.Serializable;
import java.time.Duration;
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;
import java.time.temporal.ChronoUnit;
import java.util.ArrayList;
import java.util.HashMap;

public class StageResults implements Serializable {
    /**
     * A class for the results of a stage.
     *Contains all the results for a stage allowing them to be added and calculated.
     */
    // A hashmap for the stage times obtained by the riders
    private HashMap<Integer, ArrayList<LocalTime>> stageTimesObtained = new HashMap<Integer, ArrayList<LocalTime>>();
    // A hashmap for the stage points obtained by the riders
    private HashMap<Integer, Integer> stagePointsObtained = new HashMap<Integer, Integer>();
    // A hashamp for the stages mountain points obtained by the ridsers
    private HashMap<Integer, Integer> stageMountainPointsObtained = new HashMap<Integer, Integer>();


    public void clearAllPointsObtained(){
        /**
         * A method to clear the points obtained in the hashmaps above
         *
         */
        stagePointsObtained.clear();
        stageMountainPointsObtained.clear();
    }
    public void bubbleSort (ArrayList < LocalTime > arrayList ) {
        /**
         * A method to perform a bubble sort on an arraylist containing times
         * @param arrayList
         */
        for (int i = 0; i < arrayList.size(); i++) {
            for (int j = i + 1; j < arrayList.size(); j++) {
                if (arrayList.get(i).isAfter(arrayList.get(j
```

```
39 ))) {
40                     LocalTime holder = arrayList.get(j);
41                     arrayList.set(j, arrayList.get(i));
42                     arrayList.set(i, holder);
43                 }
44             }
45         }
46     }
47 
48 
49     public StageResults() {
50         /**
51          * A function to return the results of the stage
52          * @return stageTimesObtained
53          */
54         this.stageTimesObtained = stageTimesObtained;
55     }
56 
57     public HashMap<Integer, Integer> getStagePointsObtained() {
58         /**
59          * A function to return the stage points obtained
60          * @return stagePointsObtained
61          */
62         return stagePointsObtained;
63     }
64 
65     public HashMap<Integer, Integer>
   getStageMountainPointsObtained() {
66         /**
67          * A function to return the mountain points obtained in
    the stages results
68          * @return stageMountainPointsObtained
69          */
70         return stageMountainPointsObtained;
71     }
72 
73     public void addResultToTimesObtained(int RiderID ,
   LocalTime[] time) throws DuplicatedResultException {
74         /**
75          * A method to add a result to the times obtained
   hashamp
76          * @param RiderID The ID of the rider
77          * @param Time The time they achieved in the stage
78          */
79         if(stageTimesObtained.containsKey(RiderID)){
80             throw new DuplicatedResultException();
81         }else{
82             ArrayList<LocalTime> times = new ArrayList<
```

```
 82 LocalTime>();
 83             for (int i = 0; i < time.length ; i++) {
 84                 times.add(time[i]);
 85             }
 86             //Adding the elapsed time to the arrayList
 87             LocalTime startTime = times.get(0);
 88             LocalTime finishTime = times.get(times.size()-1);
 89             Long difference = Duration.between(startTime,
    finishTime).toSeconds();
 90             String hours   = String.valueOf( (int) Math.floor(
    difference / 3600));
 91             String minutes = String.valueOf((int)Math.floor((
    difference - (Integer.parseInt(hours) * 3600)) / 60));
 92             String seconds = String.valueOf((int) Math.floor(
    difference - (Integer.parseInt(hours) * 3600) - (Integer.
    parseInt(minutes) * 60)));
 93
 94             if (Integer.valueOf(hours)   < 10) {hours   = "0"+
    hours;}
 95             if (Integer.valueOf(minutes) < 10) {minutes = "0"
     + minutes;}
 96             if (Integer.valueOf(seconds) < 10) {seconds = "0"+
    seconds;}
 97
 98             DateTimeFormatter formatter = DateTimeFormatter.
    ofPattern("HH:mm:ss");
 99             LocalTime appendElapsedTime = LocalTime.parse(
    String.valueOf(hours)+":"+String.valueOf(minutes)+":"+String.
    valueOf(seconds) , formatter);
100
101             times.add(appendElapsedTime);
102
103             stageTimesObtained.put(RiderID , times);
104         }
105     }
106
107     public HashMap<Integer, ArrayList<LocalTime>>
    getStageTimesObtained() {
108         /**
109          * A function to return the times obtained in its
    stage
110          * @return stageTimesObtained
111          */
112         return stageTimesObtained;
113     }
114
115     public LocalTime[] returnTimeResults(int riderId){
116         /**
```

```
117          * A function to return a riders time in the stage
118          * @param riderId The rider ID
119          * @return LocalTime[0]
120          */
121         if(stageTimesObtained.containsKey(riderId)){
122             return (LocalTime[]) stageTimesObtained.get(
    riderId).stream().toArray(LocalTime[]::new);
123         }else{
124             return new LocalTime[0];
125         }
126 
127     }
128 
129     public int[] getRankedRidersPointsInStage(HashMap<Integer
    , Integer> stagePoints){
130         /**
131          * A function to return the riders rank in the stage
132          * @param stagePoints The riders points in the stage
133          * @return int[] of rankedListOfPoints
134          */
135         ArrayList<Integer> rankedListOfPoints = new ArrayList
    <>();
136         int[] rankedIds = returnRankedRiderIdbyTime();
137         for(Integer rankedId : rankedIds){
138             rankedListOfPoints.add(stagePoints.get(rankedId));
139         }
140 
141         return rankedListOfPoints.stream().mapToInt(Integer::
    intValue).toArray();
142     }
143 
144     public void removeRiderResult(int riderId){
145         /**
146          * A method to remove a riders results from the stage
147          * @param riderId The rider being removed
148          */
149         stageTimesObtained.remove(riderId);
150 
151     }
152 
153     public int[] returnRankedRiderIdbyTime(){
154         /**
155          * A function to return the rank of all riders in the
    stage on the bases of time
156          * @return int[] of riders ranks ordered
157          */
158         HashMap<Integer, LocalTime> returnMap = new HashMap
    <>();
```

```
159         stageTimesObtained.forEach((key, value) ->{
160             returnMap.put(key, value.get(value.size() - 1));
161         });
162 
163 
164         for (Integer name: returnMap.keySet()) {
165             String key = name.toString();
166             String value = returnMap.get(name).toString();
167         }
168 
169         ArrayList<Integer> returnArray = new ArrayList<>();
170 
171         ArrayList<LocalTime> allRiderTimes = new ArrayList
    <>();
172         ArrayList<Integer> allRiderIds = new ArrayList<>();
173 
174         for(LocalTime time : returnMap.values()){
175             allRiderTimes.add(time);
176         }
177         for (Integer ids : returnMap.keySet()){
178             allRiderIds.add(ids);
179         }
180 
181         for (int i = 0; i < allRiderTimes.size(); i++) {
182             for (int j = i + 1; j < allRiderTimes.size(); j
    ++) {
183                 if (allRiderTimes.get(i).isAfter(allRiderTimes
    .get(j))) {
184 
185                     LocalTime timeHolder = allRiderTimes.get(j
    );
186                     Integer IdHolder = allRiderIds.get(j);
187 
188                     allRiderTimes.set(j, allRiderTimes.get(i
    ));
189                     allRiderIds.set(j, allRiderIds.get(i));
190 
191                     allRiderTimes.set(i, timeHolder);
192                     allRiderIds.set(i , IdHolder);
193                 }
194             }
195         }
196 
197         return allRiderIds.stream().mapToInt(Integer::intValue
    ).toArray();
198     }
199 
200     public RiderResult returnRiderIdAndAdjustedElapsedTime(int
```

```
200  riderId) {
201          /**
202           * A function to return the riderId and their adjusted
     elapsed time in the stage
203           * @param riderId the riders ID
204           * @return RiderResult
205           */
206          ArrayList<LocalTime> allRiderTimes = new ArrayList
    <>();
207
208          //The riders time which we wish to adjust
209          LocalTime[] riderTimes = this.returnTimeResults(
    riderId);
210          LocalTime riderFinishTimeInStage = riderTimes[
    riderTimes.length - 1]; //get the finish time in the stage for
     a rider
211
212          //Retrieving all other rider
213          for (ArrayList<LocalTime> times : this.
    getStageTimesObtained().values()) {
214              allRiderTimes.add(times.get(times.size() - 1));
215          }
216
217
218          ArrayList<LocalTime> allRiderTimesV2 = new ArrayList
    <>();
219          for (int i = 0; i < allRiderTimes.size(); i++) {
220              if (riderFinishTimeInStage.isAfter(allRiderTimes.
    get(i)) || riderFinishTimeInStage.equals(allRiderTimes.get(i
    ))) {
221                  allRiderTimesV2.add(allRiderTimes.get(i));
222              }
223          }
224          allRiderTimes = allRiderTimesV2;
225
226          bubbleSort(allRiderTimes);
227
228
229          for (int i = allRiderTimes.size() - 2; i >= 0; i--) {
230              riderFinishTimeInStage = riderFinishTimeInStage.
    minusSeconds(1);
231              if (!riderFinishTimeInStage.equals(allRiderTimes.
    get(i))) {
232                  riderFinishTimeInStage =
    riderFinishTimeInStage.plusSeconds(1);
233                  break;
234                  }
235              }
```

```
236         return new RiderResult(riderId ,
    riderFinishTimeInStage);
237     }
238
239
240 }
241
242
243
244
245
```

```java
package cycling;

/**
 * This enum is used to represent the stage types  on road races.
 *
 * @author Diogo Pacheco
 * @version 1.0
 *
 */
public enum StageType {
    /**
     * An enum type class to contain all the types a stage can be
     *
     */

    /**
     * Used for mostly flat stages.
     */
    FLAT,

    /**
     * Used for hilly finish or stages with moderate amounts of mountains.
     */
    MEDIUM_MOUNTAIN,

    /**
     * Used for high mountain finish or stages with multiple categorised climbs.
     */
    HIGH_MOUNTAIN,

    /**
     * Used for time trials.
     */
    TT;

}

```

```java
package cycling;


import java.io.Serializable;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;

public class Team implements Serializable {
    /**
     * A class for the team in the cycling portal
     *Containing the riders in the team, and the times they achieved in a rac.
     */
    // A hashmap of all the riders in a team, storing riders objects
    private HashMap<Integer, Rider> teamRiders = new HashMap<>();
    // The teams name
    private String teamName;
    // The team ID
    private int teamId;
    // The team description
    private String description;

    public Team( String teamName, int teamId, String description) {
        /**
         * A constructor to make a new team instance
         * @param teamName The team name
         * @param teamID The teams unique ID
         * @param description The team description
         */
        this.teamName = teamName;
        this.teamId = teamId;
        this.description = description;
    }

    public String getTeamName() {
        /**
         * A function to return the name of the team
         * @return teamName
         */
        return teamName;
    }

    public void setTeamName(String teamName) {
        /**
```

```
45          * A method to set the name of the team
46          * @param teamName The teams name
47          */
48         this.teamName = teamName;
49     }
50
51     public int getTeamId() {
52         /**
53          * A function to get the ID of the team
54          * @return teamId
55          */
56         return teamId;
57     }
58
59     public void setTeamId(int teamId) {
60         /**
61          * A method to set the team ID
62          * @param teamId The team ID
63          */
64         this.teamId = teamId;
65     }
66
67     public String getDescription() {
68         /**
69          * A function to get the description of the team
70          * @return description
71          */
72         return description;
73     }
74
75     public void setDescription(String description) {
76         /**
77          * A method to set the description of the team
78          * @return description
79          */
80         this.description = description;
81     }
82
83     public HashMap<Integer, Rider> getTeamRiders() {
84         /**
85          * A method to get the riders in the team
86          * @return teamRiders
87          */
88         return teamRiders;
89     }
90     public void addRiderToTeam(Integer riderId, Rider rider){
91         /**
92          * A method to add a rider object to the team
```

```
93           * @param riderId The rider ID
94           * @param rider The rider object
95           */
96          this.teamRiders.put(riderId, rider);
97      }
98
99      public void removeRiderFromTeam(Integer riderID){
100         /**
101          * A method to remove a rider from the team
102          * @param riderID the rider being removed
103          */
104         teamRiders.remove(riderID);
105     }
106 }
107
```