```java
package cycling;

import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.ArrayList;
import java.time.temporal.ChronoUnit;
import java.io.*;

/**
 * CyclingPortal --- A class implementing MiniCyclingPortalInterface.
 * Contains the attribute:
 * Session - A Session object containing all the created objects and other trackers.
 *
 * @author Matt Trenchard
 * @version 1.0
 */

public class CyclingPortal implements MiniCyclingPortalInterface{
    private Session session;

    /**
     * Used to find a team from the list of created teams.
     * @param id   The id of the team you wish to find.
     * @return If the team is found a team object with matching id is returned.
     *         Otherwise a null value is returned
     */
    public Team findTeam(int id){
        ArrayList<Team> teams=session.getAllTeams();//List of all created teams
        for(int i=0; i<teams.size();i++){
            if (teams.get(i).getId()==id){//Comparing search id and id of team being iterated
                Team team=teams.get(i);
                assert(team.getId()==id);
                return team;
            }
        }
        return null;
    }

    /**
     * Used to see if a rider exists in the system.
     *
     * @param riderId   The id the rider you want to search for
     * @return A boolean value. True if the rider exists. False if not.
     */
    public boolean doesRiderExist(int riderId){
        boolean foundRider=false;
        for (int i=0;i<session.getAllTeams().size();i++){//Iterates through all the teams in system
            if (foundRider){
                break;
            }
            Team teamToSearch=session.getAllTeams().get(i);
            for (int j=0;j<teamToSearch.getRiders().size();j++){//Iterates through the riders riding for the team being searched
                if (teamToSearch.getRiders().get(j).getId()==riderId){
                    foundRider=true;
                    break;
                }
```

```
57              }
58          }
59          return foundRider;
60      }
61
62      /**
63       * Used to find a race in the system from a raceId
64       * @param id   The ID of the race you want to find
65       * @return If the race exists, the race object with the correspnding ID.
66       *         Otherwise a null value is returned.
67       */
68      public Race findRace(int id){
69          ArrayList<Race> races=session.getAllRaces();//List of all created races
70          for(int i=0; i<races.size();i++){
71              if (races.get(i).getId()==id){//Comparing search id and id of race being
    iterated
72                  Race race=races.get(i);
73                  assert(race.getId()==id);
74                  return race;
75              }
76          }
77          return null;
78      }
79
80      /**
81       * Used to find a stage in the system from a stage ID
82       * @param stageId   The ID of the stage you want to find.
83       * @return If the stage exists, the stage object corresponding the stageId is
    returned
84       *         If not, a null value is returned
85       */
86      public Stage findStageInRace(int stageId){
87          ArrayList<Race> races=session.getAllRaces();
88          for (int i=0;i<races.size();i++){//Iterates through every created race
89              if (races.get(i).findStage(stageId)!=null){//Utilises race's findStage
    method to see if the stage is in the race being searched
90                  return races.get(i).findStage(stageId);
91              }
92          }
93          return null;
94      }
95
96      /**
97       * Used to find a segment in the system with a segment ID
98       * @param segmentId   The ID of the segment you want to find
99       * @return If the segment exists the segment object is returned.
100      *         If not, a null value is returned
101      */
102     public Segment findSegmentInStage(int segmentId){
103         ArrayList<Race> races=session.getAllRaces();
104         for (int i=0;i<races.size();i++){//Iterates through all the races in the
    system
105             ArrayList<Stage> stages=races.get(i).getAllStages();
106             for (int j=0;j<stages.size();j++){//Iterates through all the stages
    within race i
107                 Segment segment=stages.get(j).findSegment(segmentId);//Utilises
    stage's method findSegment to see if the segment is in stage j
108                 if(segment!=null){
109                     return segment;
110                 }
```

```
111             }
112         }
113         return null;
114     }
115 
116     /**
117      * Creates a team with the name and description specified
118      *
119      * @param name   The name of the team
120      * @param description   The description of the team
121      * @return The ID of the created team
122      * @throws IllegalNameException If the name already exists in the platform.
123    * @throws InvalidNameException If the new name is null, empty, has more than
124    *                              30 characters.
125      */
126     public int createTeam(String name, String description) throws
IllegalNameException,InvalidNameException{
127         for(int i=0;i<session.getAllTeams().size();i++){
128             if(session.getAllTeams().get(i).getName().equals(name)){//Searching all
created teams to check for name conflicts
129                 throw new IllegalNameException("Name already used");
130             }
131         }
132         if(name==null || name.equals("") || name.length()>30 || name.contains(" ")){
133             throw new InvalidNameException("Invalid name");
134         }
135         int nextId=session.getNextTeamId();//Fetches the next unused team ID
136         session.incrementTeamId();//Increments ID counter to ensure uniqueness
137         session.appendTeam(new Team(name,description,nextId));//Adds new team to
list of created teams
138         return nextId;
139         }
140 
141     /**
142      * Removes a team from the list of created teams.
143      * @param teamID   The ID of the team to be removed
144    * @throws IDNotRecognisedException If the ID does not match to any team in the
145    *                                  system.
146      */
147     public void removeTeam(int teamId) throws IDNotRecognisedException{
148         Team team=findTeam(teamId);
149         if (team==null){
150             throw new IDNotRecognisedException("ID not recognised");
151         }
152         session.deleteTeam(team);
153         assert(findTeam(teamId)==null);
154     }
155 
156     /**
157      * Get the list of teams created.
158    * @return The list of IDs of the created teams.
159      */
160     public int[] getTeams(){
161         ArrayList<Team> teams=session.getAllTeams();
162         int[] teamIds = new int[teams.size()];
163         for (int i=0;i<teams.size();i++){//Fetches ID of every created team and puts
into an array
164             teamIds[i]=teams.get(i).getId();
165         }
166         return teamIds;
```

```
167     }
168   /**
169    * Creates a rider with the specified parameters
170    *
171    * @param teamID   The ID rider's team.
172    * @param name     The name of the rider.
173    * @param yearOfBirth   The year of birth of the rider.
174    * @return The ID of the rider in the system.
175    * @throws IDNotRecognisedException If the ID does not match to any team in the
176    *                                  system.
177    * @throws IllegalArgumentException If the name of the rider is null or the year
178    *                                  of birth is less than 1900.
179    */
180     public int createRider(int teamId, String name, int yearOfBirth) throws
IDNotRecognisedException,IllegalArgumentException{
181         Team team=findTeam(teamId);
182         if (team==null){
183             throw new IDNotRecognisedException("ID not recognised");
184         }
185         else if (name==null || yearOfBirth<1900){
186             throw new IllegalArgumentException("Invalid attributes");
187         }
188         int nextId=session.getNextRiderId();
189         session.incrementRiderId();
190         team.appendRider(new Rider(name, yearOfBirth, nextId));//Adds new rider to
system
191         return nextId;
192     }
193
194     /**
195    * Removes a rider.
196    * @param riderId   The ID of the rider to be removed.
197    * @throws IDNotRecognisedException If the ID does not match to any rider in the
198    *                                  system.
199    */
200     public void removeRider(int riderId) throws IDNotRecognisedException{
201         ArrayList<Team> teams=session.getAllTeams();//Every created team
202         boolean found=false;
203         for (int i=0;i<teams.size();i++){
204             if(found){
205                 break;
206             }
207             ArrayList<Rider> riders=teams.get(i).getRiders();//Every rider in team i
208             for (int j=0;j<riders.size();j++){
209                 if (riders.get(j).getId()==riderId){//Comparing search id against
iterated id
210                     for(int k=0;k<session.getAllRaces().size();k++){//Will loop
through every race
211                         for(int
l=0;l<session.getAllRaces().get(k).getAllStages().size();l++){//Loops through every
stage in a race to delete all the rider's results
212
 deleteRiderResultsInStage(session.getAllRaces().get(k).getAllStages().get(l).getId(
), riderId);
213                         }
214                     }
215                     teams.get(i).deleteRider(riders.get(j));//Deleted if there is a
match
216                     found=true;
217                     break;
```

```
218                 }
219             }
220         }
221         if(found==false){
222             throw new IDNotRecognisedException("ID not recognised");
223         }
224         assert(doesRiderExist(riderId)==false);
225     }
226 
227   /**
228    * Gets every rider registered to a team.
229    * @param teamId   The ID of the team being queried.
230    * @return A list with riders' ID.
231    * @throws IDNotRecognisedException If the ID does not match to any team in the
232    *                                  system.
233    */
234     public int[] getTeamRiders(int teamId) throws IDNotRecognisedException{
235         if(findTeam(teamId)==null){
236             throw new IDNotRecognisedException("ID not recognised");
237         }
238         ArrayList<Rider> riders=findTeam(teamId).getRiders();
239         int[] riderIds = new int[riders.size()];
240         for (int i=0;i<riders.size();i++){//Goes through selected team and fetches
    every riders ID
241             riderIds[i]=riders.get(i).getId();
242         }
243         return riderIds;
244     }
245 
246     /**
247      * Get the races currently created in the platform.
248    *
249    * @return An array of race IDs in the system or an empty array if none exists.
250      */
251     public int[] getRaceIds(){
252         ArrayList<Race> races=session.getAllRaces();
253         int[] raceIds = new int[races.size()];//Creates array to hold all race IDs
254         for (int i=0;i<races.size();i++){
255             raceIds[i]=races.get(i).getId();
256         }
257         return raceIds;
258     }
259 
260   /**
261    * The method creates a staged race in the platform with the given name and
262    * description.
263    *
264    * @param name        Race's name.
265    * @param description Race's description (can be null).
266    * @throws IllegalNameException If the name already exists in the platform.
267    * @throws InvalidNameException If the name is null, empty, has more than 30
268    *                              characters, or has white spaces.
269    * @return the unique ID of the created race.
270    *
271    */
272     public int createRace(String name, String description) throws
    IllegalNameException, InvalidNameException{
273         for(int i=0;i<session.getAllRaces().size();i++){//Searching current races
    for any name conflict
274             if(session.getAllRaces().get(i).getName().equals(name)){
```

```
275                 throw new IllegalNameException("Name already used");
276             }
277         }
278         if(name==null || name.equals("") || name.length()>30 || name.contains(" ")){
279             throw new InvalidNameException("Invalid name");
280         }
281 
282         int nextId=session.getNextRaceId();//Gets next unique race ID
283         session.incrementRaceId();
284         session.appendRace(new Race(nextId,name,description));
285         return nextId;
286     }
287 
288     /**
289    * Get the details from a race.
290    *
291    * @param raceId The ID of the race being queried.
292    * @return A string of format - id, name, description, number of stages, total
   length.
293    * @throws IDNotRecognisedException If the ID does not match to any race in the
294    *                                  system.
295    */
296     public String viewRaceDetails(int raceId) throws IDNotRecognisedException{
297         Race race=findRace(raceId);
298         if(race==null){
299             throw new IDNotRecognisedException("ID not recognised");
300         }
301         ArrayList<Stage>stages=race.getAllStages();
302         double length=0;
303         for(int i=0;i<stages.size();i++){//Loop sums the length of all stages
304             length+=stages.get(i).getLength();
305         }
306         String details = String.format("ID: %d\nName: %s\nDescription: %s\nNumber of
   Stages: %d\nLength:
   %.2f",race.getId(),race.getName(),race.getDesc(),stages.size(),length);
307         return details;
308     }
309 
310     /**
311    * The method removes the race and all its related information, i.e., stages,
312    * segments, and results.
313    *
314    * @param raceId The ID of the race to be removed.
315    * @throws IDNotRecognisedException If the ID does not match to any race in the
316    *                                  system.
317    */
318     public void removeRaceById(int raceId) throws IDNotRecognisedException{
319         Race race=findRace(raceId);
320         if(race==null){
321             throw new IDNotRecognisedException("ID not recognised");
322         }
323         session.removeRace(race);
324         assert(findRace(raceId)==null);
325     }
326 
327     /**
328    * The method queries the number of stages created for a race.
329    *
330    * @param raceId The ID of the race being queried.
331    * @return The number of stages created for the race.
```

```
332     * @throws IDNotRecognisedException If the ID does not match to any race in the
333     *                                  system.
334     */
335      public int getNumberOfStages(int raceId) throws IDNotRecognisedException{
336          Race race=findRace(raceId);
337          if(race==null){
338              throw new IDNotRecognisedException("ID not recognised");
339          }
340          return race.getAllStages().size();
341      }
342
343      /**
344     * Creates a new stage and adds it to the race.
345     *
346     * @param raceId      The race which the stage will be added to.
347     * @param stageName   An identifier name for the stage.
348     * @param description A descriptive text for the stage.
349     * @param length      Stage length in kilometres.
350     * @param startTime   The date and time in which the stage will be raced. It
351     *                    cannot be null.
352     * @param type        The type of the stage. This is used to determine the
353     *                    amount of points given to the winner.
354     * @return the unique ID of the stage.
355     * @throws IDNotRecognisedException If the ID does not match to any race in the
356     *                                  system.
357     * @throws IllegalNameException     If the name already exists in the platform.
358     * @throws InvalidNameException     If the new name is null, empty, has more
359     *                                  than 30.
360     * @throws InvalidLengthException   If the length is less than 5km.
361     */
362      public int addStageToRace(int raceId, String stageName, String description,
    double length, LocalDateTime startTime, StageType type)throws
    IDNotRecognisedException, IllegalNameException, InvalidNameException,
    InvalidLengthException{
363          Race race=findRace(raceId);
364          if (race==null){
365              throw new IDNotRecognisedException("ID not recognised");
366          }
367          for (int i=0;i<session.getAllRaces().size();i++){
368              Race raceToSearch=session.getAllRaces().get(i);
369              for (int j=0;j<raceToSearch.getAllStages().size();j++){
370                  if (raceToSearch.getAllStages().get(j).getName().equals(stageName)){
371                      throw new IllegalNameException("Name already used");
372                  }
373              }
374          }
375          if(stageName==null || stageName.equals("") || stageName.length()>30 ||
    stageName.contains(" ")){
376              throw new InvalidNameException("Invalid name");
377          }
378          else if(length<5){
379              throw new InvalidLengthException("Length must be more than 5km");
380          }
381          int nextId=session.getNextStageId();//Gets the next unique stage ID
382          session.incrementStageId();
383          race.insertStage(new
    Stage(nextId,stageName,description,length,startTime,type));
384          return nextId;
385      }
386
```

```
387     /**
388    * Retrieves the list of stage IDs of a race.
389    *
390    * @param raceId The ID of the race being queried.
391    * @return The list of stage IDs ordered (from first to last) by their sequence in
       the
392    *         race.
393    * @throws IDNotRecognisedException If the ID does not match to any race in the
394    *                                  system.
395    */
396     public  int[] getRaceStages(int raceId) throws IDNotRecognisedException{
397         Race race=findRace(raceId);
398         if(race==null){
399             throw new IDNotRecognisedException("ID not recognised");
400         }
401         ArrayList<Stage> stages=race.getAllStages();
402         int[] stageIds= new int[stages.size()];
403         for (int i=0;i<stages.size();i++){//Loops through every stage in selected
       race and fetches the stage ID
404             stageIds[i]=stages.get(i).getId();
405         }
406         return stageIds;
407     }
408
409     /**
410    * Gets the length of a stage in a race, in kilometres.
411    *
412    * @param stageId The ID of the stage being queried.
413    * @return The stage's length.
414    * @throws IDNotRecognisedException If the ID does not match to any stage in the
415    *                                  system.
416    */
417     public double getStageLength(int stageId) throws IDNotRecognisedException{
418         Stage stage=findStageInRace(stageId);
419         if(stage==null){
420             throw new IDNotRecognisedException("ID not recognised");
421         }
422         return stage.getLength();
423     }
424
425     /**
426    * Removes a stage and all its related data, i.e., segments and results.
427    *
428    * @param stageId The ID of the stage being removed.
429    * @throws IDNotRecognisedException If the ID does not match to any stage in the
430    *                                  system.
431    */
432     public void removeStageById(int stageId) throws IDNotRecognisedException{
433         ArrayList<Race> races=session.getAllRaces();
434         boolean removed=false;
435         for (int i=0;i<races.size();i++){
436             if (races.get(i).findStage(stageId)!=null){
437                 races.get(i).removeStage(races.get(i).findStage(stageId));
438                 removed=true;
439                 break;
440             }
441         }
442         if (removed==false){//If nothing has been removed it means the ID has not
       been found
443             throw new IDNotRecognisedException("ID not recognised");
```

```
444         }
445         assert(findStageInRace(stageId)==null);
446     }
447 
448     /**
449    * Adds a climb segment to a stage.
450      *
451    * @param stageId         The ID of the stage to which the climb segment is
452    *                        being added.
453    * @param location        The kilometre location where the climb finishes within
454    *                        the stage.
455    * @param type            The category of the climb - {@link SegmentType#C4},
456    *                        {@link SegmentType#C3}, {@link SegmentType#C2},
457    *                        {@link SegmentType#C1}, or {@link SegmentType#HC}.
458    * @param averageGradient The average gradient for the climb.
459    * @param length          The length of the climb in kilometre.
460    * @return The ID of the segment created.
461    * @throws IDNotRecognisedException   If the ID does not match to any stage in
462    *                                    the system.
463    * @throws InvalidLocationException   If the location is out of bounds of the
464    *                                    stage length.
465    * @throws InvalidStageStateException If the stage is "waiting for results".
466    * @throws InvalidStageTypeException  Time-trial stages cannot contain any
467    *                                    segment.
468    */
469     public int addCategorizedClimbToStage(int stageId, Double location, SegmentType
    type, Double averageGradient, Double length)throws IDNotRecognisedException,
    InvalidLocationException, InvalidStageStateException,
470     InvalidStageTypeException{
471         Stage stage=findStageInRace(stageId);
472         if(stage==null){
473             throw new IDNotRecognisedException("ID not recognised");
474         }
475         else if (location>stage.getLength() || location<=0){
476             throw new InvalidLocationException("Invalid peak location");
477         }
478         else if (stage.getState().equals("wait")){
479             throw new InvalidStageStateException("Stage is out of prep phase");
480         }
481         else if (stage.getType()==StageType.TT){
482             throw new InvalidStageTypeException("Time trials cannot contain
    climbs");
483         }
484         int id = session.getNextSegmentId();
485         session.incrementSegmentId();
486         stage.insertSegment(new Segment(id, location, averageGradient, type));
487 
488         return id;
489     }
490 
491     /**
492    * Adds an intermediate sprint to a stage.
493    *
494    * @param stageId  The ID of the stage to which the intermediate sprint segment
495    *                 is being added.
496    * @param location The kilometre location where the intermediate sprint finishes
497    *                 within the stage.
498    * @return The ID of the segment created.
499    * @throws IDNotRecognisedException   If the ID does not match to any stage in
500    *                                    the system.
```

```
501     * @throws InvalidLocationException   If the location is out of bounds of the
502     *                                    stage length.
503     * @throws InvalidStageStateException If the stage is "waiting for results".
504     * @throws InvalidStageTypeException  Time-trial stages cannot contain any
505     *                                    segment.
506     */
507     public int addIntermediateSprintToStage(int stageId, double location)throws
    IDNotRecognisedException,
508     InvalidLocationException, InvalidStageStateException, InvalidStageTypeException{
509         Stage stage=findStageInRace(stageId);
510         if (stage==null){
511             throw new IDNotRecognisedException("ID not recognised");
512         }
513         else if (location>stage.getLength() || location<=0){
514             throw new InvalidLocationException("Invalid sprint location");
515         }
516         else if (stage.getState().equals("wait")){
517             throw new InvalidStageStateException("Stage is out of prep phase");
518         }
519         else if (stage.getType()==StageType.TT){
520             throw new InvalidStageTypeException("Time trials cannot contain
    sprints");
521         }
522 
523         int id = session.getNextSegmentId();
524         session.incrementSegmentId();
525         stage.insertSegment(new Segment(id, location, SegmentType.SPRINT));
526 
527         return id;
528     }
529 
530     /**
531     * Retrieves the list of segment (mountains and sprints) IDs of a stage.
532     *
533     * @param stageId The ID of the stage being queried.
534     * @return The list of segment IDs ordered (from first to last) by their location
    in the
535     *         stage.
536     * @throws IDNotRecognisedException If the ID does not match to any stage in the
537     *                                  system.
538     */
539     public int[] getStageSegments(int stageId) throws IDNotRecognisedException{
540         Stage stage=findStageInRace(stageId);
541         if (stage==null){
542             throw new IDNotRecognisedException("ID not recognised");
543         }
544         ArrayList<Segment> segments=stage.getSegments();
545         int[] segmentIds= new int[segments.size()];
546         for(int i=0;i<segmentIds.length;i++){//Goes through each segment is selected
    stage and gets the segment ID
547             segmentIds[i]=segments.get(i).getId();
548         }
549 
550         return segmentIds;
551     }
552 
553     /**
554     * Removes a segment from a stage.
555     *
556     * @param segmentId The ID of the segment to be removed.
```

```
557     * @throws IDNotRecognisedException   If the ID does not match to any segment in
558     *                                     the system.
559     * @throws InvalidStageStateException If the stage is "waiting for results".
560     */
561     public void removeSegment(int segmentId)throws IDNotRecognisedException,
    InvalidStageStateException{
562         boolean found=false;
563         ArrayList<Race> races=session.getAllRaces();
564         for (int i=0;i<races.size();i++){
565             if (found==true){//If segment has been found in the previously searched
    race the search is ended
566                 break;
567             }
568             ArrayList<Stage> stages=races.get(i).getAllStages();
569             for (int j=0;j<stages.size();j++){
570                 Segment segment=stages.get(j).findSegment(segmentId);
571                 if (segment!=null){
572                     if (stages.get(j).getState()=="wait"){
573                         throw new InvalidStageStateException("Stage is waiting for
    results");
574                     }
575                     stages.get(j).removeSegment(segment);
576                     found=true;
577                     break;
578                 }
579             }
580         }
581         if (found==false){
582             throw new IDNotRecognisedException("ID not recognised");
583         }
584         assert(findSegmentInStage(segmentId)==null);
585     }
586 
587     /**
588     * Record the times of a rider in a stage.
589     *
590     * @param stageId     The ID of the stage the result refers to.
591     * @param riderId     The ID of the rider.
592     * @param checkpoints An array of times at which the rider reached each of the
593     *                    segments of the stage, including the start time and the
594     *                    finish line.
595     * @throws IDNotRecognisedException    If the ID does not match to any rider or
596     *                                     stage in the system.
597     * @throws DuplicatedResultException   Thrown if the rider has already a result
598     *                                     for the stage. Each rider can have only
599     *                                     one result per stage.
600     * @throws InvalidCheckpointsException Thrown if the length of checkpoints is
601     *                                     not equal to n+2, where n is the number
602     *                                     of segments in the stage; +2 represents
603     *                                     the start time and the finish time of the
604     *                                     stage.
605     * @throws InvalidStageStateException  Thrown if the stage is not "waiting for
606     *                                     results". Results can only be added to a
607     *                                     stage while it is "waiting for results".
608     */
609     public void registerRiderResultsInStage(int stageId, int riderId, LocalTime...
    checkpoints)throws IDNotRecognisedException,
610     DuplicatedResultException, InvalidCheckpointsException,
    InvalidStageStateException{
611         Stage stage=findStageInRace(stageId);
```

```
612         if (stage==null){
613             throw new IDNotRecognisedException("Stage ID not recognised");
614         }
615         else if (stage.getState().equals("prep")){
616             throw new InvalidStageStateException("Stage in preperation phase");
617         }
618         else if (checkpoints.length != stage.getSegments().size()+2){
619             throw new InvalidCheckpointsException("Incorrect number of checkpoints
    submitted");
620         }
621 
622         if (doesRiderExist(riderId)==false){
623             throw new IDNotRecognisedException("Rider ID not recognised");
624         }
625 
626         boolean foundResult=false;
627         for (int i=0;i<stage.getFinishResults().size();i++){//Search for any
    existing result for rider in stage
628             if (stage.getFinishResults().get(i).getRiderId()==riderId){
629                 foundResult=true;
630             }
631         }
632         if (foundResult){
633             throw new DuplicatedResultException("Rider already has result
    registered");
634         }
635 
636         ArrayList<Segment> segments=stage.getSegments();
637         stage.insertStartTime(new RiderResult(riderId, checkpoints[0]));//Adds
    riders start time to stage
638         for (int i=0;i<segments.size();i++){//Goes through each segment in stage and
    adds the time at which the rider finish each segment
639             segments.get(i).insertCheckpoint(new RiderResult(riderId,
    checkpoints[i+1]));
640         }
641         stage.insertFinish(new RiderResult(riderId, checkpoints[checkpoints.length-
    1]));//Adds riders finish time to stage
642     }
643 
644 
645     /**
646    * Get the riders finished position in a a stage.
647    *
648    * @param stageId The ID of the stage being queried.
649    * @return A list of riders ID sorted by their elapsed time. An empty list if
650    *         there is no result for the stage.
651    * @throws IDNotRecognisedException If the ID does not match any stage in the
652    *                                  system.
653    */
654     public int[] getRidersRankInStage(int stageId) throws IDNotRecognisedException{
655         Stage stage=findStageInRace(stageId);
656         if (stage==null){
657             throw new IDNotRecognisedException("ID not recognised");
658         }
659         if (stage.getType()!=StageType.TT){//If the stage is not a time trial then
    the finish reuslts of the stage are fetched which are already sorted
660             int[] rank= new int[stage.getFinishResults().size()];
661             for (int i=0;i<stage.getFinishResults().size();i++){
662                 rank[i]=stage.getFinishResults().get(i).getRiderId();
663             }
```

```
664             return rank;
665         }
666         else{//If the stage is a time trial the time of completion must be calculated and sorted for each rider
667             ArrayList<Long> times= new ArrayList<Long>();
668             ArrayList<Integer> riderIds= new ArrayList<Integer>();
669             //Two lists are created which contain times and riders ids. The rider id at index n will always be the rider who has the time at index n in list times
670             for (int i=0;i<stage.getFinishResults().size();i++){//Iterates through the stages finish results
671                 LocalTime finishTime = stage.getFinishResults().get(i).getTime();
672                 int riderId=stage.getFinishResults().get(i).getRiderId();
673                 LocalTime startTime = stage.findRiderStart(riderId).getTime();
674                 long seconds = ChronoUnit.SECONDS.between(startTime, finishTime);//Calculates the time the rider completed TT in
675                 if(times.size()==0){//If the lists are empty just add time and rider
676                     times.add(seconds);
677                     riderIds.add(riderId);
678                 }
679                 else if(seconds>=times.get(times.size()-1)){//If the time is longer than the final element it is added onto the end of the list
680                     times.add(seconds);
681                     riderIds.add(riderId);
682                 }
683                 else{
684                     for (int j=0;j<times.size();j++){//Inserts with insertion sort logic
685                         if (seconds<=times.get(j)){
686                             times.add(j,seconds);
687                             riderIds.add(j,riderId);
688                             break;
689                         }
690                     }
691                 }
692             }
693             int[] rank= new int[riderIds.size()];//Converts riderId list into array to be returned
694             for(int i=0;i<rank.length;i++){
695                 rank[i]=riderIds.get(i);
696             }
697             return rank;
698         }
699     }
700 
701 
702     /**
703    * Concludes the preparation of a stage. After conclusion the stages state will be "wait"
704      * which stands for 'waiting for results'
705    *
706    * @param stageId The ID of the stage to be concluded.
707    * @throws IDNotRecognisedException   If the ID does not match to any stage in
708    *                                    the system.
709    * @throws InvalidStageStateException If the stage is waiting for results.
710    */
711     public void concludeStagePreparation(int stageId) throws IDNotRecognisedException,InvalidStageStateException{
712         Stage stage=findStageInRace(stageId);
713         if (stage==null){
714             throw new IDNotRecognisedException("ID not recognised");
```

```
715         }
716         else if (stage.getState().equals("wait")){
717             throw new InvalidStageStateException("Stage is already waiting for
    results");
718         }
719         stage.concludePrep();
720     }
721 
722     /**
723    * Get the times of a rider in a stage.
724    *
725    * @param stageId The ID of the stage the result refers to.
726    * @param riderId The ID of the rider.
727    * @return The array of times at which the rider reached each of the segments of
728    *         the stage and the total elapsed time. The elapsed time is the
729    *         difference between the finish time and the start time. Return an
730    *         empty array if there is no result registered for the rider in the
731    *         stage.
732    * @throws IDNotRecognisedException If the ID does not match to any rider or
733    *                                  stage in the system.
734    */
735     public LocalTime[] getRiderResultsInStage(int stageId, int riderId) throws
    IDNotRecognisedException{
736         Stage stage=findStageInRace(stageId);
737         if (stage==null){
738             throw new IDNotRecognisedException("Stage ID not recognised");
739         }
740 
741         if (doesRiderExist(riderId)==false){
742             throw new IDNotRecognisedException("Rider ID not recognised");
743         }
744 
745         boolean isResult= false;
746         if(stage.findRiderResult(riderId)!=null){//Finding if there is a result for
    the rider
747                 isResult=true;
748         }
749 
750         if (isResult){
751             LocalTime finishTime=stage.findRiderResult(riderId).getTime();
752             LocalTime[] times = new LocalTime[stage.getSegments().size()+1];
753             for (int i=0;i<stage.getSegments().size();i++){
754                 Segment segment=stage.getSegments().get(i);
755                 times[i]=segment.findRiderResult(riderId).getTime();
756             }
757             LocalTime riderStart = stage.findRiderStart(riderId).getTime();
758             long seconds=ChronoUnit.SECONDS.between(riderStart,finishTime);
759             times[times.length-1]=LocalTime.ofSecondOfDay(seconds);
760             return times;
761             //returns in format [checkpoints,elapsed time]
762         }
763         else{
764             return new LocalTime[0];
765         }
766     }
767 
768     /**
769      * If a rider finishes within a second of the rider ahead of them they geet the
    time of the rider ahead.
```

```
770      * So if 100 riders finish together with no gap bigger than 1 second between any
two riders all the riders
771      * get the same time.
772     *
773     * @param stageId The ID of the stage the result refers to.
774     * @param riderId The ID of the rider.
775      * @return The adjusted elapsed time for the rider in the stage. Return an empty
776     *         array if there is no result registered for the rider in the stage.
777     * @throws IDNotRecognisedException   If the ID does not match to any rider or
778     *                                    stage in the system.
779    */
780     public LocalTime getRiderAdjustedElapsedTimeInStage(int stageId,int
riderId)throws IDNotRecognisedException{
781         Stage stage=findStageInRace(stageId);
782         if(stage==null){
783             throw new IDNotRecognisedException("Stage ID not recognised");
784         }
785         if (doesRiderExist(riderId)==false){
786             throw new IDNotRecognisedException("Rider ID not recognised");
787         }
788         RiderResult startResult = stage.findRiderStart(riderId);
789         RiderResult finishResult = stage.findRiderResult(riderId);
790 
791         if(startResult==null){//If no result exists for the rider a null value is
returned
792             return null;
793         }
794 
795         LocalTime riderStart = startResult.getTime();
796         LocalTime finishTime = finishResult.getTime();
797 
798         if (stage.getType()!=StageType.TT){//If the stage is a time trial no
adjustments to finishing time are made
799             boolean loop=true;
800             int
count=stage.getFinishResults().indexOf(stage.findRiderResult(riderId));//Gets the
index of the riders result
801             while(loop==true && count>0){//Will loop until the gap to the rider
ahead is more than one second or the number 1 rider has been reached
802                 if(ChronoUnit.SECONDS.between(stage.getFinishResults().get(count-
1).getTime(),stage.getFinishResults().get(count).getTime())<1){
803                 //Calculates gap between current iterated rider's time and the rider
1 position ahead.
804                 //If the gap is less than 1 second the next rider is looked at
805                     count-=1;
806                 }
807                 else{
808                     loop=false;
809                 }
810                 finishTime=stage.getFinishResults().get(count).getTime();
811                 //The new finish time of the rider is adjusted every time a rider
that finished ahead is found to be in their riding group
812             }
813         }
814 
815         long seconds=ChronoUnit.SECONDS.between(riderStart,finishTime);
816         return LocalTime.ofSecondOfDay(seconds);
817     }
818 
819     /**
```

```
820     * Removes the stage results from the rider.
821     *
822     * @param stageId The ID of the stage the result refers to.
823     * @param riderId The ID of the rider.
824     * @throws IDNotRecognisedException If the ID does not match to any rider or
825     *                                  stage in the system.
826     */
827     public void deleteRiderResultsInStage(int stageId, int riderId) throws
    IDNotRecognisedException{
828         Stage stage=findStageInRace(stageId);
829         if (stage==null){
830             throw new IDNotRecognisedException("Stage ID not recognised");
831         }
832         if (doesRiderExist(riderId)==false){
833             throw new IDNotRecognisedException("Rider ID not recognised");
834         }
835         if (stage.findRiderResult(riderId)!=null){//If a result exists for rider it
    will delete it
836             stage.removeFinishResult(stage.findRiderResult(riderId));
837             stage.removeRiderStartTime(stage.findRiderStart(riderId));
838             for (int i=0;i<stage.getSegments().size();i++){
839                 Segment segment=stage.getSegments().get(i);
840                 segment.removeCheckpointResult(segment.findRiderResult(riderId));
841             }
842         }
843     }
844
845     /**
846     * Get the adjusted elapsed times of riders in a stage.
847     *
848     * @param stageId The ID of the stage being queried.
849     * @return The ranked list of adjusted elapsed times sorted by their finish
850     *         time. An empty list if there is no result for the stage. These times
851     *         will match the riders ad order returned by
852     *         {@link #getRidersRankInStage(int)}.
853     * @throws IDNotRecognisedException If the ID does not match any stage in the
854     *                                  system.
855     */
856     public LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId) throws
    IDNotRecognisedException{
857         Stage stage=findStageInRace(stageId);
858         if (stage==null){
859             throw new IDNotRecognisedException("ID not recognised");
860         }
861         int[] ranking = getRidersRankInStage(stageId);
862         LocalTime[] adjustedElapsedTimes= new
    LocalTime[stage.getFinishResults().size()];
863         for(int i=0;i<stage.getFinishResults().size();i++){
864             adjustedElapsedTimes[i]=getRiderAdjustedElapsedTimeInStage(stageId,
    ranking[i]);
865             //Uses ranking array to know which rider Id is at each position
866             //Then calculates the corresponding adjusted time for that rider
867         }
868         return adjustedElapsedTimes;
869     }
870
871     /**
872     * Get the number of points obtained by each rider in a stage.
873      *
874     * @param stageId The ID of the stage being queried.
```

```
875     * @return The ranked list of points each riders received in the stage, sorted
876     *         by their elapsed time. An empty list if there is no result for the
877     *         stage. These points will match the riders and order returned by
878     *         {@link #getRidersRankInStage(int)}.
879     * @throws IDNotRecognisedException If the ID does not match any stage in the
880     *                                  system.
881     */
882     public int[] getRidersPointsInStage(int stageId) throws
IDNotRecognisedException{
883         Stage stage = findStageInRace(stageId);
884         if (stage==null){
885             throw new IDNotRecognisedException("ID not recognised");
886         }
887         int[] rank = getRidersRankInStage(stageId);
888         int[] ridersPoints = new int[rank.length];//An array to store riders points.
Index i in this array will contain the points of the rider at position i in the rank
array
889         int[] interSprintPoints = {20,17,15,13,11,10,9,8,7,6,5,4,3,2,1};
890         int[] stagePoints;
891         if (stage.getType()==StageType.FLAT){
892             stagePoints=new int[] {50,30,20,18,16,14,12,10,8,7,6,5,4,3,2};
893         }
894         else if(stage.getType()==StageType.MEDIUM_MOUNTAIN){
895             stagePoints=new int[] {30,25,22,19,17,15,13,11,9,7,6,5,4,3,2};
896         }
897         else{
898             stagePoints=new int[] {20,17,15,13,11,10,9,8,7,6,5,4,3,2,1};
899         }
900         if (stage.getType()==StageType.TT){//If its a TT there are no sprints so
points are simply assigned by finishing position
901             for (int i=0;i<ridersPoints.length;i++){
902                 ridersPoints[i]=stagePoints[i];
903             }
904         }
905         else{
906             for(int i=0;i<stage.getFinishResults().size();i++){//Iterates through
every finish result registered
907                 if (i==15){//Points are only assigned for first 15 finishers
908                     break;
909                 }
910                 int scoringId=stage.getFinishResults().get(i).getRiderId();//Gets
the ith finishing riders id
911                 for(int j=0;j<rank.length;j++){//Goes through the rank array to find
the scoring rider and awards them their points
912                     if(rank[j]==scoringId){
913                         ridersPoints[j]+=stagePoints[i];
914                     }
915                 }
916             }
917             for(int i=0;i<stage.getSegments().size();i++){//Iterates through every
segment for a stage
918                 Segment segment=stage.getSegments().get(i);
919                 if (segment.getType()==SegmentType.SPRINT){//If the segment is a
sprint then sprint points need to be awarded
920                     for (int j=0; j<segment.getCheckpointResults().size();j++)
{//Iterates through all registered results for a segment
921                         if(j==15){//Points only awarded for first 15 finishers
922                             break;
923                         }
```

```
924                         int scoringId=segment.getCheckpointResults().get(j).getRiderId();//The id of the rider due points
925                         for(int k=0;k<rank.length;k++){//Iterates to find the correct rider and then assigns points
926                             if(rank[k]==scoringId){
927                                 ridersPoints[k]+=interSprintPoints[j];
928                             }
929                         }
930                     }
931                 }
932             }
933         }
934         return ridersPoints;
935     }
936 
937     /**
938    * Get the number of mountain points obtained by each rider in a stage.
939    *
940    * @param stageId The ID of the stage being queried.
941    * @return The ranked list of mountain points each riders received in the stage,
942    *         sorted by their finish time. An empty list if there is no result for
943    *         the stage. These points will match the riders and order returned by
944    *         {@link #getRidersRankInStage(int)}.
945    * @throws IDNotRecognisedException If the ID does not match any stage in the
946    *                                  system.
947    */
948     public int[] getRidersMountainPointsInStage(int stageId) throws IDNotRecognisedException{
949         Stage stage = findStageInRace(stageId);
950         if (stage==null){
951             throw new IDNotRecognisedException("ID not recognised");
952         }
953         int[] rank = getRidersRankInStage(stageId);
954         int[] ridersPoints = new int[rank.length];//New array to store points of riders. Index n will store the points of the rider at index n in rank array
955         int[] climbPoints;//Array which will store the points availible for each climb
956         for(int i=0;i<stage.getSegments().size();i++){
957             Segment segment=stage.getSegments().get(i);
958             boolean mountain =false;
959             //If the segemnt is a climb then the points availible are assigned and the variable mountain is set to true
960             if (segment.getType()==SegmentType.C1){
961                 climbPoints= new int[] {10,8,6,4,2,1};
962                 mountain=true;
963             }
964             else if (segment.getType()==SegmentType.C2){
965                 climbPoints= new int[] {5,3,2,1};
966                 mountain=true;
967             }
968             else if (segment.getType()==SegmentType.C3){
969                 climbPoints= new int[] {2,1};
970                 mountain=true;
971             }
972             else if (segment.getType()==SegmentType.C4){
973                 climbPoints= new int[] {1};
974                 mountain=true;
975             }
976             else if (segment.getType()==SegmentType.HC){
```

```
977                 climbPoints= new int[] {20,15,12,10,8,6,4,2};
978                 mountain=true;
979             }
980             else{
981                 climbPoints= new int[0];
982             }
983             if (mountain){
984                 for (int j=0; j<segment.getCheckpointResults().size();j++){//Gets
the checkpoint times for the segment
985                     if(j==climbPoints.length){//If j is at the length of the array
of availible points it means all the points have been awarded
986                         break;
987                     }
988                     int
scoringId=segment.getCheckpointResults().get(j).getRiderId();//The rider that
reached peak in position j that is due points
989                     for(int k=0;k<rank.length;k++){//Searching for the rider to
assign their points
990                         if(rank[k]==scoringId){
991                             ridersPoints[k]+=climbPoints[j];
992                         }
993                     }
994                 }
995             }
996         }
997         return ridersPoints;
998     }
999 
1000     /**
1001      * Erases all the data stored for the portal and creates a new session
1002      */
1003     public void eraseCyclingPortal(){
1004         session= new Session();
1005     }
1006 
1007     /**
1008    * Method saves this MiniCyclingPortalInterface contents into a serialised file,
1009    * with the filename given in the argument.
1010    *
1011    * @param filename Location of the file to be saved.
1012    * @throws IOException If there is a problem experienced when trying to save the
1013    *                     store contents to the file.
1014    */
1015     public void saveCyclingPortal(String filename) throws IOException{
1016         ObjectOutputStream out = new ObjectOutputStream (new
FileOutputStream(filename));
1017         out.writeObject(session);
1018         out.close();
1019     }
1020 
1021     /**
1022    * Method should load and replace this MiniCyclingPortalInterface contents with
the
1023    * serialised contents stored in the file given in the argument.
1024    *
1025    * @param filename Location of the file to be loaded.
1026    * @throws IOException           If there is a problem experienced when trying
1027    *                               to load the store contents from the file.
1028    * @throws ClassNotFoundException If required class files cannot be found when
1029    *                               loading.
```

```
1030    */
1031     public void loadCyclingPortal(String filename) throws IOException,
ClassNotFoundException{
1032         ObjectInputStream in = new ObjectInputStream(new FileInputStream(filename));
1033         Object obj = in.readObject();
1034         if (obj instanceof Session){
1035             session= (Session) obj;
1036         }
1037         in.close();
1038     }
1039 
1040     /**
1041      * Creates a new portal. Also creates a fresh session.
1042      */
1043     public CyclingPortal(){
1044         this.session = new Session();
1045     }
1046 }
```

```java
package cycling;

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

/**
 * Session --- A class to store all the created teams and races as well as ID counters
 * for every object that requires a unique ID.
 * This makes the saving and loading of the portal easier as only the session object
 * within the portal has to be loaded or saved.
 * Contains the following attributes:
 * allTeams (ArrayList<Team>) - An ArrayList of every team created
 * allRaces (ArrayList<Race>) - An ArrayList of every created race
 * nextTeamId (int) - The next ID to be assigned to a team
 * nextRiderId(int) - The next ID to be assigned to a rider
 * nextRaceId (int) - The next ID to be assigned to a race
 * nextStageId (int) - The next ID to be assigned to a stage
 * nextSegmentId (int) - The next ID to be assigned to a segment
 *
 * @author Matt Trenchard
 * @version 1.1
 */

public class Session implements Serializable{
    private ArrayList<Team> allTeams;
    /**
     * Gets an ArrayList of every created team.
     * @return ArrayList of every created team.
     */
    public ArrayList<Team> getAllTeams(){
        return allTeams;
    }
    private int nextTeamId;
    /**
     * Gets the next team ID to be used.
     * @return Next team ID to be used.
     */
    public int getNextTeamId(){
        return nextTeamId;
    }

    private int nextRiderId;
    /**
     * Gets the next rider ID to be used.
     * @return Next rider ID to be used.
     */
    public int getNextRiderId(){
        return nextRiderId;
    }

    private ArrayList<Race> allRaces;
    /**
     * Gets every race in the system
     * @return ArrayList of every race in the system
     */
    public ArrayList<Race> getAllRaces(){
        return allRaces;
    }
```

```
59      private int nextRaceId;
60      /**
61       * Gets the next unused race ID
62       * @return Next unused race ID
63       */
64      public int getNextRaceId(){
65          return nextRaceId;
66      }
67
68      private int nextStageId;
69      /**
70       * Gets the next unused stage ID
71       * @return Next unused stage ID
72       */
73      public int getNextStageId(){
74          return nextStageId;
75      }
76
77      private int nextSegmentId;
78      /**
79       * Gets the next unused segment ID
80       * @return Next unused segment ID
81       */
82      public int getNextSegmentId(){
83          return nextSegmentId;
84      }
85
86      /**
87       * Adds a team to the ArrayList of teams when one is created
88       * @param team   The team that has been created.
89       */
90      public void appendTeam(Team team){
91          allTeams.add(team);
92      }
93
94      /**
95       * Removes a team from the ArrayList of created teams.
96       * @param team   The team to be removed.
97       */
98      public void deleteTeam(Team team){
99          allTeams.remove(team);
100     }
101
102     /**
103      * Adds a race to the list of created races
104      * @param race   The race to be added
105      */
106     public void appendRace(Race race){
107         allRaces.add(race);
108     }
109
110     /**
111      * Removes a race from the list of created races
112      * @param race   The race to be removed
113      */
114     public void removeRace(Race race){
115         allRaces.remove(race);
116     }
117
118     /**
```

```
119      * Increments the team ID counter
120      */
121     public void incrementTeamId(){
122         nextTeamId++;
123     }
124 
125     /**
126      * Increments the rider ID counter
127      */
128     public void incrementRiderId(){
129         nextRiderId++;
130     }
131 
132     /**
133      * Increments the race ID counter
134      */
135     public void incrementRaceId(){
136         nextRaceId++;
137     }
138 
139     /**
140      * Increments the stage ID counter
141      */
142     public void incrementStageId(){
143         nextStageId++;
144     }
145 
146     /**
147      * Increments the segment ID counter
148      */
149     public void incrementSegmentId(){
150         nextSegmentId++;
151     }
152     /**
153      * Creates a new session with an empty list of teams and races and ID counters of
     1
154      */
155     public Session(){
156         allTeams= new ArrayList<Team>();
157         nextTeamId=1;
158         nextRiderId=1;
159         nextRaceId=1;
160         nextSegmentId=1;
161         nextStageId=1;
162         allRaces= new ArrayList<Race>();
163     }
164 }
```

```java
package cycling;

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

/**
 * Race --- A class to represent a cycling race which can contain multiple stages of different types.
 * Contains the following attributes:
 * allStages(ArrayList<Stage>) - An ArrayList containing all the of the stages that make up a race
 * id(int) - An id unique to each race
 * name(String) - The name of the race
 * desc(String) - The description of the race
 *
 * @author Matt Trenchard
 * @version 1.0
 */
public class Race implements Serializable{
    private ArrayList<Stage> allStages;
    /**
     * Gets all the stages in the race
     * @return ArrayList of stages in the race
     */
    public ArrayList<Stage> getAllStages(){
        return allStages;
    }
    private int id;
    /**
     * Gets the id of the race
     * @return The id of the race
     */
    public int getId(){
        return id;
    }
    private String name;
    /**
     * Gets the name of the race
     * @return The name of the race
     */
    public String getName(){
        return name;
    }
    private String desc;
    /**
     * Gets the description of the race
     * @return The race description
     */
    public String getDesc(){
        return desc;
    }


    /**
     * Used to add a stage to a race. The stage is inserted based on it's date.
     * @param stage   The stage being added.
     */
    public void insertStage(Stage stage){
        if(allStages.size()==0){
```

```
58              allStages.add(stage);
59          }
60          else
if(allStages.get(allStages.size()-1).getStartTime().isBefore(stage.getStartTime()))
{//If the stage being added is after the current last stage it is appended.
61              allStages.add(stage);
62          }
63          else{
64              for(int i=0;i<allStages.size();i++){//Inserts at position i when it finds
that current position i is after the stage to be added.
65                  if(stage.getStartTime().isBefore(allStages.get(i).getStartTime())){
66                      allStages.add(i,stage);
67                      break;
68                  }
69              }
70          }
71      }
72
73      /**
74       * Removes a stage from the race.
75       * @param stage   The stage you want to remove.
76       */
77      public void removeStage(Stage stage){
78          allStages.remove(stage);
79      }
80
81      /**
82       * Used to find a stage within the race
83       * @param id   The id of the stage you want to find
84       * @return If the stage is found in the race, the corresponding stage object is
returned.
85       *         If not, a null value is returned.
86       */
87      public Stage findStage(int id){
88          ArrayList<Stage> stages=getAllStages();//List of all created stages
89          for(int i=0; i<stages.size();i++){
90              if (stages.get(i).getId()==id){//Comparing search id and id of stage
being iterated
91                  Stage stage=stages.get(i);
92                  return stage;
93              }
94          }
95          return null;
96      }
97
98
99      /**
100      * Creates a new race object
101      * @param id   The id of the race to be created
102      * @param name   The name of the race to be created
103      * @param desc   The description of the race to be created
104      */
105     public Race(int id, String name, String desc){
106         this.id=id;
107         this.name=name;
108         this.desc=desc;
109         allStages = new ArrayList<Stage>();
110     }
111 }
112
```

```java
package cycling;

import java.util.ArrayList;
import java.io.Serializable;
import java.time.LocalDateTime;
import java.time.LocalTime;
import cycling.StageType;

/**
 * Stage --- A class to represent a stage in a race. Can contain intermediate sprints
 * and climb checkpoints.
 * Contains attributes:
 * segments(ArryList<Segment>) - All the segments that are in a stage
 * type(StageType) - An enum which can be FLAT,MEDIUM_MOUNTAIN,HIGH_MOUNTAIN,TT
 * id(int) - Unique to each stage
 * name(String) - The name of the stage
 * desc(String) - The description of the stage
 * length(double) - The length of the stage
 * startTime(LocalDateTime) - The date and time at which the stage starts
 * state(String) - Represents if the stage is in the prep phase or waiting for results
 * finishResults(ArrayList<RiderResult>) - An ArrayList of the times at which riders finished the stage
 * startTimes(ArrayList<RiderResult>) - An ArrayList of the times at which riders started the stage
 *
 * @author Matt Trenchard
 * @version 1.0
 */
public class Stage implements Serializable{
    private ArrayList<Segment> segments;
    /**
     * Gets all the segments from the stage
     * @return All the segments in the stage
     */
    public ArrayList<Segment> getSegments(){
        return segments;
    }
    private StageType type;
    /**
     * Gets the type of the stage
     * @return The stage type
     */
    public StageType getType(){
        return type;
    }
    private int id;
    /**
     * Gets the ID of the stage
     * @return The stage ID
     */
    public int getId(){
        return id;
    }
    private String name;
    /**
     * Gets the name of the stage
     * @return Name of the stage
     */
```

```
57      public String getName(){
58          return name;
59      }
60      private String desc;
61      /**
62       * Gets the stage description
63       * @return The stage description
64       */
65      public String getDesc(){
66          return desc;
67      }
68      private double length;
69      /**
70       * Gets the length of the stage
71       * @return The stage length
72       */
73      public double getLength(){
74          return length;
75      }
76      private LocalDateTime startTime;
77      /**
78       * Gets the date and time of the stage start
79       * @return Start time of the stage
80       */
81      public LocalDateTime getStartTime(){
82          return startTime;
83      }
84      private String state;
85      /**
86       * Gets the current stage state
87       * @return The stage's state. Either "wait" or "prep".
88       */
89      public String getState(){
90          return state;
91      }
92      private ArrayList<RiderResult> finishResults;
93      /**
94       * Gets all the finish times for the stage
95       * @return ArrayList of finish times
96       */
97      public ArrayList<RiderResult> getFinishResults(){
98          return finishResults;
99      }
100     private ArrayList<RiderResult> startTimes;
101     /**
102      * Gets all the rider start times for the stage
103      * @return ArrayList of start times
104      */
105     public ArrayList<RiderResult> getStartTimes(){
106         return startTimes;
107     }
108
109     /**
110      * Used to add a start time of a rider to the stage. It is inserted based on the
    time.
111      * @param result   The rider result being added
112      */
113     public void insertStartTime(RiderResult result){
114         if(startTimes.size()==0){
115             startTimes.add(result);
```

```
116         }
117         else
    if(startTimes.get(startTimes.size()-1).getTime().isBefore(result.getTime())){//If the
    time to be added is the slowest it is appended
118             startTimes.add(result);
119         }
120         else
    if(startTimes.get(startTimes.size()-1).getTime().equals(result.getTime())){//If the
    time is equal to the slowest it is appened
121             startTimes.add(result);
122         }
123         else{
124             for(int i=0;i<startTimes.size();i++){//Iterates through all current times
    until a time that is equal or after the time to be added is found
125                 if(result.getTime().isBefore(startTimes.get(i).getTime())){
126                     startTimes.add(i,result);
127                     break;
128                 }
129                 else if(result.getTime().equals(startTimes.get(i).getTime())){
130                     startTimes.add(i,result);
131                     break;
132                 }
133             }
134         }
135     }
136 
137     /**
138      * Used to find a rider's result in a stage
139      * @param riderId   The ID of the rider whose result you want to find
140      * @return The RiderResult object corresponding to the riderId.
141      *         If no result is found a null value is returned
142      */
143     public RiderResult findRiderResult(int riderId){
144         for(int i=0;i<finishResults.size();i++){
145             if(finishResults.get(i).getRiderId()==riderId){
146                 return finishResults.get(i);
147             }
148         }
149         return null;
150     }
151 
152     /**
153      * Used to find a rider's start time in a stage
154      * @param riderId   The ID of the rider whose start time you want to find
155      * @return The RiderResult object corresponding to the riderId.
156      *         If no start time is found a null value is returned
157      */
158     public RiderResult findRiderStart(int riderId){
159         for(int i=0;i<startTimes.size();i++){
160             if(startTimes.get(i).getRiderId()==riderId){
161                 return startTimes.get(i);
162             }
163         }
164         return null;
165     }
166 
167     /**
168      * Inserts a rider's finishing result. Result is inserted based on time.
169      * @param result   The finishing result to be inserted
170      */
```

```
171     public void insertFinish(RiderResult result){
172         if(finishResults.size()==0){
173             finishResults.add(result);
174         }
175         else
if(finishResults.get(finishResults.size()-1).getTime().isBefore(result.getTime()))
{//If the time to be added is the slowest it is appended
176             finishResults.add(result);
177         }
178         else
if(finishResults.get(finishResults.size()-1).getTime().equals(result.getTime())){//If
the time is equal to the slowest it is appened
179             finishResults.add(result);
180         }
181         else{
182             for(int i=0;i<finishResults.size();i++){//Iterates through all current
times until a time that is equal or after the time to be added is found
183                 if(result.getTime().isBefore(finishResults.get(i).getTime())){
184                     finishResults.add(i,result);
185                     break;
186                 }
187                 else if(result.getTime().equals(finishResults.get(i).getTime())){
188                     finishResults.add(i,result);
189                     break;
190                 }
191             }
192         }
193     }
194 
195     /**
196      * Removes a rider's start time
197      * @param result   The start result to be removed
198      */
199     public void removeRiderStartTime(RiderResult result){
200         startTimes.remove(result);
201     }
202 
203     /**
204      * Removes a rider's finish time
205      * @param result   The finish time to be removed
206      */
207     public void removeFinishResult(RiderResult result){
208         finishResults.remove(result);
209     }
210 
211     /**
212      * Adds a segment to the stage. Segments are inserted based on their location
within the stage
213      * @param segment   The segment to be added
214      */
215     public void insertSegment(Segment segment){
216         if (segments.size()==0){
217             segments.add(segment);
218         }
219         else{
220             boolean sorted=false;
221             for (int i=0;i<segments.size();i++){//Inserts with insertion sort logic
222                 if (segment.getLocation()<segments.get(i).getLocation()){
223                     segments.add(i,segment);
224                     sorted=true;
```

```
                    break;
                }
            }
            if(sorted==false){//If it has not yet been added it means it is the last segment so is appended
                segments.add(segment);
            }
        }
    }

    /**
     * Chnages the stage state from preperation to waiting for results
     */
    public void concludePrep(){
        state="wait";
    }

    /**
     * Used to find a segment within a stage
     * @param id   The ID of the segment you want to find
     * @return If the segment is found, the segment object is returned.
     *         If not, a null value is returned
     */
    public Segment findSegment(int id){
        ArrayList<Segment> segments=getSegments();//List of all created segments
        for(int i=0; i<segments.size();i++){
            if (segments.get(i).getId()==id){//Comparing search id and id of segment being iterated
                Segment segment=segments.get(i);
                return segment;
            }
        }
        return null;
    }

    /**
     * Removes a segment from a stage
     * @param segment   The segment to be removed
     */
    public void removeSegment(Segment segment){
        segments.remove(segment);
    }


    /**
     * Creates a new stage
     * @param id   ID of the stage
     * @param name   Name of the stage
     * @param desc   Description of the stage
     * @param length    Length of the stage
     * @param startTime   The date and time of the stage start
     * @param type   The stage type
     */
    public Stage(int id, String name, String desc, double length, LocalDateTime startTime, StageType type){
        this.id=id;
        this.desc=desc;
        this.name=name;
        this.length=length;
        this.startTime=startTime;
```

```
282         this.type=type;
283         segments= new ArrayList<Segment>();
284         finishResults= new ArrayList<RiderResult>();
285         startTimes=new ArrayList<RiderResult>();
286         state="prep";
287     }
288 }
289
```

```java
package cycling;

import java.io.Serializable;
import java.time.LocalTime;
import java.util.ArrayList;
import cycling.SegmentType;

/**
 * Segment --- A class to represent an intermediate sprint or climb within a stage
 * Contains attributes:
 * id(int) - The ID of the segment
 * location(double) - The location in km of the climb peak or sprint checkpoint
 * type(SegmentType) - An enum which can be SPRINT,C4,C3,C2,C1,HC
 * avgGrad(double) - The average gradient of the segment. If it's a sprint this value will be 0
 * checkpointResults(ArrayList<RiderResult>) - A list of each riders time they reached the segment checkpoint at
 */
public class Segment implements Serializable{
    private int id;
    /**
     * Gets the ID of the segment
     * @return The segment ID
     */
    public int getId(){
        return id;
    }
    private double location;
    /**
     * Gets the location of the segment in the stage
     * @return The location of the segment
     */
    public double getLocation(){
        return location;
    }
    private SegmentType type;
    /**
     * Gets the segment type
     * @return The segment type
     */
    public SegmentType getType(){
        return type;
    }
    private double avgGrad;
    /**
     * Gets the average gradient of the segment
     * @return Average gradient of the segment
     */
    public double getAvgGrad(){
        return avgGrad;
    }

    private ArrayList<RiderResult> checkpointResults;
    /**
     * Gets all the results of riders at the segment checkpoint
     * @return ArrayList of rider's checkpoint times
     */
    public ArrayList<RiderResult> getCheckpointResults(){
        return checkpointResults;
```

```
58      }
59
60      /**
61       * Adds a rider's time they reached the segment checkpoint at. Inserts based on
    time
62       * @param result
63       */
64      public void insertCheckpoint(RiderResult result){
65          if(checkpointResults.size()==0){
66              checkpointResults.add(result);
67          }
68          else if
    (checkpointResults.get(checkpointResults.size()-1).getTime().isBefore(result.getTime(
    ))){//If the time to be added is the slowest it is appended
69              checkpointResults.add(result);
70          }
71          else if
    (checkpointResults.get(checkpointResults.size()-1).getTime().equals(result.getTime())
    ){//If the time is equal to the slowest it is appened
72              checkpointResults.add(result);
73          }
74          else{
75              for(int i=0;i<checkpointResults.size();i++){//Iterates through all
    current times until a time that is equal or after the time to be added is found
76                  if(result.getTime().isBefore(checkpointResults.get(i).getTime())){
77                      checkpointResults.add(i,result);
78                      break;
79                  }
80                  else if(result.getTime().equals(checkpointResults.get(i).getTime())){
81                      checkpointResults.add(i,result);
82                      break;
83                  }
84              }
85          }
86      }
87
88      /**
89       * Removes a rider's time from the list of recorded times
90       * @param result   The result to be removed
91       */
92      public void removeCheckpointResult(RiderResult result){
93          checkpointResults.remove(result);
94      }
95
96      /**
97       * Used to find a rider's result from a segment
98       * @param riderId   The rider you want to find a result for
99       * @return A null value if the result isn't found.
100      *         If the result is found the result is returned
101      */
102     public RiderResult findRiderResult(int riderId){
103         for(int i=0;i<checkpointResults.size();i++){
104             if(checkpointResults.get(i).getRiderId()==riderId){
105                 return checkpointResults.get(i);
106             }
107         }
108         return null;
109     }
110
111
```

```
112     /**
113      * Creates a new segment. This constructor contains an avgerage gradient
parameter as it is
114      * used for creating climb segments
115      * @param id   The ID of the segment to be created
116      * @param location   The location of the segment in the stage
117      * @param avgGrad   The avergae gradient of the climb
118      * @param type   The type of the climb. C4,C3,C2,C1 or HC
119      */
120     public Segment(int id, double location, double avgGrad, SegmentType type){
121         this.id=id;
122         this.location=location;
123         this.avgGrad=avgGrad;
124         this.type=type;
125         checkpointResults= new ArrayList<RiderResult>();
126     }
127 
128     /**
129      * Creates a new segment. This constructor contains no avgerage gradient
parameter as it is
130      * used for creating sprint segments
131      * @param id   The ID of the segment to be created
132      * @param location   The location of the segment in the stage
133      * @param type   The type of the segment
134      */
135     public Segment(int id, double location, SegmentType type){
136         this.id=id;
137         this.location=location;
138         this.type=type;
139         avgGrad=0;
140         checkpointResults= new ArrayList<RiderResult>();
141     }
142 }
143 
```

```java
package cycling;

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

/**
 * Team --- A class to represent a team which contains riders.
 * Contains the following attributes:
 * name (String) - The name of the team
 * desc (String) - A description of the team
 * id (int) - Unique to each team
 * riders (ArrayList<Rider>) - An ArrayList containing every rider that rides for the team
 *
 * @author Matt Trenchard
 * @version 1.0
 */
public class Team implements Serializable{
    private String name;
    /**
     * Gets a teams name.
     * @return Team's name.
     */
    public String getName(){
        return name;
    }
    private String desc;
    /**
     * Gets a team's description.
     * @return Team's description.
     */
    public String getDesc(){
        return desc;
    }
    private int id;
    /**
     * Gets a team's ID.
     * @return Team's ID.
     */
    public int getId(){
        return id;
    }
    private ArrayList<Rider> riders;
    /**
     * Gets a list of every rider riding for the team.
     * @return ArrayList of Riders riding for the team.
     */
    public ArrayList<Rider> getRiders(){
        return riders;
    }


    /**
     * Adds a rider to the ArrayList of riders for a team.
     * @param rider   A Rider object to be added.
     */
    public void appendRider(Rider rider){
        riders.add(rider);
    }
```

```
59
60      /**
61       * Removes a rider from the ArrayList of riders for a team
62       * @param rider   A Rider object to be removed.
63       */
64      public void deleteRider(Rider rider){
65          riders.remove(rider);
66      }
67
68
69      /**
70       * Creates a team with the specified parameters and an empty ArrayList of riders.
71       * @param name   Name of the team.
72       * @param desc   The description of the team.
73       * @param id   Unique id of the team.
74       */
75      public Team(String name, String desc, int id){
76          this.name=name;
77          this.desc=desc;
78          riders= new ArrayList<Rider>();
79          this.id=id;
80      }
81  }
82
```

```java
package cycling;

import java.io.Serializable;

/**
 * Rider --- A class to represent a rider.
 * Contains attributes:
 * id (int) - Unique to each rider
 * name (string) - The name of the rider
 * yearOfBirth (int) - The year the rider was born
 *
 * @author Matt Trenchard
 * @version 1.0
 */

public class Rider implements Serializable{
    private int id;
    /**
     * Gets a rider's ID.
     * @return Rider's id.
     */
    public int getId(){
        return id;
    }
    private String name;
    /**
     * Gets a rider's name.
     * @return Rider's name.
     */
    public String getName(){
        return name;
    }
    private int yearOfBirth;
    /**
     * Gets a rider's birth year.
     * @return Rider's year of birth.
     */
    public int getYearOfBirth(){
        return yearOfBirth;
    }

    /**
     * Creates a rider with the specified parameters.
     * @param name   Name of the rider.
     * @param yearOfBirth   Year the rider was born.
     * @param id   The unique id of the rider.
     */
    public Rider(String name, int yearOfBirth, int id){
        this.id=id;
        this.name=name;
        this.yearOfBirth=yearOfBirth;
    }
}
```

```java
package cycling;

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

/**
 * RiderResult --- A class to hold a riders result at a start, checkpoint or finish line.
 * Contains the following attributes:
 * riderId(int) - The ID of the rider whom the time corresponds to
 * time(LocalTime) - The time at which the rider reached the start,checkpoint or finish.
 *
 * @author Matt Trenchard
 * @version 1.0
 */
public class RiderResult implements Serializable{
    private int riderId;
    /**
     * Gets the id of the rider whos result is stored
     * @return The rider's ID
     */
    public int getRiderId(){
        return riderId;
    }
    private LocalTime time;
    /**
     * Gets the time stored in the result.
     * @return The time stored
     */
    public LocalTime getTime(){
        return time;
    }

    /**
     * Creates a new result
     * @param riderId ID of the rider to create a result for
     * @param time The time for the result
     */
    public RiderResult(int riderId,LocalTime time){
        this.riderId=riderId;
        this.time=time;
    }
}

```