1 CyclingPortal.java

```java
package cycling;

import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.time.LocalDateTime;
import java.time.LocalTime;

import java.util.HashMap;

/**
 * CyclingPortal implements all of the functions in the CyclingPortalInterface.
 *
 * @author Charlie Goldstraw, Charlie MacDonald-Smith
 * @version 1.0
 *
 */
public class CyclingPortal implements CyclingPortalInterface {

   private int nextId = 0;
   private HashMap<Integer, Race> races = new HashMap<Integer, Race>();
   private HashMap<Integer, Team> teams = new HashMap<Integer, Team>();

   /**
    * Get a Race object by its ID.
    *
    * @param id Race's ID.
    * @throws IDNotRecognisedException If the ID does not match to any race in the
    *                                  system.
    * @return The Race object with the given ID.
    *
    */
   public Race getRace(int id) throws IDNotRecognisedException {
      if (!races.containsKey(id)) {
         String errorMessage = String.format("Race ID '%d' did not exist.", id);
         throw new IDNotRecognisedException(errorMessage);
      }
      return races.get(id);
   }

   /**
    * Get a Stage object by its ID.
    *
    * @param id Stage's ID.
    * @throws IDNotRecognisedException If the ID does not match to any stage in the
    *                                  system.
    * @return The Stage object with the given ID.
    *
```

```
53      */
54     public Stage getStage(int id) throws IDNotRecognisedException {
55        for (Race race : races.values()) {
56           for (Stage stage : race.getStages()) {
57              if (stage.getId() == id) {
58                 return stage;
59              }
60           }
61        }
62        String errorMessage = String.format("Stage ID '%d' did not exist.", id);
63        throw new IDNotRecognisedException(errorMessage);
64     }
65
66     /**
67      * Get a Segment object by its ID.
68      *
69      * @param id Segment's ID.
70      * @throws IDNotRecognisedException If the ID does not match to any segment in the
71      *                                  system.
72      * @return The Segment object with the given ID.
73      *
74      */
75     public Segment getSegment(int id) throws IDNotRecognisedException {
76        for (Race race : races.values()) {
77           for (Stage stage : race.getStages()) {
78              for (Segment segment : stage.getSegments()) {
79                 if (segment.getId() == id) {
80                    return segment;
81                 }
82              }
83           }
84        }
85        String errorMessage = String.format("Segment ID '%d' did not exist.", id);
86        throw new IDNotRecognisedException(errorMessage);
87     }
88
89     /**
90      * Get a Team object by its ID.
91      *
92      * @param id Team's ID.
93      * @throws IDNotRecognisedException If the ID does not match to any team in the
94      *                                  system.
95      * @return The Team object with the given ID.
96      *
97      */
98     public Team getTeam(int id) throws IDNotRecognisedException {
99        if (!teams.containsKey(id)) {
100          String errorMessage = String.format("Team ID '%d' did not exist.", id);
101          throw new IDNotRecognisedException(errorMessage);
102       }
103       return teams.get(id);
104    }
105
106    /**
107     * Get a Rider object by its ID.
```

```
108      *
109      * @param id Rider's ID.
110      * @throws IDNotRecognisedException If the ID does not match to any rider in the
111      *                                  system.
112      * @return The Rider object with the given ID.
113      *
114      */
115     public Rider getRider(int id) throws IDNotRecognisedException {
116        for (Team team : teams.values()) {
117           for (Rider rider : team.getRiders()) {
118              if (rider.getId() == id) {
119                 return rider;
120              }
121           }
122        }
123        String errorMessage = String.format("Rider ID '%d' did not exist.", id);
124        throw new IDNotRecognisedException(errorMessage);
125     }
126
127     /**
128      * Perform checks upon a name to ensure it is unique in the system, it is not empty,
129      * it is shorter than 30 characters, and it doesn't have any spaces.
130      *
131      * @param name Name to validate.
132      * @throws IllegalNameException If the name already exists in the system.
133      * @throws InvalidNameException If the name does not match the formatting required.
134      *
135      */
136     public void validateName(String name) throws IllegalNameException, InvalidNameException {
137        boolean usedName = false;
138        for (Race race : races.values()) {
139           if (race.getName().equals(name)) {
140              usedName = true;
141           }
142           for (Stage stage : race.getStages()) {
143              if (stage.getName().equals(name)) {
144                 usedName = true;
145              }
146           }
147        }
148        for (Team team : teams.values()) {
149           if (team.getName().equals(name)) {
150              usedName = true;
151           }
152           for (Rider rider : team.getRiders()) {
153              if (rider.getName().equals(name)) {
154                 usedName = true;
155              }
156           }
157        }
158
159        if (usedName) {
160           String errorMessage = String.format("The name '%s' already exists.", name);
161           throw new IllegalNameException(errorMessage);
162        }
```

```
163         if (name == null || name.length() == 0) {
164             throw new InvalidNameException("The name was empty.");
165         }
166         if (name.length() > 30) {
167             throw new InvalidNameException("The name was too long. (30 char limit).");
168         }
169         if (name.contains(" ")) {
170             throw new InvalidNameException("The name contains spaces.");
171         }
172     }
173 
174     /**
175      * Check if a proposed segment is within the stage's boundaries, and the relevant
176      * stage is not a time trial or "waiting for results".
177      *
178      * @param stageId The ID of the stage.
179      * @param location The location of the end of segment.
180      * @param type The SegmentType of the proposed segment.
181      * @param length The length of the segment.
182      * @throws IDNotRecognisedException If the ID does not match to any stage in the
183      *                                  system.
184      * @throws InvalidLocationException If the segment is not within the stage's bounds.
185      * @throws InvalidStageStateException If the stage is "waiting for results".
186      * @throws InvalidStageTypeException If the stage is a time trial.
187      *
188      */
189     public void validateSegmentAddition(int stageId, Double location, SegmentType type,
190             Double length) throws IDNotRecognisedException, InvalidLocationException,
                 InvalidStageStateException,
191             InvalidStageTypeException {
192         Stage stage = getStage(stageId);
193         if (location > stage.getLength() || location-length < 0) {
194             String errorMessage = "The location of the segment was invalid.";
195             throw new InvalidLocationException(errorMessage);
196         }
197         stage.assertNotWaitingForResults();
198         if (stage.getStageType() == StageType.TT) {
199             String errorMessage = "Time trials cannot have segments.";
200             throw new InvalidStageTypeException(errorMessage);
201         }
202     }
203 
204     @Override
205     public int[] getRaceIds() {
206         int[] raceIds = new int[races.size()];
207         int i = 0;
208         for (int id : races.keySet()) {
209             raceIds[i] = id;
210             i++;
211         }
212         return raceIds;
213     }
214 
215     @Override
216     public int createRace(String name, String description) throws IllegalNameException, InvalidNameException
```

```
        {
217         validateName(name);
218
219         int raceId = nextId++;
220         Race newRace = new Race(raceId, name, description);
221         races.put(raceId, newRace);
222         return raceId;
223     }
224
225     @Override
226     public String viewRaceDetails(int raceId) throws IDNotRecognisedException {
227         Race race = getRace(raceId);
228         double raceLength = 0;
229         for (Stage stage : race.getStages()) {
230             raceLength += stage.getLength();
231         }
232         String details = "";
233         details += String.format("Race ID : %d\n", raceId);
234         details += String.format("Race Name : %s\n", race.getName());
235         details += String.format("Description : %s\n", race.getDescription());
236         details += String.format("Num. of Stages : %d\n", race.getStages().length);
237         details += String.format("Total Length : %.2f", raceLength);
238
239         return details;
240     }
241
242     @Override
243     public void removeRaceById(int raceId) throws IDNotRecognisedException {
244         Race race = getRace(raceId);
245
246         for (Stage stage : race.getStages()) {
247             race.removeStage(stage);
248         }
249         races.remove(race.getId());
250     }
251
252     @Override
253     public int getNumberOfStages(int raceId) throws IDNotRecognisedException {
254         Race race = getRace(raceId);
255
256         return race.getStages().length;
257     }
258
259     @Override
260     public int addStageToRace(int raceId, String stageName, String description, double length, LocalDateTime
        startTime,
261             StageType type)
262             throws IDNotRecognisedException, IllegalNameException, InvalidNameException, InvalidLengthException
                {
263
264         Race race = getRace(raceId);
265         validateName(stageName);
266         if (stageName.length() > 30) {
267             throw new InvalidLengthException("The stage name was too long. (30 char limit).");
268         }
```

```
269         if (length < 5) {
270             throw new InvalidLengthException("The stage was too short (5km minimum).");
271         }
272         int stageId = nextId++;
273         Stage stage = new Stage(raceId, stageId, stageName, description, length, startTime, type);
274         race.addStage(stage);
275 
276         return stageId;
277     }
278 
279     @Override
280     public int[] getRaceStages(int raceId) throws IDNotRecognisedException {
281         Race race = getRace(raceId);
282         return race.getStageIds();
283     }
284 
285     @Override
286     public double getStageLength(int stageId) throws IDNotRecognisedException {
287         Stage stage = getStage(stageId);
288         return stage.getLength();
289     }
290 
291     @Override
292     public void removeStageById(int stageId) throws IDNotRecognisedException {
293         Stage stage = getStage(stageId);
294         int raceId = stage.getRaceId();
295         races.get(raceId).removeStage(stage);
296     }
297 
298     @Override
299     public int addCategorizedClimbToStage(int stageId, Double location, SegmentType type, Double
        averageGradient,
300             Double length) throws IDNotRecognisedException, InvalidLocationException,
                InvalidStageStateException,
301             InvalidStageTypeException {
302 
303         validateSegmentAddition(stageId, location, type, length);
304         Stage stage = getStage(stageId);
305         int segmentId = nextId++;
306         CategorizedClimb climb = new CategorizedClimb(stageId, segmentId, length, location, averageGradient,
            type);
307         stage.addSegment(climb);
308 
309         return segmentId;
310     }
311 
312     @Override
313     public int addIntermediateSprintToStage(int stageId, double location) throws IDNotRecognisedException,
314         InvalidLocationException, InvalidStageStateException, InvalidStageTypeException {
315 
316         validateSegmentAddition(stageId, location, SegmentType.SPRINT, 0d);
317         Stage stage = getStage(stageId);
318         int segmentId = nextId++;
319         IntermediateSprint sprint = new IntermediateSprint(stageId, segmentId, location, SegmentType.SPRINT);
320         stage.addSegment(sprint);
```

```
321
322         return segmentId;
323     }
324
325     @Override
326     public void removeSegment(int segmentId) throws IDNotRecognisedException, InvalidStageStateException {
327         Segment segment = getSegment(segmentId);
328         int stageId = segment.getStageId();
329         Stage stage = getStage(stageId);
330         stage.assertNotWaitingForResults();
331         stage.removeSegment(segment);
332
333     }
334
335     @Override
336     public void concludeStagePreparation(int stageId) throws IDNotRecognisedException,
            InvalidStageStateException {
337         Stage stage = getStage(stageId);
338         stage.assertNotWaitingForResults();
339         stage.setState("waiting for results");
340     }
341
342     @Override
343     public int[] getStageSegments(int stageId) throws IDNotRecognisedException {
344         Stage stage = getStage(stageId);
345         return stage.getSegmentIds();
346     }
347
348     @Override
349     public int createTeam(String name, String description) throws IllegalNameException, InvalidNameException
            {
350         validateName(name);
351         int teamId = nextId++;
352         Team team = new Team(teamId, name, description);
353         teams.put(teamId, team);
354         return teamId;
355     }
356
357     @Override
358     public void removeTeam(int teamId) throws IDNotRecognisedException {
359         getTeam(teamId);
360         teams.remove(teamId);
361     }
362
363     @Override
364     public int[] getTeams() {
365         int[] teamIds = new int[teams.size()];
366         int i = 0;
367         for (Team team : teams.values()) {
368             teamIds[i] = team.getId();
369             i++;
370         }
371         return teamIds;
372     }
373
```

```
374     @Override
375     public int[] getTeamRiders(int teamId) throws IDNotRecognisedException {
376        Team team = getTeam(teamId);
377        return team.getRiderIds();
378     }
379
380     @Override
381     public int createRider(int teamID, String name, int yearOfBirth)
382           throws IDNotRecognisedException, IllegalArgumentException {
383        if (name == null || name.length() == 0) {
384           throw new IllegalArgumentException("The rider's name was empty.");
385        }
386        if (yearOfBirth < 1900) {
387           throw new IllegalArgumentException("The year of birth was invalid, it must be 1900 or later.");
388        }
389        Team team = getTeam(teamID);
390
391        int riderId = nextId++;
392        Rider rider = new Rider(riderId, teamID, name, yearOfBirth);
393        team.addRider(rider);
394
395        return riderId;
396     }
397
398     @Override
399     public void removeRider(int riderId) throws IDNotRecognisedException {
400        Rider rider = getRider(riderId);
401        Team team = getTeam(rider.getTeamId());
402        team.removeRider(rider);
403     }
404
405     @Override
406     public void registerRiderResultsInStage(int stageId, int riderId, LocalTime... checkpoints)
407           throws IDNotRecognisedException, DuplicatedResultException, InvalidCheckpointsException,
408           InvalidStageStateException {
409        getRider(riderId);
410        Stage stage = getStage(stageId);
411        if (stage.getSegments().length+2 != checkpoints.length) {
412           String errorMessage = "There were an invalid number of checkpoints.";
413           throw new InvalidCheckpointsException(errorMessage);
414        }
415        if (stage.getResults(riderId).length != 0) {
416           String errorMessage = "The rider already has results for this stage.";
417           throw new DuplicatedResultException(errorMessage);
418        }
419        stage.assertWaitingForResults();
420
421        stage.addResults(riderId, checkpoints);
422     }
423
424     @Override
425     public LocalTime[] getRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException {
426        getRider(riderId);
427        Stage stage = getStage(stageId);
428        return stage.getResults(riderId);
```

```
429     }
430
431     @Override
432     public LocalTime getRiderAdjustedElapsedTimeInStage(int stageId, int riderId) throws
            IDNotRecognisedException {
433        getRider(riderId);
434        Stage stage = getStage(stageId);
435
436        LocalTime elapsedTime = stage.getRiderAdjustedElapsedTime(riderId);
437        return elapsedTime;
438     }
439
440     @Override
441     public void deleteRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException {
442        getRider(riderId);
443        Stage stage = getStage(stageId);
444
445        stage.deleteResults(riderId);
446     }
447
448     @Override
449     public int[] getRidersRankInStage(int stageId) throws IDNotRecognisedException {
450        Stage stage = getStage(stageId);
451        return stage.getRidersRanks();
452     }
453
454     @Override
455     public LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId) throws IDNotRecognisedException {
456        Stage stage = getStage(stageId);
457        return stage.getRankedAdjustedTimes();
458     }
459
460     @Override
461     public int[] getRidersPointsInStage(int stageId) throws IDNotRecognisedException {
462        Stage stage = getStage(stageId);
463        return stage.getRidersPoints();
464     }
465
466     @Override
467     public int[] getRidersMountainPointsInStage(int stageId) throws IDNotRecognisedException {
468        Stage stage = getStage(stageId);
469        return stage.getRidersMountainPoints();
470     }
471
472     @Override
473     public void eraseCyclingPortal() {
474        this.nextId = 0;
475        this.teams.clear();
476        this.races.clear();
477     }
478
479     @Override
480     public void saveCyclingPortal(String filename) throws IOException {
481        FileOutputStream fileOutputStream = new FileOutputStream(filename);
482        BufferedOutputStream bufferedOutputStream = new BufferedOutputStream(fileOutputStream);
```

```
483
484         ObjectOutputStream objectOutputStream = new ObjectOutputStream(bufferedOutputStream);
485
486         objectOutputStream.writeObject(this);
487         objectOutputStream.close();
488     }
489
490     @Override
491     public void loadCyclingPortal(String filename) throws IOException, ClassNotFoundException {
492         FileInputStream fileInputStream = new FileInputStream(filename);
493         BufferedInputStream bufferedInputStream = new BufferedInputStream(fileInputStream);
494         ObjectInputStream objectInputStream = new ObjectInputStream(bufferedInputStream);
495         CyclingPortal loadedCyclingPortal = (CyclingPortal)objectInputStream.readObject();
496         objectInputStream.close();
497
498         this.nextId = loadedCyclingPortal.nextId;
499         this.teams = loadedCyclingPortal.teams;
500         this.races = loadedCyclingPortal.races;
501     }
502
503     @Override
504     public void removeRaceByName(String name) throws NameNotRecognisedException {
505         boolean found = false;
506         for (Race race : races.values()) {
507             if (race.getName().equals(name)) {
508                 races.remove(race.getId());
509                 found = true;
510                 break;
511             }
512         }
513
514         if (!found) {
515             String errorMessage = String.format("Race name '%s' did not exist.", name);
516             throw new NameNotRecognisedException(errorMessage);
517         } else {
518         }
519     }
520
521     @Override
522     public LocalTime[] getGeneralClassificationTimesInRace(int raceId) throws IDNotRecognisedException {
523         Race race = getRace(raceId);
524         return race.getGeneralClassificationTimes();
525     }
526
527     @Override
528     public int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException {
529         Race race = getRace(raceId);
530         return race.getRidersPoints();
531     }
532
533     @Override
534     public int[] getRidersMountainPointsInRace(int raceId) throws IDNotRecognisedException {
535         Race race = getRace(raceId);
536         return race.getRidersMountainPoints();
537     }
```

```
538
539     @Override
540     public int[] getRidersGeneralClassificationRank(int raceId) throws IDNotRecognisedException {
541         Race race = getRace(raceId);
542         return race.getRidersGeneralClassificationRank();
543     }
544
545     @Override
546     public int[] getRidersPointClassificationRank(int raceId) throws IDNotRecognisedException {
547         Race race = getRace(raceId);
548         return race.getRidersPointClassificationRank();
549     }
550
551     @Override
552     public int[] getRidersMountainPointClassificationRank(int raceId) throws IDNotRecognisedException {
553         Race race = getRace(raceId);
554         return race.getRidersMountainPointClassificationRank();
555     }
556
557 }
```

2 Team.java

```
1  package cycling;
2
3  import java.io.Serializable;
4  import java.util.ArrayList;
5
6  public class Team implements Serializable {
7      private int teamId;
8      private String name;
9      private String description;
10     private ArrayList<Rider> riders = new ArrayList<Rider>();
11
12     public Team(int teamId, String name, String description) {
13         this.teamId = teamId;
14         this.name = name;
15         this.description = description;
16     }
17
18     /**
19     * Add a Rider to the team.
20     *
21     * @param rider The Rider object to add.
22     *
23     */
24     public void addRider(Rider rider) {
25         this.riders.add(rider);
26     }
27
28     /**
29     * Remove a Rider from the team.
30     *
31     * @param rider The Rider object to remove.
```

```
32      *
33      */
34      public void removeRider(Rider rider) {
35          this.riders.remove(rider);
36      }
37
38      /**
39      * Returns an array of riders in the team.
40       *
41      * @return The Rider array containing the team riders.
42      *
43      */
44      public Rider[] getRiders() {
45          return this.riders.toArray(new Rider[0]);
46      }
47
48      /**
49      * Get an array of the teams rider IDs.
50      *
51      * @return The IDs of the riders in an array.
52      *
53      */
54      public int[] getRiderIds() {
55          int[] riderIds = new int[this.riders.size()];
56          for (int i = 0; i < riderIds.length; i++) {
57              riderIds[i] = this.riders.get(i).getId();
58          }
59          return riderIds;
60      }
61
62      /**
63      * Get the name of the team.
64      *
65      * @return The String containing the name of the team.
66      *
67      */
68      public String getName() {
69          return this.name;
70      }
71
72      /**
73      * Get the ID of the team.
74      *
75      * @return The int of the team's ID.
76      *
77      */
78      public int getId() {
79          return this.teamId;
80      }
81  }
```

3 Rider.java

```
1   package cycling;
```

```java

import java.io.Serializable;

public class Rider implements Serializable {
    private int riderId;
    private int teamId;
    private String name;
    private int yearOfBirth;

    public Rider(int riderId, int teamId, String name, int yearOfBirth) {
        this.riderId = riderId;
        this.teamId = teamId;
        this.name = name;
        this.yearOfBirth = yearOfBirth;
    }

    /**
    * Get the rider's team ID.
    *
    * @return The rider's team ID.
    *
    */
    public int getTeamId() {
        return this.teamId;
    }

    /**
    * Get the rider's ID.
    *
    * @return The rider's ID.
    *
    */
    public int getId() {
        return this.riderId;
    }

    /**
    * Get the rider's name.
    *
    * @return The rider's name.
    *
    */
    public String getName() {
        return this.name;
    }
}
```

4 Race.java

```java
package cycling;

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

```
6
7  public class Race implements Serializable {
8      private int raceId;
9      private String name;
10     private String description;
11     private ArrayList<Stage> stages = new ArrayList<Stage>();
12
13     public Race(int raceId, String name, String description) {
14         this.raceId = raceId;
15         this.description = description;
16         this.name = name;
17     }
18
19     /**
20     * Get the name of the race.
21     *
22     * @return The name of the race.
23     *
24     */
25     public String getName() {
26         return this.name;
27     }
28
29     /**
30     * Get the ID of the race.
31     *
32     * @return The ID of the race.
33     *
34     */
35     public int getId() {
36         return this.raceId;
37     }
38
39     /**
40     * Add a stage to the race.
41     *
42     * @param id Stage object to add.
43     *
44     */
45     public void addStage(Stage stage) {
46         this.stages.add(stage);
47     }
48
49     /**
50     * Removes a stage from the race.
51     *
52     * @param id Stage object to remove.
53     *
54     */
55     public void removeStage(Stage stage) {
56         this.stages.remove(stage);
57     }
58
59     /**
60     * Returns a list of stages in the race
```

```
61      *
62      * @return A list containing the race's stages.
63      *
64      */
65      public Stage[] getStages() {
66          return this.stages.toArray(new Stage[0]);
67      }
68
69      /**
70      * Returns a list of stage IDs in the race
71      *
72      * @return A list containing the race's stages' IDs.
73      *
74      */
75      public int[] getStageIds() {
76          int[] stageIds = new int[this.stages.size()];
77          for (int i = 0; i < stageIds.length; i++) {
78              stageIds[i] = this.stages.get(i).getId();
79          }
80          return stageIds;
81      }
82
83      /**
84      * Returns the description of the race
85      *
86      * @return A String of the description of the race.
87      *
88      */
89      public String getDescription() {
90          return this.description;
91      }
92
93      /**
94      * Return the array of indices which sorts the riders by their
95       * elapsed time when accessed in the order of the first stage's
96       * results.
97      *
98      * @return An integer array containing the indices which sort
99       * the riders by elapsed time.
100     *
101     */
102     private int[] getSortedElapsedTimeIndices() {
103       ArrayList<Long> results = new ArrayList<Long>();
104         ArrayList<Integer> sortedIndices = new ArrayList<Integer>();
105         int unsortedIndex = 0;
106         for (Integer riderId : this.stages.get(0).getRidersRanks()) {
107             long elapsedTime = 0;
108             for (Stage stage : this.stages) {
109                 elapsedTime += stage.getRiderAdjustedElapsedTime(riderId).toNanoOfDay();
110             }
111             int index = 0;
112             for (index = 0; index < results.size(); index++) {
113                 if (results.get(index) > elapsedTime) {
114                     break;
115                 }
```

```
            }
            results.add(index, elapsedTime);
            sortedIndices.add(index, unsortedIndex);
            unsortedIndex++;
        }

        int[] sortedArr = new int[sortedIndices.size()];
        for (int i = 0; i < sortedIndices.size(); i++) {
            sortedArr[i] = sortedIndices.get(i).intValue();
        }
        return sortedArr;
    }

    /**
    * Return the array of general classification times for riders
     * sorted by the riders' elapsed times.
    *
    * @return An LocalTime array containing the GC times of the riders.
    *
    */
    public LocalTime[] getGeneralClassificationTimes() {
        int[] order = getSortedElapsedTimeIndices();
        LocalTime[] times = new LocalTime[order.length];
        int index = 0;
        for (Integer riderId : this.stages.get(0).getRidersRanks()) {
            long elapsedTime = 0;
            for (Stage stage : this.stages) {
                elapsedTime += stage.getRiderAdjustedElapsedTime(riderId).toNanoOfDay();
            }
            times[order[index]] = LocalTime.ofNanoOfDay(elapsedTime);
            index++;
        }
        return times;
    }

    /**
    * Return the array of points for riders sorted by the riders'
     * elapsed times.
    *
    * @return A LocalTime array containing the points of the riders.
    *
    */
    public int[] getRidersPoints() {
        int[] order = getSortedElapsedTimeIndices();
        int[] points = new int[order.length];
        int index = 0;
        for (Integer riderId : this.stages.get(0).getRidersRanks()) {
            int riderPoints = 0;
            for (Stage stage : this.stages) {
                riderPoints += stage.getRiderPoints(riderId);
            }
            points[order[index]] = riderPoints;
            index++;
        }
        return points;
```

```
171     }
172
173     /**
174     * Return the array of mountain points for riders
175      * sorted by the riders' elapsed times.
176     *
177     * @return An int array containing the mountain points of the riders.
178     *
179     */
180     public int[] getRidersMountainPoints() {
181         int[] order = getSortedElapsedTimeIndices();
182         int[] points = new int[order.length];
183         int index = 0;
184         for (Integer riderId : this.stages.get(0).getRidersRanks()) {
185             int riderPoints = 0;
186             for (Stage stage : this.stages) {
187                 riderPoints += stage.getRiderMountainPoints(riderId);
188             }
189             points[order[index]] = riderPoints;
190             index++;
191         }
192         return points;
193     }
194
195     /**
196     * Return the array of rider IDs sorted by the riders' elapsed times.
197     *
198     * @return An int array containing the sorted rider IDs.
199     *
200     */
201     public int[] getRidersGeneralClassificationRank() {
202         int[] order = getSortedElapsedTimeIndices();
203         int[] ranks = new int[order.length];
204         int index = 0;
205         for (Integer riderId : this.stages.get(0).getRidersRanks()) {
206             ranks[order[index]] = riderId;
207             index++;
208         }
209         return ranks;
210     }
211
212     /**
213     * Return the array of rider IDs sorted in descending order
214      * by the riders' points.
215     *
216     * @return An int array containing the sorted rider IDs.
217     *
218     */
219     public int[] getRidersPointClassificationRank() {
220       ArrayList<Integer> points = new ArrayList<Integer>();
221         ArrayList<Integer> sortedIds = new ArrayList<Integer>();
222         for (Integer riderId : this.stages.get(0).getRidersRanks()) {
223             // Calculate points
224             int riderPoints = 0;
225             for (Stage stage : this.stages) {
```

```
226                 riderPoints += stage.getRiderPoints(riderId);
227             }
228             // Find sorted (descending) position in arraylist
229             int index = 0;
230             for (index = 0; index < points.size(); index++) {
231                 if (points.get(index) < riderPoints) {
232                     break;
233                 }
234             }
235             // Insert into arraylist
236             points.add(index, riderPoints);
237             sortedIds.add(index, riderId);
238         }
239 
240         int[] sortedArr = new int[sortedIds.size()];
241         for (int i = 0; i < sortedIds.size(); i++) {
242             sortedArr[i] = sortedIds.get(i).intValue();
243         }
244         return sortedArr;
245     }
246 
247     /**
248     * Return the array of rider IDs sorted in descending order
249      * by the riders' mountain points.
250     *
251     * @return An int array containing the sorted rider IDs.
252     *
253     */
254     public int[] getRidersMountainPointClassificationRank() {
255         ArrayList<Integer> points = new ArrayList<Integer>();
256         ArrayList<Integer> sortedIds = new ArrayList<Integer>();
257         for (Integer riderId : this.stages.get(0).getRidersRanks()) {
258             // Calculate points
259             int riderPoints = 0;
260             for (Stage stage : this.stages) {
261                 riderPoints += stage.getRiderMountainPoints(riderId);
262             }
263             // Find sorted (descending) position in arraylist
264             int index = 0;
265             for (index = 0; index < points.size(); index++) {
266                 if (points.get(index) < riderPoints) {
267                     break;
268                 }
269             }
270             // Insert into arraylist
271             points.add(index, riderPoints);
272             sortedIds.add(index, riderId);
273         }
274 
275         int[] sortedArr = new int[sortedIds.size()];
276         for (int i = 0; i < sortedIds.size(); i++) {
277             sortedArr[i] = sortedIds.get(i).intValue();
278         }
279         return sortedArr;
280     }
```

```
}
```

5 Stage.java

```
package cycling;

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

public class Stage implements Serializable {
    private int raceId;
    private int stageId;
    private String name;
    private String description;
    private double length;
    private LocalDateTime startTime;
    private StageType type;
    private String state;
    private ArrayList<Segment> segments = new ArrayList<Segment>();
    private LinkedHashMap<Integer, ArrayList<LocalTime>> results = new LinkedHashMap<Integer,
        ArrayList<LocalTime>>();

    public Stage(int raceId, int stageId, String name, String description, double length, LocalDateTime
        startTime, StageType type) {
        this.raceId = raceId;
        this.stageId = stageId;
        this.description = description;
        this.name = name;
        this.length = length;
        this.startTime = startTime;
        this.type = type;
        this.state = "preparation";
    }

    /**
    * Get the stage's ID.
    *
    * @return The stage's ID.
    *
    */
    public int getId() {
        return this.stageId;
    }

    /**
    * Get the stage's name.
    *
    * @return The stage's name.
    *
    */
```

```
49      public String getName() {
50          return this.name;
51      }
52
53      /**
54      * Get the race's ID.
55      *
56      * @return The stage's race ID.
57      *
58      */
59      public int getRaceId() {
60          return this.raceId;
61      }
62
63      /**
64      * Get the stage's length.
65      *
66      * @return The stage's length.
67      *
68      */
69      public double getLength() {
70          return this.length;
71      }
72
73      /**
74      * Get the stage's type.
75      *
76      * @return The stage's StageType.
77      *
78      */
79      public StageType getStageType() {
80          return this.type;
81      }
82
83      /**
84      * Get the stage's state.
85      *
86      * @return The stage's state.
87      *
88      */
89      public String getState() {
90          return this.state;
91      }
92
93      /**
94      * Add a segment to the stage.
95      *
96       * @param segment The Segment object to add.
97      *
98      */
99      public void addSegment(Segment segment) {
100         // Ensures that the segments are stored in chronological order
101         int sortedIndex = 0;
102         for (Segment comparison : this.segments) {
103             if (comparison.getLocation() > segment.getLocation()) {
```

```
                break;
            }
            sortedIndex++;
        }

        this.segments.add(sortedIndex, segment);
    }

    /**
    * Remove a segment from the stage.
    *
     * @param segment The Segment object to remove.
    *
    */
    public void removeSegment(Segment segment) {
        this.segments.remove(segment);
    }

    /**
    * Get the array of segments in the stage.
    *
     * @return The array of Segments in the stage.
    *
    */
    public Segment[] getSegments() {
        return this.segments.toArray(new Segment[0]);
    }

    /**
    * Get the array of segment IDs in the stage.
    *
     * @return The array of segment IDs in the stage.
    *
    */
    public int[] getSegmentIds() {
        int[] segmentIds = new int[this.segments.size()];
        for (int i = 0; i < segmentIds.length; i++) {
            segmentIds[i] = this.segments.get(i).getId();
        }
        return segmentIds;
    }

    /**
    * Set the state of the stage.
    *
     * @param state The state to change to.
    *
    */
    public void setState(String state) {
        this.state = state;
    }

   /**
    * Assert if the stage is not waiting for results.
    *
```

```
159      * @throws InvalidStageStateException If the stage is waiting for results.
160      *
161      */
162     public void assertNotWaitingForResults() throws InvalidStageStateException {
163         if (this.state.equals("waiting for results")) {
164          String errorMessage = "The stage was waiting for results.";
165          throw new InvalidStageStateException(errorMessage);
166       }
167     }
168 
169    /**
170     * Assert if the stage is waiting for results.
171     *
172     * @throws InvalidStageStateException If the stage is not waiting for results.
173     *
174     */
175     public void assertWaitingForResults() throws InvalidStageStateException {
176         // Ensure the stage is waiting for results, throw an
177         // InvalidStageStateException if it is.
178         if (!this.state.equals("waiting for results")) {
179          String errorMessage = "The stage was waiting for results.";
180          throw new InvalidStageStateException(errorMessage);
181       }
182     }
183 
184    /**
185     * Add a rider's results to the stage.
186     *
187     * @param riderId Rider's ID.
188      * @param checkpoints The LocalTime array of checkpoints.
189     *
190     */
191     public void addResults(int riderId, LocalTime[] checkpoints) {
192         ArrayList<LocalTime> resultList = new ArrayList<LocalTime>();
193         for (LocalTime result : checkpoints) {
194             resultList.add(result);
195         }
196       this.results.put(riderId, resultList);
197     }
198 
199    /**
200     * Delete a rider's results from the stage
201     *
202     * @param riderId Rider's ID.
203     *
204     */
205     public void deleteResults(int riderId) {
206       this.results.remove(riderId);
207     }
208 
209     /**
210     * Get a rider's results.
211     *
212     * @param riderId Rider's ID.
213     * @return A LocalTime array of the rider's results.
```

```
214     *
215     */
216     public LocalTime[] getResults(int riderId) {
217         if (!this.results.containsKey(riderId)) {
218             return new LocalTime[0];
219         }
220         ArrayList<LocalTime> riderResults = this.results.get(riderId);
221         LocalTime[] returnResults = new LocalTime[riderResults.size()-1];
222         for (int i = 1; i < riderResults.size()-1; i++) {
223             returnResults[i-1] = riderResults.get(i);
224         }
225         LocalTime elapsed = LocalTime.ofNanoOfDay(getRiderElapsedTime(riderId));
226         returnResults[riderResults.size()-2] = elapsed;
227         return returnResults;
228     }
229 
230     /**
231     * Get a Rider's elapsed time in the stage.
232     *
233     * @param riderId Rider's ID.
234     * @return The rider's elapsed time in nanoseconds.
235     *
236     */
237     public long getRiderElapsedTime(int riderId) {
238         assert (this.results.containsKey(riderId));
239         LocalTime startTime = this.results.get(riderId).get(0);
240         int endIndex = this.segments.size() + 1;
241         LocalTime endTime = this.results.get(riderId).get(endIndex);
242         long elapsedTime = endTime.toNanoOfDay() - startTime.toNanoOfDay();
243         if (elapsedTime < 0) {
244             elapsedTime += 24L*60L*60L*1000000000L;
245         }
246         return elapsedTime;
247     }
248 
249     /**
250     * Get a rider's adjusted elapsed time. If the rider finished within 1 second
251      * of another rider, then both rider's have the elapsed time of the quicker
252      * result.
253     *
254     * @param riderId Rider's ID.
255     * @return The Rider's adjusted elapsed time in nanoseconds.
256     *
257     */
258     public LocalTime getRiderAdjustedElapsedTime(int riderId) {
259         if (!this.results.containsKey(riderId)) {
260             return null;
261         }
262 
263         long elapsedTime = getRiderElapsedTime(riderId);
264         if (this.type == StageType.TT) {
265          return LocalTime.ofNanoOfDay(elapsedTime);
266       }
267         boolean timeAdjusted = false;
268       do {
```

```
269             timeAdjusted = false;
270             for (Integer comparisonRiderId : this.results.keySet()) {
271                   long otherElapsedTime = getRiderElapsedTime(comparisonRiderId);
272                   long difference = elapsedTime - otherElapsedTime;
273                   if (difference > 0L && difference <= 1000000000L) {
274                       timeAdjusted = true;
275                       elapsedTime = otherElapsedTime;
276                   }
277             }
278         } while (timeAdjusted);
279
280         return LocalTime.ofNanoOfDay(elapsedTime);
281     }
282
283     /**
284     * Return the array of indices which sorts the riders by their
285      * elapsed time when accessed in the order of the stage's
286      * results.
287     *
288     * @return An integer array containing the indices which sort
289      * the riders by elapsed time.
290     *
291     */
292     private int[] getSortedElapsedTimeIndices() {
293       ArrayList<Long> results = new ArrayList<Long>();
294         ArrayList<Integer> sortedIndices = new ArrayList<Integer>();
295         int unsortedIndex = 0;
296         for (Integer riderId : this.results.keySet()) {
297             long elapsedTime = getRiderElapsedTime(riderId);
298             int index = 0;
299             for (index = 0; index < results.size(); index++) {
300                 if (results.get(index) > elapsedTime) {
301                     break;
302                 }
303             }
304             results.add(index, elapsedTime);
305             sortedIndices.add(index, unsortedIndex);
306             unsortedIndex++;
307         }
308
309         int[] sortedArr = new int[sortedIndices.size()];
310         for (int i = 0; i < sortedIndices.size(); i++) {
311             sortedArr[i] = sortedIndices.get(i).intValue();
312         }
313         return sortedArr;
314     }
315
316     /**
317     * Return the array of rider's IDs when sorted by their elapsed
318      * time in the stage.
319     *
320     * @return An integer array containing the rider's IDs sorted
321      * in ascending order by their elapsed time.
322     *
323     */
```

```
324     public int[] getRidersRanks() {
325         int[] order = getSortedElapsedTimeIndices();
326         int[] ranks = new int[this.results.size()];
327         int index = 0;
328         for (Integer riderId : this.results.keySet()) {
329             ranks[order[index]] = riderId;
330             index++;
331         }
332         return ranks;
333     }
334 
335     /**
336     * Return the array of rider's elapsed times when sorted by their elapsed
337      * time in the stage.
338     *
339     * @return An integer array containing the rider's elapsed times sorted
340      * in ascending order by their elapsed time.
341     *
342     */
343     public LocalTime[] getRankedAdjustedTimes() {
344         int[] order = getSortedElapsedTimeIndices();
345         LocalTime[] times = new LocalTime[this.results.size()];
346         int index = 0;
347         for (Integer riderId : this.results.keySet()) {
348             times[order[index]] = getRiderAdjustedElapsedTime(riderId);
349             index++;
350         }
351         return times;
352     }
353 
354     /**
355     * Return the rank of the rider's finish time in the segment.
356     *
357      * @param riderId The rider's ID.
358      * @param segment The segment to rank.
359     * @return An integer of the rank of the rider in the segment.
360     *
361     */
362     public int getRidersRankInSegment(int riderId, Segment segment) {
363         int resultIndex = this.segments.indexOf(segment) + 1;
364 
365         long result = this.results.get(riderId).get(resultIndex).toNanoOfDay();
366         int rank = 0;
367         for (ArrayList<LocalTime> resultTimes : results.values()) {
368             long comparison = resultTimes.get(resultIndex).toNanoOfDay();
369             if (comparison < result) {
370                 rank++;
371             }
372         }
373         return rank;
374     }
375 
376     /**
377     * Return the array of rider's points when sorted by their elapsed
378      * time in the stage.
```

```
379     *
380     * @return An integer array containing the rider's points sorted
381      * in ascending order by their elapsed time.
382     *
383     */
384     public int[] getRidersPoints() {
385         HashMap<StageType, int[]> finishPoints = new HashMap<StageType, int[]>();
386       finishPoints.put(StageType.FLAT, new int[] {50, 30, 20, 18, 16, 14, 12, 10, 8, 7, 6, 5, 4, 3, 2});
387       finishPoints.put(StageType.MEDIUM_MOUNTAIN, new int[] {30, 25, 22, 19, 17, 15, 13, 11, 9, 7, 6, 5, 4,
              3, 2});
388       finishPoints.put(StageType.HIGH_MOUNTAIN, new int[] {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2,
              1});
389       finishPoints.put(StageType.TT, new int[] {20, 17, 15, 13, 11, 10 , 9, 8, 7, 6, 5, 4, 3, 2, 1});
390       int[] sprintPoints = {20, 17, 15, 13, 11, 10 , 9, 8, 7, 6, 5, 4, 3, 2, 1};
391
392         int[] order = getSortedElapsedTimeIndices();
393         int[] points = new int[this.results.size()];
394         Segment[] segments = getSegments();
395         int i = 0;
396         for (Integer riderId : this.results.keySet()) {
397             points[order[i]] = (order[i] < 15) ? finishPoints.get(this.type)[order[i]] : 0;
398             for (Segment segment : segments) {
399             int rank = getRidersRankInSegment(riderId, segment);
400             if (segment.getSegmentType() == SegmentType.SPRINT) {
401                if (rank < 15) {
402                        points[order[i]] += sprintPoints[rank];
403                }
404             }
405          }
406             i++;
407         }
408
409         return points;
410     }
411
412     /**
413     * Return the rider's points in the stage.
414     *
415     * @return An integer of the rider's points.
416     *
417     */
418     public int getRiderPoints(int riderId) {
419         int[] ranks = getRidersRanks();
420         int i;
421         for (i = 0; i < ranks.length; i++) {
422             if (ranks[i] == riderId) {
423                 break;
424             }
425         }
426         return getRidersPoints()[i];
427     }
428
429     /**
430     * Return the array of rider's mountain points when sorted by their
431      * elapsed time in the stage.
```

```
432     *
433     * @return An integer array containing the rider's mountain points
434      * sorted in ascending order by their elapsed time.
435     *
436     */
437     public int[] getRidersMountainPoints() {
438         HashMap<SegmentType, int[]> mountainPoints = new HashMap<SegmentType, int[]>();
439       mountainPoints.put(SegmentType.C4, new int[] {1});
440       mountainPoints.put(SegmentType.C3, new int[] {2, 1});
441       mountainPoints.put(SegmentType.C2, new int[] {5, 3, 2, 1});
442       mountainPoints.put(SegmentType.C1, new int[] {10, 8, 6, 4, 2, 1});
443       mountainPoints.put(SegmentType.HC, new int[] {20, 15, 12, 10, 8, 6, 4, 2});
444 
445         int[] order = getSortedElapsedTimeIndices();
446         int[] points = new int[this.results.size()];
447         Segment[] segments = getSegments();
448         int i = 0;
449         for (Integer riderId : this.results.keySet()) {
450             points[order[i]] = 0;
451             for (Segment segment : segments) {
452             int rank = getRidersRankInSegment(riderId, segment);
453                 SegmentType segmentType = segment.getSegmentType();
454             if (segment.getSegmentType() != SegmentType.SPRINT) {
455                if (rank < mountainPoints.get(segmentType).length) {
456                         points[order[i]] += mountainPoints.get(segmentType)[rank];
457                }
458             }
459          }
460             i++;
461         }
462 
463         return points;
464     }
465 
466     /**
467     * Return the rider's mountain points in the stage.
468     *
469     * @return An integer of the rider's mountain points.
470     *
471     */
472     public int getRiderMountainPoints(int riderId) {
473         int[] ranks = getRidersRanks();
474         int i;
475         for (i = 0; i < ranks.length; i++) {
476             if (ranks[i] == riderId) {
477                 break;
478             }
479         }
480         return getRidersMountainPoints()[i];
481     }
482 }
```

6 Segment.java

```java
package cycling;

public class Segment {

    private int stageId;
    private int segmentId;
    private double location;
    private SegmentType type;

    public Segment(int stageId, int segmentId, double location, SegmentType type) {
        this.stageId = stageId;
        this.segmentId = segmentId;
        this.location = location;
        this.type = type;
    }

    /**
    * Get the segment's stage ID.
    *
    * @return The segment's stage ID.
    *
    */
    public int getStageId() {
        return this.stageId;
    }

    /**
    * Get the segment's ID.
    *
    * @return The segment's ID.
    *
    */
    public int getId() {
        return this.segmentId;
    }

    /**
    * Get the segment's location.
    *
    * @return The segment's location.
    *
    */
    public double getLocation() {
        return this.location;
    }

    /**
    * Get the segment's type.
    *
    * @return The SegmentType of the segment.
    *
    */
    public SegmentType getSegmentType() {
        return this.type;
    }
```

```
}
```

7 CategorizedClimb.java

```
package cycling;

public class CategorizedClimb extends Segment {

    private double length;
    private double averageGradient;

    public CategorizedClimb(int stageId, int segmentId, double length, double location, double
         averageGradient, SegmentType type) {
        super(stageId, segmentId, location, type);
        this.length = length;
        this.averageGradient = averageGradient;
    }

    /**
    * Get the length of the climb.
    *
    * @return The length of the climb.
    *
    */
    public double getLength() {
        return this.length;
    }
}
```

8 IntermediateSprint.java

```
package cycling;

public class IntermediateSprint extends Segment {
    public IntermediateSprint(int stageId, int segmentId, double location, SegmentType type) {
        super(stageId, segmentId, location, type);
    }
}
```