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  /**
7   * Class to represent teams in the cycling portal.
8   *
9   * @author Charlie Goldstraw, Charlie MacDonald-Smith
10  * @version 1.0
11  *
12  */
13
14 public class Team implements Serializable {
15     private int teamId;
16     private String name;
17     private String description;
18     private ArrayList<Rider> riders = new ArrayList<Rider>();
19
20     public Team(int teamId, String name, String description) {
21         this.teamId = teamId;
22         this.name = name;
23         this.description = description;
24     }
25
26     /**
27     * Add a Rider to the team.
28     *
29     * @param rider The Rider object to add.
30     *
31     */
```

```
32      public void addRider(Rider rider) {
33          this.riders.add(rider);
34      }
35
36      /**
37      * Remove a Rider from the team.
38      *
39      * @param rider The Rider object to remove.
40      *
41      */
42      public void removeRider(Rider rider) {
43          this.riders.remove(rider);
44      }
45
46      /**
47      * Returns an array of riders in the team.
48       *
49      * @return The Rider array containing the team riders.
50      *
51      */
52      public Rider[] getRiders() {
53          return this.riders.toArray(new Rider[0]);
54      }
55
56      /**
57      * Get an array of the teams rider IDs.
58      *
59      * @return The IDs of the riders in an array.
60      *
61      */
62      public int[] getRiderIds() {
63          int[] riderIds = new int[this.riders.size()];
64          for (int i = 0; i < riderIds.length; i++) {
65              riderIds[i] = this.riders.get(i).getId();
66          }
67          return riderIds;
68      }
69
70      /**
71      * Get the name of the team.
72      *
73      * @return The String containing the name of the team.
74      *
75      */
76      public String getName() {
77          return this.name;
78      }
79
80      /**
81      * Get the ID of the team.
82      *
83      * @return The int of the team's ID.
84      *
85      */
86      public int getId() {
```

```
87          return this.teamId;
88      }
89  }
```

3 Rider.java

```
1   package cycling;
2
3   import java.io.Serializable;
4
5   /**
6    * Class to represent riders in teams.
7    *
8    * @author Charlie Goldstraw, Charlie MacDonald-Smith
9    * @version 1.0
10   *
11   */
12
13  public class Rider implements Serializable {
14      private int riderId;
15      private int teamId;
16      private String name;
17      private int yearOfBirth;
18
19      public Rider(int riderId, int teamId, String name, int yearOfBirth) {
20          this.riderId = riderId;
21          this.teamId = teamId;
22          this.name = name;
23          this.yearOfBirth = yearOfBirth;
24      }
25
26      /**
27      * Get the rider's team ID.
28      *
29      * @return The rider's team ID.
30      *
31      */
32      public int getTeamId() {
33          return this.teamId;
34      }
35
36      /**
37      * Get the rider's ID.
38      *
39      * @return The rider's ID.
40      *
41      */
42      public int getId() {
43          return this.riderId;
44      }
45
46      /**
47      * Get the rider's name.
48      *
```

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

4 Race.java

```
package cycling;

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

/**
 * Class to represent races in the cycling portal.
 *
 * @author Charlie Goldstraw, Charlie MacDonald-Smith
 * @version 1.0
 *
 */

public class Race implements Serializable {
    private int raceId;
    private String name;
    private String description;
    private ArrayList<Stage> stages = new ArrayList<Stage>();

    public Race(int raceId, String name, String description) {
        this.raceId = raceId;
        this.description = description;
        this.name = name;
    }

    /**
    * Get the name of the race.
    *
    * @return The name of the race.
    *
    */
    public String getName() {
        return this.name;
    }

    /**
    * Get the ID of the race.
    *
    * @return The ID of the race.
    *
    */
    public int getId() {
        return this.raceId;
```

```
45      }
46
47      /**
48      * Add a stage to the race.
49      *
50      * @param id Stage object to add.
51      *
52      */
53      public void addStage(Stage stage) {
54          this.stages.add(stage);
55      }
56
57      /**
58      * Removes a stage from the race.
59      *
60      * @param id Stage object to remove.
61      *
62      */
63      public void removeStage(Stage stage) {
64          this.stages.remove(stage);
65      }
66
67      /**
68      * Returns a list of stages in the race
69      *
70      * @return A list containing the race's stages.
71      *
72      */
73      public Stage[] getStages() {
74          return this.stages.toArray(new Stage[0]);
75      }
76
77      /**
78      * Returns a list of stage IDs in the race
79      *
80      * @return A list containing the race's stages' IDs.
81      *
82      */
83      public int[] getStageIds() {
84          int[] stageIds = new int[this.stages.size()];
85          for (int i = 0; i < stageIds.length; i++) {
86              stageIds[i] = this.stages.get(i).getId();
87          }
88          return stageIds;
89      }
90
91      /**
92      * Returns the description of the race
93      *
94      * @return A String of the description of the race.
95      *
96      */
97      public String getDescription() {
98          return this.description;
99      }
```

```
100
101     /**
102     * Return the array of indices which sorts the riders by their
103      * elapsed time when accessed in the order of the first stage's
104      * results.
105     *
106     * @return An integer array containing the indices which sort
107      * the riders by elapsed time.
108     *
109     */
110     private int[] getSortedElapsedTimeIndices() {
111       ArrayList<Long> results = new ArrayList<Long>();
112         ArrayList<Integer> sortedIndices = new ArrayList<Integer>();
113         int unsortedIndex = 0;
114         for (Integer riderId : this.stages.get(0).getRidersRanks()) {
115             long elapsedTime = 0;
116             for (Stage stage : this.stages) {
117                 elapsedTime += stage.getRiderAdjustedElapsedTime(riderId).toNanoOfDay();
118             }
119             int index = 0;
120             for (index = 0; index < results.size(); index++) {
121                 if (results.get(index) > elapsedTime) {
122                     break;
123                 }
124             }
125             results.add(index, elapsedTime);
126             sortedIndices.add(index, unsortedIndex);
127             unsortedIndex++;
128         }
129
130         int[] sortedArr = new int[sortedIndices.size()];
131         for (int i = 0; i < sortedIndices.size(); i++) {
132             sortedArr[i] = sortedIndices.get(i).intValue();
133         }
134         return sortedArr;
135     }
136
137     /**
138     * Return the array of general classification times for riders
139      * sorted by the riders' elapsed times.
140     *
141     * @return An LocalTime array containing the GC times of the riders.
142     *
143     */
144     public LocalTime[] getGeneralClassificationTimes() {
145         int[] order = getSortedElapsedTimeIndices();
146         LocalTime[] times = new LocalTime[order.length];
147         int index = 0;
148         for (Integer riderId : this.stages.get(0).getRidersRanks()) {
149             long elapsedTime = 0;
150             for (Stage stage : this.stages) {
151                 elapsedTime += stage.getRiderAdjustedElapsedTime(riderId).toNanoOfDay();
152             }
153             times[order[index]] = LocalTime.ofNanoOfDay(elapsedTime);
154             index++;
```

```
155         }
156         return times;
157     }
158 
159     /**
160     * Return the array of points for riders sorted by the riders'
161      * elapsed times.
162     *
163     * @return A LocalTime array containing the points of the riders.
164     *
165     */
166     public int[] getRidersPoints() {
167         int[] order = getSortedElapsedTimeIndices();
168         int[] points = new int[order.length];
169         int index = 0;
170         for (Integer riderId : this.stages.get(0).getRidersRanks()) {
171             int riderPoints = 0;
172             for (Stage stage : this.stages) {
173                 riderPoints += stage.getRiderPoints(riderId);
174             }
175             points[order[index]] = riderPoints;
176             index++;
177         }
178         return points;
179     }
180 
181     /**
182     * Return the array of mountain points for riders
183      * sorted by the riders' elapsed times.
184     *
185     * @return An int array containing the mountain points of the riders.
186     *
187     */
188     public int[] getRidersMountainPoints() {
189         int[] order = getSortedElapsedTimeIndices();
190         int[] points = new int[order.length];
191         int index = 0;
192         for (Integer riderId : this.stages.get(0).getRidersRanks()) {
193             int riderPoints = 0;
194             for (Stage stage : this.stages) {
195                 riderPoints += stage.getRiderMountainPoints(riderId);
196             }
197             points[order[index]] = riderPoints;
198             index++;
199         }
200         return points;
201     }
202 
203     /**
204     * Return the array of rider IDs sorted by the riders' elapsed times.
205     *
206     * @return An int array containing the sorted rider IDs.
207     *
208     */
209     public int[] getRidersGeneralClassificationRank() {
```

```
210         int[] order = getSortedElapsedTimeIndices();
211         int[] ranks = new int[order.length];
212         int index = 0;
213         for (Integer riderId : this.stages.get(0).getRidersRanks()) {
214             ranks[order[index]] = riderId;
215             index++;
216         }
217         return ranks;
218     }
219 
220     /**
221     * Return the array of rider IDs sorted in descending order
222      * by the riders' points.
223     *
224     * @return An int array containing the sorted rider IDs.
225     *
226     */
227     public int[] getRidersPointClassificationRank() {
228       ArrayList<Integer> points = new ArrayList<Integer>();
229         ArrayList<Integer> sortedIds = new ArrayList<Integer>();
230         for (Integer riderId : this.stages.get(0).getRidersRanks()) {
231             // Calculate points
232             int riderPoints = 0;
233             for (Stage stage : this.stages) {
234                 riderPoints += stage.getRiderPoints(riderId);
235             }
236             // Find sorted (descending) position in arraylist
237             int index = 0;
238             for (index = 0; index < points.size(); index++) {
239                 if (points.get(index) < riderPoints) {
240                     break;
241                 }
242             }
243             // Insert into arraylist
244             points.add(index, riderPoints);
245             sortedIds.add(index, riderId);
246         }
247 
248         int[] sortedArr = new int[sortedIds.size()];
249         for (int i = 0; i < sortedIds.size(); i++) {
250             sortedArr[i] = sortedIds.get(i).intValue();
251         }
252         return sortedArr;
253     }
254 
255     /**
256     * Return the array of rider IDs sorted in descending order
257      * by the riders' mountain points.
258     *
259     * @return An int array containing the sorted rider IDs.
260     *
261     */
262     public int[] getRidersMountainPointClassificationRank() {
263         ArrayList<Integer> points = new ArrayList<Integer>();
264         ArrayList<Integer> sortedIds = new ArrayList<Integer>();
```

```
        for (Integer riderId : this.stages.get(0).getRidersRanks()) {
            // Calculate points
            int riderPoints = 0;
            for (Stage stage : this.stages) {
                riderPoints += stage.getRiderMountainPoints(riderId);
            }
            // Find sorted (descending) position in arraylist
            int index = 0;
            for (index = 0; index < points.size(); index++) {
                if (points.get(index) < riderPoints) {
                    break;
                }
            }
            // Insert into arraylist
            points.add(index, riderPoints);
            sortedIds.add(index, riderId);
        }

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

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;

/**
 * Class to represent stages in races.
 *
 * @author Charlie Goldstraw, Charlie MacDonald-Smith
 * @version 1.0
 *
 */

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;
```

```
27      private ArrayList<Segment> segments = new ArrayList<Segment>();
28      private LinkedHashMap<Integer, ArrayList<LocalTime>> results = new LinkedHashMap<Integer,
            ArrayList<LocalTime>>();
29
30      public Stage(int raceId, int stageId, String name, String description, double length, LocalDateTime
            startTime, StageType type) {
31          this.raceId = raceId;
32          this.stageId = stageId;
33          this.description = description;
34          this.name = name;
35          this.length = length;
36          this.startTime = startTime;
37          this.type = type;
38          this.state = "preparation";
39      }
40
41      /**
42      * Get the stage's ID.
43      *
44      * @return The stage's ID.
45      *
46      */
47      public int getId() {
48          return this.stageId;
49      }
50
51      /**
52      * Get the stage's name.
53      *
54      * @return The stage's name.
55      *
56      */
57      public String getName() {
58          return this.name;
59      }
60
61      /**
62      * Get the race's ID.
63      *
64      * @return The stage's race ID.
65      *
66      */
67      public int getRaceId() {
68          return this.raceId;
69      }
70
71      /**
72      * Get the stage's length.
73      *
74      * @return The stage's length.
75      *
76      */
77      public double getLength() {
78          return this.length;
79      }
```

```
80
81      /**
82      * Get the stage's type.
83      *
84      * @return The stage's StageType.
85      *
86      */
87      public StageType getStageType() {
88          return this.type;
89      }
90
91      /**
92      * Get the stage's state.
93      *
94      * @return The stage's state.
95      *
96      */
97      public String getState() {
98          return this.state;
99      }
100
101     /**
102     * Add a segment to the stage.
103     *
104      * @param segment The Segment object to add.
105     *
106     */
107     public void addSegment(Segment segment) {
108         // Ensures that the segments are stored in chronological order
109         int sortedIndex = 0;
110         for (Segment comparison : this.segments) {
111             if (comparison.getLocation() > segment.getLocation()) {
112                 break;
113             }
114             sortedIndex++;
115         }
116
117         this.segments.add(sortedIndex, segment);
118     }
119
120     /**
121     * Remove a segment from the stage.
122     *
123      * @param segment The Segment object to remove.
124     *
125     */
126     public void removeSegment(Segment segment) {
127         this.segments.remove(segment);
128     }
129
130     /**
131     * Get the array of segments in the stage.
132     *
133      * @return The array of Segments in the stage.
134     *
```

```
135     */
136     public Segment[] getSegments() {
137         return this.segments.toArray(new Segment[0]);
138     }
139 
140     /**
141     * Get the array of segment IDs in the stage.
142     *
143      * @return The array of segment IDs in the stage.
144     *
145     */
146     public int[] getSegmentIds() {
147         int[] segmentIds = new int[this.segments.size()];
148         for (int i = 0; i < segmentIds.length; i++) {
149             segmentIds[i] = this.segments.get(i).getId();
150         }
151         return segmentIds;
152     }
153 
154     /**
155     * Set the state of the stage.
156     *
157      * @param state The state to change to.
158     *
159     */
160     public void setState(String state) {
161         this.state = state;
162     }
163 
164    /**
165     * Assert if the stage is not waiting for results.
166     *
167     * @throws InvalidStageStateException If the stage is waiting for results.
168     *
169     */
170     public void assertNotWaitingForResults() throws InvalidStageStateException {
171         if (this.state.equals("waiting for results")) {
172          String errorMessage = "The stage was waiting for results.";
173          throw new InvalidStageStateException(errorMessage);
174       }
175     }
176 
177    /**
178     * Assert if the stage is waiting for results.
179     *
180     * @throws InvalidStageStateException If the stage is not waiting for results.
181     *
182     */
183     public void assertWaitingForResults() throws InvalidStageStateException {
184         // Ensure the stage is waiting for results, throw an
185         // InvalidStageStateException if it is.
186         if (!this.state.equals("waiting for results")) {
187          String errorMessage = "The stage was waiting for results.";
188          throw new InvalidStageStateException(errorMessage);
189       }
```

```
190     }
191
192    /**
193     * Add a rider's results to the stage.
194     *
195     * @param riderId Rider's ID.
196      * @param checkpoints The LocalTime array of checkpoints.
197     *
198     */
199     public void addResults(int riderId, LocalTime[] checkpoints) {
200         ArrayList<LocalTime> resultList = new ArrayList<LocalTime>();
201         for (LocalTime result : checkpoints) {
202             resultList.add(result);
203         }
204       this.results.put(riderId, resultList);
205     }
206
207    /**
208     * Delete a rider's results from the stage
209     *
210     * @param riderId Rider's ID.
211     *
212     */
213     public void deleteResults(int riderId) {
214       this.results.remove(riderId);
215     }
216
217     /**
218     * Get a rider's results.
219     *
220     * @param riderId Rider's ID.
221     * @return A LocalTime array of the rider's results.
222     *
223     */
224     public LocalTime[] getResults(int riderId) {
225         if (!this.results.containsKey(riderId)) {
226             return new LocalTime[0];
227         }
228         ArrayList<LocalTime> riderResults = this.results.get(riderId);
229         LocalTime[] returnResults = new LocalTime[riderResults.size()-1];
230         for (int i = 1; i < riderResults.size()-1; i++) {
231             returnResults[i-1] = riderResults.get(i);
232         }
233         LocalTime elapsed = LocalTime.ofNanoOfDay(getRiderElapsedTime(riderId));
234         returnResults[riderResults.size()-2] = elapsed;
235         return returnResults;
236     }
237
238     /**
239     * Get a Rider's elapsed time in the stage.
240     *
241     * @param riderId Rider's ID.
242     * @return The rider's elapsed time in nanoseconds.
243     *
244     */
```

```
245     public long getRiderElapsedTime(int riderId) {
246         assert (this.results.containsKey(riderId));
247         LocalTime startTime = this.results.get(riderId).get(0);
248         int endIndex = this.segments.size() + 1;
249         LocalTime endTime = this.results.get(riderId).get(endIndex);
250         long elapsedTime = endTime.toNanoOfDay() - startTime.toNanoOfDay();
251         if (elapsedTime < 0) {
252             elapsedTime += 24L*60L*60L*1000000000L;
253         }
254         return elapsedTime;
255     }
256 
257     /**
258     * Get a rider's adjusted elapsed time. If the rider finished within 1 second
259      * of another rider, then both rider's have the elapsed time of the quicker
260      * result.
261     *
262     * @param riderId Rider's ID.
263     * @return The Rider's adjusted elapsed time in nanoseconds.
264     *
265     */
266     public LocalTime getRiderAdjustedElapsedTime(int riderId) {
267         if (!this.results.containsKey(riderId)) {
268             return null;
269         }
270 
271         long elapsedTime = getRiderElapsedTime(riderId);
272         if (this.type == StageType.TT) {
273          return LocalTime.ofNanoOfDay(elapsedTime);
274       }
275         boolean timeAdjusted = false;
276       do {
277          timeAdjusted = false;
278          for (Integer comparisonRiderId : this.results.keySet()) {
279                 long otherElapsedTime = getRiderElapsedTime(comparisonRiderId);
280                 long difference = elapsedTime - otherElapsedTime;
281                 if (difference > 0L && difference <= 1000000000L) {
282                     timeAdjusted = true;
283                     elapsedTime = otherElapsedTime;
284                 }
285          }
286       } while (timeAdjusted);
287 
288       return LocalTime.ofNanoOfDay(elapsedTime);
289     }
290 
291     /**
292     * Return the array of indices which sorts the riders by their
293      * elapsed time when accessed in the order of the stage's
294      * results.
295     *
296     * @return An integer array containing the indices which sort
297      * the riders by elapsed time.
298     *
299     */
```

```
300     private int[] getSortedElapsedTimeIndices() {
301       ArrayList<Long> results = new ArrayList<Long>();
302         ArrayList<Integer> sortedIndices = new ArrayList<Integer>();
303         int unsortedIndex = 0;
304         for (Integer riderId : this.results.keySet()) {
305             long elapsedTime = getRiderElapsedTime(riderId);
306             int index = 0;
307             for (index = 0; index < results.size(); index++) {
308                 if (results.get(index) > elapsedTime) {
309                     break;
310                 }
311             }
312             results.add(index, elapsedTime);
313             sortedIndices.add(index, unsortedIndex);
314             unsortedIndex++;
315         }
316 
317         int[] sortedArr = new int[sortedIndices.size()];
318         for (int i = 0; i < sortedIndices.size(); i++) {
319             sortedArr[i] = sortedIndices.get(i).intValue();
320         }
321         return sortedArr;
322     }
323 
324     /**
325     * Return the array of rider's IDs when sorted by their elapsed
326      * time in the stage.
327     *
328     * @return An integer array containing the rider's IDs sorted
329      * in ascending order by their elapsed time.
330     *
331     */
332     public int[] getRidersRanks() {
333         int[] order = getSortedElapsedTimeIndices();
334         int[] ranks = new int[this.results.size()];
335         int index = 0;
336         for (Integer riderId : this.results.keySet()) {
337             ranks[order[index]] = riderId;
338             index++;
339         }
340         return ranks;
341     }
342 
343     /**
344     * Return the array of rider's elapsed times when sorted by their elapsed
345      * time in the stage.
346     *
347     * @return An integer array containing the rider's elapsed times sorted
348      * in ascending order by their elapsed time.
349     *
350     */
351     public LocalTime[] getRankedAdjustedTimes() {
352         int[] order = getSortedElapsedTimeIndices();
353         LocalTime[] times = new LocalTime[this.results.size()];
354         int index = 0;
```

```
355         for (Integer riderId : this.results.keySet()) {
356             times[order[index]] = getRiderAdjustedElapsedTime(riderId);
357             index++;
358         }
359         return times;
360     }
361 
362     /**
363     * Return the rank of the rider's finish time in the segment.
364     *
365      * @param riderId The rider's ID.
366      * @param segment The segment to rank.
367     * @return An integer of the rank of the rider in the segment.
368     *
369     */
370     public int getRidersRankInSegment(int riderId, Segment segment) {
371         int resultIndex = this.segments.indexOf(segment) + 1;
372 
373         long result = this.results.get(riderId).get(resultIndex).toNanoOfDay();
374         int rank = 0;
375         for (ArrayList<LocalTime> resultTimes : results.values()) {
376             long comparison = resultTimes.get(resultIndex).toNanoOfDay();
377             if (comparison < result) {
378                 rank++;
379             }
380         }
381         return rank;
382     }
383 
384     /**
385     * Return the array of rider's points when sorted by their elapsed
386      * time in the stage.
387     *
388     * @return An integer array containing the rider's points sorted
389      * in ascending order by their elapsed time.
390     *
391     */
392     public int[] getRidersPoints() {
393         HashMap<StageType, int[]> finishPoints = new HashMap<StageType, int[]>();
394       finishPoints.put(StageType.FLAT, new int[] {50, 30, 20, 18, 16, 14, 12, 10, 8, 7, 6, 5, 4, 3, 2});
395       finishPoints.put(StageType.MEDIUM_MOUNTAIN, new int[] {30, 25, 22, 19, 17, 15, 13, 11, 9, 7, 6, 5, 4,
              3, 2});
396       finishPoints.put(StageType.HIGH_MOUNTAIN, new int[] {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2,
              1});
397       finishPoints.put(StageType.TT, new int[] {20, 17, 15, 13, 11, 10 , 9, 8, 7, 6, 5, 4, 3, 2, 1});
398       int[] sprintPoints = {20, 17, 15, 13, 11, 10 , 9, 8, 7, 6, 5, 4, 3, 2, 1};
399 
400         int[] order = getSortedElapsedTimeIndices();
401         int[] points = new int[this.results.size()];
402         Segment[] segments = getSegments();
403         int i = 0;
404         for (Integer riderId : this.results.keySet()) {
405             points[order[i]] = (order[i] < 15) ? finishPoints.get(this.type)[order[i]] : 0;
406             for (Segment segment : segments) {
407             int rank = getRidersRankInSegment(riderId, segment);
```

```
408             if (segment.getSegmentType() == SegmentType.SPRINT) {
409                if (rank < 15) {
410                        points[order[i]] += sprintPoints[rank];
411                }
412             }
413          }
414             i++;
415          }
416
417          return points;
418       }
419
420       /**
421       * Return the rider's points in the stage.
422       *
423       * @return An integer of the rider's points.
424       *
425       */
426       public int getRiderPoints(int riderId) {
427          int[] ranks = getRidersRanks();
428          int i;
429          for (i = 0; i < ranks.length; i++) {
430             if (ranks[i] == riderId) {
431                break;
432             }
433          }
434          return getRidersPoints()[i];
435       }
436
437       /**
438       * Return the array of rider's mountain points when sorted by their
439        * elapsed time in the stage.
440       *
441       * @return An integer array containing the rider's mountain points
442        * sorted in ascending order by their elapsed time.
443       *
444       */
445       public int[] getRidersMountainPoints() {
446          HashMap<SegmentType, int[]> mountainPoints = new HashMap<SegmentType, int[]>();
447         mountainPoints.put(SegmentType.C4, new int[] {1});
448         mountainPoints.put(SegmentType.C3, new int[] {2, 1});
449         mountainPoints.put(SegmentType.C2, new int[] {5, 3, 2, 1});
450         mountainPoints.put(SegmentType.C1, new int[] {10, 8, 6, 4, 2, 1});
451         mountainPoints.put(SegmentType.HC, new int[] {20, 15, 12, 10, 8, 6, 4, 2});
452
453          int[] order = getSortedElapsedTimeIndices();
454          int[] points = new int[this.results.size()];
455          Segment[] segments = getSegments();
456          int i = 0;
457          for (Integer riderId : this.results.keySet()) {
458             points[order[i]] = 0;
459             for (Segment segment : segments) {
460             int rank = getRidersRankInSegment(riderId, segment);
461                SegmentType segmentType = segment.getSegmentType();
462             if (segment.getSegmentType() != SegmentType.SPRINT) {
```

```
                if (rank < mountainPoints.get(segmentType).length) {
                        points[order[i]] += mountainPoints.get(segmentType)[rank];
                }
            }
         }
            i++;
        }

        return points;
    }

    /**
    * Return the rider's mountain points in the stage.
    *
    * @return An integer of the rider's mountain points.
    *
    */
    public int getRiderMountainPoints(int riderId) {
        int[] ranks = getRidersRanks();
        int i;
        for (i = 0; i < ranks.length; i++) {
            if (ranks[i] == riderId) {
                break;
            }
        }
        return getRidersMountainPoints()[i];
    }
}
```

6 Segment.java

```
package cycling;

import java.io.Serializable;

/**
 * Class to represent general segments, which is extended by
 * CategorizedClimb and IntermediateSprint.
 *
 * @author Charlie Goldstraw, Charlie MacDonald-Smith
 * @version 1.0
 *
 */

public class Segment implements Serializable {

    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;
```

```
24          this.location = location;
25          this.type = type;
26      }
27
28      /**
29      * Get the segment's stage ID.
30      *
31      * @return The segment's stage ID.
32      *
33      */
34      public int getStageId() {
35          return this.stageId;
36      }
37
38      /**
39      * Get the segment's ID.
40      *
41      * @return The segment's ID.
42      *
43      */
44      public int getId() {
45          return this.segmentId;
46      }
47
48      /**
49      * Get the segment's location.
50      *
51      * @return The segment's location.
52      *
53      */
54      public double getLocation() {
55          return this.location;
56      }
57
58      /**
59      * Get the segment's type.
60      *
61      * @return The SegmentType of the segment.
62      *
63      */
64      public SegmentType getSegmentType() {
65          return this.type;
66      }
67  }
```

7 CategorizedClimb.java

```
1   package cycling;
2
3   /**
4    * Class to represent categorized climb segments in stages.
5    *
6    * @author Charlie Goldstraw, Charlie MacDonald-Smith
7    * @version 1.0
```

```
 *
 */
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;

/**
 * Class to represent intermediate sprint segments in stages.
 *
 * @author Charlie Goldstraw, Charlie MacDonald-Smith
 * @version 1.0
 *
 */
public class IntermediateSprint extends Segment {
    public IntermediateSprint(int stageId, int segmentId, double location, SegmentType type) {
        super(stageId, segmentId, location, type);
    }
}
```