CyclingPortal Printout

123456789 & 987654321

Contents

1 CategorizedClimb.java

```java
package cycling;

public class CategorizedClimb extends Segment {
  private final Double averageGradient;
  private final Double length;

  public CategorizedClimb(
      Stage stage, Double location, SegmentType type, Double averageGradient, Double length)
      throws InvalidLocationException, InvalidStageStateException, InvalidStageTypeException {
    super(stage, type, location);
    this.averageGradient = averageGradient;
    this.length = length;
  }
}
```

2 CyclingPortal.java

```java
package cycling;

import java.io.*;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.ArrayList;
import java.util.List;

// TODO:
//    - Asserts !!!!
//    - Code Formatting
//    - Documentation/Comments
//    - Testing
//    - each function public/private/protected/default
//    - Optimise results?

public class CyclingPortal implements CyclingPortalInterface {
  // ArrayLists for all of a cycling portal instances teams, riders, races, stages and segments.
  // Although HashMaps could have been used here to get riders by int ID, it would be slower in the
  // long run as we would need to constantly convert it back to arrays to output results.
  private ArrayList<Team> teams = new ArrayList<>();
  private ArrayList<Rider> riders = new ArrayList<>();
  private ArrayList<Race> races = new ArrayList<>();
  private ArrayList<Stage> stages = new ArrayList<>();
  private ArrayList<Segment> segments = new ArrayList<>();

  // Record that will hold all the CyclingPortals teams, riders, races, stages & segments as well
  ↪  as
  // all the Id counts for each object.
  private record SavedCyclingPortal(
      ArrayList<Team> teams,
      ArrayList<Rider> riders,
      ArrayList<Race> races,
      ArrayList<Stage> stages,
      ArrayList<Segment> segments,
      int teamIdCount,
      int riderIdCount,
      int raceIdCount,
```

```
38        int stageIdCount,
39        int segmentIdCount) {}
40
41    /**
42     * Determine if a string contains any illegal whitespace characters.
43     *
44     * @param string The input string to be tested for whitespace.
45     * @return A boolean, true if the input string contains whitespace, false if not.
46     */
47    public static boolean containsWhitespace(String string) {
48      for (int i = 0; i < string.length(); ++i) {
49        if (Character.isWhitespace(string.charAt(i))) {
50          return true;
51        }
52      }
53      return false;
54    }
55
56    /**
57     * Get a Team object by a Team ID.
58     *
59     * @param ID The int ID of the Team to be looked up.
60     * @return The Team object of the team, if one is found.
61     * @throws IDNotRecognisedException Thrown if no team is found with the given Team ID.
62     */
63    public Team getTeamById(int ID) throws IDNotRecognisedException {
64      for (Team team : teams) {
65        if (team.getId() == ID) {
66          return team;
67        }
68      }
69      throw new IDNotRecognisedException("Team ID not found.");
70    }
71
72    /**
73     * Get a Rider object by a Rider ID.
74     *
75     * @param ID The int ID of the Rider to be looked up.
76     * @return The Rider object of the Rider, if one is found.
77     * @throws IDNotRecognisedException Thrown if no rider is found with the given Rider ID.
78     */
79    public Rider getRiderById(int ID) throws IDNotRecognisedException {
80      for (Rider rider : riders) {
81        if (rider.getId() == ID) {
82          return rider;
83        }
84      }
85      throw new IDNotRecognisedException("Rider ID not found.");
86    }
87
88    /**
89     * Get a Race object by a Race ID.
90     *
91     * @param ID The int ID of the Race to be looked up.
92     * @return The Race object of the race, if one is found.
93     * @throws IDNotRecognisedException Thrown if no race is found with the given Race ID.
94     */
95    public Race getRaceById(int ID) throws IDNotRecognisedException {
```

```
96      for (Race race : races) {
97        if (race.getId() == ID) {
98          return race;
99        }
100     }
101     throw new IDNotRecognisedException("Race ID not found.");
102   }
103
104   /**
105    * Get a Stage object by a Stage ID.
106    *
107    * @param ID The int ID of the Stage to be looked up.
108    * @return The Stage object of the stage, if one is found.
109    * @throws IDNotRecognisedException Thrown if no stage is found with the given Stage ID.
110    */
111   public Stage getStageById(int ID) throws IDNotRecognisedException {
112     for (Stage stage : stages) {
113       if (stage.getId() == ID) {
114         return stage;
115       }
116     }
117     throw new IDNotRecognisedException("Stage ID not found.");
118   }
119
120   /**
121    * Get a Segment object by a Segment ID.
122    *
123    * @param ID The int ID of the Segment to be looked up.
124    * @return The Segment object of the segment, if one is found.
125    * @throws IDNotRecognisedException Thrown if no segment is found with the given Segment ID.
126    */
127   public Segment getSegmentById(int ID) throws IDNotRecognisedException {
128     for (Segment segment : segments) {
129       if (segment.getId() == ID) {
130         return segment;
131       }
132     }
133     throw new IDNotRecognisedException("Segment ID not found.");
134   }
135
136   /**
137    * Loops over all races, stages and segments to remove all of a given riders results.
138    *
139    * @param rider The Rider object whose results will be removed from the Cycling Portal.
140    */
141   public void removeRiderResults(Rider rider) {
142     for (Race race : races) {
143       race.removeRiderResults(rider);
144     }
145     for (Stage stage : stages) {
146       stage.removeRiderResults(rider);
147     }
148     for (Segment segment : segments) {
149       segment.removeRiderResults(rider);
150     }
151   }
152
153   @Override
```

```
154   public int[] getRaceIds() {
155     int[] raceIDs = new int[races.size()];
156     for (int i = 0; i < races.size(); i++) {
157       Race race = races.get(i);
158       raceIDs[i] = race.getId();
159     }
160     return raceIDs;
161   }
162 
163   @Override
164   public int createRace(String name, String description)
165       throws IllegalNameException, InvalidNameException {
166     // Check a race with this name does not already exist in the system.
167     for (Race race : races) {
168       if (race.getName().equals(name)) {
169         throw new IllegalNameException("A Race with the name " + name + " already exists.");
170       }
171     }
172     Race race = new Race(name, description);
173     races.add(race);
174     return race.getId();
175   }
176 
177   @Override
178   public String viewRaceDetails(int raceId) throws IDNotRecognisedException {
179     Race race = getRaceById(raceId);
180     return race.getDetails();
181   }
182 
183   @Override
184   public void removeRaceById(int raceId) throws IDNotRecognisedException {
185     Race race = getRaceById(raceId);
186     // Remove all the races stages from the CyclingPortal.
187     for (final Stage stage : race.getStages()) {
188       stages.remove(stage);
189     }
190     races.remove(race);
191   }
192 
193   @Override
194   public int getNumberOfStages(int raceId) throws IDNotRecognisedException {
195     Race race = getRaceById(raceId);
196     return race.getStages().size();
197   }
198 
199   @Override
200   public int addStageToRace(
201       int raceId,
202       String stageName,
203       String description,
204       double length,
205       LocalDateTime startTime,
206       StageType type)
207       throws IDNotRecognisedException, IllegalNameException, InvalidNameException,
208           InvalidLengthException {
209     Race race = getRaceById(raceId);
210     // Check a stage with this name does not already exist in the system.
211     for (final Stage stage : stages) {
```

```
212       if (stage.getName().equals(stageName)) {
213         throw new IllegalNameException("A stage with the name " + stageName + " already exists.");
214       }
215     }
216     Stage stage = new Stage(race, stageName, description, length, startTime, type);
217     stages.add(stage);
218     race.addStage(stage);
219     return stage.getId();
220   }
221
222   @Override
223   public int[] getRaceStages(int raceId) throws IDNotRecognisedException {
224     Race race = getRaceById(raceId);
225     ArrayList<Stage> raceStages = race.getStages();
226     int[] raceStagesId = new int[raceStages.size()];
227     // Gathers the Stage ID's of the Stages in the Race.
228     for (int i = 0; i < raceStages.size(); i++) {
229       Stage stage = race.getStages().get(i);
230       raceStagesId[i] = stage.getId();
231     }
232     return raceStagesId;
233   }
234
235   @Override
236   public double getStageLength(int stageId) throws IDNotRecognisedException {
237     Stage stage = getStageById(stageId);
238     return stage.getLength();
239   }
240
241   @Override
242   public void removeStageById(int stageId) throws IDNotRecognisedException {
243     Stage stage = getStageById(stageId);
244     Race race = stage.getRace();
245     // Removes stage from both the Races and Stages.
246     race.removeStage(stage);
247     stages.remove(stage);
248   }
249
250   @Override
251   public int addCategorizedClimbToStage(
252       int stageId, Double location, SegmentType type, Double averageGradient, Double length)
253       throws IDNotRecognisedException, InvalidLocationException, InvalidStageStateException,
254           InvalidStageTypeException {
255     Stage stage = getStageById(stageId);
256     CategorizedClimb climb = new CategorizedClimb(stage, location, type, averageGradient, length);
257     // Adds Categorized Climb to both the list of Segments and the Stage.
258     segments.add(climb);
259     stage.addSegment(climb);
260     return climb.getId();
261   }
262
263   @Override
264   public int addIntermediateSprintToStage(int stageId, double location)
265       throws IDNotRecognisedException, InvalidLocationException, InvalidStageStateException,
266           InvalidStageTypeException {
267     Stage stage = getStageById(stageId);
268     IntermediateSprint sprint = new IntermediateSprint(stage, location);
269     // Adds Intermediate Sprint to both the list of Segments and the Stage.
```

```
270       segments.add(sprint);
271       stage.addSegment(sprint);
272       return sprint.getId();
273     }
274
275     @Override
276     public void removeSegment(int segmentId)
277         throws IDNotRecognisedException, InvalidStageStateException {
278       Segment segment = getSegmentById(segmentId);
279       Stage stage = segment.getStage();
280       // Removes Segment from both the Stage and list of Segments.
281       stage.removeSegment(segment);
282       segments.remove(segment);
283     }
284
285     @Override
286     public void concludeStagePreparation(int stageId)
287         throws IDNotRecognisedException, InvalidStageStateException {
288       Stage stage = getStageById(stageId);
289       stage.concludePreparation();
290     }
291
292     @Override
293     public int[] getStageSegments(int stageId) throws IDNotRecognisedException {
294       Stage stage = getStageById(stageId);
295       ArrayList<Segment> stageSegments = stage.getSegments();
296       int[] stageSegmentsId = new int[stageSegments.size()];
297       // Gathers Segment ID's from the Segments in the Stage.
298       for (int i = 0; i < stageSegments.size(); i++) {
299         Segment segment = stageSegments.get(i);
300         stageSegmentsId[i] = segment.getId();
301       }
302       return stageSegmentsId;
303     }
304
305     @Override
306     public int createTeam(String name, String description)
307         throws IllegalNameException, InvalidNameException {
308       // Checks if the Team name already exists on the system.
309       for (final Team team : teams) {
310         if (team.getName().equals(name)) {
311           throw new IllegalNameException("A Team with the name " + name + " already exists.");
312         }
313       }
314       Team team = new Team(name, description);
315       teams.add(team);
316       return team.getId();
317     }
318
319     @Override
320     public void removeTeam(int teamId) throws IDNotRecognisedException {
321       Team team = getTeamById(teamId);
322       // Loops through and removes Team Riders and Team Rider Results.
323       for (final Rider rider : team.getRiders()) {
324         removeRiderResults(rider);
325         riders.remove(rider);
326       }
327       teams.remove(team);
```

```
328   }
329
330   @Override
331   public int[] getTeams() {
332     int[] teamIDs = new int[teams.size()];
333     for (int i = 0; i < teams.size(); i++) {
334       Team team = teams.get(i);
335       teamIDs[i] = team.getId();
336     }
337     return teamIDs;
338   }
339
340   @Override
341   public int[] getTeamRiders(int teamId) throws IDNotRecognisedException {
342     Team team = getTeamById(teamId);
343     ArrayList<Rider> teamRiders = team.getRiders();
344     int[] teamRiderIds = new int[teamRiders.size()];
345     // Gathers ID's of Riders in the Team.
346     for (int i = 0; i < teamRiderIds.length; i++) {
347       teamRiderIds[i] = teamRiders.get(i).getId();
348     }
349     return teamRiderIds;
350   }
351
352   @Override
353   public int createRider(int teamID, String name, int yearOfBirth)
354       throws IDNotRecognisedException, IllegalArgumentException {
355     Team team = getTeamById(teamID);
356     Rider rider = new Rider(team, name, yearOfBirth);
357     // Adds Rider to both the Team and the list of Riders.
358     team.addRider(rider);
359     riders.add(rider);
360     return rider.getId();
361   }
362
363   @Override
364   public void removeRider(int riderId) throws IDNotRecognisedException {
365     Rider rider = getRiderById(riderId);
366     removeRiderResults(rider);
367     // Removes Rider from both the Team and the list of Riders.
368     rider.getTeam().removeRider(rider);
369     riders.remove(rider);
370   }
371
372   @Override
373   public void registerRiderResultsInStage(int stageId, int riderId, LocalTime... checkpoints)
374       throws IDNotRecognisedException, DuplicatedResultException, InvalidCheckpointsException,
375           InvalidStageStateException {
376     Stage stage = getStageById(stageId);
377     Rider rider = getRiderById(riderId);
378     stage.registerResult(rider, checkpoints);
379   }
380
381   @Override
382   public LocalTime[] getRiderResultsInStage(int stageId, int riderId)
383       throws IDNotRecognisedException {
384     Stage stage = getStageById(stageId);
385     Rider rider = getRiderById(riderId);
```

```
386       StageResult result = stage.getRiderResult(rider);
387
388       if (result == null) {
389         // Returns an empty array if the Result is null.
390         return new LocalTime[] {};
391       } else {
392         LocalTime[] checkpoints = result.getCheckpoints();
393         // Rider Results will always be 1 shorter than the checkpoint length because
394         // the finish time checkpoint will be replaced with the Elapsed Time and the start time
395         // checkpoint will be ignored.
396         LocalTime[] resultsInStage = new LocalTime[checkpoints.length - 1];
397         LocalTime elapsedTime = LocalTime.MIDNIGHT.plus(result.getElapsedTime());
398         for (int i = 0; i < resultsInStage.length; i++) {
399           if (i == resultsInStage.length - 1) {
400             // Adds the Elapsed Time to the end of the array of Results.
401             resultsInStage[i] = elapsedTime;
402           } else {
403             // Adds each checkpoint to the array of Results until all have been added, skipping the
404             // Start time checkpoint.
405             resultsInStage[i] = checkpoints[i + 1];
406           }
407         }
408         return resultsInStage;
409       }
410     }
411
412     @Override
413     public LocalTime getRiderAdjustedElapsedTimeInStage(int stageId, int riderId)
414         throws IDNotRecognisedException {
415       Stage stage = getStageById(stageId);
416       Rider rider = getRiderById(riderId);
417       StageResult result = stage.getRiderResult(rider);
418       if (result == null) {
419         return null;
420       } else {
421         return result.getAdjustedElapsedLocalTime();
422       }
423     }
424
425     @Override
426     public void deleteRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException {
427       Stage stage = getStageById(stageId);
428       Rider rider = getRiderById(riderId);
429       stage.removeRiderResults(rider);
430     }
431
432     @Override
433     public int[] getRidersRankInStage(int stageId) throws IDNotRecognisedException {
434       Stage stage = getStageById(stageId);
435       // Gets a list of Riders from the Stage ordered by their Elapsed Times.
436       List<Rider> riders = stage.getRidersByElapsedTime();
437       int[] riderIds = new int[riders.size()];
438       // Gathers ID's from the ordered list of Riders.
439       for (int i = 0; i < riders.size(); i++) {
440         riderIds[i] = riders.get(i).getId();
441       }
442       return riderIds;
443     }
```

```
444
445   @Override
446   public LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId)
447       throws IDNotRecognisedException {
448     Stage stage = getStageById(stageId);
449     // Gets a list of Riders from the Stage ordered by their Elapsed Times.
450     List<Rider> riders = stage.getRidersByElapsedTime();
451     LocalTime[] riderAETs = new LocalTime[riders.size()];
452     // Gathers Riders' Adjusted Elapsed Times ordered by their Elapsed Times.
453     for (int i = 0; i < riders.size(); i++) {
454       Rider rider = riders.get(i);
455       riderAETs[i] = stage.getRiderResult(rider).getAdjustedElapsedLocalTime();
456     }
457     return riderAETs;
458   }
459
460   @Override
461   public int[] getRidersPointsInStage(int stageId) throws IDNotRecognisedException {
462     Stage stage = getStageById(stageId);
463     // Gets a list of Riders from the Stage ordered by their Elapsed Times.
464     List<Rider> riders = stage.getRidersByElapsedTime();
465     int[] riderSprinterPoints = new int[riders.size()];
466     // Gathers Sprinters' Points ordered by their Elapsed Times.
467     for (int i = 0; i < riders.size(); i++) {
468       Rider rider = riders.get(i);
469       riderSprinterPoints[i] = stage.getRiderResult(rider).getSprintersPoints();
470     }
471     return riderSprinterPoints;
472   }
473
474   @Override
475   public int[] getRidersMountainPointsInStage(int stageId) throws IDNotRecognisedException {
476     Stage stage = getStageById(stageId);
477     // Gets a list of Riders from the Stage ordered by their Elapsed Times.
478     List<Rider> riders = stage.getRidersByElapsedTime();
479     int[] riderMountainPoints = new int[riders.size()];
480     // Gathers Riders' Mountain Points ordered by their Elapsed Times.
481     for (int i = 0; i < riders.size(); i++) {
482       Rider rider = riders.get(i);
483       riderMountainPoints[i] = stage.getRiderResult(rider).getMountainPoints();
484     }
485     return riderMountainPoints;
486   }
487
488   @Override
489   public void eraseCyclingPortal() {
490     // Replaces teams, riders, races, stages and segments with empty ArrayLists.
491     teams = new ArrayList<>();
492     riders = new ArrayList<>();
493     races = new ArrayList<>();
494     stages = new ArrayList<>();
495     segments = new ArrayList<>();
496     // Sets the ID counters of the Rider, Team, Race, Stage and Segment objects back
497     // to 0.
498     Rider.resetIdCounter();
499     Team.resetIdCounter();
500     Race.resetIdCounter();
501     Stage.resetIdCounter();
```

```
502     Segment.resetIdCounter();
503   }
504
505   @Override
506   public void saveCyclingPortal(String filename) throws IOException {
507     FileOutputStream file = new FileOutputStream(filename + ".ser");
508     ObjectOutputStream output = new ObjectOutputStream(file);
509     // Saves teams, riders, races, stages and segments ArrayLists.
510     // Saves ID counters of Team, Rider, Race, Stage and Segment objects.
511     SavedCyclingPortal savedCyclingPortal =
512         new SavedCyclingPortal(
513             teams,
514             riders,
515             races,
516             stages,
517             segments,
518             Team.getIdCounter(),
519             Rider.getIdCounter(),
520             Race.getIdCounter(),
521             Stage.getIdCounter(),
522             Segment.getIdCounter());
523     output.writeObject(savedCyclingPortal);
524     output.close();
525     file.close();
526   }
527
528   @Override
529   public void loadCyclingPortal(String filename) throws IOException, ClassNotFoundException {
530     eraseCyclingPortal();
531     FileInputStream file = new FileInputStream(filename + ".ser");
532     ObjectInputStream input = new ObjectInputStream(file);
533
534     SavedCyclingPortal savedCyclingPortal = (SavedCyclingPortal) input.readObject();
535     // Imports teams, riders, races, stages and segments ArrayLists from the last save.
536     teams = savedCyclingPortal.teams;
537     riders = savedCyclingPortal.riders;
538     races = savedCyclingPortal.races;
539     stages = savedCyclingPortal.stages;
540     segments = savedCyclingPortal.segments;
541
542     // Imports ID counters of Team, Rider, Race, Stage and Segment objects from the last save.
543     Team.setIdCounter(savedCyclingPortal.teamIdCount);
544     Rider.setIdCounter(savedCyclingPortal.riderIdCount);
545     Race.setIdCounter(savedCyclingPortal.raceIdCount);
546     Stage.setIdCounter(savedCyclingPortal.stageIdCount);
547     Segment.setIdCounter(savedCyclingPortal.segmentIdCount);
548
549     input.close();
550     file.close();
551   }
552
553   @Override
554   public void removeRaceByName(String name) throws NameNotRecognisedException {
555     for (final Race race : races) {
556       if (race.getName().equals(name)) {
557         races.remove(race);
558         return;
559       }
```

```
560       }
561       throw new NameNotRecognisedException("Race name is not in the system.");
562     }
563
564     @Override
565     public int[] getRidersGeneralClassificationRank(int raceId) throws IDNotRecognisedException {
566       Race race = getRaceById(raceId);
567       List<Rider> riders = race.getRidersByAdjustedElapsedTime();
568       int[] riderIds = new int[riders.size()];
569       // Gathers Rider ID's ordered by their Adjusted Elapsed Times.
570       for (int i = 0; i < riders.size(); i++) {
571         riderIds[i] = riders.get(i).getId();
572       }
573       return riderIds;
574     }
575
576     @Override
577     public LocalTime[] getGeneralClassificationTimesInRace(int raceId)
578         throws IDNotRecognisedException {
579       Race race = getRaceById(raceId);
580       // Gets a list of Riders from the Stage ordered by their Adjusted Elapsed Times.
581       List<Rider> riders = race.getRidersByAdjustedElapsedTime();
582       LocalTime[] riderTimes = new LocalTime[riders.size()];
583       // Gathers Riders' Cumulative Adjusted Elapsed LocalTimes ordered by their Adjusted Elapsed
584       // Times.
585       for (int i = 0; i < riders.size(); i++) {
586         riderTimes[i] = race.getRiderResults(riders.get(i)).getCumulativeAdjustedElapsedLocalTime();
587       }
588       return riderTimes;
589     }
590
591     @Override
592     public int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException {
593       Race race = getRaceById(raceId);
594       List<Rider> riders = race.getRidersByAdjustedElapsedTime();
595       int[] riderIds = new int[riders.size()];
596       // Gathers Riders' Cumulative Sprinters Points ordered by their Adjusted Elapsed Times.
597       for (int i = 0; i < riders.size(); i++) {
598         riderIds[i] = race.getRiderResults(riders.get(i)).getCumulativeSprintersPoints();
599       }
600       return riderIds;
601     }
602
603     @Override
604     public int[] getRidersMountainPointsInRace(int raceId) throws IDNotRecognisedException {
605       Race race = getRaceById(raceId);
606       List<Rider> riders = race.getRidersByAdjustedElapsedTime();
607       int[] riderIds = new int[riders.size()];
608       // Gathers Riders' Cumulative Mountain Points ordered by their Adjusted Elapsed Times.
609       for (int i = 0; i < riders.size(); i++) {
610         riderIds[i] = race.getRiderResults(riders.get(i)).getCumulativeMountainPoints();
611       }
612       return riderIds;
613     }
614
615     @Override
616     public int[] getRidersPointClassificationRank(int raceId) throws IDNotRecognisedException {
617       Race race = getRaceById(raceId);
```

```
618     List<Rider> riders = race.getRidersBySprintersPoints();
619     int[] riderIds = new int[riders.size()];
620     // Gathers Rider ID's ordered by their Sprinters Points.
621     for (int i = 0; i < riders.size(); i++) {
622       riderIds[i] = riders.get(i).getId();
623     }
624     return riderIds;
625   }
626
627   @Override
628   public int[] getRidersMountainPointClassificationRank(int raceId)
629       throws IDNotRecognisedException {
630     Race race = getRaceById(raceId);
631     List<Rider> riders = race.getRidersByMountainPoints();
632     int[] riderIds = new int[riders.size()];
633     // Gathers Rider ID's ordered by their Mountain Points.
634     for (int i = 0; i < riders.size(); i++) {
635       riderIds[i] = riders.get(i).getId();
636     }
637     return riderIds;
638   }
639 }
```

3 IntermediateSprint.java

```
1  package cycling;
2
3  public class IntermediateSprint extends Segment {
4    private final double location;
5
6    public IntermediateSprint(Stage stage, double location)
7        throws InvalidLocationException, InvalidStageTypeException, InvalidStageStateException {
8      super(stage, SegmentType.SPRINT, location);
9      this.location = location;
10   }
11 }
```

4 Race.java

```
1  package cycling;
2
3  import java.time.LocalDateTime;
4  import java.util.*;
5  import java.util.stream.Collectors;
6
7  /**
8   * Race Class. This represents a Race that holds a Race's Stages and Riders, and also contains
9   * methods that deal with these.
10  */
11 public class Race {
12
13   private final String name;
14   private final String description;
15
16   private final ArrayList<Stage> stages = new ArrayList<>();
17
```

```java
  private final HashMap<Rider, RaceResult> results = new HashMap<>();

  private static int count = 0;
  private final int id;

  /**
   * Constructor method that sets up Rider with a name and a description.
   *
   * @param name: Cannot be empty, null, have a length greater than 30 or contain any whitespace.
   * @param description: A description of the race.
   * @throws InvalidNameException Thrown if the Race name is does not meet name requirements stated
   *     above.
   */
  public Race(String name, String description) throws InvalidNameException {
    if (name == null
        || name.isEmpty()
        || name.length() > 30
        || CyclingPortal.containsWhitespace(name)) {
      throw new InvalidNameException(
          "The name cannot be null, empty, have more than 30 characters, or have white spaces.");
    }
    this.name = name;
    this.description = description;
    // ID counter represents the highest known ID at the current time to ensure there
    // are no ID collisions.
    this.id = Race.count++;
  }

  /** Method that resets the static ID counter of the Race. Used for erasing and loading. */
  static void resetIdCounter() {
    count = 0;
  }

  /**
   * Method to get the current state of the static ID counter.
   *
   * @return the highest race ID stored currently.
   */
  static int getIdCounter() {
    return count;
  }

  /**
   * Method that sets the static ID counter to an inputted value.
   *
   * @param newCount: new value of the static ID counter.
   */
  static void setIdCounter(int newCount) {
    count = newCount;
  }

  /**
   * Method to get the ID of the Race object.
   *
   * @return int id: the Race's unique ID value.
   */
  public int getId() {
    return id;
```

```
76    }
77
78    /**
79     * Method to get the name of the Race.
80     *
81     * @return String name: the given name of the Race.
82     */
83    public String getName() {
84      return name;
85    }
86
87    /**
88     * Method that adds a Stage to the Race object's ordered list of Stages. It is added to the
89     * correct position based on its start time.
90     *
91     * @param stage: The stage to be added to the Race.
92     */
93    public void addStage(Stage stage) {
94      for (int i = 0; i < stages.size(); i++) {
95        // Retrieves the start time of each Stage in the Race's current Stages one by one.
96        // These are already ordered by their start times.
97        LocalDateTime iStartTime = stages.get(i).getStartTime();
98        // Adds the new Stage to the list of stages in the correct position based on
99        // its start time.
100       if (stage.getStartTime().isBefore(iStartTime)) {
101         stages.add(i, stage);
102         return;
103       }
104     }
105     stages.add(stage);
106   }
107
108   /**
109    * Method to get the list of Stages in the Race ordered by their start times.
110    *
111    * @return Arraylist<Stages> stages: The ordered list of Stages.
112    */
113   public ArrayList<Stage> getStages() {
114     return stages;
115   }
116
117   /**
118    * Method that removes a given Stage from the list of Stages.
119    *
120    * @param stage: the Stage to be deleted.
121    */
122   public void removeStage(Stage stage) {
123     stages.remove(stage);
124   }
125
126   /**
127    * Method to get then details of a Race including Race ID, name, description number of stages and
128    * total length.
129    *
130    * @return String: concatenated paragraph of details.
131    */
132   public String getDetails() {
133     double currentLength = 0;
```

```
134     for (final Stage stage : stages) {
135       currentLength = currentLength + stage.getLength();
136     }
137     return ("Race ID: "
138         + id
139         + System.lineSeparator()
140         + "Name: "
141         + name
142         + System.lineSeparator()
143         + "Description: "
144         + description
145         + System.lineSeparator()
146         + "Number of Stages: "
147         + stages.size()
148         + System.lineSeparator()
149         + "Total length: "
150         + currentLength);
151   }
152
153   /**
154    * Method to get a list of Riders in the Race, sorted by their Adjusted Elapsed Time.
155    *
156    * @return List<Rider>: correctly sorted Riders.
157    */
158   public List<Rider> getRidersByAdjustedElapsedTime() {
159     calculateResults();
160     return sortRiderResultsBy(RaceResult.sortByAdjustedElapsedTime);
161   }
162
163   /**
164    * Method to get a list of Riders in the Race, sorted by their Sprinters Points.
165    *
166    * @return List<Rider>: correctly sorted Riders.
167    */
168   public List<Rider> getRidersBySprintersPoints() {
169     calculateResults();
170     return sortRiderResultsBy(RaceResult.sortBySprintersPoints);
171   }
172
173   /**
174    * Method to get a list of Riders in the Race, sorted by their Mountain Points.
175    *
176    * @return List<Rider>: correctly sorted Riders.
177    */
178   public List<Rider> getRidersByMountainPoints() {
179     calculateResults();
180     return sortRiderResultsBy(RaceResult.sortByMountainPoints);
181   }
182
183   /**
184    * Method to get the results of a given Rider.
185    *
186    * @param rider: Rider to get the results of.
187    * @return RaceResult: Result of the Rider.
188    */
189   public RaceResult getRiderResults(Rider rider) {
190     calculateResults();
191     return results.get(rider);
```

```
192   }
193
194   /**
195    * Method to remove the Results of a given Rider.
196    *
197    * @param rider: Rider whose Results will be removed.
198    */
199   public void removeRiderResults(Rider rider) {
200     results.remove(rider);
201   }
202
203   /**
204    * Method to get a list of Riders sorted by a given comparator of their Results.
205    *
206    * @param comparison: a comparator on the Riders' Results to sort the Riders by.
207    * @return List<Rider>: List of Riders sorted by the comparator on the Results.
208    */
209   private List<Rider> sortRiderResultsBy(Comparator<RaceResult> comparison) {
210     return results.entrySet().stream()
211         .sorted(Map.Entry.comparingByValue(comparison))
212         .map(Map.Entry::getKey)
213         .collect(Collectors.toList());
214   }
215
216   /**
217    * Method to register the Rider's Result to the Stage.
218    *
219    * @param rider: Rider whose Result needs to be registered.
220    * @param stageResult: Stage that the Result will be added to.
221    */
222   private void registerRiderResults(Rider rider, StageResult stageResult) {
223     if (results.containsKey(rider)) {
224       // When the hashmap of Results already contains the Results for the given Rider,
225       // results are not re-added.
226       results.get(rider).addStageResult(stageResult);
227     } else {
228       // If the hashmap of Results does not contain the Results for the given Rider,
229       // they then are added now.
230       RaceResult raceResult = new RaceResult();
231       raceResult.addStageResult(stageResult);
232       results.put(rider, raceResult);
233     }
234   }
235
236   /** Method that calculates the results for each Rider. */
237   private void calculateResults() {
238     for (Stage stage : stages) {
239       HashMap<Rider, StageResult> stageResults = stage.getStageResults();
240       for (Rider rider : stageResults.keySet()) {
241         registerRiderResults(rider, stageResults.get(rider));
242       }
243     }
244   }
245 }
```

5 RaceResult.java

```java
package cycling;

import java.time.Duration;
import java.time.LocalTime;
import java.util.Comparator;

public class RaceResult {
  private Duration cumulativeAdjustedElapsedTime = Duration.ZERO;
  private int cumulativeSprintersPoints = 0;
  private int cumulativeMountainPoints = 0;

  // TODO: Test ordered Asc
  protected static final Comparator<RaceResult> sortByAdjustedElapsedTime =
      Comparator.comparing(RaceResult::getCumulativeAdjustedElapsedTime);

  // TODO: Test order Desc
  protected static final Comparator<RaceResult> sortBySprintersPoints =
      Comparator.comparing(RaceResult::getCumulativeSprintersPoints).reversed();
  // protected static final Comparator<RaceResult> sortBySprintersPoints = (RaceResult result1,
  //    RaceResult result2) -> Integer.compare(result2.getCumulativeSprintersPoints(),
  //        result1.getCumulativeSprintersPoints());
  protected static final Comparator<RaceResult> sortByMountainPoints =
      Comparator.comparing(RaceResult::getCumulativeMountainPoints).reversed();
  // protected static final Comparator<RaceResult> sortByMountainPoints = (RaceResult result1,
  //    RaceResult result2) -> Integer.compare(result2.getCumulativeMountainPoints(),
  //        result1.getCumulativeMountainPoints());

  public Duration getCumulativeAdjustedElapsedTime() {
    return this.cumulativeAdjustedElapsedTime;
  }

  public LocalTime getCumulativeAdjustedElapsedLocalTime() {
    return LocalTime.MIDNIGHT.plus(this.cumulativeAdjustedElapsedTime);
  }

  public int getCumulativeMountainPoints() {
    return this.cumulativeMountainPoints;
  }

  public int getCumulativeSprintersPoints() {
    return this.cumulativeSprintersPoints;
  }

  public void addStageResult(StageResult stageResult) {
    this.cumulativeAdjustedElapsedTime =
        this.cumulativeAdjustedElapsedTime.plus(stageResult.getAdjustedElapsedTime());
    this.cumulativeSprintersPoints += stageResult.getSprintersPoints();
    this.cumulativeMountainPoints += stageResult.getMountainPoints();
  }
}
```

6 Rider.java

```java
package cycling;

```

```java
public class Rider {
  private final Team team;
  private final String name;
  private final int yearOfBirth;

  private static int count = 0;
  private final int id;

  public Rider(Team team, String name, int yearOfBirth) throws IllegalArgumentException {
    if (name == null) {
      throw new java.lang.IllegalArgumentException("The rider's name cannot be null.");
    }
    if (yearOfBirth < 1900) {
      throw new java.lang.IllegalArgumentException(
          "The rider's birth year is invalid, must be greater than 1900.");
    }

    this.team = team;
    this.name = name;
    this.yearOfBirth = yearOfBirth;
    this.id = Rider.count++;
  }

  static void resetIdCounter() {
    count = 0;
  }

  static int getIdCounter() {
    return count;
  }

  static void setIdCounter(int newCount) {
    count = newCount;
  }

  public int getId() {
    return id;
  }

  public Team getTeam() {
    return team;
  }
}
```

7 Segment.java

```java
package cycling;

import java.time.LocalTime;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class Segment {
  private static int count = 0;
  private final Stage stage;
```

```
12    private final int id;
13    private final SegmentType type;
14    private final double location;
15
16    private final HashMap<Rider, SegmentResult> results = new HashMap<>();
17
18    private static final int[] SPRINT_POINTS = {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
19    private static final int[] HC_POINTS = {20, 15, 12, 10, 8, 6, 4, 2};
20    private static final int[] C1_POINTS = {10, 8, 6, 4, 2, 1};
21    private static final int[] C2_POINTS = {5, 3, 2, 1};
22    private static final int[] C3_POINTS = {2, 1};
23    private static final int[] C4_POINTS = {1};
24
25    public Segment(Stage stage, SegmentType type, double location)
26        throws InvalidLocationException, InvalidStageStateException, InvalidStageTypeException {
27      if (location > stage.getLength()) {
28        throw new InvalidLocationException("The location is out of bounds of the stage length.");
29      }
30      if (stage.isWaitingForResults()) {
31        throw new InvalidStageStateException("The stage is waiting for results.");
32      }
33      if (stage.getType().equals(StageType.TT)) {
34        throw new InvalidStageTypeException("Time-trial stages cannot contain any segments.");
35      }
36      this.stage = stage;
37      this.id = Segment.count++;
38      this.type = type;
39      this.location = location;
40    }
41
42    static void resetIdCounter() {
43      count = 0;
44    }
45
46    static int getIdCounter() {
47      return count;
48    }
49
50    static void setIdCounter(int newCount) {
51      count = newCount;
52    }
53
54    public SegmentType getType() {
55      return type;
56    }
57
58    public int getId() {
59      return id;
60    }
61
62    public Stage getStage() {
63      return stage;
64    }
65
66    public double getLocation() {
67      return location;
68    }
69
```

```java
  public void registerResults(Rider rider, LocalTime finishTime) {
    SegmentResult result = new SegmentResult(finishTime);
    results.put(rider, result);
  }

  public SegmentResult getRiderResult(Rider rider) {
    calculateResults();
    return results.get(rider);
  }

  public void removeRiderResults(Rider rider) {
    results.remove(rider);
  }

  private List<Rider> sortRiderResults() {
    return results.entrySet().stream()
        .sorted(Map.Entry.comparingByValue(SegmentResult.sortByFinishTime))
        .map(Map.Entry::getKey)
        .collect(Collectors.toList());
  }

  private void calculateResults() {
    List<Rider> riders = sortRiderResults();

    for (int i = 0; i < results.size(); i++) {
      Rider rider = riders.get(i);
      SegmentResult result = results.get(rider);
      int position = i + 1;
      // Position Calculation
      result.setPosition(position);

      // Points Calculation
      int[] pointsDistribution = getPointsDistribution();
      if (position <= pointsDistribution.length) {
        int points = pointsDistribution[i];
        if (this.type.equals(SegmentType.SPRINT)) {
          result.setSprintersPoints(points);
          result.setMountainPoints(0);
        } else {
          result.setSprintersPoints(0);
          result.setMountainPoints(points);
        }
      } else {
        result.setMountainPoints(0);
        result.setSprintersPoints(0);
      }
    }
  }

  private int[] getPointsDistribution() {
    return switch (type) {
      case HC -> HC_POINTS;
      case C1 -> C1_POINTS;
      case C2 -> C2_POINTS;
      case C3 -> C3_POINTS;
      case C4 -> C4_POINTS;
      case SPRINT -> SPRINT_POINTS;
    };
```

```
128   }
129 }
```

8 SegmentResult.java

```
1  package cycling;
2
3  import java.time.LocalTime;
4  import java.util.Comparator;
5
6  public class SegmentResult {
7    private final LocalTime finishTime;
8    private int position;
9    private int sprintersPoints;
10   private int mountainPoints;
11
12   protected static final Comparator<SegmentResult> sortByFinishTime =
13       Comparator.comparing(SegmentResult::getFinishTime);
14
15   public SegmentResult(LocalTime finishTime) {
16     this.finishTime = finishTime;
17   }
18
19   public LocalTime getFinishTime() {
20     return finishTime;
21   }
22
23   public void setPosition(int position) {
24     this.position = position;
25   }
26
27   public int getPosition() {
28     return position;
29   }
30
31   public void setMountainPoints(int points) {
32     this.mountainPoints = points;
33   }
34
35   public void setSprintersPoints(int points) {
36     this.sprintersPoints = points;
37   }
38
39   public int getMountainPoints() {
40     return this.mountainPoints;
41   }
42
43   public int getSprintersPoints() {
44     return this.sprintersPoints;
45   }
46 }
```

9 Stage.java

```
1  package cycling;
2
```

```java
import java.time.Duration;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class Stage {
  private final Race race;
  private final String name;
  private final String description;
  private final double length;
  private final LocalDateTime startTime;
  private final StageType type;
  private final int id;
  private static int count = 0;
  private boolean waitingForResults = false;
  private final ArrayList<Segment> segments = new ArrayList<>();

  private final HashMap<Rider, StageResult> results = new HashMap<>();

  private static final int[] FLAT_POINTS = {50, 30, 20, 18, 16, 14, 12, 10, 8, 7, 6, 5, 4, 3, 2};
  private static final int[] MEDIUM_POINTS = {30, 25, 22, 19, 17, 15, 13, 11, 9, 7, 6, 5, 4, 3, 2};
  private static final int[] HIGH_POINTS = {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
  private static final int[] TT_POINTS = {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

  public Stage(
      Race race,
      String name,
      String description,
      double length,
      LocalDateTime startTime,
      StageType type)
      throws InvalidNameException, InvalidLengthException {
    if (name == null
        || name.isEmpty()
        || name.length() > 30
        || CyclingPortal.containsWhitespace(name)) {
      throw new InvalidNameException(
          "Stage name cannot be null, empty, have more than 30 characters or have white spaces.");
    }
    if (length < 5) {
      throw new InvalidLengthException("Length is invalid, cannot be less than 5km.");
    }
    this.name = name;
    this.description = description;
    this.race = race;
    this.length = length;
    this.startTime = startTime;
    this.type = type;
    this.id = Stage.count++;
  }

  static void resetIdCounter() {
    count = 0;
  }
```

```
61
62    static int getIdCounter() {
63      return count;
64    }
65
66    static void setIdCounter(int newCount) {
67      count = newCount;
68    }
69
70    public int getId() {
71      return id;
72    }
73
74    public String getName() {
75      return name;
76    }
77
78    public double getLength() {
79      return length;
80    }
81
82    public Race getRace() {
83      return race;
84    }
85
86    public StageType getType() {
87      return type;
88    }
89
90    public ArrayList<Segment> getSegments() {
91      return segments;
92    }
93
94    public LocalDateTime getStartTime() {
95      return startTime;
96    }
97
98    public void addSegment(Segment segment) {
99      for (int i = 0; i < segments.size(); i++) {
100       if (segment.getLocation() < segments.get(i).getLocation()) {
101         segments.add(i, segment);
102         return;
103       }
104     }
105     segments.add(segment);
106   }
107
108   public void removeSegment(Segment segment) throws InvalidStageStateException {
109     if (waitingForResults) {
110       throw new InvalidStageStateException(
111           "The stage cannot be removed as it is waiting for results.");
112     }
113     segments.remove(segment);
114   }
115
116   public void registerResult(Rider rider, LocalTime[] checkpoints)
117       throws InvalidStageStateException, DuplicatedResultException, InvalidCheckpointsException {
118     if (!waitingForResults) {
```

```
119         throw new InvalidStageStateException(
120             "Results can only be added to a stage while it is waiting for results.");
121       }
122       if (results.containsKey(rider)) {
123         throw new DuplicatedResultException("Each rider can only have one result per Stage.");
124       }
125       if (checkpoints.length != segments.size() + 2) {
126         throw new InvalidCheckpointsException(
127             "The length of the checkpoint must equal number of Segments in the Stage + 2.");
128       }
129
130       StageResult result = new StageResult(checkpoints);
131       // Save Riders result for the Stage
132       results.put(rider, result);
133
134       // Propagate all the Riders results for each segment
135       for (int i = 0; i < segments.size(); i++) {
136         segments.get(i).registerResults(rider, checkpoints[i + 1]);
137       }
138     }
139
140     public void concludePreparation() throws InvalidStageStateException {
141       if (waitingForResults) {
142         throw new InvalidStageStateException("Stage is already waiting for results.");
143       }
144       waitingForResults = true;
145     }
146
147     public boolean isWaitingForResults() {
148       return waitingForResults;
149     }
150
151     public StageResult getRiderResult(Rider rider) {
152       calculateResults();
153       return results.get(rider);
154     }
155
156     public void removeRiderResults(Rider rider) {
157       results.remove(rider);
158     }
159
160     public List<Rider> getRidersByElapsedTime() {
161       calculateResults();
162       return sortRiderResults();
163     }
164
165     public HashMap<Rider, StageResult> getStageResults() {
166       calculateResults();
167       return results;
168     }
169
170     private List<Rider> sortRiderResults() {
171       return results.entrySet().stream()
172           .sorted(Map.Entry.comparingByValue(StageResult.sortByElapsedTime))
173           .map(Map.Entry::getKey)
174           .collect(Collectors.toList());
175     }
176
```

```
177   private void calculateResults() {
178     List<Rider> riders = sortRiderResults();
179
180     for (int i = 0; i < results.size(); i++) {
181       Rider rider = riders.get(i);
182       StageResult result = results.get(rider);
183       int position = i + 1;
184
185       // Position Calculation
186       result.setPosition(position);
187
188       // Adjusted Elapsed Time Calculations
189       if (i == 0) {
190         result.setAdjustedElapsedTime(result.getElapsedTime());
191       } else {
192         Rider prevRider = riders.get(i - 1);
193         Duration prevTime = results.get(prevRider).getElapsedTime();
194         Duration time = results.get(rider).getElapsedTime();
195
196         int timeDiff = time.minus(prevTime).compareTo(Duration.ofSeconds(1));
197         if (timeDiff <= 0) {
198           // Close Finish Condition
199           Duration prevAdjustedTime = results.get(prevRider).getAdjustedElapsedTime();
200           result.setAdjustedElapsedTime(prevAdjustedTime);
201         } else {
202           // Far Finish Condition
203           result.setAdjustedElapsedTime(time);
204         }
205       }
206
207       // Points Calculation
208       int sprintersPoints = 0;
209       int mountainPoints = 0;
210       for (Segment segment : segments) {
211         SegmentResult segmentResult = segment.getRiderResult(rider);
212         sprintersPoints += segmentResult.getSprintersPoints();
213         mountainPoints += segmentResult.getMountainPoints();
214       }
215       int[] pointsDistribution = getPointDistribution();
216       if (position <= pointsDistribution.length) {
217         sprintersPoints += pointsDistribution[i];
218       }
219       result.setSprintersPoints(sprintersPoints);
220       result.setMountainPoints(mountainPoints);
221     }
222   }
223
224   private int[] getPointDistribution() {
225     return switch (type) {
226       case FLAT -> FLAT_POINTS;
227       case MEDIUM_MOUNTAIN -> MEDIUM_POINTS;
228       case HIGH_MOUNTAIN -> HIGH_POINTS;
229       case TT -> TT_POINTS;
230     };
231   }
232 }
```

10 StageResult.java

```java
package cycling;

import java.time.Duration;
import java.time.LocalTime;
import java.util.Comparator;

public class StageResult {
  private final LocalTime[] checkpoints;
  private final Duration elapsedTime;
  private Duration adjustedElapsedTime;
  private int position;
  private int sprintersPoints;
  private int mountainPoints;

  protected static final Comparator<StageResult> sortByElapsedTime =
      Comparator.comparing(StageResult::getElapsedTime);

  public StageResult(LocalTime[] checkpoints) {
    this.checkpoints = checkpoints;
    this.elapsedTime = Duration.between(checkpoints[0], checkpoints[checkpoints.length - 1]);
  }

  public LocalTime[] getCheckpoints() {
    return this.checkpoints;
  }

  public Duration getElapsedTime() {
    return elapsedTime;
  }

  public void setPosition(int position) {
    this.position = position;
  }

  public void setAdjustedElapsedTime(Duration adjustedElapsedTime) {
    this.adjustedElapsedTime = adjustedElapsedTime;
  }

  public int getPosition() {
    return position;
  }

  public Duration getAdjustedElapsedTime() {
    return adjustedElapsedTime;
  }

  public LocalTime getAdjustedElapsedLocalTime() {
    return checkpoints[0].plus(adjustedElapsedTime);
  }

  public void setMountainPoints(int points) {
    this.mountainPoints = points;
  }

  public void setSprintersPoints(int points) {
    this.sprintersPoints = points;
```

```
57    }
58
59    public int getMountainPoints() {
60      return mountainPoints;
61    }
62
63    public int getSprintersPoints() {
64      return sprintersPoints;
65    }
66
67    // --Commented out by Inspection START (28/03/2022, 3:31 pm):
68    //  public void add(StageResult res){
69    //    this.elapsedTime = this.elapsedTime.plus(res.getElapsedTime());
70    //    this.adjustedElapsedTime = this.adjustedElapsedTime.plus(res.getAdjustedElapsedTime());
71    //    this.sprintersPoints += res.getSprintersPoints();
72    //    this.mountainPoints += res.getMountainPoints();
73    //  }
74    // --Commented out by Inspection STOP (28/03/2022, 3:31 pm)
75  }
```

11 Team.java

```
1   package cycling;
2
3   import java.util.ArrayList;
4
5   public class Team {
6     private final String name;
7     private final String description;
8
9     private final ArrayList<Rider> riders = new ArrayList<>();
10    private static int count = 0;
11    private final int id;
12
13    public Team(String name, String description) throws InvalidNameException {
14      if (name == null
15          || name.isEmpty()
16          || name.length() > 30
17          || CyclingPortal.containsWhitespace(name)) {
18        throw new InvalidNameException(
19            "Team name cannot be null, empty, have more than 30 characters or have white spaces.");
20      }
21      this.name = name;
22      this.description = description;
23      this.id = Team.count++;
24    }
25
26    static void resetIdCounter() {
27      count = 0;
28    }
29
30    static int getIdCounter() {
31      return count;
32    }
33
34    static void setIdCounter(int newCount) {
35      count = newCount;
```

```
36    }
37
38    public String getName() {
39      return name;
40    }
41
42    public int getId() {
43      return id;
44    }
45
46    public void removeRider(Rider rider) {
47      riders.remove(rider);
48    }
49
50    public ArrayList<Rider> getRiders() {
51      return riders;
52    }
53
54    public void addRider(Rider rider) {
55      riders.add(rider);
56    }
57  }
```