

1	Development log			
2				
3	Date	Time	Duration (h)	700012146
		700011062		
4	14/02/2022	13:40	0.75	Scheduling
		Scheduling		
5	15/02/2022	19:00	1.5	git / 00 Design
		00 Design / Driver		
6	16/02/2022	13:45	1.75	Driver
		Observer		
7	17/02/2022	12:40	1.75	Observer
		Driver		
8	18/02/2022	13:35	1.85	Driver
		Observer		
9	19/02/2022	15:35	1.9	Observer
		Driver		
10	20/02/2022	16:20	1.9	Driver / 00
		Design Observer / 00 Design		
11	21/02/2022	18:35	1.969832	Observing
		Driver		
12	22/02/2022	18:40	1.843333	Driver
		Observer		
13	23/02/2022	15:35	1	Observing
		Driving		
14	24/02/2022	12:45	1.5	00 Design
		00 Design		
15	25/02/2022	12:35	0.333333333	Meeting with Prof
		Meeting with Prof		
16	25/02/2022	14:00	1.5	00 Design
		00 Design		
17	26/02/2022	16:00	1.916666667	Driver
		Observer		
18	28/02/2022	14:10	2.35	Observer
		Driver		
19	02/03/2022	13:30	2.00	Driver
		Observer		
20	04/03/2022	16:50	1.166666667	Observer
		Driver		
21	05/03/2022	13:30	2.00	Driver
		Observer		
22	06/03/2022	13:45	1.50	Observer
		Driver		
23	07/03/2022	13:30	1.500000001	Driver
		Observer		

24	10/03/2022	12:50	1.6	Observer
		Driver		
25	12/03/2022	16:35	3	Driver
		Observer		
26	13/03/2022	15:30	2	Observer
		Driver		
27	14/03/2022	16:30	2	Driver
		Observer		
28	15/03/2022	14:10	13/6	Observer
		Driver		
29	16/03/2022	14:40	3	Driver
		Observer		
30	17/03/2022	12:36	2.01	Observer
		Driver		
31	18/03/2022	13:39	11/6	Driver
		Observer		
32	19/03/2022	16:37	11/6	Observer
		Driver		
33	20/03/2022	13:35	3.5	Driver
		Observer		
34	21/03/2022	14:00	3	Observer
		Driver		
35	21/03/2022	19:00	3	Driver
		Observer		
36	22/03/2022	13:40	2.5	Documentation
		Documentation + jar		
37	23/03/2022	16:40	2.67	Polishing &
		assert Jar file		
38	24/03/2022	14:00	13/6	Driving /
		Observing Driving / Observing		
39	24/03/2022	17:00	4	Driving /
		Observing Driving / Observing		

```
1 package src.cycling;
2
3 import java.io.Serializable;
4
5 /**
6  * Represents a categorised climb segment in a race. This
7  * is a subclass of Segment (which
8  * represents an intermediate sprint) as they share common
9  * features.
10 *
11 */
12 public class CategorisedClimb extends Segment implements
13 Serializable {
14
15     private static int latestId = 0;
16     private final int id;
17     private int stageId;
18     private final double averageGradient;
19     private final double length;
20     private SegmentType segmentType;
21
22     /**
23      * Constructor.
24      *
25      * @param stageId The ID of the stage that this segment
26      * is part of.
27      * @param segmentType The SegmentType of this climb (i.e
28      * . HC, C1...3)
29      * @param averageGradient The average gradient of the
30      * climb segment
31      * @param length The length (distance from segment start
32      * point to end) of the climb segment.
33      * @param location The location in the stage at which
34      * the categorised climb starts.
35      */
36     public CategorisedClimb(int stageId, SegmentType
37         segmentType, Double averageGradient,
38         Double length, Double location) {
39         super(stageId, segmentType, location);
40         this.averageGradient = averageGradient;
41         this.length = length;
42         this.id = latestId++;
43     }
44 }
```

```
36    }
37
38    /**
39     * Reset the static ID counter to 0. Used after deletion
40     * of a CyclingPortal.
41     */
42     public static void resetIdCounter() {
43         latestId = 0;
44     }
45
46    /**
47     * @return The length (in distance) of the categorised
48     * climb.
49     */
50     public double getLength() {
51         return this.length;
52     }
53
54    /**
55     * @return The average gradient of the categorised climb
56     */
57     public double getAverageGradient() {
58         return this.averageGradient;
59     }
```

```
1 package src.cycling;
2
3 import java.io.Serializable;
4
5 /**
6  * Each cyclist is treated as an object that can be
7  * assigned to a team and compete in competitions.
8  * Their results are stored in separate objects that can
9  * be accessed via the rider's ID.
10 *
11 */
12 public class Rider implements Serializable {
13
14     private static int latestId = 0; // Enumerates to get
15     unique id, with 2^32 possible ids.
16     private String name;
17     private final int id;
18     private int teamId;
19     private int yearOfBirth;
20
21     /**
22      * Constructor.
23      *
24      * @param name          Rider's name
25      * @param teamId        The ID of the team the rider
26      * belongs to
27      * @param yearOfBirth   The rider's year of birth
28      */
29     public Rider(String name, int teamId, int yearOfBirth) {
30         this.name = name;
31         this.teamId = teamId;
32         this.yearOfBirth = yearOfBirth;
33         this.id = latestId++;
34     }
35
36     /**
37      * Reset the internal ID counter.
38      */
39     public static void resetIdCounter() {
40         latestId = 0;
41     }
42 }
```

```
41  /**
42   * @return The rider's name.
43   */
44  public String getName() {
45      return this.name;
46  }
47
48  /**
49   * @param newRiderName
50   */
51  public void setName(String newRiderName) {
52      this.name = newRiderName;
53  }
54
55  /**
56   * @return The rider's ID.
57   */
58  public int getId() {
59      return this.id;
60  }
61
62  /**
63   * @return The ID of the team the rider is in.
64   */
65  public int getTeamId() {
66      return this.teamId;
67  }
68
69  /**
70   * @param newTeamId
71   */
72  public void setTeamId(int newTeamId) {
73      this.teamId = newTeamId;
74  }
75
76  /**
77   * @return The rider's year of birth.
78   */
79  public int getYearOfBirth() {
80      return yearOfBirth;
81  }
82
83  /**
84   * @param newYearOfBirth
```

```
85      */
86  public void setYearOfBirth(int newYearOfBirth) {
87      this.yearOfBirth = newYearOfBirth;
88  }
89 }
90
```

```
1 package src.cycling;
2
3 import java.io.Serializable;
4 import java.time.LocalTime;
5
6 /**
7  * Represents the result of one rider in one race,
8  * including points, mountain points, finish time,
9  * and rank by finish time.
10 */
11 * @author Adam Kaizra, Sam Barker
12 * @version 1.0
13 */
14 public class RiderRaceResult implements Comparable<
15 RiderRaceResult>, Serializable {
16
17     private int riderId;
18     private final int raceId;
19     private int points = 0;
20     private int mountainPoints = 0;
21     private int rank;
22     private LocalTime elapsedTime;
23
24     /**
25      * Constructor.
26      *
27      * @param riderId The ID of the rider that achieved this
28      * result.
29      * @param raceId The ID of the race in which this result
30      * was achieved.
31     */
32     public RiderRaceResult(int riderId, int raceId) {
33         this.riderId = riderId;
34         this.raceId = raceId;
35         this.points = 0;
36         this.mountainPoints = 0;
37     }
38
39     /**
40      * Compare two RiderRaceResults by their finish times (
41      * for the purpose of sorting).
42  }
```

```
39     *
40     * param result The result to compare the calling
41     object's finish time to.
42     * return 1 if the calling object's finish time > the
43     parameter object's.
44     * 0 if equal. -1 if less than.
45     */
46     public int compareTo(RiderRaceResult result) {
47         assert (result instanceof RiderRaceResult) : "
48             Comparing incorrect types!";
49         return this.getElapsedTime().compareTo(result.
50             getElapsedTime());
51     }
52
53     /**
54      * Compare two RiderRaceResults by their points in the
55      race (for the purpose of sorting).
56      *
57      * param result The result to compare the calling
58      object's points to.
59      * return 1 if the calling object's points are less
60      than the parameter object's.
61      * 0 if equal. -1 if greater than.
62      */
63     public int compareByPoints(RiderRaceResult result) {
64         if (this.getPoints() < result.getPoints()) {
65             return 1;
66         } else if (this.getPoints() == result.getPoints()) {
67             return 0;
68         } else {
69             return -1;
70         }
71     }
72
73     /**
74      * Compare two RiderRaceResults by their mountain points
75      in the race (for the purpose of sorting).
76      *
77      * param result The result to compare the calling
78      object's mountain points to.
79      * return 1 if the calling object's mountain points are
80      less than the parameter object's.
81      * 0 if equal. -1 if greater than.
82      */
83 }
```

```
73  public int compareByMountainPoints(RiderRaceResult
    result) {
74      if (this.getMountainPoints() < result.
        getMountainPoints()) {
75          return 1;
76      } else if (this.getMountainPoints() == result.
        getMountainPoints()) {
77          return 0;
78      } else {
79          return -1;
80      }
81  }
82
83 /**
84 * return The rank (by finish time) of the rider
     within this race.
85 */
86 public int getRank() {
87     return rank;
88 }
89
90 /**
91 * Set the rank of the rider in the race (by finish
     time).
92 *
93 * param rank
94 */
95 public void setRank(int rank) {
96     this.rank = rank;
97 }
98
99 public LocalTime getElapsedTime() {
100    return elapsedTime;
101 }
102
103 public void setElapsedTime(LocalTime elapsedTime) {
104     this.elapsedTime = elapsedTime;
105 }
106
107 /**
108 * return The ID of the rider that achieved this
     result.
109 */
110 public int getRiderId() {
```

```
111     return riderId;
112 }
113
114 /**
115  * Set the ID of the rider that achieved this result.
116 *
117 * @param riderId
118 */
119 public void setRiderId(int riderId) {
120     this.riderId = riderId;
121 }
122
123 /**
124 * @return The ID of the race in which this result was
125 achieved.
126 */
127 public int getRaceId() {
128     return raceId;
129 }
130
131 /**
132 * @return The total finish time (sum of the finish
133 times of all stages)
134 * of the rider in the race.
135 */
136 public LocalTime getFinishTime() {
137     return finishTime;
138 }
139
140 /**
141 * Set the total finish time (sum of the finish times
142 of all stages)
143 * of the rider in the race.
144 */
145 public void setFinishTime(LocalTime time) {
146     this.finishTime = time;
147 }
148
149 /**
150 * @return The total points (points classification
151 points) the rider achieved in the race.
152 */
153
```

```
151     */
152     public int getPoints() {
153         return points;
154     }
155
156     /**
157      * Set the total points (points classification points)
158      * the rider achieved in the race.
159      *
160      * @param points
161      */
162     public void setPoints(int points) {
163         this.points = points;
164     }
165
166     /**
167      * @return The total mountain classification points the
168      * rider achieved in the race.
169      * Summed from across all stages.
170      */
171     public int getMountainPoints() {
172         return mountainPoints;
173     }
174
175     /**
176      * Set the total mountain classification points the
177      * rider achieved in the race.
178      *
179      * @param points
180      */
181     public void setMountainPoints(int points) {
182         this.mountainPoints = points;
183     }
184 }
```

```
1 package src.cycling;
2
3 import java.io.Serializable;
4 import java.io.Serializable;
5 import java.time.LocalTime;
6
7 /**
8  * Represents the result of one rider in one segment of a
9  * stage, including finish time,
10 * and rank. Points are not included for individual
11 * segments, but created for stages and races,
12 * and stored in RiderStageResult and RiderRaceResult
13 * respectively.
14 *
15 * @author Sam Barker, Adam Kaizra
16 * @version 1.0
17 */
18 public class RiderSegmentResult implements Comparable<
19     RiderSegmentResult>, Serializable {
20
21     private LocalTime time;
22     private int riderId;
23     private int rank;
24
25     /**
26      * @param time The finish time of the rider in the
27      * segment.
28      * @param riderId The ID of the rider who achieved this
29      * result.
30      */
31     public RiderSegmentResult(LocalTime time, int riderId) {
32         this.time = time;
33         this.riderId = riderId;
34     }
35
36     /**
37      * Compare two RiderSegmentResults by their finish time
38      * (for the purpose of sorting).
39      *
40      * @param result
41      * @return 1 if calling object's finish time is greater
42      * than the parameter object's.
43      * 0 if equal. -1 if less than.
44      */
45 }
```

```
37  public int compareTo(RiderSegmentResult result) {
38      assert (result instanceof RiderSegmentResult) : "Comparing incorrect types!";
39      return this.getTime().compareTo(result.getTime());
40  }
41
42  /**
43   * @return The rank of the rider in the segment (by
44   * finish time).
45   */
46  public int getRank() {
47      return rank;
48  }
49
50  /**
51   * Set the rank of the rider in the segment (by finish
52   * time).
53   *
54   * @param rank
55   */
56  public void setRank(int rank) {
57      this.rank = rank;
58  }
59
60  /**
61   * @return The finish time of the rider in the segment.
62   */
63  public LocalTime getTime() {
64      return this.time;
65  }
66
67  /**
68   * Set the finish time of the rider in the segment.
69   *
70   * @param time
71   */
72  public void setTime(LocalTime time) {
73      this.time = time;
74  }
75
76  /**
77   * @return The ID of the rider who achieved this result.
```

```
78     return riderId;
79 }
80
81 /**
82 * Set the ID of the rider who achieved this result.
83 *
84 * @param riderId
85 */
86 public void setRiderId(int riderId) {
87     this.riderId = riderId;
88 }
89 }
90
```

```
1 package src.cycling;
2
3 import java.io.Serializable;
4 import java.time.LocalTime;
5
6 /**
7  * Represents the result of a rider in a stage, holding
8  * their times in all segments of that stage,
9  * their final time, their adjusted elapsed time, their
10 * points classification points and their
11 * mountain classification points. Objects of this class
12 * are compared to each other by their finish
13 * times, which functions the same as comparing them by
14 * their adjusted times as adjusted times
15 * retain order.
16 *
17 * @author Adam Kaizra, Sam Barker
18 */
19 public class RiderStageResult implements Comparable<
20 RiderStageResult>, Serializable {
21
22     private final int riderId;
23     private final int stageId;
24     private final LocalTime[] times; // Times for the rider'
25     s start, segments and finish times.
26     private final LocalTime finishTime;
27     private LocalTime adjustedFinishTime; // Adjusted to
28     take the peloton leader's time if applicable.
29     private int points; // Points classification points for
30     this stage.
31     private int mountainPoints; // Mountain classification
32     points for this stage.
33     private LocalTime elapsedTime;
34     private LocalTime adjustedElapsedTime;
35
36     /**
37      * Constructor.
38      *
39      * @param riderId
40      * @param stageId
41      * @param times First input is starting time, final
42      * input is finishing time, all other inputs (if
43      * applicable) are segment times in
44      * chronological order.
```

```
34     */
35     public RiderStageResult(int riderId, int stageId,
36         LocalTime[] times) {
37         this.riderId = riderId;
38         this.stageId = stageId;
39         this.times = times; // Times the rider crossed each
40         // checkpoint.
41         this.finishTime = times[times.length - 1]; // Final
42         // time in the array is the finish time.
43         this.elapsedTime = SumLocalTimes.subtractLocalTimes(
44             this.finishTime, times[0]);
45         this.adjustedElapsedTime = this.elapsedTime;
46         this.adjustedFinishTime = this.finishTime; // Adjusted
47         // time initialised to finish time in case
48         // the rider is not in a peloton so their time is not
49         // changed.
50     }
51
52     /**
53      * Compares the finish times of two
54      * RiderStageResult objects.
55      *
56      * @param riderStageResult The stage result to be
57      * compared against.
58      *
59      * @return The comparator value, negative if less,
60      * positive if greater, zero if equal.
61      */
62     public int compareTo(RiderStageResult riderStageResult
63     ) {
64         assert (riderStageResult instanceof RiderStageResult
65         ) : "Comparing incorrect types!";
66         return this.getFinishTime().compareTo(riderStageResult
67             .getFinishTime());
68     }
69
70     /**
71      * @return The rider's points classification points for
72      * this stage.
73      */
74     public int getPoints() {
75         return points;
76     }
77
78     /**
79      * @return The rider's classification points for this
80      * stage.
```

```
65     * @param points The rider's points classification  
66     * points for this stage.  
67     */  
68     public void setPoints(int points) {  
69         this.points = points;  
70     }  
71     /**  
72     * Adds the given input to the rider's total points  
73     * classification points.  
74     *  
75     * @param points The rider's points classification  
76     * points for this stage.  
77     */  
78     public void addPoints(int points) {  
79         this.points += points;  
80     }  
81     /**  
82     * @return The rider's mountain points classification  
83     * points for this stage.  
84     */  
85     public int getMountainPoints() {  
86         return mountainPoints;  
87     }  
88     /**  
89     * @param mountainPoints The rider's mountain points  
90     * classification points for this stage.  
91     */  
92     public void setMountainPoints(int mountainPoints) {  
93         this.mountainPoints = mountainPoints;  
94     }  
95     /**  
96     * Adds the given input to the rider's total mountain  
97     * points classification points.  
98     *  
99     * @param mountainPoints The rider's mountain points  
100    * classification points for this stage.  
101   */  
102   public void addMountainPoints(int mountainPoints) {  
103       this.mountainPoints += mountainPoints;  
104   }
```

```
102
103     /**
104      * @return The rider's ID.
105     */
106    public int getRiderId() {
107      return riderId;
108    }
109
110   /**
111    * @return The stage ID from where the results are
112    * taken from.
113   */
114   public int getStageId() {
115     return stageId;
116   }
117
118   /**
119    * @return The times the rider passed each checkpoint.
120   */
121   public LocalTime[] getTimes() {
122     return times;
123   }
124
125   /**
126    * @return The rider's finish time.
127   */
128   public LocalTime getFinishTime() {
129     return finishTime;
130   }
131
132   /**
133    * @return The rider's finish time adjusted to the
134    * peloton leader's time if applicable.
135   */
136   public LocalTime getAdjustedFinishTime() {
137     return adjustedFinishTime;
138   }
139
140   /**
141    * @param newAdjustedFinishTime
142    */
143   public void setAdjustedFinishTime(LocalTime
144     newAdjustedFinishTime) {
145     this.adjustedFinishTime = newAdjustedFinishTime;
```

```
143     }
144
145     public LocalTime getElapsed() {
146         return elapsedTime;
147     }
148
149     public void setElapsed(LocalTime elapsedTime) {
150         this.elapsed = elapsedTime;
151     }
152
153     public LocalTime getAdjustedElapsed() {
154         return adjustedElapsed;
155     }
156
157     public void setAdjustedElapsed(LocalTime
158         adjustedElapsed) {
159         this.adjustedElapsed = adjustedElapsed;
160     }
161
```

```
1 package src.cycling;
2
3 import java.io.Serializable;
4 import java.util.ArrayList;
5
6 /**
7  * Represents an intermediate sprint. CategorisedClimb is
8  * a subclass of this for categorised
9  * climbs.
10 *
11 * @author Sam Barker, Adam Kaizrea
12 * @version 1.0
13 */
14 public class Segment implements Comparable<Segment>,
15     Serializable {
16
17     private static int latestId = 0;
18     private final Integer stageId;
19     private Double location;
20     private SegmentType segmentType;
21     private final Integer id;
22     private ArrayList<RiderSegmentResult> results = new
23         ArrayList<>();
24
25     /**
26      * Constructor.
27      *
28      * @param stageId The ID of the stage this sprint is
29      * part of.
30      * @param segmentType The type of segment (SPRINT)
31      * @param location The location of the segment in the
32      * stage.
33      */
34     public Segment(int stageId, SegmentType segmentType,
35     Double location) {
36         this.stageId = stageId;
37         this.segmentType = segmentType;
38         this.id = latestId++;
39         this.location = location;
40     }
41
42     /**
43      * Reset the internal ID counter.
44      */
45 }
```

```
39  public static void resetIdCounter() {
40      latestId = 0;
41  }
42
43  /**
44   * @param segment The segment to compare the calling
45   * object to.
46   * @return 1 if location of calling object > the passed
47   * in one. 0 if equal. -1 if less than.
48   */
49  public int compareTo(Segment segment) {
50      assert (segment instanceof Segment) : "Comparing
51      incorrect types!";
52      return this.getLocation().compareTo(segment.
53          getLocation());
54  }
55
56  /**
57   * @return The segment type.
58   */
59  public SegmentType getSegmentType() {
60      return segmentType;
61  }
62
63  /**
64   * Set the segment type.
65   *
66   * @param segmentType The segment type to set this
67   * object's segmentType to.
68   */
69  public void setSegmentType(SegmentType segmentType) {
70      this.segmentType = segmentType;
71  }
72
73  /**
74   * @return The ID of the stage this segment is part of.
75   */
76  public int getStageId() {
77      return this.stageId;
78  }
79
80  /**
81   * @return The ArrayList of RiderSegmentResults objects
82   * representing the per-segment,
```

```
77     * per-rider results in this segment.
78     */
79     public ArrayList<RiderSegmentResult> getResults() {
80         return results;
81     }
82
83     /**
84      * Set the ArrayList of RiderSegmentResults objects
85      * representing the per-segment, per-rider
86      * results in this segment.
87      *
88      * @param results
89      */
90     public void setResults(ArrayList<RiderSegmentResult>
91     results) {
92         this.results = results;
93     }
94
95     /**
96      * @return The ID of this particular segment.
97      */
98     public Integer getId() {
99         return id;
100    }
101
102    /**
103     * @return The location of the segment in the stage
104     */
105    public Double getLocation() {
106        return location;
107    }
108
109    /**
110     * Set the location of the segment in the stage.
111     *
112     * @param location
113     */
114     public void setLocation(Double location) {
115         this.location = location;
116     }
```

```
1 package src.cycling;
2
3 import java.io.Serializable;
4 import java.time.LocalDateTime;
5 import java.time.LocalTime;
6 import java.util.ArrayList;
7 import java.util.Collections;
8 import java.util.HashMap;
9 import java.util.Map;
10
11 /**
12  * Represents a stage in the race and contains all the
13  * rider's results for the stage. If a stage is
14  * under development results can not be registered but
15  * segments can be added.
16  *
17  */
18 public class Stage implements Serializable, Comparable<
19     Stage> {
20
21     private static int latestId = 0; // Enumerates to get
22     unique id, with 2^32 possible ids.
23     private final Integer raceId;
24     private String name;
25     private String description;
26     private Double length;
27     private LocalDateTime startTime;
28     private StageType stageType;
29     private final Integer id;
30     private final ArrayList<RiderStageResult> results = new
31     ArrayList<>();
32     private final ArrayList<Segment> segmentsInStage = new
33     ArrayList<>();
34     private Boolean underDevelopment = true; // Either under
35     development(T) or waiting results(F).
36
37     /**
38      * Constructor.
39      *
40      * @param raceId
41      * @param stageName
42      * @param description
```

```
38     * @param length
39     * @param startTime
40     * @param stageType
41     */
42     public Stage(Integer raceId, String stageName, String
43         description, Double length,
44         LocalDateTime startTime, StageType stageType) {
45         this.raceId = raceId;
46         this.name = stageName;
47         this.description = description;
48         this.length = length;
49         this.startTime = startTime;
50         this.stageType = stageType;
51         this.id = latestId++;
52     }
53
54     public int compareTo(Stage stage) {
55         assert (stage instanceof Stage) : "Comparing incorrect
56             types!";
57         return this.getStartTime().compareTo(stage.
58             getStartTime());
59     }
60
61     /**
62      * Reset the internal ID counter.
63      */
64     public static void resetIdCounter() {
65         latestId = 0;
66     }
67
68     /**
69      * @return The stage's ID.
70     */
71     public Integer getId() {
72         return id;
73     }
74
75     /**
76      * @return The ID of the staged race this stage is in.
77     */
78     public Integer getRaceId() {
79         return raceId;
80     }
```

```
79  /**
80   * @return The name of the stage.
81   */
82  public String getName() {
83      return name;
84  }
85
86  /**
87   * @param newName
88   */
89  public void setName(String newName) {
90      this.name = newName;
91  }
92
93  /**
94   * @return The description of the stage.
95   */
96  public String getDescription() {
97      return description;
98  }
99
100 /**
101  * @param newDescription
102  */
103 public void setDescription(String newDescription) {
104     this.description = newDescription;
105 }
106
107 /**
108  * @return The length of the stage measured in
kilometers.
109 */
110 public Double getLength() {
111     return length;
112 }
113
114 /**
115  * @param newLength
116  */
117 public void setLength(Double newLength) {
118     this.length = newLength;
119 }
120
121 /**
```

```
122     * @return The local date and time of the start of the
123     * stage.
124     */
125     public LocalDateTime getStartTime() {
126         return startTime;
127     }
128     /**
129     * @param newStartTime
130     */
131     public void setStartTime(LocalDateTime newStartTime) {
132         this.startTime = newStartTime;
133     }
134
135     /**
136     * @return terrain type of the stage.
137     */
138     public StageType getStageType() {
139         return stageType;
140     }
141
142     /**
143     * @param newStageType
144     */
145     public void setStageType(StageType newStageType) {
146         this.stageType = newStageType;
147     }
148
149     /**
150     * @return An array list of all the segments in this
151     * stage.
152     */
153     public ArrayList<Segment> getSegmentsInStage() {
154         return segmentsInStage;
155     }
156     /**
157     * Adds a new segment to the stage.
158     *
159     * @param newSegment
160     */
161     public void addSegment(Segment newSegment) {
162         segmentsInStage.add(newSegment);
163         Collections.sort(segmentsInStage); // Sorts the
```

```
163 segments by location.  
164 }  
165  
166 /**
167 * @return The state of the stage. Either under
168 development (true) or waiting results (false).
169 */
170 public Boolean getUnderDevelopment() {
171     return underDevelopment;
172 }
173 /**
174 * If true the state of the stage is under development
175 . If false the state of the stage is
176     * awaiting results.
177 *
178 * @param newState
179 */
180 public void setUnderDevelopment(Boolean newState) {
181     this.underDevelopment = newState;
182 }
183 /**
184 * @return An array list containing all the
185 RiderStageResult objects relating to this stage.
186 */
187 public ArrayList<RiderStageResult> getResults() {
188     return this.results;
189 }
190 /**
191 * Removes the results of a rider from a stage.
192 *
193 * @param riderId The rider ID of the rider to be
194 removed.
195 */
196 public void removeResultByRiderId(int riderId) throws
197 IDNotRecognisedException {
198     boolean riderIdFound = false;
199     RiderStageResult finalResult = null; // Empty value
200     to be assigned if the rider is found.
201     for (RiderStageResult result : results) { // Iterates
202         through the stage's rider results to find
```

```

200      // the correct rider.
201      if (result.getRiderId() == riderId) {
202          riderIdFound = true;
203          finalResult = result;
204          break;
205      }
206  }
207  if (!riderIdFound) { // If the rider can't be found
208      it can't be removed.
209      throw new IDNotRecognisedException("Rider ID " +
210          riderId + " not found in stage!");
211  }
212  results.remove(finalResult); // Removes the result
213  from the stage results array list.
214  assert !results.contains(finalResult) : "Rider's
215  result not removed!";
216
217 /**
218  * Creates a new RiderStageResult object and adds it to
219  * an array list of rider results in this
220  * stage.
221  *
222  * @param riderId
223  * @param times The times the rider passed each
224  * checkpoint in chronological order.
225  */
226  public void addRiderResults(Integer riderId, LocalTime
227  [] times) {
228      RiderStageResult result = new RiderStageResult(
229          riderId, this.id, times); // Creates new result.
230      results.add(result); // The new result object is
231      added to an array list of all result objects in
232      // the stage.
233      assert (results.contains(result)) : "Result object
234      not created!";
235      Collections.sort(results); // Sorts the results by
236      their finishing times.
237  }
238
239 /**
240  * Searches all the stages' riders' results to find
241  * pelotons (where riders are less than one
242  * second behind the rider in front of them) and sets

```

```

231 the adjusted time of each applicable result
232     * to the peloton leader's finish time.
233     */
234     public void generateAdjustedFinishTimes() {
235         if (this.getStageType() != StageType.TT) {
236
237             // Initialise previous time and peloton leader time
238             // to zero.
239             LocalTime previousTime = LocalTime.of(0, 0, 0, 0);
240             LocalTime pelotonLeader = LocalTime.of(0, 0, 0);
241
242             for (RiderStageResult result : this.results) { // Iterates
243                 // through results of the stage.
244                 LocalTime currentTime = result.getFinishTime();
245                 // The time of this rider to compare
246                 // against.
247                 double currentTimeSeconds =
248                     (currentTime.getHour() * 3600) + (currentTime.
249                     .getMinute() * 60) +
250                         currentTime.getSecond();
251                 double previousTimeSeconds = (previousTime.
252                     getHour() * 3600) + (previousTime.getMinute() *
253                         60 + previousTime.getSecond());
254                 // Converts LocalTime into double to calculate
255                 // whether there is a 1 second or less gap
256                 // between riders.
257                 if (((currentTimeSeconds - previousTimeSeconds
258 ) <= 1.0) && (previousTimeSeconds != 0)) {
259                     result.setAdjustedFinishTime(pelotonLeader);
260                     // If there is a 1 second or less gap the
261                     // adjusted time is set to the peloton leader.
262                 } else {
263                     pelotonLeader = currentTime; // If the rider is
264                     // not in a peloton then they could be the
265                     // next peloton's leader.
266                     result.setAdjustedFinishTime(currentTime); // Adjusted
267                     // time is set to their finish time.
268                 }
269                 previousTime = currentTime; // A Peloton can
270                 // consist of multiple riders so the one second
271                 // gap accounts for between riders.
272
273                 result.setAdjustedElapsedTime(SumLocalTimes.
274                     subtractLocalTimes(

```

```
263             result.getAdjustedFinishTime(), result.  
264             getTimes()[0]));  
265         }  
266         Collections.sort(this.results); // The results are  
267         sorted by finishing times (same order as  
268         // adjusted).  
269     }  
270  
271     /**  
272      * Iterates through all the segments in the stage to  
273      * create RiderSegmentResult objects from the  
274      * results in RiderStageResults.  
275      */  
276     public void generateRiderSegmentResults() {  
277         Collections.sort(this.segmentsInStage); // Order  
278         segments by their location.  
279         int segmentCounter = 1; // Starts from 1 as initial  
280         time is start time.  
281         for (Segment segment : this.segmentsInStage) { //  
282             Iterates through every segment.  
283             ArrayList<RiderSegmentResult> riderSegmentResults  
284             = new ArrayList<>();  
285             // Stores segment results then sorts.  
286             for (RiderStageResult result : results) {  
287                 RiderSegmentResult newRiderSegmentResult = new  
288                     RiderSegmentResult(result.getTimes()[  
289                     segmentCounter], result.getRiderId());  
290                     riderSegmentResults.add(newRiderSegmentResult);  
291                     assert riderSegmentResults.contains(  
292                     newRiderSegmentResult);  
293                     }  
294                     // Sorts riders segment results in order of their  
295                     time in ascending order.  
296                     Collections.sort(riderSegmentResults);  
297                     // Stores these results in segment for more  
298                     intuitive access.  
299                     segment.setResults(riderSegmentResults);  
300  
301                     // Sets rank of each rider in segment.  
302                     int rank = 0;  
303                     for (RiderSegmentResult result :  
304                     riderSegmentResults) {  
305                         result.setRank(rank);
```

```

295         rank++;
296     }
297     segmentCounter++;
298 }
299     assert (this.segmentsInStage.isEmpty() || (results.
300         isEmpty()) || !this.segmentsInStage.get(0)
301             .getResults().isEmpty());
302 }
303 /**
304 * Calculates either mountain or point classification
305 points for a stage and stores them in each
306 * rider's stage result object.
307 *
308 * param isMountain True if mountain classification,
309 false if point classification.
310 * return An array of rider ID's ordered by their
311 points in the given classification.
312 */
313 public int[] generatePointsInStage(boolean isMountain
) {
314
315     Map<Enum, int[]> pointsConversion;
316
317     if (isMountain) {
318         // Table of rank to points for each segment type in
319         // mountain classification.
320         int[] pointsHC = {20, 15, 12, 10, 8, 6, 4, 2}; // TOP 8 ONLY.
321         int[] pointsC1 = {10, 8, 6, 4, 2, 1, 0, 0}; // TOP
322         6 ONLY.
323         int[] pointsC2 = {5, 3, 2, 1, 0, 0, 0, 0}; // TOP 4
324         ONLY.
325         int[] pointsC3 = {2, 1, 0, 0, 0, 0, 0, 0}; // TOP 2
326         ONLY.
327         int[] pointsC4 = {1, 0, 0, 0, 0, 0, 0, 0}; // TOP 1
328         ONLY.
329         pointsConversion = Map.of(
330             SegmentType.HC, pointsHC,
331             SegmentType.C1, pointsC1,
332             SegmentType.C2, pointsC2,
333             SegmentType.C3, pointsC3,
334             SegmentType.C4, pointsC4);
335     } else {

```

```

328     // Table of rank to points for each stage type in
     point classification.
329     int[] flatPointsConversion = {50, 30, 20, 18, 16,
     14, 12, 10, 8, 7, 6, 5, 4, 3, 2};
330     int[] hillyPointsConversion = {30, 25, 22, 19, 17,
     15, 13, 11, 9, 7, 6, 5, 4, 3, 2};
331     int[] mountainPointsConversion = {20, 17, 15, 13,
     11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
332     pointsConversion = Map.of(
333         StageType.FLAT, flatPointsConversion,
334         StageType.MEDIUM_MOUNTAIN,
     hillyPointsConversion,
335         StageType.HIGH_MOUNTAIN,
     mountainPointsConversion,
336         StageType.TT, mountainPointsConversion // Time
     trials are awarded the same points as high
337         // mountain stages.
338     );
339 }
340
341     if (isMountain) {
342         // Do mountain error checks.
343         if ((this.getStageType() == StageType.TT) || (this.
     getSegmentsInStage().size() == 0)) {
344             // No mountain points (no segments).
345             return new int[0];
346         }
347
348         // Check if there even are any climbs.
349         boolean climbFound = false;
350         for (Segment segment : this.getSegmentsInStage()) {
351             if (segment.getSegmentType() != SegmentType.
     SPRINT) {
352                 climbFound = true;
353             }
354         }
355         // No climbs so there will be no results.
356         if (!climbFound) {
357             return new int[0];
358         }
359     }
360
361     HashMap<Integer, Integer> riderIdsToPoints = new
     HashMap<Integer, Integer>();

```

```

362      // Sum of points for each rider for the specified
race.
363
364      // For the points classification, add on the points
from finish times which mountain doesn't
365      // include.
366      if (!isMountain) {
367          int pointsIndex = 0;
368          this.generateAdjustedFinishTimes(); // Sorts times
in ascending order.
369
370          for (RiderStageResult result : this.getResults
()) { // Iterate through rider.
371              int riderId = result.getRiderId();
372              int points;
373              if (pointsIndex < 15) { // Only first 15 riders
are awarded points.
374                  points = pointsConversion.get(this.getStageType
())[pointsIndex]; // Points for stage type.
375                  assert points > 0 : "Incorrect points assigned
!";
376              } else {
377                  points = 0;
378              }
379              riderIdsToPoints.put(riderId, points);
380              result.setPoints(points);
381              pointsIndex++;
382          }
383      }
384
385      // Add on points from segments.
386      ArrayList<Segment> segmentsInStage = this.
getSegmentsInStage();
387      this.generateRiderSegmentResults(); // Creates
segment objects.
388      for (Segment segment : segmentsInStage) {
389          // Cast for the appropriate Enum for mountain or
points as a key for their points conversion
390          // table.
391          SegmentType currentSegmentType = segment.
getSegmentType();
392          Enum mapKey;
393          if (isMountain) {
394              mapKey = currentSegmentType;

```

```

395     } else {
396         mapKey = StageType.HIGH_MOUNTAIN; // Intermediate
397         // sprints are the only type of segment in
398         // points classification and are awarded the same
399         // points as high mountain stages.
400     }
401
402     // Only allow intermediate sprint + points
403     // classification or categorised climb + mountain
404     // classification combinations.
405
406     if (((currentSegmentType == SegmentType.SPRINT
407 ) && !isMountain)
408         || (!(currentSegmentType == SegmentType.SPRINT
409 ) && isMountain)) {
410         int i = 0;
411         for (RiderSegmentResult riderResult : segment.
412             getResults()) { // Iterates through each
413             // rider's segment results.
414             int riderId = riderResult.getRiderId();
415             int riderRank = riderResult.getRank(); // Gets
416             // riders rank in segment.
417             int points;
418             int[] rowOfPointsConversion = pointsConversion.
419             get(mapKey);
420             int pointsLimit = rowOfPointsConversion.length;
421             if (riderRank < pointsLimit) {
422                 points = rowOfPointsConversion[riderRank];
423                 // Points limit prevent index out of bounds
424                 // error.
425
426                 assert points >= 0 : "Negative point value
427 assigned!";
428             } else {
429                 points = 0;
430             }
431
432             if (riderIdsToPoints.get(riderId) == null) {
433                 // If the rider is not registered with points
434                 // add them.
435                 riderIdsToPoints.put(riderId, points);
436                 if (!isMountain) {
437                     for (RiderStageResult stageResult : this.
438                         getResults()) {

```

```

427             if (stageResult.getRiderId() ==
riderResult.getRiderId()) {
428                 stageResult.setPoints(points);
429             }
430         }
431     } else {
432         for (RiderStageResult stageResult : this.
getResults()) {
433             if (stageResult.getRiderId() ==
riderResult.getRiderId()) {
434                 stageResult.setMountainPoints(points);
435             }
436         }
437     }
438 } else { // Else sum their points to their
total points.
439     riderIdsToPoints.merge(riderId, points,
Integer::sum);
440     if (!isMountain) {
441         for (RiderStageResult stageResult : this.
getResults()) {
442             if (stageResult.getRiderId() ==
riderResult.getRiderId()) {
443                 stageResult.addPoints(points);
444             }
445         }
446     } else {
447         for (RiderStageResult stageResult : this.
getResults()) {
448             if (stageResult.getRiderId() ==
riderResult.getRiderId()) {
449                 stageResult.addMountainPoints(points);
450             }
451         }
452     }
453 }
454 i++;
455 }
456 }
457 }
458 Collections.sort(this.results);
459 // Creates an int array of points ordered by rank
460 if (!(riderIdsToPoints.isEmpty())) {
461     int[] pointsOrderedByRank = new int[this.results.

```

```
461 size()];
462     int i = 0;
463     for (RiderStageResult result : this.getResults()) {
464         if (isMountain) {
465             pointsOrderedByRank[i] = result.
466             getMountainPoints();
467         } else {
468             pointsOrderedByRank[i] = result.getPoints();
469         }
470     }
471     return pointsOrderedByRank;
472 } else {
473     return new int[0];
474 }
475 }
476 }
477 }
```

```
1 package src.cycling;
2
3 import java.io.Serializable;
4 import java.time.LocalDateTime;
5 import java.util.ArrayList;
6 import java.util.Collections;
7
8 /**
9  * Represents a grand tour staged race.
10 *
11 * @author Sam Barker, Adam Kaizra
12 * @version 1.0
13 */
14 public class StagedRace implements Serializable {
15
16     private static int latestId = 0;
17     private final String name;
18     private final String description;
19     private final int raceId;
20     private LocalDateTime[] datesOfCompetitions;
21     private final ArrayList<Stage> stages = new ArrayList<>();
22     private final ArrayList<RiderRaceResult> raceResults =
23         new ArrayList<>();
24
25     /**
26      * Constructor.
27      *
28      * @param name
29      * @param description
30     */
31     public StagedRace(String name, String description) {
32         this.name = name;
33         this.description = description;
34         this.raceId = latestId++;
35     }
36
37     /**
38      * Reset the internal ID counter.
39      */
40     public static void resetIdCounter() {
41         latestId = 0;
42     }
43 }
```

```
43  /**
44   * @return The name of the staged race.
45   */
46  public String getName() {
47      return this.name;
48  }
49
50 /**
51  * @return The description of the staged race.
52  */
53  public String getDescription() {
54      return this.description;
55  }
56
57 /**
58  * @return The ID of the staged race.
59  */
60  public int getId() {
61      return this.raceId;
62  }
63
64 /**
65  * @return An array list of stages in the staged race.
66  */
67  public ArrayList<Stage> getStages() {
68      return stages;
69  }
70
71 /**
72  * @param stage The new stage to be added to the staged
73  * race.
74  */
75  public void addStage(Stage stage) {
76      this.stages.add(stage);
77  }
78
79 /**
80  * Sums all the riders stage results finish times and
81  * stores it as a race finish time in each
82  * rider's race result object.
83  */
84  public void generateRidersRaceAdjustedElapsedTimes() {
85      int sizeOfArrayOfTotalTimes;
86      if (this.getStages().size() == 0) {
```

```
85     return;
86 } else {
87     sizeOfArrayOfTotalTimes = this.getStages().get(0).
88     getResults().size();
89 }
90 LocalTime[] arrayOfTotalTimes = new LocalTime[
91     sizeOfArrayOfTotalTimes];
92 for (Stage stage : this.getStages()) { // Iterates
93     through all stages in a race.
94     int i = 0;
95     stage.generateAdjustedFinishTimes();
96
97     for (RiderStageResult riderStageResult : stage.
98         getResults()) { // Iterates through all results
99         // in a stage.
100        int riderId = riderStageResult.getRiderId();
101        boolean riderFound = false;
102        for (RiderRaceResult raceResult : this.
103            raceResults) { // Iterates through race results.
104            if (raceResult.getRiderId() == riderId) { // If
105                the race result and stage result refer to
106                // the same rider.
107                arrayOfTotalTimes[i] = SumLocalTimes.
108                addLocalTimes(arrayOfTotalTimes[i],
109                    riderStageResult.getAdjustedElapsed Time
110                    ());
111                    // Sums race results finish time with new
112                    stages finish time.
113                    riderFound = true;
114                    break;
115                    }
116                }
117                if (!riderFound) { // If no race result for rider
118                    one is made.
119                    RiderRaceResult raceResult = new
120                    RiderRaceResult(riderStageResult.getRiderId(),
121                        this.raceId);
122                    arrayOfTotalTimes[i] = riderStageResult.
123                    getAdjustedElapsed Time();
124                    int raceResultsSize = this.raceResults.size();
125                    this.raceResults.add(raceResult);
126                    assert this.raceResults.size() ==
127                    raceResultsSize + 1 : "Race result not added to array"
```

```
116             + "list!";
117         }
118         i++;
119     }
120 }
121 int i = 0;
122 for (RiderRaceResult result : this.raceResults) {
123     result.setElapsedTime(arrayOfTotalTimes[i]);
124     i++;
125 }
126 Collections.sort(this.raceResults); // Sorts race
results by total time.
127 }
128
129 public ArrayList<RiderRaceResult> getResults() {
130     return raceResults;
131 }
132
133 /**
134 * Sums all the mountain or point classification points
for each rider in the race.
135 *
136 * @param isMountain True if mountain point
classification, false if point classification.
137 * @return An array of rider's total points respective
of given classification.
138 */
139 public int[] generateRidersPointsInRace(boolean
isMountain) {
140     this.generateRidersRaceAdjustedElapsedTimes();
141
142     int[] points = new int[this.raceResults.size()]; // Points for the entire race.
143     for (Stage stage : this.getStages()) { // Iterate
through stages.
144         stage.generatePointsInStage(isMountain); // Points
ordered by rank are generated.
145         for (RiderStageResult stageResult : stage.
getResults()) {
146             int i = 0;
147             if (isMountain) { // Mountain classification.
148                 for (RiderRaceResult raceResult : this.
getResults()) { // Iterates through race results.
149                     // If the stage result rider ID and the race
```

```
149 result rider ID match.  
150         if (stageResult.getRiderId() == raceResult.  
151             getRiderId()) {  
152                 // Points are set or added to the race  
153                 total array.  
154                 if (points[i] == 0) {  
155                     points[i] = stageResult.getMountainPoints  
156                     ();  
157                 } else {  
158                     int stagePoints = stageResult.  
159                     getMountainPoints();  
160                     assert stagePoints >= 0 : "Stage mountain  
161                     points can not be negative!";  
162                     points[i] += stagePoints;  
163                 }  
164             }  
165         }  
166     }  
167     }  
168     i++;  
169 }  
170 }  
171 }  
172 }  
173 }  
174 }  
175 }  
176 }  
177 }  
178 // Iterates through race results and sets each  
179 // rider's total points for the race.  
180 int i = 0;  
181 for (RiderRaceResult result : this.getResults()) {  
182     if (isMountain) {
```

```
182         result.setMountainPoints(points[i]);
183     } else {
184         result.setPoints(points[i]);
185     }
186     i++;
187 }
188 }
189 // Sum of points for each rider for the specified
race.
190 int resultsSize = this.raceResults.size();
191 int[] racePoints = new int[resultsSize];
192 int i = 0;
193
194 // Sorts race results by either their mountain or
point classification points.
195 if (isMountain) {
196     Collections.sort(raceResults, RiderRaceResult::
compareByMountainPoints);
197 } else {
198     Collections.sort(raceResults, RiderRaceResult::
compareByPoints);
199 }
200 for (RiderRaceResult result : this.raceResults) {
201     if (isMountain) {
202         racePoints[i] = result.getMountainPoints();
203     } else {
204         racePoints[i] = result.getPoints();
205     }
206     i++;
207 }
208 Collections.sort(raceResults); // Sorts race results
back to its order by finishing times.
209 System.out.println();
210 System.out.println("start");
211 for (RiderRaceResult result : raceResults) {
212     System.out.println("ID "+result.getRiderId()+""
points "+result.getMountainPoints());
213 }
214
215 return racePoints;
216 }
217
218 /**
219 * param isMountain True if mountain point
```

```
219 classification, false if point classification.  
220     * @return An array of rider ID's ordered by their  
221     points in the given classification.  
222     */  
223     public int[] getRiderIdsOrderedByPoints(boolean  
224         isMountain) {  
225         ArrayList<RiderRaceResult> riderRaceResults = this.  
226             getResults();  
227         // Sorts race results by their points in the given  
228         // classification.  
229         if (isMountain) {  
230             Collections.sort(riderRaceResults, RiderRaceResult  
231                 ::compareByMountainPoints);  
232         } else {  
233             Collections.sort(riderRaceResults, RiderRaceResult  
234                 ::compareByPoints);  
235         }  
236         // Gets the rider ID's in order of points and adds  
237         // them to an array.  
238         int[] riderIdsByPoints = new int[riderRaceResults.  
239             size()];  
240         int i = 0;  
241         for (RiderRaceResult result : riderRaceResults) {  
242             riderIdsByPoints[i] = result.getRiderId();  
243             i++;  
244         }  
245         Collections.sort(raceResults); // Sorts race results  
246         // back to its order by finishing times.  
247         assert riderRaceResults.size() == riderIdsByPoints.  
248             length : "Gathered rider IDs incorrectly!";  
249         return riderIdsByPoints;  
250     }  
251 }
```

```
1 package src.cycling;
2
3 import java.time.LocalTime;
4
5 /**
6  * A helper object whose static methods deal with the
7  * addition of LocalTimes
8  * and conversion between seconds (doubles) and LocalTime
9  * objects, of times.
10 *
11 */
12 public class SumLocalTimes {
13
14 /**
15  * Convert a LocalTime object to the number of seconds
16  * after midnight.
17  *
18  * @param time
19  * @return The time after midnight in seconds that a
20  * LocalTime object represents.
21  */
22 public static double localTimeToSeconds(LocalTime time
23 ) {
24     if (time == null) {
25         return 0;
26     }
27     return (time.getHour() * 3600)
28         + (time.getMinute() * 60)
29         + (time.getSecond());
30 }
31
32 /**
33  * Create a LocalTime object from the number of seconds
34  * after midnight.
35  *
36  * @param timeSeconds
37  * @return The LocalTime object representing the time
38  * given.
39  */
40 public static LocalTime secondsToLocalTime(double
41 timeSeconds) {
42     int hours = (int) (timeSeconds / 3600);
```

```
37     int minutes = (int) (timeSeconds / 60) - (hours * 60);
38     int seconds = (int) ((timeSeconds % 3600) % 60);
39     return LocalTime.of(hours, minutes, seconds);
40 }
41
42 /**
43 * Add an arbitrary number of LocalTimes together.
44 *
45 * @param times
46 * @return The sum of the given LocalTimes.
47 */
48 public static LocalTime addLocalTimes(LocalTime... times
49 ) {
50     // Convert the LocalTimes to seconds, sum them all up
51     // , then convert back to LocalTime.
52     double finalTimeSeconds = 0;
53     for (LocalTime time : times) {
54         double timeSeconds = localTimeToSeconds(time);
55         finalTimeSeconds += timeSeconds;
56     }
57
58     return secondsToLocalTime(finalTimeSeconds);
59 }
60
61 /**
62 * Subtract two LocalTime variables.
63 *
64 * @param t1 The time to have t2 taken away from.
65 * @param t2 The time to take away from t1
66 * @return t1 minus t2
67 */
68 public static LocalTime subtractLocalTimes(LocalTime t1
69 , LocalTime t2) {
70     LocalTime finalTime;
71
72     // Convert times to seconds
73     double t1s = localTimeToSeconds(t1);
74     double t2s = localTimeToSeconds(t2);
75     // Subtract
76     double t3s = t1s - t2s;
77
78     assert t3s >= 0 : "LocalTime cannot represent negative
79     values!";
80 }
```

```
77      // Convert back to LocalTime and return
78      return secondsToLocalTime(t3s);
79  }
80
81 }
82
```

```
1 package src.cycling;
2
3 import java.io.Serializable;
4 import java.util.HashMap;
5
6 /**
7  * Teams are made up of riders and can be placed into
8  * races.
9  *
10 * @author Adam Kaizra, Sam Barker
11 * @version 1.0
12 */
13
14 public class Team implements Serializable {
15     private static int latestId = 0; // Enumerates to get
16     // unique id, with 2^32 possible ids.
17     private String name;
18     private String description;
19     private int id;
20     // Creates a hash map between the team's rider's Ids and
21     // the rider objects.
22     private HashMap<Integer, Rider> riderIdsToRiders = new
23     HashMap<Integer, Rider>();
24
25     /**
26      * Constructor.
27      *
28      * @param name
29     */
30     public Team(String name, String description) {
31         this.name = name;
32         this.description = description;
33         this.id = latestId++;
34     }
35
36     /**
37      * Reset the internal ID counter.
38      */
39     public static void resetIdCounter() {
40         latestId = 0;
41     }
42
43     /**
44      * @return The name of the team.
45     }
```

```
41     */
42     public String getName() {
43         return this.name;
44     }
45
46     /**
47      * param newTeamName
48      */
49     public void setName(String newTeamName) {
50         this.name = newTeamName;
51     }
52
53     /**
54      * return The description of the team.
55      */
56     public String getDescription() {
57         return this.description;
58     }
59
60     /**
61      * return The ID of the team.
62      */
63     public int getId() {
64         return this.id;
65     }
66
67     /**
68      * return A hash map between the team's rider's IDs and
69      * their corresponding rider objects.
70      */
71     public HashMap<Integer, Rider> getRiderIdsToRiders() {
72         return riderIdsToRiders;
73     }
74
75     /**
76      * param rider A rider object to be added to the team's
77      * internal hashmap of rider ID's to riders.
78      */
79     public void addRider(Rider rider) {
80         this.riderIdsToRiders.put(rider.getId(), rider);
81     }
82 }
```

```
1 package src.cycling;
2
3 import java.io.FileInputStream;
4 import java.io.FileOutputStream;
5 import java.io.IOException;
6 import java.io.ObjectInputStream;
7 import java.io.ObjectOutputStream;
8 import java.time.LocalDateTime;
9 import java.time.LocalTime;
10 import java.util.ArrayList;
11 import java.util.Arrays;
12 import java.util.Collections;
13 import java.util.HashMap;
14 import java.util.Map;
15 import java.util.Set;
16
17 /**
18 * Implementor of the CyclingPortalInterface interface.
19 *
20 * @author Adam Kaizra, Sam Barker
21 * @version 1.0
22 */
23
24 public class CyclingPortal implements
25     CyclingPortalInterface {
26
27     private HashMap<Integer, StagedRace> raceIdsToRaces =
28         new HashMap<>();
29     private HashMap<Integer, Stage> stageIdsToStages = new
30         HashMap<>();
31     private HashMap<Integer, Segment> segmentIdsToSegments =
32         new HashMap<>();
33     private HashMap<Integer, Team> teamIdsToTeams = new
34         HashMap<>();
35     private HashMap<Integer, Rider> riderIdsToRiders = new
36         HashMap<>();
37
38     /**
39      * @return The hashmap of each race's ID to its
40      * corresponding StagedRace object.
41     */
42     public HashMap<Integer, StagedRace> getRaceIdsToRaces
43     () {
44         return raceIdsToRaces;
```

```
37    }
38
39    /**
40     * @return The hashmap of each stage's ID to its
41     * corresponding Stage object.
42     */
43    public HashMap<Integer, Stage> getStageIdsToStages() {
44        return stageIdsToStages;
45    }
46
47    /**
48     * @return The hashmap of each segment's ID to its
49     * corresponding Stage object.
50     */
51    public HashMap<Integer, Segment> getSegmentIdsToSegments()
52    {
53        return segmentIdsToSegments;
54    }
55
56    /**
57     * @return The hashmap of each team's ID to its
58     * corresponding Team object.
59     */
60    public HashMap<Integer, Team> getTeamIdsToTeams() {
61        return teamIdsToTeams;
62    }
63
64    /**
65     * @return The hashmap of each rider's ID to its
66     * corresponding Rider object.
67     */
68    public HashMap<Integer, Rider> getRiderIdsToRiders() {
69        return riderIdsToRiders;
70    }
71
72    @Override
73    public int[] getRaceIds() {
74        Set<Integer> raceIdsSet = raceIdsToRaces.keySet();
75        int[] raceIds = new int[raceIdsSet.size()];
76        int index = 0;
77        for (Integer i : raceIdsSet) {
78            raceIds[index++] = i;
79        }
80    }
81
82    /**
83     * @return The hashmap of each race's ID to its
84     * corresponding Race object.
85     */
86    public HashMap<Integer, Race> getRaceIdsToRaces() {
87        return raceIdsToRaces;
88    }
89
90    /**
91     * @return The hashmap of each team's ID to its
92     * corresponding Team object.
93     */
94    public HashMap<Integer, Team> getTeamIdsToTeams() {
95        return teamIdsToTeams;
96    }
97
98    /**
99     * @return The hashmap of each rider's ID to its
100    * corresponding Rider object.
101   */
102  public HashMap<Integer, Rider> getRiderIdsToRiders() {
103      return riderIdsToRiders;
104  }
105
106  /**
107   * @return The hashmap of each race's ID to its
108   * corresponding Race object.
109  */
110 public HashMap<Integer, Race> getRaceIdsToRaces() {
111     return raceIdsToRaces;
112 }
```

```

76     return raceIds;
77 }
78
79 @Override
80 public int createRace(String name, String description)
81     throws IllegalNameException, InvalidNameException {
82     // Trim leading and trailing whitespace and check if
83     // the name is blank, is more than 30
84     // characters in length, or is null.
85     name = name.trim();
86     if (name.equals("") || name.length() > 30 || name ==
87         null) {
88         throw new InvalidNameException("Race name '" + name
89             + "' is invalid!");
90     }
91     // Search hashmap of races for one with the given
92     // name and throw exception if found.
93     for (StagedRace stagedRace : raceIdsToRaces.values
94         ()) {
95         String stagedRaceName = stagedRace.getName();
96         if (stagedRaceName.equals(name)) {
97             throw new IllegalNameException("There is already
98             a race with the name '" + name + "'!");
99         }
100    }
101
102    // For assertion.
103    int noOfRaces = raceIdsToRaces.size();
104
105    StagedRace raceToAdd = new StagedRace(name,
106        description);
107    int raceToAddId = raceToAdd.getId();
108    raceIdsToRaces.put(raceToAddId, raceToAdd);
109    assert raceIdsToRaces.size() == noOfRaces + 1 : "Race
110        not added to hashmap!";
111
112    return raceToAdd.getId();
113 }
114
115 @Override
116 public String viewRaceDetails(int raceId) throws
117     IDNotRecognisedException {
118     StagedRace race = raceIdsToRaces.get(raceId);

```

```

111     if (race == null) {
112         throw new IDNotRecognisedException("Race ID " +
113             raceId + " is not recognised!");
114     }
115
116     // Calculate total race length.
117     int sumOfStagesLengths = 0;
118     ArrayList<Stage> stages = race.getStages();
119     for (Stage stage : stages) {
120         sumOfStagesLengths += stage.getLength();
121     }
122
123     // Add details to status output string
124     return "Race ID: " + race.getId() + "\n"
125         + "Race name: " + race.getName() + "\n"
126         + "Description: " + race.getDescription() + "\n"
127         + "Number of stages: " + race.getStages().size()
128         () + "\n"
129         + "Total race length: " + sumOfStagesLengths;
130 }
131
132 @Override
133 public void removeRaceById(int raceId) throws
134 IDNotRecognisedException {
135     StagedRace race = raceIdsToRaces.get(raceId);
136     if (race == null) {
137         throw new IDNotRecognisedException("Race ID " +
138             raceId + " is not recognised!");
139     }
140
141     // Remove race result objects.
142     for (RiderRaceResult result : race.getResults()) {
143         race.getResults().remove(result);
144         assert !race.getResults().contains(result) : "Race
145             result not removed!";
146     }
147
148     // Remove stages of the race
149     for (Stage stage : race.getStages()) {
150         race.getStages().remove(stage);
151         assert !race.getStages().contains(stage) : "Stage
152             not removed!";
153         try {
154             removeStageById(stage.getId());
155         }
156     }
157 }
```

```
149         } catch (IDNotRecognisedException ex) {
150             System.out.println("Stage waiting for results and
151             cannot be deleted!");
152         }
153     }
154     // Remove race results
155
156     raceIdsToRaces.remove(raceId);
157     assert raceIdsToRaces.get(raceId) == null : "Race not
158     successfully removed!";
159 }
160
161     @Override
162     public int getNumberOfStages(int raceId) throws
163     IDNotRecognisedException {
164         StagedRace race = raceIdsToRaces.get(raceId);
165         // Does the race exist?
166         if (race == null) {
167             throw new IDNotRecognisedException("Race ID " +
168             raceId + " is not recognised!");
169         }
170
171     @Override
172     public int addStageToRace(int raceId, String name,
173     String description, double length,
174     LocalDateTime startTime, StageType type)
175     throws IDNotRecognisedException,
176     IllegalNameException, InvalidNameException,
177     InvalidLengthException {
178         // Check the race exists.
179         StagedRace race = raceIdsToRaces.get(raceId);
180         if (race == null) {
181             throw new IDNotRecognisedException("Race ID " +
182             raceId + " is not recognised!");
183         }
184         // Trim leading and trailing whitespace and check if
185         // the name is blank, is more than 30
186         // characters in length, or is null.
187         name = name.trim();
188         if (name.equals("") || name.length() > 30 || name ==
```

```
184 null) {  
185     throw new InvalidNameException("Name '" + name +  
186         "' is greater than 30 characters!");  
187     }  
188     // Check the length is >=5km.  
189     if (length < 5.0) {  
190         throw new InvalidLengthException("Length of stage  
191             must be greater than or equal to 5km!");  
192         }  
193     // For assertion.  
194     int amountOfStages = race.getStages().size();  
195  
196     // Create stage and add to hashmap for easier access.  
197     Stage stage = new Stage(raceId, name, description,  
198         length, startTime, type);  
199     stageIdsToStages.put(stage.getId(), stage);  
200     race.addStage(stage);  
201     assert race.getStages().size() == amountOfStages + 1;  
202  
203     return stage.getId();  
204 }  
205  
206 @Override  
207 public int[] getRaceStages(int raceId) throws  
208     IDNotRecognisedException {  
209     StagedRace race = raceIdsToRaces.get(raceId);  
210     if (race == null) {  
211         throw new IDNotRecognisedException("Race ID " +  
212             raceId + " is not recognised!");  
213     }  
214  
215     // Sorts stages by their starting time to get the  
216     // order.  
217     ArrayList<Stage> stages = race.getStages();  
218     Collections.sort(stages);  
219  
220     // Iterates though stages in race and gets their IDs.  
221     int[] stageIds = new int[stages.size()];  
222     int i = 0;  
223     for (Stage stage : stages) {  
224         stageIds[i] = stage.getId();  
225     }
```

```
222         i++;
223     }
224
225     assert race.getStages().size() == stageIds.length : "
226         Incorrect amount of stage IDs!";
227
228     return stageIds;
229 }
230
231 @Override
232 public double getStageLength(int stageId) throws
233     IDNotRecognisedException {
234     Stage stage = stageIdsToStages.get(stageId);
235     if (stage == null) {
236         throw new IDNotRecognisedException("Stage ID " +
237             stageId + " is not recognised!");
238     }
239     return stage.getLength();
240 }
241
242 @Override
243 public void removeStageById(int stageId) throws
244     IDNotRecognisedException {
245     Stage stage = stageIdsToStages.get(stageId);
246     if (stage == null) {
247         throw new IDNotRecognisedException("Stage ID " +
248             stageId + " is not recognised!");
249     }
250     // Remove stage results
251     for (RiderStageResult result : stage.getResults()) {
252         stage.getResults().remove(result);
253         assert !stage.getResults().contains(result) :
254             Stage result not removed!;
255     }
256     // Remove segments
257     for (Segment segment : stage.getSegmentsInStage()) {
258         try {
259             removeSegment(segment.getId());
260         } catch (InvalidStageStateException ex) {
261             System.out.println("Segment waiting for results!");
262         };
263     }
264 }
```

```
259     // Get the race object that contains it.
260     StagedRace race = raceIdsToRaces.get(stage.getRaceId
261     ());
261     race.getStages().remove(stage);
262     assert !race.getStages().contains(stage) : "Stage not
262     removed from race!";
263
264     Boolean foundId = false;
265     for (Integer stgId : stageIdsToStages.keySet()) {
266         if (stgId == stageId) {
267             stageIdsToStages.remove(stgId);
268             assert stageIdsToStages.get(stgId) == null : "
268             Stage not removed!";
269             foundId = true;
270             break;
271         }
272     }
273     if (!foundId) {
274         throw new IDNotRecognisedException("Stage ID " +
274         stageId + " is not recognised!");
275     }
276 }
277
278 @Override
279 public int addCategorizedClimbToStage(int stageId,
280     Double location, SegmentType type,
280     Double averageGradient, Double length)
281     throws IDNotRecognisedException,
281     InvalidLocationException, InvalidStageStateException,
282     InvalidStageTypeException {
283     Stage stage = stageIdsToStages.get(stageId);
284     // Does the stage exist?
285     if (stage == null) {
286         throw new IDNotRecognisedException("Stage ID " +
286         stageId + " is not recognised!");
287     }
288     // Is the location valid?
289     if ((location >= stage.getLength()) || (location <= 0
290 )) {
290         throw new InvalidLocationException("Invalid
290         location " + location + "!");
291     }
292     // Is the stage state "under development"?
293     if (!stage.getUnderDevelopment()) {
```

```
294     throw new InvalidStageStateException("Stage is
295         waiting for results!");
296     }
297     // Are you trying to add a sprint using the wrong
298     // method?
299     if (type == SegmentType.SPRINT) {
300         throw new InvalidStageTypeException("Cannot add
301             sprint to stage using this method!");
302     }
303
304     // Is the length valid?
305     if (length <= 0) {
306         throw new InvalidLocationException("Invalid length
307             entered! Enter one >= 0.");
308     }
309     if ((location + length) > stage.getLength()) {
310         throw new InvalidLocationException("Invalid length
311             exceeds length of stage!");
312     }
313
314     // Creates new categorised climb and adds it to the
315     // stage.
316     Segment categorisedClimb = new CategorisedClimb(
317         stageId, type, averageGradient,
318         length, location);
319     int ccId = categorisedClimb.getId();
320     segmentIdsToSegments.put(ccId, categorisedClimb);
321     assert segmentIdsToSegments.get(ccId) != null : "
322         Categorised climb not added to hashmap!";
323     stage.addSegment(categorisedClimb);
324
325     return categorisedClimb.getId();
326 }
```

```
323     @Override
324     public int addIntermediateSprintToStage(int stageId,
325         double location)
326         throws IDNotRecognisedException,
327             InvalidLocationException, InvalidStageStateException,
328             InvalidStageTypeException {
```

```

327     Stage stage = stageIdsToStages.get(stageId);
328     // Does the stage exist?
329     if (stage == null) {
330         throw new IDNotRecognisedException("Stage ID " +
331             stageId + " is not recognised!");
331     }
332     // Is the location valid?
333     if ((location >= stage.getLength()) || (location <= 0
334 )) {
334         throw new InvalidLocationException("Invalid
335 location " + location + " !");
335     }
336     // Is the stage type valid?
337     if (stage.getStageType() == StageType.TT) {
338         throw new InvalidStageTypeException("Cannot add
339 segments to a time trial stage.");
340     }
341     // Is the stage state "under development"?
342     if (!stage.getUnderDevelopment()) {
343         throw new InvalidStageStateException("Stage is
344 waiting for results!");
345     }
346     // Creates a new intermediate sprint and adds it to
347     // the stage.
348     Segment intermediateSprint = new Segment(stageId,
349         SegmentType.SPRINT, location);
350     int intSprintId = intermediateSprint.getId();
351     segmentIdsToSegments.put(intSprintId,
352         intermediateSprint);
353     assert segmentIdsToSegments.get(intSprintId) != null
354         : "Intermediate sprint not added to"
355         + "hashmap!";
356     stage.addSegment(intermediateSprint);
357     return intermediateSprint.getId();
358 }
359
360 @Override
361 public void removeSegment(int segmentId) throws
362     IDNotRecognisedException,
363     InvalidStageStateException {
364     // Does the segment exist?
365     if (segmentIdsToSegments.get(segmentId) == null) {

```

```

361         throw new IDNotRecognisedException("Segment ID " +
segmentId + " is not recognised!");
362     }
363     Segment segment = segmentIdsToSegments.get(segmentId
);
364     int segmentStageId = segment.getStageId();
365     // Can I do this (the stage has to be under
development).
366     // Find the stage.
367     for (Map.Entry<Integer, Stage> stageEntry :
stageIdsToStages.entrySet()) {
368         Stage stage = stageEntry.getValue();
369         if (stage.getId() == segmentStageId) {
370             // Remove segment results
371             for (RiderSegmentResult result : segment.
getResults()) {
372                 segment.getResults().remove(result);
373                 assert !segment.getResults().contains(result
) : "Result not removed from segment!";
374             }
375             // This is the right stage, now check we can
delete segments from it.
376             if (stage.getUnderDevelopment()) {
377                 segmentIdsToSegments.remove(segmentId);
378                 assert segmentIdsToSegments.get(segmentId) ==
null : "Segment not removed from hashmap!";
379             } else {
380                 throw new InvalidStageStateException("Stage "
+ stage.getId()
+ " is not under development!");
381             }
382         }
383     }
384 }
385 }
386
387 @Override
388 public void concludeStagePreparation(int stageId)
throws IDNotRecognisedException,
389     InvalidStageStateException {
390     // Does the stage exist?
391     if (stageIdsToStages.get(stageId) == null) {
392         throw new IDNotRecognisedException("Stage ID " +
stageId + " is not recognised!");
393     }

```

```

394     // Is it already waiting for results?
395     Stage stage = stageIdsToStages.get(stageId);
396     if (!stage.getUnderDevelopment()) {
397         throw new InvalidStageStateException("Stage " +
398             stageId
399             + " is already waiting for results!");
400     } else {
401         stage.setUnderDevelopment(false);
402     }
403     assert stage.getUnderDevelopment() == false : "Stage
404 preparations not concluded!";
405 }
406
407 @Override
408 public int[] getStageSegments(int stageId) throws
409 IDNotRecognisedException {
410     Stage stage = stageIdsToStages.get(stageId);
411     if (stage == null) {
412         throw new IDNotRecognisedException("Stage ID " +
413             stageId + " is not recognised!");
414     }
415
416     // Iterates through the segments in the stage and
417     // gets their IDs.
418     ArrayList<Segment> segmentsInStage = stage.
419     getSegmentsInStage();
420     int[] segmentIdsInStage = new int[segmentsInStage.
421     size()];
422     Collections.sort(segmentsInStage);
423     int i = 0;
424     for (Segment segment : segmentsInStage) {
425         segmentIdsInStage[i] = segment.getId();
426         i++;
427     }
428     assert segmentsInStage.size() == segmentIdsInStage.
429     length : "Segment IDs not gathered"
430         + "correctly!";
431     return segmentIdsInStage;
432 }
433
434 @Override
435 public int createTeam(String name, String description)
436     throws IllegalNameException,
437     InvalidNameException {

```

```

429      // Trim leading and trailing whitespace and check if
        the name is blank, is more than 30
430      // characters in length, or is null.
431      name = name.trim();
432      if (name == null || name == "" || name.length() > 30
    ) {
433          throw new InvalidNameException("Invalid name of a
        team!");
434      }
435      for (Team team : teamIdsToTeams.values()) {
436          if (team.getName() == name) {
437              throw new IllegalNameException("Team name already
        in use!");
438          }
439      }
440      // Creates a new team and then stores it in a hashmap
        for easier access.
441      Team newTeam = new Team(name, description);
442      int newTeamId = newTeam.getId();
443      teamIdsToTeams.put(newTeamId, newTeam);
444      assert teamIdsToTeams.get(newTeamId) != null : "Team
        not added to hashmap!";
445      return newTeam.getId();
446  }
447
448  @Override
449  public void removeTeam(int teamId) throws
        IDNotRecognisedException {
450      Team team = teamIdsToTeams.get(teamId);
451      if (team == null) {
452          throw new IDNotRecognisedException("Team ID " +
        teamId + " is not recognised!");
453      }
454      teamIdsToTeams.remove(teamId);
455      assert teamIdsToTeams.get(teamId) == null : "Team not
        removed from hashmap!";
456  }
457
458  @Override
459  public int[] getTeams() {
460      // Iterates through the IDs of teams from the hashmap
        and returns it as an integer array.
461      Set<Integer> teamIdsSet = teamIdsToTeams.keySet();
462      int[] teamIds = new int[teamIdsSet.size()];

```

```
463     int index = 0;
464     for (Integer i : teamIdsSet) {
465         teamIds[index++] = i;
466     }
467     assert teamIdsToTeams.size() == teamIds.length : "
468         Team IDs not gathered correctly!";
469
470     return teamIds;
471 }
472
473 @Override
474 public int[] getTeamRiders(int teamId) throws
475     IDNotRecognisedException {
476     ArrayList<Integer> riderIdArrayList = new ArrayList
477     <>();
478
479     // Check if the team exists.
480     if (teamIdsToTeams.get(teamId) == null) {
481         throw new IDNotRecognisedException("Team ID " +
482             teamId + " is not recognised!");
483     }
484
485     // Look for all riders in this team.
486     for (Map.Entry<Integer, Rider> idToRider :
487         riderIdsToRiders.entrySet()) {
488         Integer riderId = idToRider.getKey();
489         Rider rider = idToRider.getValue();
490         if (rider.getTeamId() == teamId) {
491             // Found one; add to list.
492             riderIdArrayList.add(riderId);
493         }
494     }
495
496     // Convert the ArrayList to int[].
497     return riderIdArrayList.stream().mapToInt(i -> i).
498         toArray();
499 }
500
501 @Override
502 public int createRider(int teamID, String name, int
503     yearOfBirth)
504     throws IDNotRecognisedException,
505     IllegalArgumentException {
506     name = name.trim();
```

```
499     // Check if the team exists.  
500     Team team = teamIdsToTeams.get(teamID);  
501     if (team == null) {  
502         throw new IDNotRecognisedException("Team ID " +  
503             teamID + " is not recognised!");  
504     }  
505     // Trim leading and trailing whitespace and check if  
506     // the name is blank, is more than 30  
507     // characters in length, or is null.  
508     if (name == null || name == "" || name.length() > 30)  
     ) {  
507         throw new IllegalArgumentException("Invalid name of  
      a team!");  
508     }  
509  
510     // Check YOB > 1900.  
511     if (yearOfBirth < 1900) {  
512         throw new IllegalArgumentException("Invalid year of  
      birth!");  
513     }  
514  
515     // Creates a new rider and adds it to the hashmap for  
     easier access.  
516     Rider newRider = new Rider(name, teamID, yearOfBirth)  
     );  
517     int newRiderId = newRider.getId();  
518     riderIdsToRiders.put(newRiderId, newRider);  
519     assert riderIdsToRiders.get(newRiderId) != null : "  
      Rider not added to hashmap!";  
520     team.addRider(newRider);  
521  
522     return newRider.getId();  
523 }  
524  
525 @Override  
526 public void removeRider(int riderId) throws  
IDNotRecognisedException {  
527     boolean hasBeenFound = false;  
528     for (Team team : teamIdsToTeams.values()) {  
529         HashMap<Integer, Rider> riders = team.  
getRiderIdsToRiders();  
530         try {  
531             if (riders.get(riderId) != null) {  
532                 hasBeenFound = true;
```

```
533         }
534         riders.remove(riderId);
535         assert riders.get(riderId) == null : "Rider not
536             successfully removed from their team!";
537     } catch (NullPointerException ex) { } // Errors
538     thrown for each team until ID is found.
539 }
540 if (!hasBeenCalled) {
541     throw new IDNotRecognisedException("Rider ID " +
542     riderId + " is not recognised!");
543 }
544
545     riderIdsToRiders.remove(riderId);
546     assert riderIdsToRiders.get(riderId) == null : "Rider
547         not removed from hashmap!";
548
549     // Results.
550     boolean foundRiderInRace = false;
551     for (StagedRace race : raceIdsToRaces.values()) {
552         ArrayList<RiderRaceResult> raceResults = race.
553         getResults();
554         // Destroy references to this rider in races.
555         for (RiderRaceResult raceResult : raceResults) {
556             if (raceResult.getRiderId() == riderId) {
557                 foundRiderInRace = true;
558                 raceResults.remove(raceResult);
559                 assert !raceResults.contains(raceResult) :
560                     "Rider's race result not removed!";
561
562             // Destroy references to this rider in stages.
563             for (Stage stage : race.getStages()) {
564                 ArrayList<RiderStageResult> stageResults =
565                 stage.getResults();
566
567                 for (RiderStageResult stageResult :
568                     stageResults) {
569                     if (stageResult.getRiderId() == riderId) {
570                         int stageResultsLength = stageResults.
571                         size();
572                         stageResults.remove(stageResult);
573                         assert stageResults.size() ==
574                             stageResultsLength - 1 :
575                             "Rider's stage result not"
576                             + "successfully removed!";
577 }
```

```

567
568          // Destroy references to this rider in
569          segments
570          for (Segment segment : stage.
571              getSegmentsInStage()) {
572              ArrayList<RiderSegmentResult>
573              segmentResults = segment.getResults();
574
575              for (RiderSegmentResult segmentResult
576                  : segmentResults) {
577                  if (segmentResult.getRiderId() ==
578                      riderId) {
579                      int segmentResultsLength =
580                      segmentResults.size();
581                      segmentResults.remove(segmentResult
582 );
583                      assert segmentResults.size() ==
584                      segmentResultsLength - 1 : "Rider's segment"
585                      + "result not successfully
586                      removed!";
587                  }
588
589      @Override
590      public void registerRiderResultsInStage(int stageId,
591          int riderId, LocalTime... checkpoints)
592          throws IDNotRecognisedException,
593          DuplicatedResultException, InvalidCheckpointsException,
594          InvalidStageStateException {
595          Stage stage = stageIdsToStages.get(stageId); // Gets
596          the stage.
597
598          // Error checking.
599          if (stage == null) {
600              throw new IDNotRecognisedException("Stage ID " +
601                  stageId + " is not recognised!");

```

```
598     }
599     Rider rider = riderIdsToRiders.get(riderId);
600     if (rider == null) {
601         throw new IDNotRecognisedException("Rider ID " +
602             riderId + " is not recognised!");
603     }
604     if (stage.getUnderDevelopment()) {
605         throw new InvalidStageStateException("The stage is
606             under development so can't add rider"
607             + "results!");
608     }
609     // Get the location data for each segment, order it,
610     sync it to times and store it.
611     // Check whether the result has already been
612     registered for this rider in this stage.
613     ArrayList<RiderStageResult> results = stage.
614     getResults();
615     if (!results.isEmpty()) {
616         for (RiderStageResult result : results) {
617             if (result.getRiderId() == riderId) {
618                 throw new DuplicatedResultException("Rider's
619                     result already registered in this stage!");
620             }
621         }
622     }
623     // Check the number of checkpoints in the stage
624     ArrayList<Segment> segments = stage.
625     getSegmentsInStage();
626     int checkpointLength = checkpoints.length;
627     if (!(segments == null)) {
628         if (!(checkpointLength == segments.size() + 2)) {
629             throw new InvalidCheckpointsException("Incorrect
630                 number of checkpoints in stage!");
631         }
632     } else if (checkpointLength != 2) {
633         throw new InvalidCheckpointsException("Incorrect
634             number of checkpoints in stage!");
635     }
636     // Check whether checkpoints are in chronological
637     order.
638     LocalTime previousTime = LocalTime.of(0, 0, 0);
```

```

632     for (LocalTime time : checkpoints) {
633         if (time.compareTo(previousTime) < 0) {
634             throw new InvalidCheckpointsException("Checkpoint
635 times are not in chronological order!");
636         }
637         previousTime = time;
638     }
639     int stageResultsLength = stage.getResults().size();
640     stage.addRiderResults(riderId, checkpoints);
641     assert stage.getResults().size() ==
642         stageResultsLength + 1 : "Rider not added to stage!";
643 }
644 @Override
645 public LocalTime[] getRiderResultsInStage(int stageId,
646 int riderId)
647     throws IDNotRecognisedException {
648     Stage stage = stageIdsToStages.get(stageId);
649     if (stage == null) {
650         throw new IDNotRecognisedException("Stage ID " +
651             stageId + " is not recognised!");
652     }
653     // If the rider doesn't exist / doesn't have a result
654     // in this stage.
655     RiderStageResult result = null;
656     for (RiderStageResult tmpResult : stage.getResults
657         ()) {
658         if (tmpResult.getRiderId() == riderId) {
659             result = tmpResult;
660             break;
661         }
662     }
663     if (result == null) {
664         return new LocalTime[0];
665     }
666
667     @Override
668     public LocalTime getRiderAdjustedElapsedTimeInStage(int
669         stageId, int riderId)

```

```
669     throws IDNotRecognisedException {
670     if (riderIdsToRiders.get(riderId) == null) {
671         throw new IDNotRecognisedException("Rider ID " +
672             riderId + " is not recognised!");
673     }
674     Stage stage = stageIdsToStages.get(stageId);
675     if (stage == null) {
676         throw new IDNotRecognisedException("Stage ID " +
677             stageId + " is not recognised!");
678     }
679     // Populate the RiderStageResults objects with their
680     // adjusted elapsed times.
681     stage.generateAdjustedFinishTimes();
682
683     for (RiderStageResult result : stage.getResults()) {
684         if (result.getRiderId() == riderId) {
685             return result.getAdjustedFinishTime();
686         }
687     }
688     // Returns null if no result for the rider is found.
689     return null;
690 }
691
692 @Override
693 public void deleteRiderResultsInStage(int stageId, int
694 riderId) throws IDNotRecognisedException {
695     Stage stage = stageIdsToStages.get(stageId);
696     // Does stage exist?
697     if (stage == null) {
698         throw new IDNotRecognisedException("Stage ID " +
699             stageId + " is not recognised!");
700     }
701     // Does rider exist?
702     if (riderIdsToRiders.get(riderId) == null) {
703         throw new IDNotRecognisedException("Rider ID " +
704             riderId + " is not recognised!");
705     }
706     int stageResultsLength = stage.getResults().size();
707     stage.removeResultByRiderId(riderId);
708     assert stage.getResults().size() ==
```

```
706 stageResultsLength - 1 : "Rider not removed from stage!";  
707 }  
708  
709 @Override  
710 public int[] getRidersRankInStage(int stageId) throws  
IDNotRecognisedException {  
711 // Does stage exist?  
712 Stage stage = stageIdsToStages.get(stageId);  
713 if (stage == null) {  
714 throw new IDNotRecognisedException("Stage ID " +  
stageId + " is not recognised!");  
715 }  
716  
717 ArrayList<Integer> riderIdsList = new ArrayList<>();  
718 ArrayList<RiderStageResult> results = stage.  
getResults();  
719  
720 // Sort results by finish time.  
721 Collections.sort(results);  
722  
723 // Fill the IDs list with the rider ids which are now  
in rank order.  
724 for (RiderStageResult result : results) {  
725 riderIdsList.add(result.getRiderId());  
726 }  
727  
728 // Convert the ArrayList to int[].  
729 return riderIdsList.stream().mapToInt(i -> i).toArray()  
();  
730 }  
731  
732 @Override  
733 public LocalTime[] getRankedAdjustedElapsedTimesInStage  
(int stageId)  
throws IDNotRecognisedException {  
734 Stage stage = stageIdsToStages.get(stageId);  
735 if (stage == null) { // Checks race is in cycling  
portal.  
736 throw new IDNotRecognisedException("Stage ID " +  
stageId + " is not recognised!");  
737 }  
738  
739 // Creates the adjusted results and adds the  
attribute to the result object.
```

```

741     // Also orders results by finishing time (rank).
742     stage.generateAdjustedFinishTimes();
743
744     // Iterates through stage results and gets their
745     // adjusted time in rank order.
746     ArrayList<RiderStageResult> results = stage.
747     getResults();
748     LocalTime[] rankAdjustedElapsedTimesInStage = new
749     LocalTime[results.size()];
750     int i = 0;
751     for (RiderStageResult result : stage.getResults()) {
752         rankAdjustedElapsedTimesInStage[i] = result.
753         getAdjustedFinishTime();
754         i++;
755     }
756     return rankAdjustedElapsedTimesInStage;
757 }
758
759 @Override
760 public int[] getRidersPointsInStage(int stageId) throws
761 IDNotRecognisedException {
762     Stage stage = stageIdsToStages.get(stageId);
763     if (stage == null) { // Checks race is in cycling
764         portal.
765         throw new IDNotRecognisedException("Stage ID " +
766             stageId + " is not recognised!");
767     }
768
769     // Generates the points in the stage and returns
770     // rider IDs ordered by points.
771     return stage.generatePointsInStage(false);
772 }
773
774 @Override
775 public int[] getRidersMountainPointsInStage(int stageId
776 ) throws IDNotRecognisedException {
777     Stage stage = stageIdsToStages.get(stageId);
778     if (stage == null) { // Checks race is in cycling
779         portal.
780         throw new IDNotRecognisedException("Stage ID " +
781             stageId + " is not recognised!");
782     }
783     if ((stage.getStageType() == StageType.TT) || (stage.
784         getSegmentsInStage().size() == 0)) {

```

```
773     // No mountain points (no segments).
774     return new int[0];
775 }
776
777 // Check if there even are any climbs.
778 boolean climbFound = false;
779 for (Segment segment : stage.getSegmentsInStage()) {
780     if (segment.getSegmentType() != SegmentType.SPRINT
781 ) {
782         climbFound = true;
783     }
784     // No climbs so there will be no results.
785     if (!climbFound) {
786         return new int[0];
787     }
788     // Generates the mountain points in the stage and
789     // returns rider IDs ordered by mountain
790     // points.
791     return stage.generatePointsInStage(true);
792 }
793
794 @Override
795 public void eraseCyclingPortal() {
796     // Reset all internal counters.
797     Rider.resetIdCounter();
798     Team.resetIdCounter();
799     Stage.resetIdCounter();
800     StagedRace.resetIdCounter();
801     Segment.resetIdCounter();
802     CategorisedClimb.resetIdCounter();
803     // Erase all references and get them garbage
804     // collected.
805     this.raceIdsToRaces = new HashMap<>();
806     this.stageIdsToStages = new HashMap<>();
807     this.segmentIdsToSegments = new HashMap<>();
808     this.teamIdsToTeams = new HashMap<>();
809     this.riderIdsToRiders = new HashMap<>();
810 }
811
812 @Override
813 public void saveCyclingPortal(String filename) throws
814 IOException {
815     FileOutputStream fileOutputStream = new
```

```

812 FileOutputStream(filename);
813     ObjectOutputStream objectOutputStream = new
814         ObjectOutputStream(fileOutputStream);
815     objectOutputStream.writeObject(this);
816     objectOutputStream.flush();
817     objectOutputStream.close();
818 }
819
820 @Override
821 public void loadCyclingPortal(String filename) throws
822     IOException, ClassNotFoundException {
823     FileInputStream fileInputStream = new FileInputStream
824     (filename);
825     ObjectInputStream objectInputStream = new
826     ObjectInputStream(fileInputStream);
827     CyclingPortal tmp = (CyclingPortal) objectInputStream
828     .readObject();
829     this.raceIdsToRaces = tmp.raceIdsToRaces;
830     this.stageIdsToStages = tmp.stageIdsToStages;
831     this.segmentIdsToSegments = tmp.segmentIdsToSegments;
832     this.teamIdsToTeams = tmp.teamIdsToTeams;
833     this.riderIdsToRiders = tmp.riderIdsToRiders;
834 }
835
836 @Override
837 public void removeRaceByName(String name) throws
838     NameNotRecognisedException {
839     boolean foundRace = false;
840     // Find the named race's ID.
841     for (Map.Entry<Integer, StagedRace> stagedRaceEntry
842         : raceIdsToRaces.entrySet()) {
843         int raceId = stagedRaceEntry.getKey();
844         StagedRace stagedRace = stagedRaceEntry.getValue();
845
846         // Remove race result objects.
847         for (RiderRaceResult result : stagedRace.getResults
848             ()) {
849             stagedRace.getResults().remove(result);
850             assert !stagedRace.getResults().contains(result
851             ) : "Race result not removed!";
852         }
853
854         // Remove stages of the race.
855         for (Stage stage : stagedRace.getStages()) {

```

```

847         stagedRace.getStages().remove(stage);
848         assert !stagedRace.getStages().contains(stage) :
849             "Stage not removed!";
850         try {
851             removeStageById(stage.getId());
852         } catch (IDNotRecognisedException ex) {
853             System.out.println("Stage waiting for results
854                 and cannot be deleted!");
855         }
856     // Remove the race object.
857     if (stagedRace.getName().equals(name)) {
858         // Found it.
859         foundRace = true;
860         raceIdsToRaces.remove(raceId);
861         assert raceIdsToRaces.get(raceId) == null : "Race
862             not removed from hashmap!";
863     }
864     // Gone through the whole hashmap and still haven't
865     // found it.
866     if (!foundRace) {
867         throw new NameNotRecognisedException("No race found
868             with name '" + name + "'!");
869     }
870     @Override
871     public LocalTime[] getGeneralClassificationTimesInRace(
872         int raceId)
873         throws IDNotRecognisedException {
874     StagedRace race = raceIdsToRaces.get(raceId);
875     if (race == null) {
876         throw new IDNotRecognisedException("Race ID " +
877             raceId + " is not recognised!");
878     }
879     // Calculates the finish times for riders in the race
880     .
881     race.generateRidersRaceAdjustedElapsedTimes();
882     ArrayList<RiderRaceResult> raceResults = race.
883     getResults();
884     // Iterates through race results and adds finish time

```

```
881 to an array.
882     LocalTime[] finishTimes = new LocalTime[raceResults.
883     size()];
884     int i = 0;
885     for (RiderRaceResult result : raceResults) {
886         finishTimes[i] = result.getElapsedTime();
887         i++;
888     }
889     assert raceResults.size() == finishTimes.length : "
890     Finish times gathered incorrectly!";
891     return finishTimes;
892 }
893
894 @Override
895 public int[] getRidersPointsInRace(int raceId) throws
896 IDNotRecognisedException {
897     StagedRace race = raceIdsToRaces.get(raceId);
898     if (race == null) { // Error check.
899         throw new IDNotRecognisedException("Race ID " +
900             raceId + " is not recognised!");
901     }
902     // Calculate points and returns an array of points
903     // sorted by descending order.
904     return race.generateRidersPointsInRace(false);
905 }
906
907 @Override
908 public int[] getRidersMountainPointsInRace(int raceId)
909 throws IDNotRecognisedException {
910     StagedRace race = raceIdsToRaces.get(raceId);
911     if (race == null) { // Error check.
912         throw new IDNotRecognisedException("Race ID " +
913             raceId + " is not recognised!");
914     }
915     // Calculate mountain points and returns an array of
916     // mountain points sorted by descending order.
917     return race.generateRidersPointsInRace(true);
918 }
919
920 @Override
921 public int[] getRidersGeneralClassificationRank(int
922 raceId) throws IDNotRecognisedException {
923     StagedRace race = raceIdsToRaces.get(raceId); // Gets
924     the race.
```

```
915     if (race == null) { // Error check.
916         throw new IDNotRecognisedException("Race ID " +
917             raceId + " is not recognised!");
918         race.generateRidersRaceAdjustedElapsedTimes(); // 
919             Returns race results also stores in race result list.
920             ArrayList<RiderRaceResult> raceResults = race.
921                 getResults(); // Gets race result arraylist.
922 
923             // Iterates through race results to fill an array
924             with rider IDs sorted by their rank.
925             int raceResultsSize = raceResults.size();
926             int[] riderIdsOrderedByRank = new int[raceResultsSize
927 ];
928             assert raceResultsSize == riderIdsOrderedByRank.
929                 length : "Integer array created incorrectly!";
930             for (int i = 0; i < raceResultsSize; i++) {
931                 riderIdsOrderedByRank[i] = raceResults.get(i).
932                     getRiderId();
933 
934             return riderIdsOrderedByRank;
935         }
936 
937         @Override
938         public int[] getRidersPointClassificationRank(int
939             raceId) throws IDNotRecognisedException {
940             StagedRace race = raceIdsToRaces.get(raceId);
941             if (race == null) {
942                 throw new IDNotRecognisedException("Race ID " +
943                     raceId + " is not recognised!");
944             }
945             // Calculate points in race.
946             race.generateRidersPointsInRace(false);
947 
948             return race.getRiderIdsOrderedByPoints(false);
949         }
950 
951         @Override
952         public int[] getRidersMountainPointClassificationRank(
953             int raceId)
954             throws IDNotRecognisedException {
955             StagedRace race = raceIdsToRaces.get(raceId);
956             if (race == null) {
```

```
949         throw new IDNotRecognisedException("Race ID " +
raceId + " is not recognised!");
950     }
951     // Calculate mountain points in race.
952     race.generateRidersPointsInRace(true);
953
954
955     return race.getRiderIdsOrderedByPoints(true);
956 }
957 }
```