File - Team.java

```java
package src.cycling;

import java.io.Serializable;
import java.util.HashMap;

/**
 * Teams are made up of riders and can be placed into races.
 *
 * @author Adam Kaizra, Sam Barker
 * @version 1.0
 */
public class Team implements Serializable {

  private static int latestId = 0; // Enumerates to get unique id, with 2^32 possible ids.
  private String name;
  private String description;
  private int id;
  // Creates a hash map between the team's rider's Ids and the rider objects.
  private HashMap<Integer, Rider> riderIdsToRiders = new HashMap<Integer, Rider>();

  /**
   * Constructor.
   *
   * @param name
   */
  public Team(String name, String description) {
    this.name = name;
    this.description = description;
    this.id = latestId++;
  }

  /**
   * Reset the internal ID counter.
   */
  public static void resetIdCounter() {
    latestId = 0;
  }

  /**
   * @return The name of the team.
```

```java
    */
   public String getName() {
     return this.name;
   }

   /**
    * @param newTeamName
    */
   public void setName(String newTeamName) {
     this.name = newTeamName;
   }

   /**
    * @return The description of the team.
    */
   public String getDescription() {
     return this.description;
   }

   /**
    * @return The ID of the team.
    */
   public int getId() {
     return this.id;
   }

   /**
    * @return A hash map between the team's rider's IDs and their corresponding rider objects.
    */
   public HashMap<Integer, Rider> getRiderIdsToRiders() {
     return riderIdsToRiders;
   }

   /**
    * @param rider A rider object to be added to the team's internal hashmap of rider ID's to riders.
    */
   public void addRider(Rider rider) {
     this.riderIdsToRiders.put(rider.getId(), rider);
   }
 }
```

File - SumLocalTimes.java

```java
package src.cycling;

import java.time.LocalTime;

/**
 * A helper object whose static methods deal with the addition of LocalTimes
 * and conversion between seconds (doubles) and LocalTime objects, of times.
 *
 * @author Sam Barker, Adam Kaizra
 * @version 1.0
 */
public class SumLocalTimes {

  /**
   * Convert a LocalTime object to the number of seconds after midnight.
   *
   * @param time
   * @return The time after midnight in seconds that a LocalTime object represents.
   */
  public static double localTimeToSeconds(LocalTime time) {
    if (time == null) {
      return 0;
    }
    return (time.getHour() * 3600)
        + (time.getMinute() * 60)
        + (time.getSecond());
  }

  /**
   * Create a LocalTime object from the number of seconds after midnight.
   *
   * @param timeSeconds
   * @return The LocalTime object representing the time given.
   */
  public static LocalTime secondsToLocalTime(double timeSeconds) {
    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      // Convert the LocalTimes to seconds, sum them all up
    , then convert back to LocalTime.
50      double finalTimeSeconds = 0;
51      for (LocalTime time : times) {
52        double timeSeconds = localTimeToSeconds(time);
53        finalTimeSeconds += timeSeconds;
54      }
55
56      return secondsToLocalTime(finalTimeSeconds);
57    }
58
59    /**
60     * Subtract two LocalTime variables.
61     *
62     * @param t1 The time to have t2 taken away from.
63     * @param t2 The time to take away from t1
64     * @return t1 minus t2
65     */
66    public static LocalTime subtractLocalTimes(LocalTime t1
    , LocalTime t2) {
67      LocalTime finalTime;
68
69      // Convert times to seconds
70      double t1s = localTimeToSeconds(t1);
71      double t2s = localTimeToSeconds(t2);
72      // Subtract
73      double t3s = t1s - t2s;
74
75      assert t3s >= 0 : "LocalTime cannot represent negative
     values!";
76
```

File - SumLocalTimes.java

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

File - StagedRace.java

```java
package src.cycling;

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

/**
 * Represents a grand tour staged race.
 *
 * @author Sam Barker, Adam Kaizra
 * @version 1.0
 */
public class StagedRace implements Serializable {

  private static int latestId = 0;
  private final String name;
  private final String description;
  private final int raceId;
  private LocalTime[] datesOfCompetitions;
  private final ArrayList<Stage> stages = new ArrayList
<>();
  private final ArrayList<RiderRaceResult> raceResults =
new ArrayList<>();

  /**
   * Constructor.
   *
   * @param name
   * @param description
   */
  public StagedRace(String name, String description) {
    this.name = name;
    this.description = description;
    this.raceId = latestId++;
  }

  /**
   * Reset the internal ID counter.
   */
  public static void resetIdCounter() {
    latestId = 0;
  }

```

```
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
   race.
73     */
74    public void addStage(Stage stage) {
75      this.stages.add(stage);
76    }
77
78    /**
79     * Sums all the riders stage results finish times and
   stores it as a race finish time in each
80     * rider's race result object.
81     */
82    public void generateRidersRaceAdjustedElapsedTimes() {
83      int sizeOfArrayOfTotalTimes;
84      if (this.getStages().size() == 0) {
```

```
 85         return;
 86       } else {
 87         sizeOfArrayOfTotalTimes = this.getStages().get(0).
    getResults().size();
 88       }
 89
 90       LocalTime[] arrayOfTotalTimes = new LocalTime[
    sizeOfArrayOfTotalTimes];
 91       for (Stage stage : this.getStages()) { // Iterates
    through all stages in a race.
 92         int i = 0;
 93         stage.generateAdjustedFinishTimes();
 94
 95         for (RiderStageResult riderStageResult : stage.
    getResults()) { // Iterates through all results
 96           // in a stage.
 97           int riderId = riderStageResult.getRiderId();
 98           boolean riderFound = false;
 99           for (RiderRaceResult raceResult : this.
    raceResults) { // Iterates through race results.
100             if (raceResult.getRiderId() == riderId) { // If
     the race result and stage result refer to
101               // the same rider.
102               arrayOfTotalTimes[i] = SumLocalTimes.
    addLocalTimes(arrayOfTotalTimes[i],
103                   riderStageResult.getAdjustedElapsedTime
    ());
104               // Sums race results finish time with new
    stages finish time.
105               riderFound = true;
106               break;
107             }
108           }
109           if (!riderFound) { // If no race result for rider
     one is made.
110             RiderRaceResult raceResult = new
    RiderRaceResult(riderStageResult.getRiderId(),
111                 this.raceId);
112             arrayOfTotalTimes[i] = riderStageResult.
    getAdjustedElapsedTime();
113             int raceResultsSize = this.raceResults.size();
114             this.raceResults.add(raceResult);
115             assert this.raceResults.size() ==
    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.
    getRiderId()) {
151               // Points are set or added to the race
    total array.
152               if (points[i] == 0) {
153                 points[i] = stageResult.getMountainPoints
    ();
154               } else {
155                 int stagePoints = stageResult.
    getMountainPoints();
156                 assert stagePoints >= 0 : "Stage mountain
     points can not be negative!";
157                 points[i] += stagePoints;
158               }
159             }
160             i++;
161           }
162         } else { // Point classification.
163           for (RiderRaceResult raceResult : this.
    getResults()) {
164             if (stageResult.getRiderId() == raceResult.
    getRiderId()) {
165               // Points are sent to an array that stores
    each rider's total points in the race.
166               if (points[i] == 0) {
167                 points[i] = stageResult.getPoints();
168               } else {
169                 int stagePoints = stageResult.getPoints
    ();
170                 assert stagePoints >= 0 : "Stage points
    can not be negative!";
171                 points[i] += stagePoints;
172               }
173             }
174             i++;
175           }
176         }
177       }
178       // Iterates through race results and sets each
    rider's total points for the race.
179       int i = 0;
180       for (RiderRaceResult result : this.getResults()) {
181         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
    points in the given classification.
221     */
222   public int[] getRiderIdsOrderedByPoints(boolean
    isMountain) {
223     ArrayList<RiderRaceResult> riderRaceResults = this.
    getResults();
224     // Sorts race results by their points in the given
    classification.
225     if (isMountain) {
226       Collections.sort(riderRaceResults, RiderRaceResult
    ::compareByMountainPoints);
227     } else {
228       Collections.sort(riderRaceResults, RiderRaceResult
    ::compareByPoints);
229     }
230
231     // Gets the rider ID's in order of points and adds
    them to an array.
232     int[] riderIdsByPoints = new int[riderRaceResults.
    size()];
233     int i = 0;
234     for (RiderRaceResult result : riderRaceResults) {
235       riderIdsByPoints[i] = result.getRiderId();
236       i++;
237     }
238     Collections.sort(raceResults); // Sorts race results
    back to its order by finishing times.
239     assert riderRaceResults.size() == riderIdsByPoints.
    length : "Gathered rider IDs incorrectly!";
240     return riderIdsByPoints;
241   }
242 }
243
```

File - Stage.java

```java
package src.cycling;

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

/**
 * Represents a stage in the race and contains all the rider's results for the stage. If a stage is
 * under development results can not be registered but segments can be added.
 *
 * @author Adam Kaizra, Sam Barker
 * @version 1.0
 */
public class Stage implements Serializable, Comparable<Stage> {

  private static int latestId = 0; // Enumerates to get unique id, with 2^32 possible ids.
  private final Integer raceId;
  private String name;
  private String description;
  private Double length;
  private LocalDateTime startTime;
  private StageType stageType;
  private final Integer id;
  private final ArrayList<RiderStageResult> results = new ArrayList<>();
  private final ArrayList<Segment> segmentsInStage = new ArrayList<>();
  private Boolean underDevelopment = true; // Either under development(T) or waiting results(F).

  /**
   * Constructor.
   *
   * @param raceId
   * @param stageName
   * @param description
```

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

```java
  /**
   * @return The name of the stage.
   */
  public String getName() {
    return name;
  }

  /**
   * @param newName
   */
  public void setName(String newName) {
    this.name = newName;
  }

  /**
   * @return The description of the stage.
   */
  public String getDescription() {
    return description;
  }

  /**
   * @param newDescription
   */
  public void setDescription(String newDescription) {
    this.description = newDescription;
  }

  /**
   * @return The length of the stage measured in
  kilometers.
   */
  public Double getLength() {
    return length;
  }

  /**
   * @param newLength
   */
  public void setLength(Double newLength) {
    this.length = newLength;
  }

  /**
```

```
122     * @return The local date and time of the start of the
    stage.
123     */
124   public LocalDateTime getStartTime() {
125     return startTime;
126   }
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
    stage.
151    */
152   public ArrayList<Segment> getSegmentsInStage() {
153     return segmentsInStage;
154   }
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
    development (true) or waiting results (false).
168    */
169   public Boolean getUnderDevelopment() {
170     return underDevelopment;
171   }
172
173   /**
174    * If true the state of the stage is under development
    . If false the state of the stage is
175    * awaiting results.
176    *
177    * @param newState
178    */
179   public void setUnderDevelopment(Boolean newState) {
180     this.underDevelopment = newState;
181   }
182
183   /**
184    * @return An array list containing all the
    RiderStageResult objects relating to this stage.
185    */
186   public ArrayList<RiderStageResult> getResults() {
187     return this.results;
188   }
189
190   /**
191    * Removes the results of a rider from a stage.
192    *
193    * @param riderId The rider ID of the rider to be
    removed.
194    * @throws IDNotRecognisedException
195    */
196   public void removeResultByRiderId(int riderId) throws
    IDNotRecognisedException {
197     boolean riderIdFound = false;
198     RiderStageResult finalResult = null; // Empty value
    to be assigned if the rider is found.
199     for (RiderStageResult result : results) { // Iterates
     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
    it can't be removed.
208       throw new IDNotRecognisedException("Rider ID " +
    riderId + " not found in stage!");
209     }
210     results.remove(finalResult); // Removes the result
    from the stage results array list.
211     assert !results.contains(finalResult) : "Rider's
    result not removed!";
212   }
213
214   /**
215    * Creates a new RiderStageResult object and adds it to
     an array list of rider results in this
216    * stage.
217    *
218    * @param riderId
219    * @param times The times the rider passed each
    checkpoint in chronological order.
220    */
221   public void addRiderResults(Integer riderId, LocalTime
    [] times) {
222     RiderStageResult result = new RiderStageResult(
    riderId, this.id, times); // Creates new result.
223     results.add(result); // The new result object is
    added to an array list of all result objects in
224     // the stage.
225     assert (results.contains(result)) : "Result object
    not created!";
226     Collections.sort(results); // Sorts the results by
    their finishing times.
227   }
228
229   /**
230    * Searches all the stages' riders' results to find
    pelotons (where riders are less than one
231    * 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
     to zero.
238       LocalTime previousTime = LocalTime.of(0, 0, 0, 0);
239       LocalTime pelotonLeader = LocalTime.of(0, 0, 0);
240 
241       for (RiderStageResult result : this.results) { //
    Iterates through results of the stage.
242         LocalTime currentTime = result.getFinishTime();
    // The time of this rider to compare
243         // against.
244         double currentTimeSeconds =
245             (currentTime.getHour() * 3600) + (currentTime
    .getMinute() * 60) +
246                 currentTime.getSecond();
247         double previousTimeSeconds = (previousTime.
    getHour() * 3600) + (previousTime.getMinute() *
248             60 + previousTime.getSecond());
249         // Converts LocalTime into double to calculate
    whether there is a 1 second or less gap
250         // between riders.
251         if (((currentTimeSeconds - previousTimeSeconds
    ) <= 1.0) && (previousTimeSeconds != 0)) {
252           result.setAdjustedFinishTime(pelotonLeader);
    // If there is a 1 second or less gap the
253           // adjusted time is set to the peloton leader.
254         } else {
255           pelotonLeader = currentTime; // If the rider is
     not in a peloton then they could be the
256           // next peloton's leader.
257           result.setAdjustedFinishTime(currentTime); //
    Adjusted time is set to their finish time.
258         }
259         previousTime = currentTime; // A Peloton can
    consist of multiple riders so the one second
260         // gap accounts for between riders.
261 
262         result.setAdjustedElapsedTime(SumLocalTimes.
    subtractLocalTimes(
```

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

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

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

```
359     // Sum of points for each rider for the specified
    race.
360
361     // For the points classification, add on the points
    from finish times which mountain doesn't
362     // include.
363     if (!isMountain) {
364       int pointsIndex = 0;
365       this.generateAdjustedFinishTimes(); // Sorts times
    in ascending order.
366
367       for (RiderStageResult result : this.getResults
    ()) { // Iterate through rider.
368         int riderId = result.getRiderId();
369         int points;
370         if (pointsIndex < 15) { // Only first 15 riders
    are awarded points.
371           points = pointsConversion.get(this.getStageType
    ())[pointsIndex]; // Points for stage type.
372           assert points > 0 : "Incorrect points assigned
    !";
373         } else {
374           points = 0;
375         }
376         riderIdsToPoints.put(riderId, points);
377         result.setPoints(points);
378         pointsIndex++;
379       }
380     }
381
382     // Add on points from segments.
383     ArrayList<Segment> segmentsInStage = this.
    getSegmentsInStage();
384     this.generateRiderSegmentResults(); // Creates
    segment objects.
385     for (Segment segment : segmentsInStage) {
386       // Cast for the appropriate Enum for mountain or
    points as a key for their points conversion
387       // table.
388       SegmentType currentSegmentType = segment.
    getSegmentType();
389       Enum mapKey;
390       if (isMountain) {
391         mapKey = currentSegmentType;
```

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

```
424                 if (stageResult.getRiderId() ==
    riderResult.getRiderId()) {
425                   stageResult.setPoints(points);
426                 }
427               }
428             } else {
429               for (RiderStageResult stageResult : this.
    getResults()) {
430                 if (stageResult.getRiderId() ==
    riderResult.getRiderId()) {
431                   stageResult.setMountainPoints(points);
432                 }
433               }
434             }
435           } else { // Else sum their points to their
    total points.
436             riderIdsToPoints.merge(riderId, points,
    Integer::sum);
437             if (!isMountain) {
438               for (RiderStageResult stageResult : this.
    getResults()) {
439                 if (stageResult.getRiderId() ==
    riderResult.getRiderId()) {
440                   stageResult.addPoints(points);
441                 }
442               }
443             } else {
444               for (RiderStageResult stageResult : this.
    getResults()) {
445                 if (stageResult.getRiderId() ==
    riderResult.getRiderId()) {
446                   stageResult.addMountainPoints(points);
447                 }
448               }
449             }
450           }
451           i++;
452         }
453       }
454     }
455     Collections.sort(this.results);
456     // Creates an int array of points ordered by rank
457     if (!(riderIdsToPoints.isEmpty())) {
458       int[] pointsOrderedByRank = new int[this.results.
```

File - Stage.java

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

File - Segment.java

```java
package src.cycling;

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

/**
 * Represents an intermediate sprint. CategorisedClimb is a subclass of this for categorised
 * climbs.
 *
 * @author Sam Barker, Adam Kaizrea
 * @version 1.0
 */
public class Segment implements Comparable<Segment>, Serializable {

  private static int latestId = 0;
  private final Integer stageId;
  private Double location;
  private SegmentType segmentType;
  private final Integer id;
  private ArrayList<RiderSegmentResult> results = new ArrayList<>();

  /**
   * Constructor.
   *
   * @param stageId The ID of the stage this sprint is part of.
   * @param segmentType The type of segment (SPRINT)
   * @param location The location of the segment in the stage.
   */
  public Segment(int stageId, SegmentType segmentType, Double location) {
    this.stageId = stageId;
    this.segmentType = segmentType;
    this.id = latestId++;
    this.location = location;
  }

  /**
   * Reset the internal ID counter.
   */
```

```java
  public static void resetIdCounter() {
    latestId = 0;
  }

  /**
   * @param segment The segment to compare the calling object to.
   * @return 1 if location of calling object > the passed in one. 0 if equal. -1 if less than.
   */
  public int compareTo(Segment segment) {
    assert (segment instanceof Segment) : "Comparing incorrect types!";
    return this.getLocation().compareTo(segment.getLocation());
  }

  /**
   * @return The segment type.
   */
  public SegmentType getSegmentType() {
    return segmentType;
  }

  /**
   * Set the segment type.
   *
   * @param segmentType The segment type to set this object's segmentType to.
   */
  public void setSegmentType(SegmentType segmentType) {
    this.segmentType = segmentType;
  }

  /**
   * @return The ID of the stage this segment is part of.
   */
  public int getStageId() {
    return this.stageId;
  }

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

File - RiderStageResult.java

```java
package src.cycling;

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

/**
 * Represents the result of a rider in a stage, holding their times in all segments of that stage,
 * their final time, their adjusted elapsed time, their points classification points and their
 * mountain classification points. Objects of this class are compared to each other by their finish
 * times, which functions the same as comparing them by their adjusted times as adjusted times
 * retain order.
 *
 * @author Adam Kaizra, Sam Barker
 */
public class RiderStageResult implements Comparable<RiderStageResult>, Serializable {

  private final int riderId;
  private final int stageId;
  private final LocalTime[] times; // Times for the rider's start, segments and finish times.
  private final LocalTime finishTime;
  private LocalTime adjustedFinishTime; // Adjusted to take the peloton leader's time if applicable.
  private int points; // Points classification points for this stage.
  private int mountainPoints; // Mountain classification points for this stage.
  private LocalTime elapsedTime;
  private LocalTime adjustedElapsedTime;

  /**
   * Constructor.
   *
   * @param riderId
   * @param stageId
   * @param times First input is starting time, final input is finishing time, all other inputs (if
   *              applicable) are segment times in chronological order.
```

File - RiderStageResult.java

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

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

```
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
    taken from.
112    */
113   public int getStageId() {
114     return stageId;
115   }
116
117   /**
118    * @return The times the rider passed each checkpoint.
119    */
120   public LocalTime[] getTimes() {
121     return times;
122   }
123
124   /**
125    * @return The rider's finish time.
126    */
127   public LocalTime getFinishTime() {
128     return finishTime;
129   }
130
131   /**
132    * @return The rider's finish time adjusted to the
    peloton leader's time if applicable.
133    */
134   public LocalTime getAdjustedFinishTime() {
135     return adjustedFinishTime;
136   }
137
138   /**
139    * @param newAdjustedFinishTime
140    */
141   public void setAdjustedFinishTime(LocalTime
    newAdjustedFinishTime) {
142     this.adjustedFinishTime = newAdjustedFinishTime;
```

```
143   }
144
145   public LocalTime getElapsedTime() {
146     return elapsedTime;
147   }
148
149   public void setElapsedTime(LocalTime elapsedTime) {
150     this.elapsedTime = elapsedTime;
151   }
152
153   public LocalTime getAdjustedElapsedTime() {
154     return adjustedElapsedTime;
155   }
156
157   public void setAdjustedElapsedTime(LocalTime
    adjustedElapsedTime) {
158     this.adjustedElapsedTime = adjustedElapsedTime;
159   }
160 }
161
```

File - RiderSegmentResult.java

```java
package src.cycling;

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

/**
 * Represents the result of one rider in one segment of a stage, including finish time,
 * and rank. Points are not included for individual segments, but created for stages and races,
 * and stored in RiderStageResult and RiderRaceResult respectively.
 *
 * @author Sam Barker, Adam Kaizra
 * @version 1.0
 */
public class RiderSegmentResult implements Comparable<RiderSegmentResult>, Serializable {

  private LocalTime time;
  private int riderId;
  private int rank;

  /**
   * @param time The finish time of the rider in the segment.
   * @param riderId The ID of the rider who achieved this result.
   */
  public RiderSegmentResult(LocalTime time, int riderId) {
    this.time = time;
    this.riderId = riderId;
  }

  /**
   * Compare two RiderSegmentResults by their finish time (for the purpose of sorting).
   *
   * @param result
   * @return 1 if calling object's finish time is greater than the parameter object's.
   * 0 if equal. -1 if less than.
   */
  public int compareTo(RiderSegmentResult result) {
```

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

```
  }

  /**
   * Set the ID of the rider who achieved this result.
   *
   * @param riderId
   */
  public void setRiderId(int riderId) {
    this.riderId = riderId;
  }
}

```

File - RiderRaceResult.java

```java
package src.cycling;

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

/**
 * Represents the result of one rider in one race, including points, mountain points, finish time,
 * and rank by finish time.
 *
 * @author Adam Kaizra, Sam Barker
 * @version 1.0
 */
public class RiderRaceResult implements Comparable<RiderRaceResult>, Serializable {

  private int riderId;
  private final int raceId;
  private int points = 0;
  private int mountainPoints = 0;
  private int rank;
  private LocalTime elapsedTime;

  private LocalTime finishTime; // The sum of all stage finish times (the GC time)

  /**
   * Constructor.
   *
   * @param riderId The ID of the rider that achieved this result.
   * @param raceId The ID of the race in which this result was achieved.
   */
  public RiderRaceResult(int riderId, int raceId) {
    this.riderId = riderId;
    this.raceId = raceId;
    this.points = 0;
    this.mountainPoints = 0;
  }

  /**
   * Compare two RiderRaceResults by their finish times (for the purpose of sorting).
```

File - RiderRaceResult.java

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

```
 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
    achieved.
125    */
126   public int getRaceId() {
127     return raceId;
128   }
129
130   /**
131    * @return The total finish time (sum of the finish
    times of all stages)
132    * of the rider in the race.
133    */
134   public LocalTime getFinishTime() {
135     return finishTime;
136   }
137
138   /**
139    * Set the total finish time (sum of the finish times
    of all stages)
140    * of the rider in the race.
141    *
142    * @param time
143    */
144   public void setFinishTime(LocalTime time) {
145     this.finishTime = time;
146   }
147
148   /**
149    * @return The total points (points classification
    points) the rider achieved in the race.
150    * Summed from across all stages.
```

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

File - Rider.java

```java
package src.cycling;

import java.io.Serializable;

/**
 * Each cyclist is treated as an object that can be assigned to a team and compete in competitions.
 * Their results are stored in separate objects that can be accessed via the rider's ID.
 *
 * @author Sam Barker, Adam Kaizra
 * @version 1.0
 */
public class Rider implements Serializable {

  private static int latestId = 0; // Enumerates to get unique id, with 2^32 possible ids.
  private String name;
  private final int id;
  private int teamId;
  private int yearOfBirth;

  /**
   * Constructor.
   *
   * @param name        Rider's name
   * @param teamId      The ID of the team the rider belongs to
   * @param yearOfBirth The rider's year of birth
   */
  public Rider(String name, int teamId, int yearOfBirth) {
    this.name = name;
    this.teamId = teamId;
    this.yearOfBirth = yearOfBirth;
    this.id = latestId++;
  }

  /**
   * Reset the internal ID counter.
   */
  public static void resetIdCounter() {
    latestId = 0;
  }

```

File - Rider.java

```
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
```

File - Rider.java

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

File - CyclingPortal.java

```java
package src.cycling;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/**
 * Implementor of the CyclingPortalInterface interface.
 *
 * @author Adam Kaizra, Sam Barker
 * @version 1.0
 */

public class CyclingPortal implements CyclingPortalInterface {

  private HashMap<Integer, StagedRace> raceIdsToRaces = new HashMap<>();
  private HashMap<Integer, Stage> stageIdsToStages = new HashMap<>();
  private HashMap<Integer, Segment> segmentIdsToSegments = new HashMap<>();
  private HashMap<Integer, Team> teamIdsToTeams = new HashMap<>();
  private HashMap<Integer, Rider> riderIdsToRiders = new HashMap<>();

  /**
   * @return The hashmap of each race's ID to its corresponding StagedRace object.
   */
  public HashMap<Integer, StagedRace> getRaceIdsToRaces() {
    return raceIdsToRaces;
```

```
37   }
38
39   /**
40    * @return The hashmap of each stage's ID to its
   corresponding Stage object.
41    */
42   public HashMap<Integer, Stage> getStageIdsToStages() {
43     return stageIdsToStages;
44   }
45
46   /**
47    * @return The hashmap of each segment's ID to its
   corresponding Stage object.
48    */
49   public HashMap<Integer, Segment> getSegmentIdsToSegments
   () {
50     return segmentIdsToSegments;
51   }
52
53   /**
54    * @return The hashmap of each team's ID to its
   corresponding Team object.
55    */
56   public HashMap<Integer, Team> getTeamIdsToTeams() {
57     return teamIdsToTeams;
58   }
59
60   /**
61    * @return The hashmap of each rider's ID to its
   corresponding Rider object.
62    */
63   public HashMap<Integer, Rider> getRiderIdsToRiders() {
64     return riderIdsToRiders;
65   }
66
67   @Override
68   public int[] getRaceIds() {
69     Set<Integer> raceIdsSet = raceIdsToRaces.keySet();
70     int[] raceIds = new int[raceIdsSet.size()];
71     int index = 0;
72     for (Integer i : raceIdsSet) {
73       raceIds[index++] = i;
74     }
75
```

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

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

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

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

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

```
259     // Get the race object that contains it.
260     StagedRace race = raceIdsToRaces.get(stage.getRaceId
    ());
261     race.getStages().remove(stage);
262     assert !race.getStages().contains(stage) : "Stage not
     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 : "
    Stage not removed!";
269         foundId = true;
270         break;
271       }
272     }
273     if (!foundId) {
274       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
275     }
276   }
277
278   @Override
279   public int addCategorizedClimbToStage(int stageId,
    Double location, SegmentType type,
280       Double averageGradient, Double length)
281       throws IDNotRecognisedException,
    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 " +
    stageId + " is not recognised!");
287     }
288     // Is the location valid?
289     if ((location >= stage.getLength()) || (location <= 0
    )) {
290       throw new InvalidLocationException("Invalid
    location " + location + "!");
291     }
292     // Is the stage state "under development"?
293     if (!stage.getUnderDevelopment()) {
```

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

File - CyclingPortal.java

```
327     Stage stage = stageIdsToStages.get(stageId);
328     // Does the stage exist?
329     if (stage == null) {
330       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
331     }
332     // Is the location valid?
333     if ((location >= stage.getLength()) || (location <= 0
    )) {
334       throw new InvalidLocationException("Invalid
    location " + location + " !");
335     }
336     // Is the stage type valid?
337     if (stage.getStageType() == StageType.TT) {
338       throw new InvalidStageTypeException("Cannot add
    segments to a time trial stage.");
339     }
340     // Is the stage state "under development"?
341     if (!stage.getUnderDevelopment()) {
342       throw new InvalidStageStateException("Stage is
    waiting for results!");
343     }
344
345     // Creates a nwe intermediate sprint and adds it to
    the stage.
346     Segment intermediateSprint = new Segment(stageId,
    SegmentType.SPRINT, location);
347     int intSprintId = intermediateSprint.getId();
348     segmentIdsToSegments.put(intSprintId,
    intermediateSprint);
349     assert segmentIdsToSegments.get(intSprintId) != null
     : "Intermediate sprint not added to"
350         + "hashmap!";
351     stage.addSegment(intermediateSprint);
352
353     return intermediateSprint.getId();
354   }
355
356   @Override
357   public void removeSegment(int segmentId) throws
    IDNotRecognisedException,
358       InvalidStageStateException {
359     // Does the segment exist?
360     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()
381               + " is not under development!");
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 " +
    stageId
398           + " is already waiting for results!");
399     } else {
400       stage.setUnderDevelopment(false);
401     }
402     assert stage.getUnderDevelopment() == false : "Stage
    preparations not concluded!";
403   }
404
405   @Override
406   public int[] getStageSegments(int stageId) throws
    IDNotRecognisedException {
407     Stage stage = stageIdsToStages.get(stageId);
408     if (stage == null) {
409       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
410     }
411
412     // Iterates through the segments in the stage and
    gets their IDs.
413     ArrayList<Segment> segmentsInStage = stage.
    getSegmentsInStage();
414     int[] segmentIdsInStage = new int[segmentsInStage.
    size()];
415     Collections.sort(segmentsInStage);
416     int i = 0;
417     for (Segment segment : segmentsInStage) {
418       segmentIdsInStage[i] = segment.getId();
419       i++;
420     }
421     assert segmentsInStage.size() == segmentIdsInStage.
    length : "Segment IDs not gathered"
422         + "correctly!";
423     return segmentIdsInStage;
424   }
425
426   @Override
427   public int createTeam(String name, String description)
    throws IllegalNameException,
428       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 : "
    Team IDs not gathered correctly!";
468
469     return teamIds;
470   }
471
472   @Override
473   public int[] getTeamRiders(int teamId) throws
    IDNotRecognisedException {
474     ArrayList<Integer> riderIdArrayList = new ArrayList
    <>();
475
476     // Check if the team exists.
477     if (teamIdsToTeams.get(teamId) == null) {
478       throw new IDNotRecognisedException("Team ID " +
    teamId + " is not recognised!");
479     }
480
481     // Look for all riders in this team.
482     for (Map.Entry<Integer, Rider> idToRider :
    riderIdsToRiders.entrySet()) {
483       Integer riderId = idToRider.getKey();
484       Rider rider = idToRider.getValue();
485       if (rider.getTeamId() == teamId) {
486         // Found one; add to list.
487         riderIdArrayList.add(riderId);
488       }
489     }
490
491     // Convert the ArrayList to int[].
492     return riderIdArrayList.stream().mapToInt(i -> i).
    toArray();
493   }
494
495   @Override
496   public int createRider(int teamID, String name, int
    yearOfBirth)
497       throws IDNotRecognisedException,
    IllegalArgumentException {
498     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 " +
    teamID + " is not recognised!");
503     }
504     // Trim leading and trailing whitespace and check if
    the name is blank, is more than 30
505     // characters in length, or is null.
506     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
    successfully removed from their team!";
536         break;
537       } catch (NullPointerException ex) { } // Errors
    thrown for each team until ID is found.
538     }
539     if (!hasBeenFound) {
540       throw new IDNotRecognisedException("Rider ID " +
    riderId + " is not recognised!");
541     }
542
543     riderIdsToRiders.remove(riderId);
544     assert riderIdsToRiders.get(riderId) == null : "Rider
     not removed from hashmap!";
545
546     // Results.
547     boolean foundRiderInRace = false;
548     for (StagedRace race : raceIdsToRaces.values()) {
549       ArrayList<RiderRaceResult> raceResults = race.
    getResults();
550       // Destroy references to this rider in races.
551       for (RiderRaceResult raceResult : raceResults) {
552         if (raceResult.getRiderId() == riderId) {
553           foundRiderInRace = true;
554           raceResults.remove(raceResult);
555           assert !raceResults.contains(raceResult) : "
    Rider's race result not removed!";
556
557           // Destroy references to this rider in stages.
558           for (Stage stage : race.getStages()) {
559             ArrayList<RiderStageResult> stageResults =
    stage.getResults();
560
561             for (RiderStageResult stageResult :
    stageResults) {
562               if (stageResult.getRiderId() == riderId) {
563                 int stageResultsLength = stageResults.
    size();
564                 stageResults.remove(stageResult);
565                 assert stageResults.size() ==
    stageResultsLength - 1 : "Rider's stage result not"
566                     + "successfully removed!";
```

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

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

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

```
669       throws IDNotRecognisedException {
670     if (riderIdsToRiders.get(riderId) == null) {
671       throw new IDNotRecognisedException("Rider ID " +
    riderId + " is not recognised!");
672     }
673     Stage stage = stageIdsToStages.get(stageId);
674     if (stage == null) {
675       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
676     }
677
678     // Populate the RiderStageResults objects with their
    adjusted elapsed times.
679     stage.generateAdjustedFinishTimes();
680
681     for (RiderStageResult result : stage.getResults()) {
682       if (result.getRiderId() == riderId) {
683         return result.getAdjustedFinishTime();
684       }
685     }
686
687     // Returns null if no result for the rider is found.
688     return null;
689   }
690
691   @Override
692   public void deleteRiderResultsInStage(int stageId, int
    riderId) throws IDNotRecognisedException {
693     Stage stage = stageIdsToStages.get(stageId);
694     // Does stage exist?
695     if (stage == null) {
696       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
697     }
698
699     // Does rider exist?
700     if (riderIdsToRiders.get(riderId) == null) {
701       throw new IDNotRecognisedException("Rider ID " +
    riderId + " is not recognised!");
702     }
703
704     int stageResultsLength = stage.getResults().size();
705     stage.removeResultByRiderId(riderId);
706     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)
734       throws IDNotRecognisedException {
735     Stage stage = stageIdsToStages.get(stageId);
736     if (stage == null) { // Checks race is in cycling
    portal.
737       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
738     }
739
740     // 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
    adjusted time in rank order.
745     ArrayList<RiderStageResult> results = stage.
    getResults();
746     LocalTime[] rankAdjustedElapsedTimesInStage = new
    LocalTime[results.size()];
747     int i = 0;
748     for (RiderStageResult result : stage.getResults()) {
749       rankAdjustedElapsedTimesInStage[i] = result.
    getAdjustedFinishTime();
750       i++;
751     }
752     return rankAdjustedElapsedTimesInStage;
753   }
754
755   @Override
756   public int[] getRidersPointsInStage(int stageId) throws
     IDNotRecognisedException {
757     Stage stage = stageIdsToStages.get(stageId);
758     if (stage == null) { // Checks race is in cycling
    portal.
759       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
760     }
761
762     // Generates the points in the stage and returns
    rider IDs ordered by points.
763     return stage.generatePointsInStage(false);
764   }
765
766   @Override
767   public int[] getRidersMountainPointsInStage(int stageId
    ) throws IDNotRecognisedException {
768     Stage stage = stageIdsToStages.get(stageId);
769     if (stage == null) { // Checks race is in cycling
    portal.
770       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
771     }
772     if ((stage.getStageType() == StageType.TT) || (stage.
    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         climbFound = true;
782       }
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
    returns rider IDs ordered by  mountain
789     // points.
790     return stage.generatePointsInStage(true);
791   }
792
793   @Override
794   public void eraseCyclingPortal() {
795     // Reset all internal counters.
796     Rider.resetIdCounter();
797     Team.resetIdCounter();
798     Stage.resetIdCounter();
799     StagedRace.resetIdCounter();
800     Segment.resetIdCounter();
801     CategorisedClimb.resetIdCounter();
802     // Erase all references and get them garbage
    collected.
803     this.raceIdsToRaces = new HashMap<>();
804     this.stageIdsToStages = new HashMap<>();
805     this.segmentIdsToSegments = new HashMap<>();
806     this.teamIdsToTeams = new HashMap<>();
807     this.riderIdsToRiders = new HashMap<>();
808   }
809
810   @Override
811   public void saveCyclingPortal(String filename) throws
    IOException {
812     FileOutputStream fileOutputStream = new
```

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

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

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

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

File - CategorisedClimb.java

```java
package src.cycling;

import java.io.Serializable;

/**
 * Represents a categorised climb segment in a race. This is a subclass of Segment (which
 * represents an intermediate sprint) as they share common features.
 *
 * @author Sam Barker, Adam Kaizra
 * @version 1.0
 */
public class CategorisedClimb extends Segment implements Serializable {

  private static int latestId = 0;
  private final int id;
  private int stageId;
  private final double averageGradient;
  private final double length;
  private SegmentType segmentType;

  /**
   * Constructor.
   *
   * @param stageId The ID of the stage that this segment is part of.
   * @param segmentType The SegmentType of this climb (i.e. HC, C1...3)
   * @param averageGradient The average gradient of the climb segment
   * @param length The length (distance from segment start point to end) of the climb segment.
   * @param location The location in the stage at which the categorised climb starts.
   */
  public CategorisedClimb(int stageId, SegmentType segmentType, Double averageGradient,
      Double length, Double location) {
    super(stageId, segmentType, location);
    this.averageGradient = averageGradient;
    this.length = length;
    this.id = latestId++;
```

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