File - CategorisedClimb.java

```java
package src.cycling;

/**
 * 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 {

  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++;
  }

  /**
```

```java
    * Reset the static ID counter to 0. Used after deletion of a CyclingPortal.
    */
  public static void resetIdCounter() {
    latestId = 0;
  }

  /**
   * @return The length (in distance) of the categorised climb.
   */
  public double getLength() {
    return this.length;
  }

  /**
   * @return The average gradient of the categorised climb.
   */
  public double getAverageGradient() {
    return this.averageGradient;
  }
}

```

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.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;
  }
```

```java

  /**
   * @return The hashmap of each stage's ID to its corresponding Stage object.
   */
  public HashMap<Integer, Stage> getStageIdsToStages() {
    return stageIdsToStages;
  }

  /**
   * @return The hashmap of each segment's ID to its corresponding Stage object.
   */
  public HashMap<Integer, Segment> getSegmentIdsToSegments() {
    return segmentIdsToSegments;
  }

  /**
   * @return The hashmap of each team's ID to its corresponding Team object.
   */
  public HashMap<Integer, Team> getTeamIdsToTeams() {
    return teamIdsToTeams;
  }

  /**
   * @return The hashmap of each rider's ID to its corresponding Rider object.
   */
  public HashMap<Integer, Rider> getRiderIdsToRiders() {
    return riderIdsToRiders;
  }

  @Override
  public int[] getRaceIds() {
    Set<Integer> raceIdsSet = raceIdsToRaces.keySet();
    int[] raceIds = new int[raceIdsSet.size()];
    int index = 0;
    for (Integer i : raceIdsSet) {
      raceIds[index++] = i;
    }

    return raceIds;
```

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

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

```
148   }
149
150   @Override
151   public int addStageToRace(int raceId, String name,
    String description, double length,
152       LocalDateTime startTime, StageType type)
153       throws IDNotRecognisedException,
    IllegalNameException, InvalidNameException,
154       InvalidLengthException {
155     // Check the race exists.
156     StagedRace race = raceIdsToRaces.get(raceId);
157     if (race == null) {
158       throw new IDNotRecognisedException("Race ID " +
    raceId + " is not recognised!");
159     }
160     // Trim leading and trailing whitespace and check if
    the name is blank, is more than 30
161     // characters in length, or is null.
162     name = name.trim();
163     if (name.equals("") || name.length() > 30 || name ==
    null) {
164       throw new InvalidNameException("Name '" + name +
    "' is greater than 30 characters!");
165     }
166
167     // For assertion.
168     int amountOfStages = race.getStages().size();
169
170     // Check the length is >=5km.
171     Stage stage = new Stage(raceId, name, description,
    length, startTime, type);
172     stageIdsToStages.put(stage.getId(), stage);
173     race.addStage(stage);
174
175     assert race.getStages().size() == amountOfStages + 1;
176
177     return stage.getId();
178   }
179
180   @Override
181   public int[] getRaceStages(int raceId) throws
    IDNotRecognisedException {
182     StagedRace race = raceIdsToRaces.get(raceId);
183     if (race == null) {
```

```
184       throw new IDNotRecognisedException("Race ID " +
    raceId + " is not recognised!");
185     }
186
187     // Iterates though stages in race and gets their IDs.
188     ArrayList<Stage> stages = race.getStages();
189     int[] stageIds = new int[stages.size()];
190     int i = 0;
191     for (Stage stage : stages) {
192       stageIds[i] = stage.getId();
193       i++;
194     }
195
196     assert race.getStages().size() == stageIds.length : "
    Incorrect amount of stage IDs!";
197
198     return stageIds;
199   }
200
201   @Override
202   public double getStageLength(int stageId) throws
    IDNotRecognisedException {
203     for (Map.Entry<Integer, Stage> idToStg :
    stageIdsToStages.entrySet()) {
204       if (idToStg.getKey() == stageId) {
205         return idToStg.getValue().getLength();
206       }
207     }
208     throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
209   }
210
211   @Override
212   public void removeStageById(int stageId) throws
    IDNotRecognisedException {
213     Stage stage = stageIdsToStages.get(stageId);
214     if (stage == null) {
215       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
216     }
217     // Get the race object that contains it.
218     StagedRace race = raceIdsToRaces.get(stage.getRaceId
    ());
219     race.getStages().remove(stage);
```

```
220     assert !race.getStages().contains(stage) : "Stage not
     removed from race!";
221
222     Boolean foundId = false;
223     for (Integer stgId : stageIdsToStages.keySet()) {
224       if (stgId == stageId) {
225         stageIdsToStages.remove(stgId);
226         assert stageIdsToStages.get(stgId) == null : "
    Stage not removed!";
227         foundId = true;
228         break;
229       }
230     }
231     if (!foundId) {
232       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
233     }
234   }
235
236   @Override
237   public int addCategorizedClimbToStage(int stageId,
    Double location, SegmentType type,
238       Double averageGradient, Double length)
239       throws IDNotRecognisedException,
    InvalidLocationException, InvalidStageStateException,
240       InvalidStageTypeException {
241     Stage stage = stageIdsToStages.get(stageId);
242     // Does the stage exist?
243     if (stage == null) {
244       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
245     }
246     // Is the location valid?
247     if ((location >= stage.getLength()) || (location <= 0
    )) {
248       throw new InvalidLocationException("Invalid
    location " + location + "!");
249     }
250     // Is the stage state "under development"?
251     if (!stage.getUnderDevelopment()) {
252       throw new InvalidStageStateException("Stage is
    waiting for results!");
253     }
254     // Are you trying to add a sprint using the wrong
```

```
254 method?
255     if (type == SegmentType.SPRINT) {
256       throw new InvalidStageTypeException("Cannot add
    sprint to stage using this method!");
257     }
258     if (stage.getStageType() == StageType.TT) {
259       throw new InvalidStageTypeException("Cannot add
    segments to a time trial stage.");
260     }
261
262     // Is the length valid?
263     if (length <= 0) {
264       throw new InvalidLocationException("Invalid length
    entered! Enter one >= 0.");
265     }
266     if ((location + length) > stage.getLength()) {
267       throw new InvalidLocationException("Invalid length
    exceeds length of stage!");
268     }
269
270     // Creates new categorised climb and adds it to the
    stage.
271     Segment categorisedClimb = new CategorisedClimb(
    stageId, type, averageGradient,
272         length, location);
273     int ccId = categorisedClimb.getId();
274     segmentIdsToSegments.put(ccId, categorisedClimb);
275     assert segmentIdsToSegments.get(ccId) != null : "
    Categorised climb not added to hashmap!";
276     stage.addSegment(categorisedClimb);
277
278     return categorisedClimb.getId();
279   }
280
281   @Override
282   public int addIntermediateSprintToStage(int stageId,
    double location)
283       throws IDNotRecognisedException,
    InvalidLocationException, InvalidStageStateException,
284       InvalidStageTypeException {
285     Stage stage = stageIdsToStages.get(stageId);
286     // Does the stage exist?
287     if (stage == null) {
288       throw new IDNotRecognisedException("Stage ID " +
```

```
288 stageId + " is not recognised!");
289     }
290     // Is the location valid?
291     if ((location >= stage.getLength()) || (location <= 0
    )) {
292       throw new InvalidLocationException("Invalid
    location " + location + " !");
293     }
294     // Is the stage type valid?
295     if (stage.getStageType() == StageType.TT) {
296       throw new InvalidStageTypeException("Cannot add
    segments to a time trial stage.");
297     }
298     // Is the stage state "under development"?
299     if (!stage.getUnderDevelopment()) {
300       throw new InvalidStageStateException("Stage is
    waiting for results!");
301     }
302
303     // Creates a nwe intermediate sprint and adds it to
    the stage.
304     Segment intermediateSprint = new Segment(stageId,
    SegmentType.SPRINT, location);
305     int intSprintId = intermediateSprint.getId();
306     segmentIdsToSegments.put(intSprintId,
    intermediateSprint);
307     assert segmentIdsToSegments.get(intSprintId) != null
     : "Intermediate sprint not added to"
308         + "hashmap!";
309     stage.addSegment(intermediateSprint);
310
311     return intermediateSprint.getId();
312   }
313
314   @Override
315   public void removeSegment(int segmentId) throws
    IDNotRecognisedException,
316       InvalidStageStateException {
317     // Does the segment exist?
318     if (segmentIdsToSegments.get(segmentId) == null) {
319       throw new IDNotRecognisedException("Segment ID " +
    segmentId + " is not recognised!");
320     }
321     Segment segment = segmentIdsToSegments.get(segmentId
```

```
321 );
322     int segmentStageId = segment.getStageId();
323     // Can I do this (the stage has to be under
    development).
324     // Find the stage.
325     for (Map.Entry<Integer, Stage> stageEntry :
    stageIdsToStages.entrySet()) {
326       Stage stage = stageEntry.getValue();
327       if (stage.getId() == segmentStageId) {
328         // This is the right stage, now check we can
    delete segments from it.
329         if (stage.getUnderDevelopment()) {
330           segmentIdsToSegments.remove(segmentId);
331           assert segmentIdsToSegments.get(segmentId) ==
    null : "Segment not removed from hashmap!";
332         } else {
333           throw new InvalidStageStateException("Stage "
     + stage.getId()
334               + " is not under development!");
335         }
336       }
337     }
338   }
339
340   @Override
341   public void concludeStagePreparation(int stageId)
    throws IDNotRecognisedException,
342       InvalidStageStateException {
343     // Does the stage exist?
344     if (stageIdsToStages.get(stageId) == null) {
345       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
346     }
347     // Is it already waiting for results?
348     Stage stage = stageIdsToStages.get(stageId);
349     if (!stage.getUnderDevelopment()) {
350       throw new InvalidStageStateException("Stage " +
    stageId
351           + " is already waiting for results!");
352     } else {
353       stage.setUnderDevelopment(false);
354     }
355     assert stage.getUnderDevelopment() == false : "Stage
    preparations not concluded!";
```

```
356   }
357
358   @Override
359   public int[] getStageSegments(int stageId) throws
    IDNotRecognisedException {
360     Stage stage = stageIdsToStages.get(stageId);
361     if (stage == null) {
362       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
363     }
364
365     // Iterates through the segments in the stage and
    gets their IDs.
366     ArrayList<Segment> segmentsInStage = stage.
    getSegmentsInStage();
367     int[] segmentIdsInStage = new int[segmentsInStage.
    size()];
368     Collections.sort(segmentsInStage);
369     int i = 0;
370     for (Segment segment : segmentsInStage) {
371       segmentIdsInStage[i] = segment.getId();
372       i++;
373     }
374     assert segmentsInStage.size() == segmentIdsInStage.
    length : "Segment IDs not gathered"
375         + "correctly!";
376     return segmentIdsInStage;
377   }
378
379   @Override
380   public int createTeam(String name, String description)
    throws IllegalNameException,
381       InvalidNameException {
382     // Trim leading and trailing whitespace and check if
    the name is blank, is more than 30
383     // characters in length, or is null.
384     name = name.trim();
385     if (name == null || name == "" || name.length() > 30
    ) {
386       throw new InvalidNameException("Invalid name of a
    team!");
387     }
388     for (Team team : teamIdsToTeams.values()) {
389       if (team.getName() == name) {
```

```
390         throw new IllegalNameException("Team name already
     in use!");
391       }
392     }
393     // Creates a new team and then stores it in a hashmap
     for easier access.
394     Team newTeam = new Team(name, description);
395     int newTeamId = newTeam.getId();
396     teamIdsToTeams.put(newTeamId, newTeam);
397     assert teamIdsToTeams.get(newTeamId) != null : "Team
    not added to hashmap!";
398     return newTeam.getId();
399   }
400
401   @Override
402   public void removeTeam(int teamId) throws
    IDNotRecognisedException {
403     Team team = teamIdsToTeams.get(teamId);
404     if (team == null) {
405       throw new IDNotRecognisedException("Team ID " +
    teamId + " is not recognised!");
406     }
407     teamIdsToTeams.remove(teamId);
408     assert teamIdsToTeams.get(teamId) == null : "Team not
     removed from hashmap!";
409   }
410
411   @Override
412   public int[] getTeams() {
413     // Iterates through the IDs of teams from the hashmap
     and returns it as an integer array.
414     Set<Integer> teamIdsSet = teamIdsToTeams.keySet();
415     int[] teamIds = new int[teamIdsSet.size()];
416     int index = 0;
417     for (Integer i : teamIdsSet) {
418       teamIds[index++] = i;
419     }
420     assert teamIdsToTeams.size() == teamIds.length : "
    Team IDs not gathered correctly!";
421
422     return teamIds;
423   }
424
425   @Override
```

```
426   public int[] getTeamRiders(int teamId) throws
    IDNotRecognisedException {
427     ArrayList<Integer> riderIdArrayList = new ArrayList
    <>();
428
429     // Check if the team exists.
430     if (teamIdsToTeams.get(teamId) == null) {
431       throw new IDNotRecognisedException("Team ID " +
    teamId + " is not recognised!");
432     }
433
434     // Look for all riders in this team.
435     for (Map.Entry<Integer, Rider> idToRider :
    riderIdsToRiders.entrySet()) {
436       Integer riderId = idToRider.getKey();
437       Rider rider = idToRider.getValue();
438       if (rider.getTeamId() == teamId) {
439         // Found one; add to list.
440         riderIdArrayList.add(riderId);
441       }
442     }
443
444     // Convert the ArrayList to int[].
445     return riderIdArrayList.stream().mapToInt(i -> i).
    toArray();
446   }
447
448   @Override
449   public int createRider(int teamID, String name, int
    yearOfBirth)
450       throws IDNotRecognisedException,
    IllegalArgumentException {
451     name = name.trim();
452     // Check if the team exists.
453     Team team = teamIdsToTeams.get(teamID);
454     if (team == null) {
455       throw new IDNotRecognisedException("Team ID " +
    teamID + " is not recognised!");
456     }
457     // Trim leading and trailing whitespace and check if
    the name is blank, is more than 30
458     // characters in length, or is null.
459     if (name == null || name == "" || name.length() > 30
    ) {
```

```
460       throw new IllegalArgumentException("Invalid name of
     a team!");
461     }
462
463     // Creates a new rider and adds it to the hashmap for
     easier access.
464     Rider newRider = new Rider(name, teamID, yearOfBirth
    );
465     int newRiderId = newRider.getId();
466     riderIdsToRiders.put(newRiderId, newRider);
467     assert riderIdsToRiders.get(newRiderId) != null : "
    Rider not added to hashmap!";
468     team.addRider(newRider);
469
470     return newRider.getId();
471   }
472
473   @Override
474   public void removeRider(int riderId) throws
    IDNotRecognisedException {
475     boolean hasBeenFound = false;
476     for (Team team : teamIdsToTeams.values()) {
477       HashMap<Integer, Rider> riders = team.
    getRiderIdsToRiders();
478       try {
479         if (riders.get(riderId) != null) {
480           hasBeenFound = true;
481         }
482         riders.remove(riderId);
483         assert riders.get(riderId) == null : "Rider not
    successfully removed from their team!";
484         break;
485       } catch (NullPointerException ex) { } // Errors
    thrown for each team until ID is found.
486     }
487     if (!hasBeenFound) {
488       throw new IDNotRecognisedException("Rider ID " +
    riderId + " is not recognised!");
489     }
490
491     riderIdsToRiders.remove(riderId);
492     assert riderIdsToRiders.get(riderId) == null : "Rider
     not removed from hashmap!";
493
```

File - CyclingPortal.java

```
494     // Results.
495     boolean foundRiderInRace = false;
496     for (StagedRace race : raceIdsToRaces.values()) {
497       ArrayList<RiderRaceResult> raceResults = race.
    getResults();
498       // Destroy references to this rider in races.
499       for (RiderRaceResult raceResult : raceResults) {
500         if (raceResult.getRiderId() == riderId) {
501           foundRiderInRace = true;
502           raceResults.remove(raceResult);
503           assert !raceResults.contains(raceResult) : "
    Rider's race result not removed!";
504
505           // Destroy references to this rider in stages.
506           for (Stage stage : race.getStages()) {
507             ArrayList<RiderStageResult> stageResults =
    stage.getResults();
508
509             for (RiderStageResult stageResult :
    stageResults) {
510               if (stageResult.getRiderId() == riderId) {
511                 int stageResultsLength = stageResults.
    size();
512                 stageResults.remove(stageResult);
513                 assert stageResults.size() ==
    stageResultsLength - 1 : "Rider's stage result not"
514                     + "successfully removed!";
515
516                 // Destroy references to this rider in
    segments
517                 for (Segment segment : stage.
    getSegmentsInStage()) {
518                   ArrayList<RiderSegmentResult>
    segmentResults = segment.getResults();
519
520                   for (RiderSegmentResult segmentResult
     : segmentResults) {
521                     if (segmentResult.getRiderId() ==
    riderId) {
522                       int segmentResultsLength =
    segmentResults.size();
523                       segmentResults.remove(segmentResult
    );
524                       assert segmentResults.size() ==
```

```
524 segmentResultsLength - 1 : "Rider's segment"
525                             + "result not successfully
    removed!";
526                       }
527                     }
528                   }
529                 }
530               }
531             }
532           }
533         }
534       }
535     }
536
537     @Override
538     public void registerRiderResultsInStage(int stageId,
    int riderId, LocalTime... checkpoints)
539         throws IDNotRecognisedException,
    DuplicatedResultException, InvalidCheckpointsException,
540         InvalidStageStateException {
541       Stage stage = stageIdsToStages.get(stageId); // Gets
    the stage.
542
543       // Error checking.
544       if (stage == null) {
545         throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
546       }
547       Rider rider = riderIdsToRiders.get(riderId);
548       if (rider == null) {
549         throw new IDNotRecognisedException("Rider ID " +
    riderId + " is not recognised!");
550       }
551
552       if (stage.getUnderDevelopment()) {
553         throw new InvalidStageStateException("The stage is
    under development so can't add rider"
554             + "results!");
555       }
556       // Get the location data for each segment, order it,
    sync it to times and store it.
557       // Check whether the result has already been
    registered for this rider in this stage.
558       ArrayList<RiderStageResult> results = stage.
```

```
558 getResults();
559     if (!results.isEmpty()) {
560       for (RiderStageResult result : results) {
561         if (result.getRiderId() == riderId) {
562           throw new DuplicatedResultException("Rider's
    result already registered in this stage!");
563         }
564       }
565     }
566 
567     // Check the number of checkpoints in the stage
568     ArrayList<Segment> segments = stage.
    getSegmentsInStage();
569     int checkpointLength = checkpoints.length;
570     if (!(segments == null)) {
571       if (!(checkpointLength == segments.size() + 2)) {
572         throw new InvalidCheckpointsException("Incorrect
    number of checkpoints in stage!");
573       }
574     } else if (checkpointLength != 2) {
575       throw new InvalidCheckpointsException("Incorrect
    number of checkpoints in stage!");
576     }
577 
578     // Check whether checkpoints are in chronological
    order.
579     LocalTime previousTime = LocalTime.of(0, 0, 0);
580     for (LocalTime time : checkpoints) {
581       if (time.compareTo(previousTime) < 0) {
582         throw new InvalidCheckpointsException("Checkpoint
     times are not in chronological order!");
583       }
584       previousTime = time;
585     }
586 
587     int stageResultsLength = stage.getResults().size();
588     stage.addRiderResults(riderId, checkpoints);
589     assert stage.getResults().size() ==
    stageResultsLength + 1 : "Rider not added to stage!";
590   }
591 
592   @Override
593   public LocalTime[] getRiderResultsInStage(int stageId,
    int riderId)
```

```
594       throws IDNotRecognisedException {
595     Stage stage = stageIdsToStages.get(stageId);
596     if (stage == null) {
597       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
598     }
599 
600     // If the rider doesn't exist / doesn't have a result
     in this stage.
601     RiderStageResult result = null;
602     for (RiderStageResult tmpResult : stage.getResults
    ()) {
603       if (tmpResult.getRiderId() == riderId) {
604         result = tmpResult;
605         break;
606       }
607     }
608     if (result == null) {
609       return new LocalTime[0];
610     }
611 
612     return result.getTimes();
613   }
614 
615   @Override
616   public LocalTime getRiderAdjustedElapsedTimeInStage(int
     stageId, int riderId)
617       throws IDNotRecognisedException {
618     if (riderIdsToRiders.get(riderId) == null) {
619       throw new IDNotRecognisedException("Rider ID " +
    riderId + " is not recognised!");
620     }
621     Stage stage = stageIdsToStages.get(stageId);
622     if (stage == null) {
623       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
624     }
625 
626     // Populate the RiderStageResults objects with their
    adjusted elapsed times.
627     stage.generateAdjustedResults();
628 
629     for (RiderStageResult result : stage.getResults()) {
630       if (result.getRiderId() == riderId) {
```

```
631         return result.getAdjustedFinishTime();
632       }
633     }
634
635     // Returns null if no result for the rider is found.
636     return null;
637   }
638
639   @Override
640   public void deleteRiderResultsInStage(int stageId, int
    riderId) throws IDNotRecognisedException {
641     // Does stage exist?
642     if (stageIdsToStages.get(stageId) == null) {
643       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
644     }
645     Stage stage = stageIdsToStages.get(stageId);
646     // Does rider exist?
647     if (riderIdsToRiders.get(riderId) == null) {
648       throw new IDNotRecognisedException("Rider ID " +
    riderId + " is not recognised!");
649     }
650
651     int stageResultsLength = stage.getResults().size();
652     stage.removeResultByRiderId(riderId);
653     assert stage.getResults().size() ==
    stageResultsLength - 1 : "Rider not removed from stage!";
654   }
655
656   @Override
657   public int[] getRidersRankInStage(int stageId) throws
    IDNotRecognisedException {
658     // Does stage exist?
659     Stage stage = stageIdsToStages.get(stageId);
660     if (stage == null) {
661       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
662     }
663
664     ArrayList<Integer> riderIdsList = new ArrayList<>();
665     ArrayList<RiderStageResult> results = stage.
    getResults();
666
667     // Fill the IDs list with the rider ids which are now
```

```
667  in rank order.
668      for (RiderStageResult result : results) {
669        riderIdsList.add(result.getRiderId());
670      }
671
672      // Convert the ArrayList to int[].
673      return riderIdsList.stream().mapToInt(i -> i).toArray
    ();
674    }
675
676    @Override
677    public LocalTime[] getRankedAdjustedElapsedTimesInStage
    (int stageId)
678        throws IDNotRecognisedException {
679      Stage stage = stageIdsToStages.get(stageId);
680      if (stage == null) { // Checks race is in cycling
    portal.
681        throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
682      }
683
684      // Creates the adjusted results and adds the
    attribute to the result object.
685      // Also orders results by finishing time (rank).
686      stage.generateAdjustedResults();
687
688      // Iterates through stage results and gets their
    adjusted time in rank order.
689      ArrayList<RiderStageResult> results = stage.
    getResults();
690      LocalTime[] rankAdjustedElapsedTimesInStage = new
    LocalTime[results.size()];
691      int i = 0;
692      for (RiderStageResult result : stage.getResults()) {
693        rankAdjustedElapsedTimesInStage[i] = result.
    getAdjustedFinishTime();
694        i++;
695      }
696      return rankAdjustedElapsedTimesInStage;
697    }
698
699    @Override
700    public int[] getRidersPointsInStage(int stageId) throws
     IDNotRecognisedException {
```

```
701     Stage stage = stageIdsToStages.get(stageId);
702     if (stage == null) { // Checks race is in cycling
    portal.
703       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
704     }
705
706     // Generates the points in the stage and returns
    rider IDs ordered by points.
707     return stage.generatePointsInStage(false);
708   }
709
710   @Override
711   public int[] getRidersMountainPointsInStage(int stageId
    ) throws IDNotRecognisedException {
712     Stage stage = stageIdsToStages.get(stageId);
713     if (stage == null) { // Checks race is in cycling
    portal.
714       throw new IDNotRecognisedException("Stage ID " +
    stageId + " is not recognised!");
715     }
716     if ((stage.getStageType() == StageType.TT) || (stage.
    getSegmentsInStage().size() == 0)) {
717       // No mountain points (no segments).
718       return new int[0];
719     }
720
721     // Check if there even are any climbs.
722     boolean climbFound = false;
723     for (Segment segment : stage.getSegmentsInStage()) {
724       if (segment.getSegmentType() != SegmentType.SPRINT
    ) {
725         climbFound = true;
726       }
727     }
728     // No climbs so there will be no results.
729     if (!climbFound) {
730       return new int[0];
731     }
732     // Generates the mountain points in the stage and
    returns rider IDs ordered by  mountain
733     // points.
734     return stage.generatePointsInStage(true);
735   }
```

```
736
737   @Override
738   public void eraseCyclingPortal() {
739     // Reset all internal counters.
740     Rider.resetIdCounter();
741     Team.resetIdCounter();
742     Stage.resetIdCounter();
743     StagedRace.resetIdCounter();
744     Segment.resetIdCounter();
745     CategorisedClimb.resetIdCounter();
746     // Erase all references and get them garbage
    collected.
747     this.raceIdsToRaces = new HashMap<>();
748     this.stageIdsToStages = new HashMap<>();
749     this.segmentIdsToSegments = new HashMap<>();
750     this.teamIdsToTeams = new HashMap<>();
751     this.riderIdsToRiders = new HashMap<>();
752   }
753
754   @Override
755   public void saveCyclingPortal(String filename) throws
    IOException {
756     FileOutputStream fileOutputStream = new
    FileOutputStream(filename);
757     ObjectOutputStream objectOutputStream = new
    ObjectOutputStream(fileOutputStream);
758     objectOutputStream.writeObject(this);
759     objectOutputStream.flush();
760     objectOutputStream.close();
761   }
762
763   @Override
764   public void loadCyclingPortal(String filename) throws
    IOException, ClassNotFoundException {
765     FileInputStream fileInputStream = new FileInputStream
    (filename);
766     ObjectInputStream objectInputStream = new
    ObjectInputStream(fileInputStream);
767     CyclingPortal tmp = (CyclingPortal) objectInputStream
    .readObject();
768     this.raceIdsToRaces = tmp.raceIdsToRaces;
769     this.stageIdsToStages = tmp.stageIdsToStages;
770     this.segmentIdsToSegments = tmp.segmentIdsToSegments;
771     this.teamIdsToTeams = tmp.teamIdsToTeams;
```

```
772     this.riderIdsToRiders = tmp.riderIdsToRiders;
773   }
774
775   @Override
776   public void removeRaceByName(String name) throws
    NameNotRecognisedException {
777     boolean foundRace = false;
778     // Find the named race's ID.
779     for (Map.Entry<Integer, StagedRace> stagedRaceEntry
     : raceIdsToRaces.entrySet()) {
780       int raceId = stagedRaceEntry.getKey();
781       StagedRace stagedRace = stagedRaceEntry.getValue();
782       if (stagedRace.getName().equals(name)) {
783         // Found it.
784         foundRace = true;
785         raceIdsToRaces.remove(raceId);
786         assert raceIdsToRaces.get(raceId) == null : "Race
     not removed from hashmap!";
787       }
788     }
789     // Gone through the whole hashmap and still haven't
    found it.
790     if (!foundRace) {
791       throw new NameNotRecognisedException("No race found
     with name '" + name + "'!");
792     }
793   }
794
795   @Override
796   public LocalTime[] getGeneralClassificationTimesInRace(
    int raceId)
797       throws IDNotRecognisedException {
798     StagedRace race = raceIdsToRaces.get(raceId);
799     if (race == null) {
800       throw new IDNotRecognisedException("Race ID " +
    raceId + " is not recognised!");
801     }
802     // Calculates the finish times for riders in the race
    .
803     race.generateRidersRaceFinishTimes();
804     ArrayList<RiderRaceResult> raceResults = race.
    getResults();
805
806     // Iterates through race results and adds finish time
```

```
806  to an array.
807     LocalTime[] finishTimes = new LocalTime[raceResults.
    size()];
808     int i = 0;
809     for (RiderRaceResult result : raceResults) {
810       finishTimes[i] = result.getFinishTime();
811       i++;
812     }
813     assert raceResults.size() == finishTimes.length : "
    Finish times gathered incorrectly!";
814     return finishTimes;
815   }
816 
817   @Override
818   public int[] getRidersPointsInRace(int raceId) throws
    IDNotRecognisedException {
819     StagedRace race = raceIdsToRaces.get(raceId);
820     if (race == null) { // Error check.
821       throw new IDNotRecognisedException("Race ID " +
    raceId + " is not recognised!");
822     }
823     // Calculate points and returns an array of points
    sorted by descending order.
824     return race.generateRidersPointsInRace(false);
825   }
826 
827   @Override
828   public int[] getRidersMountainPointsInRace(int raceId)
    throws IDNotRecognisedException {
829     StagedRace race = raceIdsToRaces.get(raceId);
830     if (race == null) { // Error check.
831       throw new IDNotRecognisedException("Race ID " +
    raceId + " is not recognised!");
832     }
833     // Calculate mountain points and returns an array of
    mountain points sorted by descending order.
834     return race.generateRidersPointsInRace(true);
835   }
836 
837   @Override
838   public int[] getRidersGeneralClassificationRank(int
    raceId) throws IDNotRecognisedException {
839     StagedRace race = raceIdsToRaces.get(raceId); // Gets
     the race.
```

```
840     if (race == null) { // Error check.
841       throw new IDNotRecognisedException("Race ID " +
    raceId + " is not recognised!");
842     }
843     race.generateRidersRaceFinishTimes(); // Returns race
     results also stores in race result list.
844     ArrayList<RiderRaceResult> raceResults = race.
    getResults(); // Gets race result arraylist.
845
846     // Iterates through race results to fill an array
    with rider IDs sorted by their rank.
847     int raceResultsSize = raceResults.size();
848     int[] riderIdsOrderedByRank = new int[raceResultsSize
    ];
849     assert raceResultsSize == riderIdsOrderedByRank.
    length : "Integer array created incorrectly!";
850     for (int i = 0; i < raceResultsSize; i++) {
851       riderIdsOrderedByRank[i] = raceResults.get(i).
    getRiderId();
852     }
853
854     return riderIdsOrderedByRank;
855   }
856
857   @Override
858   public int[] getRidersPointClassificationRank(int
    raceId) throws IDNotRecognisedException {
859     StagedRace race = raceIdsToRaces.get(raceId);
860     if (race == null) {
861       throw new IDNotRecognisedException("Race ID " +
    raceId + " is not recognised!");
862     }
863     // Calculate points in race.
864     race.generateRidersPointsInRace(false);
865
866     return race.getRiderIdsOrderedByPoints(false);
867   }
868
869   @Override
870   public int[] getRidersMountainPointClassificationRank(
    int raceId)
871       throws IDNotRecognisedException {
872     StagedRace race = raceIdsToRaces.get(raceId);
873     if (race == null) {
```

File - CyclingPortal.java

```
874         throw new IDNotRecognisedException("Race ID " +
    raceId + " is not recognised!");
875       }
876       // Calculate mountain points in race.
877       race.generateRidersPointsInRace(true);
878 
879       return race.getRiderIdsOrderedByPoints(true);
880     }
881 }
```

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;
  }

```

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

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

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

  /**
   * @return The ID of the team the rider is in.
   */
  public int getTeamId() {
    return this.teamId;
  }

  /**
   * @param newTeamId
   */
  public void setTeamId(int newTeamId) {
    this.teamId = newTeamId;
  }

  /**
   * @return The rider's year of birth.
   */
  public int getYearOfBirth() {
    return yearOfBirth;
  }

  /**
   * @param newYearOfBirth
```

File - Rider.java

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

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

File - RiderRaceResult.java

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

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

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

File - RiderSegmentResult.java

```java
package src.cycling;

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> {

  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) {
    assert (result instanceof RiderSegmentResult) : "
```

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

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

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.

  /**
   * 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.
   */
  public RiderStageResult(int riderId, int stageId,
```

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

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

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

File - Segment.java

```java
package src.cycling;

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> {

  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.
   */
  public static void resetIdCounter() {
    latestId = 0;
```

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

```java
  public ArrayList<RiderSegmentResult> getResults() {
    return results;
  }

  /**
   * Set the ArrayList of RiderSegmentResults objects representing the per-segment, per-rider
   * results in this segment.
   *
   * @param results
   */
  public void setResults(ArrayList<RiderSegmentResult> results) {
    this.results = results;
  }

  /**
   * @return The ID of this particular segment.
   */
  public Integer getId() {
    return id;
  }

  /**
   * @return The location of the segment in the stage
   */
  public Double getLocation() {
    return location;
  }

  /**
   * Set the location of the segment in the stage.
   *
   * @param location
   */
  public void setLocation(Double location) {
    this.location = location;
  }
}

```

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 {

  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
   * @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    /**
54     * Reset the internal ID counter.
55     */
56    public static void resetIdCounter() {
57      latestId = 0;
58    }
59
60    /**
61     * @return The stage's ID.
62     */
63    public Integer getId() {
64      return id;
65    }
66
67    /**
68     * @return The ID of the staged race this stage is in.
69     */
70    public Integer getRaceId() {
71      return raceId;
72    }
73
74    /**
75     * @return The name of the stage.
76     */
77    public String getName() {
78      return name;
79    }
80
81    /**
```

```java
     * @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;
    }

    /**
     * @return The local date and time of the start of the stage.
     */
    public LocalDateTime getStartTime() {
        return startTime;
    }

    /**
```

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

```
165     return underDevelopment;
166   }
167
168   /**
169    * If true the state of the stage is under development
    . If false the state of the stage is
170    * awaiting results.
171    *
172    * @param newState
173    */
174   public void setUnderDevelopment(Boolean newState) {
175     this.underDevelopment = newState;
176   }
177
178   /**
179    * @return An array list containing all the
    RiderStageResult objects relating to this stage.
180    */
181   public ArrayList<RiderStageResult> getResults() {
182     return this.results;
183   }
184
185   /**
186    * Removes the results of a rider from a stage.
187    *
188    * @param riderId The rider ID of the rider to be
    removed.
189    * @throws IDNotRecognisedException
190    */
191   public void removeResultByRiderId(int riderId) throws
    IDNotRecognisedException {
192     boolean riderIdFound = false;
193     RiderStageResult finalResult = null; // Empty value
    to be assigned if the rider is found.
194     for (RiderStageResult result : results) { // Iterates
     through the stage's rider results to find
195       // the correct rider.
196       if (result.getRiderId() == riderId) {
197         riderIdFound = true;
198         finalResult = result;
199         break;
200       }
201     }
202     if (!riderIdFound) { // If the rider can't be found
```

```
202 it can't be removed.
203       throw new IDNotRecognisedException("Rider ID " +
    riderId + " not found in stage!");
204     }
205     results.remove(finalResult); // Removes the result
    from the stage results array list.
206     assert !results.contains(finalResult) : "Rider's
    result not removed!";
207   }
208
209   /**
210    * Creates a new RiderStageResult object and adds it to
     an array list of rider results in this
211    * stage.
212    *
213    * @param riderId
214    * @param times The times the rider passed each
    checkpoint in chronological order.
215    */
216   public void addRiderResults(Integer riderId, LocalTime
    [] times) {
217     RiderStageResult result = new RiderStageResult(
    riderId, this.id, times); // Creates new result.
218     results.add(result); // The new result object is
    added to an array list of all result objects in
219     // the stage.
220     assert (results.contains(result)) : "Result object
    not created!";
221     Collections.sort(results); // Sorts the results by
    their finishing times.
222   }
223
224   /**
225    * Searches all the stages' riders' results to find
    pelotons (where riders are less than one
226    * second behind the rider in front of them) and sets
    the adjusted time of each applicable result
227    * to the peloton leader's finish time.
228    */
229   public void generateAdjustedResults() {
230     // Initialise previous time and peloton leader time
    to zero.
231     LocalTime previousTime = LocalTime.of(0, 0, 0, 0);
232     LocalTime pelotonLeader = LocalTime.of(0, 0, 0);
```

```
233
234     for (RiderStageResult result : this.results) { // Iterates through results of the stage.
235       LocalTime currentTime = result.getFinishTime(); // The time of this rider to compare against.
236       double currentTimeSeconds =
237           (currentTime.getHour() * 3600) + (currentTime.getMinute() * 60) + currentTime.getSecond();
238       double previousTimeSeconds = (previousTime.getHour() * 3600) + (previousTime.getMinute() * 60
239           + previousTime.getSecond());
240       // Converts LocalTime into double to calculate whether there is a 1 second or less gap between
241       // riders.
242       if (((currentTimeSeconds - previousTimeSeconds) <= 1.0) && (previousTimeSeconds != 0)) {
243         result.setAdjustedFinishTime(pelotonLeader); // If there is a 1 second or less gap the
244         // adjusted time is set to the peloton leader.
245       } else {
246         pelotonLeader = currentTime; // If the rider is not in a peloton then they could be the next
247         // peloton's leader.
248         result.setAdjustedFinishTime(currentTime); // Adjusted time is set to their finish time.
249       }
250       previousTime = currentTime; // A Peloton can consist of multiple riders so the one second gap
251       // accounts for between riders.
252     }
253     Collections.sort(this.results); // The results are sorted by finishing times (same order as
254     // adjusted).
255   }
256
257   /**
258    * Iterates through all the segments in the stage to create RiderSegmentResult objects from the
259    * results in RiderStageResults.
260    */
261   public void generateRiderSegmentResults() {
262     Collections.sort(this.segmentsInStage); // Order segments by their location.
263     int segmentCounter = 1; // Starts from 1 as initial
```

```
263 time is start time.
264     for (Segment segment : this.segmentsInStage) { //
    Iterates through every segment.
265       ArrayList<RiderSegmentResult> riderSegmentResults
     = new ArrayList<>();
266       // Stores segment results then sorts.
267       for (RiderStageResult result : results) {
268         RiderSegmentResult newRiderSegmentResult = new
269             RiderSegmentResult(result.getTimes()[
    segmentCounter], result.getRiderId());
270         riderSegmentResults.add(newRiderSegmentResult);
271         assert riderSegmentResults.contains(
    newRiderSegmentResult);
272       }
273       // Sorts riders segment results in order of their
    time in ascending order.
274       Collections.sort(riderSegmentResults);
275       // Stores these results in segment for more
    intuitive access.
276       segment.setResults(riderSegmentResults);
277
278       // Sets rank of each rider in segment.
279       int rank = 0;
280       for (RiderSegmentResult result :
    riderSegmentResults) {
281         result.setRank(rank);
282         rank++;
283       }
284     }
285     assert (this.segmentsInStage.isEmpty()) || (results.
    isEmpty()) || !this.segmentsInStage.get(0)
286         .getResults().isEmpty();
287   }
288
289   /**
290    * Calculates either mountain or point classification
    points for a stage and stores them in each
291    * rider's stage result object.
292    *
293    * @param isMountain True if mountain classification,
    false if point classification.
294    * @return An array of rider ID's ordered by their
    points in the given classification.
295    */
```

File - Stage.java

```
296   public int[] generatePointsInStage(boolean isMountain
    ) {
297
298     Map<Enum, int[]> pointsConversion;
299
300     if (isMountain) {
301       // Table of rank to points for each segment type in
     mountain classification.
302       int[] pointsHC = {20, 15, 12, 10, 8, 6, 4, 2}; //
    TOP 8 ONLY.
303       int[] pointsC1 = {10, 8, 6, 4, 2, 1, 0, 0}; // TOP
    6 ONLY.
304       int[] pointsC2 = {5, 3, 2, 1, 0, 0, 0, 0}; // TOP 4
     ONLY.
305       int[] pointsC3 = {2, 1, 0, 0, 0, 0, 0, 0}; // TOP 2
     ONLY.
306       int[] pointsC4 = {1, 0, 0, 0, 0, 0, 0, 0}; // TOP 1
     ONLY.
307       pointsConversion = Map.of(
308           SegmentType.HC, pointsHC,
309           SegmentType.C1, pointsC1,
310           SegmentType.C2, pointsC2,
311           SegmentType.C3, pointsC3,
312           SegmentType.C4, pointsC4);
313     } else {
314       // Table of rank to points for each stage type in
    point classification.
315       int[] flatPointsConversion = {50, 30, 20, 18, 16,
    14, 12, 10, 8, 7, 6, 5, 4, 3, 2};
316       int[] hillyPointsConversion = {30, 25, 22, 19, 17,
    15, 13, 11, 9, 7, 6, 5, 4, 3, 2};
317       int[] mountainPointsConversion = {20, 17, 15, 13,
    11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
318       pointsConversion = Map.of(
319           StageType.FLAT, flatPointsConversion,
320           StageType.MEDIUM_MOUNTAIN,
    hillyPointsConversion,
321           StageType.HIGH_MOUNTAIN,
    mountainPointsConversion,
322           StageType.TT, mountainPointsConversion // Time
    trials are awarded the same points as high
323           // mountain stages.
324       );
325     }
```

```
326
327     if (isMountain) {
328       // Do mountain error checks.
329       if ((this.getStageType() == StageType.TT) || (this.
    getSegmentsInStage().size() == 0)) {
330         // No mountain points (no segments).
331         return new int[0];
332       }
333
334       // Check if there even are any climbs.
335       boolean climbFound = false;
336       for (Segment segment : this.getSegmentsInStage()) {
337         if (segment.getSegmentType() != SegmentType.
    SPRINT) {
338           climbFound = true;
339         }
340       }
341       // No climbs so there will be no results.
342       if (!climbFound) {
343         return new int[0];
344       }
345     }
346
347     HashMap<Integer, Integer> riderIdsToPoints = new
    HashMap<Integer, Integer>();
348     // Sum of points for each rider for the specified
    race.
349
350     // For the points classification, add on the points
    from finish times which mountain doesn't
351     // include.
352     if (!isMountain) {
353       int pointsIndex = 0;
354       this.generateAdjustedResults(); // Sorts times in
    ascending order.
355
356       for (RiderStageResult result : this.getResults
    ()) { // Iterate through rider.
357         int riderId = result.getRiderId();
358         int points;
359         if (pointsIndex < 15) { // Only first 15 riders
    are awarded points.
360           points = pointsConversion.get(this.getStageType
    ())[pointsIndex]; // Points for stage type.
```

```
361           assert points > 0 : "Incorrect points assigned
    !";
362         } else {
363           points = 0;
364         }
365         riderIdsToPoints.put(riderId, points);
366         result.setPoints(points);
367         pointsIndex++;
368       }
369     }
370
371     // Add on points from segments.
372     ArrayList<Segment> segmentsInStage = this.
    getSegmentsInStage();
373     this.generateRiderSegmentResults(); // Creates
    segment objects.
374     for (Segment segment : segmentsInStage) {
375       // Cast for the appropriate Enum for mountain or
    points as a key for their points conversion
376       // table.
377       SegmentType currentSegmentType = segment.
    getSegmentType();
378       Enum mapKey;
379       if (isMountain) {
380         mapKey = currentSegmentType;
381       } else {
382         mapKey = StageType.HIGH_MOUNTAIN; // Intermediate
     sprints are the only type of segment in
383         // points classification and are awarded the same
     points as high mountain stages.
384       }
385
386       // Only allow intermediate sprint + points
    classification or categorised climb + mountain
387       // classification combinations.
388
389       if (((currentSegmentType == SegmentType.SPRINT
    ) && !isMountain)
390           || (!(currentSegmentType == SegmentType.SPRINT
    ) && isMountain)) {
391         int i = 0;
392         for (RiderSegmentResult riderResult : segment.
    getResults()) { // Iterates through each
393           // rider's segment results.
```

```
394           int riderId = riderResult.getRiderId();
395           int riderRank = riderResult.getRank(); // Gets
    riders rank in segment.
396           int points;
397           int[] rowOfPointsConversion = pointsConversion.
    get(mapKey);
398           int pointsLimit = rowOfPointsConversion.length;
399           if (riderRank < pointsLimit) {
400             points = rowOfPointsConversion[riderRank];
    // Points limit prevent index out of bounds
401             // error.
402             assert points >= 0 : "Negative point value
    assigned!";
403           } else {
404             points = 0;
405           }
406
407           if (riderIdsToPoints.get(riderId) == null) {
    // If the rider is not registered with points
408             // add them.
409             riderIdsToPoints.put(riderId, points);
410             if (!isMountain) {
411               this.getResults().get(i).setPoints(points);
412             } else {
413               this.getResults().get(i).setMountainPoints(
    points);
414             }
415           } else { // Else sum their points to their
    total points.
416             riderIdsToPoints.merge(riderId, points,
    Integer::sum);
417             if (!isMountain) {
418               this.getResults().get(i).addPoints(points);
419             } else {
420               this.getResults().get(i).addMountainPoints(
    points);
421             }
422           }
423           i++;
424         }
425       }
426     }
427     Collections.sort(this.results);
428     // Creates an int array of points ordered by rank
```

```
429     if (!(riderIdsToPoints.isEmpty())) {
430       int[] pointsOrderedByRank = new int[this.results.
    size()];
431       int i = 0;
432       for (RiderStageResult result : this.getResults()) {
433         if (isMountain) {
434           pointsOrderedByRank[i] = result.
    getMountainPoints();
435         } else {
436           pointsOrderedByRank[i] = result.getPoints();
437         }
438         i++;
439       }
440
441       return pointsOrderedByRank;
442     } else {
443       return new int[0];
444     }
445   }
446 }
447
```

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 generateRidersRaceFinishTimes() {
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        for (RiderStageResult riderStageResult : stage.
    getResults()) { // Iterates through all results
 94          // in a stage.
 95          int riderId = riderStageResult.getRiderId();
 96          boolean riderFound = false;
 97          for (RiderRaceResult raceResult : this.
    raceResults) { // Iterates through race results.
 98            if (raceResult.getRiderId() == riderId) { // If
     the race result and stage result refer to
 99              // the same rider.
100              arrayOfTotalTimes[i] = SumLocalTimes.
    addLocalTimes(arrayOfTotalTimes[i],
101                  riderStageResult.getFinishTime());
102              // Sums race results finish time with new
    stages finish time.
103              riderFound = true;
104              break;
105            }
106          }
107          if (!riderFound) { // If no race result for rider
     one is made.
108            RiderRaceResult raceResult = new
    RiderRaceResult(riderStageResult.getRiderId(),
109                this.raceId);
110            arrayOfTotalTimes[i] = riderStageResult.
    getFinishTime();
111            int raceResultsSize = this.raceResults.size();
112            this.raceResults.add(raceResult);
113            assert this.raceResults.size() ==
    raceResultsSize + 1 : "Race result not added to array"
114                + "list!";
115          }
116          i++;
```

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

File - StagedRace.java

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

```
183         }
184         i++;
185       }
186     }
187     // Sum of points for each rider for the specified
    race.
188     int resultsSize = this.raceResults.size();
189     int[] racePoints = new int[resultsSize];
190     int i = 0;
191
192     // Sorts race results by either their mountain or
    point classification points.
193     if (isMountain) {
194       Collections.sort(raceResults, RiderRaceResult::
    compareByMountainPoints);
195     } else {
196       Collections.sort(raceResults, RiderRaceResult::
    compareByPoints);
197     }
198     for (RiderRaceResult result : this.raceResults) {
199       if (isMountain) {
200         racePoints[i] = result.getMountainPoints();
201       } else {
202         racePoints[i] = result.getPoints();
203       }
204       i++;
205     }
206     Collections.sort(raceResults); // Sorts race results
    back to its order by finishing times.
207     return racePoints;
208   }
209
210   /**
211    * @param isMountain True if mountain point
    classification, false if point classification.
212    * @return An array of rider ID's ordered by their
    points in the given classification.
213    */
214   public int[] getRiderIdsOrderedByPoints(boolean
    isMountain) {
215     ArrayList<RiderRaceResult> riderRaceResults = this.
    getResults();
216     // Sorts race results by their points in the given
    classification.
```

```
217     if (isMountain) {
218       Collections.sort(riderRaceResults, RiderRaceResult
    ::compareByMountainPoints);
219     } else {
220       Collections.sort(riderRaceResults, RiderRaceResult
    ::compareByPoints);
221     }
222 
223     // Gets the rider ID's in order of points and adds
    them to an array.
224     int[] riderIdsByPoints = new int[riderRaceResults.
    size()];
225     int i = 0;
226     for (RiderRaceResult result : riderRaceResults) {
227       riderIdsByPoints[i] = result.getRiderId();
228       i++;
229     }
230     Collections.sort(raceResults); // Sorts race results
    back to its order by finishing times.
231     assert riderRaceResults.size() == riderIdsByPoints.
    length : "Gathered rider IDs incorrectly!";
232     return riderIdsByPoints;
233   }
234 }
235 
```

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
```

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.
```

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