```java
package cycling;

/**
 * This enum is used to represent the stage types  on road races.
 *
 * @author Diogo Pacheco
 * @version 1.0
 *
 */
public enum StageType {

    /**
     * Used for mostly flat stages.
     */
    FLAT,

    /**
     * Used for hilly finish or stages with moderate amounts of mountains.
     */
    MEDIUM_MOUNTAIN,

    /**
     * Used for high mountain finish or stages with multiple categorised climbs.
     */
    HIGH_MOUNTAIN,

    /**
     * Used for time trials.
     */
    TT;

}

```

```java
package cycling;

/**
 * This enum is used to represent the segment types within stages on road races.
 *
 * @author Diogo Pacheco
 * @version 1.0
 *
 */
public enum SegmentType {

    /**
     * An intermediate sprint.
     */
    SPRINT,

    /**
     * A categorised 4 climb. The easiest categorised climbs of all, under 2km long
     * with an average grade of around 5% or 2-3% up to 5km long.
     */
    C4,

    /**
     * A categorised 3 climb. This could be a climb as short as 1km with a steep
     * gradient of about 10% or a mellower climb up to 10km long with up to a 5%
     * gradient.
     */
    C3,

    /**
     * A categorised 2 climb. Category 2 could be a short climb, for example 5km at
     * 8 percent, or as long as 15km at 4. percent
     */
    C2,

```

```
36      /**
37       * A categorised 1 climb. Still a very
   significant climb, it could be a big
38       * mountain climb with a lesser gradient or a
   shorter climb with a steep pitch,
39       * for example 8km at 8% through to 20km at 5%.
40       */
41      C1,
42
43      /**
44       * From the French term "Hors Categorie" (HC)
   meaning beyond categorisation. The
45       * toughest of the tough. The longest or steepest
    climbs, often both combined.
46       */
47      HC;
48  }
49
```

```java
package cycling;

import cycling.types.*;

import java.io.*;
import java.time.Duration;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;
import java.util.*;

public class CyclingPortal implements CyclingPortalInterface {
    // Data Store
    private ArrayList<Team>  teams = new ArrayList<>();
    private ArrayList<Rider> riders = new ArrayList<>();

    private ArrayList<Race> races = new ArrayList<>();
    private ArrayList<Stage> stages = new ArrayList<>();
    private ArrayList<Segment> segments = new ArrayList<>();

    // Private helper functions
    private Race getRaceByIDOrNull(int raceId) {
        return races.stream()
                .filter(r -> r.id == raceId)
                .findAny().orElse(null);
    }

    private Team getTeamByIDOrNull(int teamId){
        return teams.stream()
                .filter(t -> t.id == teamId)
                .findAny().orElse(null);
    }

    private Stage getStageByIDOrNull(int stageId){
        return stages.stream()
```

```
36                  .filter(s -> s.id == stageId)
37                  .findAny().orElse(null);
38      }
39
40      private Rider getRiderByIDOrNull(int riderId) {
41          return riders.stream()
42                  .filter(r -> r.id == riderId)
43                  .findAny().orElse(null);
44      }
45
46      private Segment getSegmentByIDOrNull(int
    segmentId) {
47          return segments.stream()
48                  .filter(s -> s.id == segmentId)
49                  .findAny().orElse(null);
50      }
51
52      private int stagePoints(StageType stageType, int
    rank) {
53          if (rank > 15) return 0;
54          return switch (stageType) {
55              case FLAT ->                 new int[]{50
    , 30, 20, 18, 16, 14, 12, 10, 8, 7, 6, 5, 4, 3, 2}[
    rank];
56              case MEDIUM_MOUNTAIN ->     new int[]{30
    , 25, 22, 19, 17, 15, 13, 11, 9, 7, 6, 5, 4, 3, 2}[
    rank];
57              case HIGH_MOUNTAIN, TT ->   new int[]{20
    , 17, 15, 13, 11, 10,  9,  8, 7, 6, 5, 4, 3, 2, 1}[
    rank];
58          };
59      }
60
61      private int segmentPoints(SegmentType segmentType
    , int rank) {
62          if (rank > 15) return 0;
63          return switch (segmentType) {
64              case SPRINT ->  new int[]{20, 17, 15, 13
    , 11, 10,  9,  8, 7, 6, 5, 4, 3, 2, 1}[rank];
65              case HC ->      new int[]{20, 15, 12, 10
    ,  8,  6,  4,  2, 0, 0, 0, 0, 0, 0, 0}[rank];
```

```
66              case C1 ->      new int[]{10,  8,  6,  4
    ,  2,  1,  0,  0, 0, 0, 0, 0, 0, 0, 0, 0}[rank];
67              case C2 ->      new int[]{ 5,  3,  2,  1
    ,  0,  0,  0,  0, 0, 0, 0, 0, 0, 0, 0, 0}[rank];
68              case C3 ->      new int[]{ 2,  1,  0,  0
    ,  0,  0,  0,  0, 0, 0, 0, 0, 0, 0, 0, 0}[rank];
69              case C4 ->      new int[]{ 1,  0,  0,  0
    ,  0,  0,  0,  0, 0, 0, 0, 0, 0, 0, 0, 0}[rank];
70          };
71      }
72
73      /**
74       * The method removes the race and all its
    related information, i.e., stages,
75       * segments, and results.
76       * <p>
77       * The state of this MiniCyclingPortalInterface
    must be unchanged if any
78       * exceptions are thrown.
79       *
80       * @param name The name of the race to be
    removed.
81       * @throws NameNotRecognisedException If the
    name does not match to any race in
82       *                                      the system
    .
83       */
84      @Override
85      public void removeRaceByName(String name) throws
     NameNotRecognisedException {
86          Race race = races.stream()
87                  .filter(r -> r.name.equals(name))
88                  .findAny()
89                  .orElse(null);
90          if (race == null) throw new
    NameNotRecognisedException("Race not found in
    removeRaceByName");
91
92          races.remove(race);
93      }
94
```

```
 95     /**
 96      * Get the general classification rank of riders in a race.
 97      * <p>
 98      * The state of this MiniCyclingPortalInterface must be unchanged if any
 99      * exceptions are thrown.
100      *
101      * @param raceId The ID of the race being queried.
102      * @return A ranked list of riders' IDs sorted ascending by the sum of their
103      * adjusted elapsed times in all stages of the race. That is, the first
104      * in this list is the winner (least time). An empty list if there is no
105      * result for any stage in the race.
106      * @throws IDNotRecognisedException If the ID does not match any race in the
107      *                                  system.
108      */
109     @Override
110     public int[] getRidersGeneralClassificationRank(int raceId) throws IDNotRecognisedException {
111         int[] stageIds = getRaceStages(raceId);
112         // Value is the adjusted and  accumulated elapsed times from all stages in the race that the rider takes part in.
113         HashMap<Rider, Long> results = new HashMap<>();
114 
115         for (int stageId : stageIds) {
116             Stage stage = getStageByIDOrNull(stageId);
117             if (stage == null) throw new IDNotRecognisedException();
118 
119             Rider[] applicableRiders = riders.stream().filter(r -> r.results.containsKey(stage.id)).toArray(Rider[]::new);
120 
```

```
121             // Riders with their adjusted elapsed
    time in a stage
122             for (Rider rider : applicableRiders) {
123                 long val =
    getRiderAdjustedElapsedTimeInStage(stageId, rider.id
    ).toSecondOfDay();
124                 if (results.containsKey(rider))
    results.put(rider, results.get(rider) + val);
125                 else results.put(rider, val);
126             }
127         }
128 
129         HashMap<Long, Rider> resultsHMFlipped = new
    HashMap<>();
130         results.forEach((rider, val) ->
    resultsHMFlipped.put(val, rider));
131         return hashMapValuesSortedByComparableKey(
    resultsHMFlipped).stream().mapToInt(r -> r.id).
    toArray();
132     }
133 
134     /**
135      * Get the general classification times of
    riders in a race.
136      * <p>
137      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
138      * exceptions are thrown.
139      *
140      * @param raceId The ID of the race being
    queried.
141      * @return A list of riders' times sorted by the
     sum of their adjusted elapsed
142      * times in all stages of the race. An empty
    list if there is no result
143      * for any stage in the race. These times should
     match the riders
144      * returned by {@link #
    getRidersGeneralClassificationRank(int)}.
145      * @throws IDNotRecognisedException If the ID
    does not match any race in the
```

```
146      *                                          system.
147      */
148     @Override
149     public LocalTime[]
    getGeneralClassificationTimesInRace(int raceId)
    throws IDNotRecognisedException {
150         // Make sure that I'm returning the correct
    type of times
151         // I am: https://vle.exeter.ac.uk/mod/forum/
    discuss.php?d=227212
152 
153         int[] stageIds = getRaceStages(raceId);
154 
155         int[] riderIds =
    getRidersGeneralClassificationRank(raceId);
156         Rider[] riders = Arrays.stream(riderIds).
    mapToObj(this::getRiderByIDOrNull).toArray(Rider[]::
    new);
157         if (Arrays.stream(riders).anyMatch(Objects::
    isNull)) throw new IDNotRecognisedException();
158 
159         return Arrays.stream(riders).map(r -> Arrays
    .stream(stageIds)
160                 .mapToObj(sid -> {
161                     try {
162                         return
    getRiderAdjustedElapsedTimeInStage(sid, r.id);
163                     } catch (
    IDNotRecognisedException e) {
164                         return LocalTime.
    MIDNIGHT;
165                     }
166                 })
167                 .filter(time -> time !=
    LocalTime.MIDNIGHT)
168                 .reduce(LocalTime.MIDNIGHT, (
    subtotal, time) -> LocalTime.MIDNIGHT.plus(
169                         Duration.between(
    LocalTime.MIDNIGHT, time).plus(Duration.between(
    LocalTime.MIDNIGHT, subtotal))
170                 ))
```

```
171         ).toArray(LocalTime[]::new);
172     }
173 
174     /**
175      * Get the overall points of riders in a race.
176      * <p>
177      * The state of this MiniCyclingPortalInterface must be unchanged if any
178      * exceptions are thrown.
179      *
180      * @param raceId The ID of the race being queried.
181      * @return A list of riders' points (i.e., the sum of their points in all stages
182      * of the race), sorted by the total elapsed time. An empty list if
183      * there is no result for any stage in the race. These points should
184      * match the riders returned by {@link #getRidersGeneralClassificationRank(int)}.
185      * @throws IDNotRecognisedException If the ID does not match any race in the
186      *                                  system.
187      */
188     @Override
189     public int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException {
190         int[] rankedRiderIds = getRidersGeneralClassificationRank(raceId); // Order of rider ids to match
191 
192         Race race = getRaceByIDOrNull(raceId);
193         if (race == null) throw new IDNotRecognisedException();
194         int[] stageIds = getRaceStages(race.id);
195         Stage[] stages = Arrays.stream(stageIds).mapToObj(this::getStageByIDOrNull).toArray(Stage[]::new);
196         if (Arrays.stream(stages).anyMatch(Objects::isNull)) throw new IDNotRecognisedException();
197 
```

```
198         // Key riderId. Val Points
199         HashMap<Integer, Integer> riderPoints = new
    HashMap<>();
200         for (Stage stage : stages) {
201             int[] ridersInStage =
    getRidersRankInStage(stage.id);
202             int[] pointsInStage =
    getRidersPointsInStage(stage.id);
203 
204             for (int i = 0; i < ridersInStage.length
    ; i++) {
205                 int rid = ridersInStage[i];
206                 if (!riderPoints.containsKey(rid))
    riderPoints.put(rid, 0);
207                 riderPoints.put(rid, riderPoints.get
    (rid) + pointsInStage[i]);
208             }
209         }
210 
211         ArrayList<Integer> orderedPoints = new
    ArrayList<>(); // Ordered by rankedRiderIds
212         for (int rid : rankedRiderIds) {
213             orderedPoints.add(riderPoints.get(rid));
214         }
215 
216         return orderedPoints.stream().mapToInt(i ->
    i).toArray();
217     }
218 
219     /**
220      * Get the overall mountain points of riders in
    a race.
221      * <p>
222      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
223      * exceptions are thrown.
224      *
225      * @param raceId The ID of the race being
    queried.
226      * @return A list of riders' mountain points (i.
    e., the sum of their mountain
```

```
227         * points in all stages of the race), sorted by
    the total elapsed time.
228         * An empty list if there is no result for any
    stage in the race. These
229         * points should match the riders returned by
230         * {@link #getRidersGeneralClassificationRank(
    int)}.
231         * @throws IDNotRecognisedException If the ID
    does not match any race in the
232         *                                  system.
233         */
234        @Override
235        public int[] getRidersMountainPointsInRace(int
    raceId) throws IDNotRecognisedException {
236            int[] rankedRiderIds =
    getRidersGeneralClassificationRank(raceId); // Order
     of rider ids to match
237
238            Race race = getRaceByIDOrNull(raceId);
239            if (race == null) throw new
    IDNotRecognisedException();
240            int[] stageIds = getRaceStages(race.id);
241            Stage[] stages = Arrays.stream(stageIds).
    mapToObj(this::getStageByIDOrNull).toArray(Stage[]::
    new);
242            if (Arrays.stream(stages).anyMatch(Objects::
    isNull)) throw new IDNotRecognisedException();
243
244            // Key riderId. Val Points
245            HashMap<Integer, Integer> riderPoints = new
    HashMap<>();
246            for (Stage stage : stages) {
247                int[] ridersInStage =
    getRidersRankInStage(stage.id);
248                int[] pointsInStage =
    getRidersMountainPointsInStage(stage.id);
249
250                for (int i = 0; i < ridersInStage.length
    ; i++) {
251                    int rid = ridersInStage[i];
252                    if (!riderPoints.containsKey(rid))
```

```
252 riderPoints.put(rid, 0);
253                 riderPoints.put(rid, riderPoints.get
    (rid) + pointsInStage[i]);
254             }
255         }
256 
257         ArrayList<Integer> orderedPoints = new
    ArrayList<>(); // Ordered by rankedRiderIds
258         for (int rid : rankedRiderIds) {
259             orderedPoints.add(riderPoints.get(rid));
260         }
261 
262         return orderedPoints.stream().mapToInt(i ->
    i).toArray();
263     }
264 
265     /**
266      * Get the ranked list of riders based on the
    point classification in a race.
267      * <p>
268      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
269      * exceptions are thrown.
270      *
271      * @param raceId The ID of the race being
    queried.
272      * @return A ranked list of riders' IDs sorted
    descending by the sum of their
273      * points in all stages of the race. That is,
    the first in this list is
274      * the winner (more points). An empty list if
    there is no result for any
275      * stage in the race.
276      * @throws IDNotRecognisedException If the ID
    does not match any race in the
277      *                                  system.
278      */
279     @Override
280     public int[] getRidersPointClassificationRank(
    int raceId) throws IDNotRecognisedException {
281         ArrayList<Integer> rankedRiderIds = new
```

```
281 ArrayList<>(Arrays.stream(
    getRidersGeneralClassificationRank(raceId)).boxed().
    toList());
282         ArrayList<Integer> rankedRidersPoints = new
    ArrayList<>(Arrays.stream(getRidersPointsInRace(
    raceId)).boxed().toList());
283         // Same as HashMap just different interface
284         rankedRiderIds.sort(Comparator.comparingInt(
    rankedRidersPoints::indexOf)); // Check if ide
    optimisations are good
285         return rankedRiderIds.stream().mapToInt(i
     -> i).toArray();
286     }
287
288     /**
289      * Get the ranked list of riders based on the
    mountain classification in a race.
290      * <p>
291      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
292      * exceptions are thrown.
293      *
294      * @param raceId The ID of the race being
    queried.
295      * @return A ranked list of riders' IDs sorted
    descending by the sum of their
296      * mountain points in all stages of the race.
    That is, the first in this
297      * list is the winner (more points). An empty
    list if there is no result
298      * for any stage in the race.
299      * @throws IDNotRecognisedException If the ID
    does not match any race in the
300      *                                  system.
301      */
302     @Override
303     public int[]
    getRidersMountainPointClassificationRank(int raceId
    ) throws IDNotRecognisedException {
304         int[] raceStageIds = getRaceStages(raceId);
305         Stage[] raceStages = Arrays.stream(
```

```
305 raceStageIds).mapToObj(this::getStageByIDOrNull).
    toArray(Stage[]::new);
306         if (Arrays.stream(raceStages).anyMatch(
    Objects::isNull)) throw new IDNotRecognisedException
    ();
307
308         // Key riderId. Val Points. Points are
    accumulated in the loop
309         HashMap<Integer, Integer> riderPointsInRace
     = new HashMap<>();
310         for (Stage stage : raceStages) {
311             int[] riderIdsRanked =
    getRidersRankInStage(stage.id);
312             int[] riderPoints =
    getRidersMountainPointsInStage(stage.id);
313             assert riderIdsRanked.length ==
    riderPoints.length; // If this isn't true then
    something is catastrophically wrong
314
315             for (int i = 0; i < riderIdsRanked.
    length; i++) {
316                 int rid = riderIdsRanked[i];
317                 if (!riderPointsInRace.containsKey(
    rid)) riderPointsInRace.put(rid, 0);
318                 riderPointsInRace.put(
319                         riderIdsRanked[i],
320                         riderPointsInRace.get(rid
    ) + riderPoints[i]
321                 );
322             }
323         }
324
325         ArrayList<Integer> riderIds = new ArrayList
    <>(riderPointsInRace.keySet().stream().toList());
326         ArrayList<Integer> riderPoints = new
    ArrayList<>(riderPointsInRace.values().stream().
    toList());
327
328         riderIds.sort(Comparator.comparingInt(
    riderPoints::indexOf));
329
```

```
330         return riderIds.stream().mapToInt(i -> i).
    toArray();
331     }
332 
333     /**
334      * Get the races currently created in the
    platform.
335      *
336      * @return An array of race IDs in the system or
     an empty array if none exists.
337      */
338     @Override
339     public int[] getRaceIds() {
340         return races.stream().mapToInt(r -> r.id).
    toArray();
341     }
342 
343     /**
344      * The method creates a staged race in the
    platform with the given name and
345      * description.
346      * <p>
347      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
348      * exceptions are thrown.
349      *
350      * @param name        Race's name.
351      * @param description Race's description (can be
     null).
352      * @return the unique ID of the created race.
353      * @throws IllegalNameException If the name
    already exists in the platform.
354      * @throws InvalidNameException If the name is
    null, empty, has more than 30
355      *                              characters, or
    has white spaces.
356      */
357     @Override
358     public int createRace(String name, String
    description) throws IllegalNameException,
    InvalidNameException {
```

```
359         if (name == null || name.equals("") || name.
    length() > 30) throw new InvalidNameException();
360
361         int currentMaxRaceID = 0;
362         for (Race r : races) {
363             if (r.id > currentMaxRaceID)
    currentMaxRaceID = r.id;
364             if (r.name.equals(name)) throw new
    IllegalNameException();
365         }
366
367         Race newRace = new Race();
368         newRace.id = currentMaxRaceID +1;
369         newRace.name = name;
370         newRace.description = description;
371         races.add(newRace);
372         return newRace.id;
373     }
374
375     /**
376      * Get the details from a race.
377      * <p>
378      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
379      * exceptions are thrown.
380      *
381      * @param raceId The ID of the race being
    queried.
382      * @return Any formatted string containing the
    race ID, name, description, the
383      * number of stages, and the total length (i.e
    ., the sum of all stages'
384      * length).
385      * @throws IDNotRecognisedException If the ID
    does not match to any race in the
386      *                                     system.
387      */
388     @Override
389     public String viewRaceDetails(int raceId) throws
     IDNotRecognisedException {
390         Race race = getRaceByIDOrNull(raceId);
```

```
391             if (race == null) throw new
    IDNotRecognisedException("Race not found in
    viewRaceDetails");
392             int raceLength = Arrays.stream(getRaceStages
    (raceId)).sum();
393             return("Race ID: " + race.id + ", Race
    description: " + race.description + ", Number of
    stages: " + race.stages.size() + ", Length of race:"
     + raceLength);
394         }
395
396         /**
397          * The method removes the race and all its
    related information, i.e., stages,
398          * segments, and results.
399          * <p>
400          * The state of this MiniCyclingPortalInterface
    must be unchanged if any
401          * exceptions are thrown.
402          *
403          * @param raceId The ID of the race to be
    removed.
404          * @throws IDNotRecognisedException If the ID
    does not match to any race in the
405          *                                     system.
406          */
407         @Override
408         public void removeRaceById(int raceId) throws
    IDNotRecognisedException {
409             Race item = getRaceByIDOrNull(raceId);
410             if (item == null) throw new
    IDNotRecognisedException("Race not found in
    removeRaceByID");
411
412
413             int[] raceStages = getRaceStages(raceId);
414             for (int i : raceStages){
415                 int[] stageSegments = getStageSegments(i
    );
416                 for (int j : stageSegments){
417                     segments.remove(j);
```

```
418             }
419             Rider[] applicableRiders = riders.stream
    ().filter(r -> r.results.containsKey(i)).toArray(
    Rider[]::new);
420             for (Rider k : applicableRiders){
421                 deleteRiderResultsInStage(i, k.id);
422             }
423             stages.remove(i);
424         }
425         races.remove(item);
426     }
427 
428     /**
429      * The method queries the number of stages
    created for a race.
430      * <p>
431      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
432      * exceptions are thrown.
433      *
434      * @param raceId The ID of the race being
    queried.
435      * @return The number of stages created for the
    race.
436      * @throws IDNotRecognisedException If the ID
    does not match to any race in the
437      *                                  system.
438      */
439     @Override
440     public int getNumberOfStages(int raceId) throws
    IDNotRecognisedException {
441         Race race = getRaceByIDOrNull(raceId);
442         if (race == null) throw new
    IDNotRecognisedException("Race not found in
    removeRaceByID");
443 
444         return race.stages.size();
445     }
446 
447     /**
448      * Creates a new stage and adds it to the race.
```

```
449      * <p>
450      * The state of this MiniCyclingPortalInterface must be unchanged if any
451      * exceptions are thrown.
452      *
453      * @param raceId      The race which the stage will be added to.
454      * @param stageName   An identifier name for the stage.
455      * @param description A descriptive text for the stage.
456      * @param length      Stage length in kilometres.
457      * @param startTime   The date and time in which the stage will be raced. It
458      *                    cannot be null.
459      * @param type        The type of the stage. This is used to determine the
460      *                    amount of points given to the winner.
461      * @return the unique ID of the stage.
462      * @throws IDNotRecognisedException If the ID does not match to any race in the
463      *                                  system.
464      * @throws IllegalNameException     If the name already exists in the platform.
465      * @throws InvalidNameException     If the new name is null, empty, has more
466      *                                  than 30.
467      * @throws InvalidLengthException   If the length is less than 5km.
468      */
469     @Override
470     public int addStageToRace(int raceId, String stageName, String description, double length, LocalDateTime startTime, StageType type) throws IDNotRecognisedException, IllegalNameException, InvalidNameException, InvalidLengthException {
471         Race race = getRaceByIDOrNull(raceId);
472
473         if (race == null) throw new
```

```
473 IDNotRecognisedException();
474         if (stageName == null || stageName.equals("") || stageName.length() > 30) throw new InvalidNameException("Name is null, empty or has more than 30 characters");
475         if (stages.stream().anyMatch(s -> Objects.equals(s.name, stageName))) throw new IllegalNameException();
476         if (length < 5) throw new InvalidLengthException();
477 
478         Stage newStage = new Stage();
479         newStage.name = stageName; newStage.description = description; newStage.length = length;  newStage.type = type; newStage.startTime = startTime;
480         stages.add(new Stage());
481 
482         int currentMaxStageID = stages.stream().mapToInt(s -> s.id).max().orElse(0);
483         newStage.id = currentMaxStageID + 1;
484 
485         race.stages.add(newStage.id);
486         stages.add(newStage);
487 
488         return newStage.id;
489     }
490 
491     /**
492      * Retrieves the list of stage IDs of a race.
493      * <p>
494      * The state of this MiniCyclingPortalInterface must be unchanged if any
495      * exceptions are thrown.
496      *
497      * @param raceId The ID of the race being queried.
498      * @return The list of stage IDs ordered (from first to last) by their sequence in the
499      * race.
500      * @throws IDNotRecognisedException If the ID
```

```
500 does not match to any race in the
501      *                                     system.
502      */
503     @Override
504     public int[] getRaceStages(int raceId) throws
    IDNotRecognisedException {
505         Race race = getRaceByIDOrNull(raceId);
506         if (race == null) throw new
    IDNotRecognisedException();
507 
508 
509         return race.stages.stream().mapToInt(i->i).
    toArray();
510     }
511 
512     /**
513      * Gets the length of a stage in a race, in
    kilometres.
514      * <p>
515      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
516      * exceptions are thrown.
517      *
518      * @param stageId The ID of the stage being
    queried.
519      * @return The stage's length.
520      * @throws IDNotRecognisedException If the ID
    does not match to any stage in the
521      *                                     system.
522      */
523     @Override
524     public double getStageLength(int stageId) throws
     IDNotRecognisedException {
525         Stage stage = getStageByIDOrNull(stageId);
    if (stage == null) throw new
    IDNotRecognisedException();
526 
527         return stage.length;
528     }
529 
530     /**
```

```
531      * Removes a stage and all its related data, i.e
    ., segments and results.
532      * <p>
533      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
534      * exceptions are thrown.
535      *
536      * @param stageId The ID of the stage being
    removed.
537      * @throws IDNotRecognisedException If the ID
    does not match to any stage in the
538      *                                  system.
539      */
540     @Override
541     public void removeStageById(int stageId) throws
    IDNotRecognisedException {
542         Stage stage = getStageByIDOrNull(stageId);
543         if (stage == null) throw new
    IDNotRecognisedException();
544 
545         Race race = races.stream().filter(r -> r.
    stages.contains(stage.id)).findAny().orElse(null);
546         if (race == null) {System.out.println("
    Consistency error. Corrupted data?"); return;}
547 
548         race.stages.remove(race.id - 1);
549         stages.remove(stageId -1);
550         int[] stageSegments = getStageSegments(
    stageId);
551         for (int i : stageSegments){
552             segments.remove(i);
553         }
554         Rider[] applicableRiders = riders.stream().
    filter(r -> r.results.containsKey(stageId)).toArray(
    Rider[]::new);
555         for (Rider k : applicableRiders){
556             deleteRiderResultsInStage(stageId, k.id
    );
557         }
558     }
559 
```

```
560     /**
561      * Adds a climb segment to a stage.
562      * <p>
563      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
564      * exceptions are thrown.
565      *
566      * @param stageId         The ID of the stage to
     which the climb segment is
567      *                        being added.
568      * @param location        The kilometre location
     where the climb finishes within
569      *                        the stage.
570      * @param type            The category of the
    climb - {@link SegmentType#C4},
571      *                        {@link SegmentType#C3
    }, {@link SegmentType#C2},
572      *                        {@link SegmentType#C1
    }, or {@link SegmentType#HC}.
573      * @param averageGradient The average gradient
    for the climb.
574      * @param length          The length of the
    climb in kilometre.
575      * @return The ID of the segment created.
576      * @throws IDNotRecognisedException   If the ID
    does not match to any stage in
577      *                                    the system
    .
578      * @throws InvalidLocationException   If the
    location is out of bounds of the
579      *                                    stage
    length.
580      * @throws InvalidStageStateException If the
    stage is "waiting for results".
581      * @throws InvalidStageTypeException  Time-trial
     stages cannot contain any
582      *                                    segment.
583      */
584     @Override
585     public int addCategorizedClimbToStage(int
    stageId, Double location, SegmentType type, Double
```

```
585 averageGradient, Double length) throws
    IDNotRecognisedException, InvalidLocationException,
    InvalidStageStateException,
    InvalidStageTypeException {
586         Stage stage = getStageByIDOrNull(stageId);
587         if (stage == null) throw new
    IDNotRecognisedException();
588         if (stage.state != StageState.SETUP) throw
    new InvalidStageStateException();
589         if (stage.type == StageType.TT) throw new
    InvalidStageTypeException();
590         if (location > stage.length) throw new
    InvalidLocationException();
591
592         Segment newSegment = new Segment();
593         newSegment.id = segments.stream().mapToInt(s
     -> s.id).max().orElse(0) + 1;
594         newSegment.location = location;
595         newSegment.type = type;
596         newSegment.averageGradient = averageGradient
    ;
597
598         segments.add(newSegment);
599         stage.segments.add(newSegment.id);
600
601         return newSegment.id;
602     }
603
604     /**
605      * Adds an intermediate sprint to a stage.
606      * <p>
607      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
608      * exceptions are thrown.
609      *
610      * @param stageId  The ID of the stage to which
    the intermediate sprint segment
611      *                 is being added.
612      * @param location The kilometre location where
    the intermediate sprint finishes
613      *                 within the stage.
```

```
614         * @return The ID of the segment created.
615         * @throws IDNotRecognisedException   If the ID
    does not match to any stage in
616         *                                    the system
    .
617         * @throws InvalidLocationException   If the
    location is out of bounds of the
618         *                                    stage
    length.
619         * @throws InvalidStageStateException If the
    stage is "waiting for results".
620         * @throws InvalidStageTypeException  Time-trial
     stages cannot contain any
621         *                                    segment.
622         */
623        @Override
624        public int addIntermediateSprintToStage(int
    stageId, double location) throws
    IDNotRecognisedException, InvalidLocationException,
    InvalidStageStateException,
    InvalidStageTypeException {
625            Stage stage = getStageByIDOrNull(stageId);
626            if (stage == null) throw new
    IDNotRecognisedException();
627            if (stage.state != StageState.SETUP) throw
    new InvalidStageStateException();
628            if (stage.type == StageType.TT) throw new
    InvalidStageTypeException();
629            if (location > stage.length) throw new
    InvalidLocationException();
630
631            Segment newSegment = new Segment();
632            newSegment.id = segments.stream().mapToInt(s
     -> s.id).max().orElse(0) + 1;
633            newSegment.location = location;
634            newSegment.type = SegmentType.SPRINT;
635
636            segments.add(newSegment);
637            stage.segments.add(newSegment.id);
638
639            return newSegment.id;
```

```
640     }
641
642     /**
643      * Removes a segment from a stage.
644      * <p>
645      * The state of this MiniCyclingPortalInterface must be unchanged if any
646      * exceptions are thrown.
647      *
648      * @param segmentId The ID of the segment to be removed.
649      * @throws IDNotRecognisedException   If the ID does not match to any segment in
650      *                                    the system.
651      * @throws InvalidStageStateException If the stage is "waiting for results".
652      */
653     @Override
654     public void removeSegment(int segmentId) throws IDNotRecognisedException, InvalidStageStateException {
655         Segment segment = getSegmentByIDOrNull(segmentId);
656         if (segment == null) throw new IDNotRecognisedException();
657         Stage stage = stages.stream().filter(s -> s.segments.contains(segment.id)).findAny().orElse(null);
658         if (stage == null) {System.out.println("Consistency error. Corrupted data?"); return;}
659         if (stage.state == StageState.WAITING_FOR_RESULTS) throw new InvalidStageStateException();
660
661         segments.remove(segment);
662         stage.segments.remove(stage.segments.stream().filter(sid -> sid == segment.id).findAny().orElse(0)); // Or else will never happen bc we check for consistency earlier
663     }
```

```
664
665     /**
666      * Concludes the preparation of a stage. After
    conclusion, the stage's state
667      * should be "waiting for results".
668      * <p>
669      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
670      * exceptions are thrown.
671      *
672      * @param stageId The ID of the stage to be
    concluded.
673      * @throws IDNotRecognisedException   If the ID
    does not match to any stage in
674      *                                    the system
    .
675      * @throws InvalidStageStateException If the
    stage is "waiting for results".
676      */
677     @Override
678     public void concludeStagePreparation(int stageId
    ) throws IDNotRecognisedException,
    InvalidStageStateException {
679         Stage stage = getStageByIDOrNull(stageId);
680         if (stage == null) throw new
    IDNotRecognisedException();
681         if (stage.state == StageState.
    WAITING_FOR_RESULTS) throw new
    InvalidStageStateException();
682
683         stage.state = StageState.WAITING_FOR_RESULTS
    ;
684     }
685
686     /**
687      * Retrieves the list of segment (mountains and
    sprints) IDs of a stage.
688      * <p>
689      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
690      * exceptions are thrown.
```

```
691      *
692      * @param stageId The ID of the stage being
    queried.
693      * @return The list of segment IDs ordered (from
     first to last) by their location in the
694      * stage.
695      * @throws IDNotRecognisedException If the ID
    does not match to any stage in the
696      *                                   system.
697      */
698     @Override
699     public int[] getStageSegments(int stageId)
    throws IDNotRecognisedException {
700         Stage stage = getStageByIDOrNull(stageId);
701         if (stage == null) throw new
    IDNotRecognisedException();
702
703         return stage.segments.stream().mapToInt(i
     -> i).toArray();
704     }
705
706     /**
707      * Creates a team with name and description.
708      * <p>
709      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
710      * exceptions are thrown.
711      *
712      * @param name        The identifier name of the
     team.
713      * @param description A description of the team.
714      * @return The ID of the created team.
715      * @throws IllegalNameException If the name
    already exists in the platform.
716      * @throws InvalidNameException If the new name
    is null, empty, has more than
717      *                                 30 characters.
718      */
719     @Override
720     public int createTeam(String name, String
    description) throws IllegalNameException,
```

```
720 InvalidNameException {
721         // Check data
722         if (name == null || name.equals("") || name.
    length() > 30) throw new InvalidNameException("Name
    is null, empty or has more than 30 characters");
723 
724         // Find duplicates and next valid ID
725         int currentMaxTeamID = 0;
726         for (Team t : teams) {
727             if (t.id > currentMaxTeamID)
    currentMaxTeamID = t.id;
728             if (t.name.equals(name)) throw new
    IllegalNameException("A Team with the same name
    already exists");
729         }
730 
731         // Create the team
732         Team newTeam = new Team();
733         newTeam.id = currentMaxTeamID + 1;
734         newTeam.name = name;
735         newTeam.description = description;
736 
737         teams.add(newTeam);
738         return newTeam.id;
739     }
740 
741     /**
742      * Removes a team from the system.
743      * <p>
744      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
745      * exceptions are thrown.
746      *
747      * @param teamId The ID of the team to be
    removed.
748      * @throws IDNotRecognisedException If the ID
    does not match to any team in the
749      *                                  system.
750      */
751     @Override
752     public void removeTeam(int teamId) throws
```

```
752 IDNotRecognisedException {
753         Team team = getTeamByIDOrNull(teamId);
754         if (team == null) throw new
    IDNotRecognisedException("ID not found in removeTeam
    ");
755 
756         teams.remove(teamId - 1);
757 
758     }
759 
760     /**
761      * Get the list of teams' IDs in the system.
762      * <p>
763      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
764      * exceptions are thrown.
765      *
766      * @return The list of IDs from the teams in the
     system. An empty list if there
767      * are no teams in the system.
768      */
769     @Override
770     public int[] getTeams() {
771         return teams.stream()
772                 .mapToInt(t -> t.id)
773                 .toArray();
774     }
775 
776 
777     /**
778      * Get the riders of a team.
779      * <p>
780      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
781      * exceptions are thrown.
782      *
783      * @param teamId The ID of the team being
    queried.
784      * @return A list with riders' ID.
785      * @throws IDNotRecognisedException If the ID
    does not match to any team in the
```

```
786      *                                    system.
787      */
788     @Override
789     public int[] getTeamRiders(int teamId) throws
    IDNotRecognisedException {
790         Team team = getTeamByIDOrNull(teamId);
791         if (team == null) throw new
    IDNotRecognisedException("Team ID not found in
    getTeamRiders");
792 
793         return team.riders.stream().mapToInt(r -> r
    ).toArray();
794     }
795 
796     /**
797      * Creates a rider.
798      * <p>
799      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
800      * exceptions are thrown.
801      *
802      * @param teamID      The ID rider's team.
803      * @param name        The name of the rider.
804      * @param yearOfBirth The year of birth of the
    rider.
805      * @return The ID of the rider in the system.
806      * @throws IDNotRecognisedException If the ID
    does not match to any team in the
807      *                                    system.
808      * @throws IllegalArgumentException If the name
    of the rider is null or the year
809      *                                    of birth is
    less than 1900.
810      */
811     @Override
812     public int createRider(int teamID, String name,
    int yearOfBirth) throws IDNotRecognisedException,
    IllegalArgumentException {
813         Team team = getTeamByIDOrNull(teamID); if (
    team == null) throw new IDNotRecognisedException();
814         if (name == null || yearOfBirth < 1900)
```

```
814 throw new IllegalArgumentException();
815
816         Rider newRider = new Rider();
817         newRider.name = name; newRider.yearOfBirth
     = yearOfBirth;
818
819         int maxRiderID = riders.stream().mapToInt(r
     -> r.id).max().orElse(0);
820         newRider.id = maxRiderID + 1;
821         newRider.results = new HashMap<>();
822
823         team.riders.add(newRider.id);
824         riders.add(newRider);
825         return newRider.id;
826     }
827
828     /**
829      * Removes a rider from the system. When a rider
     is removed from the platform,
830      * all of its results should be also removed.
    Race results must be updated.
831      * <p>
832      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
833      * exceptions are thrown.
834      *
835      * @param riderId The ID of the rider to be
    removed.
836      * @throws IDNotRecognisedException If the ID
    does not match to any rider in the
837      *                                  system.
838      */
839     @Override
840     public void removeRider(int riderId) throws
    IDNotRecognisedException {
841         Rider rider = getRiderByIDOrNull(riderId);
    if (rider == null) throw new
    IDNotRecognisedException();
842
843         Team team = teams.stream().filter(t -> t.
    riders.contains(rider.id)).findAny().orElse(null);
```

```
844         if (team == null) {System.out.println("
    Consistency error. Corrupted data?"); return;}
845 
846         riders.remove(rider);
847         team.riders.remove(rider.id - 1);
848     }
849 
850     /**
851      * Record the times of a rider in a stage.
852      * <p>
853      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
854      * exceptions are thrown.
855      *
856      * @param stageId     The ID of the stage the
    result refers to.
857      * @param riderId     The ID of the rider.
858      * @param checkpoints An array of times at which
     the rider reached each of the
859      *                    segments of the stage,
    including the start time and the
860      *                    finish line.
861      * @throws IDNotRecognisedException    If the ID
     does not match to any rider or
862      *                                     stage in
    the system.
863      * @throws DuplicatedResultException   Thrown if
     the rider has already a result
864      *                                     for the
    stage. Each rider can have only
865      *                                     one
    result per stage.
866      * @throws InvalidCheckpointsException Thrown if
     the length of checkpoints is
867      *                                     not equal
     to n+2, where n is the number
868      *                                     of
    segments in the stage; +2 represents
869      *                                     the start
     time and the finish time of the
870      *                                     stage.
```

```
871      * @throws InvalidStageStateException  Thrown if the stage is not "waiting for
872      *                                     results". Results can only be added to a
873      *                                     stage while it is "waiting for results".
874      */
875     @Override
876     public void registerRiderResultsInStage(int stageId, int riderId, LocalTime... checkpoints) throws IDNotRecognisedException,
        DuplicatedResultException,
        InvalidCheckpointsException,
        InvalidStageStateException {
877         Rider rider = getRiderByIDOrNull(riderId);
878         Stage stage = getStageByIDOrNull(stageId);
879         if (rider == null) throw new IDNotRecognisedException("rider ID not recognized");
880         if (stage == null) throw new IDNotRecognisedException("stage not recognized");
881         if (rider.results.containsKey(stageId)) throw new DuplicatedResultException("results already exist");
882         if (stage.state != StageState.WAITING_FOR_RESULTS) throw new InvalidStageStateException("invalid stage state");
883         if (checkpoints.length != stage.segments.size() + 2) throw new InvalidCheckpointsException();
884
885         rider.results.put(stageId, checkpoints);
886     }
887
888     /**
889      * Get the times of a rider in a stage.
890      * <p>
891      * The state of this MiniCyclingPortalInterface must be unchanged if any
892      * exceptions are thrown.
893      *
894      * @param stageId The ID of the stage the result refers to.
```

```
895         * @param riderId The ID of the rider.
896         * @return The array of times at which the rider
     reached each of the segments of
897         * the stage and the total elapsed time. The
    elapsed time is the
898         * difference between the finish time and the
    start time. Return an
899         * empty array if there is no result registered
    for the rider in the
900         * stage.
901         * @throws IDNotRecognisedException If the ID
    does not match to any rider or
902         *                                      stage in the
     system.
903         */
904        @Override
905        public LocalTime[] getRiderResultsInStage(int
    stageId, int riderId) throws
    IDNotRecognisedException {
906            Rider rider = getRiderByIDOrNull(riderId);
907            if (rider == null) throw new
    IDNotRecognisedException("Rider ID not found");
908            return rider.results.get(stageId);
909
910        }
911
912        /**
913         * For the general classification, the
    aggregated time is based on the adjusted
914         * elapsed time, not the real elapsed time.
    Adjustments are made to take into
915         * account groups of riders finishing very close
     together, e.g., the peloton. If
916         * a rider has a finishing time less than one
    second slower than the
917         * previous rider, then their adjusted elapsed
    time is the smallest of both. For
918         * instance, a stage with 200 riders finishing "
    together" (i.e., less than 1
919         * second between consecutive riders), the
    adjusted elapsed time of all riders
```

```
920         * should be the same as the first of these
    riders, even if the real gap
921         * between the 200th and the 1st rider is much
    bigger than 1 second. There is no
922         * adjustments on elapsed time on time-trials.
923         * <p>
924         * The state of this MiniCyclingPortalInterface
    must be unchanged if any
925         * exceptions are thrown.
926         *
927         * @param stageId The ID of the stage the result
     refers to.
928         * @param riderId The ID of the rider.
929         * @return The adjusted elapsed time for the
    rider in the stage. Return an empty
930         * array if there is no result registered for
    the rider in the stage.
931         * @throws IDNotRecognisedException If the ID
    does not match to any rider or
932         *                                      stage in the
     system.
933         */
934        @Override
935        public LocalTime
    getRiderAdjustedElapsedTimeInStage(int stageId, int
    riderId) throws IDNotRecognisedException {
936            Stage stage = getStageByIDOrNull(stageId);
937            if (stage == null) throw new
    IDNotRecognisedException();
938            Rider rider = getRiderByIDOrNull(riderId);
939            if (rider == null) throw new
    IDNotRecognisedException();
940
941            if (!rider.results.containsKey(stageId))
    return LocalTime.MIDNIGHT; // WARNING: docstring
    says to return empty array. The function does not
    return an array, so I used localtime.MIDNIGHT
942            LocalTime thisRiderFinishTime = rider.
    results.get(stageId)[rider.results.get(stageId).
    length - 1];
943            LocalTime thisRiderStartTime = rider.results
```

```
943 .get(stageId)[0];
944         if (stage.type == StageType.TT) return
    LocalTime.MIDNIGHT.plus(Duration.between(
    thisRiderStartTime, thisRiderFinishTime));
945 
946         // If this rider finished this stage less
    than a second after another rider then return the
    other rider's time.
947         // This is the adjusted time
948         Rider[] ridersThatTookPartInTheStage =
    riders.stream().filter(r -> r.results.containsKey(
    stageId)).toArray(Rider[]::new);
949         // If the rider's time was adjusted then it
    may now be less than a second after another rider's
    time.
950         // If so then run the adjustment again until
     no rider is in range
951         boolean anAdjustmentWasMadeLastIteration =
    true;
952         while (anAdjustmentWasMadeLastIteration) {
953             for (Rider otherRider :
    ridersThatTookPartInTheStage) {
954                 LocalTime otherRiderFinishTime =
    otherRider.results.get(stageId)[otherRider.results.
    get(stageId).length - 1];
955                 long differenceBetweenFinishTimes =
    ChronoUnit.MILLIS.between(otherRiderFinishTime,
    thisRiderFinishTime);
956 
957                 if (differenceBetweenFinishTimes <
    1000 && differenceBetweenFinishTimes > 0) { // If I
    finish less than one second after another rider
958                     thisRiderFinishTime =
    otherRiderFinishTime;
959                     anAdjustmentWasMadeLastIteration
     = true;
960                 } else {
961                     anAdjustmentWasMadeLastIteration
     = false;
962                 }
963             }
```

```
964         }
965
966         return LocalTime.MIDNIGHT.plus(Duration.
    between(thisRiderStartTime, thisRiderFinishTime));
967     }
968
969     /**
970      * Removes the stage results from the rider.
971      * <p>
972      * The state of this MiniCyclingPortalInterface
     must be unchanged if any
973      * exceptions are thrown.
974      *
975      * @param stageId The ID of the stage the
    result refers to.
976      * @param riderId The ID of the rider.
977      * @throws IDNotRecognisedException If the ID
    does not match to any rider or
978      *                                    stage in
    the system.
979      */
980     @Override
981     public void deleteRiderResultsInStage(int
    stageId, int riderId) throws
    IDNotRecognisedException {
982         Rider rider = getRiderByIDOrNull(riderId);
983         if (rider == null) throw new
    IDNotRecognisedException("Rider ID not found");
984
985         rider.results.remove(stageId);
986     }
987
988     /**
989      * Get the riders finished position in a stage.
990      * <p>
991      * The state of this MiniCyclingPortalInterface
     must be unchanged if any
992      * exceptions are thrown.
993      *
994      * @param stageId The ID of the stage being
    queried.
```

```
 995      * @return A list of riders ID sorted by their elapsed time. An empty list if
 996      * there is no result for the stage.
 997      * @throws IDNotRecognisedException If the ID does not match any stage in the
 998      *                                  system.
 999      */
1000     @Override
1001     public int[] getRidersRankInStage(int stageId) throws IDNotRecognisedException {
1002         Stage stage = getStageByIDOrNull(stageId);
1003         if (stage == null) throw new IDNotRecognisedException();
1004 
1005         Rider[] applicableRiders = riders.stream().filter(r -> r.results.containsKey(stage.id)).toArray(Rider[]::new);
1006 
1007         HashMap<Long, Rider> results = new HashMap<>();
1008         Arrays.stream(applicableRiders).forEach(r -> {
1009             LocalTime[] times = r.results.get(stageId);
1010             Long time = ChronoUnit.MICROS.between(times[0], times[times.length - 1]);
1011 
1012             // If someone else has the exact same time then increase the time by one microsecond
1013             // I shouldn't have to do this, but I don't know any other decent way
1014             // I use this form because results.containsKey is always false according to the IDE
1015             while (results.containsKey(time)) time++;
1016 
1017             results.put(time, r);
1018         });
1019 
1020         ArrayList<Rider> ridersInOrder = hashMapValuesSortedByComparableKey(results);
```

```
1021            return ridersInOrder.stream().mapToInt(r
      -> r.id).toArray();
1022        }
1023
1024        private <Key extends Comparable<? super Key>, O
      > ArrayList<O> hashMapValuesSortedByComparableKey(
      HashMap<Key, O> hm) {
1025            ArrayList<O> objectsInOrder =  new
      ArrayList<>();
1026            ArrayList<Key> keys = new ArrayList<>(hm.
      keySet());
1027
1028            while(keys.size() > 0) {
1029                // Find the smallest key
1030                Key smallestKey = keys.stream().min(Key
      ::compareTo).orElse(null);
1031                keys.remove(smallestKey);
1032                objectsInOrder.add(hm.get(smallestKey
      ));
1033            }
1034
1035            return objectsInOrder;
1036        }
1037
1038
1039
1040
1041
1042        /**
1043         * Get the adjusted elapsed times of riders in
      a stage.
1044         * <p>
1045         * The state of this MiniCyclingPortalInterface
       must be unchanged if any
1046         * exceptions are thrown.
1047         *
1048         * @param stageId The ID of the stage being
      queried.
1049         * @return The ranked list of adjusted elapsed
      times sorted by their finish
1050         * time. An empty list if there is no result
```

```
1050 for the stage. These times
1051      * should match the riders returned by
1052      * {@link #getRidersRankInStage(int)}.
1053      * @throws IDNotRecognisedException If the ID
     does not match any stage in the
1054      *                                    system.
1055      */
1056     @Override
1057     public LocalTime[]
     getRankedAdjustedElapsedTimesInStage(int stageId)
     throws IDNotRecognisedException {
1058         int[] rankedRiderIds = getRidersRankInStage
     (stageId);
1059         // Key: RiderId. Val: Adjusted elapsed
     times.
1060         HashMap<Integer, LocalTime> adjustedTimes
      = new HashMap<>();
1061         for (int rid : rankedRiderIds) {
1062             adjustedTimes.put(rid,
     getRiderAdjustedElapsedTimeInStage(stageId, rid));
1063         }
1064
1065         ArrayList<LocalTime> orderedResults = new
     ArrayList<>();
1066         for (int rid : rankedRiderIds) {
1067             orderedResults.add(adjustedTimes.get(
     rid));
1068         }
1069
1070         return orderedResults.toArray(LocalTime[]::
     new);
1071     }
1072
1073     /**
1074      * Get the number of points obtained by each
     rider in a stage.
1075      * <p>
1076      * The state of this MiniCyclingPortalInterface
      must be unchanged if any
1077      * exceptions are thrown.
1078      *
```

```
1079      * @param stageId The ID of the stage being queried.
1080      * @return The ranked list of points each rider received in the stage, sorted
1081      * by their elapsed time. An empty list if there is no result for the
1082      * stage. These points should match the riders returned by
1083      * {@link #getRidersRankInStage(int)}.
1084      * @throws IDNotRecognisedException If the ID does not match any stage in the
1085      *                                  system.
1086      */
1087     @Override
1088     public int[] getRidersPointsInStage(int stageId) throws IDNotRecognisedException {
1089         // Should this include intermediate sprints? YES
1090         Stage stage = getStageByIDOrNull(stageId);
1091         if (stage == null) throw new IDNotRecognisedException();
1092
1093         int[] rankedRiderIds = getRidersRankInStage(stageId);
1094
1095         ArrayList<Integer> riderPoints = new ArrayList<>();
1096         for (int riderIdIndex = 0; riderIdIndex < rankedRiderIds.length; riderIdIndex++) {
1097             riderPoints.add(stagePoints(stage.type, riderIdIndex) + getRidersSegmentPointsInStage(stageId, false)[riderIdIndex]);
1098         }
1099         return riderPoints.stream().mapToInt(i -> i).toArray();
1100     }
1101
1102
1103
1104     /**
1105      * Get the number of mountain points obtained
```

```
1105 by each rider in a stage.
1106      * <p>
1107      * The state of this MiniCyclingPortalInterface
      must be unchanged if any
1108      * exceptions are thrown.
1109      *
1110      * @param stageId The ID of the stage being
     queried.
1111      * @return The ranked list of mountain points
     each rider received in the stage,
1112      * sorted by their finish time. An empty list
     if there is no result for
1113      * the stage. These points should match the
     riders returned by
1114      * {@link #getRidersRankInStage(int)}.
1115      * @throws IDNotRecognisedException If the ID
     does not match any stage in the
1116      *                                  system.
1117      */
1118     @Override
1119     public int[] getRidersMountainPointsInStage(int
      stageId) throws IDNotRecognisedException {
1120         return getRidersSegmentPointsInStage(
     stageId, true);
1121     }
1122
1123     private int[] getRidersSegmentPointsInStage(int
      stageId, boolean mountainTrueSprintFalse) throws
     IDNotRecognisedException {
1124         // Should I include the intermediate sprint
      segment points? NO
1125         Stage stage = getStageByIDOrNull(stageId);
1126         if (stage == null) throw new
     IDNotRecognisedException();
1127
1128         Segment[] segments = stage.segments.stream
     ().map(this::getSegmentByIDOrNull).toArray(Segment
     []::new);
1129         if (Arrays.stream(segments).anyMatch(
     Objects::isNull)) throw new
     IDNotRecognisedException();
```

```
1130
1131            int[] rankedRiderIds = getRidersRankInStage
    (stageId);
1132            Rider[] rankedRiders = Arrays.stream(
    rankedRiderIds).mapToObj(this::getRiderByIDOrNull).
    toArray(Rider[]::new);
1133            if (Arrays.stream(rankedRiders).anyMatch(
    Objects::isNull)) throw new
    IDNotRecognisedException();
1134
1135            // Foreach segment work out the rank
1136            HashMap<Rider, Integer> riderPointsInStage
     = new HashMap<>();
1137            for (int segmentIndex = 0; segmentIndex <
    segments.length; segmentIndex++) {
1138                if (mountainTrueSprintFalse && segments
    [segmentIndex].type == SegmentType.SPRINT) continue
    ; // TODO Check if it works okay? // Excludes all
    intermediate sprints
1139                if (!mountainTrueSprintFalse &&
    segments[segmentIndex].type != SegmentType.SPRINT)
    continue;
1140
1141                HashMap<Duration, Rider>
    ridersDurationsInASegment = new HashMap<>();
1142                for (Rider rider : rankedRiders) {
1143                    LocalTime[] ridersResults = rider.
    results.get(stageId);
1144                    ridersDurationsInASegment.put(
1145                            Duration.between(
    ridersResults[segmentIndex], ridersResults[
    segmentIndex + 1]),
1146                            rider
1147                    );
1148                }
1149                ArrayList<Rider> ridersRankInASegment
     = hashMapValuesSortedByComparableKey(
    ridersDurationsInASegment);
1150
1151                // Convert a rider's rank into points
1152                // Key rider. Value points
```

```
1153            HashMap<Rider, Integer> riderSegPoints
     = new HashMap<>();
1154            for (int riderRnkSegIndex = 0;
    riderRnkSegIndex < ridersRankInASegment.size();
    riderRnkSegIndex++) {
1155                riderSegPoints.put(
1156                        ridersRankInASegment.get(
    riderRnkSegIndex),
1157                        segmentPoints(segments[
    segmentIndex].type, riderRnkSegIndex)
1158                );
1159            }
1160
1161            // riderSegPoints is now representative
     of each rider's total points in this segment
1162            for (Rider rider : riderSegPoints.
    keySet()) {
1163                if (!riderPointsInStage.containsKey
    (rider)) riderPointsInStage.put(rider, 0);
1164                riderPointsInStage.put(rider,
    riderPointsInStage.get(rider) + riderSegPoints.get(
    rider));
1165            }
1166        }
1167
1168        // Now we have riderPointsInStage filled
    out
1169        ArrayList<Integer> rankedRiderPoints = new
    ArrayList<>();
1170        for (int riderId : rankedRiderIds) {
1171            Rider thisRiderObj = getRiderByIDOrNull
    (riderId); // Will not be null
1172            rankedRiderPoints.add(
    riderPointsInStage.get(thisRiderObj));
1173        }
1174
1175        // Filter out riders with no associated
    points
1176        rankedRiderPoints = new ArrayList<>(
    rankedRiderPoints.stream().map(p -> p == null ? 0
     : p).toList());
```

```
1177
1178            return rankedRiderPoints.stream().mapToInt(
     i -> i).toArray();
1179        }
1180
1181        /**
1182         * Method empties this
     MiniCyclingPortalInterface of its contents and
     resets all
1183         * internal counters.
1184         */
1185        @Override
1186        public void eraseCyclingPortal() {
1187            teams = new ArrayList<>();
1188            riders = new ArrayList<>();
1189            races = new ArrayList<>();
1190            stages = new ArrayList<>();
1191            segments = new ArrayList<>();
1192        }
1193
1194        /**
1195         * Method saves this MiniCyclingPortalInterface
      contents into a serialised file,
1196         * with the filename given in the argument.
1197         * <p>
1198         * The state of this MiniCyclingPortalInterface
      must be unchanged if any
1199         * exceptions are thrown.
1200         *
1201         * @param filename Location of the file to be
     saved.
1202         * @throws IOException If there is a problem
     experienced when trying to save the
1203         *                     store contents to the
     file.
1204         */
1205        @Override
1206        public void saveCyclingPortal(String filename)
     throws IOException {
1207            FileOutputStream file = new
     FileOutputStream(filename);
```

```
1208            ObjectOutputStream objOut = new
    ObjectOutputStream(file);
1209            objOut.writeObject(this);
1210            objOut.flush();
1211            objOut.close();
1212        }
1213
1214        /**
1215         * Method should load and replace this
    MiniCyclingPortalInterface contents with the
1216         * serialised contents stored in the file given
     in the argument.
1217         * <p>
1218         * The state of this MiniCyclingPortalInterface
     must be unchanged if any
1219         * exceptions are thrown.
1220         *
1221         * @param filename Location of the file to be
    loaded.
1222         * @throws IOException             If there is a
     problem experienced when trying
1223         *                                 to load the
    store contents from the file.
1224         * @throws ClassNotFoundException If required
    class files cannot be found when
1225         *                                 loading.
1226         */
1227        @Override
1228        public void loadCyclingPortal(String filename)
    throws IOException, ClassNotFoundException {
1229            FileInputStream file = new FileInputStream(
    filename);
1230            ObjectInputStream objIn = new
    ObjectInputStream(file);
1231            CyclingPortal c = (CyclingPortal) objIn.
    readObject();
1232
1233            teams = c.teams;
1234            riders = c.riders;
1235            races = c.races;
1236            stages = c.stages;
```

```
1237            segments = c.segments;
1238
1239            objIn.close();
1240            file.close();
1241        }
1242    }
1243
```

```java
package cycling;

/**
 * Thrown when attempting to assign a race name already in use in the
 * system.
 *
 * @author Diogo Pacheco
 * @version 1.0
 *
 */
public class IllegalNameException extends Exception {

    /**
     * Constructs an instance of the exception with no message
     */
    public IllegalNameException() {
        // do nothing
    }

    /**
     * Constructs an instance of the exception containing the message argument
     *
     * @param message message containing details regarding the exception cause
     */
    public IllegalNameException(String message) {
        super(message);
    }

}

```

```java
package cycling;

/**
 * Thrown when attempting to assign a race name null, empty or having more than
 * the system limit of characters. A name must be a single word, i.e., no
 * white spaces allowed.
 *
 * @author Diogo Pacheco
 * @version 1.0
 *
 */
public class InvalidNameException extends Exception {

    /**
     * Constructs an instance of the exception with no message
     */
    public InvalidNameException() {
        // do nothing
    }

    /**
     * Constructs an instance of the exception containing the message argument
     *
     * @param message message containing details regarding the exception cause
     */
    public InvalidNameException(String message) {
        super(message);
    }

}

```

```java
package cycling;

import java.time.LocalTime;

/**
 * CyclingPortalInterface interface. The no-argument constructor of a class
 * implementing this interface should initialise the CyclingPortalInterface as
 * an empty platform with no initial racing teams nor races within it. For pair
 * submissions ONLY.
 *
 * @author Diogo Pacheco
 * @version 1.1
 *
 */
public interface CyclingPortalInterface extends MiniCyclingPortalInterface {

    /**
     * The method removes the race and all its related information, i.e., stages,
     * segments, and results.
     * <p>
     * The state of this MiniCyclingPortalInterface must be unchanged if any
     * exceptions are thrown.
     *
     * @param name The name of the race to be removed.
     * @throws NameNotRecognisedException If the name does not match to any race in
     *                                    the system.
     */
    void removeRaceByName(String name) throws NameNotRecognisedException;

    /**
     * Get the general classification rank of riders in a race.
```

```
32      * <p>
33      * The state of this MiniCyclingPortalInterface must be unchanged if any
34      * exceptions are thrown.
35      *
36      * @param raceId The ID of the race being queried.
37      * @return A ranked list of riders' IDs sorted ascending by the sum of their
38      *         adjusted elapsed times in all stages of the race. That is, the first
39      *         in this list is the winner (least time). An empty list if there is no
40      *         result for any stage in the race.
41      * @throws IDNotRecognisedException If the ID does not match any race in the
42      *                                  system.
43      */
44     int[] getRidersGeneralClassificationRank(int raceId) throws IDNotRecognisedException;
45
46     /**
47      * Get the general classification times of riders in a race.
48      * <p>
49      * The state of this MiniCyclingPortalInterface must be unchanged if any
50      * exceptions are thrown.
51      *
52      * @param raceId The ID of the race being queried.
53      * @return A list of riders' times sorted by the sum of their adjusted elapsed
54      *         times in all stages of the race. An empty list if there is no result
55      *         for any stage in the race. These times should match the riders
56      *         returned by {@link #getRidersGeneralClassificationRank(int)}.
57      * @throws IDNotRecognisedException If the ID does not match any race in the
```

```
58      *                                       system.
59      */
60     LocalTime[] getGeneralClassificationTimesInRace(
   int raceId) throws IDNotRecognisedException;
61
62     /**
63      * Get the overall points of riders in a race.
64      * <p>
65      * The state of this MiniCyclingPortalInterface
   must be unchanged if any
66      * exceptions are thrown.
67      *
68      * @param raceId The ID of the race being queried
   .
69      * @return A list of riders' points (i.e., the
   sum of their points in all stages
70      *         of the race), sorted by the total
   elapsed time. An empty list if
71      *         there is no result for any stage in
   the race. These points should
72      *         match the riders returned by {@link #
   getRidersGeneralClassificationRank(int)}.
73      * @throws IDNotRecognisedException If the ID
   does not match any race in the
74      *                                       system.
75      */
76     int[] getRidersPointsInRace(int raceId) throws
   IDNotRecognisedException;
77
78     /**
79      * Get the overall mountain points of riders in a
    race.
80      * <p>
81      * The state of this MiniCyclingPortalInterface
   must be unchanged if any
82      * exceptions are thrown.
83      *
84      * @param raceId The ID of the race being queried
   .
85      * @return A list of riders' mountain points (i.e
   ., the sum of their mountain
```

```
 86      *          points in all stages of the race),
    sorted by the total elapsed time.
 87      *          An empty list if there is no result
    for any stage in the race. These
 88      *          points should match the riders
    returned by
 89      *          {@link #
    getRidersGeneralClassificationRank(int)}.
 90      * @throws IDNotRecognisedException If the ID
    does not match any race in the
 91      *                                     system.
 92      */
 93     int[] getRidersMountainPointsInRace(int raceId)
    throws IDNotRecognisedException;
 94
 95     /**
 96      * Get the ranked list of riders based on the
    points classification in a race.
 97      * <p>
 98      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
 99      * exceptions are thrown.
100      *
101      * @param raceId The ID of the race being
    queried.
102      * @return A ranked list of riders' IDs sorted
    descending by the sum of their
103      *          points in all stages of the race.
    That is, the first in this list is
104      *          the winner (more points). An empty
    list if there is no result for any
105      *          stage in the race.
106      * @throws IDNotRecognisedException If the ID
    does not match any race in the
107      *                                     system.
108      */
109     int[] getRidersPointClassificationRank(int
    raceId) throws IDNotRecognisedException;
110
111     /**
112      * Get the ranked list of riders based on the
```

```
112 mountain classification in a race.
113      * <p>
114      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
115      * exceptions are thrown.
116      *
117      * @param raceId The ID of the race being
    queried.
118      * @return A ranked list of riders' IDs sorted
    descending by the sum of their
119      *         mountain points in all stages of the
    race. That is, the first in this
120      *         list is the winner (more points). An
    empty list if there is no result
121      *         for any stage in the race.
122      * @throws IDNotRecognisedException If the ID
    does not match any race in the
123      *                                  system.
124      */
125     int[] getRidersMountainPointClassificationRank(
    int raceId) throws IDNotRecognisedException;
126
127 }
128
```

```java
package cycling;

/**
 * Thrown when attempting to assign a race length null or less than 5 (kilometres).
 *
 * @author Diogo Pacheco
 * @version 1.0
 *
 */
public class InvalidLengthException extends Exception {

    /**
     * Constructs an instance of the exception with no message
     */
    public InvalidLengthException() {
        // do nothing
    }

    /**
     * Constructs an instance of the exception containing the message argument
     *
     * @param message message containing details regarding the exception cause
     */
    public InvalidLengthException(String message) {
        super(message);
    }

}

```

```java
package cycling;

/**
 * Thrown when attempting to use an ID that does not exit in the system.
 *
 * @author Diogo Pacheco
 * @version 1.0
 *
 */
public class IDNotRecognisedException extends Exception {

    /**
     * Constructs an instance of the exception with no message
     */
    public IDNotRecognisedException() {
        // do nothing
    }

    /**
     * Constructs an instance of the exception containing the message argument
     *
     * @param message message containing details regarding the exception cause
     */
    public IDNotRecognisedException(String message) {
        super(message);
    }

}

```

```java
package cycling;

/**
 * Thrown when attempting to assign a location outside the bounds of the stage,
 * i.e. 0 {@literal < } location {@literal <=} stage's length.
 *
 * @author Diogo Pacheco
 * @version 1.0
 *
 */
public class InvalidLocationException extends Exception {

    /**
     * Constructs an instance of the exception with no message
     */
    public InvalidLocationException() {
        // do nothing
    }

    /**
     * Constructs an instance of the exception containing the message argument
     *
     * @param message message containing details regarding the exception cause
     */
    public InvalidLocationException(String message) {
        super(message);
    }

}


```

```java
package cycling;

/**
 * Each rider can only have a single result in a race. This exception is thrown
 * when attempting to create another record for the same rider in a the same
 * registered race.
 *
 * @author Diogo Pacheco
 * @version 1.0
 *
 */
public class DuplicatedResultException extends Exception {

    /**
     * Constructs an instance of the exception with no message
     */
    public DuplicatedResultException() {
        // do nothing
    }

    /**
     * Constructs an instance of the exception containing the message argument
     *
     * @param message message containing details regarding the exception cause
     */
    public DuplicatedResultException(String message) {
        super(message);
    }

}

```

```java
package cycling;

/**
 * Thrown when attempting to add segments (sprints or mountains) to a time-trial stage.
 *
 * @author Diogo Pacheco
 * @version 1.0
 *
 */
public class InvalidStageTypeException extends Exception {

    /**
     * Constructs an instance of the exception with no message
     */
    public InvalidStageTypeException() {
        // do nothing
    }

    /**
     * Constructs an instance of the exception containing the message argument
     *
     * @param message message containing details regarding the exception cause
     */
    public InvalidStageTypeException(String message) {
        super(message);
    }

}

```

```java
package cycling;

/**
 * Thrown when attempting to perform an action within a stage that is
 * incompatible with its current state. For instance, when trying to add results
 * to a stage, but the stage is still under development, i.e., not concluded.
 *
 * @author Diogo Pacheco
 * @version 1.0
 *
 */
public class InvalidStageStateException extends Exception {

    /**
     * Constructs an instance of the exception with no message
     */
    public InvalidStageStateException() {
        // do nothing
    }

    /**
     * Constructs an instance of the exception containing the message argument
     *
     * @param message message containing details regarding the exception cause
     */
    public InvalidStageStateException(String message) {
        super(message);
    }

}

```

```java
package cycling;

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

/**
 * MiniCyclingPortalInterface interface. The no-argument constructor of a class
 * implementing this interface should initialise the MiniCyclingPortalInterface as
 * an empty platform with no initial racing teams nor races within it.
 *
 * @author Diogo Pacheco
 * @version 1.1
 *
 */
public interface MiniCyclingPortalInterface extends Serializable {

    /**
     * Get the races currently created in the platform.
     *
     * @return An array of race IDs in the system or an empty array if none exists.
     */
    int[] getRaceIds();

    /**
     * The method creates a staged race in the platform with the given name and
     * description.
     * <p>
     * The state of this MiniCyclingPortalInterface must be unchanged if any
     * exceptions are thrown.
     *
     * @param name        Race's name.
```

```
34      * @param description Race's description (can be
   null).
35      * @throws IllegalNameException If the name
   already exists in the platform.
36      * @throws InvalidNameException If the name is
   null, empty, has more than 30
37      *                              characters, or
   has white spaces.
38      * @return the unique ID of the created race.
39      *
40      */
41     int createRace(String name, String description)
   throws IllegalNameException, InvalidNameException;
42
43     /**
44      * Get the details from a race.
45      * <p>
46      * The state of this MiniCyclingPortalInterface
   must be unchanged if any
47      * exceptions are thrown.
48      *
49      * @param raceId The ID of the race being queried
   .
50      * @return Any formatted string containing the
   race ID, name, description, the
51      *         number of stages, and the total length
    (i.e., the sum of all stages'
52      *         length).
53      * @throws IDNotRecognisedException If the ID
   does not match to any race in the
54      *                                  system.
55      */
56     String viewRaceDetails(int raceId) throws
   IDNotRecognisedException;
57
58     /**
59      * The method removes the race and all its
   related information, i.e., stages,
60      * segments, and results.
61      * <p>
62      * The state of this MiniCyclingPortalInterface
```

```
62 must be unchanged if any
63      * exceptions are thrown.
64      *
65      * @param raceId The ID of the race to be
   removed.
66      * @throws IDNotRecognisedException If the ID
   does not match to any race in the
67      *                                  system.
68      */
69     void removeRaceById(int raceId) throws
   IDNotRecognisedException;
70
71     /**
72      * The method queries the number of stages
   created for a race.
73      * <p>
74      * The state of this MiniCyclingPortalInterface
   must be unchanged if any
75      * exceptions are thrown.
76      *
77      * @param raceId The ID of the race being
   queried.
78      * @return The number of stages created for the
   race.
79      * @throws IDNotRecognisedException If the ID
   does not match to any race in the
80      *                                  system.
81      */
82     int getNumberOfStages(int raceId) throws
   IDNotRecognisedException;
83
84     /**
85      * Creates a new stage and adds it to the race.
86      * <p>
87      * The state of this MiniCyclingPortalInterface
   must be unchanged if any
88      * exceptions are thrown.
89      *
90      * @param raceId      The race which the stage
   will be added to.
91      * @param stageName   An identifier name for the
```

```
 91  stage.
 92       * @param description A descriptive text for the
     stage.
 93       * @param length      Stage length in kilometres
    .
 94       * @param startTime   The date and time in which
     the stage will be raced. It
 95       *                    cannot be null.
 96       * @param type        The type of the stage.
    This is used to determine the
 97       *                    amount of points given to
    the winner.
 98       * @return the unique ID of the stage.
 99       * @throws IDNotRecognisedException If the ID
    does not match to any race in the
100       *                                  system.
101       * @throws IllegalNameException     If the name
    already exists in the platform.
102       * @throws InvalidNameException     If the new
    name is null, empty, has more
103       *                                  than 30.
104       * @throws InvalidLengthException   If the
    length is less than 5km.
105       */
106      int addStageToRace(int raceId, String stageName
    , String description, double length, LocalDateTime
    startTime,
107              StageType type)
108              throws IDNotRecognisedException,
    IllegalNameException, InvalidNameException,
    InvalidLengthException;
109
110      /**
111       * Retrieves the list of stage IDs of a race.
112       * <p>
113       * The state of this MiniCyclingPortalInterface
    must be unchanged if any
114       * exceptions are thrown.
115       *
116       * @param raceId The ID of the race being
    queried.
```

```
117      * @return The list of stage IDs ordered (from
    first to last) by their sequence in the
118      *         race.
119      * @throws IDNotRecognisedException If the ID
    does not match to any race in the
120      *                                     system.
121      */
122     int[] getRaceStages(int raceId) throws
    IDNotRecognisedException;
123
124     /**
125      * Gets the length of a stage in a race, in
    kilometres.
126      * <p>
127      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
128      * exceptions are thrown.
129      *
130      * @param stageId The ID of the stage being
    queried.
131      * @return The stage's length.
132      * @throws IDNotRecognisedException If the ID
    does not match to any stage in the
133      *                                     system.
134      */
135     double getStageLength(int stageId) throws
    IDNotRecognisedException;
136
137     /**
138      * Removes a stage and all its related data, i.e
    ., segments and results.
139      * <p>
140      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
141      * exceptions are thrown.
142      *
143      * @param stageId The ID of the stage being
    removed.
144      * @throws IDNotRecognisedException If the ID
    does not match to any stage in the
145      *                                     system.
```

```
146      */
147     void removeStageById(int stageId) throws
    IDNotRecognisedException;
148 
149     /**
150      * Adds a climb segment to a stage.
151      * <p>
152      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
153      * exceptions are thrown.
154      *
155      * @param stageId         The ID of the stage to
     which the climb segment is
156      *                        being added.
157      * @param location        The kilometre location
     where the climb finishes within
158      *                        the stage.
159      * @param type            The category of the
    climb - {@link SegmentType#C4},
160      *                        {@link SegmentType#C3
    }, {@link SegmentType#C2},
161      *                        {@link SegmentType#C1
    }, or {@link SegmentType#HC}.
162      * @param averageGradient The average gradient
    for the climb.
163      * @param length          The length of the
    climb in kilometre.
164      * @return The ID of the segment created.
165      * @throws IDNotRecognisedException   If the ID
    does not match to any stage in
166      *                                    the system
    .
167      * @throws InvalidLocationException   If the
    location is out of bounds of the
168      *                                    stage
    length.
169      * @throws InvalidStageStateException If the
    stage is "waiting for results".
170      * @throws InvalidStageTypeException  Time-trial
     stages cannot contain any
171      *                                    segment.
```

```
172      */
173     int addCategorizedClimbToStage(int stageId,
    Double location, SegmentType type, Double
    averageGradient,
174             Double length) throws
    IDNotRecognisedException, InvalidLocationException,
    InvalidStageStateException,
175             InvalidStageTypeException;
176
177     /**
178      * Adds an intermediate sprint to a stage.
179      * <p>
180      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
181      * exceptions are thrown.
182      *
183      * @param stageId  The ID of the stage to which
    the intermediate sprint segment
184      *                 is being added.
185      * @param location The kilometre location where
    the intermediate sprint finishes
186      *                 within the stage.
187      * @return The ID of the segment created.
188      * @throws IDNotRecognisedException   If the ID
    does not match to any stage in
189      *                                    the system
    .
190      * @throws InvalidLocationException   If the
    location is out of bounds of the
191      *                                    stage
    length.
192      * @throws InvalidStageStateException If the
    stage is "waiting for results".
193      * @throws InvalidStageTypeException  Time-trial
     stages cannot contain any
194      *                                    segment.
195      */
196     int addIntermediateSprintToStage(int stageId,
    double location) throws IDNotRecognisedException,
197             InvalidLocationException,
    InvalidStageStateException,
```

```
197 InvalidStageTypeException;
198
199     /**
200      * Removes a segment from a stage.
201      * <p>
202      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
203      * exceptions are thrown.
204      *
205      * @param segmentId The ID of the segment to be
    removed.
206      * @throws IDNotRecognisedException   If the ID
    does not match to any segment in
207      *                                    the system
    .
208      * @throws InvalidStageStateException If the
    stage is "waiting for results".
209      */
210     void removeSegment(int segmentId) throws
    IDNotRecognisedException, InvalidStageStateException
    ;
211
212     /**
213      * Concludes the preparation of a stage. After
    conclusion, the stage's state
214      * should be "waiting for results".
215      * <p>
216      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
217      * exceptions are thrown.
218      *
219      * @param stageId The ID of the stage to be
    concluded.
220      * @throws IDNotRecognisedException   If the ID
    does not match to any stage in
221      *                                    the system
    .
222      * @throws InvalidStageStateException If the
    stage is "waiting for results".
223      */
224     void concludeStagePreparation(int stageId)
```

```
224 throws IDNotRecognisedException,
    InvalidStageStateException;
225
226     /**
227      * Retrieves the list of segment (mountains and
    sprints) IDs of a stage.
228      * <p>
229      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
230      * exceptions are thrown.
231      *
232      * @param stageId The ID of the stage being
    queried.
233      * @return The list of segment IDs ordered (from
     first to last) by their location in the
234      *         stage.
235      * @throws IDNotRecognisedException If the ID
    does not match to any stage in the
236      *                                  system.
237      */
238     int[] getStageSegments(int stageId) throws
    IDNotRecognisedException;
239
240     /**
241      * Creates a team with name and description.
242      * <p>
243      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
244      * exceptions are thrown.
245      *
246      * @param name        The identifier name of the
     team.
247      * @param description A description of the team.
248      * @return The ID of the created team.
249      * @throws IllegalNameException If the name
    already exists in the platform.
250      * @throws InvalidNameException If the new name
    is null, empty, has more than
251      *                              30 characters.
252      */
253     int createTeam(String name, String description)
```

```
253 throws IllegalNameException, InvalidNameException;
254
255     /**
256      * Removes a team from the system.
257      * <p>
258      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
259      * exceptions are thrown.
260      *
261      * @param teamId The ID of the team to be
    removed.
262      * @throws IDNotRecognisedException If the ID
    does not match to any team in the
263      *                                  system.
264      */
265     void removeTeam(int teamId) throws
    IDNotRecognisedException;
266
267     /**
268      * Get the list of teams' IDs in the system.
269      * <p>
270      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
271      * exceptions are thrown.
272      *
273      * @return The list of IDs from the teams in the
     system. An empty list if there
274      *         are no teams in the system.
275      *
276      */
277     int[] getTeams();
278
279     /**
280      * Get the riders of a team.
281      * <p>
282      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
283      * exceptions are thrown.
284      *
285      * @param teamId The ID of the team being
    queried.
```

```
286      * @return A list with riders' ID.
287      * @throws IDNotRecognisedException If the ID
    does not match to any team in the
288      *                                  system.
289      */
290     int[] getTeamRiders(int teamId) throws
    IDNotRecognisedException;
291 
292     /**
293      * Creates a rider.
294      * <p>
295      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
296      * exceptions are thrown.
297      *
298      * @param teamID      The ID rider's team.
299      * @param name        The name of the rider.
300      * @param yearOfBirth The year of birth of the
    rider.
301      * @return The ID of the rider in the system.
302      * @throws IDNotRecognisedException If the ID
    does not match to any team in the
303      *                                  system.
304      * @throws IllegalArgumentException If the name
    of the rider is null or the year
305      *                                  of birth is
    less than 1900.
306      */
307     int createRider(int teamID, String name, int
    yearOfBirth) throws IDNotRecognisedException,
    IllegalArgumentException;
308 
309     /**
310      * Removes a rider from the system. When a rider
     is removed from the platform,
311      * all of its results should be also removed.
    Race results must be updated.
312      * <p>
313      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
314      * exceptions are thrown.
```

```
315      *
316      * @param riderId The ID of the rider to be
    removed.
317      * @throws IDNotRecognisedException If the ID
    does not match to any rider in the
318      *                                  system.
319      */
320     void removeRider(int riderId) throws
    IDNotRecognisedException;
321
322     /**
323      * Record the times of a rider in a stage.
324      * <p>
325      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
326      * exceptions are thrown.
327      *
328      * @param stageId     The ID of the stage the
    result refers to.
329      * @param riderId     The ID of the rider.
330      * @param checkpoints An array of times at which
     the rider reached each of the
331      *                    segments of the stage,
    including the start time and the
332      *                    finish line.
333      * @throws IDNotRecognisedException    If the ID
     does not match to any rider or
334      *                                     stage in
    the system.
335      * @throws DuplicatedResultException   Thrown if
     the rider has already a result
336      *                                     for the
    stage. Each rider can have only
337      *                                     one
    result per stage.
338      * @throws InvalidCheckpointsException Thrown if
     the length of checkpoints is
339      *                                     not equal
     to n+2, where n is the number
340      *                                     of
    segments in the stage; +2 represents
```

```
341      *                                          the start
     time and the finish time of the
342      *                                          stage.
343      * @throws InvalidStageStateException  Thrown if
     the stage is not "waiting for
344      *                                          results
    ". Results can only be added to a
345      *                                          stage
    while it is "waiting for results".
346      */
347     void registerRiderResultsInStage(int stageId,
    int riderId, LocalTime... checkpoints) throws
    IDNotRecognisedException,
348             DuplicatedResultException,
    InvalidCheckpointsException,
    InvalidStageStateException;
349
350     /**
351      * Get the times of a rider in a stage.
352      * <p>
353      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
354      * exceptions are thrown.
355      *
356      * @param stageId The ID of the stage the result
     refers to.
357      * @param riderId The ID of the rider.
358      * @return The array of times at which the rider
     reached each of the segments of
359      *         the stage and the total elapsed time
    . The elapsed time is the
360      *         difference between the finish time
    and the start time. Return an
361      *         empty array if there is no result
    registered for the rider in the
362      *         stage.
363      * @throws IDNotRecognisedException If the ID
    does not match to any rider or
364      *                                     stage in the
     system.
365      */
```

```
366     LocalTime[] getRiderResultsInStage(int stageId,
    int riderId) throws IDNotRecognisedException;
367
368     /**
369      * For the general classification, the
    aggregated time is based on the adjusted
370      * elapsed time, not the real elapsed time.
    Adjustments are made to take into
371      * account groups of riders finishing very close
     together, e.g., the peloton. If
372      * a rider has a finishing time less than one
    second slower than the
373      * previous rider, then their adjusted elapsed
    time is the smallest of both. For
374      * instance, a stage with 200 riders finishing "
    together" (i.e., less than 1
375      * second between consecutive riders), the
    adjusted elapsed time of all riders
376      * should be the same as the first of all these
    riders, even if the real gap
377      * between the 200th and the 1st rider is much
    bigger than 1 second. There is no
378      * adjustments on elapsed time on time-trials.
379      * <p>
380      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
381      * exceptions are thrown.
382      *
383      * @param stageId The ID of the stage the result
     refers to.
384      * @param riderId The ID of the rider.
385      * @return The adjusted elapsed time for the
    rider in the stage. Return an empty
386      *         array if there is no result
    registered for the rider in the stage.
387      * @throws IDNotRecognisedException   If the ID
    does not match to any rider or
388      *                                    stage in
    the system.
389      */
390     LocalTime getRiderAdjustedElapsedTimeInStage(int
```

```
390  stageId, int riderId)
391              throws IDNotRecognisedException;
392
393      /**
394       * Removes the stage results from the rider.
395       * <p>
396       * The state of this MiniCyclingPortalInterface
    must be unchanged if any
397       * exceptions are thrown.
398       *
399       * @param stageId The ID of the stage the result
     refers to.
400       * @param riderId The ID of the rider.
401       * @throws IDNotRecognisedException If the ID
    does not match to any rider or
402       *                                    stage in the
     system.
403       */
404      void deleteRiderResultsInStage(int stageId, int
    riderId) throws IDNotRecognisedException;
405
406      /**
407       * Get the riders finished position in a a stage
    .
408       * <p>
409       * The state of this MiniCyclingPortalInterface
    must be unchanged if any
410       * exceptions are thrown.
411       *
412       * @param stageId The ID of the stage being
    queried.
413       * @return A list of riders ID sorted by their
    elapsed time. An empty list if
414       *         there is no result for the stage.
415       * @throws IDNotRecognisedException If the ID
    does not match any stage in the
416       *                                    system.
417       */
418      int[] getRidersRankInStage(int stageId) throws
    IDNotRecognisedException;
419
```

```
420     /**
421      * Get the adjusted elapsed times of riders in a
     stage.
422      * <p>
423      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
424      * exceptions are thrown.
425      *
426      * @param stageId The ID of the stage being
    queried.
427      * @return The ranked list of adjusted elapsed
    times sorted by their finish
428      *         time. An empty list if there is no
    result for the stage. These times
429      *         should match the riders returned by
430      *         {@link #getRidersRankInStage(int)}.
431      * @throws IDNotRecognisedException If the ID
    does not match any stage in the
432      *                                     system.
433      */
434     LocalTime[] getRankedAdjustedElapsedTimesInStage
    (int stageId) throws IDNotRecognisedException;
435
436     /**
437      * Get the number of points obtained by each
    rider in a stage.
438      * <p>
439      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
440      * exceptions are thrown.
441      *
442      * @param stageId The ID of the stage being
    queried.
443      * @return The ranked list of points each riders
     received in the stage, sorted
444      *         by their elapsed time. An empty list
    if there is no result for the
445      *         stage. These points should match the
    riders returned by
446      *         {@link #getRidersRankInStage(int)}.
447      * @throws IDNotRecognisedException If the ID
```

```
447 does not match any stage in the
448      *                                    system.
449      */
450     int[] getRidersPointsInStage(int stageId) throws
     IDNotRecognisedException;
451 
452     /**
453      * Get the number of mountain points obtained by
     each rider in a stage.
454      * <p>
455      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
456      * exceptions are thrown.
457      *
458      * @param stageId The ID of the stage being
    queried.
459      * @return The ranked list of mountain points
    each riders received in the stage,
460      *         sorted by their finish time. An empty
     list if there is no result for
461      *         the stage. These points should match
    the riders returned by
462      *         {@link #getRidersRankInStage(int)}.
463      * @throws IDNotRecognisedException If the ID
    does not match any stage in the
464      *                                    system.
465      */
466     int[] getRidersMountainPointsInStage(int stageId
    ) throws IDNotRecognisedException;
467 
468     /**
469      * Method empties this
    MiniCyclingPortalInterface of its contents and
    resets all
470      * internal counters.
471      */
472     void eraseCyclingPortal();
473 
474     /**
475      * Method saves this MiniCyclingPortalInterface
    contents into a serialised file,
```

```
476      * with the filename given in the argument.
477      * <p>
478      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
479      * exceptions are thrown.
480      *
481      * @param filename Location of the file to be
    saved.
482      * @throws IOException If there is a problem
    experienced when trying to save the
483      *                     store contents to the
    file.
484      */
485     void saveCyclingPortal(String filename) throws
    IOException;
486
487     /**
488      * Method should load and replace this
    MiniCyclingPortalInterface contents with the
489      * serialised contents stored in the file given
    in the argument.
490      * <p>
491      * The state of this MiniCyclingPortalInterface
    must be unchanged if any
492      * exceptions are thrown.
493      *
494      * @param filename Location of the file to be
    loaded.
495      * @throws IOException            If there is a
    problem experienced when trying
496      *                                to load the
    store contents from the file.
497      * @throws ClassNotFoundException If required
    class files cannot be found when
498      *                                loading.
499      */
500     void loadCyclingPortal(String filename) throws
    IOException, ClassNotFoundException;
501
502 }
503
```

```java
package cycling;

/**
 * Thrown when attempting to use a name that does not exit in the
 * system.
 *
 * @author Diogo Pacheco
 * @version 1.0
 *
 */
public class NameNotRecognisedException extends Exception {

    /**
     * Constructs an instance of the exception with no message
     */
    public NameNotRecognisedException() {
        // do nothing
    }

    /**
     * Constructs an instance of the exception containing the message argument
     *
     * @param message message containing details regarding the exception cause
     */
    public NameNotRecognisedException(String message) {
        super(message);
    }

}

```

```java
package cycling;

/**
 * Each race result should contains the times for each segment (mountain and
 * sprint) within a stage, plus the start time and the finish time. The list of checkpoints must
 * follow its chronological sequence, i.e., checkpoint_i {@literal <=} checkpoint_i+1.
 *
 * @author Diogo Pacheco
 * @version 1.0
 *
 */
public class InvalidCheckpointsException extends Exception {

    /**
     * Constructs an instance of the exception with no message
     */
    public InvalidCheckpointsException() {
        // do nothing
    }

    /**
     * Constructs an instance of the exception containing the message argument
     *
     * @param message message containing details regarding the exception cause
     */
    public InvalidCheckpointsException(String message) {
        super(message);
    }

}

```

```java
package cycling.types;

import cycling.StageType;

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

public class Race implements Serializable {
    public int id;
    public String name;
    public String description;
    public StageType type;
    public final ArrayList<Integer> stages = new ArrayList<>();
}

```

```java
package cycling.types;

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

public class Team implements Serializable {
    public int id;
    public String name;
    public String description;
    public final List<Integer> riders = new ArrayList
<>();
}

```

```java
package cycling.types;

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

public class Rider implements Serializable {
    public int id;
    public String name;
    public int yearOfBirth;
    // Integer key is the stage id, the array of times is the rider's times for the respective segments
    public HashMap<Integer, LocalTime[]> results;
}

```

```java
package cycling.types;

import cycling.StageType;

import java.io.Serializable;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

public class Stage implements Serializable {
    public int id;
    public String name;
    public String description;
    public double length;
    public StageType type;
    public LocalDateTime startTime;
    public final List<Integer> segments = new ArrayList<>();
    public StageState state = StageState.SETUP;
}
```

```java
package cycling.types;

import cycling.SegmentType;

import java.io.Serializable;

public class Segment implements Serializable {
    public int id;
    public SegmentType type;
    public double location;
    public double averageGradient;
}

```

```java
package cycling.types;

public enum StageState {
    SETUP,
    WAITING_FOR_RESULTS
}

```