University of Exeter

Faculty of Environment, Science and Economy

ECM1410

Object-Oriented Programming

Coursework

Date Set: 16th February 2024
Date Due: 28th March 2024

This coursework comprises 100% of the overall module assessment.

This is a **pair** exercise, and your attention is drawn to the guidelines on collaboration and plagiarism in the University Handbook (`exeter.ac.uk/students/administration/complaintsandappeals/academicmisconduct/`).

This assessment covers the use and implementation of a range of object-oriented concepts using the Java programming language that you are covering in ECM1410. The assignment is *summative*. Please ensure you read the entire document before you begin the assessment.

This coursework will help you enhance your understanding of object-oriented programming. Remember to follow the established deadline and submission guidelines in the module's ELE page.

1 Development paradigm

This summative coursework for ECM1410 is a pair submission — with details as circulated previously in the document *"ECM1410 Object-Oriented Programming Development paradigm in summative coursework"*. To reiterate the key points from this earlier document:

- The expectation is that the submission will be weighted 50:50 between pair members. In the unusual circumstance that members of the pair do not contribute 50:50, you have the opportunity to indicate a different ratio you have both agreed on the cover page of the submission, up to a maximum divergence of 60:40.
- Pair programming is categorically **not** two developers working separately on two different machines. Side-by-side (or virtual, sharing screen) communication developing on a single machine is a key aspect of the approach.
- **The module leader reserves the right to split pairs where one student is not engaging with the coursework. The coordinator also reserves the right to assign non-contributing students a mark of 0**. In the rare situation that you are paired with a student who is not contributing (e.g. not replying to emails and/or not meeting up for pair-programming sessions) you must inform Dr. Pacheco of the situation **within one week of release of the coursework specification** to facilitate the aforementioned splitting of pairs if necessary. Both parties of a split pair will be assigned an individual variant of the coursework (however, if there are multiple pairs in this situation it may be possible to reform pairs consisting of participating students, and of non-participating students). It is not permitted for a student working on the *individual* variant of the coursework to collaborate with any other student.

  Given the above process and timelines, please ensure that you arrange to meet in person or virtually and start the work as soon as the coursework specification is released to reassure yourself that you are partnered with a student who wants to actively contribute to the coursework. It is an expectation that pairs have *at a minimum* two pair-programming sessions in the first week of release of the coursework.
- It is not permitted for students working on the pair programming assignment to collaborate on the assignment with any other student *apart* from their named partner. Those doing so will be subject to academic misconduct regulations and penalties. Please refer to the undergraduate handbook for details on collusion, plagiarism, etc (see web link on coversheet of this document).

2 Assignment - Coding (100 marks)

This assignment is based around the development of a back-end Java package. The system's required functionality has already been determined, and an *interface* is provided such that your back-end can communicate with the front-end being developed by the module leader. The module leader should be able to simply compile in the jar file of the package you develop, with the rest of their system, to result in a fully functioning solution.

2.1 The problem

Cycling is a very popular sport and it has been shown to cause positive effects beyond the sport's boundaries. Cities engaged with cycling races usually tend to have more commuters. The increasingly usage of bicycles in daily activities has also been related to health improvements.

To foster future athletes and a healthier community, a local cycling club has decided to create a cycling staged race. The idea is to have the same structure and rules as the famous cycling *Grand Tours*. Therefore, the club needs a system to support this and future events.

The system can be split in three parts:

1. **Rider's Management** – the system must be able to manage riders and teams.
2. **Race Design** – the system must have functionalities to allow the creation of stage races. That is, one must be able to manage stages within races and configure them. Among the configurations, for instance, one should be able to define intermediate sprints and mountain summits.
3. **Race Results** – the system must have functionalities to handle the realisation of races following the specified design. In other words, the system must allow recording the results of riders on races.

A **staged race** is a type of cycling race that is distributed over multiple stages raced in different days. There are also multiple competitions co-occurring in a staged race. For simplicity, the club decided to have only three competitions within races. They are listed below, sorted by importance:

1. **General Classification (GC)** – this is the most important prize of the race. The winner is the rider who finishes all stages in the least amount of time in total. Note, this rider may never win a single stage, but still be the fastest when one aggregates the overall times.
2. **Points (Sprinter's) Classification** – in this classification the actual time is not considered, but the ranks of riders. Finishing a stage in the top ranks give riders a certain amount of points depending on the *type* of the stage. In addition to the finish line points, stages can have *intermediate sprints*. These sprints act as intermediate finish lines and riders get points according to their position crossing them. The winner in the points classification is the rider who sums the most points after all stages in a race. See points distribution in Figure 1.
3. **Mountain Classification** – is a special competition which prizes the best climbers in the race. Similar to the sprinter's classification, mountain points are given to riders who first reach the summit of mountains throughout stages. As expected, tougher climbs worth more points. At the end of the race, the *King/Queen of the Mountains* (KOM/QOM) is the rider who collects the most of the mountain points. See mountain points distribution in Figure 2.

Stages represent the diverse courses on which cycling races unfold, each offering unique challenges and opportunities for riders. These stages are classified into four distinct categories: *flat*, *hilly finish/medium mountain*, *high mountain*, and *time-trial*, each requiring different strategies and skillsets from the cyclists. Within these stages, there are checkpoints, such as intermediate sprints and categorized climbs, which contribute points towards the sprinter's and mountains classifications, respectively (see Figure 3 for an example). Notably, time-trial stages stand apart as they lack a mass start, with riders competing individually against the clock, each starting at offset times. Moreover, unlike other stages, time-trials do not feature checkpoint points. For a visual representation of the point distribution for the sprinter's classification, refer to Figure 1, while Figure 2 outlines the points awarded for mountain climbs based on their category.

The club wants the back-end of the new cycling portal to be compatible with the front-end which is being developed by another group, as such they have already designed a Java interface for the new system, which their front-end application will use. You are to develop a class that implements this interface, and also develop the necessary additional supporting classes in the Java package called `cycling`. The operational correctness of the back-end system will be tested through this provided interface on submission.

Your task is to design and write the additional package members to complete the `cycling` package. You will need to design and write a class that implements the `CyclingPortal` interface. Individuals not paired should implement a simplified version defined by the `MiniCyclingPortal` interface. Both interfaces are available on the ECM1410 Assignment ELE page, as well as at the end of this document.

The implementor class **must** be a public class called `CyclingPortalImpl`. If it is not, then the front-end system will be unable to compile with your back-end solution, and the operational component of your mark will be **0**. You will need to also write any other package members you deem appropriate to support this class and its functionality. All classes developed must reside in the `cycling` package. Alongside the interfaces, I have provided in the package a set of exception classes which each interface requires.

Type	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	11th	12th	13th	14th	15th
"flat" stage finish	50	30	20	18	16	14	12	10	8	7	6	5	4	3	2
"hilly finish/medium mountain" stage finish	30	25	22	19	17	15	13	11	9	7	6	5	4	3	2
"high mountain" stage finish	20	17	15	13	11	10	9	8	7	6	5	4	3	2	1
individual time trial	20	17	15	13	11	10	9	8	7	6	5	4	3	2	1
intermediate sprint	20	17	15	13	11	10	9	8	7	6	5	4	3	2	1

Figure 1: Sprint points distribution according to rider's rank for different types of stage and for the intermediate sprint checkpoint. Extracted from `http://en.wikipedia.org/wiki/Points_classification_in_the_Tour_de_France` (accessed 11 February 2022).

Pos.	4C	3C	2C	1C	HC
1st	1	2	5	10	20
2nd		1	3	8	15
3rd			2	6	12
4th			1	4	10
5th				2	8
6th				1	6
7th					4
8th					2

Figure 2: Mountain points distribution according to rider's rank for different types of mountain checkpoints. Extracted from `en.wikipedia.org/wiki/Mountains_classification_in_the_Tour_de_France` (accessed 11 February 2022).

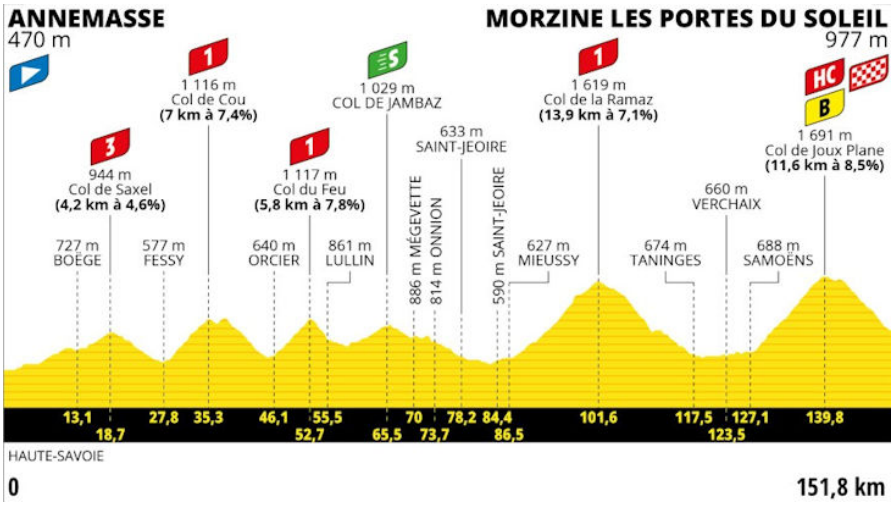

Figure 3: Stage 14 of the 2023 Tour de France. It has five mountain checkpoints and one intermediate sprint. Extracted from `https://www.cyclingstage.com/tour-de-france-2023-route/` (accessed 13 February 2024).

2.2 Development considerations

The following points should be noted:

- Your source code should include appropriate comments and assertions.
- When a rider is removed from the platform, all of its results should be also removed. Race results must be updated.

You will not need to submit an executable application (i.e. you do not need to submit a class with a `public static void main` method within the `cycling` package). This notwithstanding, it is strongly advised that you do write an application to test that your package conforms to the requirements prior to submission. We have provided a skeleton test class (`CyclingPortalTestApp.java`) in the released package.

Apart from the classes you develop yourself, or that you have been given as part of the coursework, you must only use those available in the Java built-in packages (`java.*`). The use of any other packages will result in a **penalty of 10 marks**.

You should consult the `MiniCyclingPortal` and the `CyclingPortal` interfaces for a more detailed description of expected behaviour of a class which implements the interface which is appopriate for you or your pair (provided in the Javadoc).

3 Submission

The coursework requires electronic submission to the ELE platform. Upload your file (see details below) by midday on the due date specified on the cover page of this document. The paths mentioned below (folder structure) should be all lowercase, but the files within the folders should follow the Java naming convention.

Your working environment should have at least five folders:

- `TestSystem` – (optional) containing your test files.
- `src` – to keep the source code (*.java files), including those provided as part of the coursework.
- `bin` – to keep the bytecode (*.class files).
- `doc` – to save the Javadoc from your code.
- `res` – to add any other resource you need in addition to the class diagram and cover page.

1. **Class Diagram** – include a PDF file with a class diagram representing the class `cycling.CyclingPortalImpl` and all auxiliary classes you created. There is no need to include interfaces or exception classes. For each class, include all attributes and public methods. You can use any software to create your diagram, for instance, word, powerpoint, drawio, etc. as long as you export it to a PDF.
2. **Cover Page** – a cover page which details *both* of the student numbers of the corresponding pair. If, unusually, you have agreed a split which is not 50:50, this page should also detail how you would like the final mark to be allocated to the pair, based upon your agreed input. This cannot differ by more than 60:40. As the submission is anonymous, again please use your student numbers. This page should have a development log, which includes date, time and duration of pair programming sessions, including which role(s) each developer took in these sessions, with each log entry identified by both members using your student numbers to ensure anonymous marking. If you are working solo, you are still required to submit the cover page with your student number, but you don't need to add the development log. This file should be named `cover_sheet.pdf`. I have added an example of a cover page in the ELE page[1].

   Failure to submit any coversheet detailing both members student numbers and development log will incur a **penalty of 5 marks**.

[1] Inside the released zip file, go to `res/cover_sheet_example.pdf`. This example highlights *what* information the cover page needs to include. You don't need to follow this exact format.

- `build` – to save the jar file that will be submitted. You need to generate a **jar Package** with a copy of your **full** finished package, named `cycling.jar`. The jar file must include: (i) the bytecode (*.class*), (ii) source files (*.java*) of your submitted package, including the `CyclingPortal` interface and all the exception classes provided to you as part of the coursework, (iii) the Javadoc for the package, and (iv) the cover page and the class diagram. I.e. it should be a complete self-contained package, that my test program can interact with, via your `cycling.CyclingPortalImpl` class.

Assuming you followed the above-mentioned folder structure, here is an example[2] of how you can generate the expected jar package:

```
>> ls
TestSystem    bin    build     doc     res     src
>> javac -d bin/ src/cycling/*.java
>> jar cvf build/cycling.jar -C bin .
(listing files added, omitted output)
>> jar uvf build/cycling.jar -C src .
(listing files added, omitted output)
>> jar uvf build/cycling.jar doc
(listing files added, omitted output)
>> jar uvf build/cycling.jar res
```

You may want to *check* the files within the jar file:

```
>> jar -tf build/cycling.jar
```

or to *extract* its content:

```
>>jar -xvf  build/cycling.jar
```

You can test your package using the `CyclingPortalTestApp.java` class provided with the interface and other files.

```
>> cd TestSystem
>> javac -cp ../build/cycling.jar CyclingPortalTestApp.java
>> java -cp .:../build/cycling.jar CyclingPortalTestApp
The system compiled and started the execution...
```

Finally, you will submit to ELE only the jar file `build/cycling.jar`.

4 Advice

1. Do not jump straight into the coding: take time to consider the design of your solution first. Think about the objects that you will use, the data they will contain, what the methods they should provide are (in addition to those mandated via the interface), how they relate to one another, etc. Once you are happy with your design, then start programming. Don't be afraid to reassess your design as you go through, but check on the implications of making a changes on all the other objects in your system that use the changed part.

2. Check your objects behave as you intend — use a testing application and use assertions.

3. Slowly fill out functionality — it is far better to submit a solution that supplies most but not all of the required operations correctly, rather than one that doesn't provide any/doesn't compile, as a submission which does not provide any correct functionality at all will get a 0 for the operation criteria. Start off with a `CyclingPortalImpl`

[2]This commands were tested in a Mac machine but they should be the same for Linux machines. If you have Windows you can either: use the native *Windows PowerShell* app to use the same commands; or replace the command `ls` by `dir`, the forward slashes "/" by backslashes "\", and colon ":" by semi-colon ";" if you use the *cmd* app.

class that compiles and slowly (incrementally) add functionality. I have provided a class that implements `CyclingPortal` on the released package (called `BadCyclingPortalImpl`) that does just this — it compiles, but provides none of the correct functionality. The solo version is called `BadMiniCyclingPortalImpl`.

4. Keep copies of your working code. If the worst happens and you had a version that worked on 50% of the operations and you've made changes that seem to have broken everything, it is useful to be able to 'roll-back' to the earlier version and try again.

5. **Do not** change the interface and classes that I have provided for you. If you change them, the markers will not be able to compile my codebase with your submission, and you will receive an 'Operation' component mark of **0**, as the interface will not be able to connect to the front-end of the system.

6. Practice creating the jar file using the following GitHub Classroom assignment `https://classroom.github.com/a/icpw4zB4`.

5 Marking Criteria

This assessment will be marked using the following criteria.

Criterion	**Description**	**Marks Available**
Comments & annotations.	The degree of quality and appropriateness of documentation comments, code comments and annotations.	/5
Java conventions.	The degree of adherence to Java naming conventions and formatting. See lecture notes and e.g. `https://google.github.io/styleguide/javaguide.html`	/5
Operation.	The degree to which the provided `CyclingPortalImpl` class operates as required, as supported by the package members. Submission of a jar file that cannot be compiled in with the test code (due to e.g. the interface definition being changed, required package members missing, etc.) will receive an operation mark of 0. Operations are assessed by the delivered functionality. For instance, if the system fails to create a race it will necessarily fail most other functionalities as one would not be able to create checkpoints nor results.	/50
OO design.	The degree to which the code is object-oriented, well structured and presented, with a coherent design and clear and appropriate management of object states, with well encapsulated objects, appropriate distribution of computational load across objects and appropriate use of types and assertions.	/40
Penalty.	Use of non-permitted packages.	−10
Penalty.	Non-submission of coversheet with pair membership details and development log.	−5
Penalty.	Non-submission of classes diagram.	−10

6 MiniCyclingPortal.java

```
package cycling;

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

/**
 * MiniCyclingPortal interface. The no-argument constructor of a class
 * implementing this interface should initialise the MiniCyclingPortal as
 * an empty platform with no initial racing teams nor races within it.
 *
 * @author Diogo Pacheco
 * @version 2.0
 *
 */
public interface MiniCyclingPortal 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 MiniCyclingPortal must be unchanged if any
    * exceptions are thrown.
    *
    * @param name      Race's name.
    * @param description Race's description (can be null).
    * @throws IllegalNameException If the name already exists in the platform.
    * @throws InvalidNameException If the name is null, empty, has more than 30
    *                              characters, or has white spaces.
    * @return the unique ID of the created race.
    *
    */
   int createRace(String name, String description) throws IllegalNameException, InvalidNameException;

   /**
    * Get the details from a race.
    * <p>
    * The state of this MiniCyclingPortal must be unchanged if any
    * exceptions are thrown.
    *
    * @param raceId The ID of the race being queried.
    * @return Any formatted string containing the race ID, name, description, the
    *         number of stages, and the total length (i.e., the sum of all stages'
    *         length).
```

```
     * @throws IDNotRecognisedException If the ID does not match to any race in the
     *                                  system.
     */
    String viewRaceDetails(int raceId) throws IDNotRecognisedException;

    /**
     * The method removes the race and all its related information, i.e., stages,
     * checkpoints, and results.
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param raceId The ID of the race to be removed.
     * @throws IDNotRecognisedException If the ID does not match to any race in the
     *                                  system.
     */
    void removeRaceById(int raceId) throws IDNotRecognisedException;

    /**
     * The method queries the number of stages created for a race.
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param raceId The ID of the race being queried.
     * @return The number of stages created for the race.
     * @throws IDNotRecognisedException If the ID does not match to any race in the
     *                                  system.
     */
    int getNumberOfStages(int raceId) throws IDNotRecognisedException;

    /**
     * Creates a new stage and adds it to the race.
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param raceId    The race which the stage will be added to.
     * @param stageName An identifier name for the stage.
     * @param description A descriptive text for the stage.
     * @param length    Stage length in kilometres.
     * @param startTime The date and time in which the stage will be raced. It
     *                  cannot be null.
     * @param type      The type of the stage. This is used to determine the
     *                  amount of points given to the winner.
     * @return the unique ID of the stage.
     * @throws IDNotRecognisedException If the ID does not match to any race in the
     *                                  system.
     * @throws IllegalNameException  If the name already exists in the platform.
     * @throws InvalidNameException  If the name is null, empty, has more than 30
     *                               characters, or has white spaces.
     * @throws InvalidLengthException If the length is less than 5km.
     */
    int addStageToRace(int raceId, String stageName, String description, double length, LocalDateTime
        startTime, StageType type)
```

```
    throws IDNotRecognisedException, IllegalNameException, InvalidNameException, InvalidLengthException;

/**
 * Retrieves the list of stage IDs of a race.
 * <p>
 * The state of this MiniCyclingPortal must be unchanged if any
 * exceptions are thrown.
 *
 * @param raceId The ID of the race being queried.
 * @return An array of stage IDs ordered (from first to last) by their sequence in the
 *       race or an empty array if none exists.
 * @throws IDNotRecognisedException If the ID does not match to any race in the
 *                                  system.
 */
int[] getRaceStages(int raceId) throws IDNotRecognisedException;

/**
 * Gets the length of a stage in a race, in kilometres.
 * <p>
 * The state of this MiniCyclingPortal must be unchanged if any
 * exceptions are thrown.
 *
 * @param stageId The ID of the stage being queried.
 * @return The stage's length.
 * @throws IDNotRecognisedException If the ID does not match to any stage in the
 *                                  system.
 */
double getStageLength(int stageId) throws IDNotRecognisedException;

/**
 * Removes a stage and all its related data, i.e., checkpoints and results.
 * <p>
 * The state of this MiniCyclingPortal must be unchanged if any
 * exceptions are thrown.
 *
 * @param stageId The ID of the stage being removed.
 * @throws IDNotRecognisedException If the ID does not match to any stage in the
 *                                  system.
 */
void removeStageById(int stageId) throws IDNotRecognisedException;

/**
 * Adds a climb checkpoint to a stage.
 * <p>
 * The state of this MiniCyclingPortal must be unchanged if any
 * exceptions are thrown.
 *
 * @param stageId      The ID of the stage to which the climb checkpoint is
 *                     being added.
 * @param location     The kilometre location where the climb finishes within
 *                     the stage.
 * @param type         The category of the climb - {@link CheckpointType#C4},
 *                     {@link CheckpointType#C3}, {@link CheckpointType#C2},
 *                     {@link CheckpointType#C1}, or {@link CheckpointType#HC}.
 * @param averageGradient The average gradient for the climb.
```

```
     * @param length        The length of the climb in kilometre.
     * @return The ID of the checkpoint created.
     * @throws IDNotRecognisedException If the ID does not match to any stage in
     *                                  the system.
     * @throws InvalidLocationException If the location is out of bounds of the
     *                                  stage length.
     * @throws InvalidStageStateException If the stage is "waiting for results".
     * @throws InvalidStageTypeException Time-trial stages cannot contain any
     *                                  checkpoint.
     */
    int addCategorizedClimbToStage(int stageId, Double location, CheckpointType type, Double averageGradient,
         Double length) throws IDNotRecognisedException, InvalidLocationException,
             InvalidStageStateException, InvalidStageTypeException;

    /**
     * Adds an intermediate sprint to a stage.
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param stageId The ID of the stage to which the intermediate sprint checkpoint
     *                is being added.
     * @param location The kilometre location where the intermediate sprint finishes
     *                within the stage.
     * @return The ID of the checkpoint created.
     * @throws IDNotRecognisedException If the ID does not match to any stage in
     *                                  the system.
     * @throws InvalidLocationException If the location is out of bounds of the
     *                                  stage length.
     * @throws InvalidStageStateException If the stage is "waiting for results".
     * @throws InvalidStageTypeException Time-trial stages cannot contain any
     *                                  checkpoint.
     */
    int addIntermediateSprintToStage(int stageId, double location) throws IDNotRecognisedException,
         InvalidLocationException, InvalidStageStateException, InvalidStageTypeException;

    /**
     * Removes a checkpoint from a stage.
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param checkpointId The ID of the checkpoint to be removed.
     * @throws IDNotRecognisedException If the ID does not match to any checkpoint in
     *                                  the system.
     * @throws InvalidStageStateException If the stage is "waiting for results".
     */
    void removeCheckpoint(int checkpointId) throws IDNotRecognisedException, InvalidStageStateException;

    /**
     * Concludes the preparation of a stage. After conclusion, the stage's state
     * should be "waiting for results".
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any
     * exceptions are thrown.
```

```
 *
 * @param stageId The ID of the stage to be concluded.
 * @throws IDNotRecognisedException If the ID does not match to any stage in
 *                                  the system.
 * @throws InvalidStageStateException If the stage is "waiting for results".
 */
void concludeStagePreparation(int stageId) throws IDNotRecognisedException, InvalidStageStateException;

/**
 * Retrieves the list of checkpoint (mountains and sprints) IDs of a stage.
 * <p>
 * The state of this MiniCyclingPortal must be unchanged if any
 * exceptions are thrown.
 *
 * @param stageId The ID of the stage being queried.
 * @return The list of checkpoint IDs ordered (from first to last) by their location in the
 *        stage.
 * @throws IDNotRecognisedException If the ID does not match to any stage in the
 *                                  system.
 */
int[] getStageCheckpoints(int stageId) throws IDNotRecognisedException;

/**
 * Creates a team with name and description.
 * <p>
 * The state of this MiniCyclingPortal must be unchanged if any
 * exceptions are thrown.
 *
 * @param name      The identifier name of the team.
 * @param description A description of the team.
 * @return The ID of the created team.
 * @throws IllegalNameException If the name already exists in the platform.
 * @throws InvalidNameException If the name is null, empty, has more than 30
 *                              characters, or has white spaces.
 */
int createTeam(String name, String description) throws IllegalNameException, InvalidNameException;

/**
 * Removes a team from the system.
 * <p>
 * The state of this MiniCyclingPortal must be unchanged if any
 * exceptions are thrown.
 *
 * @param teamId The ID of the team to be removed.
 * @throws IDNotRecognisedException If the ID does not match to any team in the
 *                                  system.
 */
void removeTeam(int teamId) throws IDNotRecognisedException;

/**
 * Get the list of teams' IDs in the system.
 * <p>
 * The state of this MiniCyclingPortal must be unchanged if any
 * exceptions are thrown.
 *
```

```
     * @return The list of IDs from the teams in the system. An empty list if there
     *         are no teams in the system.
     *
     */
    int[] getTeams();

    /**
     * Get the riders of a team.
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param teamId The ID of the team being queried.
     * @return A list with riders' ID.
     * @throws IDNotRecognisedException If the ID does not match to any team in the
     *                                  system.
     */
    int[] getTeamRiders(int teamId) throws IDNotRecognisedException;

    /**
     * Creates a rider.
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param teamID    The ID rider's team.
     * @param name      The name of the rider.
     * @param yearOfBirth The year of birth of the rider.
     * @return The ID of the rider in the system.
     * @throws IDNotRecognisedException If the ID does not match to any team in the
     *                                  system.
     * @throws IllegalArgumentException If the name of the rider is null or empty,
     *                                  or the year of birth is less than 1900.
     */
    int createRider(int teamID, String name, int yearOfBirth) throws IDNotRecognisedException,
        IllegalArgumentException;

    /**
     * Removes a rider from the system. When a rider is removed from the platform,
     * all of its results should be also removed. Race results must be updated.
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param riderId The ID of the rider to be removed.
     * @throws IDNotRecognisedException If the ID does not match to any rider in the
     *                                  system.
     */
    void removeRider(int riderId) throws IDNotRecognisedException;

    /**
     * Record the times of a rider in a stage.
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any
     * exceptions are thrown.
```

```
     *
     * @param stageId   The ID of the stage the result refers to.
     * @param riderId   The ID of the rider.
     * @param checkpointTimes An array of times at which the rider reached each of the
     *                  checkpoints of the stage, including the start time and the
     *                  finish line.
     * @throws IDNotRecognisedException If the ID does not match to any rider or
     *                                  stage in the system.
     * @throws DuplicatedResultException Thrown if the rider has already a result
     *                                  for the stage. Each rider can have only
     *                                  one result per stage.
     * @throws InvalidCheckpointTimesException Thrown if the length of checkpointTimes is
     *                                  not equal to n+2, where n is the number
     *                                  of checkpoints in the stage; +2 represents
     *                                  the start time and the finish time of the
     *                                  stage.
     * @throws InvalidStageStateException Thrown if the stage is not "waiting for
     *                                  results". Results can only be added to a
     *                                  stage while it is "waiting for results".
     */
    void registerRiderResultsInStage(int stageId, int riderId, LocalTime... checkpointTimes)
         throws IDNotRecognisedException, DuplicatedResultException, InvalidCheckpointTimesException,
             InvalidStageStateException;

    /**
     * Get the times of a rider in a stage.
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any exceptions are
     * thrown.
     *
     * @param stageId The ID of the stage the result refers to.
     * @param riderId The ID of the rider.
     * @return The array of times at which the rider reached each of the checkpoints
     *        of the stage and the total elapsed time. The elapsed time is the
     *        difference between the finish time and the start time. Return an
     *        empty array if there is no result registered for the rider in the
     *        stage. Assume the total elapsed time of a stage never exceeds 24h
     *        and, therefore, can be represented by a LocalTime variable. There is
     *        no need to check for this condition or raise any exception.
     * @throws IDNotRecognisedException If the ID does not match to any rider or
     *                               stage in the system.
     */
    LocalTime[] getRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException;

    /**
     * For the general classification, the aggregated time is based on the adjusted
     * elapsed time, not the real elapsed time. Adjustments are made to take into
     * account groups of riders finishing very close together, e.g., the peloton. If
     * a rider has a finishing time less than one second slower than the
     * previous rider, then their adjusted elapsed time is the smallest of both. For
     * instance, a stage with 200 riders finishing "together" (i.e., less than 1
     * second between consecutive riders), the adjusted elapsed time of all riders
     * should be the same as the first of all these riders, even if the real gap
     * between the 200th and the 1st rider is much bigger than 1 second. There is no
     * adjustments on elapsed time on time-trials.
```

```
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param stageId The ID of the stage the result refers to.
     * @param riderId The ID of the rider.
     * @return The adjusted elapsed time for the rider in the stage. Return null if
     *         there is no result registered for the rider in the stage.
     * @throws IDNotRecognisedException If the ID does not match to any rider or
     *                                  stage in the system.
     */
    LocalTime getRiderAdjustedElapsedTimeInStage(int stageId, int riderId)
          throws IDNotRecognisedException;

    /**
     * Removes the stage results from the rider.
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param stageId The ID of the stage the result refers to.
     * @param riderId The ID of the rider.
     * @throws IDNotRecognisedException If the ID does not match to any rider or
     *                                  stage in the system.
     */
    void deleteRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException;

    /**
     * Get the riders finished position in a a stage.
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param stageId The ID of the stage being queried.
     * @return A list of riders ID sorted by their elapsed time. An empty list if
     *         there is no result for the stage.
     * @throws IDNotRecognisedException If the ID does not match any stage in the
     *                                  system.
     */
    int[] getRidersRankInStage(int stageId) throws IDNotRecognisedException;

    /**
     * Get the adjusted elapsed times of riders in a stage.
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any exceptions are
     * thrown.
     *
     * @param stageId The ID of the stage being queried.
     * @return The ranked list of adjusted elapsed times sorted by their finish
     *         time. An empty list if there is no result for the stage. These times
     *         should match the riders returned by
     *         {@link #getRidersRankInStage(int)}. Assume the total elapsed time of
     *         in a stage never exceeds 24h and, therefore, can be represented by a
     *         LocalTime variable. There is no need to check for this condition or
     *         raise any exception.
```

```
     * @throws IDNotRecognisedException If the ID does not match any stage in the
     *                                  system.
     */
    LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId) throws IDNotRecognisedException;

    /**
     * Get the number of points obtained by each rider in a stage.
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param stageId The ID of the stage being queried.
     * @return The ranked list of points each riders received in the stage, sorted
     *         by their elapsed time. An empty list if there is no result for the
     *         stage. These points should match the riders returned by
     *         {@link #getRidersRankInStage(int)}.
     * @throws IDNotRecognisedException If the ID does not match any stage in the
     *                                  system.
     */
    int[] getRidersPointsInStage(int stageId) throws IDNotRecognisedException;

    /**
     * Get the number of mountain points obtained by each rider in a stage.
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param stageId The ID of the stage being queried.
     * @return The ranked list of mountain points each riders received in the stage,
     *         sorted by their finish time. An empty list if there is no result for
     *         the stage. These points should match the riders returned by
     *         {@link #getRidersRankInStage(int)}.
     * @throws IDNotRecognisedException If the ID does not match any stage in the
     *                                  system.
     */
    int[] getRidersMountainPointsInStage(int stageId) throws IDNotRecognisedException;

    /**
     * Method empties this MiniCyclingPortal of its contents and resets all
     * internal counters.
     */
    void eraseCyclingPortal();

    /**
     * Method saves this MiniCyclingPortal contents into a serialised file,
     * with the filename given in the argument.
     * <p>
     * The state of this MiniCyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param filename Location of the file to be saved.
     * @throws IOException If there is a problem experienced when trying to save the
     *                     store contents to the file.
     */
    void saveCyclingPortal(String filename) throws IOException;
```

```
   /**
    * Method should load and replace this MiniCyclingPortal contents with the
    * serialised contents stored in the file given in the argument.
    * <p>
    * The state of this MiniCyclingPortal must be unchanged if any
    * exceptions are thrown.
    *
    * @param filename Location of the file to be loaded.
    * @throws IOException         If there is a problem experienced when trying
    *                             to load the store contents from the file.
    * @throws ClassNotFoundException If required class files cannot be found when
    *                             loading.
    */
   void loadCyclingPortal(String filename) throws IOException, ClassNotFoundException;

}
```

7 CyclingPortal.java

```
package cycling;

import java.time.LocalTime;

/**
 * CyclingPortal interface. The no-argument constructor of a class
 * implementing this interface should initialise the CyclingPortal as
 * an empty platform with no initial racing teams nor races within it. For pair
 * submissions ONLY.
 *
 * @author Diogo Pacheco
 * @version 2.0
 *
 */
public interface CyclingPortal extends MiniCyclingPortal {

   /**
    * The method removes the race and all its related information, i.e., stages,
    * checkpoints, and results.
    * <p>
    * The state of this CyclingPortal 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.
    * <p>
    * The state of this CyclingPortal must be unchanged if any
    * exceptions are thrown.
```

```
     *
     * @param raceId The ID of the race being queried.
     * @return A ranked list of riders' IDs sorted ascending by the sum of their
     *         adjusted elapsed times in all stages of the race. That is, the first
     *         in this list is the winner (least time). An empty list if there is no
     *         result for any stage in the race.
     * @throws IDNotRecognisedException If the ID does not match any race in the
     *                                  system.
     */
    int[] getRidersGeneralClassificationRank(int raceId) throws IDNotRecognisedException;

    /**
     * Get the general classification times of riders in a race.
     * <p>
     * The state of this CyclingPortal must be unchanged if any exceptions are
     * thrown.
     *
     * @param raceId The ID of the race being queried.
     * @return A list of riders' times sorted by the sum of their adjusted elapsed
     *         times in all stages of the race. An empty list if there is no result
     *         for any stage in the race. These times should match the riders
     *         returned by {@link #getRidersGeneralClassificationRank(int)}. Assume
     *         the total elapsed time of a race (the sum of all of its stages) never
     *         exceeds 24h and, therefore, can be represented by a LocalTime
     *         variable. There is no need to check for this condition or raise any
     *         exception.
     * @throws IDNotRecognisedException If the ID does not match any race in the
     *                                  system.
     */
    LocalTime[] getGeneralClassificationTimesInRace(int raceId) throws IDNotRecognisedException;

    /**
     * Get the overall points of riders in a race.
     * <p>
     * The state of this CyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param raceId The ID of the race being queried.
     * @return An array of riders' points (i.e., the sum of their points in all stages
     *         of the race), sorted by the total adjusted elapsed time. An empty array if
     *         there is no result for any stage in the race. These points should
     *         match the riders returned by {@link #getRidersGeneralClassificationRank(int)}.
     * @throws IDNotRecognisedException If the ID does not match any race in the
     *                                  system.
     */
    int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException;

    /**
     * Get the overall mountain points of riders in a race.
     * <p>
     * The state of this CyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param raceId The ID of the race being queried.
     * @return An array of riders' mountain points (i.e., the sum of their mountain
```

```
     *        points in all stages of the race), sorted by the total adjusted elapsed time.
     *        An empty array if there is no result for any stage in the race. These
     *        points should match the riders returned by
     *        {@link #getRidersGeneralClassificationRank(int)}.
     * @throws IDNotRecognisedException If the ID does not match any race in the
     *                                  system.
     */
    int[] getRidersMountainPointsInRace(int raceId) throws IDNotRecognisedException;

    /**
     * Get the ranked list of riders based on the points classification in a race.
     * <p>
     * The state of this CyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param raceId The ID of the race being queried.
     * @return A ranked list of riders' IDs sorted descending by the sum of their
     *        points in all stages of the race. That is, the first in this list is
     *        the winner (more points). An empty list if there is no result for any
     *        stage in the race.
     * @throws IDNotRecognisedException If the ID does not match any race in the
     *                                  system.
     */
    int[] getRidersPointClassificationRank(int raceId) throws IDNotRecognisedException;

    /**
     * Get the ranked list of riders based on the mountain classification in a race.
     * <p>
     * The state of this CyclingPortal must be unchanged if any
     * exceptions are thrown.
     *
     * @param raceId The ID of the race being queried.
     * @return A ranked list of riders' IDs sorted descending by the sum of their
     *        mountain points in all stages of the race. That is, the first in this
     *        list is the winner (more points). An empty list if there is no result
     *        for any stage in the race.
     * @throws IDNotRecognisedException If the ID does not match any race in the
     *                                  system.
     */
    int[] getRidersMountainPointClassificationRank(int raceId) throws IDNotRecognisedException;

}
```