```java
package cycling;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.ProcessBuilder.Redirect.Type;
import java.time.Duration;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;
import java.util.*;
import java.util.Map.Entry;
import java.util.stream.Collectors;

import java.util.HashMap;
import java.util.ArrayList;


/**
 * CyclingPortal is a minimally compiling, but non-functioning implementor
 * of the CyclingPortalInterface interface.
 *
 * @author Kaloyan Gaydarov
 * @author Taariq Fadhill
 * @version 4.20
 * @since 14/02/2022
 *
 */
public class CyclingPortal implements CyclingPortalInterface {
  //------------- Initialization of ArrayLists used to store the objects -----------------//
  private ArrayList<Race> raceArray = new ArrayList<>();
  private ArrayList<Team> teamArray = new ArrayList<>();
  private ArrayList<Rider> riderArray = new ArrayList<>();

  //-------------------------------RACE METHODS--------------------------------//

  /**
   * This method is used to initialise a new session of the cycling portal
   *
   */
  public CyclingPortal() {
        teamArray = new ArrayList<>();
        raceArray = new ArrayList<>();
    riderArray = new ArrayList<>();
    }

  /**
   * This method is used to find a race using its ID
   *
   * @param id The ID of the race that is being searched up.
   * @return The race object of the given ID.
   * @return The race object of the given ID.
   * @return Else it returns an empty race object.
   */
  private Race getRaceByID(int id) {
        for (Race race : raceArray) {
      if (race.getRaceID() == id) {
        return race;
      }
    }
        return new Race();
    }

  /**
   * This method is used to find a race using the ID of a given stage.
   *
   * @param id The ID of the stage that is being searched up.
   * @return The race object of the given stage ID.
   * @return Else it returns an empty race object.
   */
  private Race getRaceByStageId(int id) {
        for (Race race : raceArray) {
            for (Stage stage : race.getStages()) {
          if (stage.getStageId() == id) {
            return race;
          }
        }
    }
    return new Race();
  }
 /**
   * This method is used to get a stage from the ID of a given segment.
   *
   * @param id The ID of the segment that is being searched up.
   * @return The stage object of the given segment ID.
   * @return Empty stage object if Null.
   */
  private Stage getStageBySegmentId(int id) {
    for (Race race: raceArray){
      for (Stage stage : race.getStages()) {
        for (Segment segment : stage.getStageSegments()) {
          if (segment.getSegmentID() == id) {
            return stage;
          }
        }
      }
    }
    return new Stage();
  }

  /**
   * This method is used to find a stage using its ID
   *
   * @param id The ID of the stage that is being searched up.
   * @return The stage object of the given ID.
   * @return Else it returns an empty stage object.
   */
  */
 private Stage getStageByID(int id) {
  for (Race race : raceArray){
   for (Stage stage : race.getStages()) {
    if (stage.getStageId() == id) {
     return stage;
    }
   }
  }
    return new Stage();
  }

 /**
  * This method is used to find a segment using its ID
  *
  * @param id The ID of the segment that is being searched up.
  * @return The segment object of the given ID.
  * @return Else it returns an empty segment object.
  */
 private Segment getSegmentByID(int id) {
  for (Race race : raceArray) {
   for (Stage stage : race.getStages()) {
    for (Segment segment : stage.getStageSegments()) {
     if (segment.getSegmentID() == id) {
      return segment;
     }
    }
   }
  }
    return new Segment();
  }

 /**
  * This method is used to find a team using its ID
  *
  * @param id The ID of the team that is being searched up.
  * @return The team object of the given ID.
  * @return Else it returns an empty team object.
  */
 private Team getTeamByID(int id) {
    for (Team team : teamArray) {
   if (team.getTeamID() == id) {
    return team;
   }
  }
    return new Team();
  }

 /**
  * This method is used to find a rider using its ID
  *
  * @param id The ID of the rider that is being searched up.
  * @return The rider object of the given ID.
  * @return Else it returns an empty rider object.
  */
 private Rider getRiderByID(int id) {
```

```java
  private Rider getRiderByID(int id) {
        for (Rider rider: riderArray) {
      if (rider.getRiderID() == id) {
        return rider;
      }
    }
        return new Rider();
    }

  //-------------------------------PORTAL METHODS-------------------------------//

  /**
   * This method is used to get all race IDs on the platform
   *
   * @return Array of int containing the race IDs
   */
  @Override
  public int[] getRaceIds() {
    //Temp array of integers to hold all IDs
    int[] raceIDs = new int[raceArray.size()];
    //Add each race ID to the array of integers and then return the array
    for (int i=0; i<raceArray.size(); i++) {
            raceIDs[i] = raceArray.get(i).getRaceID();
        }
    return raceIDs;
  }

  /**
   * This method is used to create a staged race with a given name and description.
   *
   * @param name The name of the race.
   * @param description The description of the race (Can be null).
   * @return The unique ID of the creted race.
   * @exception IllegalNametException Thrown when attempting to assign a race name already in use in the system.
   * @exception InvalidNameException If the name is null, empty, has more than 30 characters, or has white spaces.
   */
  @Override
  public int createRace(String name, String description) throws IllegalNameException, InvalidNameException {
    // Check name matches the requirements needed
    if((name == null) || (name.length() > 30) || (name.contains(" ") || (name == ""))) {
      throw new InvalidNameException("Race name doesn't match the requirements");
    }
    // Checks race Name doesnt exists already
    for(int i = 0;i < raceArray.size(); i++) {
      if (name.equals(raceArray.get(i).getRaceName())) {
        throw new IllegalNameException("Race name " + name + " already exists");
      }
    }
    }
    //Create a new race object
    Race race = new Race(name , description);
    raceArray.add(race);
    return race.getRaceID();
  }

  /**
   * This method is used to view the race details and get them returned.
   *
   * @param raceId The unique id of the race to see its details.
   * @return String of all details of the race concatenated together.
   * @exception IDNotRecognisedException The ID of the race is not existint
in the platform.
   */
  @Override
  public String viewRaceDetails(int raceId) throws IDNotRecognisedException
{
    // If raceId doesnt relate to a stage Name, throw exception
    if(getRaceByID(raceId).getRaceName() == "Null"){
      throw new IDNotRecognisedException("ID "+ raceId + " doesnt exist.");
    }else{
      // Else return a string with all details for the race
      return getRaceByID(raceId).getRaceDetails();
    }
  }

  /**
   * This method is used to remoove a race by its ID.
   *
   * @param raceId The unique id of the race to be deleted.
   * @exception IDNotRecognisedException The ID of the race to be deleted
does not exist.
   */
  @Override
  public void removeRaceById(int raceId) throws IDNotRecognisedException {
    // Checks if race exists with the given ID then removes it
    if(getRaceByID(raceId).getRaceName() == "Null"){
      throw new IDNotRecognisedException("ID " + raceId + " does not exist't
exist");
    }else{
      Race race = getRaceByID(raceId);
      Stage[] stages = race.getStagesV2();
      for(Stage stage: stages){
        Segment[] segments = stage.getStageSegments();
        for(Segment segment: segments){
          try{
            removeSegment(segment.getSegmentID());
          } catch (Exception e){
            System.out.println(e);
          }
        }
        removeStageById(stage.getStageId());
      }
    raceArray.remove(raceId-1);
```

```java
    }
  }

  /**
   * This method is used to get the number of stages in a race.
   *
   * @param raceId The unique id of the race to see the number of stages.
   * @return Integer of number of stages.
   * @exception IDNotRecognisedException The ID of the race is not existint in the platform.
   */
  @Override
  public int getNumberOfStages(int raceId) throws IDNotRecognisedException {
    // If raceId doesnt relate to a stage Name, throw exception
    if(getRaceByID(raceId).getRaceName() == "Null"){
      throw new IDNotRecognisedException("ID "+ raceId + " doesnt exist.");
    }else{
      // Else returns the number of stages
      assert(getRaceByID(raceId).getNumberOfStages() < 0):"Number of stages is invalid";
      return getRaceByID(raceId).getNumberOfStages();
    }
  }

  /**
   * This method is used to add a stage to a race.
   *
   * @param raceId The unique id of the race to see its details.
   * @param stageName The name of the stage to be added to the race.
   * @param description The description of the stage to be added to the race.
   * @param length The length in kilometres to be added to the race.
   * @param stageTime The start time of the stage to be added to the race.
   * @param type The type of the stage to be added to the race.
   * @return The ID of the stage that is added to the race.
   * @exception IDNotRecognisedException The ID of the race is not existint in the platform.
   * @exception IllegalNameException The name of the stage already exists in the platform.
   * @exception InvalidNameException The name of the stage does not meet the requirements.
   * @exception InvalidLengthException The length of the stage must be larger than 5 kilometres.
   */
  @Override
  public int addStageToRace(int raceId, String stageName, String description, double length, LocalDateTime startTime,
      StageType type)
      throws IDNotRecognisedException, IllegalNameException, InvalidNameException, InvalidLengthException {
        // Check parameters
        if ((stageName == null) || (stageName.isEmpty()) || (stageName.length() > 30)){
          throw new InvalidNameException("Stage name does not meet requirements");
        }
        // Check length
        if (length<5) {
          throw new InvalidLengthException("Stage length cannot be less than 5 km");
        }
        // Check stage Name doesnt exist already
        for (Race race : raceArray) {
          for (Stage stage : race.getStages()) {
            if (stage.getStageName().equals(stageName)) {
              throw new IllegalNameException("Stage name " + stageName + " already exists");
            }
          }
        }
        // Temp holding the race object with the raceId given
        Race raceTemp  = getRaceByID(raceId);
        // Check raceId exists
        if (getRaceByID(raceId).getRaceName() == "Null"){
          throw new IDNotRecognisedException("ID "+ raceId + " doesnt exist.");
        } else {
          // Add stage to the race object
          Stage stage = new Stage(stageName, description, length, startTime,type);
          raceTemp.addStage(stage);
          return stage.getStageId();
        }
      }

  /**
   * This method is used to get a race stage.
   *
   * @param raceId The unique id of the race to see its details.
   * @return The ID of the stage that is added to the race.
   */
  @Override
  public int[] getRaceStages(int raceId) throws IDNotRecognisedException {
    // Gets the race objects from the given race id
    Race race = getRaceByID(raceId);
    // Uses race object to get all the stages
        Stage[] stages = race.getStagesV2();
        int[] stageIds = new int[stages.length];
    // For each stage get its ID and adds it to an array of integers
    assert(stages.length < 0):"Invalid amount of stages";
        for (int i=0; i<stages.length; i++) {
            stageIds[i] = stages[i].getStageId();
        }
    return stageIds;
  }

  /**
   * This method is used to get a race stage.
   *
   * @param raceId The unique id of the race to see its details.
```

```java
	 * @return The ID of the stage that is added to the race.
	 */
	@Override
	public double getStageLength(int stageId) throws IDNotRecognisedException
{
		// If stageId doesnt relate to a stage Name, throw exception
		if(getStageByID(stageId).getStageName() == "Null"){
			throw new IDNotRecognisedException("ID "+ stageId + " doesnt exist.");
		} else {
			// Else returns the length of the stage
			assert(getStageByID(stageId).getStageLength() < 0):"Invalid length of
stage";
			return getStageByID(stageId).getStageLength();
		}
	}
	/**
	 * This method is used to remove a race by its ID.
	 *
	 * @param raceId The unique id of the race to be deleted.
	 * @exception IDNotRecognisedException The ID of the race to be deleted
does not exist.
	 */
	@Override
	public void removeStageById(int stageId) throws IDNotRecognisedException {
		// If stageId doesnt relate to a stage Name, throw exception
		if(getStageByID(stageId).getStageName() == "Null"){
			throw new IDNotRecognisedException("ID "+ stageId + " doesnt exist.");
		} else {
			// Else removes the stage
			getRaceByStageId(stageId).removeStage(getStageByID(stageId));
		}
	}

	/**
	 * This method is used to add a climb segment to a stage.
	 *
	 * @param stageId The stage ID for the the segement to be added to the
stage.
	 * @param location The kilometre location of the segment finishes to be
added to the stage.
	 * @param type The category of the climb to be added to the stage.
	 * @param averageGradient The average gradient of the segment to be added
to the stage.
	 * @param length The length of the segment in kilometres to be added to
the stage.
	 * @return The ID of the segment which was added to the stage.
	 * @exception IDNotRecognisedException The ID of the stage is not existsnt
in the platform.
	 * @exception InvalidLocationException The location of the segment is out
of the bounds of the stage length.
	 * @exception InvalidStageStateException The stage is in the process of
receiving a result and can't receive this results.
	 * @exception InvalidStageTypeException A time-trial stage cannot contain
any segments.
	 */
  @Override
  public int addCategorizedClimbToStage(int stageId, Double location, SegmentType type, Double averageGradient,
      Double length) throws IDNotRecognisedException, InvalidLocationException, InvalidStageStateException,
      InvalidStageTypeException {
    //Check if stage exists
    if (getStageByID(stageId).getStageName() == "Null"){
      throw new IDNotRecognisedException("ID "+ stageId + " doesnt exist.");
    } else {
      //Define the stage to be added to
      Stage stage = getStageByID(stageId);
      //Check if stage location allows the length to be added and is large than zero
      if (stage.getStageLength() < location){
        throw new InvalidLocationException("Location is out of bounds of the stage length by " + (location-(stage.getStageLength())));
      }
      if (location < 0){
        throw new InvalidLocationException("Location must be more than zero.");
      }
      //Check if stage is time-trialed
      if (stage.getStageType() == StageType.TT) {
        throw new InvalidStageTypeException("Time-trial stages cannot contain any segment.");
      }
      //Check if stage is prepared to get results added to
      if (!stage.isPrepared()) {
        throw new InvalidStageStateException("Stage cannot be added as its not prepaired.");
      }
      //Add segment to the objects
      Segment segment = new Segment(location,type,averageGradient,length);
      stage.addStageSegment(segment);
      return segment.getSegmentID();
    }
  }

  /**
   * This method is used to add a intermediate sprint segment to a stage.
   *
   * @param stageId The stage ID for the the segement to be added to the stage.
   * @param location The kilometre location of the segment finishes to be added to the stage.
   * @return The ID of the segment which was added to the stage.
   * @exception IDNotRecognisedException The ID of the stage is not existsnt in the platform.
   * @exception InvalidLocationException The location of the segment is out of the bounds of the stage length.
   * @exception InvalidStageStateException The stage is in the process of receiving a result and can't receive this results.
   * @exception InvalidStageTypeException A time-trial stage cannot contain any segments.
   */
  @Override
  public int addIntermediateSprintToStage(int stageId, double location) throws IDNotRecognisedException,
      InvalidLocationException, InvalidStageStateException, InvalidStageTypeException {
    //Check if stage exists
    if (getStageByID(stageId).getStageName() == "Null"){
      throw new IDNotRecognisedException("ID "+ stageId + " doesnt exist.");
    } else {
      //Define the stage to be added to
      Stage stage = getStageByID(stageId);
      //Check if stage location allows the length to be added and is large than zero
      if (stage.getStageLength() < location){
        throw new InvalidLocationException("Location is out of bounds of the stage length by " + (location-(stage.getStageLength())));
      }
      if (location < 0){
        throw new InvalidLocationException("Location must be more than zero.");
      }
      //Check if stage is not time-trialed
      if (stage.getStageType() == StageType.TT) {
        throw new InvalidStageTypeException("Time-trial stages cannot contain any segment.");
      }
      //Check if stage is prepared to get results added to
      if (!stage.isPrepared()) {
        throw new InvalidStageStateException("Stage cannot be added as its not prepaired.");
      }

      //Add segment to the objects
      Segment segment = new Segment(location,SegmentType.SPRINT);
      stage.addStageSegment(segment);
      return segment.getSegmentID();
    }
  }

  /**
   * This method is used to remove a segement using its ID.
   *
   * @param segmentId The unique id of the segment to be deleted.
   * @exception IDNotRecognisedException The ID of the segment to be deleted does not exist.
   * @exception InvalidStageStateException The stage is in the process of receiving a result and can't remove a stage.
   */
  @Override
  public void removeSegment(int segmentId) throws IDNotRecognisedException, InvalidStageStateException {
    //Check if segment exists
    if (getSegmentByID(segmentId).getSegmentID() == 0){
      throw new IDNotRecognisedException("ID "+ segmentId + " doesnt
exist.");
    } else {
      //Find the stage the segment belongs to.
      Stage stage = getStageBySegmentId(segmentId);
      //Check if the stage is waiting for a result.
      if (!stage.isPrepared()){
        throw new InvalidStageStateException("Stage cannot be removed as its not prepaired.");
      }
      //Then remove the segment from the stage.
      stage.removeStageSegment(getSegmentByID(segmentId));
    }
  }

  /**
   * This method is used to prepare a stage.
   *
   * @param stagetId The unique id of the stage to be preapred.
   * @exception IDNotRecognisedException The ID of the stage to be preapred does not exist.
   * @exception InvalidStageStateException The stage is in the process of receiving a result and can't be prepared.
   */
  @Override
  public void concludeStagePreparation(int stageId) throws IDNotRecognisedException, InvalidStageStateException {
    //Check if stage exists
    if (getStageByID(stageId).getStageName() == "Null"){
      throw new IDNotRecognisedException("ID "+ stageId + " doesnt exist.");
    } else {
      //Check if stage is waiting for a result
      if (!getStageByID(stageId).isPrepared()) {
        throw new InvalidStageStateException("Stage is processing a result.");
      } else {
        //Set stage as prepared.
        getStageByID(stageId).prepare();
      }
    }
  }

  /**
   * This method is used to get the IDs of the segments in a given stage by its ID.
   *
   * @param stageId The unique id of the stage which its segments are wanted.
   * @return A integer array of the IDs of the segments in the stage.
   * @exception IDNotRecognisedException The ID of the stage which its segments are wanted does not exist in the system.
   */
  @Override
  public int[] getStageSegments(int stageId) throws IDNotRecognisedException {
    //Check if stage exists
    //Check if stage exists
    if (getStageByID(stageId).getStageName() == "Null"){
      throw new IDNotRecognisedException("ID "+ stageId + " doesnt exist.");
    } else {
      //Get segments from the stage and store in array of segments
      Segment[] segments = getStageByID(stageId).getStageSegments();
      assert(segments.length < 0):"Invalid amount of segments";
      //Set a temp array of integers which will hold the IDs of the segments which is the length of the array of segments
      int[] segmentIDs = new int[segments.length];
      // For each segment add its ID to array of IDs
      for (int i = 0; i < segmentIDs.length; i++){
        segmentIDs[i] = segments[i].getSegmentID();
      }
      //Return the integer array of segment IDs.
      return segmentIDs;
    }
  }

  /**
   * This method is used to create a new team.
   *
   * @param name The name of the team.
   * @param description The description of the team.
   * @return The unique ID of the creted team.
   * @exception IllegalNametException Thrown when attempting to assign a team name already in use in the system.
   * @exception InvalidNameException If the name is null, empty, has more than 30 characters, or has white spaces.
   */
  @Override
  public int createTeam(String name, String description) throws IllegalNameException, InvalidNameException {
    //Check if the team name already exists
    for(int i = 0;i < teamArray.size(); i++) {
      if (teamArray.get(i).getTeamName() == name) {
        throw new IllegalNameException("Team name already exists in the system");
      }
    }
    //Check if team name meets the requirements
    if((name == null) || (name.length() > 30) || (name.contains(" ") || (name == ""))) {
      throw new InvalidNameException("Team name cannot be null, empty, longer than 30 characters or contain a white space");
    }
    //Create a new team object
    Team team = new Team(name , description);
    teamArray.add(team);
    return team.getTeamID();

  }

  /**
   * This method is used to remove a team.
   *
   *
   * @param teamId The team ID to be removed.
   * @exception IDNotRecognisedException The ID of the team to be removed does not exist.
   */
  @Override
  public void removeTeam(int teamId) throws IDNotRecognisedException {
    //Check if the teamId exists
    if (getTeamByID(teamId).getTeamName() == "Null"){
      throw new IDNotRecognisedException("ID "+ teamId + " doesnt exist.");
    } else {
      //Remove team from array
      teamArray.remove(getTeamByID(teamId));
    }
  }

  /**
   * This method is used to get all team IDs on the platform.
   *
   * @return Array of integers containing the team IDs.
   */
  @Override
  public int[] getTeams() {
    //Temp array of integers to hold all IDs
    int[] teamIDs = new int[teamArray.size()];
    //Add each team ID to the array of integers and then return the array
    for (int i=0; i<teamArray.size(); i++) {
      teamIDs[i] = teamArray.get(i).getTeamID();
    }
    return teamIDs;
  }

  /**
   * This method is used to get all rider IDs in the team.
   *
   * @param teamId The team ID which all riders that are a part of are wanted.
   * @return Array of integers containing the rider IDs that are a part of this team.
   * @exception IDNotRecognisedException The ID of the team to find the riders does not exists.
   */
  @Override
  public int[] getTeamRiders(int teamId) throws IDNotRecognisedException {
    //Check if the teamId exists
    if (getTeamByID(teamId).getTeamName() == "Null"){
      throw new IDNotRecognisedException("ID "+ teamId + " doesnt exist.");
    } else {
      //Create a temp integer array of size of number of riders in the team
      Rider[] riders = getTeamByID(teamId).getRiders();
      int[] riderIds = new int[riders.length];
      //For each rider in team add there ID to the array
      for (int i=0; i<riderIds.length; i++) {
        riderIds[i] = riders[i].getRiderID();
      }
      //Return the integer array of rider IDs in the team
```

```java
      //Return the integer array of rider IDs in the team
      return riderIds;
    }
  }

  /**
   * This method is used to create a new rider.
   *
   * @param teamId The team which the rider belongs to.
   * @param name The name of the rider.
   * @param yearOfBirth The year of birth of the rider.
   * @return The unique ID of the created rider.
   * @exception IDNotRecognisedException The teamID does not match a team in the system.
   * @exception IllegalArgumentException The name of the rider is null or the year of birth is less than 1900.
   */
  @Override
  public int createRider(int teamID, String name, int yearOfBirth) throws IDNotRecognisedException, IllegalArgumentException {
    //Check if the teamID exists
    if (getTeamByID(teamID).getTeamName() == "Null"){
      throw new IDNotRecognisedException("ID "+ teamID + " doesnt exist.");
        }else {
      //Check if name and year of birth match the requirements
      if((name == null) || (yearOfBirth < 1900)){
        throw new IllegalArgumentException("Name or Year of Birth do not match the requirements.");
      } else {
        //Create a new rider object
        assert(yearOfBirth>1900 && yearOfBirth<2010):"Invalid rider year of birth";
        Rider rider = new Rider(teamID, name , yearOfBirth);
        riderArray.add(rider);
        getTeamByID(teamID).addRider(rider);
        return rider.getRiderID();
      }
    }
  }

  /**
   * This method is used to remove a rider.
   *
   * @param teamId The rider ID to be removed.
   * @exception IDNotRecognisedException The ID of the rider to be removed does not exist.
   */
  @Override
  public void removeRider(int riderId) throws IDNotRecognisedException {
    for (Team team : teamArray) {
      for (Rider rider : team.getRiders()) {
        if (rider.getRiderID() == riderId) {
          team.removeRider(getRiderByID(riderId));
          Results[] results = rider.getRiderResults();
          for(Results result: results){
```

```java
      Stage stage = result.getResultStage();
      deleteRiderResultsInStage(stage.getStageId(), riderId);
     }
     rider.setRiderName("Null");
    }
   }
  }
  assert (getRiderByID(riderId).getRiderName() != null) :
   new IDNotRecognisedException("ID " + riderId + " does not match any riders in system.");
 }

 /**
  * This method is used to register a result for a rider.
  *
  * @param stageId The stage ID the result refers to being added.
  * @param riderId The ID of the rider that the result is being added for.
  * @param checkpoints An array of times at which the rider reached each of the segments of the stage (including start and finish time).
  * @exception IDNotRecognisedException The ID of the rider or stage does not exist.
  * @exception DuplicatedResultException The rider has already had a result added for this specific stage.
  * @exception InvalidCheckpointsException The length of the checkpoints must be equal to n+2 of the number of segments in the stage
  *                      (+2 indicates the start and finish times being includes aswell)
  * @exception InvalidStageStateException the stage is waiting for a result and is not prepared to register another result yet.
  */
 @Override
 public void registerRiderResultsInStage(int stageId, int riderId, LocalTime... checkpoints)
   throws IDNotRecognisedException, DuplicatedResultException, InvalidCheckpointsException,
   InvalidStageStateException {
  //Check rider exists
  if(getRiderByID(riderId).getRiderName() == "Null"){
   throw new IDNotRecognisedException("ID " + riderId + "does not exist");
  }
  //Check stage exists
  if(getStageByID(stageId).getStageName() == "Null"){
   throw new IDNotRecognisedException("ID " + stageId + "does not exist");
  }
  //Get the objects of the rider and stage IDs relate to
  Rider rider = getRiderByID(riderId);
  Stage stage = getStageByID(stageId);
  //Check stage is prepared for registering a result
  if(!stage.isPrepared()){
   throw new InvalidStageStateException("Stage is waiting for a result");
  }
  //Check length of checkpoints is number of segments + 2
  if(checkpoints.length != stage.getStageSegments().length + 2){
    throw new InvalidCheckpointsException("Number of checkpoints must be number of segments + 2");
  }
  //Check the rider doesnt have a result for this stage already
  for(int i = 0; i < stage.getStageResults().size(); i++){
    if (stage.getStageResults().get(i).getResultRider() == rider){
      throw new DuplicatedResultException("Stage already has a result for this rider");
    }
  }
  //Register the results
  Results result = new Results(stage , rider, checkpoints);
  stage.addStageResults(result);
  rider.addRiderResult(result);
}

/**
 * This method gets the all the results of a given rider in a given stage, segments included.
 * @param stageId The ID of the stage where the results will come from.
 * @param riderId The ID of the rider which will be used to get the results.
 * @exception IDNotRecognisedException The ID of the rider or stage does not exist.
 * @return Result times based on the rider's ID and Stage ID.
 */
@Override
public LocalTime[] getRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException {
  boolean exist = false;
  //Check rider exists
  if(getRiderByID(riderId).getRiderName() == null){
    throw new IDNotRecognisedException("ID " + riderId + "does not exist");
  }
  //Check stage exists
  if(getStageByID(stageId).getStageName() == null){
    throw new IDNotRecognisedException("ID " + stageId + "does not exist");
  }
  //Get the objects of the rider and stage object related to their IDs
  Rider rider = getRiderByID(riderId);
  Stage stage = getStageByID(stageId);

  //Check rider does not have a result in that stage
  for (Results result : rider.getRiderResults()) {
    if (result.getResultStage().equals(stage)) {
      exist = true;
      return result.getResultTimes();
    }

  }
  //Check rider doesnt have result
  if(!exist){
    throw new IDNotRecognisedException("Rider " + riderId + " doesnt have
a result");
  }
  return null;

}
/**
 * This method finds the elapsed time the rider was in the stage for.
 * @param stageId The ID of the stage where the results will come from.
 * @param riderId The ID of the rider which will be used to get the
results.
 * @exception IDNotRecognisedException The ID of the rider or stage does
not exist.
 * @return The finish time - start time to give you the duration between
the start and finish.
 */
@Override
public LocalTime getRiderAdjustedElapsedTimeInStage(int stageId, int
riderId) throws IDNotRecognisedException {
  //Check rider exists
  if(getRiderByID(riderId).getRiderName() == null){
    throw new IDNotRecognisedException("ID " + riderId + "does not
exist");
  }
  //Check stage exists
  if(getStageByID(stageId).getStageName() == null){
    throw new IDNotRecognisedException("ID " + stageId + "does not
exist");
  }
  //Get rider results as an array of times
  LocalTime[] temp = getRiderResultsInStage(stageId, riderId);
  //Find last time
  int lastTime = temp.length - 1;
  //If rider has at least 2 times registered
  if(lastTime < 1) {
    return null;
  } else {
    //assert (temp[lastTime] == temp[0]) : return null;
    //Find the duration between the start time and end time for hours,mins
and secs
    int timeHor = (int) Duration.between(temp[0],
temp[lastTime]).toHoursPart();
    int timeMin = (int) Duration.between(temp[0],
temp[lastTime]).toMinutesPart();
    int timeSec = (int) Duration.between(temp[0],
temp[lastTime]).toSecondsPart();
    //Set a LocalTime for the durations between the start and finish
    LocalTime timeOvr = LocalTime.of(timeHor, timeMin, timeSec);
    return timeOvr;
  }
}
/**
 * This method deletes the results of a given rider in a given stage,
based on the rider and stage ID.
 * @param stageId The ID of the stage where the results will come from.
 * @param riderId The ID of the rider which will be used to get the
```

```
results.
     * @exception IDNotRecognisedException The ID of the rider or stage does
not exist.
     */
    @Override
    public void deleteRiderResultsInStage(int stageId, int riderId) throws
IDNotRecognisedException {
      boolean found = false;
      //Check rider exists
      if(getRiderByID(riderId).getRiderName() == null){
        throw new IDNotRecognisedException("ID " + riderId + "does not
exist");
      }
      //Check stage exists
      if(getStageByID(stageId).getStageName() == null){
        throw new IDNotRecognisedException("ID " + stageId + "does not
exist");
      }
      //Set rider and stage objects from there IDs
      Stage stage = getStageByID(stageId);
      Rider rider = getRiderByID(riderId);
      //Get riders results as an array
      Results[] results = rider.getRiderResults();
      //Check that for the stage the rider has a result
      for (Results results2 : results) {
        if(results2.getResultStage().equals(stage)){
          if(results2.getResultRider().equals(rider)){
            //Remove result from both stage and rider
            stage.removeResults(results2);
            rider.removeResults(results2);
            found = true;
          }
        }
      }
      //If not found
      if(!found){
        throw new IDNotRecognisedException("Stage doesnt have a result for
this rider");
      }
    }

    /**
     * This gets a list of rider IDs sorted by there finishing time.
     * @param stageId The ID of the stage where the results will come from.
     * @param riderId The ID of the rider which will be used to get the
results.
     * @exception IDNotRecognisedException The ID of the rider or stage does
not exist.
     */
    @Override
    public int[] getRidersRankInStage(int stageId) throws
IDNotRecognisedException {
      //Get sorted times of riders
      LocalTime[] timesSorted = getRankedAdjustedElapsedTimesInStage(stageId);
      //Set stage and result objets
      Stage stage = getStageByID(stageId);
      //If no stages return empty list
      int[] leader = new int[0];
      if(stage.getStageLength() == 0){
        return leader;
      }
      ArrayList<Results> results = stage.getStageResults();
      //Get a temporal leaderboard
      int[] temporal= new int[riderArray.size()];
      //Check that for the stage the rider has a result
      int i = 1;
      for (Results results2 : results) {
        if(results2.getResultStage().equals(stage)){
          temporal[i] = results2.getResultRider().getRiderID();
          i++;
        }
      }
      //Get the length the leaderboard needs to be
      int count = 0;
      for(int x = 0; x<riderArray.size();x++){
        if(temporal[x] != 0){
          count++;
        }
      }
      //Set leaderboard of times to right size so theres no overhang
      int[] leaderboard = new int[count];
      //For each index increment so its added to right leaderboard spot
      int flagging = 0;
      for(int y = 0; y<riderArray.size(); y++){
        if(temporal[y] != 0){
          leaderboard[flagging] = temporal[y];
          flagging++;
        }
      }
      //Set leaderboard of IDs to the right size
      int[] sortedLeaderboard = new int[count];
      int a = 0;
      for(int b = 0; b<count; b++){
        //Set the rider ID for its coresponding position in the sorted time
array
        //Similar to linear search
        if(getRiderAdjustedElapsedTimeInStage(stageId,
leaderboard[b]).equals(timesSorted[a])){
          sortedLeaderboard[a] = leaderboard[b];
          a++;
          if(a==count){
            break;
          }
          //Resets search
          b=-1;
        }
      }
      return sortedLeaderboard;
    }
    /**
```

```
     * This method gets the adjusted alapsed times of the riders in the stage, and ranks them based on fastest elapsed time to slowest.
     * @param stageId The ID of the stage where the results will come from.
     * @exception IDNotRecognisedException The ID of the rider or stage does not exist.
     * @return Result times based on Stage ID and ranked with fastest elapsed time to slowest.
     */
    @Override
    public LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId) throws IDNotRecognisedException {
      //Check stage exists
      if(getStageByID(stageId).getStageName() == null){
        throw new IDNotRecognisedException("Stage " + stageId + "doesnt exist");
      }
      //Set stage and result objects
      Stage stage = getStageByID(stageId);
      ArrayList<Results> results = stage.getStageResults();
      //Get a temporal leaderboard
      int[] temporal= new int[riderArray.size()];
      //Check that for the stage the rider has a result
      int i = 1;
      for (Results results2 : results) {
        if(results2.getResultStage().equals(stage)){
          temporal[i] = results2.getResultRider().getRiderID();
          i++;
        }
      }
      //Get the length the leaderboard needs to be
      int count = 0;
      for(int x = 0; x<riderArray.size();x++){
        if(temporal[x] != 0){
          count++;
        }
      }
      //Set leaderboard to right size so theres no overhang
      int[] leaderboard = new int[count];
      //For each index increment so its added to right leaderboard spot
      int flagging = 0;
      for(int y = 0; y<riderArray.size(); y++){
        if(temporal[y] != 0){
          leaderboard[flagging] = temporal[y];
          flagging++;
        }
      }
      //Get elapsed time
      LocalTime[] times = new LocalTime[count];;
      for(int j = 0; j<count; j++){
        if(leaderboard[j] == 0){
          break;
        } else {
          times[j] = getRiderAdjustedElapsedTimeInStage(stageId, leaderboard[j]);
        }
      }
      //Sort times
      Arrays.sort(times);
      return times;
    }

    /**
     * This gets a list of rider points in the stage by there finishing time.
     * @param stageId The ID of the stage where the results will come from.
     * @return an array of rider points in a specific stage.
     * @exception IDNotRecognisedException The ID of the stage does not exist.
     */
    @Override
    public int[] getRidersPointsInStage(int stageId) throws
  IDNotRecognisedException {
      //Check stage exists
      if(getStageByID(stageId).getStageName() == null){
        throw new IDNotRecognisedException("Stage ID "+stageId+" doesnt
  exist.");
      }
      //Get objects for stage, array of sorted riders & stage type
      Stage stage = getStageByID(stageId);
      int[] sortedRiders = getRidersRankInStage(stageId);
      //Checks if no results exists return empty list
      int[] leader = new int[0];
      if(sortedRiders.length == 0){
        return leader;
      }
      StageType stageType = stage.getStageType();
      //Create arrays for points
      int[] stagePoints = new int[sortedRiders.length];
      //For each rider in relative position
      for(int i = 0; i<sortedRiders.length; i++){
        //Get rider to then set there points in object class
        Rider rider = getRiderByID(sortedRiders[i]);
        //If position is more than 15 no points
        if(i>15){
          stagePoints[i] = 0;
        } else {
          //Switch for each type of stage
          switch (stageType) {
            case FLAT:
              //Switch for each position to reward points
              switch (i+1) {
                case 1:
                  stagePoints[i] = 50;
                  break;
                case 2:
                  stagePoints[i] = 30;
                  break;
                case 3:
                  stagePoints[i] = 20;
                  break;
                case 4:
                  stagePoints[i] = 18;
                  stagePoints[i] = 18;
                  break;
                case 5:
                  stagePoints[i] = 16;
                  break;
                case 6:
                  stagePoints[i] = 14;
                  break;
                case 7:
                  stagePoints[i] = 12;
                  break;
                case 8:
                  stagePoints[i] = 10;
                  break;
                case 9:
                  stagePoints[i] = 8;
                  break;
                case 10:
                  stagePoints[i] = 7;
                  break;
                case 11:
                  stagePoints[i] = 6;
                  break;
                case 12:
                  stagePoints[i] = 5;
                  break;
                case 13:
                  stagePoints[i] = 4;
                  break;
                case 14:
                  stagePoints[i] = 3;
                  break;
                case 15:
                  stagePoints[i] = 2;
                  break;
                default:
                  //Incase didnt catch position
                  stagePoints[i] = 0;
                  break;
              }
              break;
            case MEDIUM_MOUNTAIN:
              //Switch for each position to reward points
              switch (i+1) {
                case 1:
                  stagePoints[i] = 30;
                  break;
                case 2:
                  stagePoints[i] = 25;
                  break;
                case 3:
                  stagePoints[i] = 22;
                  break;
                case 4:
                  stagePoints[i] = 19;
                  break;
```

```
                break;
              case 5:
                stagePoints[i] = 17;
                break;
              case 6:
                stagePoints[i] = 15;
                break;
              case 7:
                stagePoints[i] = 13;
                break;
              case 8:
                stagePoints[i] = 11;
                break;
              case 9:
                stagePoints[i] = 9;
                break;
              case 10:
                stagePoints[i] = 7;
                break;
              case 11:
                stagePoints[i] = 6;
                break;
              case 12:
                stagePoints[i] = 5;
                break;
              case 13:
                stagePoints[i] = 4;
                break;
              case 14:
                stagePoints[i] = 3;
                break;
              case 15:
                stagePoints[i] = 2;
                break;
              default:
                //Incase didnt catch position
                stagePoints[i] = 0;
                break;
              }
            break;
          case HIGH_MOUNTAIN:
            //Switch for each position to reward points
            switch (i+1) {
              case 1:
                stagePoints[i] = 20;
                break;
              case 2:
                stagePoints[i] = 17;
                break;
              case 3:
                stagePoints[i] = 15;
                break;
              case 4:
                stagePoints[i] = 13;
                break;
```

```
       case 5:
        stagePoints[i] = 11;
        break;
       case 6:
        stagePoints[i] = 10;
        break;
       case 7:
        stagePoints[i] = 9;
        break;
       case 8:
        stagePoints[i] = 8;
        break;
       case 9:
        stagePoints[i] = 7;
        break;
       case 10:
        stagePoints[i] = 6;
        break;
       case 11:
        stagePoints[i] = 5;
        break;
       case 12:
        stagePoints[i] = 4;
        break;
       case 13:
        stagePoints[i] = 3;
        break;
       case 14:
        stagePoints[i] = 2;
        break;
       case 15:
        stagePoints[i] = 1;
        break;
       default:
        //Incase didnt catch position
        stagePoints[i] = 0;
        break;
      }
     default:
      //Incase didnt catch position
      stagePoints[i] = 0;
      break;

    }
   //Set rider points in object
   rider.addRiderPoints(stagePoints[i]);
   }
 }
  return stagePoints;
 }

 /**
  * This gets a list of rider points in the mountain stage by getting the time of each segment they crossed.
  * @param stageId The ID of the stage where the results will come from.
```

```
   * @return an array of rider points in a specifi stage where the segments are mountains.
   * @exception IDNotRecognisedException The ID of the stage does not exist.
   */
  @Override
  public int[] getRidersMountainPointsInStage(int stageId) throws IDNotRecognisedException {
    //Check stage exists
    if(getStageByID(stageId).getStageName() == null){
      throw new IDNotRecognisedException("Stage ID "+stageId+" doesnt exist.");
    }
    //Get objects for stage, array of sorted riders & stage segments
    Stage stage = getStageByID(stageId);
    int[] sortedRiders = getRidersRankInStage(stageId);
    //Checks if no results exists return empty list
    int[] leader = new int[0];
    if(sortedRiders.length == 0){
      return leader;
    }
    Segment[] segmentType = stage.getStageSegments();
    //Create arrays for riders and points
    Rider[] riders = new Rider[sortedRiders.length];
    int[] pointStage = new int[sortedRiders.length];
    //Set a HashMap that maps rider objects to times for each segment
    HashMap<Rider, LocalTime> resultMap = new HashMap<Rider, LocalTime>();
    //Sets the rider IDs into rider Objects in a array
    for (int z=0;z<sortedRiders.length;z++) {
      riders[z] = getRiderByID(sortedRiders[z]);
    }
    //For each segment type
    for(int j = 0; j<segmentType.length; j++){
      SegmentType type = segmentType[j].getSegmentType();
      //Check that segment type is not sprint
      if(type != SegmentType.SPRINT){
        //Loop through results
        for(int i = 0; i<=segmentType.length; i++){
          //Check for each rider if they exists
          for (Rider rider : riders) {
            if(rider.getRiderName() == null){
              continue;
            }
            //Get the riders segment times for each segment starting from first segment to last one
            LocalTime[] results = getRiderResultsInStage(stageId, rider.getRiderID());
            if (results != null) {
              assert (results.length == segmentType.length):"Cant set a result for a segment that doesnt exist";
              int timeHor = (int) Duration.between(results[0], results[i+1]).toHoursPart();
              int timeMin = (int) Duration.between(results[0], results[i+1]).toMinutesPart();
              int timeSec = (int) Duration.between(results[0], results[i+1]).toSecondsPart();
```

```java
       LocalTime segmentOvr = LocalTime.of(timeHor, timeMin, timeSec);
       resultMap.put(rider, segmentOvr);
      }
     }
     //Sort HashMap by segment crossing
     Map<Rider, LocalTime> sortedMap =
      resultMap.entrySet().stream()
      .sorted(Entry.comparingByValue())
      .collect(Collectors.toMap(Entry::getKey, Entry::getValue,(e1, e2) -> e1, LinkedHashMap::new));
     //Set position position of rider
     int pos = 1;
     //For each rider in HashMap
     for (Rider rider : sortedMap.keySet()) {
      //If there position is more than 8 they dont get points
      if (pos > 8) {
       break;
      }
      //Check each rider ID relates to the rider in the HashMap
      for (int b=0;b<riders.length;b++) {
       if (riders[b].equals(rider)){
        //Switch for each type of segment
        switch (type) {
         case C4:
          //Switch for each position to reward points
          switch(pos){
           case 1:
            pointStage[b] = 1;
            pos++;
            break;
           default:
            pointStage[b] = 0;
            pos++;
            break;
          }
          break;
         case C3:
          //Switch for each position to reward points
          switch(pos){
           case 1:
            pointStage[b] = 2;
            pos++;
            break;
           case 2:
            pointStage[b] = 1;
            pos++;
            break;
           default:
            pointStage[b] = 0;
            pos++;
            break;
          }
          break;
         case C2:
```

```
                    //Switch for each position to reward points
                    switch(pos){
                      case 1:
                        pointStage[b] = 5;
                        pos++;
                        break;
                      case 2:
                        pointStage[b] = 3;
                        pos++;
                        break;
                      case 3:
                        pointStage[b] = 2;
                        pos++;
                        break;
                      default:
                        pointStage[b] = 0;
                        pos++;
                        break;
                    }
                    break;
                  case C1:
                    //Switch for each position to reward points
                    switch(pos){
                      case 1:
                        pointStage[b] = 10;
                        pos++;
                        break;
                      case 2:
                        pointStage[b] = 8;
                        pos++;
                        break;
                      case 3:
                        pointStage[b] = 6;
                        pos++;
                        break;
                      case 4:
                        pointStage[b] = 4;
                        pos++;
                        break;
                      case 5:
                        pointStage[b] = 2;
                        pos++;
                        break;
                      case 6:
                        pointStage[b] = 1;
                        pos++;
                        break;
                      default:
                        pointStage[b] = 0;
                        pos++;
                        break;
                    }
                    break;
                  case HC:
                    //Switch for each position to reward points
```

```
                        switch(pos){
                          case 1:
                            pointStage[b] = 20;
                            pos++;
                            break;
                          case 2:
                            pointStage[b] = 15;
                            pos++;
                            break;
                          case 3:
                            pointStage[b] = 12;
                            pos++;
                            break;
                          case 4:
                            pointStage[b] = 10;
                            pos++;
                            break;
                          case 5:
                            pointStage[b] = 8;
                            pos++;
                            break;
                          case 6:
                            pointStage[b] = 6;
                            pos++;
                            break;
                          case 7:
                            pointStage[b] = 4;
                            pos++;
                            break;
                          case 8:
                            pointStage[b] = 2;
                            pos++;
                            break;
                          default:
                            pointStage[b] = 0;
                            pos++;
                            break;
                        }
                        break;
                      default:
                        pointStage[b] = 0;
                        pos++;
                        break;
                    }
                  }
                  rider.addRiderPoints(pointStage[b]);
                }
              }
            }
          }
        }

        return pointStage;
        }
```

```
  /**
   * Erases all the records from the system as if it was new.
   */
  @Override
  public void eraseCyclingPortal() {
        // Clear Arrays
      teamArray.clear();
      raceArray.clear();
      riderArray.clear();

      // Reset counts of everything
      Race.resetNumberOfRaces();
      Race.resetNumberOfStages();
      Rider.resetNumberOfRiders();
      Segment.resetNumberOfSegments();
      Stage.resetNumberOfSegments();
      Stage.resetNumberOfStages();
      Team.resetNumberOfRiders();
      Team.resetNumberOfTeams();
  }

  /**
   * Saves the system data onto a file of a given name.
   * @param filename the name of the file to be saved.
   * @exception IOException failed or interrupted I/O operation.
   */
  @Override
  public void saveCyclingPortal(String filename) throws IOException {
    ObjectOutputStream saveFile = new ObjectOutputStream(new
FileOutputStream(filename));
    try{
          saveFile.writeObject(this);
    } finally {
          saveFile.close();
    }
  }

  /**
   * Loads the data from a given file name onto the system from where it was
saved.
   * @param filename the name of the file to be loaded.
   * @exception IOException failed or interrupted I/O operation.
   * @exception ClassNotFoundException class with name could not be found.
   */
  @Override
  public void loadCyclingPortal(String filename) throws IOException,
ClassNotFoundException {
    ObjectInputStream loadFile = new ObjectInputStream(new
FileInputStream(filename));
    try {
      CyclingPortal obj = (CyclingPortal) loadFile.readObject();
      this.raceArray = obj.raceArray;
      this.teamArray = obj.teamArray;
      this.riderArray = obj.riderArray;
      }
```

```java
      finally {
        loadFile.close();
      }

    }

    /**
     * This method is used to remove a race by its name.
     *
     * @param name The name of the race to be deleted.
     * @exception NameNotRecognisedException when using a name that does not
exist.
     */
    @Override
    public void removeRaceByName(String name) throws
NameNotRecognisedException {
      //For each race in the race Array
      //Linear Search Big O: O(N) Space complexity: O(1)
      for (Race race : raceArray) {
        //If the race object name is equalivant
        if (race.getRaceName() == name) {
          //Remove that race object from the array
          raceArray.remove(race);
          return;
        }
      }
      //If no race is found throw error
      throw new NameNotRecognisedException("No race with name " + name + "
exists.");
    }

    /**
     * This method returns a list of race elapsed times based on sorted
elapsed overall times.
     * @param raceId the ID of the race.
     * @return an array of finish times.
     * @exception IDNotRecognisedException ID does not exist in the system.
     */
    @Override
    public LocalTime[] getGeneralClassificationTimesInRace(int raceId) throws
IDNotRecognisedException {
      //Check race exists
      if(getRaceByID(raceId).getRaceName() == null){
        throw new IDNotRecognisedException("Race with ID " + raceId + " doesnt
exist in the system");
      }
      //Get the race by its ID
      Race race = getRaceByID(raceId);
      //Get stages in a array by the race ID
      Stage[] stages = race.getStagesV2();
      //If no stages in race return empty array
      LocalTime[] leader = new LocalTime[0];
      if(stages.length == 0){
        return leader;
      }
      //Creates a new array list of riders
      ArrayList<Rider> riders = new ArrayList<>();
      //Setting all riders in race in the array using the stages
      for(Stage stageFindRider : stages){
        int[] tempHoldID = getRidersRankInStage(stageFindRider.getStageId());
        for(int tempIDs : tempHoldID){
          if(riders.contains(getRiderByID(tempIDs))){
            break;
          } else {
            riders.add(getRiderByID(tempIDs));
          }
        }
      }
      //Create a HashMap to calculate elapsed race time through elapsed stage time
      HashMap<Rider, LocalTime> riderRaceElapsed = new HashMap<Rider, LocalTime>();
      for (Rider rider : riders) {
        riderRaceElapsed.put(rider, LocalTime.of(0, 0, 0));
        for (Stage stage : stages) {
          LocalTime tempTimes = getRiderAdjustedElapsedTimeInStage(stage.getStageId(),rider.getRiderID());
          if (tempTimes != null) {
            riderRaceElapsed.replace(rider, riderRaceElapsed.get(rider).plusHours(tempTimes.getHour())
  .plusMinutes(tempTimes.getMinute()).plusSeconds(tempTimes.getSecond()));
          }
        }
      }
      //Sort by elapsed race time
      Map<Rider, LocalTime> sortedRiders =
              riderRaceElapsed.entrySet().stream()
              .sorted(Entry.comparingByValue())
              .collect(Collectors.toMap(Entry::getKey, Entry::getValue,(e1, e2) -> e1, LinkedHashMap::new));
      //Transform HashMap to an array
      LocalTime[] timesSorted = new LocalTime[riders.size()];
      sortedRiders.values().toArray(timesSorted);
      //Return array of race finish times
      return timesSorted;

    }

    /**
     * This method returns a list of riders' points, sorted by the total elapsed time.
     * @param raceId ID of the race.
     * @return an array of riders' points in the race.
     * @exception IDNotRecognisedException ID does not exist in the system.
     */
    @Override
    public int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException {
      //Check race exists
      //check race exists
      if(getRaceByID(raceId).getRaceName() == null){
        throw new IDNotRecognisedException("Race with ID " + raceId + " doesn't exist in the system");
      }
      //Get the race by its ID
      Race race = getRaceByID(raceId);
      //Get stages in a array by the race ID
      Stage[] stages = race.getStagesV2();
      //If no stages in race return empty array
      int[] leader = new int[0];
      if(stages.length == 0){
        return leader;
      }
      //Creates a new array list of riders
      ArrayList<Rider> riders = new ArrayList<>();
      //Setting all riders in race in the array using the stages
      for(Stage stageFindRider : stages){
        int[] tempHoldID = getRidersRankInStage(stageFindRider.getStageId());
        for(int tempIDs : tempHoldID){
          if(riders.contains(getRiderByID(tempIDs))){
            break;
          } else {
            riders.add(getRiderByID(tempIDs));
          }
        }
      }

      Map<Rider, Integer> riderRacePoints = new HashMap<Rider, Integer>();
      for (Rider rider : riders) {
        // Riders are added to HashMap
        riderRacePoints.put(rider, 0);
        for (Stage stage : stages) {
          // retrieves an array of ranked rider IDs in the stages
          int[] ranks = getRidersRankInStage(stage.getStageId());
          // Finds the index of the current rider in the array
          int indexOfRider = -1;
          for (int i=0; i<ranks.length; i++) {
            if (ranks[i] == rider.getRiderID()) {
              indexOfRider = i;
            }
          }
          if (indexOfRider != -1) {
            int[] pointsArr = getRidersPointsInStage(stage.getStageId());
            int points = pointsArr[indexOfRider];

            // Adds stage points to existing race points
            riderRacePoints.replace(rider, riderRacePoints.get(rider) + points);
          }
        }
      }

      // Creates array to store points
      int[] sortedPoints = new int[riders.size()];
      // Creates array of rider IDs sorted by elapsed time
```

```java
    // Creates array of rider IDs sorted by elapsed time
    int[] riderRanks = getRidersGeneralClassificationRank(raceId);
    for ( int i=0; i<riderRanks.length; i++ ) {
      sortedPoints[i] = riderRacePoints.get(getRiderByID(riderRanks[i]));
    }
    return sortedPoints;
  }

  /**
   * This method returns a list of riders' points for the mountain segments, sorted by the total elapsed time.
   * @param raceId ID of the race.
   * @return an array of riders' points in the mountain segments race.
   * @exception IDNotRecognisedException ID does not exist in the system.
   */
  @Override
  public int[] getRidersMountainPointsInRace(int raceId) throws IDNotRecognisedException {
    //Check race exists
    if(getRaceByID(raceId).getRaceName() == null){
      throw new IDNotRecognisedException("Race with ID " + raceId + " doesnt exist in the system");
    }
    //Get the race by its ID
    Race race = getRaceByID(raceId);
    //Get stages in a array by the race ID
    Stage[] stages = race.getStagesV2();
    //If no stages in race return empty array
    int[] leader = new int[0];
    if(stages.length == 0){
      return leader;
    }
    //Creates a new array list of riders
    ArrayList<Rider> riders = new ArrayList<>();
    //Setting all riders in race in the array using the stages
    for(Stage stageFindRider : stages){
      int[] tempHoldID = getRidersRankInStage(stageFindRider.getStageId());
      for(int tempIDs : tempHoldID){
        if(riders.contains(getRiderByID(tempIDs))){
          break;
        } else {
          riders.add(getRiderByID(tempIDs));
        }
      }
    }

    Map<Rider, Integer> riderRacePoints = new HashMap<Rider, Integer>();
    for (Rider rider : riders) {
      // Riders are added to HashMap
      riderRacePoints.put(rider, 0);
      for (Stage stage : stages) {
        // retrieves an array of ranked rider IDs in the stages
        int[] ranks = getRidersRankInStage(stage.getStageId());
        // Finds the index of the current rider in the array
        int indexOfRider = -1;
    for (int i=0; i<ranks.length; i++) {
     if (ranks[i] == rider.getRiderID()) {
      indexOfRider = i;
     }
    }
    if (indexOfRider != -1) {
     int[] pointsArr =
getRidersMountainPointsInStage(stage.getStageId());
     int points = pointsArr[indexOfRider];

     // Adds stage points to existing race points
     riderRacePoints.replace(rider, riderRacePoints.get(rider) +
points);
    }
   }
  }

  // Creates array to store points
  int[] sortedPoints = new int[riders.size()];
  // Creates array of rider IDs sorted by elapsed time
  int[] riderRanks = getRidersGeneralClassificationRank(raceId);
  for ( int i=0; i<riderRanks.length; i++ ) {
   sortedPoints[i] = riderRacePoints.get(getRiderByID(riderRanks[i]));
  }
  return sortedPoints;
 }

 /**
  * This method returns a list of rider IDs from the race, sorted by the
elapsed time.
  * @param raceId ID of the race.
  * @return an array of riders' points in the segments race.
  * @exception IDNotRecognisedException ID does not exist in the system.
  */
 @Override
 public int[] getRidersGeneralClassificationRank(int raceId) throws
IDNotRecognisedException {
  //Check race exists
  if(getRaceByID(raceId).getRaceName() == null){
   throw new IDNotRecognisedException("Race with ID " + raceId + " doesnt
exist in the system");
  }
  //Get the race by its ID
  Race race = getRaceByID(raceId);
  //Get stages in a array by the race ID
  Stage[] stages = race.getStagesV2();
  //If no stages in race return empty array
  int[] leader = new int[0];
  if(stages.length == 0){
   return leader;
  }
  //Creates a new array list of riders
  ArrayList<Rider> riders = new ArrayList<>();
  //Setting all riders in race in the array using the stages
  for(Stage stageFindRider : stages){
   int[] tempHoldID = getRidersRankInStage(stageFindRider.getStageId());
   for(int tempIDs : tempHoldID){
    if(riders.contains(getRiderByID(tempIDs))){
     break;
    } else {
     riders.add(getRiderByID(tempIDs));
    }
   }
  }
  //Create a HashMap to calculate elapsed race time through elapsed stage
time
  HashMap<Integer, LocalTime> riderRaceElapsed = new HashMap<Integer,
LocalTime>();
  for (Rider rider : riders) {
   riderRaceElapsed.put(rider.getRiderID(), LocalTime.of(0, 0, 0));
   for (Stage stage : stages) {
    LocalTime tempTimes =
getRiderAdjustedElapsedTimeInStage(stage.getStageId(),rider.getRiderID());
    if (tempTimes != null) {
     riderRaceElapsed.replace(rider.getRiderID(),
riderRaceElapsed.get(rider.getRiderID()).plusHours(tempTimes.getHour())

.plusMinutes(tempTimes.getMinute()).plusSeconds(tempTimes.getSecond()));
    }
   }
  }
  //Sort by elapsed race time
  Map<Integer, LocalTime> sortedRiders =
      riderRaceElapsed.entrySet().stream()
      .sorted(Entry.comparingByValue())
      .collect(Collectors.toMap(Entry::getKey, Entry::getValue,(e1,
e2) -> e1, LinkedHashMap::new));
  //Transform HashMap to an array
  int[] ridersSorted = new int[riders.size()];
  int index = 0;
  for(Map.Entry<Integer, LocalTime> mapEntry : sortedRiders.entrySet()){
   ridersSorted[index] = mapEntry.getKey();
   index++;
  }
  //Return array of race IDs
  return ridersSorted;

 }

 /**
  * This method returns a list of rider IDs for the race, sorted by there
points.
  * @param raceId ID of the race.
  * @return an array of rider IDs in the segments race ordered by points.
  * @exception IDNotRecognisedException ID does not exist in the system.
  */
 @Override
 public int[] getRidersPointClassificationRank(int raceId) throws
IDNotRecognisedException {
  //Check race exists
```

```java
    if(getRaceByID(raceId).getRaceName() == null){
      throw new IDNotRecognisedException("Race with ID " + raceId + " doesnt exist in the system");
    }
    //Get the race by its ID
    Race race = getRaceByID(raceId);
    //Get stages in a array by the race ID
    Stage[] stages = race.getStagesV2();
    //If no stages in race return empty array
    int[] leader = new int[0];
    if(stages.length == 0){
      return leader;
    }
    //Creates a new array list of riders
    ArrayList<Rider> riders = new ArrayList<>();
    //Setting all riders in race in the array using the stages
    for(Stage stageFindRider : stages){
      int[] tempHoldID = getRidersRankInStage(stageFindRider.getStageId());
      for(int tempIDs : tempHoldID){
        if(riders.contains(getRiderByID(tempIDs))){
          break;
        } else {
          riders.add(getRiderByID(tempIDs));
        }
      }
    }

    Map<Rider, Integer> riderRacePoints = new HashMap<Rider, Integer>();
    for (Rider rider : riders) {
      // Riders are added to HashMap
      riderRacePoints.put(rider, 0);
      for (Stage stage : stages) {
        // retrieves an array of ranked rider IDs in the stages
        int[] ranks = getRidersRankInStage(stage.getStageId());
        // Finds the index of the current rider in the array
        int indexOfRider = -1;
        for (int i=0; i<ranks.length; i++) {
          if (ranks[i] == rider.getRiderID()) {
            indexOfRider = i;
          }
        }
        if (indexOfRider != -1) {
          int[] pointsArr = getRidersPointsInStage(stage.getStageId());
          int points = pointsArr[indexOfRider];

          // Adds stage points to existing race points
          riderRacePoints.replace(rider, riderRacePoints.get(rider) + points);
        }
      }
    }
    Map<Rider, Integer> sortedRiders =
            riderRacePoints.entrySet().stream()
            .sorted(Entry.comparingByValue())
            .collect(Collectors.toMap(Entry::getKey, Entry::getValue,(e1,
e2) -> e1, LinkedHashMap::new));
    //Transform HashMap to an array
    int[] ridersSorted = new int[riders.size()];
    int index = 0;
    for(Map.Entry<Rider, Integer> mapEntry : sortedRiders.entrySet()){
      ridersSorted[index] = mapEntry.getKey().getRiderID();
      index++;
    }
    //Return array of race IDs
    return ridersSorted;

  }

  /**
   * This method returns a list of rider IDs for the race, sorted by there
points in the mountain segments.
   * @param raceId ID of the race.
   * @return an array of rider IDs in the segments race ordered by points in
mountain segment.
   * @exception IDNotRecognisedException ID does not exist in the system.
   */
  @Override
  public int[] getRidersMountainPointClassificationRank(int raceId) throws
IDNotRecognisedException {
    //Check race exists
    if(getRaceByID(raceId).getRaceName() == null){
      throw new IDNotRecognisedException("Race with ID " + raceId + " doesnt
exist in the system");
    }
    //Get the race by its ID
    Race race = getRaceByID(raceId);
    //Get stages in a array by the race ID
    Stage[] stages = race.getStagesV2();
    //If no stages in race return empty array
    int[] leader = new int[0];
    if(stages.length == 0){
      return leader;
    }
    //Creates a new array list of riders
    ArrayList<Rider> riders = new ArrayList<>();
    //Setting all riders in race in the array using the stages
    for(Stage stageFindRider : stages){
      int[] tempHoldID = getRidersRankInStage(stageFindRider.getStageId());
      for(int tempIDs : tempHoldID){
        if(riders.contains(getRiderByID(tempIDs))){
          break;
        } else {
          riders.add(getRiderByID(tempIDs));
        }
      }
    }

    Map<Rider, Integer> riderRacePoints = new HashMap<Rider, Integer>();
    for (Rider rider : riders) {
      // Riders are added to HashMap
      riderRacePoints.put(rider, 0);
      for (Stage stage : stages) {
        // retrieves an array of ranked rider IDs in the stages
        int[] ranks = getRidersRankInStage(stage.getStageId());
        // Finds the index of the current rider in the array
        int indexOfRider = -1;
        for (int i=0; i<ranks.length; i++) {
          if (ranks[i] == rider.getRiderID()) {
            indexOfRider = i;
          }
        }
        if (indexOfRider != -1) {
          int[] pointsArr =
getRidersMountainPointsInStage(stage.getStageId());
          int points = pointsArr[indexOfRider];

          // Adds stage points to existing race points
          riderRacePoints.replace(rider, riderRacePoints.get(rider) +
points);
        }
      }
    }

    Map<Rider, Integer> sortedRiders =
            riderRacePoints.entrySet().stream()
            .sorted(Entry.comparingByValue())
            .collect(Collectors.toMap(Entry::getKey, Entry::getValue,(e1,
e2) -> e1, LinkedHashMap::new));
    //Transform HashMap to an array
    int[] ridersSorted = new int[riders.size()];
    int index = 0;
    for(Map.Entry<Rider, Integer> mapEntry : sortedRiders.entrySet()){
      ridersSorted[index] = mapEntry.getKey().getRiderID();
      index++;
    }
    //Return array of race IDs
    return ridersSorted;


  }
}
```

```java
package cycling;
import java.io.Serializable;
import java.util.ArrayList;
/**
 * CyclingPortal is a minimally compiling, but non-functioning implementor
 * of the CyclingPortalInterface interface.
 *
 * @author Taariq Fadhill
 * @author Kaloyan Gaydarov
 * @version 4.20
 * @since 14/02/2022
 *
 */

public class Race implements Serializable{
    //-------------------------------Private Initial Varriables--------------
--------------//
    // Initializes number of races to 0.
    private static int numberOfRaces = 0;
    // Initializes raceID.
    private int raceID;
    // Initializes raceName.
    private String raceName;
    // Initializes raceDesc.
    private String raceDesc;
    // Initializes number of stages to 0.
    private static int numberOfStages = 0;
    // Initializes array of objects of stages.
    private ArrayList<Stage> stages;

    //-----------------------------------Setter methods----------------------
--------------//
    /**
   * Sets the race name according to the raceName and raceDesc objects.
     * @param raceName name of the race.
     *
     * Sets the race description according to the raceName and raceDesc
objects.
     * @param raceDesc description of the race.
     *
     * Resets number of races, setting to 0.
     * Resets number of stages in a race, setting to 0.
     * Adds a stage from the objects stage.
     * @param stage where object stages are stored.
     */
    public void setRaceName(String raceName){this.raceName= raceName;}
    public void setraceDesc(String raceDesc){this.raceDesc= raceDesc;}
    public static void resetNumberOfRaces() {numberOfRaces = 0;}
    public static  void resetNumberOfStages() {numberOfStages = 0;}
    public void addStage(Stage stage) {
        stages.add(stage);
        numberOfStages++;
    }

    //-----------------------------------Getter methods-----------------------------------//
    /**
   * Gets the race ID of the object.
     * @return the race ID.
     * Gets the race name of the object.
     * @return the race name.
     * Gets the race description.
     * @return the race description.
     * Gets the array list of object: stages.
     * @return array list of object: stages.
     * Gets the number of stages.
     * @return the number of stages in the object.
     *
     *
     * Gets the details of the race.
     * @param raceId the ID of the race.
     * @param raceName the name of the race.
     * @param raceDesc the description of the race.
     * @param numberOfStages the number of stages in the race.
     * @param totalLength the total length of the race in km.
   */
    public int getRaceID() {return raceID;}
    public String getRaceName(){return raceName;}
    public String getRaceDesc(){return raceDesc;}
    public ArrayList<Stage> getStages() {return stages;}
    public Stage[] getStagesV2() {
        Stage[] stageArray = new Stage[stages.size()];
        stageArray = stages.toArray(stageArray);
        return stageArray;
    }
    public int getNumberOfStages() {return numberOfStages;}
    double getTotalLength() {
        double totalLength = 0;
        double length;
        for (Stage stage : stages) {
            length = stage.getStageLength();
            assert (length >= 0);
            totalLength += length;
        }
        return totalLength;
    }
    public String getRaceDetails() {
        double totalLength = getTotalLength();
        return "ID: "+raceID+" | Name: "+raceName+" | Description: "+raceDesc+" | Number of Stages: "+numberOfStages+" | Total Length: "+totalLength;
    }

    //---------------------------------Constructer methods----------------------------------//
    /**
   * Creates Object 'Race' with initialized values of 'Null', 'Null' and '0' respectively, and an empty array.
   */
   */
    public Race(){
        this.raceName = "Null";
        this.raceDesc = "Null";
        this.raceID = 0;
        this.stages = new ArrayList<>();
    }

    /**
   * Creates Object 'Race' with initialized values.
     * @param raceName the name of a given race.
     * @param raceDesc the description of a given race.
   */
    public Race(String raceName, String raceDesc){
        this.raceName = raceName;
        this.raceDesc =raceDesc;
        this.raceID = ++numberOfRaces;
        this.stages = new ArrayList<>();
    }

    //---------------------------------Remover methods----------------------------------//
    /**
   * Creates method for removing a stage.
     * @param stage object 'stage' that will be removed.
     * @exception IDNotRecognisedException The ID of the stage does not exist.
   */
    public void removeStage(Stage stage) throws IDNotRecognisedException {
        if (!stages.contains(stage)) {
            throw new IDNotRecognisedException("stage does not exist in race with Id '"+raceID+"'");
        }
        stages.remove(stage);
        numberOfStages--;
    }

}
```

```java
package cycling;
import java.io.Serializable;
import java.time.LocalTime;
/**
 * CyclingPortal is a minimally compiling, but non-functioning implementor
 * of the CyclingPortalInterface interface.
 *
 * @author Kaloyan Gaydarov
 * @author Taariq Fadhill
 * @version 4.20
 * @since 14/02/2022
 *
 */

public class Results implements Serializable{
    //-------------------------------Private Initial Varriables---------------------------//
    //Gets the number of results for the stage, initializing at 0.
    private static int numberOfResults = 0;
    //initializes resultID.
    private int resultID;
    //initializes stage.
    private Stage stage;
    //initializes rider.
    private Rider rider;
    //initializes array of local times.
    private LocalTime[] times;

    //-----------------------------------Setter methods-----------------------------------//
    /**
     * Sets the results in a stage.
     * @param stage object stage.
     * Sets the results to a rider.
     * @param rider object rider.
     * Sets the result time based on local time array.
     * @param times collected from local time array.
     * Resets number of results in a stage, setting to 0.
    */


    public void setResultStage(Stage stage) {this.stage = stage;}
    public void setResultRider(Rider rider) {this.rider = rider;}
    public void setResultTime(LocalTime[] times) {this.times = times;}
    public static void resetNumberOfResults() {numberOfResults = 0;}


    //-----------------------------------Getter methods-----------------------------------//
    /**
     * Gets the results ID of the object.
     * @return the result ID.
     * Gets the stage object of where the given result is.
     * @return the stage object.
     * Gets the rider based on their result in the stage.
     * Gets the rider based on their result in the stage.
     * @return the rider object.
     * Gets the times of a given result.
     * @return the result times as an array.
     */


    public int getResultsId() {return resultID;}
    public Stage getResultStage() {return stage;}
    public Rider getResultRider() {return rider;}
    public LocalTime[] getResultTimes() {return times;}


    //---------------------------------Constructer methods-----------------------------------//
    /**
     * Creates object 'Results' with initialized values.
     * @param stage the stage the results will be stored in.
     * @param rider the rider associated with the result.
     * @param times the time of the given result.
     */
    public Results(Stage stage, Rider rider , LocalTime... times) {
        this.stage = stage;
        this.rider = rider;
        this.times = times;
        this.resultID = ++numberOfResults;
    }
    /**
     * Creates empty object 'Results' and sets the result ID to 0.
     */
    public Results(){
        this.stage = new Stage();
        this.resultID = 0;
    }

}
```

```java
package cycling;

import java.io.Serializable;
import java.util.ArrayList;
/**
 * CyclingPortal is a minimally compiling, but non-functioning implementor
 * of the CyclingPortalInterface interface.
 *
 * @author Kaloyan Gaydarov
 * @author Taariq Fadhill
 * @version 4.20
 * @since 14/02/2022
 *
 */

public class Rider implements Serializable{
    //-------------------------------Private Initial Varriables---------------------------//
    // Initializes the number of riders to 0.
    private static int numberOfRiders = 0;
    // Initializes teamID.
    private int teamID;
    // Initializes riderName.
    private String riderName;
    // Initializes yearOfBirth.
    private int yearOfBirth;
    // Initializes riderID.
    private int riderID;
    // Initializes points.
    private int points;
    // Initializes array of objects of results.
    private ArrayList<Results> resultsArray = new ArrayList<Results>();

    //-------------------------------------Setter methods-----------------------------------//
    /**
     * Sets the rider name.
     * @param riderName name of the rider.
     *
     * Sets the team ID.
     * @param teamID ID of the team the rider is in.
     *
     * Sets the year of birth.
     * @param yearOfBirth year that the rider was born, must be 2010 < 
yearOfBirth > 1900.
     *
     * Resets the number of riders in a team, setting to 0 .
     *
     * Sets points for the rider.
     * @param points points that will be stored in the rider object.
     *
     * Adds points to the rider object.
     * @param points points that will be stored in the rider object.
     *
     * Adds the result of the rider to the results array.
     * Adds the result of the rider to the results array.
     * @param results result of the rider that will be stored in the results array.
     */
    public void setRiderName(String riderName){this.riderName= riderName;}
    public void setTeamID(int teamID){this.teamID= teamID;}
    public void setYearOfBirth(int yearOfBirth){this.yearOfBirth= yearOfBirth; }
    public static void resetNumberOfRiders() {numberOfRiders = 0;}
    public void setRiderPoints(int points){this.points = points;}
    public void addRiderPoints(int points){this.points = this.points + points;}
    public void addRiderResult(Results results){resultsArray.add(results);}

    //-----------------------------------Getter methods-----------------------------------//
    /**
     * Gets the team ID of the object.
     * @return the team ID.
     * Gets the name of the rider.
     * @return name of the rider.
     * Gets the year of birth of the rider.
     * @return year that the rider was born, must be 2010 < yearOfBirth > 1900.
     * Gets the rider ID of the object.
     * @return the rider ID.
     * Gets the points for the given rider.
     * @return points assigned to the specific rider.
     * Gets the array list of object: results.
     * @return array list of object: results.
     *
     *
     * Gets the details of the rider.
     * @param riderID the ID of the rider.
     * @param riderName the name of the rider.
     * @param teamID the ID of the team the rider is assigned to.
     * @param yearOfBirth year that the rider was born, must be 2010 < yearOfBirth > 1900.
     * @param points points the rider as acquired.
     */
    public int getTeamID() {return teamID;}
    public String getRiderName() {return riderName;}
    public int getYearOfBirth() {return yearOfBirth;}
    public int getRiderID() {return riderID;}
    public int getRiderPoints() {return points;}
    public Results[] getRiderResults() {
        Results[] resultArr = new Results[resultsArray.size()];
        resultArr = resultsArray.toArray(resultArr);
        return resultArr;
    };
    public String getRiderDetails() {
        return "Rider ID: "+riderID+" | Name: "+riderName+" | Team ID: "+teamID+" | Year Of Birth: "+yearOfBirth+" | Points: "+points;
    }

    //-----------------------------------Constructer methods-------------------
    //                                Constructer methods
---------------//
    /**
     * Creates object 'Rider' with initialized values of 0, 'Null', 0, 0 and
0 respectively.
     */
    public Rider() {
        teamID = 0;
        riderName = "Null";
        yearOfBirth = 0;
        riderID = 0;
        points = 0;
    }

    /**
     * Creates object 'Rider' with initialized values.
     * @param teamID ID of the team the rider is assigned to.
     * @param riderName name of the rider.
     * @param yearOfBirth year that the rider was born, must be 2010 <
yearOfBirth > 1900.
     * @param riderID ID of the rider.
     * @param points points acquired by the rider.
     */
    public Rider(int teamID, String riderName , int yearOfBirth) {
        this.teamID = teamID;
        this.riderName = riderName;
        this.yearOfBirth = yearOfBirth;
        this.riderID = ++numberOfRiders;
        this.points = 0;

    }
    //--------------------------------Remover methods-----------------------
-----------//
    /**
     * Creates method for removing a rider.
     * @param result object result that will be removed.
     * @throws IDNotRecognisedException The ID of the rider does not exist.
     */
    public void removeResults(Results result) throws IDNotRecognisedException
{
        resultsArray.remove(result);
    }

}


```

```java
package cycling;

import java.io.Serializable;
/**
 * CyclingPortal is a minimally compiling, but non-functioning implementor
 * of the CyclingPortalInterface interface.
 *
 * @author Kaloyan Gaydarov
 * @author Taariq Fadhill
 * @version 4.20
 * @since 14/02/2022
 *
 */

public class Segment implements Serializable{
    //-------------------------------Private Initial Varriables--------------
-------------//
    // Initializes the number of segments to 0.
    private static int numberOfSegments = 0;
    // Initializes segmentID.
    private int segmentID;
    // Initializes location.
    private double location;
    // Initializes segment types.
    private SegmentType type;
    // Initializes average gradient.
    private double averageGradient;
    // Initializes length.
    private double length;
    // Initializes stage.
    private Stage stage;

    //-----------------------------------Setter methods----------------------
-------------//
    /**
     * Sets the segment location.
     * @param location where the segments will be.
     *
     * Sets the average gradient of a given segment.
     * @param averageGradient the average gradient of the segment as a
percentage.
     *
     * Sets the length of the segment.
     * @param length length of the segment in km.
     *
     * Sets the stage the segment will be held.
     * @param stage the type of stage.
     *
     * Resets the number of segments to 0.
     * @param numberOfSegments the number of segments in a given stage.
     */
    public void setSegementLocation(double location) {this.location =
location;}
    public void setSegmentAverageGradient(double averageGradient)
{this.averageGradient = averageGradient;}
```

```java
{this.averageGradient = averageGradient;}
    public void setSegmentLength(double length) {this.length = length;}
    public void setSegmentStage(Stage stage) {this.stage = stage;}
    public static void resetNumberOfSegments() {numberOfSegments = 0;}


     //-----------------------------------Getter methods-------------------------------------//
    /**
     * Gets the segment ID of the object.
     * @return the segment ID.
     * Gets the location of the Segment.
     * @return location of the segment.
     * Gets the type of segment.
     * @return type of segment in a given stage.
     * Gets the average gradient of a given segment.
     * @return the average gradient as a percentage.
     * Gets the length of a given segment.
     * @return length of the segment in km.
     * Gets the stage a given segment is in.
     * @return stage of a given segment.
     *
     *
     * Gets the details of the segment.
     * @param segmentID the ID of the segment.
     * @param location location of the segment.
     * @param type type of segment.
     * @param averageGradient the average gradient of the segment as a percentage.
     * @param length length of the segment in km.
     * @param stage stage that the segment is in.
     */
     public int getSegmentID() {return segmentID;}
    public double getSegmentLocation() {return location;}
    public SegmentType getSegmentType() {return type;}
    public double getSegmentAverageGradient() {return averageGradient;}
    public double getSegmentLength() {return length;}
    public Stage getSegmentStage() {return stage;}
    public String getSegementDetails() {
        return "ID: "+segmentID+" | Location: "+location+" | Type:: "+type+" | Average Gradient: "+averageGradient+" | Length: "+length+" | Stage: "+stage;
    }



    //---------------------------------Constructer methods-----------------------------------//
    /**
     * Creates object 'Segment' with initialized values of 0, 0 and 0 respectively.
     */
    public Segment() {
        this.location = 0;
        this.segmentID = 0;
        this.averageGradient = 0;
        this.averageGradient = 0;
    }

    /**
     * Creates object 'Rider' with initialized values.
     * @param location location of the segment.
     * @param type type of segment.
     * @param averageGradient average gradient of the segment as a percentage.
     * @param length length of the segment in km.
     */
    public Segment(double location, SegmentType type, double averageGradient, double length) {
        this.location = location;
        this.type = type;
        this.averageGradient = averageGradient;
        this.length = length;
        this.segmentID = ++numberOfSegments;
    }

    Segment(double location, SegmentType type) {
        this.location = location;
        this.type = type;
        this.segmentID = ++numberOfSegments;
    }

}
```

```java
package cycling;
import java.io.Serializable;
import java.time.LocalDateTime;
import java.util.ArrayList;
/**
 * CyclingPortal is a minimally compiling, but non-functioning implementor
 * of the CyclingPortalInterface interface.
 *
 * @author Taariq Fadhill
 * @author Kaloyan Gaydarov
 * @version 4.20
 * @since 14/02/2022
 *
 */

public class Stage implements Serializable{
    //--------------------------------Private Initial Varriables---------------------------//
    // Initializes number of stages to 0.
    private static int numberOfStages = 0;
    // Initializes stageID.
    private int stageID;
    //Initializes stageName.
    private String stageName;
    // Initializes stageDesc.
    private String stageDesc;
    // Initializes length.
    private double length;
    // Initializes type of stage.
    private StageType type;
    // Creates array list 'Segment'.
    private ArrayList<Segment> segmentsArray = new ArrayList<Segment>();;
    // Creates array list 'Results'
    private ArrayList<Results> resultsArray = new ArrayList<Results>();
    // Initializes number of segments to 0.
    private static int numberOfSegments = 0;
    // Initializes start time.
    private LocalDateTime startTime;
    // Initializes prepared object as 'true'.
    private boolean prepared = true;


    //------------------------------------Prepare methods------------------------------------//
    /**
     * Prepares function isPrepared()
     * @return state of is/isn't prepared
     */

    public boolean isPrepared() {return prepared;}
    public void prepare() {prepared = true;}


    //------------------------------------Setter methods------------------------------------//
```

```java
                //
    /**
     * Sets the start time of the stage.
     * @param startTime start time of the stage.
     * Sets the name of the stage.
     * @param stageName name of the stage.
     * Sets the description of the stage.
     * @param stageDesc description of the stage.
     * Sets the length of the stage.
     * @param length of the stage.
     * Sets the type of stage.
     * @param type of stage
     *
     * Resets the number of stages to 0.
     * Resets the number of segements in a given stage to 0.
     */
    public void setStartTime(LocalDateTime startTime){this.startTime = startTime;}
    public void setStageName(String stageName){this.stageName = stageName;}
    public void setStageDesc(String stageDesc){this.stageDesc = stageDesc;}
    public void setStageLength(int length){this.length = length;}
    public void setType(StageType type){this.type = type;}
    public static void resetNumberOfStages() {numberOfStages = 0;}
    public static void resetNumberOfSegments() {numberOfSegments = 0;}


    //-----------------------------------Getter methods-----------------------------------//
    /**
     * Gets the stage ID of the object.
     * @return the stage ID.
     * Gets the name of the stage.
     * @return the name of the stage.
     * Gets the description of the stage.
     * @return description of the stage.
     * Gets the length of the stage.
     * @return the length of the stage in km.
     * Gets the type of stage.
     * @return type of stage.
     * Gets the stage results from the array list 'Results'.
     * @return results in the stage as an array.
     * Gets the start time in the stage.
     * @return the start time of the races in that stage.
     * Gets the segments which are in that stage.
     * @return segments as an array.
     *
     *
     * Gets the details of the stage.
     * @param stageID the ID of the stage.
     * @param stageName the name of the stage.
     * @param stageDesc the description of the stage.
     * @param numberOfSegments the number of segments in the stage.
     * @param stageLength the length of the stage in km.
     */
    public int getStageId() {return stageID;}
    public String getStageName() {return stageName;}
    public String getStageName() {return stageName;}
    public String getStageDescription() {return stageDesc;}
    public double getStageLength() {return length;}
    public StageType getStageType() {return type;}
    public ArrayList<Results> getStageResults() {return resultsArray;}
    public LocalDateTime getStartTime(){return startTime;}
    public Segment[] getStageSegments() {
        Segment[] segmentArray = new Segment[segmentsArray.size()];
        segmentArray = segmentsArray.toArray(segmentArray);
        return segmentArray;
    }
    public String getStageDetails() {
        double stageLength = getStageLength();
        return "ID: "+stageID+" | Name: "+stageName+" | Description: "+stageDesc+" | Number of Segements: "+numberOfSegments+" | Stage Length: "+stageLength;
    }


    //--------------------------------Constructer methods-----------------------------------//
    /**
     * Creates object 'Stage' with initialized values of 'Null', 'Null', 0 and 0 respectively.
     */
    public Stage() {
        this.stageName = "Null";
        this.stageDesc = "Null";
        this.length = 0;
        this.stageID = 0;
    }

    /**
     * Creates object 'Stage' with initialized values.
     * @param stageName the name of the stage.
     * @param stageDesc the description of a given stage.
     * @param length length of the stage in km.
     * @param startTime start time of races in the stage.
     * @param type type of stage.
     */
    public Stage(String stageName, String stageDesc, double length,LocalDateTime startTime, StageType type) {
        this.stageName = stageName;
        this.stageDesc = stageDesc;
        this.length = length;
        this.startTime = startTime;
        this.type = type;
        this.stageID = ++numberOfStages;
    }


    //------------------------------Remover & Adder methods-------------------------------//
    /**
     * Creates method for removing a segment from a given stage.
     * @param segment object 'segement' that will be removed
     * @param segment object 'segement' that will be removed.
     *
     * Creates method for adding a segment to a given stage.
     * @param segment object 'segment' that will be added.
     *
     * Creates method for adding results to a stage.
     * @param result object 'result' that will be added.
     *
     * Creates method for removing results from a stage.
     * @param result object 'result' that will be removed.
     */
    public void removeStageSegment(Segment segment) {
        segmentsArray.remove(segment);
        --numberOfSegments;
    }
    public void addStageSegment(Segment segment) {
        segmentsArray.add(segment);
        ++numberOfSegments;
    }
    public void addStageResults(Results result) {
        resultsArray.add(result);
    }
    public void removeResults(Results result) throws IDNotRecognisedException
{
        resultsArray.remove(result);
    }
}
```

```java
package cycling;
import java.io.Serializable;
import java.util.ArrayList;
/**
 * CyclingPortal is a minimally compiling, but non-functioning implementor
 * of the CyclingPortalInterface interface.
 *
 * @author Taariq Fadhill
 * @author Kaloyan Gaydarov
 * @version 4.20
 * @since 14/02/2022
 *
 */

public class Team implements Serializable{
    //------------------------------Private Initial Varriables---------------------------//
    // Initializes number of teams to 0.
    private static int numberOfTeams = 0;
    // Initializes teamID.
    private int teamID;
    // Initializes teamName.
    private String teamName;
    // Initializes teamDesc.
    private String teamDesc;
    // Initializes points.
    private int points;
    // Creates array list 'Rider'.
    private ArrayList<Rider> teamRiders = new ArrayList<Rider>();
    // Initializes number of riders to 0.
    private static int numberOfRiders = 0;

    //----------------------------------Setter methods------------------------------------//
    /**
     * Sets the name of the team.
     * @param teamName name of the team.
     * Sets the description of the team.
     * @param teamDesc description of the team.
     * Sets the total points for the team.
     * @param points
     * Adds points to the total points for the team.
     * @param points
     * Resets the number of teams to 0.
     * Resets the number of riders in a team to 0.
     */
    public void setTeamName(String teamName){this.teamName = teamName;}
    public void setTeamDesc(String teamDesc){this.teamDesc = teamDesc;}
    public void setTeamPoints(int points){this.points = points;}
    public void addTeamPoints(int points){this.points = this.points + points;}
    public static void resetNumberOfTeams() {numberOfTeams = 0;}
    public static void resetNumberOfRiders() {numberOfRiders= 0;}

    //----------------------------------Getter methods----------------------
    //                                    Getter methods
--------------//
    /**
     * Gets the team ID of the object.
     * @return the team ID.
     * Gets the name of the team.
     * @return the name of the team.
     * Gets the description of the team.
     * @return description of the team.
     * Gets the total points for a team.
     * @return total points.
     * Gets the number of riders in a team.
     * @return number of riders.
     * Gets riders which are in a team.
     * @return riders in a given team as an array.
     * Gets riders based on their riderID.
     * @return riders as an object.
     *
     *
     * Gets the details of the team.
     * @param teamID the ID of the team.
     * @param teamName the name of the team.
     * @param teamDesc description of the team.
     * @param points total points for the team.
     */
    public int getTeamID() {return teamID;}
    public String getTeamName() {return teamName;}
    public String getTeamDesc() {return teamDesc;}
    public int getTeamPoints(){return points;}
    public int getNumberOfTeamRiders(){ return numberOfRiders;}
    public Rider[] getRiders() {
        Rider[] riderArray = new Rider[teamRiders.size()];
        riderArray = teamRiders.toArray(riderArray);
        return riderArray;
    }
    public Rider getRider(int riderID)  {
        for (Rider rider : teamRiders) {
            if (rider.getRiderID() == riderID) {
                return rider;
            }
        }
        return new Rider();
    }
    public String getTeamDetails() {
        return "ID: "+teamID+" | Name: "+teamName+" | Description:
"+teamDesc+" | Points: "+points;
    }

    //---------------------------------Constructer methods-------------------
---------------//
    /**
     * Creates object 'Team' with initialized values of 'Null', 'Null', 0 and
0 respectively.
     */
    public Team() {
        this.teamName = "Null";
        this.teamName = "Null";
        this.teamDesc = "Null";
        this.teamID = 0;
        this.points = 0;
    }

    /**
     * Creates object 'Team' with initialized values.
     * @param teamName the name of the team.
     * @param teamDesc description of a given team.
     * @param teamID ID of the team.
     * @param points points of the team.
     */
    public Team(String teamName, String teamDesc) {
        this.teamName = teamName;
        this.teamDesc = teamDesc;
        this.teamID = ++numberOfTeams;
        this.points = 0;
    }

     //---------------------------------Remover & Adder methods----------------------------------//
    /**
     * Creates method for removing a rider from a given team.
     * @param rider object 'rider' that will be removed.
     *
     * Creates method for adding a rider to a given team.
     * @param rider object 'rider' that will be added.
     */
     public void removeRider(Rider rider) {
        teamRiders.remove(rider);
        --numberOfRiders;
    }
    public void addRider(Rider newRider) {
        teamRiders.add(newRider);
        numberOfRiders++;
    }
}
```