CyclingPortal Printout

123456789 & 987654321

Contents

1 CategorizedClimb.java

```java
package cycling;

/**
 * Categorised Climb class. This represents a type of Segment that has a stage, location, type,
 * average gradient and a length.
 */
public class CategorizedClimb extends Segment {
  private final Double averageGradient;
  private final Double length;

  /**
   * Constructor method that sets up the Categorised Climb with a stage, location, type, average
   * gradient and length.
   *
   * @param stage that the Categorised Climb is in.
   * @param location of the Categorised Climb.
   * @param type of Categorised Climb.
   * @param averageGradient of the Categorised Climb.
   * @param length of the Categorised Climb.
   * @throws InvalidLocationException Thrown if the location is out of bounds of the Stage length.
   * @throws InvalidStageStateException Thrown if the Stage is waiting for results.
   * @throws InvalidStageTypeException Thrown if the type is a time trial.
   */
  public CategorizedClimb(
      Stage stage, Double location, SegmentType type, Double averageGradient, Double length)
      throws InvalidLocationException, InvalidStageStateException, InvalidStageTypeException {
    super(stage, type, location);
    this.averageGradient = averageGradient;
    this.length = length;
  }
}
```

2 CyclingPortal.java

```java
package cycling;

import java.io.*;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.ArrayList;
import java.util.List;

// TODO:
//    - Documentation/Comments

public class CyclingPortal implements CyclingPortalInterface {
  // ArrayLists for all of a cycling portal instances teams, riders, races, stages and segments.
  // Although HashMaps could have been used here to get riders by int ID, it would be slower in the
  // long run as we would need to constantly convert it back to arrays to output results.
  private ArrayList<Team> teams = new ArrayList<>();
  private ArrayList<Rider> riders = new ArrayList<>();
  private ArrayList<Race> races = new ArrayList<>();
  private ArrayList<Stage> stages = new ArrayList<>();
  private ArrayList<Segment> segments = new ArrayList<>();

  /**
   * Determine if a string contains any illegal whitespace characters.
   *
   * @param string The input string to be tested for whitespace.
   * @return A boolean, true if the input string contains whitespace, false if not.
   */
  public static boolean containsWhitespace(String string) {
    for (int i = 0; i < string.length(); ++i) {
      if (Character.isWhitespace(string.charAt(i))) {
        return true;
      }
    }
    return false;
  }

  /**
   * Get a Team object by a Team ID.
   *
   * @param ID The int ID of the Team to be looked up.
   * @return The Team object of the team, if one is found.
   * @throws IDNotRecognisedException Thrown if no team is found with the given Team ID.
   */
  public Team getTeamById(int ID) throws IDNotRecognisedException {
    for (Team team : teams) {
      if (team.getId() == ID) {
        return team;
      }
    }
    throw new IDNotRecognisedException("Team ID not found.");
  }

  /**
   * Get a Rider object by a Rider ID.
   *
   * @param ID The int ID of the Rider to be looked up.
   * @return The Rider object of the Rider, if one is found.
   * @throws IDNotRecognisedException Thrown if no rider is found with the given Rider ID.
   */
  public Rider getRiderById(int ID) throws IDNotRecognisedException {
    for (Rider rider : riders) {
      if (rider.getId() == ID) {
        return rider;
      }
    }
    throw new IDNotRecognisedException("Rider ID not found.");
  }

  /**
   * Get a Race object by a Race ID.
   *
   * @param ID The int ID of the Race to be looked up.
   * @return The Race object of the race, if one is found.
   * @throws IDNotRecognisedException Thrown if no race is found with the given Race ID.
   */
  public Race getRaceById(int ID) throws IDNotRecognisedException {
    for (Race race : races) {
      if (race.getId() == ID) {
        return race;
      }
    }
    throw new IDNotRecognisedException("Race ID not found.");
  }

  /**
   * Get a Stage object by a Stage ID.
   *
   * @param ID The int ID of the Stage to be looked up.
   * @return The Stage object of the stage, if one is found.
   * @throws IDNotRecognisedException Thrown if no stage is found with the given Stage ID.
   */
  public Stage getStageById(int ID) throws IDNotRecognisedException {
    for (Stage stage : stages) {
      if (stage.getId() == ID) {
        return stage;
      }
    }
    throw new IDNotRecognisedException("Stage ID not found.");
  }

  /**
   * Get a Segment object by a Segment ID.
   *
   * @param ID The int ID of the Segment to be looked up.
   * @return The Segment object of the segment, if one is found.
   * @throws IDNotRecognisedException Thrown if no segment is found with the given Segment ID.
   */
  public Segment getSegmentById(int ID) throws IDNotRecognisedException {
    for (Segment segment : segments) {
      if (segment.getId() == ID) {
        return segment;
      }
    }
    throw new IDNotRecognisedException("Segment ID not found.");
  }

  /**
   * Loops over all races, stages and segments to remove all of a given riders results.
   *
   * @param rider The Rider object whose results will be removed from the Cycling Portal.
   */
  public void removeRiderResults(Rider rider) {
    for (Race race : races) {
      race.removeRiderResults(rider);
    }
    for (Stage stage : stages) {
      stage.removeRiderResults(rider);
    }
    for (Segment segment : segments) {
      segment.removeRiderResults(rider);
    }
  }

  @Override
  public int[] getRaceIds() {
    int[] raceIDs = new int[races.size()];
    for (int i = 0; i < races.size(); i++) {
      Race race = races.get(i);
      raceIDs[i] = race.getId();
    }
    return raceIDs;
  }

  @Override
  public int createRace(String name, String description)
      throws IllegalNameException, InvalidNameException {
    // Check a race with this name does not already exist in the system.
    for (Race race : races) {
      if (race.getName().equals(name)) {
        throw new IllegalNameException("A Race with the name " + name + " already exists.");
      }
    }
    Race race = new Race(name, description);
    races.add(race);
    return race.getId();
  }

  @Override
  public String viewRaceDetails(int raceId) throws IDNotRecognisedException {
    Race race = getRaceById(raceId);
    return race.getDetails();
  }

  @Override
  public void removeRaceById(int raceId) throws IDNotRecognisedException {
    Race race = getRaceById(raceId);
    // Remove all the races stages from the CyclingPortal.
    for (final Stage stage : race.getStages()) {
      stages.remove(stage);
    }
    races.remove(race);
  }

  @Override
  public int getNumberOfStages(int raceId) throws IDNotRecognisedException {
    Race race = getRaceById(raceId);
    return race.getStages().size();
  }

  @Override
  public int addStageToRace(
      int raceId,
      String stageName,
      String description,
      double length,
      LocalDateTime startTime,
      StageType type)
      throws IDNotRecognisedException, IllegalNameException, InvalidNameException,
          InvalidLengthException {
    Race race = getRaceById(raceId);
    // Check a stage with this name does not already exist in the system.
    for (final Stage stage : stages) {
      if (stage.getName().equals(stageName)) {
        throw new IllegalNameException("A stage with the name " + stageName + " already exists.");
      }
    }
    Stage stage = new Stage(race, stageName, description, length, startTime, type);
    stages.add(stage);
    race.addStage(stage);
    return stage.getId();
  }

  @Override
  public int[] getRaceStages(int raceId) throws IDNotRecognisedException {
    Race race = getRaceById(raceId);
    ArrayList<Stage> raceStages = race.getStages();
    int[] raceStagesId = new int[raceStages.size()];
    // Gathers the Stage ID's of the Stages in the Race.
    for (int i = 0; i < raceStages.size(); i++) {
      Stage stage = race.getStages().get(i);
      raceStagesId[i] = stage.getId();
    }
    return raceStagesId;
  }

  @Override
  public double getStageLength(int stageId) throws IDNotRecognisedException {
    Stage stage = getStageById(stageId);
    return stage.getLength();
  }

  @Override
  public void removeStageById(int stageId) throws IDNotRecognisedException {
    Stage stage = getStageById(stageId);
    Race race = stage.getRace();
    // Removes stage from both the Races and Stages.
    race.removeStage(stage);
    stages.remove(stage);
  }

  @Override
  public int addCategorizedClimbToStage(
      int stageId, Double location, SegmentType type, Double averageGradient, Double length)
      throws IDNotRecognisedException, InvalidLocationException, InvalidStageStateException,
          InvalidStageTypeException {
    Stage stage = getStageById(stageId);
    CategorizedClimb climb = new CategorizedClimb(stage, location, type, averageGradient, length);
    // Adds Categorized Climb to both the list of Segments and the Stage.
    segments.add(climb);
    stage.addSegment(climb);
    return climb.getId();
  }

  @Override
  public int addIntermediateSprintToStage(int stageId, double location)
      throws IDNotRecognisedException, InvalidLocationException, InvalidStageStateException,
          InvalidStageTypeException {
    Stage stage = getStageById(stageId);
    IntermediateSprint sprint = new IntermediateSprint(stage, location);
    // Adds Intermediate Sprint to both the list of Segments and the Stage.
    segments.add(sprint);
    stage.addSegment(sprint);
    return sprint.getId();
  }

  @Override
  public void removeSegment(int segmentId)
      throws IDNotRecognisedException, InvalidStageStateException {
    Segment segment = getSegmentById(segmentId);
    Stage stage = segment.getStage();
    // Removes Segment from both the Stage and list of Segments.
    stage.removeSegment(segment);
    segments.remove(segment);
  }

  @Override
  public void concludeStagePreparation(int stageId)
      throws IDNotRecognisedException, InvalidStageStateException {
    Stage stage = getStageById(stageId);
    stage.concludePreparation();
  }

  @Override
  public int[] getStageSegments(int stageId) throws IDNotRecognisedException {
    Stage stage = getStageById(stageId);
    ArrayList<Segment> stageSegments = stage.getSegments();
    int[] stageSegmentsId = new int[stageSegments.size()];
    // Gathers Segment ID's from the Segments in the Stage.
    for (int i = 0; i < stageSegments.size(); i++) {
      Segment segment = stageSegments.get(i);
      stageSegmentsId[i] = segment.getId();
    }
    return stageSegmentsId;
  }

  @Override
  public int createTeam(String name, String description)
      throws IllegalNameException, InvalidNameException {
    // Checks if the Team name already exists on the system.
    for (final Team team : teams) {
      if (team.getName().equals(name)) {
        throw new IllegalNameException("A Team with the name " + name + " already exists.");
      }
    }
    Team team = new Team(name, description);
    teams.add(team);
    return team.getId();
  }

  @Override
  public void removeTeam(int teamId) throws IDNotRecognisedException {
    Team team = getTeamById(teamId);
    // Loops through and removes Team Riders and Team Rider Results.
    for (final Rider rider : team.getRiders()) {
      removeRiderResults(rider);
      riders.remove(rider);
    }
    teams.remove(team);
  }

  @Override
```

```java
  public int[] getTeams() {
    int[] teamIDs = new int[teams.size()];
    for (int i = 0; i < teams.size(); i++) {
      Team team = teams.get(i);
      teamIDs[i] = team.getId();
    }
    return teamIDs;
  }

  @Override
  public int[] getTeamRiders(int teamId) throws IDNotRecognisedException {
    Team team = getTeamById(teamId);
    ArrayList<Rider> teamRiders = team.getRiders();
    int[] teamRiderIds = new int[teamRiders.size()];
    // Gathers ID's of Riders in the Team.
    for (int i = 0; i < teamRiderIds.length; i++) {
      // Assert the rider is actually on the team.
      assert teamRiders.get(i).getTeam().equals(team);
      // Return the rider id.
      teamRiderIds[i] = teamRiders.get(i).getId();
    }
    return teamRiderIds;
  }

  @Override
  public int createRider(int teamID, String name, int yearOfBirth)
      throws IDNotRecognisedException, IllegalArgumentException {
    Team team = getTeamById(teamID);
    Rider rider = new Rider(team, name, yearOfBirth);
    // Adds Rider to both the Team and the list of Riders.
    team.addRider(rider);
    riders.add(rider);

    // Assert at least one rider has been added
    assert riders.size() > 0;

    return rider.getId();
  }

  @Override
  public void removeRider(int riderId) throws IDNotRecognisedException {
    Rider rider = getRiderById(riderId);
    removeRiderResults(rider);
    // Removes Rider from both the Team and the list of Riders.
    rider.getTeam().removeRider(rider);
    riders.remove(rider);
  }

  @Override
  public void registerRiderResultsInStage(int stageId, int riderId, LocalTime... checkpoints)
      throws IDNotRecognisedException, DuplicatedResultException, InvalidCheckpointsException,
          InvalidStageStateException {
    Stage stage = getStageById(stageId);
    Rider rider = getRiderById(riderId);
    stage.registerResult(rider, checkpoints);
  }

  @Override
  public LocalTime[] getRiderResultsInStage(int stageId, int riderId)
      throws IDNotRecognisedException {
    Stage stage = getStageById(stageId);
    Rider rider = getRiderById(riderId);
    StageResult result = stage.getRiderResult(rider);

    if (result == null) {
      // Returns an empty array if the Result is null.
      return new LocalTime[] {};
    } else {
      LocalTime[] checkpoints = result.getCheckpoints();
      // Rider Results will always be 1 shorter than the checkpoint length because
      // the finish time checkpoint will be replaced with the Elapsed Time and the start time
      // checkpoint will be ignored.
      LocalTime[] resultsInStage = new LocalTime[checkpoints.length - 1];
      LocalTime elapsedTime = LocalTime.MIDNIGHT.plus(result.getElapsedTime());
      for (int i = 0; i < resultsInStage.length; i++) {
        if (i == resultsInStage.length - 1) {
          // Adds the Elapsed Time to the end of the array of Results.
          resultsInStage[i] = elapsedTime;
        } else {
          // Adds each checkpoint to the array of Results until all have been added, skipping the
          // Start time checkpoint.
          resultsInStage[i] = checkpoints[i + 1];
        }
      }
      return resultsInStage;
    }
  }

  @Override
  public LocalTime getRiderAdjustedElapsedTimeInStage(int stageId, int riderId)
      throws IDNotRecognisedException {
    Stage stage = getStageById(stageId);
    Rider rider = getRiderById(riderId);
    StageResult result = stage.getRiderResult(rider);
    if (result == null) {
      return null;
    } else {
      return result.getAdjustedElapsedLocalTime();
    }
  }

  @Override
  public void deleteRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException {
    Stage stage = getStageById(stageId);
    Rider rider = getRiderById(riderId);
    stage.removeRiderResults(rider);
  }

  @Override
  public int[] getRidersRankInStage(int stageId) throws IDNotRecognisedException {
    Stage stage = getStageById(stageId);
    // Gets a list of Riders from the Stage ordered by their Elapsed Times.
    List<Rider> riders = stage.getRidersByElapsedTime();
    int[] riderIds = new int[riders.size()];
    // Gathers ID's from the ordered list of Riders.
    for (int i = 0; i < riders.size(); i++) {
      riderIds[i] = riders.get(i).getId();
    }
    return riderIds;
  }

  @Override
  public LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId)
      throws IDNotRecognisedException {
    Stage stage = getStageById(stageId);
    // Gets a list of Riders from the Stage ordered by their Elapsed Times.
    List<Rider> riders = stage.getRidersByElapsedTime();
    LocalTime[] riderAETs = new LocalTime[riders.size()];
    // Gathers Riders' Adjusted Elapsed Times ordered by their Elapsed Times.
    for (int i = 0; i < riders.size(); i++) {
      Rider rider = riders.get(i);
      riderAETs[i] = stage.getRiderResult(rider).getAdjustedElapsedLocalTime();
    }
    return riderAETs;
  }

  @Override
  public int[] getRidersPointsInStage(int stageId) throws IDNotRecognisedException {
    Stage stage = getStageById(stageId);
    // Gets a list of Riders from the Stage ordered by their Elapsed Times.
    List<Rider> riders = stage.getRidersByElapsedTime();
    int[] riderSprinterPoints = new int[riders.size()];
    // Gathers Sprinters' Points ordered by their Elapsed Times.
    for (int i = 0; i < riders.size(); i++) {
      Rider rider = riders.get(i);
      riderSprinterPoints[i] = stage.getRiderResult(rider).getSprintersPoints();
    }
    return riderSprinterPoints;
  }

  @Override
  public int[] getRidersMountainPointsInStage(int stageId) throws IDNotRecognisedException {
    Stage stage = getStageById(stageId);
    // Gets a list of Riders from the Stage ordered by their Elapsed Times.
    List<Rider> riders = stage.getRidersByElapsedTime();
    int[] riderMountainPoints = new int[riders.size()];
    // Gathers Riders' Mountain Points ordered by their Elapsed Times.
    for (int i = 0; i < riders.size(); i++) {
      Rider rider = riders.get(i);
      riderMountainPoints[i] = stage.getRiderResult(rider).getMountainPoints();
    }
    return riderMountainPoints;
  }

  @Override
  public void eraseCyclingPortal() {
    // Replaces teams, riders, races, stages and segments with empty ArrayLists.
    teams = new ArrayList<>();
    riders = new ArrayList<>();
    races = new ArrayList<>();
    stages = new ArrayList<>();
    segments = new ArrayList<>();
    // Sets the ID counters of the Rider, Team, Race, Stage and Segment objects back
    // to 0.
    Rider.resetIdCounter();
    Team.resetIdCounter();
    Race.resetIdCounter();
    Stage.resetIdCounter();
    Segment.resetIdCounter();

    // Assert the portal is erased.
    assert teams.size() == 0;
    assert races.size() == 0;
  }

  @Override
  public void saveCyclingPortal(String filename) throws IOException {
    FileOutputStream file = new FileOutputStream(filename + ".ser");
    ObjectOutputStream output = new ObjectOutputStream(file);
    // Saves teams, riders, races, stages and segments ArrayLists.
    // Saves ID counters of Team, Rider, Race, Stage and Segment objects.
    SavedCyclingPortal savedCyclingPortal =
        new SavedCyclingPortal(
            teams,
            riders,
            races,
            stages,
            segments,
            Team.getIdCounter(),
            Rider.getIdCounter(),
            Race.getIdCounter(),
            Stage.getIdCounter(),
            Segment.getIdCounter());
    output.writeObject(savedCyclingPortal);
    output.close();
    file.close();
  }

  @Override
  public void loadCyclingPortal(String filename) throws IOException, ClassNotFoundException {
    eraseCyclingPortal();
    FileInputStream file = new FileInputStream(filename + ".ser");
    ObjectInputStream input = new ObjectInputStream(file);

    SavedCyclingPortal savedCyclingPortal = (SavedCyclingPortal) input.readObject();
    // Imports teams, riders, races, stages and segments ArrayLists from the last save.
    teams = savedCyclingPortal.teams;
    riders = savedCyclingPortal.riders;
    races = savedCyclingPortal.races;
    stages = savedCyclingPortal.stages;
    segments = savedCyclingPortal.segments;

    // Imports ID counters of Team, Rider, Race, Stage and Segment objects from the last save.
    Team.setIdCounter(savedCyclingPortal.teamIdCount);
    Rider.setIdCounter(savedCyclingPortal.riderIdCount);
    Race.setIdCounter(savedCyclingPortal.raceIdCount);
    Stage.setIdCounter(savedCyclingPortal.stageIdCount);
    Segment.setIdCounter(savedCyclingPortal.segmentIdCount);

    input.close();
    file.close();
  }
```

```java

  @Override
  public void removeRaceByName(String name) throws NameNotRecognisedException {
    for (final Race race : races) {
      if (race.getName().equals(name)) {
        races.remove(race);
        return;
      }
    }
    throw new NameNotRecognisedException("Race name is not in the system.");
  }

  @Override
  public int[] getRidersGeneralClassificationRank(int raceId) throws IDNotRecognisedException {
    Race race = getRaceById(raceId);
    List<Rider> riders = race.getRidersByAdjustedElapsedTime();
    int[] riderIds = new int[riders.size()];
    // Gathers Rider ID's ordered by their Adjusted Elapsed Times.
    for (int i = 0; i < riders.size(); i++) {
      riderIds[i] = riders.get(i).getId();
    }
    return riderIds;
  }

  @Override
  public LocalTime[] getGeneralClassificationTimesInRace(int raceId)
      throws IDNotRecognisedException {
    Race race = getRaceById(raceId);
    // Gets a list of Riders from the Stage ordered by their Adjusted Elapsed Times.
    List<Rider> riders = race.getRidersByAdjustedElapsedTime();
    LocalTime[] riderTimes = new LocalTime[riders.size()];
    // Gathers Riders' Cumulative Adjusted Elapsed LocalTimes ordered by their Adjusted Elapsed
    // Times.
    for (int i = 0; i < riders.size(); i++) {
      riderTimes[i] = race.getRiderResults(riders.get(i)).getCumulativeAdjustedElapsedLocalTime();
    }
    return riderTimes;
  }

  @Override
  public int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException {
    Race race = getRaceById(raceId);
    List<Rider> riders = race.getRidersByAdjustedElapsedTime();
    int[] riderIds = new int[riders.size()];
    // Gathers Riders' Cumulative Sprinters Points ordered by their Adjusted Elapsed Times.
    for (int i = 0; i < riders.size(); i++) {
      riderIds[i] = race.getRiderResults(riders.get(i)).getCumulativeSprintersPoints();
    }
    return riderIds;
  }

  @Override
  public int[] getRidersMountainPointsInRace(int raceId) throws IDNotRecognisedException {
    Race race = getRaceById(raceId);
    List<Rider> riders = race.getRidersByAdjustedElapsedTime();
    int[] riderIds = new int[riders.size()];
    // Gathers Riders' Cumulative Mountain Points ordered by their Adjusted Elapsed Times.
    for (int i = 0; i < riders.size(); i++) {
      riderIds[i] = race.getRiderResults(riders.get(i)).getCumulativeMountainPoints();
    }
    return riderIds;
  }

  @Override
  public int[] getRidersPointClassificationRank(int raceId) throws IDNotRecognisedException {
    Race race = getRaceById(raceId);
    List<Rider> riders = race.getRidersBySprintersPoints();
    int[] riderIds = new int[riders.size()];
    // Gathers Rider ID's ordered by their Sprinters Points.
    for (int i = 0; i < riders.size(); i++) {
      riderIds[i] = riders.get(i).getId();
    }
    return riderIds;
  }

  @Override
  public int[] getRidersMountainPointClassificationRank(int raceId)
      throws IDNotRecognisedException {
    Race race = getRaceById(raceId);
    List<Rider> riders = race.getRidersByMountainPoints();
    int[] riderIds = new int[riders.size()];
    // Gathers Rider ID's ordered by their Mountain Points.
    for (int i = 0; i < riders.size(); i++) {
      riderIds[i] = riders.get(i).getId();
    }
    return riderIds;
  }
}
```

3 IntermediateSprint.java

```java
package cycling;

/** Intermediate Sprint class. This represents a type of Segment that has a stage and a location.
↪  */
public class IntermediateSprint extends Segment {
  private final double location;

  /**
   * Constructor method that sets the Intermediate Sprint up with a stage and a location.
   *
   * @param stage of the Intermediate Sprint.
   * @param location of the Intermediate Sprint
   * @throws InvalidLocationException Thrown if the location is out of bounds of the Stage length.
   * @throws InvalidStageStateException Thrown if the Stage is waiting for results.
   * @throws InvalidStageTypeException Thrown if the type is a time trial.
   */
  public IntermediateSprint(Stage stage, double location)
      throws InvalidLocationException, InvalidStageTypeException, InvalidStageStateException {
    super(stage, SegmentType.SPRINT, location);
    this.location = location;
  }
}
```

4 Race.java

```java
package cycling;

import java.io.Serializable;
import java.time.LocalDateTime;
import java.util.*;
import java.util.stream.Collectors;

/**
 * Race Class. This represents a Race that holds a Race's Stages, Riders Results, and also contains
 * methods that deal with these.
 */
public class Race implements Serializable {

  private final String name;
  private final String description;

  private final ArrayList<Stage> stages = new ArrayList<>();

  private HashMap<Rider, RaceResult> results = new HashMap<>();

  private static int count = 0;
  private final int id;

  /**
   * Constructor method that sets up Rider with a name and a description.
   *
   * @param name: Cannot be empty, null, have a length greater than 30 or contain any whitespace.
   * @param description: A description of the race.
   * @throws InvalidNameException Thrown if the Race name does not meet name requirements stated
   *     above.
   */
  public Race(String name, String description) throws InvalidNameException {
    if (name == null
        || name.isEmpty()
        || name.length() > 30
        || CyclingPortal.containsWhitespace(name)) {
      throw new InvalidNameException(
          "The name cannot be null, empty, have more than 30 characters, or have white spaces.");
    }
    this.name = name;
    this.description = description;
    // ID counter represents the highest known ID at the current time to ensure there
    // are no ID collisions.
    this.id = Race.count++;
  }

  /** Method that resets the static ID counter of the Race. Used for erasing and loading. */
  static void resetIdCounter() {
    count = 0;
  }

  /**
   * Method to get the current state of the static ID counter.
   *
   * @return the highest race ID stored currently.
   */
  static int getIdCounter() {
    return count;
  }

  /**
   * Method that sets the static ID counter to a given value. Used when loading to avoid ID
   * collisions.
   *
   * @param newCount: new value of the static ID counter.
   */
  static void setIdCounter(int newCount) {
    count = newCount;
  }

  /**
   * Method to get the ID of the Race object.
   *
   * @return id: the Race's unique ID value.
   */
  public int getId() {
    return id;
  }

  /**
   * Method to get the name of the Race.
   *
   * @return name: the given name of the Race.
   */
  public String getName() {
    return name;
  }

  /**
   * Method that adds a Stage to the Race object's ordered list of Stages. It is added to the
   * correct position based on its start time.
   *
   * @param stage: The stage to be added to the Race.
   */
  public void addStage(Stage stage) {
    // Loops over stages in the race to insert the new stage in the correct place such that
    // all of the stages are sorted by their start time.
    for (int i = 0; i < stages.size(); i++) {
      // Retrieves the start time of each Stage in the Race's current Stages one by one.
      // These are already ordered by their start times.
      LocalDateTime iStartTime = stages.get(i).getStartTime();
      // Adds the new Stage to the list of stages in the correct position based on
      // its start time.
      if (stage.getStartTime().isBefore(iStartTime)) {
        stages.add(i, stage);
        return;
      }
    }
    stages.add(stage);
  }

  /**
   * Method to get the list of Stages in the Race ordered by their start times.
   *
```

```java
   * @return stages: The ordered list of Stages.
   */
  public ArrayList<Stage> getStages() {
    // stages is already sorted, so no sorting needs to be done.
    return stages;
  }

  /**
   * Method that removes a given Stage from the list of Stages.
   *
   * @param stage: the Stage to be deleted.
   */
  public void removeStage(Stage stage) {
    stages.remove(stage);
  }

  /**
   * Method to get then details of a Race including Race ID, name, description number of stages and
   * total length.
   *
   * @return Concatenated paragraph of race details.
   */
  public String getDetails() {
    double currentLength = 0;
    for (final Stage stage : stages) {
      currentLength = currentLength + stage.getLength();
    }
    return ("Race ID: "
        + id
        + System.lineSeparator()
        + "Name: "
        + name
        + System.lineSeparator()
        + "Description: "
        + description
        + System.lineSeparator()
        + "Number of Stages: "
        + stages.size()
        + System.lineSeparator()
        + "Total length: "
        + currentLength);
  }

  /**
   * Method to get a list of Riders in the Race, sorted by their Adjusted Elapsed Time.
   *
   * @return The correctly sorted Riders.
   */
  public List<Rider> getRidersByAdjustedElapsedTime() {
    // First generate the race result to calculate each riders Adjusted Elapsed Time.
    calculateResults();
    // Then return the riders sorted by their Adjusted Elapsed Time.
    return sortRiderResultsBy(RaceResult.sortByAdjustedElapsedTime);
  }

  /**
   * Method to get a list of Riders in the Race, sorted by their Sprinters Points.
   *
   * @return The correctly sorted Riders.
   */
  public List<Rider> getRidersBySprintersPoints() {
    // First generate the race result to calculate each riders Sprinters Points.
    calculateResults();
    // Then return the riders sorted by their sprinters points.
    return sortRiderResultsBy(RaceResult.sortBySprintersPoints);
  }

  /**
   * Method to get a list of Riders in the Race, sorted by their Mountain Points.
   *
   * @return The correctly sorted Riders.
   */
  public List<Rider> getRidersByMountainPoints() {
    // First generate the race result to calculate each riders Mountain Points.
    calculateResults();
    // Then return the riders sorted by their mountain points.
    return sortRiderResultsBy(RaceResult.sortByMountainPoints);
  }

  /**
   * Method to get the results of a given Rider.
   *
   * @param rider: Rider to get the results of.
   * @return RaceResult: Result of the Rider.
   */
  public RaceResult getRiderResults(Rider rider) {
    // First generate the race result to calculate each riders results.
    calculateResults();
    // Then return the riders result object.
    return results.get(rider);
  }

  /**
   * Method to remove the Results of a given Rider.
   *
   * @param rider: Rider whose Results will be removed.
   */
  public void removeRiderResults(Rider rider) {
    results.remove(rider);
  }

  /**
   * Method to get a list of Riders sorted by a given comparator of their Results. Will only return
   * riders who have results registered in their name.
   *
   * @param comparison: a comparator on the Riders' Results to sort the Riders by.
   * @return List<Rider>: List of Riders (who posses recorded results) sorted by the comparator on
   *     the Results.
   */
  private List<Rider> sortRiderResultsBy(Comparator<RaceResult> comparator) {
    // convert the hashmap into a set
    return results.entrySet().stream()
        // Sort the set by the comparator on the results.
        .sorted(Map.Entry.comparingByValue(comparator))
        // Get the rider element of the set and ignore the results now they have been sorted.
        .map(Map.Entry::getKey)
```

```java
        // Convert to a list of riders.
        .collect(Collectors.toList());
  }

  /**
   * Method to register the Rider's Result to the Stage.
   *
   * @param rider: Rider whose Result needs to be registered.
   * @param stageResult: Stage that the Result will be added to.
   */
  private void registerRiderResults(Rider rider, StageResult stageResult) {
    if (results.containsKey(rider)) {
      // If results already exist for a given rider add the current stage results
      // to the existing total race results.
      results.get(rider).addStageResult(stageResult);
    } else {
      // If no race results exists, create a new RaceResult object based on the current
      // stage results.
      RaceResult raceResult = new RaceResult();
      raceResult.addStageResult(stageResult);
      results.put(rider, raceResult);
    }
  }

  /** Private method that calculates the results for each Rider. */
  private void calculateResults() {
    // Clear existing results.
    results = new HashMap<>();
    // We must loop over all stages and collect their results for each rider as each riders results
    // are dependent on their position in the race, and thus the results of the other riders.
    for (Stage stage : stages) {
      HashMap<Rider, StageResult> stageResults = stage.getStageResults();
      for (Rider rider : stageResults.keySet()) {
        registerRiderResults(rider, stageResults.get(rider));
      }
    }
  }
}
```

5 RaceResult.java

```java
package cycling;

import java.io.Serializable;
import java.time.Duration;
import java.time.LocalTime;
import java.util.Comparator;

/**
 * This represents a given riders results in a race. The riders adjusted elapsed time, sprinters
 * points and mountain points over all stages and segments are recorded here.
 */
public class RaceResult implements Serializable {
  private Duration cumulativeAdjustedElapsedTime = Duration.ZERO;
  private int cumulativeSprintersPoints = 0;
  private int cumulativeMountainPoints = 0;

  // A comparator which sorts RaceResults based on Adjusted Elapsed Time in ascending order. The
  // result with the shortest time will come first.
  protected static final Comparator<RaceResult> sortByAdjustedElapsedTime =
      Comparator.comparing(RaceResult::getCumulativeAdjustedElapsedTime);

  // A comparator which sorts RaceResults based on Sprinters Points in descending order. The result
  // with the most points will come first.
  protected static final Comparator<RaceResult> sortBySprintersPoints =
      (RaceResult result1, RaceResult result2) ->
          Integer.compare(
              result2.getCumulativeSprintersPoints(), result1.getCumulativeSprintersPoints());

  // A comparator which sorts RaceResults based on Mountain Points in descending order. The result
  // with the most points will come first.
  protected static final Comparator<RaceResult> sortByMountainPoints =
      (RaceResult result1, RaceResult result2) ->
          Integer.compare(
              result2.getCumulativeMountainPoints(), result1.getCumulativeMountainPoints());

  /**
   * A method to get the recorded Adjusted Elapsed Time over all stages.
   *
   * @return The cumulative adjusted elapsed time as a duration.
   */
  public Duration getCumulativeAdjustedElapsedTime() {
    return this.cumulativeAdjustedElapsedTime;
  }

  /**
   * A method to get the recorded Adjusted Elapsed Time over all stages as a LocalTime.
   *
   * @return The cumulative adjusted elapsed time as a Local Time
   */
  public LocalTime getCumulativeAdjustedElapsedLocalTime() {
    // Calculated the AET as a Local time by adding the duration to midnight: 0:00 + Duration
    return LocalTime.MIDNIGHT.plus(this.cumulativeAdjustedElapsedTime);
  }

  /**
   * A method to get the recorded Mountain Points over all stages and segments.
   *
   * @return The cumulative mountain points.
   */
  public int getCumulativeMountainPoints() {
    return this.cumulativeMountainPoints;
  }

  /**
   * A method to get the recorded Sprinters Points over all stages and segments.
   *
   * @return The cumulative sprinters points.
   */
  public int getCumulativeSprintersPoints() {
    return this.cumulativeSprintersPoints;
  }

  /**
```

```java
   * A method to add a stage result to the race result. This is useful as a riders results in a
↪  race
   * is just a sum of their results in all a races stages. E.g. RaceResults = Stage1Result +
   * Stage2Result + Stage3Result + ...
   *
   * @param stageResult the stage results which should be added to a race result.
   */
  public void addStageResult(StageResult stageResult) {
    this.cumulativeAdjustedElapsedTime =
        this.cumulativeAdjustedElapsedTime.plus(stageResult.getAdjustedElapsedTime());
    this.cumulativeSprintersPoints += stageResult.getSprintersPoints();
    this.cumulativeMountainPoints += stageResult.getMountainPoints();
  }
}
```

6 Rider.java

```java
package cycling;

import java.io.Serializable;

public class Rider implements Serializable {
  private final Team team;
  private final String name;
  private final int yearOfBirth;

  private static int count = 0;
  private final int id;

  public Rider(Team team, String name, int yearOfBirth) throws IllegalArgumentException {
    if (name == null) {
      throw new java.lang.IllegalArgumentException("The rider's name cannot be null.");
    }
    if (yearOfBirth < 1900) {
      throw new java.lang.IllegalArgumentException(
          "The rider's birth year is invalid, must be greater than 1900.");
    }

    this.team = team;
    this.name = name;
    this.yearOfBirth = yearOfBirth;
    this.id = Rider.count++;
  }

  static void resetIdCounter() {
    count = 0;
  }

  static int getIdCounter() {
    return count;
  }

  static void setIdCounter(int newCount) {
    count = newCount;
  }

  public int getId() {
    return id;
  }

  public Team getTeam() {
    return team;
  }
}
```

7 SavedCyclingPortal.java

```java
package cycling;

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

public class SavedCyclingPortal implements Serializable {
  final ArrayList<Team> teams;
  final ArrayList<Rider> riders;
  final ArrayList<Race> races;
  final ArrayList<Stage> stages;
  final ArrayList<Segment> segments;
  final int teamIdCount;
  final int riderIdCount;
  final int raceIdCount;
  final int stageIdCount;
  final int segmentIdCount;

  public SavedCyclingPortal(
      ArrayList<Team> teams,
      ArrayList<Rider> riders,
      ArrayList<Race> races,
      ArrayList<Stage> stages,
      ArrayList<Segment> segments,
      int teamIdCount,
      int riderIdCount,
      int raceIdCount,
      int stageIdCount,
      int segmentIdCount) {
    this.teams = teams;
    this.riders = riders;
    this.races = races;
    this.stages = stages;
    this.segments = segments;
    this.teamIdCount = teamIdCount;
    this.riderIdCount = riderIdCount;
    this.raceIdCount = raceIdCount;
    this.stageIdCount = stageIdCount;
    this.segmentIdCount = segmentIdCount;
  }
}
```

8 Segment.java

```java
package cycling;

import java.io.Serializable;
import java.time.LocalTime;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

/**
 * Segment Class. This represents a segment of a stage in a rice in the cycling portal. This deals
 * with details about the segment as well as the segments results.
 */
public class Segment implements Serializable {
  private static int count = 0;
  private final Stage stage;
  private final int id;
  private final SegmentType type;
  private final double location;

  private final HashMap<Rider, SegmentResult> results = new HashMap<>();

  // Segment sprinters/mountain points.
  private static final int[] SPRINT_POINTS = {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
  private static final int[] HC_POINTS = {20, 15, 12, 10, 8, 6, 4, 2};
  private static final int[] C1_POINTS = {10, 8, 6, 4, 2, 1};
  private static final int[] C2_POINTS = {5, 3, 2, 1};
  private static final int[] C3_POINTS = {2, 1};
  private static final int[] C4_POINTS = {1};

  /**
   * Constructor method that creates a segment for a given stage, segment type and location.
   *
   * @param stage The stage object which this segment is in. The stage cannot be waiting for
↪  results
   *     or be a time-trial stage.
   * @param type The type of segment, can be either SPRINT, C4, C3, C2, C1, or HC.
   * @param location The location of the segment in the stage in kilometers, cannot be longer than
   *     the length of the stage.
   * @throws InvalidLocationException
   * @throws InvalidStageStateException
   * @throws InvalidStageTypeException
   */
  public Segment(Stage stage, SegmentType type, double location)
      throws InvalidLocationException, InvalidStageStateException, InvalidStageTypeException {
    if (location > stage.getLength()) {
      throw new InvalidLocationException("The location is out of bounds of the stage length.");
    }
    if (stage.isWaitingForResults()) {
      throw new InvalidStageStateException("The stage is waiting for results.");
    }
    if (stage.getType().equals(StageType.TT)) {
      throw new InvalidStageTypeException("Time-trial stages cannot contain any segments.");
    }
    this.stage = stage;
    // ID counter represents the highest known ID at the current time to ensure
    // there
    // are no ID collisions.
    this.id = Segment.count++;
    this.type = type;
    this.location = location;
  }

  /** Reset the static segment ID counter. Used for erasing/loading the CyclingPortal. */
  static void resetIdCounter() {
    count = 0;
  }

  /**
   * Method to get the current state of the static ID counter.
   *
   * @return the highest segment ID stored currently.
   */
  static int getIdCounter() {
    return count;
  }

  /**
   * Method that sets the static ID counter to a given value. Used when loading to avoid ID
   * collisions.
   *
   * @param newCount: new value of the static ID counter.
   */
  static void setIdCounter(int newCount) {
    count = newCount;
  }

  /**
   * Method to get the ID of the segment object.
   *
   * @return id: the Segments's unique ID value.
   */
  public int getId() {
    return id;
  }

  /**
   * Method to get the Stage which the segment exists in.
   *
   * @return The stage object.
   */
  public Stage getStage() {
    return stage;
  }

  /**
   * Method to get the location of the segment within the stage.
   *
   * @return the location in kilometers as a double.
   */
  public double getLocation() {
    return location;
  }

  /**
   * Method to register the time which a given rider completed the segment.
   *
   * @param rider The rider which finished the segment.
   * @param finishTime The time which the rider finished the segment.
```

```java
     */
    public void registerResults(Rider rider, LocalTime finishTime) {
      // Create a segment result for the rider.
      SegmentResult result = new SegmentResult(finishTime);
      // Associate the result with the rider in the result HashMap.
      results.put(rider, result);
    }

    /**
     * Method to get a given riders results in this segment.
     *
     * @param rider The rider whose results will be returned.
     * @return The results the rider received in the segment.
     */
    public SegmentResult getRiderResult(Rider rider) {
      // First calculate the segments results, such as riders position and points.
      calculateResults();
      // Then return the results for the requested rider.
      return results.get(rider);
    }

    /**
     * Method to remove a given riders results from the segment.
     *
     * @param rider The rider object whose results should be removed.
     */
    public void removeRiderResults(Rider rider) {
      results.remove(rider);
    }

    /**
     * Private function to sort all the riders who have results registered by their finish time.
     * Useful for getting each riders position.
     *
     * @return All riders who have a registered result sorted by their finish time.
     */
    private List<Rider> sortRiderResults() {
      // convert the hashmap into a set
      return results.entrySet().stream()
          // Sort the set by the finish time of the results
          .sorted(Map.Entry.comparingByValue(SegmentResult.sortByFinishTime))
          // Get the rider element of the set and ignore the results now they have been
          // sorted and convert to a list.
          .map(Map.Entry::getKey)
          .collect(Collectors.toList());
    }

    /** Private method to calculate the results for this segment. */
    private void calculateResults() {
      // First get a list of riders sorted by their finish time.
      List<Rider> riders = sortRiderResults();

      for (int i = 0; i < results.size(); i++) {
        Rider rider = riders.get(i);
        SegmentResult result = results.get(rider);
        int position = i + 1;
        // Position Calculation
        result.setPosition(position); // Set the riders position

        // Points Calculation
        int[] pointsDistribution =
            getPointsDistribution(); // Get the point distribution based on the segment type.
        if (position <= pointsDistribution.length) {
          // Get the riders points based on their position
          int points = pointsDistribution[i];
          if (this.type.equals(SegmentType.SPRINT)) {
            // If the segment is a sprint, set the riders points as sprinters points.
            result.setSprintersPoints(points);
            result.setMountainPoints(0);
          } else {
            // If the segment is not a sprint, set the riders points as mountain points.
            result.setSprintersPoints(0);
            result.setMountainPoints(points);
          }
        } else {
          // If the rider does not finish in a point-awarding position, reward 0 points.
          result.setMountainPoints(0);
          result.setSprintersPoints(0);
        }
      }
    }

    /**
     * Private method to get the point distribution of the segment based on the type of segment.
     *
     * @return an array of integers that represent the points that should be rewarded based on the
     *     segment type.
     */
    private int[] getPointsDistribution() {
      return switch (type) {
        case HC -> HC_POINTS;
        case C1 -> C1_POINTS;
        case C2 -> C2_POINTS;
        case C3 -> C3_POINTS;
        case C4 -> C4_POINTS;
        case SPRINT -> SPRINT_POINTS;
      };
    }
  }
```

9 SegmentResult.java

```java
package cycling;

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

/** This represents a given recorded result in a segment. */
public class SegmentResult implements Serializable {
  private final LocalTime finishTime;
  private int position;
  private int sprintersPoints;
  private int mountainPoints;

```

```java
    // A comparator which sorts SegmentResults based on Elapsed Time in ascending order. The
    // result with the shortest time will come first.
    protected static final Comparator<SegmentResult> sortByFinishTime =
        Comparator.comparing(SegmentResult::getFinishTime);

    /**
     * Constructor for a given result in a stage.
     *
     * @param finishTime The time at which the segment was finished.
     */
    public SegmentResult(LocalTime finishTime) {
      this.finishTime = finishTime;
    }

    /**
     * A method to get the time at which the segment was finished.
     *
     * @return The LocalTime at which the segment was finished.
     */
    public LocalTime getFinishTime() {
      return finishTime;
    }

    /**
     * A method to set the position in a stage.
     *
     * @param position the position of the rider in the stage.
     */
    public void setPosition(int position) {
      this.position = position;
    }

    /**
     * A method to set the mountain points in a stage.
     *
     * @param points the mountain points received in the stage.
     */
    public void setMountainPoints(int points) {
      this.mountainPoints = points;
    }

    /**
     * A method to set the sprinters points in a stage.
     *
     * @param points the sprinters points received in the stage.
     */
    public void setSprintersPoints(int points) {
      this.sprintersPoints = points;
    }

    /**
     * A method to get the mountain points in a stage.
     *
     * @return the mountain points received in the stage
     */
    public int getMountainPoints() {
      return this.mountainPoints;
    }
```

```java

  /**
   * A method to get the sprinters points in a stage.
   *
   * @return the sprinters points received in the stage.
   */
  public int getSprintersPoints() {
    return this.sprintersPoints;
  }
}
```

10 Stage.java

```java
package cycling;

import java.io.Serializable;
import java.time.Duration;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

/** Stage Class. This represents a stage in a race of the CyclingPortal */
public class Stage implements Serializable {
  private final Race race;
  private final String name;
  private final String description;
  private final double length;
  private final LocalDateTime startTime;
  private final StageType type;
  private final int id;
  private static int count = 0;
  private boolean waitingForResults = false;
  private final ArrayList<Segment> segments = new ArrayList<>();

  private final HashMap<Rider, StageResult> results = new HashMap<>();

  // Segment sprinters/mountain points.
  private static final int[] FLAT_POINTS = {50, 30, 20, 18, 16, 14, 12, 10, 8, 7, 6, 5, 4, 3, 2};
  private static final int[] MEDIUM_POINTS = {30, 25, 22, 19, 17, 15, 13, 11, 9, 7, 6, 5, 4, 3, 2};
  private static final int[] HIGH_POINTS = {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
  private static final int[] TT_POINTS = {20, 17, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

  /**
   * Constructor method that sets a Stage up with a race, name, description, length startTime and
   * type.
   *
   * @param race: Race that the Stage is in.
   * @param name: name of the Stage, cannot be null, empty, have more than 30 characters or have
   *     white space.
   * @param description: description of the Stage.
   * @param length: length of the Stage in kilometers, cannot be less than 5km.
   * @param startTime: start time of the Stage.
   * @param type: the type of Stage, can be either FLAT, MEDIUM_MOUNTAIN, HIGH_MOUNTAIN, TT.
   * @throws InvalidNameException Thrown if the name is empty, null, longer than 30 characters or
   *     contains whitespace.
   * @throws InvalidLengthException Thrown if the length is less than 5km.
   */
  public Stage(
      Race race,
      String name,
      String description,
      double length,
      LocalDateTime startTime,
      StageType type)
      throws InvalidNameException, InvalidLengthException {
    if (name == null
        || name.isEmpty()
        || name.length() > 30
        || CyclingPortal.containsWhitespace(name)) {
      throw new InvalidNameException(
          "Stage name cannot be null, empty, have more than 30 characters or have white spaces.");
    }
    if (length < 5) {
      throw new InvalidLengthException("Length is invalid, cannot be less than 5km.");
    }
    this.name = name;
    this.description = description;
    this.race = race;
    this.length = length;
    this.startTime = startTime;
    this.type = type;
    // ID counter represents the highest known ID at the current time to ensure there
    // are no ID collisions.
    this.id = Stage.count++;
  }

  /** Method that resets the static ID counter of the Race. Used for erasing and loading. */
  static void resetIdCounter() {
    count = 0;
  }

  /**
   * Method to get the current state of the static ID counter.
   *
   * @return the highest race ID stored currently.
   */
  static int getIdCounter() {
    return count;
  }

  /**
   * Method that sets the static ID counter to a given value. Used when loading to avoid ID
   * collisions.
   *
   * @param newCount: new value of the static ID counter.
   */
  static void setIdCounter(int newCount) {
    count = newCount;
  }

  /**
```

```
   * Method to get the ID of the Race object.
   *
   * @return id: the Race's unique ID value.
   */
  public int getId() {
    return id;
  }

  /**
   * Method to get the name of the Stage.
   *
   * @return name: the given name of the Stage.
   */
  public String getName() {
    return name;
  }

  /**
   * Method to get the length of the Stage.
   *
   * @return length: the given length of the Stage.
   */
  public double getLength() {
    return length;
  }

  /**
   * Method to get the Stage's Race.
   *
   * @return race: the given Race that the Stage is in.
   */
  public Race getRace() {
    return race;
  }

  /**
   * Method to get the Stage's type.
   *
   * @return type: the given type of the Stage
   */
  public StageType getType() {
    return type;
  }

  /**
   * Method to get the Segments in the Stage.
   *
   * @return segments: a list of Segments in the Stage.
   */
  public ArrayList<Segment> getSegments() {
    return segments;
  }

  /**
   * Method to get the start time of the Stage.
   *
   * @return startTime: the given start time of the Stage.
   */
```

```java
  public LocalDateTime getStartTime() {
    return startTime;
  }

  /**
   * Method that adds a Segment to the Stage. It is added to the list of Segments based on its
   * location in the Stage.
   *
   * @param segment: Segment that will be added to the Stage.
   */
  public void addSegment(Segment segment) {
    // Loops through the ordered list of segments to find the correct place for the new
    // Segment to be added.
    for (int i = 0; i < segments.size(); i++) {
      // Compares the Segments based on their locations.
      // The new Segment is inserted if its location is less than the location of the
      // current Segment it is being compared to.
      if (segment.getLocation() < segments.get(i).getLocation()) {
        segments.add(i, segment);
        return;
      }
    }
    segments.add(segment);
  }

  /**
   * Method that removes a given Segment from the Stage's Segments.
   *
   * @param segment: the Segment intended to be removed.
   * @throws InvalidStageStateException Thrown if the Stage is waiting for results.
   */
  public void removeSegment(Segment segment) throws InvalidStageStateException {
    if (waitingForResults) {
      throw new InvalidStageStateException(
          "The segment cannot be removed as it is waiting for results.");
    }
    segments.remove(segment);
  }

  /**
   * Method that registers a Rider's result and adds it to the Stage.
   *
   * @param rider: the Rider whose results will be registered.
   * @param checkpoints: the Rider's results.
   * @throws InvalidStageStateException Thrown if the Stage is not waiting for results.
   * @throws DuplicatedResultException Thrown if the Rider already has results registered in the
   *     Stage.
   * @throws InvalidCheckpointsException Thrown if the number checkpoints doesn't equal the number
   *     of Segments in the Stage + 2
   */
  public void registerResult(Rider rider, LocalTime[] checkpoints)
      throws InvalidStageStateException, DuplicatedResultException, InvalidCheckpointsException {
    if (!waitingForResults) {
      throw new InvalidStageStateException(
          "Results can only be added to a stage while it is waiting for results.");
    }
    if (results.containsKey(rider)) {
      throw new DuplicatedResultException("Each rider can only have one result per Stage.");
    }
    if (checkpoints.length != segments.size() + 2) {
      throw new InvalidCheckpointsException(
          "The length of the checkpoint must equal the number of Segments in the Stage + 2.");
    }

    StageResult result = new StageResult(checkpoints);
    // Save Riders result for the Stage
    results.put(rider, result);

    // Propagate all the Riders results for each segment
    for (int i = 0; i < segments.size(); i++) {
      segments.get(i).registerResults(rider, checkpoints[i + 1]);
    }
  }

  /**
   * Method that concludes the Stage preparation and ensures that the Stage is now waiting for
   * results.
   *
   * @throws InvalidStageStateException Thrown if the Stage is already waiting for results.
   */
  public void concludePreparation() throws InvalidStageStateException {
    if (waitingForResults) {
      throw new InvalidStageStateException("Stage is already waiting for results.");
    }
    waitingForResults = true;
  }

  /**
   * Method to identify whether the Stage is waiting for results.
   *
   * @return A boolean, true if the Stage is waiting for results, false if it is not.
   */
  public boolean isWaitingForResults() {
    return waitingForResults;
  }

  /**
   * Method to calculate and return the results of a given Rider.
   *
   * @param rider: Rider whose results are desired.
   * @return results of the Rider.
   */
  public StageResult getRiderResult(Rider rider) {
    calculateResults();
    return results.get(rider);
  }

  /**
   * Method to remove the results of a Rider.
   *
   * @param rider whose results are to be removed.
   */
  public void removeRiderResults(Rider rider) {
    results.remove(rider);
  }

  /**
   * Method to get a list of Riders sorted by their Elapsed Time in the stage.
   *
   * @return List of Riders sorted by their Elapsed Time in the stage.
   */
  public List<Rider> getRidersByElapsedTime() {
    calculateResults();
    return sortRiderResults();
  }

  /**
   * Method to get the HashMap of Riders and their associated Results in the stage.
   *
   * @return The HashMap of Riders and their associated Results in the stage.
   */
  public HashMap<Rider, StageResult> getStageResults() {
    calculateResults();
    return results;
  }

  /**
   * Sort all the riders with a registered result in the stage by their elapsed time.
   *
   * @return A list of riders sorted in ascending order of their elapsed time in the stage.
   */
  private List<Rider> sortRiderResults() {
    return results.entrySet().stream()
        .sorted(Map.Entry.comparingByValue(StageResult.sortByElapsedTime))
        .map(Map.Entry::getKey)
        .collect(Collectors.toList());
  }

  /** A private method to calculate all riders results in the stage. */
  private void calculateResults() {
    // Get a list of all riders with registered results sorted by their elapsed time.
    List<Rider> riders = sortRiderResults();

    for (int i = 0; i < results.size(); i++) {
      Rider rider = riders.get(i);
      StageResult result = results.get(rider);
      int position = i + 1;

      // Position Calculation
      result.setPosition(position); // Assign the rider their position.

      // Adjusted Elapsed Time Calculations
      if (i == 0) {
        // If the rider is the first in the race then their adjusted time = elapsed time.
        result.setAdjustedElapsedTime(result.getElapsedTime());
      } else {
        // Get the previous riders & current riders times.
        Rider prevRider = riders.get(i - 1);
        Duration prevTime = results.get(prevRider).getElapsedTime();
        Duration time = results.get(rider).getElapsedTime();

        // If the difference between the current riders time and the previous time is less than 1
        // second.
        int timeDiff = time.minus(prevTime).compareTo(Duration.ofSeconds(1));
        if (timeDiff <= 0) {
          // Close Finish Condition
          // Set the current riders adjusted time to be the same as the previous riders adjusted
          // time.
          Duration prevAdjustedTime = results.get(prevRider).getAdjustedElapsedTime();
          result.setAdjustedElapsedTime(prevAdjustedTime);
        } else {
          // Far Finish Condition
          // Set the current riders adjusted time = elapsed time.
          result.setAdjustedElapsedTime(time);
        }
      }

      // Points Calculation
      int sprintersPoints = 0;
      int mountainPoints = 0;
      for (Segment segment : segments) {
        // Sum the riders points in each segment.
        SegmentResult segmentResult = segment.getRiderResult(rider);
        sprintersPoints += segmentResult.getSprintersPoints();
        mountainPoints += segmentResult.getMountainPoints();
      }
      int[] pointsDistribution = getPointDistribution();
      if (position <= pointsDistribution.length) {
        // Add any sprinters points the rider may have gained for the stage.
        sprintersPoints += pointsDistribution[i];
      }
      result.setSprintersPoints(sprintersPoints);
      result.setMountainPoints(mountainPoints);
    }
  }

  /**
   * Private method to get the point distribution based on the stage type.
   *
   * @return the distribution of points based on the stage type.
   */
  private int[] getPointDistribution() {
    return switch (type) {
      case FLAT -> FLAT_POINTS;
      case MEDIUM_MOUNTAIN -> MEDIUM_POINTS;
      case HIGH_MOUNTAIN -> HIGH_POINTS;
      case TT -> TT_POINTS;
    };
  }
}
```

11 StageResult.java

```
package cycling;

import java.io.Serializable;
import java.time.Duration;
import java.time.LocalTime;
import java.util.Comparator;

/** This represents a given recorded result in a stage. */
```

```java
public class StageResult implements Serializable {
  private final LocalTime[] checkpoints;
  private final Duration elapsedTime;
  private Duration adjustedElapsedTime;
  private int position;
  private int sprintersPoints;
  private int mountainPoints;

  // A comparator which sorts StageResults based on Elapsed Time in ascending order. The
  // result with the shortest time will come first.
  protected static final Comparator<StageResult> sortByElapsedTime =
      Comparator.comparing(StageResult::getElapsedTime);

  /**
   * Constructor for a given results in a stage.
   *
   * @param checkpoints The array of LocalTimes at which each checkpoint was crossed/
   */
  public StageResult(LocalTime[] checkpoints) {
    this.checkpoints = checkpoints;
    this.elapsedTime = Duration.between(checkpoints[0], checkpoints[checkpoints.length - 1]);
  }

  /**
   * A method to get the times at which each checkpoint was crossed.
   *
   * @return The array of LocalTimes at which each checkpoint was crossed.
   */
  public LocalTime[] getCheckpoints() {
    return this.checkpoints;
  }

  /**
   * A method to get the elapsed time since the start of the stage.
   *
   * @return The duration of time since the stage started.
   */
  public Duration getElapsedTime() {
    return elapsedTime;
  }

  /**
   * A method to set the position in a stage.
   *
   * @param position the position of the rider in the stage.
   */
  public void setPosition(int position) {
    this.position = position;
  }

  /**
   * A method to set the adjusted elapsed time in a stage.
   *
   * @param adjustedElapsedTime the adjusted elapsed time in the stage.
   */
  public void setAdjustedElapsedTime(Duration adjustedElapsedTime) {
    this.adjustedElapsedTime = adjustedElapsedTime;
  }

  /**
   * A method to get the adjusted elapsed time in a stage as a duration.
   *
   * @return the adjusted elapsed time as a duration.
   */
  public Duration getAdjustedElapsedTime() {
    return adjustedElapsedTime;
  }

  /**
   * A method to get the adjusted elapsed time in a stage as a duration.
   *
   * @return the adjusted elapsed time as a duration.
   */
  public LocalTime getAdjustedElapsedLocalTime() {
    return checkpoints[0].plus(adjustedElapsedTime);
  }

  /**
   * A method to set the mountain points in a stage.
   *
   * @param points the mountain points received in the stage.
   */
  public void setMountainPoints(int points) {
    this.mountainPoints = points;
  }

  /**
   * A method to set the sprinters points in a stage.
   *
   * @param points the sprinters points received in the stage.
   */
  public void setSprintersPoints(int points) {
    this.sprintersPoints = points;
  }

  /**
   * A method to get the mountain points in a stage.
   *
   * @return the mountain points received in the stage
   */
  public int getMountainPoints() {
    return mountainPoints;
  }

  /**
   * A method to get the sprinters points in a stage.
   *
   * @return the sprinters points received in the stage.
   */
  public int getSprintersPoints() {
    return sprintersPoints;
  }
}
```

12 Team.java

```java
package cycling;

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

/** Team class. This represents a team of riders. */
public class Team implements Serializable {
  private final String name;
  private final String description;

  private final ArrayList<Rider> riders = new ArrayList<>();
  private static int count = 0;
  private final int id;

  /**
   * Constructor method that sets up the Team with a name and a description.
   *
   * @param name of the team.
   * @param description of the team.
   * @throws InvalidNameException Thrown if the team name is null, empty, has more than 30
   *     characters or contains any whitespace.
   */
  public Team(String name, String description) throws InvalidNameException {
    if (name == null
        || name.isEmpty()
        || name.length() > 30
        || CyclingPortal.containsWhitespace(name)) {
      throw new InvalidNameException(
          "Team name cannot be null, empty, have more than 30 characters or have white spaces.");
    }
    this.name = name;
    this.description = description;
    this.id = Team.count++;
  }

  /** Method to reset the static ID counter. */
  static void resetIdCounter() {
    count = 0;
  }

  /**
   * Method to get the current state of the static ID counter.
   *
   * @return the highest race ID stored currently.
   */
  static int getIdCounter() {
    return count;
  }

  /**
   * Method that sets the static ID counter to a given value. Used when loading to avoid ID
   * collisions.
   *
   * @param newCount: new value of the static ID counter.
   */
  static void setIdCounter(int newCount) {
    count = newCount;
  }

  /**
   * Method that gets the name of the Team.
   *
   * @return name of the Team.
   */
  public String getName() {
    return name;
  }

  /**
   * Method that gets the ID of the Team.
   *
   * @return ID of the Team.
   */
  public int getId() {
    return id;
  }

  /**
   * Method that removes a Rider from the Team.
   *
   * @param rider to be removed.
   */
  public void removeRider(Rider rider) {
    riders.remove(rider);
  }

  /**
   * Method to get the Riders in the Team.
   *
   * @return A list of Riders in the Team.
   */
  public ArrayList<Rider> getRiders() {
    return riders;
  }

  /**
   * Method that adds a Rider to the Team.
   *
   * @param rider to be added to the Team.
   */
  public void addRider(Rider rider) {
    riders.add(rider);
  }
}
```