1 Result.java

```java
package cycling;

import java.time.LocalTime;
import java.util.*;

/**
 * Result class used to store the results of riders as objects, most functions to return a value related in some way to checkpoint are handled here or in race
 * @author Nathan
 *
 */
public class Result {
   int stageID, riderID;
   public static ArrayList<Result> allResultList = new ArrayList<Result>();
   LocalTime checkpoints[];
   public Result(int stageId, int riderId, LocalTime... checkpoints) throws IDNotRecognisedException, DuplicatedResultException, InvalidCheckpointsException,
   InvalidStageStateException{ /* registerRiderResultsInStage */
      if (Stage.stageIdExists(stageId) == false) {
         throw new IDNotRecognisedException("Stage id does not exists in the system"); /* ID not recognised check for stages */
      }
      else if (Rider.riderIdExists(riderId) == false) {
         throw new IDNotRecognisedException("Rider id does not exists in the system"); /* ID not recognised check for riders */
      }
      else if (resultExists(stageId, riderId) == true) {
         throw new DuplicatedResultException("Result for this rider and stage already registered"); /* Duplicate result check */
      }
      else if (Stage.getStageObj(stageId).getConcluded() == false) {
         throw new InvalidStageStateException("Stage has not been concluded and results cannot be stored for it until done so"); /* stage state check */
      }
      this.stageID = stageId;
      this.riderID = riderId;
      this.checkpoints = checkpoints;
      allResultList.add(this);
   }

   public static LocalTime[] getRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException {
      LocalTime[] result;
      try {
         result = getResultObj(stageId, riderId, allResultList).getRiderResults();
      }
      catch (NullPointerException exception) { /* exception handling for if the array is empty */
         result = new LocalTime[]{};
      }

      return result;
   }
```

```
46
47      public static int[] getRidersPointsInStage(int stageId) throws IDNotRecognisedException {
48          Stage.calculatePoints(stageId);
49          ArrayList<Integer> ridersPoints = new ArrayList<Integer>();
50          int riderPointArr[];
51          Stage temp;
52          temp = Stage.getStageObj(stageId);
53          Hashtable<Integer, Integer> tempTable = temp.getRiderPointsDictionary(); /* gets Hashtable that stores
                the stages rider points */
54
55          Set<Integer> intKeys = tempTable.keySet();
56          Set<String> keys = new HashSet<String>(tempTable.size());
57          for (Integer integer : intKeys) {
58              keys.add(integer.toString()); /* converts integer keys to strings allowing for iteration */
59          }
60
61          for(String key: keys){
62              ridersPoints.add(tempTable.get(Integer.parseInt(key))); /* iterates through and adds keys to
                    riderpoints ArrayList */
63          }
64          riderPointArr = new int[ridersPoints.size()];
65          for(int i = 0; i < ridersPoints.size(); i ++) {
66              riderPointArr[i] = ridersPoints.get(i); /* stores values within ArrayList in correct return format
                    */
67          }
68          return riderPointArr;
69      }
70
71      public static int[] getRidersMountainPointsInStage(int stageId) throws IDNotRecognisedException {
72          Stage temp;
73          temp = Stage.getStageObj(stageId);
74          Stage.calculateMountainPoints(stageId);
75          int riderPointArr[];
76          ArrayList<Integer> ridersMountainPoints = new ArrayList<Integer>();
77          Hashtable<Integer, Integer> tempTable = temp.getRiderMountainPointsDictionary();
78          Set<Integer> intKeys = tempTable.keySet();
79          Set<String> keys = new HashSet<String>(tempTable.size());
80          for (Integer integer : intKeys) {
81              keys.add(integer.toString());
82          }
83
84          for(String key: keys){
85              ridersMountainPoints.add(tempTable.get(Integer.parseInt(key)));
86          }
87          riderPointArr = new int[ridersMountainPoints.size()];
88          for(int i = 0; i < ridersMountainPoints.size(); i ++) {
89              riderPointArr[i] = ridersMountainPoints.get(i);
90          }
91          return riderPointArr; /*will be returned int the order of the ranks of the riders for the first
                mountain checkpoint */
92      }
93
94      public static LocalTime getRiderAdjustedElapsedTimeInStage(int stageId, int riderId) throws
            IDNotRecognisedException {
95          ArrayList<Result> allRiderResults= getStageResultList(stageId);
```

```
96          LocalTime[] riderTimes = getRiderResultsInStage(stageId, riderId);
97          LocalTime nullExcep = null;
98
99          if(riderTimes.length == 0) {
100            return nullExcep;  /* returns nothing in the event the riderID does exist however is not value for
                   this stage */
101         }
102
103         LocalTime riderFinish = riderTimes[riderTimes.length - 1], finTimeUppBound =
                riderFinish.plusSeconds(1);
104         allRiderResults.sort(new ResultComparator()); /* sorts all objects within ArrayList by finishing time
                */
105
106         for (int i = 0; i < allRiderResults.size(); i++) { /* only one loop required as list is sorted due to
                comparator */
107           if (allRiderResults.get(i).getRiderResults()[riderTimes.length - 1].isAfter(riderFinish)
108                && allRiderResults.get(i).getRiderResults()[riderTimes.length - 1].isBefore(finTimeUppBound)){
                       /* if next rider time is after passed in riders time but still within a second of it */
109
110             riderFinish = allRiderResults.get(i).getRiderResults()[riderTimes.length - 1];
111             finTimeUppBound = riderFinish.plusSeconds(1);
112           }
113         }
114         return riderFinish;
115     }
116
117     public static void deleteRiderResultsInStage(int stageId, int riderId) throws IDNotRecognisedException {
118         int index = allResultList.indexOf(getResultObj(stageId, riderId, allResultList));
119         if (index == -1) {
120           throw new IDNotRecognisedException("Rider cannot be deleted, already does not exist for this
                  stage");
121         }
122         else {
123           allResultList.remove(index); /* removes the rider at the specific index */
124         }
125     }
126
127     public static int[] getRidersRankInStage(int stageId) throws IDNotRecognisedException {
128         ArrayList<Result> listOfRiderResults = getStageResultList(stageId); /*gets list of all stage results */
129         int[] riderIDs = new int[listOfRiderResults.size()];
130
131         listOfRiderResults.sort(new ResultComparator()); /* uses custom comparator to sort list by finishing
                result */
132
133         for(int i = 0; i < listOfRiderResults.size(); i++) {
134           riderIDs[i] = listOfRiderResults.get(i).getRiderID(); /* puts sorted results into correct output
                  format */
135         }
136         return riderIDs;
137     }
138     public static LocalTime[] getRankedAdjustedElapsedTimesInStage(int stageId) throws
            IDNotRecognisedException {
139         ArrayList<Result> relevantStageResults = getAdjustedResultObject(stageId);
140         LocalTime[] adjustedResults = new LocalTime[relevantStageResults.size()];
141         int length = relevantStageResults.get(0).getRiderResults().length - 1;
```

```
142
143         for(int i = 0; i < relevantStageResults.size(); i++) {
144             adjustedResults[i] = relevantStageResults.get(i).getRiderResults()[length]; /* returns all adjusted
                    final results for the riders in the stage */
145         }
146         return adjustedResults;
147     }
148     /**
149      * Used to find a specified stages finishing times and sort them in ascending order
150      * @param stageId ID of stage to be searched
151      * @return ArrayList of sorted finishing times
152      * @throws IDNotRecognisedException
153      * @author Nathan
154      */
155     public static ArrayList<Result> getAdjustedResultObject (int stageId) throws IDNotRecognisedException{
156         ArrayList<Result> relevantStageResults = getStageResultList(stageId);
157         int length = relevantStageResults.get(0).getRiderResults().length - 1;
158
159         for (int t = 0; t < relevantStageResults.size(); t++) {  /* takes a value and compares it to all other
                values */
160             for (int i = 0; i < relevantStageResults.size(); i++){
161                 LocalTime time1 = relevantStageResults.get(t).getRiderResults()[length];
162                 LocalTime time2 = relevantStageResults.get(i).getRiderResults()[length];
163                 if(time2.isAfter(time1) && time2.isBefore(time1.plusSeconds(1))) { /* check to ensure that the
                        time is within 1 second after */
164                     relevantStageResults.get(t).getRiderResults()[length] =
                            relevantStageResults.get(i).getRiderResults()[length]; /* in event above statement is */
165                 }                                                                  /*  true adjusts
                        the time */
166             }
167         }
168         relevantStageResults.sort(new ResultComparator()); /* sorts results using result comparator */
169         return relevantStageResults;
170     }
171     /** Rider Checkpoints getter
172      * @return LocalTime[] */
173     public LocalTime[] getRiderResults() { /* rider results getter */
174         return this.checkpoints;
175     }
176     /** Stage ID getter */
177     public int getStageID() { /* stage ID getter */
178         return this.stageID;
179     }
180     /** Rider ID getter */
181     public int getRiderID() { /* rider ID getter */
182         return this.riderID;
183     }
184     /**
185      * Method to assist with Duplicate Result exception
186      * @param stageId Relevant stage ID
187      * @param riderID Relevant rider ID
188      * @return True if result already exists / false if not
189      * @author Nathan
190      */
191     public boolean resultExists(int stageId, int riderID) {
```

```
192       boolean exists = false;
193       for(int i = 0; i <allResultList.size(); i++) {
194          if (allResultList.get(i).getStageID() == stageId) {
195             if (allResultList.get(i).getRiderID() == riderID) {
196                exists = true;
197             }
198          }
199       }
200       return exists;
201    }
202    /**
203     * Returns the requested object based on their stage and rider ID
204     * @param stageId Id of the stage
205     * @param riderId Id of the rider
206     * @param specifiedList ArrayList to be searched to find the required object
207     * @return The requested result object within the specified list
208     * @throws IDNotRecognisedException
209     * @author Nathan
210     */
211    public static Result getResultObj(int stageId, int riderId, ArrayList<Result> specifiedList) throws
           IDNotRecognisedException{
212       Result obj = null;
213       boolean validForStage = true, stageExists = false, riderExists = false;
214       for(int i = 0; i <specifiedList.size(); i++) {
215          if (specifiedList.get(i).getStageID() == stageId) {
216             stageExists = true;
217             if (specifiedList.get(i).getRiderID() == riderId) {
218                obj = specifiedList.get(i); /* returns the correct result in the event both the stage and
                       rider id contain a related result */
219             }
220          }
221       }
222       if (obj == null) {
223          if (stageExists == true){
224             for(int y = 0; y <specifiedList.size(); y++) {
225                if (specifiedList.get(y).getRiderID() == riderId) { /*check to see if the rider id exists just
                       not for this particular stage */
226                   validForStage = false;
227                   riderExists = true;
228                   break;
229                }
230             }
231             if (validForStage == false && riderExists == true) { /* in the event both exist however the
                    rider does not exist for this stage */
232                return null;
233             }
234             else {
235                throw new IDNotRecognisedException("ID not recognised"); /* in the event stage exists but
                       rider does not */
236             }
237          }
238          else {
239             throw new IDNotRecognisedException("ID not recognised"); /* in the event stage doesnt exist */
240          }
241       }
```

```
242         return obj;
243     }
244     /** Loops through the result ArrayList and puts all the results of the related stage into its own
            ArrayList.
245      *
246      * @param stageId The ID of the stage you require the results of
247      * @return ArrayList of the required results
248      * @author Nathan
249      */
250     public static ArrayList<Result> getStageResultList(int stageId) throws IDNotRecognisedException {
251         ArrayList<Result> relevantResults = new ArrayList<Result>();
252         for (int i = 0; i < allResultList.size(); i++) {
253             if (allResultList.get(i).getStageID() == stageId) {
254                 relevantResults.add(allResultList.get(i)); /* stores list of related stage results */
255             }
256         }
257         if (relevantResults.isEmpty()) {
258             throw new IDNotRecognisedException("ID not recognised exception");
259         }
260         return relevantResults;
261     }
262     /**
263      * Comparator used to compare the final index of the checkpoint array
264      * @return Values sorted based on their finishing times
265      * @author Nathan
266      *
267      */
268     static class ResultComparator implements Comparator<Result> {
269         public int compare(Result result1, Result result2) {
270             return result1.getRiderResults()[result1.getRiderResults().length - 1]
271                 .compareTo(result2.getRiderResults()[result1.getRiderResults().length - 1]);
272         }
273      }
274     /** Custom comparator used to compare checkpoint variables
275      * @return list of checkpoint LocalTimes sorted based on their times
276      * @author Sam
277      *
278      */
279     static class CheckpointComparator implements Comparator <LocalTime> {
280         public int compare(LocalTime checkpoint1, LocalTime checkpoint2) {
281             return checkpoint1.compareTo(checkpoint2); /* compares two passed in localtime checkpoints */
282         }
283     }
284 }
```

2 Race.java

```
1  package cycling;
2
3  import java.util.*;
4  import java.io.FileOutputStream;
5  import java.io.IOException;
6  import java.io.ObjectOutputStream;
7  import java.time.Duration;
```

```
8  import java.time.LocalDateTime;
9  import java.time.LocalTime;
10
11 /**
12  * Race class used to store the details, description, raceID and stage list within race objects that are
       stored in a larger ArrayList of races
13  * @author Nathan
14  *
15  */
16 public class Race {
17    String name, description;
18     int raceID;
19     public static ArrayList<Race> raceList = new ArrayList<Race>();
20     public ArrayList<Stage> raceStageList = new ArrayList<Stage>();
21
22     public Race(String name, String description) throws IllegalNameException, InvalidNameException{
23
24         int temp;
25         this.name = name;
26         this.description = description;
27
28         for (int i = 0; i < raceList.size(); i++) { /* check for Illegal name exception */
29          if (raceList.get(i).getRaceName() == name){
30             throw new IllegalNameException("This name is currently being used by another race");
31          }
32       }
33         if (name.isEmpty() || name.length() >= 256 || name.contains(" ")) { /* check for invalid name
               exception */
34          throw new InvalidNameException("Name cannot be empty, include whitespaces, "
35                + "or be larger than the system limit of characters");
36         }
37
38         temp = raceList.size();
39         if (temp > 0){
40               this.raceID = raceList.get(temp - 1).raceID + 1; /* generates raceID */
41         }
42         else {
43               this.raceID = 0;
44         }
45         raceList.add(this); /* adds new race object to list of race objects */
46     }
47     public static int[] getRaceIDs() {
48       int[] raceIDs = new int[]{};
49       if (raceList.isEmpty()) {
50          return raceIDs;
51       }
52       else {
53          for(int i=0; i <raceList.size(); i++) {
54             raceIDs[i] = raceList.get(i).getRaceID(); /* loops through and returns race IDs */
55          }
56       }
57       return null;
58     }
59     public static void removeRace(int raceId) throws IDNotRecognisedException{
60       if (raceIdExists(raceId) == false) {
```

```
61          throw new IDNotRecognisedException("Race id does not exists in the system");
62       }
63       else {
64          Race temp;
65             temp = getRaceObj(raceId);
66             for (int i = 0; i < temp.raceStageList.size(); i++) {
67                 Stage.removeStageById(temp.raceStageList.get(i).getStageID()); /* removes race based on the
                       corresponding id */
68             }
69             raceList.remove(raceList.indexOf(temp));
70       }
71    }
72
73    public static int addStageToRace(int raceId, String stageName, String description, double length,
          LocalDateTime startTime, StageType type)
74          throws IDNotRecognisedException, IllegalNameException, InvalidNameException, InvalidLengthException{
75       if (raceIdExists(raceId) == false) {
76          throw new IDNotRecognisedException("Race id does not exists in the system"); /* check for id not
              recognised exception */
77       }
78       else {
79          for (int i = 0; i < Stage.stageList.size(); i++) {
80             if (Stage.stageList.get(i).getStageName() == stageName){
81                throw new IllegalNameException("This name is currently being used by another stage"); /* check
                      for illegal name exception */
82             }
83          }
84          if (stageName.isEmpty() || stageName.length() >= 256 || stageName.contains(" ")) {
85             throw new InvalidNameException("Name cannot be empty, include whitespaces, " /* check for
                   invalid name exception */
86                 + "or be larger than the system limit of characters");
87          }
88          else if (length < 5 || length == 0) {
89             throw new InvalidLengthException("Length cannot be null or less than 5km"); /* check for invalid
                   length exception */
90          }
91          Stage stageObj = new Stage(raceId, stageName, description, length, startTime, type); /* if all
                checks are passed creates new object and adds to both relevant arraylists */
92          for(int i=0; i < raceList.size(); i++) {
93             if (raceList.get(i).getRaceID() == raceId) {
94                raceList.get(i).setRaceStageList(stageObj);
95             }
96          }
97          return stageObj.getStageID();
98       }
99    }
100   public static int[] getRaceStages(int raceId) throws IDNotRecognisedException {
101      if (raceIdExists(raceId) == false) {
102         throw new IDNotRecognisedException("Race id does not exists in the system");
103      }
104      else {
105         Race race = getRaceObj(raceId);
106         int[] raceStages = new int[race.raceStageList.size()];
107         for (int i = 0; i < race.raceStageList.size(); i++) {
108            if (race.raceStageList.get(i).getRaceID() == raceId) {
```

```
109             raceStages[i] = race.raceStageList.get(i).getStageID(); /* returns the races stages in the
                    event the raceId is correct */
110           }
111         }
112           return raceStages;
113       }
114     }
115
116     public static int getNumberOfStages(int raceId) throws IDNotRecognisedException{
117       if (raceIdExists(raceId) == false) {
118         throw new IDNotRecognisedException("Race id does not exists in the system");
119       }
120       else {
121         Race temp;
122         temp = getRaceObj(raceId);
123           int number = temp.raceStageList.size(); /* returns number of stages if raceId is correct */
124           return number;
125       }
126     }
127     public static String viewRaceDetails(int raceId) throws IDNotRecognisedException{
128       if (raceIdExists(raceId) == false) {
129         throw new IDNotRecognisedException("Race id does not exists in the system");
130       }
131       else {
132         Race temp;
133         temp = getRaceObj(raceId);
134         int raceLen = 0;
135         for (int i = 0; i < temp.raceStageList.size(); i++) {
136           raceLen += temp.raceStageList.get(i).getLength();
137         }
138         return (temp.getRaceID() + " , " + temp.getRaceName() + " , " + temp.raceStageList.size() + " , " +
                raceLen); /* returns list of races details, relatively self explanatory */
139       }
140     }
141     /**
142      * Checks for if a raceID is current in use (exists) or if it isn't.
143      * @param raceId ID of race to check
144      * @return boolean true / false
145      */
146     public static boolean raceIdExists(int raceId) {
147       boolean exists = false;
148       for(int i=0; i <raceList.size(); i++) {
149         if (raceList.get(i).getRaceID() == raceId) { /* check performed usually used in checks for invalid
                name exception to return if the result exists or not */
150           exists = true;
151           break;
152         }
153       }
154       if (exists == true) {
155         return true;
156       }
157       else {
158         return false;
159       }
160     }
```

```
161     /**
162      * Adds the new stage object to the ArrayList of stage objects currently stored for this race object
163      * @param passedObj Passed in stage object
164      * @author Nathan
165      */
166     public void setRaceStageList(Stage passedObj) {
167       raceStageList.add(passedObj);
168     }
169     /**
170      * Race ID getter
171      * @return raceID
172      * @author Nathan
173      */
174     public int getRaceID() {
175       return this.raceID;
176     }
177     /**
178      * Race name getter
179      * @return Race name
180      * @author Nathan
181      */
182     public String getRaceName() {
183       return this.name;
184     }
185     /**
186      * Race description getter
187      * @return Race description
188      * @author Nathan
189      */
190     public String getRaceDescription() {
191       return this.description;
192     }
193     /**
194      * Returns the race object that stores the specific ID
195      * @param passedID ID of race object you wish to obtain
196      * @return Race object
197      * @author Nathan
198      */
199     public static Race getRaceObj(int passedID){
200         for (int i = 0; i < raceList.size(); i++) {
201             if (raceList.get(i).getRaceID() == passedID) { /* returns the object with the corresponding
                    stored raceId */
202                 return raceList.get(i);
203             }
204         }
205         return null;
206     }
207     public static LocalTime[] getGeneralClassificationTimesInRace(int raceId) throws
            IDNotRecognisedException {
208       if (raceIdExists(raceId) == false) {
209         throw new IDNotRecognisedException("Race id does not exists in the system");
210       }
211
212       key.keyList.clear(); /* clears the list of keys to help java garbage collection remove previously used
              key objects */
```

```
213       List<LocalTime> valuesList = new ArrayList<LocalTime>();
214       getAdjustedTimesAndId(raceId);
215       for (int i = 0; i < key.keyList.size(); i++) {
216          valuesList.add(key.keyList.get(i).getTime()); /* returns the time stored within the keys */
217       }
218       LocalTime[] p = new LocalTime[valuesList.size()];
219       valuesList.sort(new TimeComparator()); /* sorts list of times */
220       for (int x = 0; x < valuesList.size(); x++) {
221          p[x] = valuesList.get(x); /* stores values in relevant returned datatype */
222       }
223       return p;
224    }
225    public static int[] getRidersGeneralClassificationRank(int raceId) throws IDNotRecognisedException {
226       if (raceIdExists(raceId) == false) {
227          throw new IDNotRecognisedException("Race id does not exists in the system");
228       }
229       List<LocalTime> timeList = new ArrayList<LocalTime>();
230       for (int i = 0; i < key.keyList.size(); i++) {
231          timeList.add(key.keyList.get(i).getTime());
232       }
233       LocalTime[] riderTimeStore = getGeneralClassificationTimesInRace(raceId);
234       int[] riderIdStore = new int[timeList.size()];
235       for (int x = 0; x < key.keyList.size(); x++) {
236          for (int y = 0; y < key.keyList.size(); y++) {
237             if (riderTimeStore[x] == key.keyList.get(y).getTime()){ /* very similar to above function except
                    returns corresponding ids in order of times */
238                riderIdStore[x] = key.keyList.get(y).getID();
239             }
240          }
241       }
242       return riderIdStore;
243    }
244    /**
245     * Creates a set of new key objects for each individual rider and aggregate time they have for this
            current race
246     * @param raceId ID of race that requires the keys
247     * @throws IDNotRecognisedException
248     * @author Nathan
249     */
250    public static void getAdjustedTimesAndId (int raceId) throws IDNotRecognisedException {
251       Race temp = getRaceObj(raceId);
252       boolean exists = false;
253       LocalTime[] currentRiderTimes; /* current checkpoint array being used within the loop */
254       LocalTime stageDuration; /* time between the first and the last element in the checkpoint array */
255       int includedStages[] = new int[temp.raceStageList.size()]; /* list of stage ids corresponding to the
            race */
256       int tempId, length;
257       for (int i = 0; i < temp.raceStageList.size(); i++) {
258          includedStages[i] = temp.raceStageList.get(i).getStageID();
259       }
260
261       ArrayList<Result> relevantStageResults = new ArrayList<Result>();
262       for (int i = 0; i < includedStages.length; i++) {
263          relevantStageResults = Result.getAdjustedResultObject(includedStages[i]); /* assigns the arraylist
               the adjusted times for the current stage */
```

```
264         for (int t = 0; t < relevantStageResults.size(); t++) { /* loop for looping through stage arraylist
                for results */
265            currentRiderTimes = relevantStageResults.get(t).getRiderResults();
266            length = relevantStageResults.get(t).getRiderResults().length - 1;
267            tempId = relevantStageResults.get(t).getRiderID();
268            stageDuration = currentRiderTimes[length].minusNanos(currentRiderTimes[0].toNanoOfDay());
269
270            for (int x = 0; x < key.keyList.size(); x++) {
271               if (key.keyList.get(x).getID() == tempId) {
272                  key.keyList.get(x).setTime(stageDuration); /* checks to see if the key is already present
                        from another stage and aggregates the time in the event it does */
273                  exists = true;
274               }
275            }
276            if (exists == false) { /* if key doesnt already exist creates a new key object */
277               key newKey = new key(stageDuration, tempId);
278               key.keyList.add(newKey);
279               newKey = null;
280            }
281         }
282      }
283   }
284
285
286   /**
287    * Custom comparator that compares a list of aggregate times and returns them ordered in ascending order
288    * @author Nathan
289    *
290    */
291   static class TimeComparator implements Comparator<LocalTime> {
292       public int compare(LocalTime result1, LocalTime result2) {
293           return result1.compareTo(result2); /* compares two passed in localtimes */
294       }
295   }
296   /**
297    * Class used to store the rider ID and time of that specific rider in an object such that they can be
          accessed via one another
298    * @author Nathan
299    *
300    */
301   static class key {
302     LocalTime time;
303     int riderId;
304     /**
305      * ArrayList containing the current list of key objects in use
306      */
307     public static ArrayList<key> keyList = new ArrayList<key>();
308     /**
309      * Creates a new instance of the key object
310      * @param time Finishing time of the specific rider
311      * @param riderId ID of the specified rider
312      *
313      */
314     public key(LocalTime time, int riderId) {
315        this.time = time;
```

```
316         this.riderId = riderId; /* adding to the keyList is done externally typically when the constructor
                is called */
317       }
318       /**
319        * Aggregate time getter
320        * @return the aggregate time of this key object
321        */
322       public LocalTime getTime() {
323          return this.time;
324       }
325       /**
326        * ID getter
327        * @return The Rider ID of this key object
328        */
329       public int getID() {
330          return this.riderId;
331       }
332       /**
333        * Adds time passed in to the aggregate time stored for this key object
334        * @param passed passed in LocalTime value
335        */
336       public void setTime(LocalTime passed) {
337          this.time = this.time.plus(Duration.ofNanos(passed.toNanoOfDay())); /* aggregates the time */
338       }
339    }
340    /**
341     * Removes a race from the list of race objects by name attribute
342     * @param name name of race you wish to remove
343     */
344    public static void removeRaceByName(String name) {
345        for (int i=0; i < raceList.size(); i++) {
346            if (name == raceList.get(i).getRaceName()) {
347                raceList.remove(raceList.get(i).getRaceID());
348            }
349        }
350    }
351    public static int[] getRidersPointsInRace(int raceId) throws IDNotRecognisedException {
352        Hashtable<Integer, Integer> pointsTable = new Hashtable<Integer, Integer>();
353        ArrayList<Integer> ridersPoints = new ArrayList<Integer>();
354        int[] riderPointArr;
355        int[] tempStageList = Race.getRaceStages(raceId);
356        int[] stagePoints = Result.getRidersPointsInStage(tempStageList[0]);
357        int[] tempRiders = Result.getRidersRankInStage(tempStageList[0]);
358        for (int i = 0; i < tempRiders.length; i++) {
359            pointsTable.put(tempRiders[i], stagePoints[i]);
360        }
361        for (int i = 1; i < tempStageList.length; i++) {
362            stagePoints = Result.getRidersPointsInStage(tempStageList[i]);
363            tempRiders = Result.getRidersRankInStage(tempStageList[i]);
364            for (int j = 1; j < stagePoints.length; j++) {
365                pointsTable.put(tempRiders[j], pointsTable.get(tempRiders[j])+stagePoints[i]);
366            }
367        }
368        Set<Integer> keys = pointsTable.keySet();
369        for(int key: keys){
```

```
            ridersPoints.add(pointsTable.get(key));
        }
        riderPointArr = new int[ridersPoints.size()];
        for(int i = 0; i < ridersPoints.size(); i ++) {
         riderPointArr[i] = ridersPoints.get(i);
      }
        return riderPointArr;
    }
    public static int[] getRidersMountainPointsInRace(int raceId) throws IDNotRecognisedException {
        Hashtable<Integer, Integer> pointsTable = new Hashtable<Integer, Integer>();
        ArrayList<Integer> ridersPoints = new ArrayList<Integer>();
        int[] riderPointArr;
        int[] tempStageList = Race.getRaceStages(raceId);
        int[] stagePoints = Result.getRidersMountainPointsInStage(tempStageList[0]);
        int[] tempRiders = Result.getRidersRankInStage(tempStageList[0]);
        for (int i = 0; i < tempRiders.length; i++) {
            pointsTable.put(tempRiders[i], stagePoints[i]);
        }
        for (int i = 1; i < tempStageList.length; i++) {
            stagePoints = Result.getRidersMountainPointsInStage(tempStageList[i]);
            tempRiders = Result.getRidersRankInStage(tempStageList[i]);
            for (int j = 1; j < stagePoints.length; j++) {
                pointsTable.put(tempRiders[j], pointsTable.get(tempRiders[j])+stagePoints[i]);
            }
        }
        Set<Integer> keys = pointsTable.keySet();
        for(int key: keys){
            ridersPoints.add(pointsTable.get(key));
        }
        riderPointArr = new int[ridersPoints.size()];
        for(int i = 0; i < ridersPoints.size(); i ++) {
         riderPointArr[i] = ridersPoints.get(i);
      }
        return riderPointArr;
    }
    public static int[] getRidersPointClassificationRank(int raceId) throws IDNotRecognisedException{
        Race tempRace;
        tempRace = Race.getRaceObj(raceId);
        Hashtable<Integer, Integer> tempTable = new Hashtable<Integer, Integer>();
        int[] riderIds = Result.getRidersRankInStage((Race.getRaceStages(raceId))[0]);
        int[] points = getRidersPointsInRace(raceId);
        for (int i = 0; i < points.length; i++) {
            tempTable.put(points[i], riderIds[i]);
        }
        for (int i = 0; i < points.length; i++) {
            for(int j = i+1; j < points.length; j++) {
                int tempI = points[i];
                int tempJ = points[j];
                if(tempI > tempJ){
                    points[i]= tempJ;
                    points[j]= tempI;
                }
            }
        }
        for (int i = 0; i < points.length; i++) {
```

```
            riderIds[i] = tempTable.get(points[i]);
        }
        return riderIds;
    }
    public static int[] getRidersMountainPointClassificationRank(int raceId) throws
        IDNotRecognisedException{
        Race tempRace;
        tempRace = Race.getRaceObj(raceId);
        Hashtable<Integer, Integer> tempTable = new Hashtable<Integer, Integer>();
        int[] riderIds = Result.getRidersRankInStage((Race.getRaceStages(raceId))[0]);
        int[] points = getRidersMountainPointsInRace(raceId);
        for (int i = 0; i < points.length; i++) {
            tempTable.put(points[i], riderIds[i]);
        }
        for (int i = 0; i < points.length; i++) {
            for(int j = i+1; j < points.length; j++) {
                int tempI = points[i];
                int tempJ = points[j];
                if(tempI > tempJ){
                    points[i]= tempJ;
                    points[j]= tempI;
                }
            }
        }
        for (int i = 0; i < points.length; i++) {
            riderIds[i] = tempTable.get(points[i]);
        }
        return riderIds;
    }
}
```

3 Stage.java

```
package cycling;

import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.*;
/**
 * Class used to store the stages of a race and their relevant information
 * @author Nathan
 *
 */
public class Stage {
   int raceID, stageID;
    String stageName, description;
    double length;
    LocalDateTime startTime;
    StageType type;
    String state = "incomplete";
    public boolean concluded = false;
    /**
     * List of all stages currently stored
     */
```

```
22      public static ArrayList<Stage> stageList = new ArrayList<Stage>();
23      /**
24       * List of all segments for this particular stage
25       */
26      public ArrayList<Segment> stageSegmentList = new ArrayList<Segment>();
27     public Hashtable<Integer, Integer> riderPointsDictionary = new Hashtable<Integer, Integer>();
28     public Hashtable<Integer, Integer> riderMountainPointsDictionary = new Hashtable<Integer, Integer>();
29
30
31      public Stage(int raceID, String stageName, String description, double length, LocalDateTime startTime,
            StageType type){
32        /* exception handling handled in race method that calls this constructor */
33          int temp;
34          this.raceID = raceID;
35          this.stageName = stageName;
36          this.description = description;
37          this.length = length;
38          this.startTime = startTime;
39          this.type = type;
40          temp = stageList.size();
41          if (temp > 0){
42                this.stageID = stageList.get(temp - 1).stageID + 1;
43          }
44          else {
45                this.stageID = 0;
46          }
47          stageList.add(this);
48      }
49      /**
50       * Returns stage ID
51       * @return stageID
52       * @author Nathan
53       */
54      public int getStageID() {
55        return this.stageID;
56     }
57      public static double getStageLength(int stageId) throws IDNotRecognisedException{
58        if (stageIdExists(stageId) == false) {
59           throw new IDNotRecognisedException("Stage id does not exists in the system");
60        }
61        else {
62           Stage temp;
63              temp = getStageObj(stageId);
64              return temp.getLength(); /* returns length of stage */
65        }
66      }
67      /**
68       * Returns true if stage exists, false if it does not
69       * @param stageId ID of stage wished to have its existence verified
70       * @return false / true
71       * @author Nathan
72       */
73      public static boolean stageIdExists(int stageId) {
74        boolean exists = false;
75        for(int i=0; i <stageList.size(); i++) {
```

```
76          if (stageList.get(i).getStageID() == stageId) { /*check for if stage exists */
77             exists = true;
78             break;
79          }
80       }
81       if (exists == true) {
82          return true;
83       }
84       else {
85          return false;
86       }
87    }
88    public static Stage getStageObj(int passedID){
89        for (int i = 0; i < stageList.size(); i++) {
90            if (stageList.get(i).getStageID() == passedID) {
91                return stageList.get(i); /* returns stage object with relevant ID in the event it is found */
92            }
93        }
94        return null;
95    }
96
97    public static void removeStageById(int stageId) throws IDNotRecognisedException{
98       if (stageIdExists(stageId) == false) {
99          throw new IDNotRecognisedException("Stage id does not exists in the system");
100      }
101      else {
102          Stage tempStage;
103              Race tempRace;
104              tempStage = getStageObj(stageId);
105              tempRace = Race.getRaceObj(tempStage.getRaceID());
106              for (int i = 0; i < tempStage.stageSegmentList.size(); i++) {
107                  Segment.removeSegment(tempStage.stageSegmentList.get(i).getSegmentID());
108              }
109              tempRace.raceStageList.remove(tempRace.raceStageList.indexOf(tempStage));
110              stageList.remove(stageList.indexOf(tempStage)); /* removes stage and cascades down to remove
                     all segments from this stage */
111      }
112
113   }
114
115   public static int addCategorizedClimbToStage(int stageId, Double location, SegmentType type, Double
          averageGradient,
116             Double length) throws IDNotRecognisedException, InvalidLocationException,
                  InvalidStageStateException, InvalidStageTypeException{
117      /* comments for this and method below are almost exactly the same, see below */
118      if (stageIdExists(stageId) == false) {
119         throw new IDNotRecognisedException("Stage id does not exists in the system");
120      }
121      else if (getStageObj(stageId).getConcluded() == true) {
122         throw new InvalidStageStateException("Stage has been concluded and cannot be changed");
123      }
124      else if (location < 0 || location >= getStageObj(stageId).getLength()) {
125         throw new InvalidLocationException("Location is out of range");
126      }
127      else if (getStageObj(stageId).getType() == StageType.TT) {
```

```
128             throw new InvalidStageTypeException("Cannot add a sprint or mountain to time trial");
129         }
130           Segment segmentObj = new Segment( stageId, location, type, averageGradient, length);
131           getStageObj(stageId).stageSegmentList.add(segmentObj);
132           return segmentObj.getSegmentID();
133       }
134
135       public static int addIntermediateSprintToStage(int stageId, double location) throws
              IDNotRecognisedException, InvalidLocationException,
136       InvalidStageStateException, InvalidStageTypeException{
137         if (stageIdExists(stageId) == false) {
138           throw new IDNotRecognisedException("Stage id does not exists in the system"); /* check for if ID is
                  invalid */
139         }
140         else if (getStageObj(stageId).getConcluded() == true) {
141           throw new InvalidStageStateException("Stage has been concluded and cannot be changed"); /* check
                  for if stage is concluded */
142         }
143         else if (location < 0 || location >= getStageObj(stageId).getLength()) {
144           throw new InvalidLocationException("Location is out of range"); /* check to ensure location is
                  within bounds */
145         }
146         else if (getStageObj(stageId).getType() == StageType.TT) {
147           throw new InvalidStageTypeException("Cannot add a sprint or mountain to time trial"); /* check to
                  ensure stage is not a time trial */
148         }
149           Segment segmentObj = new Segment(stageId, location);
150           getStageObj(stageId).stageSegmentList.add(segmentObj); /* adds new segment object to list of
                  segments for this stage */
151           return segmentObj.getSegmentID(); /* returns new segments ID */
152
153       }
154       public static void concludeStagePreperation(int stageId) throws IDNotRecognisedException,
              InvalidStageStateException {
155         if (stageIdExists(stageId) == false) {
156           throw new IDNotRecognisedException("Stage id does not exists in the system"); /* check for if ID is
                  invalid */
157         }
158         if (getStageObj(stageId).getConcluded() == true) {
159           throw new InvalidStageStateException("Stage is already concluded"); /* check for if state is
                  already concluded */
160         }
161         getStageObj(stageId).setConcluded(true); /* sets stages concluded boolean */
162         getStageObj(stageId).setState("waiting for results"); /* sets the stage state */
163
164       }
165
166       public static int[] getStageSegments(int stageId) throws IDNotRecognisedException{
167         if (stageIdExists(stageId) == false) {
168           throw new IDNotRecognisedException("Stage id does not exists in the system"); /* ID not recognised
                  check */
169         }
170         Stage temp = getStageObj(stageId); /* gets object with relevant ID */
171           int[] stageSegments = new int[temp.stageSegmentList.size()]; /* declares amount of segments */
172
```

```
173
174       for(int i = 0; i < temp.stageSegmentList.size(); i++) {
175          stageSegments[i] = temp.stageSegmentList.get(i).getSegmentID(); /* puts segment IDS in array */
176       }
177       return stageSegments; /* returns segment IDs */
178    }
179    public static void calculatePoints(int stageId) throws IDNotRecognisedException {
180       ArrayList<Integer> stagePoints = new ArrayList<Integer>();
181       Hashtable<Integer, Integer> tempDictionary = new Hashtable<Integer, Integer>();
182       Stage temp;
183       temp = getStageObj(stageId);
184       StageType tempType = temp.getType();
185       int riderIdArr[];
186       ArrayList<Integer> ridersIds = new ArrayList<Integer>();
187       riderIdArr = Result.getRidersRankInStage(stageId);
188       for (int i = 0; i < riderIdArr.length; i++) {
189          ridersIds.add(riderIdArr[i]);
190       }
191       if (tempType == StageType.FLAT) {
192          stagePoints = distributePoints(ridersIds, Arrays.asList(50,30,20,18,16,14,12,10,8,7,6,5,4,3,2));
193       } else if (tempType == StageType.MEDIUM_MOUNTAIN) {
194          stagePoints = distributePoints(ridersIds, Arrays.asList(30,25,22,19,17,15,13,11,9,7,6,5,4,3,2));
195       } else if (tempType == StageType.HIGH_MOUNTAIN) {
196          stagePoints = distributePoints(ridersIds, Arrays.asList(20,17,15,13,11,10,9,8,7,6,5,4,3,2,1));
197       } else if (tempType == StageType.TT) {
198          stagePoints = distributePoints(ridersIds, Arrays.asList(20,17,15,13,11,10,9,8,7,6,5,4,3,2,1));
199       }
200       for (int i = 0; i <= temp.stageSegmentList.size() - 1; i++) {
201          boolean tempBool = temp.stageSegmentList.get(i).getIsIntermediateSprint();
202          if (tempBool == true) {
203             ArrayList<Integer> checkpointRiderIds = getRidersRanksForCheckpoint(stageId, i);
204             ArrayList<Integer> tempPoints = distributePoints(ridersIds,
                      Arrays.asList(20,17,15,13,11,10,9,8,7,6,5,4,3,2,1));
205             for (int k = 0; k <= tempPoints.size() - 1; k++) {
206                for (int j = 0; j <= checkpointRiderIds.size() - 1; j++) {
207                   if (ridersIds.get(k) == checkpointRiderIds.get(j)) {
208                      int currentPoints = stagePoints.get(k);
209                      stagePoints.set(k, currentPoints+tempPoints.get(j));
210                   }
211                }
212             }
213
214          }
215       }
216       for (int i = 0; i <= stagePoints.size() - 1; i++) {
217          tempDictionary.put(ridersIds.get(i), stagePoints.get(i));
218       }
219       temp.setRiderPointsDictionary(tempDictionary);
220    }
221
222    public static void calculateMountainPoints(int stageId) throws IDNotRecognisedException {
223       Stage temp;
224       temp = Stage.getStageObj(stageId);
225       boolean tempBool, firstIteration;
226       ArrayList<Integer> points = new ArrayList<Integer>();
```

```
227         ArrayList<Integer> checkpointRiderIds = new ArrayList<Integer>();
228         Hashtable<Integer, Integer> mountainPointsDictionary = new Hashtable<Integer, Integer>();
229         firstIteration = true;
230         for (int i = 0; i <= temp.stageSegmentList.size() - 1; i++) {
231            tempBool = temp.stageSegmentList.get(i).getIsIntermediateSprint();
232            if (tempBool == false) {
233               checkpointRiderIds = getRidersRanksForCheckpoint(stageId, i);
234               if (firstIteration == true) {
235                  for (int j = 0; j <= checkpointRiderIds.size() - 1; j++) {
236                     mountainPointsDictionary.put(checkpointRiderIds.get(j),0);
237                  }
238                  firstIteration = false;
239               }
240               SegmentType tempType = temp.stageSegmentList.get(i).getType();
241               if (tempType == SegmentType.C1) {
242                  points = distributePoints(checkpointRiderIds, Arrays.asList(10,8,6,4,2,1));
243               } else if (tempType == SegmentType.C2) {
244                  points = distributePoints(checkpointRiderIds, Arrays.asList(5,3,2,1));
245               } else if (tempType == SegmentType.C3) {
246                  points = distributePoints(checkpointRiderIds, Arrays.asList(2,1));
247               } else if (tempType == SegmentType.C4) {
248                  points = distributePoints(checkpointRiderIds, Arrays.asList(1));
249               } else if (tempType == SegmentType.HC) {
250                  points = distributePoints(checkpointRiderIds, Arrays.asList(20,15,12,10,8,6,4,2));
251               }
252               for (int j = 0; j <= checkpointRiderIds.size() - 1; j++) {
253                  mountainPointsDictionary.put(checkpointRiderIds.get(j),
                       mountainPointsDictionary.get(checkpointRiderIds.get(j)) + points.get(j));
254               }
255            }
256         }
257         temp.setRiderMountainPointsDictionary(mountainPointsDictionary);
258      }
259
260      public static ArrayList<Integer> getRidersRanksForCheckpoint(int stageId, int location) throws
            IDNotRecognisedException {
261         int[] tempRiders = Result.getRidersRankInStage(stageId);
262         Hashtable<LocalTime, Integer> checkpointsDictionary = new Hashtable<LocalTime, Integer>();
263         LocalTime tempCheckpoint;
264         ArrayList<LocalTime> checkpoints = new ArrayList<LocalTime>();
265         for (int i = 0; i <= tempRiders.length - 1; i++) {
266            tempCheckpoint = Result.getRiderResultsInStage(stageId, tempRiders[i])[location];
267            checkpoints.add(tempCheckpoint);
268            checkpointsDictionary.put(tempCheckpoint, tempRiders[i]);
269         }
270         checkpoints.sort(new Result.CheckpointComparator());
271         ArrayList<Integer> riderIds = new ArrayList<Integer>();
272         for (int x = 0; x <= checkpoints.size() - 1; x++) {
273            riderIds.add(checkpointsDictionary.get(checkpoints.get(x)));
274         }
275         return riderIds;
276      }
277
278      public static ArrayList<Integer> distributePoints(ArrayList<Integer> riderIds, List<Integer> pointsSet)
            {
```

```
279         ArrayList<Integer> points = new ArrayList<Integer>();
280         int tempPoints = 0;
281         for (int i = 0; i < riderIds.size(); i++) {
282            tempPoints = 0;
283            if (i < pointsSet.size()) {
284               tempPoints = pointsSet.get(i);
285            }
286            points.add(tempPoints);
287         }
288         return points;
289      }
290       /**
291        * Returns Race ID
292        * @return raceID
293        * @author Nathan
294        */
295       public int getRaceID() {
296         return this.raceID;
297       }
298       /**
299        * Returns stage concluded state
300        * @return concluded
301        * @author Nathan
302        */
303       public boolean getConcluded() {
304         return this.concluded;
305       }
306       /**
307        * Gets stage segment list
308        * @return ArrayList stageSegmentList
309        * @author Nathan
310        */
311       public ArrayList<Segment> getStageSegmentList() {
312         return this.stageSegmentList;
313       }
314       /**
315        * Sets concluded boolean
316        * @param value true / false
317        * @author Nathan
318        */
319       public void setConcluded(boolean value) {
320         this.concluded = value;
321       }
322       /**
323        * Sets stage state
324        * @param value
325        * @author Nathan
326        */
327       public void setState(String value) {
328         this.state = value;
329       }
330       /**
331        * Returns stage state
332        * @return state
333        * @author Nathan
```

```
334       */
335      public String getState() {
336        return this.state;
337      }
338      /**
339       * Returns stage name
340       * @return stage name
341       * @author Nathan
342       */
343      public String getStageName() {
344        return this.stageName;
345      }
346      /**
347       * Returns description
348       * @return description
349       * @author Nathan
350       */
351      public String getDescription() {
352        return this.description;
353      }
354      /**
355       * Returns length type
356       * @return length
357       * @author Nathan
358       */
359      public double getLength() {
360        return this.length;
361      }
362      /**
363       * Returns start time
364       * @return start time
365       * @author Nathan
366       */
367      public LocalDateTime getStartTime() {
368        return this.startTime;
369      }
370      /**
371       * Returns stage type
372       * @return type
373       * @author Nathan
374       */
375      public StageType getType() {
376        return this.type;
377      }
378      /**
379       * Hashtable that stores the riders points
380       * @return riderPointsDictionary
381       * @author Sam
382       */
383     public Hashtable<Integer, Integer> getRiderPointsDictionary() {
384       return this.riderPointsDictionary;
385      }
386     /**
387       * Hashtable that stores the riders mountain points
388       * @return riderPointsDictionary
```

```
389       * @author Sam
390       */
391     public Hashtable<Integer, Integer> getRiderMountainPointsDictionary() {
392        return this.riderMountainPointsDictionary;
393      }
394     /**
395       * Rider mountain points Hashtable setter
396       * @author Sam
397       */
398     public void setRiderMountainPointsDictionary(Hashtable<Integer, Integer> newTable) {
399        this.riderMountainPointsDictionary = newTable;
400      }
401     /**
402       * Rider points Hashtable setter
403       * @author Sam
404       */
405     public void setRiderPointsDictionary(Hashtable<Integer, Integer> newtable) {
406        this.riderPointsDictionary = newtable;
407     }
408 }
```

4 Segment.java

```
1  package cycling;
2
3  import java.util.ArrayList;
4  /**
5   * Class containing all related methods and variables of segments of a stage
6   * @author Nathan
7   *
8   */
9  public class Segment {
10    int stageID, segmentID;
11     double location, averageGradient, length;
12     SegmentType type;
13     boolean isIntermediateSprint = false;
14     /**
15      * List of all segments currently stores for all stages
16      */
17     public static ArrayList<Segment> segmentList = new ArrayList<Segment>();
18
19    public Segment(int stageId, double location, SegmentType type, Double averageGradient, /* Constructor
          for Climb */
20           Double length) {
21       int temp;
22       this.location = location;
23       this.type = type;
24       this.averageGradient = averageGradient;
25       this.length = length;
26       this.stageID = stageId;
27       temp = segmentList.size();
28         if (temp > 0){                                  /* check for if this is the first id */
29               this.segmentID = segmentList.get(temp).segmentID + 1; /* in the system or if it needs to be
                      calculated */
```

```
30          }                                           /* based of previous ID, this is performed in every
                constructor */
31          else {                                      /* and is only commented on here */
32                this.segmentID = 0;
33          }
34          segmentList.add(this);
35      }
36
37      public Segment(int stageId, Double location) { /* Constructor for Sprint */
38         int temp;
39         this.location = location;
40         this.stageID = stageId;
41         this.isIntermediateSprint = true;
42         temp = segmentList.size();
43           if (temp > 0){
44                 this.segmentID = segmentList.get(temp - 1).segmentID + 1;
45           }
46           else {
47                 this.segmentID = 0;
48           }
49           segmentList.add(this);
50      }
51      /**
52       * Returns segment ID
53       * @return segmentID
54       * @author Nathan
55       */
56      public int getSegmentID() {
57         return this.segmentID;
58      }
59      /**
60       * Returns the segment object with the stored ID passed in
61       * @param passedID Segment ID to be checked against
62       * @return relevant Segment object
63       * @author Nathan
64       */
65      public static Segment getSegmentObj(int passedID){
66          for (int i = 0; i < segmentList.size(); i++) {
67              if (segmentList.get(i).getSegmentID() == passedID) {
68                  return segmentList.get(i); /* returns relevant segment ID */
69              }
70          }
71          return null;
72       }
73      /**
74       * Verifies if segment ID is currently in use
75       * @param segmentId ID to be checked against
76       * @return true if exists / false if not
77       * @author Nathan
78       */
79      public static boolean segmentIdExists(int segmentId) {
80         boolean exists = false;
81         for(int i=0; i <segmentList.size(); i++) {
82            if (segmentList.get(i).getSegmentID() == segmentId) {
83               exists = true; /* segment ID is currently stored */
```

```
84              break;
85           }
86        }
87        if (exists == true) {
88           return true;
89        }
90        else {
91           return false;
92        }
93     }
94     public static void removeSegment(int segmentID) throws IDNotRecognisedException{
95        if (segmentIdExists(segmentID) == false) {
96           throw new IDNotRecognisedException("Segment id does not exists in the system"); /* Invalid ID check
                 */
97        }
98          Segment tempSegment;
99          Stage tempStage;
100         tempSegment = getSegmentObj(segmentID);
101         tempStage = Stage.getStageObj(tempSegment.getStageID());
102         tempStage.stageSegmentList.remove(tempStage.stageSegmentList.indexOf(tempSegment));
103         segmentList.remove(segmentList.indexOf(tempSegment)); /* removes segment based on ID */
104     }
105     /**
106      * Location getter
107      * @return location
108      * @author Nathan
109      */
110     public double getLocation() {
111        return this.location;
112     }
113     /**
114      * Type getter
115      * @return type
116      * @author Nathan
117      */
118     public SegmentType getType() {
119        return this.type;
120     }
121     /**
122      * Avg Gradient getter
123      * @return averageGradient
124      * @author Nathan
125      */
126     public double avgGradient() {
127        return this.averageGradient;
128     }
129     /**
130      * Length getter
131      * @return length
132      * @author Nathan
133      */
134     public double getLength() {
135        return this.length;
136     }
137     /**
```

```
138     * Stage ID getter
139     * @return stageID
140     * @author Nathan
141     */
142    public int getStageID() {
143       return this.stageID;
144    }
145    /**
146     * Intermediate Sprint boolean getter
147     * @return isIntermediateSprint
148     * @author Nathan
149     */
150    public boolean getIsIntermediateSprint() {
151       return this.isIntermediateSprint;
152    }
153 }
```

5 Rider.java

```
1  package cycling;
2
3  import java.util.ArrayList;
4  /**
5   * Class containing all rider related methods and variables
6   * @author Nathan
7   *
8   */
9  public class Rider {
10     String name;
11     private int yearOfBirth, teamID, riderID;
12     /**
13      * Stores all current rider object
14      * @author Nathan
15      */
16     public static ArrayList<Rider> riderList = new ArrayList<Rider>();
17       public Rider(String name, int yearOfBirth, int teamID) throws IDNotRecognisedException,
             IllegalArgumentException{
18          try {
19           this.name = name;
20             this.yearOfBirth = yearOfBirth;
21             this.teamID = teamID;
22         }
23           catch (IllegalArgumentException e) {
24           throw new IllegalArgumentException("Illegal argument passed via one of the variables"); /*
                 Illegal argument check */
25         }
26
27           if (Team.teamIdExists(teamID) == false) {
28            throw new IDNotRecognisedException("Team id does not exists in the system"); /* ID check */
29         }
30
31           int x = riderList.size();
32           if (x > 0){
33            this.riderID = riderList.get(x - 1).riderID + 1;
```

```
34            }
35            else {
36             this.riderID = 0;
37            }
38            riderList.add(this);
39
40        }
41        public static void removeRider(int riderID) throws IDNotRecognisedException {
42          if (riderIdExists(riderID) == false) {
43             throw new IDNotRecognisedException("Team id does not exists in the system");
44          }
45          Rider temp;
46          temp = getRiderObj(riderID);
47            riderList.remove(riderList.indexOf(temp)); /* removes rider from ArrayList based on the index of
                  the object with the related passed ID */
48        }
49        /**
50         * Rider ID getter
51         * @return Rider ID
52         * @author Nathan
53         */
54        public int getRiderID(){
55          return this.riderID;
56       }
57         /**
58         * Gets rider object with the related passed ID and returns it
59         * @param passedID ID of rider you wish to get
60         * @return Rider object
61         * @author Nathan
62         */
63        public static Rider getRiderObj(int passedID){
64            for (int i = 0; i < riderList.size(); i++) {
65                if (riderList.get(i).getRiderID() == passedID) {
66                    return riderList.get(i);
67                }
68            }
69            return null;
70        }
71        /**
72         * Check for if this rider ID is already in use (exists)
73         * @param riderId ID you wish to check against
74         * @return true if exists / false if not
75         * @author Nathan
76         */
77        public static boolean riderIdExists(int riderId) {
78          boolean exists = false;
79          for(int i=0; i <riderList.size(); i++) {
80             if (riderList.get(i).getRiderID() == riderId) {
81                exists = true; /* rider ID is found to exist */
82                break;
83             }
84          }
85          if (exists == true) {
86             return true;
87          }
```

```
88          else {
89             return false;
90          }
91       }
92        /**
93         * Name getter
94         * @return name
95         * @author Nathan
96         */
97        public String getName() {
98          return this.name;
99        }
100       /**
101        * Year of birth getter
102        * @return yearOfBirth
103        * @author Nathan
104        */
105       public int getYearOfBirth() {
106         return this.yearOfBirth;
107       }
108       /**
109        * Team ID getter
110        * @return teamID
111        * @author Nathan
112        */
113       public int getTeamId() {
114         return this.teamID;
115       }
116    }
```

6 Team.java

```
1  package cycling;
2
3  import java.util.*;
4
5  /**
6   * Class containing all Team related methods and variables
7   * @author Nathan
8   *
9   */
10 public class Team {
11    private String name, description;
12    private int teamID;
13    /**
14     * Arraylist of all currently created teams
15     * @author Nathan
16     */
17    static public ArrayList<Team> teamStore = new ArrayList<Team>();
18
19    public Team(String name, String description) throws IllegalNameException, InvalidNameException{
20       this.description = description;
21       this.name = name;
22
```

```
23        for (int i = 0; i < teamStore.size(); i++) {
24           if (teamStore.get(i).getName() == name){
25              throw new IllegalNameException("This name is currently being used by another team"); /* Name
                    exception check */
26           }
27        }
28          if (name.isEmpty() || name.length() >= 256 || name.contains(" ")) {
29           throw new InvalidNameException("Name cannot be empty, include whitespaces, "
30                 + "or be larger than the system limit of characters"); /* Invalid name exception check */
31          }
32        int x = teamStore.size();
33          if (x > 0){
34           this.teamID = teamStore.get(x - 1).teamID + 1;
35          }
36          else {
37           this.teamID = 0;
38          }
39
40        teamStore.add(this);
41     }
42     /**
43      * Returns the team object with the stored passed in team ID
44      * @param passedID Team ID of object you wish to get
45      * @return Team object
46      * @author Nathan
47      */
48     public static Team getTeamObj(int passedID){
49         for (int i = 0; i < teamStore.size(); i++) {
50             if (teamStore.get(i).getTeamID() == passedID) {
51                 return teamStore.get(i);
52             }
53         }
54         return null;
55      }
56     /**
57      * Check performed to see if passed in team ID is in use (exists)
58      * @param teamId team ID you wish to check against
59      * @return true if exists / false if not
60      */
61     public static boolean teamIdExists(int teamId) {
62        boolean exists = false;
63        for(int i=0; i <teamStore.size(); i++) {
64           if (teamStore.get(i).getTeamID() == teamId) {
65              exists = true; /* teamId is found to exist */
66              break;
67           }
68        }
69        if (exists == true) {
70           return true;
71        }
72        else {
73           return false;
74        }
75     }
76     /**
```

```
77      * Team ID getter
78      * @return teamID
79      * @author Nathan
80      */
81     public int getTeamID(){
82        return this.teamID;
83     }
84     /**
85      * Team ID setter
86      * @param passed
87      * @author Nathan
88      */
89     public void setTeamID(int passed){
90        this.teamID = passed;
91     }
92     /**
93      * Name getter
94      * @return name
95      * @author Nathan
96      */
97     public String getName(){
98        return this.name;
99     }
100    /**
101     * Description getter
102     * @return description
103     * @author Nathan
104     */
105    public String getDescription(){
106       return this.description;
107    }
108
109    public static int[] getAllTeamIDs() {
110       if (teamStore.isEmpty()) {
111          return new int[]{}; /* if no teams exist returns empty array */
112       }
113       int length;
114       length = teamStore.size();
115       int[] teamIDs = new int[length];
116
117
118       for(int i = 0; i <= teamStore.size() - 1; i++) {
119          teamIDs[i] = teamStore.get(i).getTeamID(); /* gets all team IDs and stores them in an array */
120       }
121       return teamIDs;
122    }
123    public static void removeTeam(int teamId) throws IDNotRecognisedException {
124       if (teamIdExists(teamId) == false) {
125          throw new IDNotRecognisedException("Team id does not exists in the system"); /* ID not recognised
               check */
126       }
127         Team tempTeam;
128         tempTeam = getTeamObj(teamId);
129         for (int i = 0; i < Rider.riderList.size(); i++) {
130          if (Rider.riderList.get(i).getTeamId() == teamId){
```

```
131             Rider.removeRider(Rider.riderList.get(i).getRiderID()); /* removes team based on ID and cascades
                    down to remove riders */
132          }
133         }
134         teamStore.remove(teamStore.indexOf(tempTeam));
135     }
136
137     public static int[] getTeamRiders(int teamId) throws IDNotRecognisedException {
138        if (teamIdExists(teamId) == false) {
139           throw new IDNotRecognisedException("Team id does not exists in the system"); /* ID not recognised
                   check */
140        }
141        int[] riderIdList;
142        int temp = 0;
143        for (int i = 0; i < Rider.riderList.size(); i++) {
144           if (Rider.riderList.get(i).getTeamId() == teamId){
145              temp += 1; /* gets how many riders are stored for this particular team */
146           }
147         }
148        riderIdList = new int[temp];
149        for (int i = 0; i < Rider.riderList.size(); i++) {
150           if (Rider.riderList.get(i).getTeamId() == teamId){
151              riderIdList[i] = Rider.riderList.get(i).getRiderID(); /* stores all rider IDs for this team */
152           }
153         }
154     return riderIdList; /* returns list of rider IDS */
155     }
156
157 }
```