**Question 1**

1a)

/\*\*

\* Sets the colour of the specified TrafficLight to the specified colour.

\* @param aLight which is a reference to TrafficLight.

\* @param aColour which is an OUColour.

\*/

public void colourLight(TrafficLight aLight, OUColour aColour)

{

aLight.setColour(aColour);

}

1b) i)

private int state; // Pattern of lights corresponds to value stored.

1b) ii)

/\*\*

\* Constructor for objects of class TrafficSystem.

\*/

public TrafficSystem(TrafficLight aNorth, TrafficLight aSouth,

TrafficLight aEast, TrafficLight aWest)

{

this.north = aNorth;

this.south = aSouth;

this.east = aEast;

this.west = aWest;

this.state = 0;

this.setPositions();

}

1c) i)

/\*\*

\* Cycles to the next state:

\* from 1 to 2, from 2 to 3, from 3 to 4 and back to 1.

\*/

private void cycleState()

{

this.state += 1;

if(this.state > 4)

{

this.state = 1;

}

}

1c) ii)

/\*\*

\* Sets the colours of the four TrafficLight objects based

\* on the current state.

\*/

private void colourAllLights()

{

if(this.state == 0)

{

this.colourLight(north, OUColour.RED);

this.colourLight(south, OUColour.RED);

this.colourLight(east, OUColour.RED);

this.colourLight(west, OUColour.RED);

}

else if(this.state == 1)

{

this.colourLight(north, OUColour.GREEN);

this.colourLight(south, OUColour.GREEN);

this.colourLight(east, OUColour.RED);

this.colourLight(west, OUColour.RED);

}

else if(this.state == 2)

{

this.colourLight(north, OUColour.ORANGE);

this.colourLight(south, OUColour.ORANGE);

this.colourLight(east, OUColour.RED);

this.colourLight(west, OUColour.RED);

}

else if(this.state == 3)

{

this.colourLight(north, OUColour.RED);

this.colourLight(south, OUColour.RED);

this.colourLight(east, OUColour.GREEN);

this.colourLight(west, OUColour.GREEN);

}

else // State 4

{

this.colourLight(north, OUColour.RED);

this.colourLight(south, OUColour.RED);

this.colourLight(east, OUColour.ORANGE);

this.colourLight(west, OUColour.ORANGE);

}

}

1c) iii)

/\*\*

\* Constructor for objects of class TrafficSystem.

\*/

public TrafficSystem(TrafficLight aNorth, TrafficLight aSouth,

TrafficLight aEast, TrafficLight aWest)

{

this.north = aNorth;

this.south = aSouth;

this.east = aEast;

this.west = aWest;

this.state = 0;

this.setPositions();

this.colourAllLights(); // Initialise colour of lights.

}

1d) i)

private boolean go; // Determines automation of traffic lights.

/\*\*

\* Constructor for objects of class TrafficSystem.

\*/

public TrafficSystem(TrafficLight aNorth, TrafficLight aSouth,

TrafficLight aEast, TrafficLight aWest)

{

this.north = aNorth;

this.south = aSouth;

this.east = aEast;

this.west = aWest;

this.state = 0;

this.setPositions();

this.colourAllLights(); // Initialise colour of lights.

this.go = false; // System initially not working automatically.

}

1d) ii)

/\*\*

\* Setter method for instance variable go.

\*/

public void setGo(boolean isGo)

{

this.go = isGo;

}

1d) iii)

/\*\*

\* Runs the traffic lights automatically when instance variable go

\* is true. Cycles through each state which change the traffic

\* light colours with a delay between each change.

\*/

public void runLights()

{

while(this.go)

{

this.cycleState();

this.colourAllLights();

this.delay(2000);

}

}

1e)

/\*\*

\* Manually override the traffic light system.

\* An integer value between 0 and 4 is to be entered to set

\* the state of theTrafficLight objects. A integer value

\* outside this range will cause a out of range message.

\* @throws NumberFormatException if newState value is not

\* an integer.

\*/

public void manualOverride()

{

if (!this.go)

{

try

{

int newState = Integer.parseInt(OUDialog.request

("Please give the state you want to change to - between 0 and 4 inclusive"));

if(newState >= 0 && newState <= 4)

{

this.state = newState;

this.colourAllLights();

}

else

{

OUDialog.alert("State value " + newState +

" entered not in the 0 to 4 value range");

}

}

catch(NumberFormatException anException)

{

OUDialog.alert("State value inappropriate, integer expected");

}

}

}

**Question 2**

2a) i)

// Instance variable references a map which will hold String value

// division for the key and a List of teams as the value.

private Map<String, List<Team>> teams;

2a) ii)

/\*\*

\* Constructor for objects of class LeagueAdmin.

\*/

public LeagueAdmin()

{

this.teams = new HashMap<String, List<Team>>();

}

2a) iii)

/\*\*

\* Adds a team to a division. If division already exists,

\* team is added to division otherwise a new division is

\* created and then team is added to the new division.

\* @param division which team is to be added to.

\* @param team is Team object that will be added to a division.

\*/

public void addTeam(String division, Team team)

{

if(this.teams.containsKey(division)) // Key already exists.

{

this.teams.get(division).add(team);

}

else // Key does not exist.

{

List<Team>teamList = new ArrayList<Team>();

teamList.add(team);

this.teams.put(division, teamList);

}

}

2b)

/\*\*

\* Updates two specific teams from a division depending on the

\* method argument teamAScore and teamBScore.

\* @param division a String indicates which division result is for.

\* @param teamA a String which team played (one of two).

\* @param teamB a String which team played (one of two).

\* @param teamAScore an integer of number goals teamA scored.

\* @param teamBScore an integer of number goals teamB scored.

\*/

public void recordResult(String division, String teamA, String teamB,

int teamAScore, int teamBScore)

{

List<Team> teamList = new ArrayList<>(this.teams.get(division));

for(Team aTeam : teamList)

{

if(aTeam.getName().equals(teamA)) // teamA test

{

if(teamAScore > teamBScore)

{

aTeam.incWon();

}

else if(teamAScore < teamBScore)

{

aTeam.incLost();

}

else // TeamA drew

{

aTeam.incDrew();

}

}

if(aTeam.getName().equals(teamB)) // teamB test

{

if(teamAScore > teamBScore)

{

aTeam.incLost();

}

else if(teamAScore < teamBScore)

{

aTeam.incWon();

}

else // TeamA drew

{

aTeam.incDrew();

}

}

}

}

2c)

/\*\*

\* Prints out the statistics for all teams in the

\* division given by the method argument.

\* @param division String of division to be displayed.

\*/

public void printOutTable(String division)

{

System.out.println(division);

System.out.format("%-20s %2s %2s %2s %2s \n",

"Name","W","L","D"," Pts" );

for(Team aTeam : this.teams.get(division))

{

System.out.format("%-20s %2d %2d %2d %2d\n", aTeam.getName(),

aTeam.getWon(), aTeam.getLost(),

aTeam.getDrew(), aTeam.getPoints());

}

}

2d) i)

The fellow student may have considered an array because a division of teams is normally a set number. However, lists are more suitable for this because:

* Lists can dynamically change size, allowing teams to be added/removed without worrying about the maximum size required. This could be important in this instance because the List is the value part of a Map, therefore it is entirely possible that one of the values of a key will be a different size to another one of the values of a key (i.e. Premier will have less teams than the Championship). An array length is initialised at time of creation. This would mean that once the array is full of teams a new larger array would have to be created and everything copied to this new array; a lot more complex than it needs to be.
* Lists are part of the Collections Framework. This means that they work smoothly with the other collection classes which have a number of methods available to manipulate lists. For example, iterating through a List of a particular key of a Map can be performed more efficiently or Sorting the List by points a team have. With an Array, working with the Map Interface and its Collection Classes is more awkward because it is not part of the Collections Framework. Performing the same processes would be more complex, more effort and may be harder to follow.

2d) ii)

Ordering the elements in a Collection is where the elements in the collection follow a natural order, the order is independent of the element value.. For example, a List uses an index to remember where an element has been stored. This allows an element to be added to/ removed from a particular place in the List.

Sorting the elements in a Collection is where the ordering depends on the value of an element. The collection could be sorted alphabetically or numerically depending the element value. For example, a Set of Surnames could be sorted using the SortedSet Interface and the TreeSet Collection Class, so they are sorted in alphabetical order.

**Question 3**

3a)

|  |  |
| --- | --- |
| Requirement | Answer |
| General scenario | A delivery company keeps records of delivery routes using a sorted map. Each key of the map is a String representing the name of the delivery route and the values are Sets of Strings where each String in the Set represents the name of the business where a delivery should be made. |
| Type of key (String or integer) | String |
| Type of value (must be from Collections framework) | Set |
| Type of map (sorted or unsorted) | Sorted |
| Description of any additional classes | No additional classes. |

Note: There is test data in the readme file associated with TMA03Q3\_Sol

3b)

import java.util.\*;

/\*\*

\* This class simulates a number of delivery routes.

\* Each delivery routes data can be manipulated in a

\* variety of ways.

\*

\* @author Wayne Sandford

\* @version 05-04-19

\*/

public class DeliveryRoutes

{

// instance variable

private SortedMap<String, Set<String>> routes;

/\*\*

\* Constructor for objects of class DeliveryRoutes

\*/

public DeliveryRoutes()

{

this.routes = new TreeMap<String, Set<String>>();

}

/\*\*

\* Populates the map with test data

\*/

public void populate()

{

Set<String> businesses = new HashSet<>();

businesses.add("Falcon News");

businesses.add("H R Newsagents");

businesses.add("WHSmith");

this.routes.put("Portsmouth", businesses);

businesses = new HashSet<>();

businesses.add("The News Box");

businesses.add("Harbour News");

businesses.add("WHSmith");

this.routes.put("Gosport", businesses);

businesses = new HashSet<>();

businesses.add("Kiosk News");

this.routes.put("Havant", businesses);

}

}

3c)

/\*\*

\* Prints out each delivery route and their associated

\* businesses that are on that route.

\*/

public void printMap()

{

for(String eachRoute : routes.keySet())

{

System.out.println("Delivery addresses for route: " + eachRoute);

for(String eachBusiness : routes.get(eachRoute))

{

System.out.println(eachBusiness);

}

System.out.println();

}

}

/\*\*

\* Prints out the businesses of a route if the method

\* argument is a valid route otherwise prints out

\* route not found message.

\* @param routeName a String representing the route name.

\*/

public void printMapValue(String routeName)

{

if(this.routes.containsKey(routeName)) // Route exists.

{

System.out.println("There are currently "

+ this.routes.get(routeName).size()

+ " businesses :" );

for(String eachBusiness : this.routes.get(routeName))

{

System.out.println(eachBusiness);

}

}

else // Route does not exist.

{

System.out.println("The delivery route for "

+ routeName + " has not been found.");

}

}

/\*\*

\* Adds a route and its associated businesses to the map,

\* which are taken from the method argument. If the route name is

\* already present the existing business values are overwritten.

\* @param routeName a String representing the route name.

\* @param setOfBusinesses a Set of Strings representing businesses.

\*/

public void addMapEntry(String routeName, Set<String> setOfBusinesses)

{

this.routes.put(routeName, setOfBusinesses);

}

3d)

/\*\*

\* Takes an argument representing a route name in the map.

\* If the route name exists, route and associated businesses

\* are deleted from the map and returns true, otherwise returns false.

\* @param routeName a String representing the route name.

\*/

public boolean deleteEntry(String routeName)

{

return (this.routes.remove(routeName) != null);

}

/\*\*

\* Returns a map of all the businesses in each route that have

\* the same names as those that are in the method arguments Set

\* of business names.

\* @param aBusinessesNames a Set of Strings representing the

\* names of businesses to find in the Map values.

\* @return newMap a Map containing only the businesses that

\* match ones that appear in aBusinessesNames.

\*/

public SortedMap<String, Set<String>> selectValues(Set<String> aBusinessesNames)

{

SortedMap<String, Set<String>> newMap = new TreeMap<>();

Set<String> eachRouteOfBusinesses;

Set<String> intersection;

for(String route : this.routes.keySet())

{

eachRouteOfBusinesses = this.routes.get(route);

intersection = new HashSet<>(aBusinessesNames);

intersection.retainAll(eachRouteOfBusinesses);

newMap.put(route, intersection);

}

return newMap;

}

/\*\*

\* Adds a new business to the Set of businesses for a

\* particular routeName (key). It assumes that the

\* routeName exists in the map.

\* @param routeName a String representing the key.

\* @param aBusiness a String representing the business.

\*/

public void addToValue(String routeName, String aBusiness)

{

Set<String> setOfBusinesses = this.routes.get(routeName);

setOfBusinesses.add(aBusiness);

}

**Question 4**

4a) i)

private static int nextNumber = 1; // Next entrant number to be allocated

4a) ii)

number = nextNumber++;

4b) i)

Instance variable:

List<Entrant> entrants; // Reference a list of Entrant objects

Added to constructor:

entrants = new ArrayList<>();

4b) ii)

/\*\*

\* Prompts the user for a pathname and then attempts to open a

\* stream of the specified entrants file which must be in

\* CSV format. Each line of data from the file creates a new

\* Entrant object which is then stored in the entrants list.

\* It is assumed all file data is valid.

\*/

public void readInEntrants()

{

String pathname = OUFileChooser.getFilename();

File entrantsFile = new File(pathname);

Scanner bufferedScanner = null;

Entrant entrant;

try

{

Scanner lineScanner;

String currentLine;

bufferedScanner = new Scanner(new BufferedReader

(new FileReader(entrantsFile)));

while(bufferedScanner.hasNextLine()) // While not end of file.

{

currentLine = bufferedScanner.nextLine();

lineScanner = new Scanner(currentLine);

lineScanner.useDelimiter(",");

entrant = new Entrant();

entrant.setName(lineScanner.next());

entrant.setCategory(lineScanner.next());

entrants.add(entrant);

}

}

catch(Exception anException)

{

System.out.println("Error: " + anException);

}

finally

{

try

{

bufferedScanner.close();

}

catch(Exception anException)

{

System.out.println("Error: " + anException);

}

}

}

4b) iii)

/\*\*

\* Calculates a random double value between 30.00 (inclusive)

\* and 60.00 (exclusive).

\* @return a double between 30.00 (inclusive)

\* and 60.00 (exclusive).

\*/

private double generateTime()

{

Random randomNumber = new Random();

return 30.00 + (60.00 - 30.00) \* randomNumber.nextDouble();

}

4b) iv)

/\*\*

\* Iterates over the list of entrants, and for each entrant generates

\* a random number between 30.00 (inclusive) and 60.00 (exclusive)

\* which is used to set the time (in minutes) for that entrant.

\*/

public void runRally()

{

for(Entrant eachEntrant : entrants)

{

eachEntrant.setTime(generateTime());

}

}

4c) i)

Modified class header:

public class Entrant implements Comparable<Entrant>

compareTo() method:

/\*\*

\* Compares the time of receiver against the time of the

\* Entrant object from the method argument.

\* @return an integer value of 0 if the arguments are numerically equal,

\* a negative value if the first argument is less than the second,

\* and a positive value if the first argument is greater than the

\* second.

\*/

public int compareTo(Entrant anotherEntrant)

{

return Double.compare(this.getTime(), anotherEntrant.getTime());

}

4c) ii)

/\*\*

\* Sorts the entrants list by the time each entrant has taken to

\* complete the rally.

\*/

public void sortEntrantList()

{

Collections.sort(entrants);

}

4d) i)

SortedMap<String, Double> e1Results; // Category E1 cars

SortedMap<String, Double> e2Results; // Category E2 cars

SortedMap<String, Double> e3Results; // Category E3 cars

4d) ii)

Added to constructor:

e1Results = new TreeMap<>();

e2Results = new TreeMap<>();

e3Results = new TreeMap<>();

4d) iii)

/\*\*

\* Iterates over entrants list, populating the maps e1Results,

\* e2Results and e3Results, with the correct names and

\* times of entrants.

\*/

public void categorise()

{

for(Entrant eachEntrant : entrants)

{

if(eachEntrant.getCategory().equals("E1"))

{

e1Results.put(eachEntrant.getName(), eachEntrant.getTime());

}

else if(eachEntrant.getCategory().equals("E2"))

{

e2Results.put(eachEntrant.getName(), eachEntrant.getTime());

}

else // "E3"

{

e3Results.put(eachEntrant.getName(), eachEntrant.getTime());

}

}

}