**Class Description**

1. **Snake**

Attributes:

1. direction: char (Direction for the movement of the snake)
2. xcor : LinkedList<Integer> (LinkedList of x-coordinates of the snake body)
3. ycor : LinkedList<Integer>(LinkedList of y-coordinates of the snake body)
4. boardsize: int (size of the game board)

Methods:

* 1. public boolean foodEaten(int[]): This method takes as parameter the array of food coordinates. It returns TRUE if the snake eats the food and returns FALSE otherwise.
  2. public boolean collisionWithSelf(): If the head of the snake collides with the rest of its body, it returns TRUE, else it returns FALSE.
  3. public boolean collisionWithWall(): If the head of the snake collides with the wall, it returns TRUE, otherwise it returns FALSE.
  4. public boolean updateSnake(int[]) : This method takes as parameter the array of food coordinates. It runs a switch case on the direction inputted by the user and moves on the board accordingly. Based on the coordinates of the food unit and the direction of movement of the snake, if the food gets eaten by the snake, the length of the snake is increased by one and the method returns true. If the snake does not eat the food after moving in the inputted direction, the coordinates of the snake get updated and the method returns FALSE.

1. **GameBoard**

Attributes:

1. boardSize: int (Size of game board)
2. board: char[][] (matrix for game board)
3. snake
4. food: Food (
5. foodCount: int (number of food units consumed by the snake)

Methods:

* 1. public void generateFood(): This method calls the generateNewFood() method of the Food class to generate new food on a random location on the gameboard.
  2. public void defaultGameboard(): This method sets the gameboard for the first iteration. It populates the board with the initial coordinates of the snake.
  3. public void updateGameboard(): This method is responsible for updating the gameboard after every iteration. It updates the coordinates of the snake and food on the gameboard after every iteration.
  4. public void displayGameboard(): This method simply displays the gameboard to the user after every iteration.
  5. public int getFoodCount(): returns the total number of food units consumed by the snake.
  6. public boolean handleInput(char): This method takes the direction inputted by the user as its parameter. It checks if the direction inputted by the user is valid or not. If the input is found valid, it simply sets the direction. It calls the updateSnake() method of the Snake class to update the snake coordinates according to the direction inputted and checks whether the food is eaten by the snake. If so, it increments the food count, calls generateFood() and calls updateGameboard(). It calls collisionWithSelf() and collisionWithWall() methods of the Snake class and returns FALSE if the snake collides with itself or with the wall (These cases indicate that the game is over). It returns TRUE otherwise, indicating that the game is still continuing.

1. **Food**

Attributes:

1. foodCor: int[] (stores the x and y coordinates of food unit)

Methods:

* 1. public void generateNewFood(char[][]): It takes as parameters the entire board matrix and randomly generates coordinates of the food such that the coordinates of food do not lie over coordinates of the snake body.

1. **Player**

Attributes:

1. username : String (username of the Player)
2. password : String (password of the Player)
3. score10 : long (score of the player for board size 10x10)
4. score15 : long (score of the player for board size 15x15)
5. score20 : long (score of the player for board size 20x20)

Methods:

* 1. public void updateScore(int, long): This method takes boardsize and score as parameters. Based on the board size, it updates the score of the current Player object.
  2. public static Player userExistence(LinkedList<Player>, String): This method takes a LinkedList of Player objects and a string name as parameters. It checks whether the user already exists by iterating over the list and finding the user with the same username as the string name in the parameter. If such a user is found, it is returned. If not, the method returns null.

1. **Score10:** This class implements the Comparable interface to sort the objects of the Player class on the basis of their score in the board size of 10x10.

Methods:

* 1. public int compare(Player, Player): This method takes two objects of the Player class as parameters and is used later to sort them in descending order on the basis of their scores. It returns an integer value after comparing the scores of the two players in 10x10 board sizes.

1. **Score15:** This class implements the Comparable interface to sort the objects of the Player class on the basis of their score in the board size of 15x15.

Methods:

* 1. public int compare(Player, Player): This method takes two objects of the Player class as parameters and is used later to sort them in descending order on the basis of their scores. It returns an integer value after comparing the scores of the two players in 15x15 board sizes.

1. **Score20**: This class implements the Comparable interface to sort the objects of the Player class on the basis of their score in the board size of 20x20.

Methods:

* 1. public int compare(Player, Player): This method takes two objects of the Player class as parameters and is used later to sort them in descending order on the basis of their scores. It returns an integer value after comparing the scores of the two players in 20x20 board sizes.

1. **Game**: This class is responsible for accepting user inputs.

Methods:

* 1. public static void displayRules(): This method displays all the basic instructions and rules required to play the game.
  2. public static boolean validGameLevel(char): This method takes character c as parameter for difficulty level. Returns TRUE if the user inputs valid input for the difficulty level (i.e, inputs ‘e’, ‘h’ or ‘m’). It returns FALSE otherwise.
  3. public static int getFoodPoint(char): This method is used to return the default score points that a user gains upon consuming a food unit, on the basis of difficulty level. It takes a character parameter for difficulty level. If the user inputs ‘e’ (easy level), 5 is returned. If the user inputs ‘m’ (medium level), 10 is returned. In all other cases 15 is returned.
  4. public static int getTime(char): This method is used to return the default time that a user gets for inputting direction for the snake movement, on the basis of difficulty level. It takes a character parameter for difficulty level. If the user inputs ‘e’ (easy level), 10 is returned. If the user inputs ‘m’ (medium level), 7 is returned. In all other cases 4 is returned.
  5. main(): An object of TimedScanner is created t. We also made objects of Score10, Score15 and Score20 classes. This method takes care of the inputs required for login, signup, end gaming and seeing leaderboard. Further, we take inputs for username and password as well. To display the leaderboard for a given boardsize, Collections.sort() is used. After logging in, the player gets options to choose from different board sizes and difficulty levels. An object of the GameBoard class is created. The defaultGameboard() and generateFood() methods are called via the object of the GameBoard class. The user is prompted to input directions. The direction is passed to the handleInput() method of the GameBoard object. If the user enters invalid input or does not enter any input at all in the stipulated time, the snake ends up moving in the default direction. According to the value returned by the handleInput() method, the game continues or gets over. Once the game gets over, food eaten by the snake is displayed via the getFoodCount() method of the GameBoard object. The score of the current player is updated.

1. **TimedScanner** (this class implements Runnable interface):

Attributes:

1. **scanner** : Scanner (used to accept user input in the run() method)
2. **buffer**: StringBuilder (used to temporarily store user input accepted in run() method)
3. **reading**: boolean (TRUE if a user input is required and FALSE if a user input is currently not required to be read)
4. **t**: Thread (stores reference to the thread created from the TimedScanner class. t is built in the TimedScanner constructor itself and taking reference of the new object being created for which the constructor is called. Thread t is started in the constructor.)

Methods:

* 1. public String nextLine(int ): changes the value of attribute **reading** to TRUE indicating that an input is required from the user.

IF the integer passed as argument is positive and non-zero, a countdown timer in seconds is implemented from the passed number to 0 while simultaneously giving the user a chance to enter input through the running Thread **t**. If the user inputs while the countdown timer is still going on, the trimmed inputted string is returned from the method. If the user does not provide any input before the timer ends, null is returned.

ELSE, the program waits till the user has inputted a string which contains characters other than the default delimiters. This user inputted string is trimmed and returned.

**Instruction Manual**

Our game has three board sizes available: 10x10, 15x15, 20x20.

For each board size, we have three levels, namely easy, medium and hard.

The score points for every unit of food consumed and time for user input are categorized on the basis of difficulty as:

| Difficulty Level | Score Points | Time for input (in secs) |
| --- | --- | --- |
| Easy | 5 | 10 |
| Medium | 10 | 7 |
| Hard | 15 | 4 |

1. The user gets a choice to login, signup, end gaming or see leaderboard by inputting ‘l’, ‘s’, ‘e’ and ‘w’ respectively.
2. If the user decides to see the leaderboard, (s)he further gets options to view the leaderboard for board sizes 10x10, 15x15, 20x20 by inputting ‘a’, ‘b’ and ‘c’ respectively. Upon inputting an invalid input, the user is prompted to select board size for leaderboard again. After the leaderboard has been displayed, along with the name and scores of different players, the user is redirected to step 1.
3. If the user decides to end gaming, the program stops running.
4. If the user decides to login, (s)he will be prompted to enter username. In case the user does not exist, the user will be suggested to sign up and will be given all four options available in step 1. If the user opts for signup in step 1, (s)he will have to enter a non-existent username and a new password. If the user enters an existing username, (s)he will be redirected to step 1.
5. Once a username has been entered for login or signup, the user will be asked to enter password. If the password is wrong (in case of login), the user will be notified and will again go back to the options available in step 1.
6. After a successful login/signup, the user will be given a choice to logout or play a game by entering ‘l’ or ‘p’ respectively.
7. If the user decides to logout, (s)he will be redirected to step 1.
8. If the user decides to play a game, (s)he will be asked to select a boardsize from the available three, and will be prompted to do so repeatedly if (s)he enters a different boardsize from those available in the options.
9. After the selection of an appropriate board size, the user gets a choice to select 3 different levels, namely easy, medium and hard by entering ‘e’, ‘m’ and ‘h’ respectively.
10. After the selection of an appropriate difficulty level, the game starts and the board is displayed, with the snake at the center of the board. The body of the snake is denoted by \* and its head in the initial board is represented as >. The head of the snake determines the direction of its default movement.
11. The user can input ‘>’ for moving to the right, ‘^’ for moving upwards, ‘<’ for moving leftwards and ‘d’ for moving downwards. In case the user inputs an invalid input, presses ‘enter’ or does not enter any input in the stipulated time (according to the chosen difficulty level), the snake ends up moving in the default direction (which is the direction of its head).
12. Food is denoted by ‘@’. The user can repeatedly input different directions to reach the food. Once the snake eats the food (i.e, head of the snake collides with the food unit), the user gets points (according to the chosen difficulty level). The length of the snake is also increased by 1. After the food gets eaten, a new food unit is generated at a random position on the game board.
13. In case the snake ends up colliding with the wall or with itself, the game gets over and the score for the current game is displayed.
14. The user will get an option to logout or play a game (step 4).
15. If the user decides to logout, (s)he gets all four options specified in step 1.

NOTE: At each stage where the user is expected to input a single character and if the user ends up inputting a multiple character string, only the first character of the trimmed inputted string is considered as input and the rest of the string is ignored.

All string inputs are trimmed before further usage in the program.

**Analysis of OOP Principles**

1. **Striving for loosely coupled designs between objects that interact.**

This principle is not incorporated in our code. In our code there is high coupling between the classes. A few places in code which indicate high coupling: The classes GameBoard, Snake, Food are highly coupled. The functions in GameBoard class directly use functions in Snake class and return values of functions like CollisionWithWall() of Snake class, etc. decide at various stages the fate of the ongoing game, hence they are highly coupled and any change made to the Snake class would demand the GameBoard class to change as well. Similarly, the classes Food and GameBoard are highly coupled as the Food classe uses the GameBoard object to create the coordinates of the new food on the board. The Snake class uses the coordinates of food stored in the Food class object for features like checking if the snake has eaten a food on the gameboard. Hence a structural change made in any of these classes force similar changes in the other classes indicating high coupling.

Drawback:

* We cannot easily extend the functionality of any of these classes without making changes to other classes.

1. **Composition over inheritance**

This OO Principle is incorporated in our code. The GameBoard class holds references of Snake class and Food class. Various methods of the GameBoard class including generateFood(), defaultGameboard(), updateGameboard() and handleInput() make use of the Food and Snake objects to carry out their respective functionalities.

Advantage:

* This way, we can use references of both classes simultaneously unlike inheritance where we cannot inherit both these classes simultaneously.
* In case we had programmed to an interface and made concrete classes out of it for different varieties of snake, composition would benefit us over inheritance.
* Inheritance is compile time while composition is runtime.

1. **Classes should be open for extension but closed for modification:**

This principle is not incorporated in our code. In the main() method of Game class, we have incorporated the functionality of letting the user choose from 3 given board sizes and 3 given difficulty levels. This choice is hardcoded in our game.

Drawbacks:

* If we wish to increase the options for board sizes and difficulty levels, we will have to modify the code. We cannot simply extend the given code because we have hard coded these functionalities.

If we had rather handled these choices given to the user in a separate class, then for adding more boardsizes or difficulty levels in our game we could have simply extended that separate class without modifying our code in the main() function.

1. **Depend on abstraction, do not depend on concrete classes.**

This OO Principle is not incorporated into our code. We have used concrete classes instead of interfaces and abstract classes. This limits certain functionalities of our code.

Drawbacks:

* The initial size of the snake is fixed in our game. It cannot be changed without thorough modification.
* The initial position of the snake is fixed to the center of the board, according to our code. If we had used an interface, we could have had different types of default snake positions, which we could have used accordingly.
* Currently, the snake dies if it collides with a wall. If we wanted to implement a functionality which allows the snake to reappear from the other side of the wall. The code will have to be heavily modified to add this functionality.

**Analysis of Creational Design Pattern**

Our project is following the creational design pattern.

When we create an object of the GameBoard class, the parameter ‘boardsize’ is passed, which determines the size of the board. Currently, we have implementations only for the board sizes 10x10, 15x15 and 20x20. But, if we wish to extend this to other sizes, this can be implemented in our code without making any significant modifications to the code. It can be achieved just by passing appropriate parameters for the board size.