**DOKUZ EYLUL UNIVERSITY**

**ENGINEERING FACULTY**

**DEPARTMENT OF COMPUTER ENGINEERING**

**CME1252 PROJECT BASED LEARNING – II**

**PROGRESS REPORT**

**PROJECT – II**

**Chain Project**

**by**

**YASAMIN VALISHARIATPANAHI**

**BAHADIR SEMİH KORKMAZ**

**TALHA MUSTAFA ANTEP**

**ABDULLAH DEMİRCİ**

**ELİF DORUK**

**GÖKBERK SARI**

**Lecturers**

**Prof. Dr. Alp KUT**

**Dr. Öğr. Üyesi Özge KART TOKMAK**

**29.05.2023**

# CHAPTER ONE

PROGRESS DESCRIPTION

the project "Chain" involves several key stages. The initial phase consists of project planning and design, where the goal and scope of the project are defined. The objective is to create a number game where players construct number chains to achieve the highest score. Game elements, such as the board, numbers (1-4), plus signs, chains, and the table for accumulated chains, are identified.

Next, the game initialization process is implemented. A 10x16 board is created using a 2D array, and the number area is filled with random numbers based on a specified probability. Additionally, players are given the option to set the random number seed.

The game loop and user input handling are then implemented. A loop is created that continues until the player decides to end the game. User input is processed, including cursor movement, insertion/removal of plus signs, and ending the round or the game. Actions performed by the player are validated to ensure they adhere to the rules for constructing chains.

The chain construction and validation process is implemented next. Players are able to construct chains by inserting plus signs into suitable places on the board. The constructed chain is then validated for correctness and adherence to the chain rules. The score of the chain is calculated based on the number of elements, and the player's score is updated accordingly. The correct chain is added to the table for accumulation.

The game screens are updated and displayed after each user action. This includes the board, round number, player's score, and the accumulated chains in the table. Appropriate messages are displayed in case the chain is incorrect or the game is over.

An end-of-game and high score table feature is implemented. The player can exit the game by pressing the 'E' key. The default high score table is loaded from a file (e.g., "highscore.txt") into a Doubly Linked List. At the end of the game, the high score table is displayed in descending order. If the player achieves a new high score, it is updated in the table.

Testing and debugging are carried out to ensure the game functions correctly and provides the expected gameplay experience. Various scenarios and user inputs are tested to identify and fix any issues or errors encountered.

After testing, the code is refined and enhanced. A review is conducted to improve readability, efficiency, and maintainability. Additional features or improvements may be added to enhance the gameplay experience.

Finally, a final review of the project is performed, ensuring all requirements are met and the code is properly organized. The project files are packaged, and the final version is delivered.

# CHAPTER TWO

TASK SUMMARY

## Completed Tasks

YASAMIN VALISHARIATPANAHI:

I completed the initial evaluation by implementing the necessary classes for the game, such as the Chain class, SingleLinkedList, DoubleLinkedList, and their respective nodes. I also created the main class to coordinate the game's functionality. One of the interactive features I added was allowing the player to input a seed value to determine their initial number of seeds, which were randomly generated. Additionally, I developed a user-friendly menu system with three options: starting the game, displaying the highscore table, and exiting the game. To personalize the game experience, I included a name prompt at the beginning to capture the player's name, which later appeared on the highscore table.

During the collaboration phase, I worked closely with another team to refine the player's movement mechanics. Together, we identified and fixed several errors and made necessary adjustments to the code, ultimately improving the gameplay experience. We also decided to initiate the game by simply pressing the enter key for convenience.

Visually, I designed the game screen and menu to be visually appealing and intuitive. The screen included all the necessary elements for an immersive gaming experience, while the menu allowed for smooth navigation through different options. Throughout the collaboration with two groups, we integrated code contributions from all team members, leveraging each person's expertise to create a cohesive and efficient game.

In terms of data management, I prepared the Node, SingleLinkedList, and DoubleLinkedList classes, which were crucial for organizing and manipulating data structures within the game. They facilitated tasks such as storing game data and managing linked lists efficiently. To track the game's progress, I implemented a status system that provided real-time feedback on the current round, score, seed value, highscore table, and game over conditions.

To introduce an element of randomness, I enabled players to input a seed value of their choice, which generated random initial seed values for unpredictable gameplay. Additionally, I took responsibility for implementing the highscore table functionality. After each game session, players were presented with the highscore table, showcasing the top scores and player names. I created a file where the scores and names were stored, allowing for future reference and comparison. The menu also featured an option to view the highscore table at any time, adding a competitive aspect to the game and enabling players to track their progress over time.

In conclusion, I successfully completed the tasks by designing and implementing the game screen and menu, integrating code contributions from multiple teams, creating essential classes for data management, establishing a comprehensive status system, and improving the player's movement mechanics through collaboration and iteration. I also implemented the highscore table functionality, allowing players to record their achievements and compare their scores with others.

BAHADIR SEMİH KORKMAZ: I created the base of the code. I had the codes from other members of the group merged. I created a 2-dimensional array. And I called it gameMap. This gameMap is constantly printing in the game loop. I helped create the menu and did the "How to play" part. I made cursor movements with the help of a class named "Cursor" and arrow keys. I wrote the codes that help to put "+" with the gameMap "Space" key. You can only put these '+'s next to the last element. I wrote the codes to exit the game by pressing the 'E' key. I made the "chain" appear during the game. And you can see the length of this "chain". I checked the validity of "Chain" (+1 , -1, length). I wrote the codes to get the "chain" set up properly.

TALHA MUSTAFA ANTEP:

I created data structures and classes and then created the board by seed and random. I also created chain structure and prepare the poster. Finally I helped my friends on the other subject.

ABDULLAH DEMİRCİ:

I created a multilinkedlist class and chain table.

ELİF DORUK:

Draft class design, writing highscore class codes.

GÖKBERK SARI:

Player movement, key assignments.

## Incomplete Tasks: Reasons and Explanations

To address the issue of names and scores not being properly synchronized in the high score table during sorting, several attempts were made to rectify the problem, but none proved successful. Despite multiple revisions to the code, the sorting procedure still failed to align the names with their corresponding scores accurately.

## Additional Improvements

YASAMIN VALISHARIATPANAHI:

In addition to the mentioned progress, I have made an improvement to the highscore table functionality by asking the player for their name. At the beginning of the game, I prompt the player to enter their name, which will be used to identify them in the highscore table. This personalization adds a meaningful touch to the highscore table, as players can now see their own name associated with their scores. It enhances the sense of achievement and recognition, making the gameplay experience more engaging and rewarding. By including players' names in the highscore table, it creates a stronger connection between the player and their accomplishments. This improvement not only displays the achieved scores but also adds a personal touch by including the names of the players who achieved those scores. It motivates players to compete and strive for higher rankings on the highscore table, fostering a competitive spirit and encouraging replayability.

BAHADIR SEMİH KORKMAZ: I made the updated "chain" appear directly on the game screen every time you put a '+'(Live Chain). I also showed the length(Length). I did the "How to play" part in the menu. I made our name appear during the game.

TALHA MUSTAFA ANTEP:

At the start of the game, I ask the player to input their name, which will be used to identify them in the highscore table.

ABDULLAH DEMİRCİ:

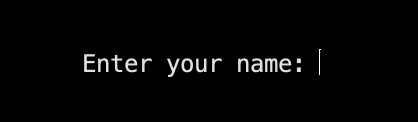
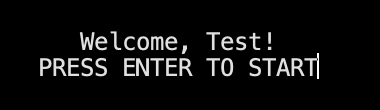
I didn’t make any additinioal improvement.

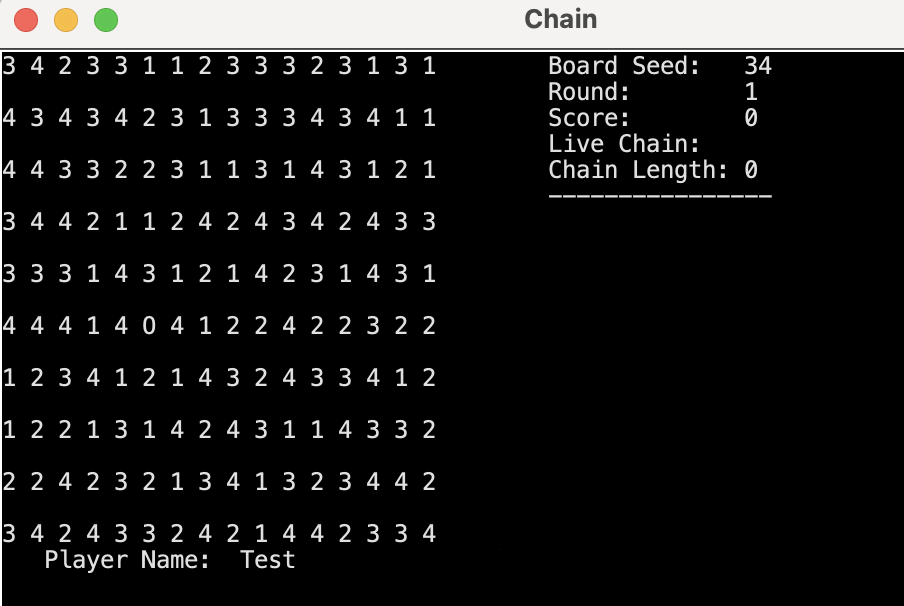
# CHAPTER THREE

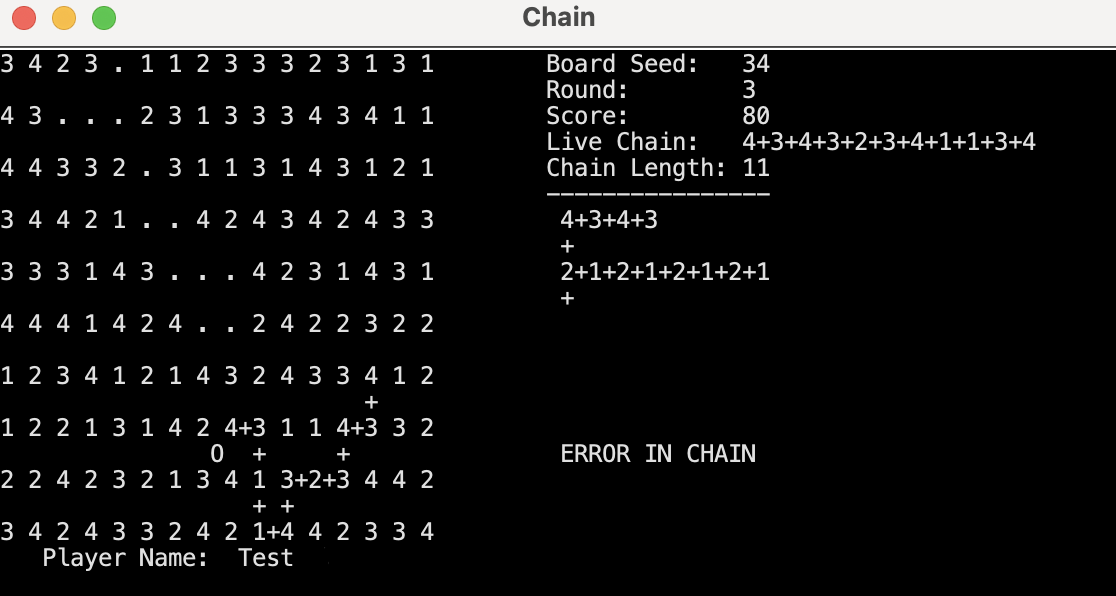
EXPLANATION of algorıthms

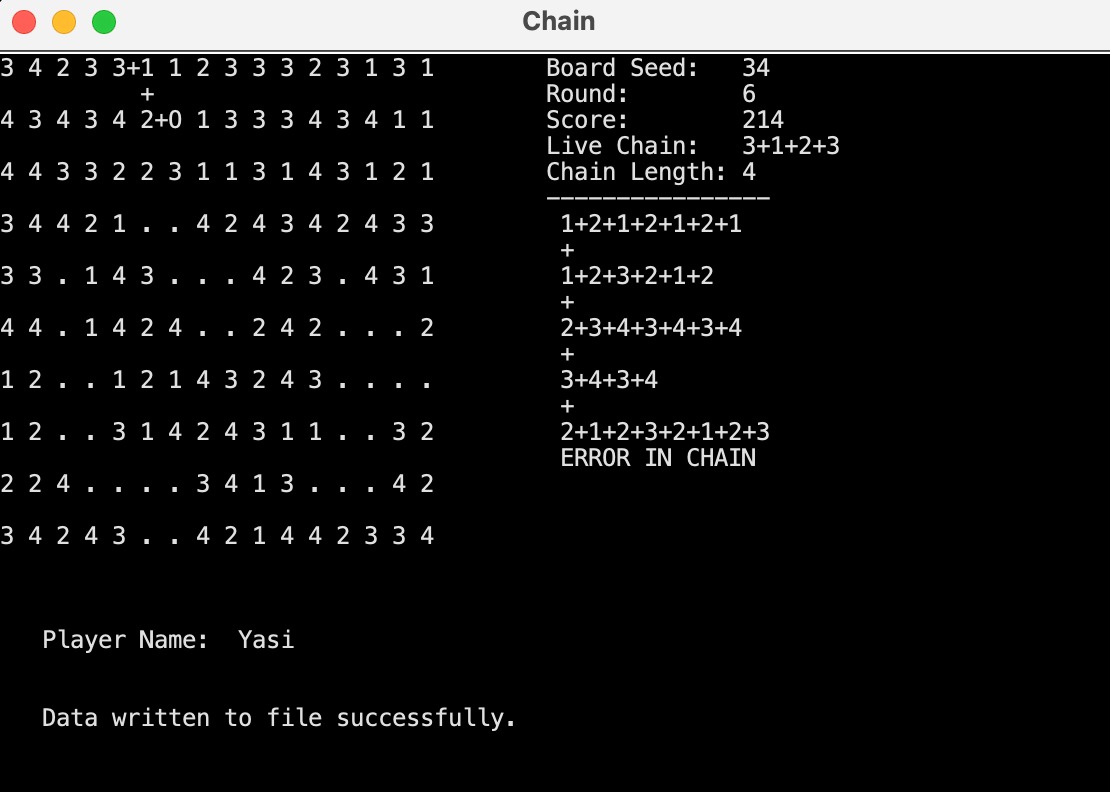
## Screenshots

* *







## Functions

Function: **main**

The **main** function serves as the entry point for the program. It creates an instance of the **Chain** class.

Class: **Node**

Attributes:

* **data**: Represents the data stored in the node.
* **link**: Represents the reference to the next node in a linked list.
* **prev**: Represents the reference to the previous node in a doubly linked list.
* **next**: Represents the reference to the next node in a doubly linked list.

Constructor:

* **Node(Object dataToAdd)**: Initializes a new **Node** object with the given data. It sets the **data** attribute to the provided value and initializes the **link**, **prev**, and **next** attributes to **null**.

Getters and Setters:

* **getData()**: Retrieves the data stored in the node.
* **setData(Object data)**: Sets the data of the node to the provided value.
* **getLink()**: Retrieves the reference to the next node in a linked list.
* **setLink(Node link)**: Sets the reference to the next node in a linked list.
* **getPrev()**: Retrieves the reference to the previous node in a doubly linked list.
* **setPrev(Node prev)**: Sets the reference to the previous node in a doubly linked list.
* **getNext()**: Retrieves the reference to the next node in a doubly linked list.
* **setNext(Node next)**: Sets the reference to the next node in a doubly linked list.

The **Node** class represents a node used in linked lists, both singly and doubly linked. It contains attributes to store data and references to the next and previous nodes. The class provides methods to access and modify the data and node references.

Class: **SingleLinkedList**

Attributes:

* **head**: Represents the reference to the first node in the linked list.

Methods:

* **size()**: Calculates and returns the number of nodes in the linked list.
* **add(Object dataToAdd)**: Adds a new node with the provided data to the end of the linked list.
* **sortAdd(Object dataToAdd)**: Adds a new node with the provided data to the linked list in a sorted manner based on ascending order.
* **delete(Object dataToDelete)**: Deletes nodes in the linked list that match the provided data.
* **display()**: Prints the data of all nodes in the linked list.
* **search(Object item)**: Searches for the presence of the provided item in the linked list and returns a boolean value indicating if it is found.
* **isEmpty()**: Checks if the linked list is empty and returns a boolean value.
* **getDataFromIndex(int index)**: Retrieves the data from the node at the specified index in the linked list.

The **SingleLinkedList** class represents a singly linked list data structure. It provides methods to perform common operations on the linked list, such as adding nodes, deleting nodes, searching for data, and retrieving data from specific positions. Additionally, it includes methods to calculate the size of the linked list and check if it is empty. The class supports sorted insertion of nodes based on ascending order. The implementation assumes that the data in the linked list is of type **Object**.

Class: **DoubleLinkedList**

* Description: Represents a doubly linked list data structure.

Constructor:

* **DoubleLinkedList()**: Initializes an empty doubly linked list.

Methods:

* **getHead()**: Returns the head node of the linked list.
* **setHead(Node node)**: Sets the head node of the linked list to the specified node.
* **getTail()**: Returns the tail node of the linked list.
* **setTail(Node node)**: Sets the tail node of the linked list to the specified node.
* **size()**: Calculates and returns the size (number of nodes) in the linked list.
* **get(int index)**:
  + Retrieves the element at the specified index in the linked list.
  + Throws an **IndexOutOfBoundsException** if the index is out of bounds.
* **set(int index, Object element)**:
  + Sets the element at the specified index in the linked list to the given value.
  + Throws an **IndexOutOfBoundsException** if the index is out of bounds.
* **getNodeAtIndex(int index)** (private):
  + Retrieves the node at the specified index in the linked list.
  + Throws an **IndexOutOfBoundsException** if the index is out of bounds.
* **size()**: Returns the number of nodes in the linked list.
* **add(Object dataToAdd)**: Adds a new node with the specified data to the end of the linked list.
* **add(String dataToAdd)**: Adds a new node with the specified data to the front of the linked list, maintaining ascending order based on string comparison.
* **delete(Object dataToDelete)**: Deletes the node containing the specified data from the linked list.
* **search(Object searchData)**: Searches for a node containing the specified data and returns the node if found, or null if not found.
* **display1()**: Prints all the items in the linked list from head to tail.
* **display2()**: Prints all the items in the linked list from tail to head.
* **sortDescending()**: Sorts the linked list in descending order based on the data stored in each node.

Class: **MultiNode**

* Description: Represents a node in a multi-level linked structure.

Attributes:

* **layer**: An object representing the layer value for the node.
* **down**: A reference to the next node in the down direction (vertical).
* **right**: A reference to the next node in the right direction (horizontal).

Methods:

* **MultiNode(Object dataToAdd)**: Constructor method that initializes a **MultiNode** object with the given **dataToAdd**.
* **getLayer()**: Getter method that returns the layer value of the node.
* **setLayer(Object layer)**: Setter method that sets the layer value of the node.
* **getDown()**: Getter method that returns the reference to the next node in the down direction.
* **setDown(MultiNode down)**: Setter method that sets the reference to the next node in the down direction.
* **getRight()**: Getter method that returns the reference to the next node in the right direction.
* **setRight(MultiNode2 right)**: Setter method that sets the reference to the next node in the right direction.

The **MultiNode** class represents a node in a multi-level linked structure. Each node contains a **layer** attribute that stores the layer value of the node. The **down** attribute is used to link the node to the next node in the down direction, while the **right** attribute is used to link the node to the next node in the right direction.

Class: **MultiLinkedList**

* Description: Represents a multi-level linked list structure.

Attributes:

* **head**: The head node of the multi-level linked list.

Methods:

* **addLayer(Object dataToAdd)**: Adds a new layer to the multi-level linked list with the specified **dataToAdd**. If the **head** is null, it creates a new **MultiNode** and assigns it as the **head**. Otherwise, it traverses the existing layers until it reaches the last layer and adds a new layer after it.
* **addItem(Object Category, Object Item)**: Adds a new item to the multi-level linked list under the specified **Category**. It searches for the layer node with the given **Category** value and adds a new **MultiNode2** containing the **Item** to its **right** reference. If the **right** reference is null, it creates a new **MultiNode2** and assigns it as the **right** reference. Otherwise, it traverses the existing **right** nodes until it reaches the last node and adds a new node after it.
* **display()**: Displays the content of the multi-level linked list. It iterates through each layer and prints the items stored in the **right** references of the nodes.

The **MultiLinkedList** class represents a multi-level linked list structure, where each layer is represented by a **MultiNode** object. Each layer node can have multiple items represented by **MultiNode2** objects linked through the **right** references. The class provides methods for adding layers and items to the multi-level linked list and displaying its contents.

Class: **Cursor**

* Description: Represents a cursor with an X and Y position.

Constructor:

* **Cursor(int posX, int posY)**: Initializes a cursor with the specified X and Y positions.

Methods:

* **getX()**: Returns the X position of the cursor.
* **getY()**: Returns the Y position of the cursor.
* **setX(int input)**: Sets the X position of the cursor to the specified input value.
* **setY(int input)**: Sets the Y position of the cursor to the specified input value.

The **Cursor** class has two attributes: **positionX** and **positionY**, which represent the X and Y coordinates of the cursor, respectively.

Class: **ChainTable**

* Description: Represents a chain table.

Attributes:

* **chaintable**: An instance of the **MultiLinkedList** class used to store the chain table.
* **a**: An integer variable used as a reference for adding layers to the chain table.
* **b**: An integer variable used as a reference for adding items to the chain table.

Methods:

* **construtTable(SingleLinkedList chain)**: Constructs the chain table based on the provided **chain** object.

The **ChainTable** class includes a static **chaintable** variable of type **MultiLinkedList**, which is used to store the chain table. It also has two static integer variables, **a** and **b**, initialized with values 0 and 50, respectively.

The **construtTable** method takes a **chain** object of type **SingleLinkedList** as input and constructs the chain table based on its data. It starts by adding a layer with value **a** to the **chaintable**, followed by adding the first item from the **chain**. Then, it iterates over the remaining items in the **chain** and adds a "+" separator and the respective item to the **chaintable** within the **a** layer.

After adding all items from the **chain** to the **chaintable** within the **a** layer, it adds a layer with value **b** to the **chaintable** and appends a "+" separator. Finally, it calls the **display** method of the **chaintable** to print the constructed chain table.

Class: **Chain**

* Class Variables:
  + **cn**: An instance of the Enigma Console class used for console input and output.
  + **ct**: An instance of the Enigma TextWindow class used for text manipulation.
  + **chain**: An instance of the SingleLinkedList class to store the chain of numbers in the game.
  + **names**: An instance of the DoubleLinkedList class to store the names of players for high scores.
  + **scores**: An instance of the DoubleLinkedList class to store the scores of players for high scores.
  + **THREADTIME**: An integer variable representing the thread time in the game.
  + **SEED**: An integer variable representing the seed value for random number generation.
  + **score**: An integer variable representing the current score in the game.
  + **playerName**: A string variable representing the name of the player.
* Methods:
  + **clear()**: Clears the console by setting all characters to empty spaces.
  + **elementFinder()**: Checks if an element with given coordinates exists in a 2D array.
  + **loadScoresFromFile(filename)**: Loads high scores from a file and stores them in the names and scores DoubleLinkedLists.
  + **saveScoresToFile(filename)**: Saves high scores to a file from the names and scores DoubleLinkedLists.
  + **highscoreSort(names, scores)**: Sorts the names and scores DoubleLinkedLists in descending order based on scores.
  + **getPlayerName()**: Returns the name of the player.
  + **getScore()**: Returns the current score in the game.
  + **Chain()**: Constructor method that initializes the Chain game.

## Algorithms and Solution Strategies

YASAMIN VALISHARIATPANAHI:

To load scores from a file, I begin by opening the file specified by the given filename using a File object. I then create a Scanner to read the file's contents. While there are more lines in the file, I split each line into name and score parts, trim any whitespace, and combine them into an entry string. I add this entry to the highScoreTable linked list. Finally, I close the scanner.

To save scores to a file, I create a BufferedWriter using the given filename to write to the file. I traverse the highScoreTable linked list using a Node variable, starting from the head. For each node, I retrieve the entry string and write it to the file. After writing each entry, I add a new line character to separate them. Once all entries are written, I close the writer.

To display high scores on the screen, when the game ends or when the player selects to view high scores in the menu, I traverse the highScoreTable linked list using a Node variable, starting from the head. For each node, I retrieve the entry string and print it on the screen. After printing each entry, I move to the next node. This way, the high scores are displayed on the screen for the player to view.

To sort the double linked list in descending order based on scores, I first check if the list is empty or contains only one element. If so, I don't need to sort it and return. I then initialize a boolean variable called "swapped" to track swaps during the sorting process. I use a do-while loop to repeat the sorting until no more swaps are made. Within the loop, I traverse the list using a current reference and compare each node's score with the next node's score. If they are in the wrong order, I swap their positions using a helper function. After each swap, I set the "swapped" variable to true. The loop continues until no more swaps are made. By following this algorithm, the double linked list is sorted in descending order based on scores.

BAHADIR SEMİH KORKMAZ: Our code has a gameLoop and this loop controls the game. I make the game map by navigating the screen in a 2D array with a for loop. I control the cursor with the help of a class named "Cursor" and the arrow keys. I have set 3 main rules of chain control in our game. First, you can put the first '+' anywhere you want(with check of horizontal or vertical). Secondly, one side of the '+' you put must be the last element of the "chain". Third, both sides of the '+' cannot be in the "chain". I store the positions of the "chain" elements in an array called numberPositions, and the positions of the '+'s in an array called plusPositions. In this way, I can distinguish the elements from each other. This makes it easier for me to control the "Chain". In the game, I print the "chain" with the help of Singlelinkedlist.

TALHA MUSTAFA ANTEP:

When printing the game board, I initially asked the user if they wanted to view the highscore table. Then, using the random library and for loops, I created a game field with dimensions of 19\*31 and displayed it on the screen. To add plus signs to the board, we utilized the provided code and made improvements.

Whenever the Enter key was pressed, the game added the chain to the table. To achieve this, we first added the chain to a singly linked list (SLL) and controlled it using various functions. We also enforced game rules, such as ensuring the chain length was greater than 4 and that the difference between elements was 1, which we checked using if statements.

Furthermore, we converted the SLL to a multi-level linked list (MLL) in order to print the table, and utilized the display function to add the chain to the board.

ABDULLAH DEMİRCİ:

I created a mutlilinkedlist class and it’s node class. I created chain Table class and constructable method which take the chain as the parameter.Firstly, I created two integer variable a and b. Then, I add the first layer and first item of the chain. Then, I added the second and before the last item and + sign. Then , I added the last item and + sign.Then, I called the dispay method.

ELİF DORUK:

1. I assign the size of the `scores` and `names` lists to the `size` variable.

2. I use two loops to compare scores and names. The outer loop covers indexes from 0 to `size-1`, and the inner loop covers indexes after the index in the outer loop.

3. In each iteration of the outer loop, we take the node (`currentNode`) in the current index and assign the next node (`nextNode`) to it. I also define `minNode` and its previous node (`minPrevNode`) to keep track of the node with the smallest score.

4. In each iteration of the inner loop, i compare the score of `nextNode` with the score of `minNode`. If `nextNode` has a larger score, i update `minNode` and `minPrevNode` to `nextNode` and `currentNode`.

5. If the score of `nextNode` is equal to the score of `minNode`, i compare the names. If `name` (name corresponding to index `j` in `names list) is smaller than `name` (name corresponding to index i in `names list) , `minNode` and `minPrevNode` will be changed to `nextNode` and `currentNode` as i update.

6. When the inner loop is complete, the node with the smallest score and name (`minNode`)

is found.

7. To swap scores, we swap values ​​between `currentNode` and `minNode`. 8. To swap names, index the `minNode` in the `names` list.

We replace the corresponding names using (`names.indexOf(minNode)`) and the index `i`.

9. The outer loop repeats this process `size-1` times, so that the biggest scores are placed in the correct position in the previous iterations.

As a result, this algorithm compares lists of `scores` and `names` with each other, ranking high scores by score and name. In each iteration of the loops, sorting is performed by finding the node with the largest score and name. This algorithm uses a variation of the selection sort algorithm and performs an optimized sorting on the given DoubleLinkedList structure.

GÖKBERK SARI:

In highscore table part, I tried to get names and their scores line by line and compare them.

# CHAPTER FOUR

PROBLEMS ENCOUNTERED

YASAMIN VALISHARIATPANAHI:

During the project, I encountered several problems, some of which were related to the challenges posed by online education. One major issue was the loss of motivation among some students. This affected our team's ability to collaborate effectively, as some team members were less responsive and engaged in the project. Despite these difficulties, we managed to work on the project as best as we could, with only a few team members actively participating. Another setback I faced was my own health-related problem. Unfortunately, I fell ill and had to be hospitalized, which significantly impacted my ability to contribute to the project. Despite my illness, I made efforts to support the team by preparing the necessary PowerPoint presentation and report, providing guidelines for my teammates to fill in their parts. Fortunately, in terms of code development, I did not encounter many problems after the initial evaluation. However, the lack of active collaboration from some team members and my own health issues posed significant challenges throughout the project. Despite these obstacles, I remained determined to contribute to the best of my ability and complete the project to the best of our team's collective efforts.

BAHADIR SEMİH KORKMAZ: At first, I thought I could control the "chain" with the help of a nested for loop of the entire map, but then I realized that it would be difficult to check for the false '+'s of the discrete. That's why I made sure that '+' cannot be put in the wrong place. This way I don't have to check for remote '+'s. We had trouble communicating after the groups merged, and I think it was very wrong to merge the groups. We had some issues with HighScoreTable.

TALHA MUSTAFA ANTEP:

I encountered some difficulties while creating the chain and checking the numbers around the plus sign. Additionally, I struggled a bit with understanding and getting familiar with doubly linked lists (DLL) and multi-level linked lists (MLL) since I recently learned about them.

ABDULLAH DEMİRCİ:

I encountered an error when I created the chain table , I was taken “chain is empty” because when I created the contruct table, I created a new single linked list so it’s size is zero.But then, I take the chain as parameter , I fixed that error.

ELİF DORUK:

Although it was not my task, I tried to set up the chain algorithm. We helped Gökberk in this regard. But we could not establish an efficient algorithm.

GÖKBERK SARI:

While generating chain, I couldn't find the right algorithm. As well as highscore table.

# CHAPTER FIVE

conclusıon

In conclusion, the Chain project aimed to develop an engaging number game where players construct chains to accumulate the highest score. The game was played on a 10x16 board, consisting of numbers (1-4) and plus signs. By strategically placing plus signs and creating chains with adjacent numbers, players aimed to maximize their scores.

Throughout the development process, we successfully implemented the game's core elements, including the game board, numbers, plus signs, and chain rules. The player had the ability to navigate the board using cursor keys, insert or remove plus signs with the space key, and end rounds by pressing enter. The game allowed for the construction of chains, which needed to adhere to specific rules, such as maintaining a difference of 1 between adjacent numbers and having a minimum length of four elements.

During gameplay, the constructed chains were accumulated in the table, contributing to the player's overall score. The game offered the flexibility to set a random number seed, ensuring varied gameplay experiences.

To provide a satisfying gameplay loop, each round required the player to construct a single chain. Upon completion, the chain was evaluated for correctness. If errors were present or the chain didn't meet the requirements, the game would end with the player's current score. However, for correct chains, the score was calculated based on the number of elements in the chain, added to the player's score, and the next round began.

The game allowed players to exit by pressing the E key, and upon completion, the high score table was displayed in descending order. This provided a competitive element and allowed players to compare their achievements with others.

Despite the challenges encountered, such as the difficulty in team collaboration and personal health issues, we persevered and completed the project to the best of our abilities. By overcoming these obstacles and implementing key features, we created an enjoyable and competitive game experience.

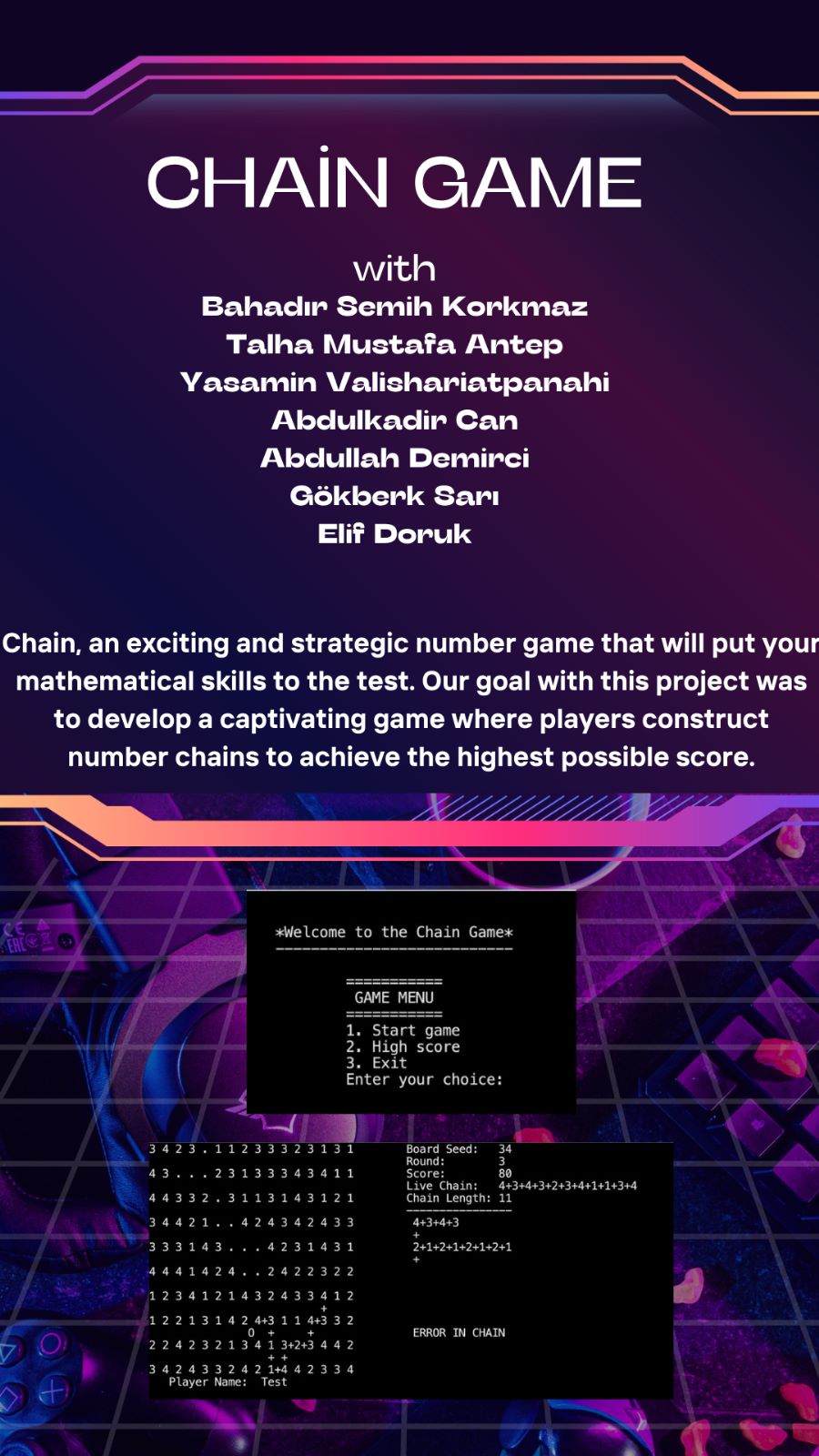
Overall, the Chain project enabled us to gain valuable experience in game development, teamwork, and problem-solving. It showcased our ability to create a well-designed and functional game, providing players with an entertaining challenge and a sense of accomplishment.

REFERENCES

* Herbert Schildt The Complete Reference Java
* Seventh Edition
* C. Thomas Wu an introduction to Object-Oriented Programming with java
* ﻿﻿httos://www.iava.com/tr/
* ﻿﻿https://dev.iava
* <https://miw.eclipse.org>
* https://pitch.com

**AppendIx A**

Poster/Web page of the Project



**AppendIx B**

Code of the Project

**public** **class** Node {

// Attributes

**private** Object data;

**private** Node link;

**private** Node prev;

**private** Node next;

// Multiple Linked List

**private** Object layer; // for every chain layer.

**private** Node down;

**private** Node right;

// Constructor

**public** Node(Object dataToAdd) {

data = dataToAdd;

link = **null**;

prev = **null**;

next = **null**;

layer = dataToAdd;

down = **null**;

right = **null**;

}

// Getters & Setters

**public** Object getData() { **return** data; }

**public** **void** setData(Object data) { **this**.data = data; }

**public** Node getLink() { **return** link; }

**public** **void** setLink(Node link) { **this**.link = link; }

**public** Node getPrev() { **return** prev; }

**public** **void** setPrev(Node prev) { **this**.prev = prev; }

**public** Node getNext() { **return** next; }

**public** **void** setNext(Node next) { **this**.next = next; }

**public** Object getLayer() { **return** layer; }

**public** **void** setLayer(Object layer) { **this**.layer = layer; }

**public** Node getDown() { **return** down; }

**public** **void** setDown(Node down) { **this**.down = down; }

**public** Node getRight() { **return** right; }

**public** **void** setRight(Node right) { **this**.right = right; }

}

**public** **class** SingleLinkedList {

// Attributes

Node head;

**public** **boolean** isEmpty() {

**if**(head == **null**) **return** **true**;

**else** **return** **false**;

}

**public** **int** size() {

**int** count = 0;

**if**(head == **null**)

System.***out***.println("linked list is empty");

**else** {

Node temp = head;

**while**(temp != **null**) {

count++;

temp = temp.getLink();

}

}

**return** count;

}

**public** **void** add(Object dataToAdd) {

**if**(head == **null**) { // list is empty

Node newnode = **new** Node(dataToAdd);

head = newnode;

} **else** {

Node temp = head;

**while**(temp.getLink() != **null**)

temp = temp.getLink();

Node newnode = **new** Node(dataToAdd);

temp.setLink(newnode);

}

}

**public** **void** sortAdd(Object dataToAdd) {

**if**(head == **null**) { // list is empty

Node newnode = **new** Node(dataToAdd);

head = newnode;

} **else** **if**((Integer)dataToAdd < (Integer)head.getData()) { // insert at front

Node newnode = **new** Node(dataToAdd);

newnode.setLink(head);

head = newnode;

} **else** {

Node temp = head;

Node previous = **null**;

**while**(temp != **null** && (Integer)dataToAdd > (Integer)temp.getData()) {

previous = temp;

temp = temp.getLink();

}

**if**(temp == **null**) { // insert at last

Node newnode = **new** Node(dataToAdd);

previous.setLink(newnode);

} **else** { // insert in between

Node newnode = **new** Node(dataToAdd);

newnode.setLink(previous.getLink());

previous.setLink(newnode);

}

}

}

**public** **void** delete(Object dataToDelete) {

**if**(head == **null**) {

System.***out***.println("Linked list is empty");

**return**;

} **else** {

// Delete nodes at the beginning of the list that match dataToDelete

**while**(head != **null** && head.getData().equals(dataToDelete)) {

head = head.getLink();

}

// Delete nodes in the middle or end of the list that match dataToDelete

Node temp = head;

Node previous = **null**;

**while**(temp != **null**) {

**if**(temp.getData().equals(dataToDelete)) {

**if**(previous == **null**) { // the matching node is the first node

head = temp.getLink();

} **else** {

previous.setLink(temp.getLink());

}

} **else** {

previous = temp;

}

temp = temp.getLink();

}

}

}

**public** **boolean** search(Object item){

**boolean** flag = **false**;

**if** (head == **null**)

System.***out***.println("linked list is empty");

**else** {

Node temp = head;

**while** (temp != **null**) {

**if** (item == temp.getData()) flag = **true**;

temp = temp.getLink();

}

}

**return** flag;

}

**public** Object getDataFromIndex(**int** index) {

**int** counter = 0;

Node temp = head;

**if**(head == **null**) {

System.***out***.println("Linked List is empty.");

} **else** {

**while**(counter != index) {

temp = temp.getLink();

counter++;

}

}

**return** temp.getData();

}

**public** **void** display() {

**if** (head == **null**)

System.***out***.println("linked list is empty");

**else** {

Node temp = head;

**while** (temp != **null**) {

System.***out***.print(temp.getData() + " ");

temp = temp.getLink();

}

}

}

}

**public** **class** DoubleLinkedList {

// Attributes

**private** Node head;

**private** Node tail;

**public** DoubleLinkedList() {

head = **null**;

tail = **null**;

}

**public** Node getHead() { **return** head; }

**public** **void** setHead(Node node) { head = node; }

**public** Node getTail() { **return** tail; }

**public** **void** setTail(Node node) { tail = node; }

**public** **int** size() {

**int** count = 0;

**if**(head == **null**) {

//System.out.println("Linked List is empty");

}

**else** {

Node temp = head;

**while**(temp != **null**) {

count++;

temp = temp.getNext();

}

}

**return** count;

}

// Get the element at the specified index

**public** Object get(**int** index) {

**if** (index < 0 || index >= size()) {

**throw** **new** IndexOutOfBoundsException("Index out of bounds: " + index);

}

Node current = getNodeAtIndex(index);

**return** current.getData();

}

// Set the element at the specified index

**public** **void** set(**int** index, Object element) {

**if** (index < 0 || index >= size()) {

**throw** **new** IndexOutOfBoundsException("Index out of bounds: " + index);

}

Node current = getNodeAtIndex(index);

current.setData(element);

}

**private** Node getNodeAtIndex(**int** index) {

**if** (index < 0 || index >= size()) {

**throw** **new** IndexOutOfBoundsException("Index out of bounds: " + index);

}

Node current = head;

**int** counter = 0;

**while** (counter < index) {

current = current.getNext();

counter++;

}

**return** current;

}

**public** **void** add(Object dataToAdd) {

// add a new Node to the end

**if**(head == **null** && tail == **null**) {

Node newNode = **new** Node(dataToAdd);

head = newNode; /\* pointing the first Node \*/

tail = newNode; /\* pointing the last Node \*/

} **else** {

Node newNode = **new** Node(dataToAdd);

newNode.setPrev(tail);

tail.setNext(newNode);

tail = newNode;

}

}

**public** **void** add(String dataToAdd) {

// add a new Node to the front

**if**(head == **null** && tail == **null**) {

Node newNode = **new** Node(dataToAdd);

head = newNode; /\* pointing the first Node \*/

tail = newNode; /\* pointing the last Node \*/

} **else** {

Node newNode = **new** Node(dataToAdd);

**if**(dataToAdd.compareTo((String) head.getData()) < 0) {

newNode.setNext(head);

head.setPrev(newNode);

head = newNode;

} **else** {

Node temp = head;

**while**(temp.getNext() != **null** && dataToAdd.compareTo((String) temp.getNext().getData()) > 0) {

temp = temp.getNext();

}

newNode.setPrev(temp);

newNode.setNext(temp.getNext());

**if**(temp.getNext() != **null**) // adding between Nodes

temp.getNext().setPrev(newNode);

**else** // adding to the end

tail = newNode;

temp.setNext(newNode);

}

}

}

**public** **void** delete(Object dataToDelete) {

**if**(head == **null**) {

//System.out.println("Linked List is empty");

} **else** {

Node temp = head;

**while**(temp != **null**) {

**if**(temp.getData().equals(dataToDelete)) {

**if**(temp == head) {

head = temp.getNext();

**if**(head != **null**) {

head.setPrev(**null**);

}

**if**(temp == tail) {

tail = **null**;

}

} **else** **if**(temp == tail) {

tail = temp.getPrev();

tail.setNext(**null**);

} **else** {

temp.getPrev().setNext(temp.getNext());

temp.getNext().setPrev(temp.getPrev());

}

**break**;

}

temp = temp.getNext();

}

**if**(temp == **null**) {

System.***out***.println("Item not found in the Linked List");

}

}

}

**public** Node search(Object searchData) {

Node temp = head;

**while**(temp != **null**) {

**if**(temp.getData().equals(searchData)) {

**return** temp;

}

temp = temp.getNext();

}

System.***out***.println("Item not found in the Linked List");

**return** **null**;

}

**public** Object getDataFromIndex(**int** index) {

**int** counter = 0;

Node temp = head;

**if**(head == **null**) {

System.***out***.println("Linked List is empty.");

} **else** {

**while**(counter != index) {

temp = temp.getNext();

counter++;

}

}

**return** temp.getData();

}

// (which prints all items in the list from head to tail)

**public** **void** display1() {

**if**(head == **null**) {

//System.out.println("Linked List is empty");

} **else** {

Node temp = head;

**while**(temp != **null**) {

System.***out***.println(temp.getData() + " ");

temp = temp.getNext();

}

System.***out***.println();

}

}

// (which prints all items in the list from tail to head)

**public** **void** display2() {

**if**(head == **null**) {

//System.out.println("Linked List is empty");

} **else** {

Node temp = tail;

**while**(temp != **null**) {

System.***out***.println(temp.getData() + " ");

temp = temp.getPrev();

}

System.***out***.println();

}

}

}

**public** **class** MultiLinkedList {

// Attributes

**private** Node head;

**public** **void** addLayer(Object dataToAdd) {

Node temp;

**if** (head == **null**) {

temp = **new** Node(dataToAdd);

head = temp;

} **else** {

temp = head;

**while** (temp.getDown() != **null**)

temp = temp.getDown();

Node newnode = **new** Node(dataToAdd);

temp.setDown(newnode);

}

}

**public** **void** addItem(Object Category, Object Item) {

**if** (head == **null**)

System.***out***.println("Add a Category before Item");

**else** {

Node temp = head;

**while** (temp != **null**) {

**if** (Category.equals(temp.getLayer())) {

Node temp2 = temp.getRight();

**if** (temp2 == **null**) {

temp2 = **new** Node(Item);

temp.setRight(temp2);

} **else** {

**while** (temp2.getNext() != **null**)

temp2 = temp2.getNext();

Node newnode = **new** Node(Item);

temp2.setNext(newnode);

}

}

temp = temp.getDown();

}

}

}

**public** **void** display() {

**int** i = 6;

**if** (head == **null**) {

// System.out.println("linked list is empty");

} **else** {

Node temp = head;

**while** (temp != **null**) {

Chain.*cn*.getTextWindow().setCursorPosition(40, i);

Node temp2 = temp.getRight();

**while** (temp2 != **null**) {

System.***out***.print(temp2.getData());

temp2 = temp2.getNext();

}

temp = temp.getDown();

i++;

}

}

}

}

**public** **class** Cursor {

// attributes

**private** **int** positionX;

**private** **int** positionY;

// ----------------------------------------------------

**public** Cursor(**int** posX, **int** posY) {

positionX = posX;

positionY = posY;

}

// ----------------------------------------------------

// getters

**public** **int** getX() { **return** positionX; }

**public** **int** getY() { **return** positionY; }

// ----------------------------------------------------

// setters

**public** **void** setX(**int** input) { positionX = input; }

**public** **void** setY(**int** input) { positionY = input; }

}

**public** **class** ChainTable {

// Attributes

**static** MultiLinkedList *chaintable* = **new** MultiLinkedList();

**static** **int** *a* = 0;

**static** **int** *b* = 50;

**public** **static** **void** construtTable(SingleLinkedList chain) {

**int** chainSize = chain.size();

*chaintable*.addLayer(*a*);

*chaintable*.addItem(*a*, chain.getDataFromIndex(0));

**for** (**int** i = 1; i < chainSize; i++) {

*chaintable*.addItem(*a*, "+");

*chaintable*.addItem(*a*, chain.getDataFromIndex(i));

}

*chaintable*.addLayer(*b*);

*chaintable*.addItem(*b*, "+");

*chaintable*.display();

*a*++;

*b*++;

}

}

**import** enigma.core.Enigma;

**import** enigma.event.TextMouseListener;

**import** java.awt.event.KeyEvent;

**import** java.awt.event.KeyListener;

**import** java.io.BufferedReader;

**import** java.io.BufferedWriter;

**import** java.io.FileReader;

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** java.util.InputMismatchException;

**import** java.util.Random;

**import** java.util.Scanner;

**public** **class** Chain {

**public** **static** enigma.console.Console *cn* = Enigma.*getConsole*("Chain");

**public** enigma.console.TextWindow ct = *cn*.getTextWindow();

SingleLinkedList chain = **new** SingleLinkedList();

DoubleLinkedList names = **new** DoubleLinkedList();

DoubleLinkedList scores = **new** DoubleLinkedList();

**private** **int** THREADTIME = 50;

**private** **int** SEED = 34;

**private** **int** score = 0;

**private** String playerName = "";

**public** TextMouseListener tmlis;

**public** KeyListener klis;

Random rnd = **new** Random(SEED);

// ------ Standard variables for mouse and keyboard ------

**public** **int** mousepr; // mouse pressed?

**public** **int** mousex, mousey; // mouse text coords.

**public** **int** keypr; // key pressed?

**public** **int** rkey; // key (for press/release)

// ----------------------------------------------------

**public** **void** clear()

{

**for**(**int** j = 0;j<30;j++) **for**(**int** i = 0;i<70;i++) ct.output(i,j, ' ');

}

**public** **boolean** elementFinder(**int**[][] arr,**int** inputX, **int** inputY , **int** size)

{

**for**(**int** i = 0; i<size;i++)

{

**if**(arr[i][0] == inputX && arr[i][1] == inputY)

{

**return** **true**;

}

}

**return** **false**;

}

**public** **void** loadScoresFromFile(String filename) {

// Reading the unsorted file using the BufferedReader class

BufferedReader br = **null**;

**try** {

br = **new** BufferedReader(**new** FileReader(filename));

String line = br.readLine().trim();

**while** (line != **null**) {

// Splitting each line into name and score

String[] parts = line.split(" ");

**if** (parts.length != 2) {

System.***out***.println("Invalid file format!");

**return**;

}

names.add(parts[0]); // names

scores.add(Integer.*parseInt*(parts[1])); // scores

line = br.readLine();

}

} **catch** (IOException e) {

System.***out***.println("Error reading file: " + e.getMessage());

**return**;

} **catch** (NumberFormatException e) {

System.***out***.println("Invalid file format: " + e.getMessage());

**return**;

} **finally** {

**if** (br != **null**) {

**try** {

br.close();

} **catch** (IOException e) {

System.***out***.println("We encountered an error: " + e.getMessage());

}

}

}

}

**public** **void** saveScoresToFile(String filename) {

BufferedWriter bw = **null**;

**try** {

bw = **new** BufferedWriter(**new** FileWriter(filename));

Node namesNode = names.getHead();

Node scoresNode = scores.getHead();

**while** (namesNode != **null** && scoresNode != **null**) {

String playerName = (String) namesNode.getData();

**int** score = (**int**) scoresNode.getData();

namesNode = namesNode.getNext();

scoresNode = scoresNode.getNext();

String entry = playerName + " " + score;

bw.write(entry);

bw.newLine();

}

*cn*.getTextWindow().setCursorPosition(3, 25);

System.***out***.println("Data written to file successfully.");

} **catch** (IOException e) {

System.***out***.println("Error writing to file: " + e.getMessage());

} **finally** {

**if** (bw != **null**) {

**try** {

bw.close();

} **catch** (IOException e) {

System.***out***.println("We encountered an error: " + e.getMessage());

}

}

}

}

**public** **void** highscoreSort(DoubleLinkedList names, DoubleLinkedList scores) {

**int** size = scores.size();

// Create a 2D array to store the names and scores together

Object[][] data = **new** Object[size][2];

// Populate the 2D array with names and scores

**for** (**int** i = 0; i < size; i++) {

data[i][0] = names.get(i);

data[i][1] = scores.get(i);

}

// Sort the 2D array based on scores in descending order

**for** (**int** i = 0; i < size - 1; i++) {

**for** (**int** j = 0; j < size - i - 1; j++) {

**if** ((**int**) data[j][1] < (**int**) data[j + 1][1]) {

// Swap rows

Object[] temp = data[j];

data[j] = data[j + 1];

data[j + 1] = temp;

} **else** **if** ((**int**) data[j][1] == (**int**) data[j + 1][1]) {

// If scores are equal, compare names

String name1 = (String) data[j][0];

String name2 = (String) data[j + 1][0];

**if** (name1.compareTo(name2) > 0) {

// Swap rows

Object[] temp = data[j];

data[j] = data[j + 1];

data[j + 1] = temp;

}

}

}

}

// Update the original lists with the sorted names and scores

**for** (**int** i = 0; i < size; i++) {

names.set(i, data[i][0]);

scores.set(i, data[i][1]);

}

}

**public** String getPlayerName() { **return** playerName; }

**public** **int** getScore() { **return** score;}

**public** Chain() **throws** Exception { // --- Constructor

**char**[][] gameMap = **new** **char**[20][32];

**int**[][] plusPositions = **new** **int**[999][2];

**int**[][] numberPositions = **new** **int**[999][2];

**int** plusCounter = 0;

**int** numberCounter = 0;

**int** round = 1;

**boolean** gameBegin = **true**;

**boolean** finish = **false**;

**boolean** menu = **true**;

Cursor cursor = **new** Cursor(10,10);

// Map Declaring

**for**(**int** j = 0; j<19;j++)

{

**for**(**int** i = 0; i<31;i++)

{

**if**(i%2==0 && j%2==0) {gameMap[j][i] = Character.*forDigit*(rnd.nextInt(4)+1, 10);}

**else**

{

gameMap[j][i] = ' ';

}

}

}

klis=**new** KeyListener() {

**public** **void** keyTyped(KeyEvent e) {}

**public** **void** keyPressed(KeyEvent e) {

**if**(keypr==0) {

keypr=1;

rkey=e.getKeyCode();

}

}

**public** **void** keyReleased(KeyEvent e) {}

};

*cn*.getTextWindow().addKeyListener(klis);

// ----------------------------------------------------

ct.setCursorPosition(10, 2);

ct.output(" \*Welcome to the Chain Game\*");

ct.setCursorPosition(10, 3);

ct.output(" ---------------------------");

Scanner inputMenu = **new** Scanner(System.***in***);

**int** choice = 0;

**while** (menu) {

clear();

ct.setCursorPosition(30, 5);

System.***out***.println("===========");

ct.setCursorPosition(29, 6);

System.***out***.println(" GAME MENU");

ct.setCursorPosition(30, 7);

System.***out***.println("===========");

ct.setCursorPosition(30, 8);

System.***out***.println("1. Start game");

ct.setCursorPosition(30, 9);

System.***out***.println("2. High score");

ct.setCursorPosition(30, 10);

System.***out***.println("3. How To Play");

ct.setCursorPosition(30, 11);

System.***out***.println("4. Exit");

ct.setCursorPosition(30, 12);

System.***out***.print("Enter your choice: ");

**try** {

choice = inputMenu.nextInt();

**switch** (choice) {

**case** 1:

menu = **false**;

**break**;

// ----------------------------------------------------

**case** 2:

clear();

ct.setCursorPosition(30, 5);

System.***out***.println("============");

ct.setCursorPosition(29, 6);

System.***out***.println(" HIGH SCORE");

ct.setCursorPosition(30, 7);

System.***out***.println("============");

ct.setCursorPosition(30, 8);

loadScoresFromFile("highscore.txt");

// Sorting the scores in descending order

highscoreSort(names, scores);

**int** k = 0;

Node namesNode = names.getHead();

Node scoresNode = scores.getHead();

**while** (namesNode != **null** && scoresNode != **null**) {

String name = (String) namesNode.getData();

namesNode = namesNode.getNext();

**int** score = (**int**) scoresNode.getData();

scoresNode = scoresNode.getNext();

// Print the name and score

ct.setCursorPosition(32, 8 + k);

*cn*.getTextWindow().output(String.*format*("%s: %d", name, score));

k++;

}

ct.setCursorPosition(0, 0);

inputMenu.next();

**break**;

// ----------------------------------------------------

**case** 3:

clear();

ct.setCursorPosition(30, 5);

System.***out***.println("===========");

ct.setCursorPosition(30, 6);

System.***out***.println("HOW TO PLAY");

ct.setCursorPosition(30, 7);

System.***out***.println("===========");

ct.setCursorPosition(30, 8);

System.***out***.println("First Plus");

ct.setCursorPosition(15, 9);

System.***out***.println("You can put the first '+' anywhere you want.");

ct.setCursorPosition(32, 11);

System.***out***.println("Others");

ct.setCursorPosition(10, 12);

System.***out***.println("Other pluses should be next to the last chain element.");

ct.setCursorPosition(9, 13);

System.***out***.println("Both sides of the plus you put in cannot be in the chain.");

inputMenu.next();

**break**;

**case** 4:

System.*exit*(0);

**break**;

// ----------------------------------------------------

**default**:

ct.setCursorPosition(30, 12);

System.***out***.println("Invalid choice. Please try again.");

**continue**; // loop back to show the menu again

}

} **catch** (InputMismatchException e) {

ct.setCursorPosition(30, 13);

System.***out***.println(" ");

System.***out***.println("Invalid input. Please enter a number.");

inputMenu.nextLine();

}

}

clear();

**while**(!finish) {

**if** (gameBegin) {

Scanner input = **new** Scanner(System.***in***);

// Ask for player's name

ct.setCursorPosition(28, 5);

System.***out***.print("Enter your name: ");

ct.setCursorPosition(45, 5);

playerName = *cn*.readLine();

ct.setCursorPosition(45, 5);

ct.output(" ");

ct.setCursorPosition(28, 4);

System.***out***.println("Welcome, " + playerName + "!");

ct.setCursorPosition(25, 5);

ct.output("PRESS ENTER TO START");

System.***in***.read();

ct.setCursorPosition(25, 5);

ct.output(" ");

clear();

gameBegin = **false**;

}

**if**(mousepr==1) { // if mouse button pressed

}

**if**(keypr==1) { // if keyboard button pressed

**if**(rkey==KeyEvent.***VK\_LEFT***)

**if**(cursor.getX()>0)

{

*cn*.getTextWindow().output(cursor.getX(), cursor.getY(), ' ');

cursor.setX(cursor.getX()-1);

}

**if**(rkey==KeyEvent.***VK\_RIGHT***)

**if**(cursor.getX()<30)

{

*cn*.getTextWindow().output(cursor.getX(), cursor.getY(), ' ');

cursor.setX(cursor.getX()+1);

}

**if**(rkey==KeyEvent.***VK\_UP***)

**if**(cursor.getY()>0)

{

*cn*.getTextWindow().output(cursor.getX(), cursor.getY(), ' ');

cursor.setY(cursor.getY()-1);

}

**if**(rkey==KeyEvent.***VK\_DOWN***)

**if**(cursor.getY()<18)

{

*cn*.getTextWindow().output(cursor.getX(), cursor.getY(), ' ');

cursor.setY(cursor.getY()+1);

}

**if**(rkey==KeyEvent.***VK\_ENTER***)

{

**boolean** error = **false**;

**if**(numberCounter <4)

{

error = **true**;

}

**for**(**int** i = 0 ; i<chain.size()-1;i++)

{

**if**(Integer.*parseInt*(chain.getDataFromIndex(i).toString())+1 == Integer.*parseInt*(chain.getDataFromIndex(i+1).toString()) || Integer.*parseInt*(chain.getDataFromIndex(i).toString())-1 == Integer.*parseInt*(chain.getDataFromIndex(i+1).toString()))

{

}

**else** {error = **true**;}

}

**if**(error)

{

*cn*.getTextWindow().setCursorPosition(40, 15);

*cn*.getTextWindow().output(String.*format*("ERROR IN CHAIN" ));

finish = **true**;

}

**else**

{

ChainTable.*construtTable*(chain);

score+=(chain.size()\*chain.size());

round+=1;

**for**(**int** i = 0;i<chain.size()\*2;i++)

{

*cn*.getTextWindow().setCursorPosition(53+i, 3);

*cn*.getTextWindow().output(String.*format*(" "));

*cn*.getTextWindow().setCursorPosition(53+i, 4);

*cn*.getTextWindow().output(String.*format*(" "));

}

**for**(**int** i = 0;i<plusCounter;i++)

{

gameMap[plusPositions[i][1]][plusPositions[i][0]] = ' ';

plusPositions[i][0] = 0;

plusPositions[i][1] = 0;

}

plusCounter = 0;

**for**(**int** i = 0;i<numberCounter;i++)

{

gameMap[numberPositions[i][1]][numberPositions[i][0]] = '.';

numberPositions[i][0] = 0;

numberPositions[i][1] = 0;

}

numberCounter = 0;

chain = **new** SingleLinkedList();

}

}

**if**(rkey==KeyEvent.***VK\_E***) finish = **true**;

**if**(rkey==KeyEvent.***VK\_SPACE***)

{

**boolean** flag = **false**;

**boolean** cursorCorrect = **false**;

**if**(cursor.getX()%2==1 && cursor.getY()%2==1);

**else** **if**(cursor.getX()%2==0 && cursor.getY()%2==0);

**else** **if**(gameMap[cursor.getY()][cursor.getX()] == '+')

{

**for**(**int** i = 0;i<chain.size()\*2;i++)

{

*cn*.getTextWindow().setCursorPosition(53+i, 3);

*cn*.getTextWindow().output(String.*format*(" "));

*cn*.getTextWindow().setCursorPosition(53+i, 4);

*cn*.getTextWindow().output(String.*format*(" "));

}

**for**(**int** i = 0;i<plusCounter;i++)

{

gameMap[plusPositions[i][1]][plusPositions[i][0]] = ' ';

plusPositions[i][0] = 0;

plusPositions[i][1] = 0;

}

plusCounter = 0;

**for**(**int** i = 0;i<numberCounter;i++)

{

numberPositions[i][0] = 0;

numberPositions[i][1] = 0;

}

numberCounter = 0;

chain = **new** SingleLinkedList();

}

**else**

{

**if**(plusCounter > 0 && cursor.getX()%2==1 && cursor.getY()%2==0 && ((elementFinder(numberPositions,cursor.getX()-1,cursor.getY(),numberCounter) && !elementFinder(numberPositions,cursor.getX()+1,cursor.getY(),numberCounter)) || (!elementFinder(numberPositions,cursor.getX()-1,cursor.getY(),numberCounter) && elementFinder(numberPositions,cursor.getX()+1,cursor.getY(),numberCounter)))) // horizontal cursorChecker

{

**if**(gameMap[cursor.getY()][cursor.getX()-1] != '.' && gameMap[cursor.getY()][cursor.getX()+1] != '.')

{cursorCorrect = **true**;}

}

**else** **if**(plusCounter > 0 && cursor.getX()%2==0 && cursor.getY()%2==1 && ((elementFinder(numberPositions,cursor.getX(),cursor.getY()-1,numberCounter) && !elementFinder(numberPositions,cursor.getX(),cursor.getY()+1,numberCounter)) || (!elementFinder(numberPositions,cursor.getX(),cursor.getY()-1,numberCounter) && elementFinder(numberPositions,cursor.getX(),cursor.getY()+1,numberCounter)))) // vertical cursorChecker

{

**if**(gameMap[cursor.getY()-1][cursor.getX()] != '.' && gameMap[cursor.getY()+1][cursor.getX()] != '.')

{cursorCorrect = **true**;}

}

**if**(plusCounter == 0) // First +

{

**boolean** cursorCorrectForFirst= **false**;

**if**(cursor.getX() == 0 && gameMap[cursor.getY()-1][cursor.getX()] != '.' && gameMap[cursor.getY()+1][cursor.getX()] != '.') {cursorCorrectForFirst = **true**;}

**else** **if**(cursor.getY() == 0 && gameMap[cursor.getY()][cursor.getX()-1] != '.' && gameMap[cursor.getY()][cursor.getX()+1] != '.') {cursorCorrectForFirst = **true**;}

**else** **if**(gameMap[cursor.getY()][cursor.getX()-1] != '.' && gameMap[cursor.getY()][cursor.getX()+1] != '.' && gameMap[cursor.getY()-1][cursor.getX()] != '.' && gameMap[cursor.getY()+1][cursor.getX()] != '.')

{cursorCorrectForFirst = **true**;}

**if**(cursorCorrectForFirst)

{

gameMap[cursor.getY()][cursor.getX()] = '+';

plusPositions[plusCounter][0] = cursor.getX();

plusPositions[plusCounter][1] = cursor.getY();

plusCounter++;

// Check vertical or horizontal?

**if**(cursor.getX()%2==1 && cursor.getY()%2==0) // horizontal

{

// First left

numberPositions[numberCounter][0] = cursor.getX()-1;

numberPositions[numberCounter][1] = cursor.getY();

chain.add(gameMap[numberPositions[numberCounter][1]][numberPositions[numberCounter][0]]);

numberCounter++;

// Next right

numberPositions[numberCounter][0] = cursor.getX()+1;

numberPositions[numberCounter][1] = cursor.getY();

chain.add(gameMap[numberPositions[numberCounter][1]][numberPositions[numberCounter][0]]);

numberCounter++;

}

**else** **if** (cursor.getX()%2==0 && cursor.getY()%2==1) // vertical

{

// First up

numberPositions[numberCounter][0] = cursor.getX();

numberPositions[numberCounter][1] = cursor.getY()-1;

chain.add(gameMap[numberPositions[numberCounter][1]][numberPositions[numberCounter][0]]);

numberCounter++;

// Next down

numberPositions[numberCounter][0] = cursor.getX();

numberPositions[numberCounter][1] = cursor.getY()+1;

chain.add(gameMap[numberPositions[numberCounter][1]][numberPositions[numberCounter][0]]);

numberCounter++;

}

}

}

**else** **if**(cursorCorrect && plusPositions[plusCounter-1][0]%2 == 0 && plusPositions[plusCounter-1][1]%2 == 1) // last plus is vertical

{

**if**(plusPositions[plusCounter-1][0] == cursor.getX() && plusPositions[plusCounter-1][1]+2 == cursor.getY() && numberPositions[numberCounter-1][0] == cursor.getX() && numberPositions[numberCounter-1][1] == cursor.getY()-1) // 1 x0 y+ down

{

flag = **true**;

numberPositions[numberCounter][0] = cursor.getX();

numberPositions[numberCounter][1] = cursor.getY()+1;

}

**else** **if**(plusPositions[plusCounter-1][0] == cursor.getX() && plusPositions[plusCounter-1][1]-2 == cursor.getY() && numberPositions[numberCounter-1][0] == cursor.getX() && numberPositions[numberCounter-1][1] == cursor.getY()+1) // 2 x0 y- up

{

flag = **true**;

numberPositions[numberCounter][0] = cursor.getX();

numberPositions[numberCounter][1] = cursor.getY()-1;

}

**else** **if**(plusPositions[plusCounter-1][0]+1 == cursor.getX() && plusPositions[plusCounter-1][1]+1 == cursor.getY() && numberPositions[numberCounter-1][0] == cursor.getX()-1 && numberPositions[numberCounter-1][1] == cursor.getY()) // 3 x+ y+ right

{

flag = **true**;

numberPositions[numberCounter][0] = cursor.getX()+1;

numberPositions[numberCounter][1] = cursor.getY();

}

**else** **if**(plusPositions[plusCounter-1][0]-1 == cursor.getX() && plusPositions[plusCounter-1][1]+1 == cursor.getY() && numberPositions[numberCounter-1][0] == cursor.getX()+1 && numberPositions[numberCounter-1][1] == cursor.getY()) // 4 x- y+ left

{

flag = **true**;

numberPositions[numberCounter][0] = cursor.getX()-1;

numberPositions[numberCounter][1] = cursor.getY();

}

**else** **if**(plusPositions[plusCounter-1][0]+1 == cursor.getX() && plusPositions[plusCounter-1][1]-1 == cursor.getY() && numberPositions[numberCounter-1][0] == cursor.getX()-1 && numberPositions[numberCounter-1][1] == cursor.getY()) // 5 x+ y- right

{

flag = **true**;

numberPositions[numberCounter][0] = cursor.getX()+1;

numberPositions[numberCounter][1] = cursor.getY();

}

**else** **if**(plusPositions[plusCounter-1][0]-1 == cursor.getX() && plusPositions[plusCounter-1][1]-1 == cursor.getY() && numberPositions[numberCounter-1][0] == cursor.getX()+1 && numberPositions[numberCounter-1][1] == cursor.getY()) // 6 x- y- left

{

flag = **true**;

numberPositions[numberCounter][0] = cursor.getX()-1;

numberPositions[numberCounter][1] = cursor.getY();

}

}

**else** **if**(cursorCorrect && plusPositions[plusCounter-1][0]%2 == 1 && plusPositions[plusCounter-1][1]%2 == 0) // last plus is horizontal

{

**if**(plusPositions[plusCounter-1][0]+2 == cursor.getX() && plusPositions[plusCounter-1][1] == cursor.getY() && numberPositions[numberCounter-1][0] == cursor.getX()-1 && numberPositions[numberCounter-1][1] == cursor.getY()) // 1 x+ y0 right

{

flag = **true**;

numberPositions[numberCounter][0] = cursor.getX()+1;

numberPositions[numberCounter][1] = cursor.getY();

}

**else** **if**(plusPositions[plusCounter-1][0]-2 == cursor.getX() && plusPositions[plusCounter-1][1] == cursor.getY() && numberPositions[numberCounter-1][0] == cursor.getX()+1 && numberPositions[numberCounter-1][1] == cursor.getY()) // 2 x- y0 left

{

flag = **true**;

numberPositions[numberCounter][0] = cursor.getX()-1;

numberPositions[numberCounter][1] = cursor.getY();

}

**else** **if**(plusPositions[plusCounter-1][0]+1 == cursor.getX() && plusPositions[plusCounter-1][1]+1 == cursor.getY() && numberPositions[numberCounter-1][0] == cursor.getX() && numberPositions[numberCounter-1][1] == cursor.getY()-1) // 3 x+ y+ down

{

flag = **true**;

numberPositions[numberCounter][0] = cursor.getX();

numberPositions[numberCounter][1] = cursor.getY()+1;

}

**else** **if**(plusPositions[plusCounter-1][0]-1 == cursor.getX() && plusPositions[plusCounter-1][1]+1 == cursor.getY() && numberPositions[numberCounter-1][0] == cursor.getX() && numberPositions[numberCounter-1][1] == cursor.getY()-1) // 4 x- y+ down

{

flag = **true**;

numberPositions[numberCounter][0] = cursor.getX();

numberPositions[numberCounter][1] = cursor.getY()+1;

}

**else** **if**(plusPositions[plusCounter-1][0]+1 == cursor.getX() && plusPositions[plusCounter-1][1]-1 == cursor.getY() && numberPositions[numberCounter-1][0] == cursor.getX() && numberPositions[numberCounter-1][1] == cursor.getY()+1) // 5 x+ y- up

{

flag = **true**;

numberPositions[numberCounter][0] = cursor.getX();

numberPositions[numberCounter][1] = cursor.getY()-1;

}

**else** **if**(plusPositions[plusCounter-1][0]-1 == cursor.getX() && plusPositions[plusCounter-1][1]-1 == cursor.getY() && numberPositions[numberCounter-1][0] == cursor.getX() && numberPositions[numberCounter-1][1] == cursor.getY()+1) // 6 x- y- up

{

flag = **true**;

numberPositions[numberCounter][0] = cursor.getX();

numberPositions[numberCounter][1] = cursor.getY()-1;

}

}

**if** (flag)

{

gameMap[cursor.getY()][cursor.getX()] = '+';

plusPositions[plusCounter][0] = cursor.getX();

plusPositions[plusCounter][1] = cursor.getY();

plusCounter++;

chain.add(gameMap[numberPositions[numberCounter][1]][numberPositions[numberCounter][0]]);

numberCounter++;

}

}

}

keypr=0; // last action

}

// Map Displaying

**for**(**int** j = 0; j<=18;j++) **for**(**int** i = 0; i<=30;i++) *cn*.getTextWindow().output(i, j,gameMap[j][i]);

**for**(**int** j = 0;j<=18;j++) *cn*.getTextWindow().output(31, j, ' ');

**for**(**int** i = 0;i<=31;i++) *cn*.getTextWindow().output(i, 20, ' ');

*cn*.getTextWindow().setCursorPosition(39, 0);

*cn*.getTextWindow().output(String.*format*("Board Seed: %d ", SEED));

*cn*.getTextWindow().setCursorPosition(39, 1);

*cn*.getTextWindow().output(String.*format*("Round: %d ", round));

*cn*.getTextWindow().setCursorPosition(39, 2);

*cn*.getTextWindow().output(String.*format*("Score: %d ", score));

*cn*.getTextWindow().setCursorPosition(39, 3);

*cn*.getTextWindow().output(String.*format*("Live Chain: "));

*cn*.getTextWindow().setCursorPosition(39, 4);

*cn*.getTextWindow().output(String.*format*("Chain Length: %d ", numberCounter));

*cn*.getTextWindow().setCursorPosition(39, 5);

*cn*.getTextWindow().output(String.*format*("---------------- "));

*cn*.getTextWindow().setCursorPosition(3, 22);

*cn*.getTextWindow().output(String.*format*("Player Name: %s " , playerName));

**for**(**int** i = 0;i<chain.size();i++)

{

*cn*.getTextWindow().setCursorPosition(53+(i\*2), 3);

**if**(i == chain.size()-1) *cn*.getTextWindow().output(String.*format*("%c ", chain.getDataFromIndex(i)));

**else** *cn*.getTextWindow().output(String.*format*("%c+", chain.getDataFromIndex(i)));

}

// Cursor Display

*cn*.getTextWindow().output(cursor.getX(), cursor.getY(), 'O');

Thread.*sleep*(THREADTIME);

}

// After the game finishes

// Add the new score to the list

String newWinner = playerName;

names.add(newWinner);

scores.add(score);

// Sorting the scores in descending order

highscoreSort(names, scores);

loadScoresFromFile("highscore.txt");

saveScoresToFile("highscore.txt");

Thread.*sleep*(2000);

clear();

ct.setCursorPosition(30, 2);

System.***out***.println("============");

ct.setCursorPosition(29, 3);

System.***out***.println(" HIGH SCORE");

ct.setCursorPosition(30, 4);

System.***out***.println("============");

ct.setCursorPosition(30, 5);

// Sorting the scores in descending order

highscoreSort(names, scores);

**int** k = 0;

Node namesNode = names.getHead();

Node scoresNode = scores.getHead();

**while** (namesNode != **null** && scoresNode != **null**) {

String name = (String) namesNode.getData();

namesNode = namesNode.getNext();

**int** score = (**int**) scoresNode.getData();

scoresNode = scoresNode.getNext();

// Print the name and score

ct.setCursorPosition(32, 6 + k);

*cn*.getTextWindow().output(String.*format*("%s: %d", name, score));

k++;

}

ct.setCursorPosition(0, 0);

}

}

**public** **class** Main {

**public** **static** **void** main(String[] args) **throws** Exception {

Chain chain = **new** Chain();

}

}