

COMP132: Advanced Programming

Programming Project Report

Food Chain Through Time: Simulation Design and Development

<Mesut KILIÇ, 87121>

<Fall 2025>



Table of Contents

<u>1. PART 1 – GENERAL DEMO INFORMATION</u>	4
<u>1.1 HOW TO RUN THE APPLICATION</u>	4
<u>1.2 DEMONSTRATING FOODCHAIN.TXT CHANGES</u>	5
<u>1.3 STARTING A NEW GAME FROM GUI</u>	7
<u>1.3.1 ERA SELECTION (PAST / PRESENT / FUTURE)</u>	7
<u>1.3.2 GRID SIZE SELECTION</u>	8
<u>1.3.3 NUMBER OF ROUNDS SELECTION</u>	9
<u>1.4 GAMEPLAY</u>	9
<u>1.4.1 PLAYER MOVEMENT</u>	10
<u>1.4.2 SPECIAL MOVES</u>	10
<u>1.4.3 POINT UPDATES</u>	12
<u>1.4.4 RESPAWN BEHAVIOUR</u>	13
<u>1.4.5 GAME COMPLETION</u>	14
<u>1.4.6 LOGS FOR THE CURRENT GAME</u>	16
<u>1.5 SAVE GAME AND SHOW SAVED STATE</u>	18
<u>1.6 ERA DIFFERENCES IN SPECIAL MOVES AND ICONS</u>	20
<u>2. PART 2 – PROJECT DESIGN DESCRIPTION</u>	21

<u>2.1 MODEL PACKAGE</u>	22
<u>2.2 IO PACKAGE.....</u>	23
<u>2.3 EXCEPTIONS PACKAGE.....</u>	26
<u>2.4 MAIN PACKAGE</u>	27
<u>2.5 LOGIC PACKAGE</u>	27
<u>2.6 GUI PACKAGE</u>	30
<u>3. REFERENCES</u>	34

1. Part 1 – General Demo Information

This report demonstrates the usage and technical design of the Food Chain Game, a turn-based grid simulation implemented in Java with a Swing-based graphical user interface. The application allows the user to start a new game by selecting the Era (Past/Present/Future), grid size, and total number of rounds, and then play by clicking on highlighted valid target cells. The gameplay includes both normal movement and era-dependent special abilities with cooldown constraints, along with scoring, eating interactions, and automatic respawn behavior.

1.1 How to Run the Application

This project is a Java Swing desktop application. To run the game, compile and execute the `Main` class, which launches the GUI using `SwingUtilities.invokeLater(...)` to ensure thread-safe Swing initialization.

```
public static void main(String[] args) {  
  
    SwingUtilities.invokeLater(new Runnable() {  
  
        @Override  
  
        public void run() {  
  
            try {  
  
                GameFrame frame = new GameFrame();  
  
                frame.setVisible(true);  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
        }  
    });  
}
```

```
    } catch (Exception e) {  
  
        e.printStackTrace();  
  
    }  
}
```

The user must ensure that the required .txt, sound and image files are present in the class path; however, even if these files are missing, the game bypasses such situations and starts with default values. Then, the user is directed to the Start Panel.

1.2 Demonstrating foodchain.txt Changes

All entities in the game (ApexPredator, Predator, Prey, Food) are created by randomly selecting one of the lines from the files future.txt, past.txt, or present.txt located in the project classpath. Depending on the era chosen by the user, a random line from the corresponding .txt file is selected, and names and other attributes are assigned accordingly.

Example present.txt line:

Food Chain 6: Crocodile, Leopard, Gazelle, Acacia

```
// Select a random chain from the file  
  
SecureRandom rnd = new SecureRandom();  
  
String selected = chains.get(rnd.nextInt(chains.size()));
```

The selection of different lines leads to various differences. Each entity receives a distinct name,

and this variation results in different images being assigned, thereby ensuring that the visuals are not identical. As a result, instead of constantly encountering the same creatures, you may find yourself watching a dinosaur fight from behind the bushes, wandering through a zoo, or escaping from aliens.

```
public static String[] loadFoodChainNames(String mode) throws IOException {  
    String fileName = mode.toLowerCase() + ".txt";
```

The `loadFoodChainNames` method in the `FileManager` class returns a formatted String array based on a randomly selected line, assisting the `initializeGame` method in the `GameEngine` class to assign names to the entities.

```
String[] names = FileManager.loadFoodChainNames(currentMode);  
  
String apexName = names[0];  
  
String predatorName = names[1];  
  
String preyName = names[2];  
  
String foodName = names[3];
```

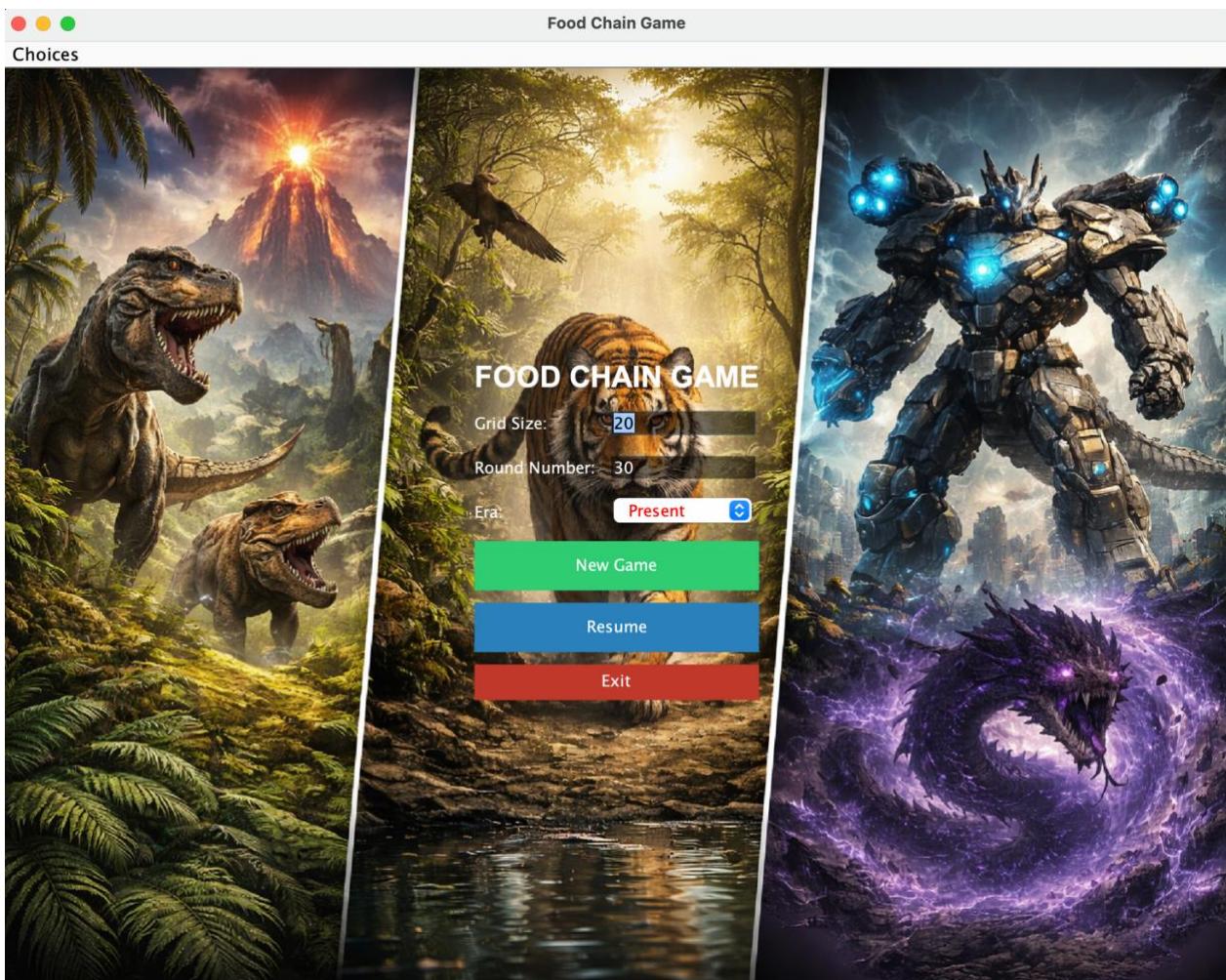
The image path of each entity is generated using the

```
key = Entity.getName().toLowerCase().replaceAll("[\\s-]", "");
```

which ensures standardization and allows the appropriate image to be found.

1.3 Starting a New Game from GUI

When the user runs the program, the Start screen appears as shown in the image.



On this screen, the user can set the number of rounds, grid size, and era options. By selecting the appropriate values for rounds, grid size, and era, the user can start a new game. In addition, the user may continue a previously saved and unfinished game or choose to exit instead.

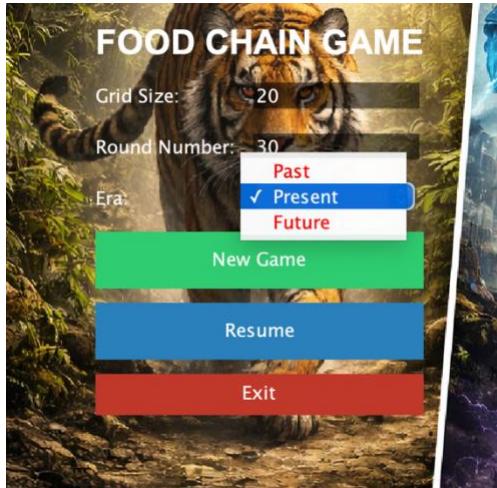
1.3.1 Era Selection (Past / Present / Future)

The user can choose the era option as past, present, or future. This selection determines the current round by setting the mode in the `StartPanel` class, and the `GameFrame` class then takes this as a parameter. As showed here:

```

String mode = (String) cmbMode.getSelectedItem();
gameFrame.startGame(size, rounds, mode);

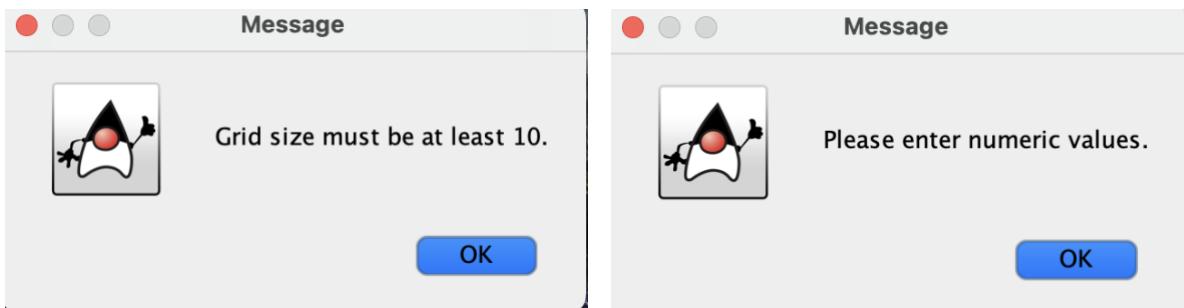
```



The era selection made by the user ensures that special movements are processed correctly and that a line is chosen from the appropriate .txt file.

1.3.2 Grid Size Selection

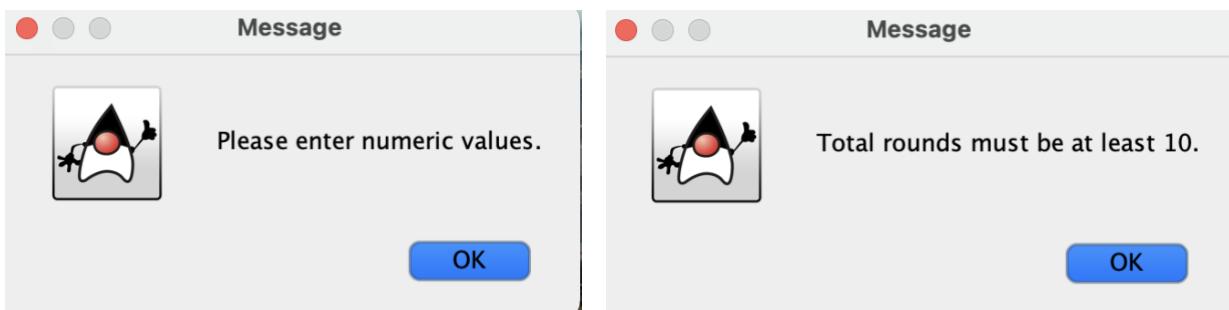
The $n \times n$ grid size selected by the user is important for the game panel on which the game will be played. While the minimum grid size is set to 10, no upper limit has been defined. If the user enters a value less than 10, the warning 'grid size must be at least 10' is displayed; if non-numeric values are entered, the warning 'Please enter numeric values' is shown.



1.3.3 Number of Rounds Selection

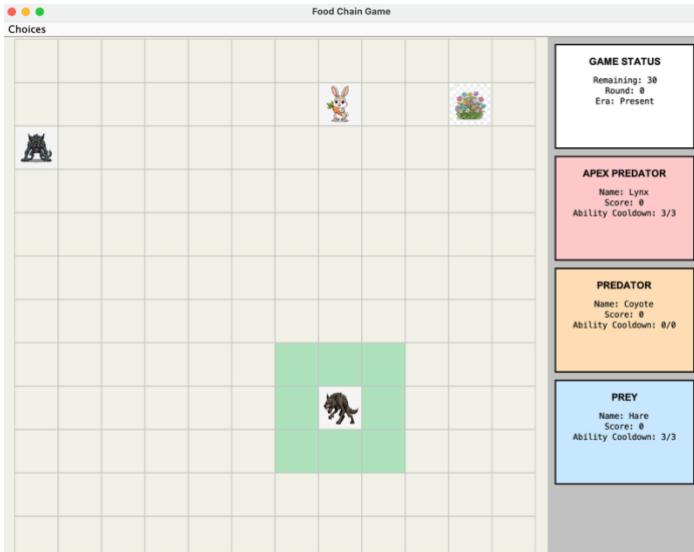
The user specifies the number of rounds to be played by entering input in the *Round Number:* field. This determines the duration of the game. Since there is no 'die' mechanic in the game and the winner is decided based on the highest score at the end of the rounds, the round count is crucial.

Here as well, the minimum round count is 10; therefore, similar warnings are displayed when the user enters numeric values less than 10 or non-numeric inputs.



1.4 Gameplay

The user can start the game by selecting the era, number of rounds, and grid size, then clicking the New Game button. The game order proceeds as Prey → Predator → Apex, meaning that the Prey's first move is already executed at the start. When it is the user's turn, they can choose from several available options



1.4.1 Player Movement

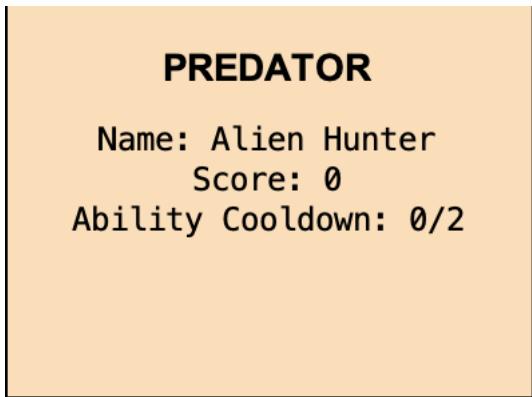
When it is the user's turn as the Predator, there are essentially three basic options by via mouse:

1. Remain in place
2. Move to the squares highlighted in green,
3. Use a special ability to move to the squares highlighted in yellow.

The objective here is to score points by eating the Prey while simultaneously trying to escape from the Apex Predator.

1.4.2 Special Moves

Each animal has a Special Move (Ability) that enables movement other than normal walking rules. The special moves depends on the cooldown mechanism, which is displayed on the right-side Info Panel as:



Ability Cooldown: current / max.

When the cooldown reaches 0, the special move becomes available.



UI Demonstration (Highlighted Cells): When the ability is not available ($\text{cooldown} > 0$), the board highlights only normal move cells (walk).



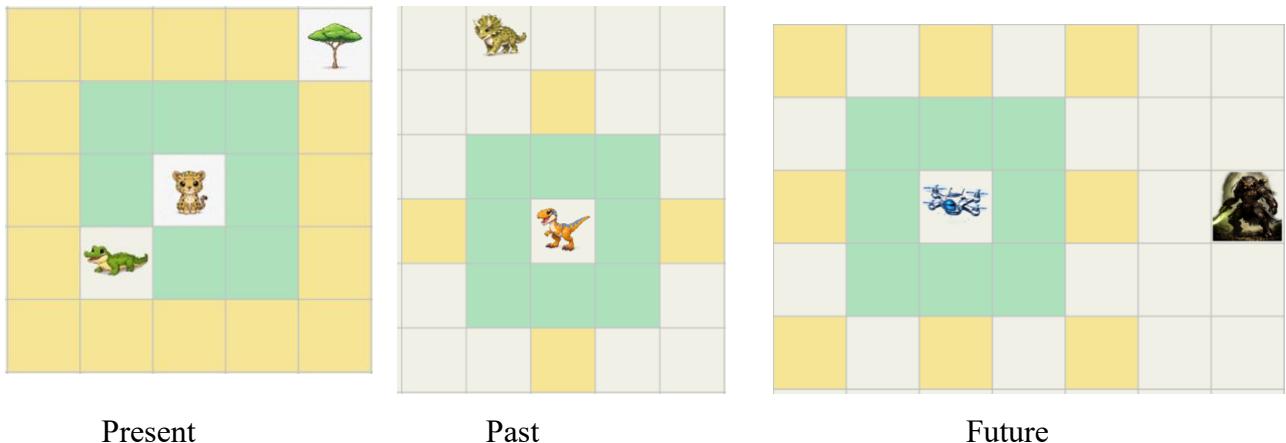
When the ability is available ($\text{cooldown} = 0$), the board additionally highlights special-move targets with a different color overlay, indicating cells reachable via the ability.

Using a Special Move: The player triggers a special move by clicking a highlighted special target cell.

After the ability is used:

The cooldown is reset to its maximum value. The ability becomes unavailable until the cooldown

decreases back to 0 across subsequent rounds.



Era-based differences: Special moves and their movement geometries vary by Era (Past / Present / Future), therefore the highlighted special target cells also change accordingly. This allows demonstrating distinct gameplay behaviours across eras.

1.4.3 Point Updates

Scores are updated dynamically based on eating interactions in the game. Each successful consumption triggers an immediate score change, and the values can be observed from the Info Panel under each entity's score field.

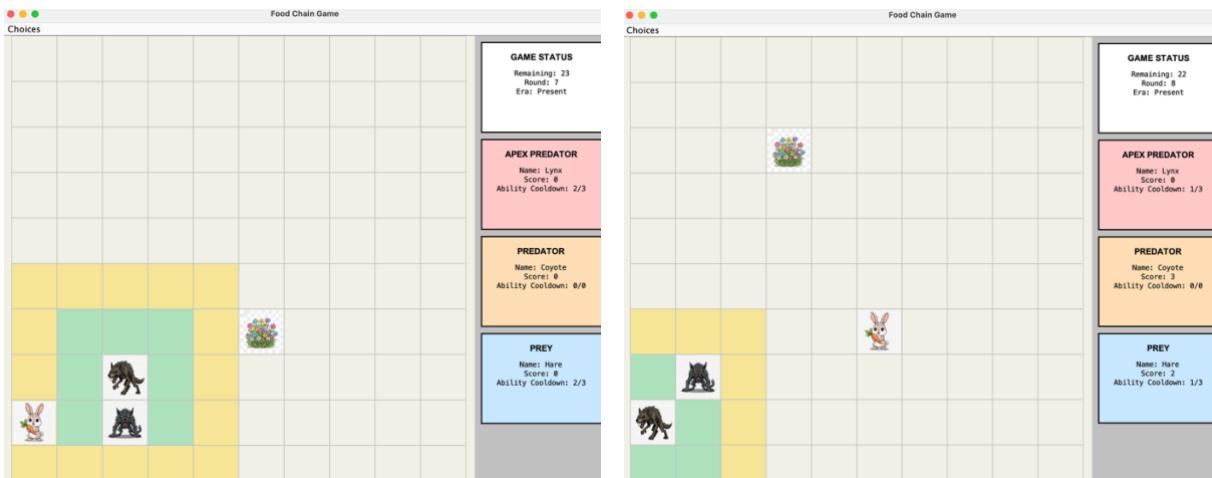


Scoring rules:

Prey eats Food: Prey gains +3 points.

Predator eats Prey: Predator gains +3 points, and the eaten Prey receives -1 point.

Apex eats an Animal (Predator or Prey): Apex gains +1 point, and the eaten Animal receives -1 point.



Before eating Prey (Round 7)

Predator gained +3 point

1.4.4 Respawn Behaviour

In this game, when something gets eaten it doesn't really go away forever. It just comes back again because there's a respawn system. This way the world doesn't feel empty and the game keeps going until the round limit is done.

As a result of the respawn action, the location where the entity will reappear is chosen randomly.

```
private void spawnEntityRandomly(Entity e) {  
    int size = grid.getSize();  
    int x, y;  
  
    do {  
        x = random.nextInt(size);  
        y = random.nextInt(size);  
    } while (!grid.getCell(x, y).isEmpty());  
  
    grid.placeEntity(e, x, y);  
  
    if (e instanceof Animal) {  
        Animal animal = (Animal) e;  
        animal.setPosition(new Point(x, y));  
        animal.respawn(x, y);  
    }  
}
```

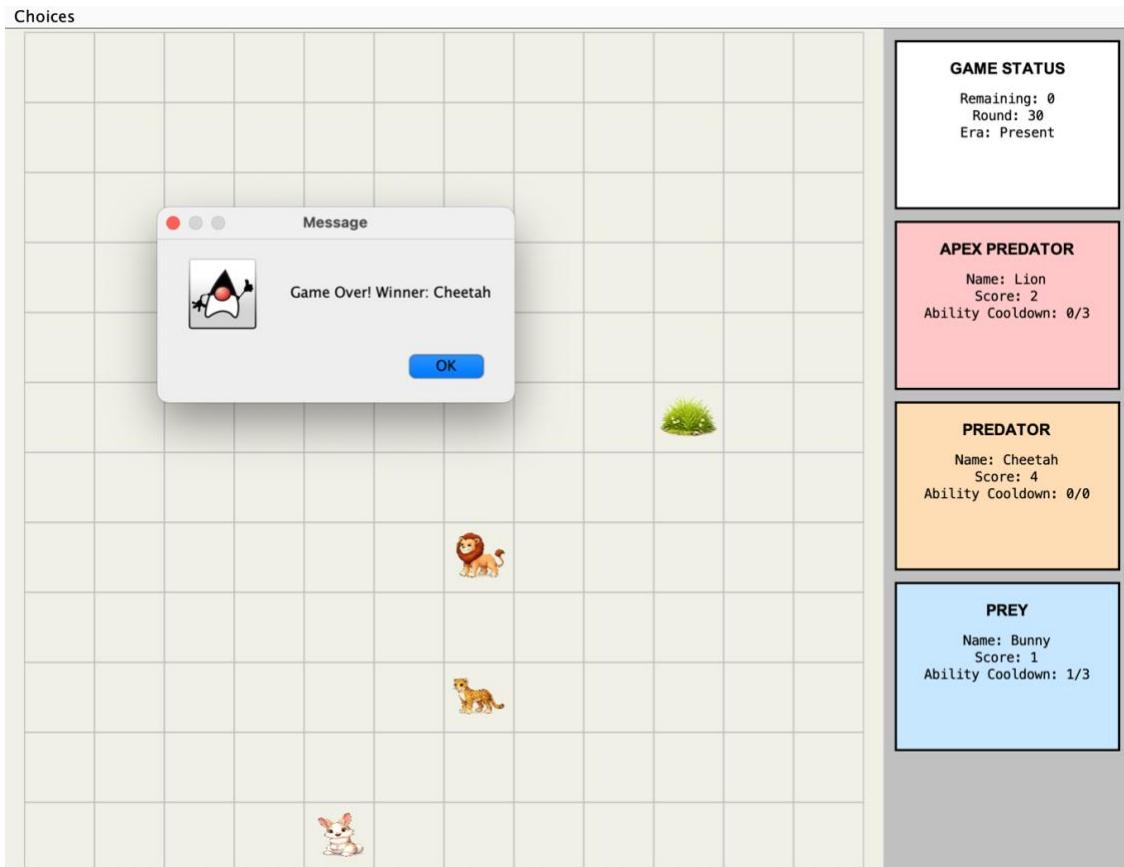
1.4.5 Game Completion

A game session finishes when the total number of rounds reaches the maximum.

At that point, the engine marks the game as finished and determines the winner by comparing final scores.

Winner is entity which has most points at end of the game.

If there are equality of maximum points gained then the there is a draw and both entities have most points are winner.



The winner is calculated and displayed on the screen.

```
public String getWinner() {
    if (!player.isAlive()) {
        return apex.getName() + " (Player Eliminated)";
    }

    int pScore = player.getScore();
    int aScore = apex.getScore();
    int prScore = prey.getScore();
```

```
if (pScore > aScore && pScore > prScore) return player.getName();
if (aScore > pScore && aScore > prScore) return apex.getName();
if (prScore > pScore && prScore > aScore) return prey.getName();

return "Draw";
}
```

1.4.6 Logs for the Current Game

The game makes a log file called `game_log.txt` to keep track of the main things that happen while playing. This log helps check if the game rules and flow are working right, and it also acts like a simple debug record of the session.

```

1 [02:20:03] GAME_START era=Present totalRounds=30 playerRole=Predator
2 [02:20:03] SPAWN player=Leopard(Predator) (x=9,y=7) apex=Crocodile(Apex) (x=8,y=5) prey=Gazelle(Prey) (x=5,y=4)
3 [02:20:03] ROUND_BEGIN r=0/30 era=Present playerRole=Predator
4 [02:20:03] MOVE AI actor=Gazelle(Prey) from=(5,4) to=(4,5)
5 [02:20:06] MOVE PLAYER actor=Leopard(Predator) from=(9,7) to=(8,8)
6 [02:20:06] MOVE AI actor=Crocodile(Apex) from=(8,5) to=(8,6)
7 [02:20:06] ROUND_END r=0/30 era=Present scores: player=0 apex=0 prey=0
8 [02:20:06] ROUND_BEGIN r=1/30 era=Present playerRole=Predator
9 [02:20:06] MOVE AI actor=Gazelle(Prey) from=(4,5) to=(3,5)
10 [02:20:07] MOVE PLAYER actor=Leopard(Predator) from=(8,8) to=(7,8)
11 [02:20:07] MOVE AI actor=Crocodile(Apex) from=(8,6) to=(7,7)
12 [02:20:07] ROUND_END r=1/30 era=Present scores: player=0 apex=0 prey=0
13 [02:20:07] ROUND_BEGIN r=2/30 era=Present playerRole=Predator
14 [02:20:07] MOVE AI actor=Gazelle(Prey) from=(3,5) to=(2,4)
15 [02:20:08] MOVE PLAYER actor=Leopard(Predator) from=(7,8) to=(6,8)
16 [02:20:08] MOVE AI actor=Crocodile(Apex) from=(8,6) to=(7,7)
17 [02:20:08] ROUND_END r=2/30 era=Present scores: player=0 apex=0 prey=0
18 [02:20:08] ROUND_BEGIN r=3/30 era=Present playerRole=Predator
19 [02:20:08] MOVE AI actor=Gazelle(Prey) from=(2,4) to=(2,2)
20 [02:20:08] Gazelle used Hop
21 [02:20:10] MOVE PLAYER actor=Leopard(Predator) from=(6,8) to=(5,7)
22 [02:20:10] MOVE AI actor=Crocodile(Apex) from=(8,6) to=(6,8)
23 [02:20:10] Crocodile used Sprint
24 [02:20:10] ROUND_END r=3/30 era=Present scores: player=0 apex=0 prey=0
25 [02:20:10] ROUND_BEGIN r=4/30 era=Present playerRole=Predator
26 [02:20:10] MOVE AI actor=Gazelle(Prey) from=(2,2) to=(3,1)
27 [02:20:12] MOVE PLAYER actor=Leopard(Predator) from=(5,7) to=(5,6)
28 [02:20:12] MOVE AI actor=Crocodile(Apex) from=(6,8) to=(5,7)
29 [02:20:12] ROUND_END r=4/30 era=Present scores: player=0 apex=0 prey=0
30 [02:20:12] ROUND_BEGIN r=5/30 era=Present playerRole=Predator
31 [02:20:12] MOVE AI actor=Gazelle(Prey) from=(3,1) to=(2,0)
32 [02:20:14] Leopard used Dash
33 [02:20:14] MOVE PLAYER actor=Leopard(Predator) from=(5,6) to=(3,4)
34 [02:20:14] MOVE AI actor=Crocodile(Apex) from=(5,7) to=(5,6)
35 [02:20:14] ROUND_END r=5/30 era=Present scores: player=0 apex=0 prey=0
36 [02:20:14] ROUND_BEGIN r=6/30 era=Present playerRole=Predator
37 [02:20:14] MOVE AI actor=Gazelle(Prey) from=(2,0) to=(1,0)
38 [02:20:15] MOVE PLAYER actor=Leopard(Predator) from=(3,4) to=(3,3)
39 [02:20:15] MOVE AI actor=Crocodile(Apex) from=(5,6) to=(2,3)
40 [02:20:15] Crocodile used Sprint
41 [02:20:15] ROUND_END r=6/30 era=Present scores: player=0 apex=0 prey=0
42 [02:20:15] ROUND_BEGIN r=7/30 era=Present playerRole=Predator
43 [02:20:15] MOVE AI actor=Gazelle(Prey) from=(1,0) to=(3,0)
44 [02:20:15] Gazelle used Hop
45 [02:20:17] Leopard used Dash
46 [02:20:17] MOVE PLAYER actor=Leopard(Predator) from=(3,3) to=(4,1)
47 [02:20:17] MOVE AI actor=Crocodile(Apex) from=(2,3) to=(3,2)
48 [02:20:17] ROUND_END r=7/30 era=Present scores: player=0 apex=0 prey=0
49 [02:20:17] ROUND_BEGIN r=8/30 era=Present playerRole=Predator
50 [02:20:17] MOVE AI actor=Gazelle(Prey) from=(3,0) to=(2,0)
51 [02:20:18] SCORE_GAIN Leopard(Predator) gain:3 points reason:PREDATOR_EATS_PREY
52 [02:20:18] SCORE_LOSS Gazelle(Prey) loss:1 point reason:BE_EATEN
53 [02:20:18] Gazelle respawns
54 [02:20:18] Leopard used Dash
55 [02:20:18] MOVE PLAYER actor=Leopard(Predator) from=(4,1) to=(2,0)
56 [02:20:18] MOVE AI actor=Crocodile(Apex) from=(3,2) to=(3,1)
57 [02:20:18] ROUND_END r=8/30 era=Present scores: player=3 apex=0 prey=-1
58 [02:20:18] ROUND_BEGIN r=9/30 era=Present playerRole=Predator
59 [02:20:18] MOVE AI actor=Gazelle(Prey) from=(5,10) to=(6,11)
60 [02:20:18] Leopard used Dash

```

What goes into the log During a normal game, the log writes down things like:

- Game start info (Era and total rounds)
- First spawns (where Player / Apex / Prey appear)
- Round start and end markers (with scores)

- Moves (AI and player moves with from/to positions)
- Ability use (when a special move happens and cooldown starts)
- Score changes (when something eats another entity or food)
- Respawn events (when prey or food comes back)
- Game over summary (who won and the final state)

```

1 [02:31:49] GAME_START era=Future totalRounds=10 playerRole=Predator
2 [02:31:49] SPAWN player=Alien Hunter(Predator) (x=3,y=7)- apex=Alien Overlord(Apex) (x=0,y=8)- prey=Human(Prey) (x=4,y=6)
3 [02:31:49] ROUND_BEGIN r=0/10 era=Future playerRole=Predator
4 [02:31:49] MOVE AI actor=Alien Hunter(Predator) from=(3,7) to=(4,6) %
5 [02:31:49] MOVE PLAYER actor=Alien Hunter(Predator) from=(3,7) to=(4,6) %
6 [02:31:49] MOVE AI actor=Alien Overlord(Apex) from=(0,8) to=(1,7) %
7 [02:31:49] ROUND_END r=0/10 era=Future scores: player=+ apex=0 prey=0
8 [02:31:49] ROUND_BEGIN r=1/10 era=Future playerRole=Predator
9 [02:31:49] MOVE AI actor=Human(Prey) from=(1,5) to=(1,6) %
10 [02:31:49] MOVE PLAYER actor=Alien Hunter(Predator) from=(4,6) to=(5,7) %
11 [02:31:49] MOVE AI actor=Alien Overlord(Apex) from=(1,7) to=(2,8) %
12 [02:31:49] ROUND_END r=1/10 era=Future scores: player=+ apex=0 prey=0
13 [02:31:49] ROUND_BEGIN r=2/10 era=Future playerRole=Predator
14 [02:31:49] SCORE_GAIN Human(Prey) gain 3 points reason:AI_FOOD
15 [02:31:49] Move AI
16 [02:31:49] MOVE AI actor=Human(Prey) from=(6,8) to=(5,9) %
17 [02:31:49] SCORE_GAIN Alien Hunter(Predator) gain 3 points reason:PREATOR_EATS_PREY
18 [02:31:49] SCORE_LOSS Human(Prey) loss 1 point reason:BE_EATEN
19 [02:31:49] Human respawns
20 [02:31:49] MOVE PLAYER actor=Alien Hunter(Predator) from=(5,7) to=(5,9) %
21 [02:31:49] Alien Hunter used special ability (Dash)%
22 [02:31:49] MOVE AI actor=Alien Overlord(Apex) from=(2,8) to=(3,9) %
23 [02:31:49] ROUND_END r=2/10 era=Future scores: player=3 apex=0 prey=2
24 [02:31:49] ROUND_BEGIN r=3/10 era=Future playerRole=Predator
25 [02:31:49] MOVE AI actor=Human(Prey) from=(4,0) to=(7,0) %
26 [02:31:49] Move AI
27 [02:31:49] MOVE PLAYER actor=Alien Hunter(Predator) from=(5,9) to=(6,8) %
28 [02:31:49] SCORE_GAIN Alien Overlord(Apex) gains 1 point reason:APEX_EATS_ANIMAL
29 [02:31:49] SCORE_LOSS Alien Hunter(Predator) loss 1 point reason:BE_EATEN
30 [02:31:49] Alien Hunter respawns
31 [02:31:49] MOVE AI actor=Alien Overlord(Apex) from=(3,9) to=(6,8) %
32 [02:31:49] Alien Overlord used Sprint
33 [02:31:49] ROUND_END r=3/10 era=Future scores: player=2 apex=1 prey=2
34 [02:31:49] ROUND_BEGIN r=4/10 era=Future playerRole=Predator
35 [02:31:49] MOVE AI actor=Human(Prey) from=(7,0) to=(6,1) %
36 [02:32:00] MOVE PLAYER actor=Alien Hunter(Predator) from=(9,0) to=(7,2) %
37 [02:32:00] Alien Hunter used special ability (Dash)%
38 [02:32:00] MOVE AI actor=Alien Overlord(Apex) from=(6,8) to=(7,7) %
39 [02:32:00] ROUND_END r=4/10 era=Future scores: player=+ apex=1 prey=2
40 [02:32:00] ROUND_BEGIN r=5/10 era=Future playerRole=Predator
41 [02:32:00] MOVE AI actor=Human(Prey) from=(6,1) to=(3,1) %
42 [02:32:00] Human used Hop
43 [02:32:00] MOVE AI actor=Alien Hunter(Predator) from=(7,2) to=(6,1) %
44 [02:32:00] MOVE AI actor=Alien Overlord(Apex) from=(7,7) to=(8,6) %
45 [02:32:05] ROUND_END r=5/10 era=Future scores: player=2 apex=1 prey=2
46 [02:32:05] ROUND_BEGIN r=6/10 era=Future playerRole=Predator
47 [02:32:05] MOVE AI actor=Human(Prey) from=(3,1) to=(2,2) %
48 [02:32:06] MOVE PLAYER actor=Alien Hunter(Predator) from=(6,1) to=(4,1) %
49 [02:32:06] Alien Hunter used special ability (Dash)%
50 [02:32:06] MOVE AI actor=Alien Overlord(Apex) from=(8,6) to=(5,3) %
51 [02:32:06] Alien Overlord used Sprint
52 [02:32:06] ROUND_END r=6/10 era=Future scores: player=2 apex=1 prey=2
53 [02:32:06] ROUND_BEGIN r=7/10 era=Future playerRole=Predator
54 [02:32:06] MOVE AI actor=Human(Prey) from=(2,2) to=(2,5) %
55 [02:32:06] Move AI
56 [02:32:06] MOVE PLAYER actor=Alien Hunter(Predator) from=(4,1) to=(3,1) %
57 [02:32:06] MOVE AI actor=Alien Overlord(Apex) from=(5,3) to=(4,2) %
58 [02:32:06] ROUND_END r=7/10 era=Future scores: player=+ apex=1 prey=2
59 [02:32:06] ROUND_BEGIN r=8/10 era=Future playerRole=Predator
60 [02:32:06] MOVE AI actor=Human(Prey) from=(2,2) to=(3,6) %
61 [02:32:06] Alien Hunter used Dodge from=(3,1) to=(1,3) %
62 [02:32:06] Alien Hunter used special ability (Dash)%
63 [02:32:09] MOVE AI actor=Alien Overlord(Apex) from=(4,2) to=(3,1) %
64 [02:32:09] ROUND_END r=8/10 era=Future scores: player=2 apex=1 prey=2
65 [02:32:09] ROUND_BEGIN r=9/10 era=Future playerRole=Predator
66 [02:32:09] MOVE AI actor=Human(Prey) from=(3,6) to=(6,9) %
67 [02:32:09] Move AI
68 [02:32:11] MOVE PLAYER actor=Alien Hunter(Predator) from=(1,3) to=(2,4) %
69 [02:32:11] SCORE_GAIN Alien Overlord(Apex) gains 1 point reason:APEX_EATS_ANIMAL
70 [02:32:11] SCORE_LOSS Alien Hunter(Predator) loss 1 point reason:BE_EATEN
71 [02:32:11] Alien Hunter respawns
72 [02:32:11] MOVE AI actor=Alien Overlord(Apex) from=(3,1) to=(2,4) %
73 [02:32:11] Alien Overlord used Sprint
74 [02:32:11] ROUND_END r=9/10 era=Future scores: player=1 apex=2 prey=2
75 [02:32:11] GAME_OVER era=Future totalRounds=10 winner=Draw

```

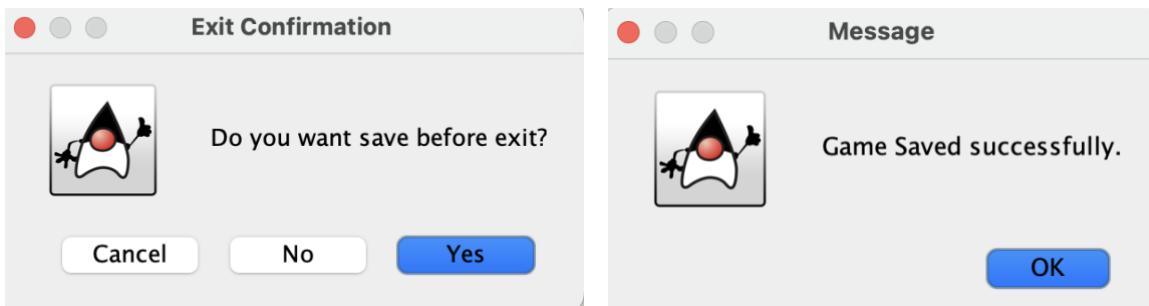
DRAW (10 round/10 grid size/ era future)

The log shows the main game flow: GAME_START → SPAWN → ROUND_BEGIN/END. Each round goes Prey AI, Player, Apex AI. It also records score changes from food, prey being eaten (+3, -1), respawns, and ability use (Dash-Hop-Sprint). GAME_OVER closes the session.

1.5 Save Game and Show Saved State

The game can be saved at any time while it is running via the menu option Choices → Save. When the save action is triggered, the current game state is serialized into a text file This file stores the

essential metadata (Era/Mode, Grid Size, Current Round, Max Rounds) and the full list of entities on the grid. For each entity, its type, name, coordinates, and (for animals) score and ability cooldown values are written, allowing the session to be reconstructed later from the exact same state.



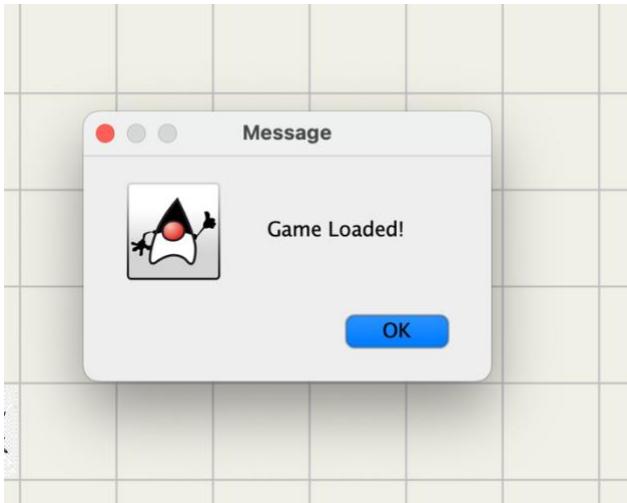
In addition, when the user clicks the exit button in the choices menu, a warning appears asking:

“Do you want to save before exiting?”

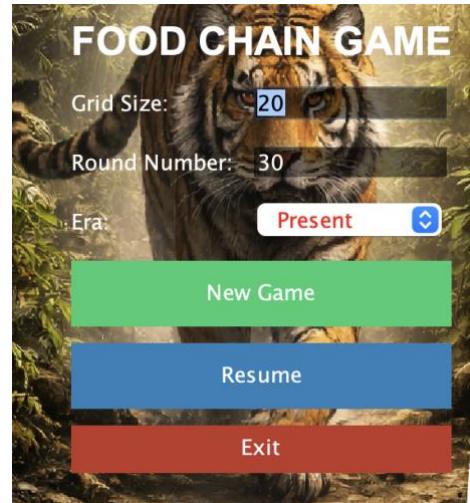
Later, if the player wants to continue from where they left off, they can click the Resume button on the StartPanel. The game then uses the information stored in savegame.txt to restore the current state.

```
1 MODE:Present
2 GRID_SIZE:12
3 ROUND:12
4 MAX_ROUNDS:30
5 ENTITY:FOOD,Plankton,1,8
6 ENTITY:Prey,Tuna,1,11,-3,1
7 ENTITY:Predator,Orca,2,8,5,0
8 ENTITY:Apex,Great White,4,7,2,0
9
```

It represented as “ENTITY:X, X, getName(), X.getx(), X.gety(), points, ability cooldown



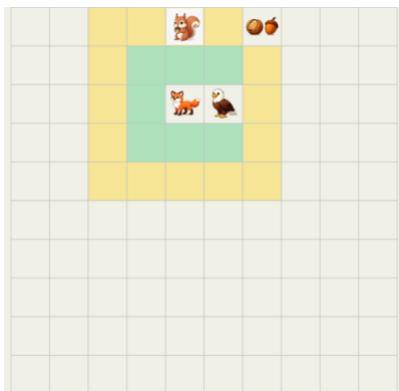
“Game loaded!”



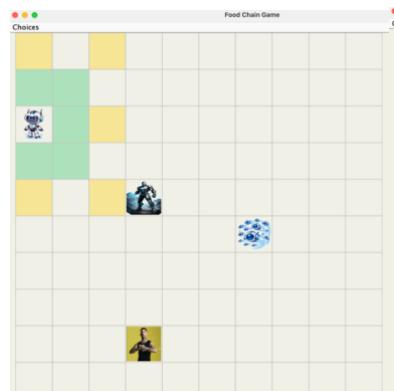
Resume button

1.6 Era Differences in Special Moves and Icons

The game has three eras: Past, Present, Future. Each era changes how special moves work and how things look. In every era, the highlighted cells are different because ability range and cooldown rules change. Also, the icons for animals/characters are loaded based on the era’s food chain setup, so sprites (or simple shapes if no image) look different too. This means if you start the same game in another era, the movement strategy and the board’s visual style will feel different.



Present Era



Future Era



Past Era

2. Part 2 – Project Design Description

Structure of my project:

- gui (package)
 - GameFrame
 - StartPanel
 - GamePanel
 - InfoPanel
- logic (package)
 - GameEngine
 - Grid
 - Cell
 - AIController
- model (package)
 - Entity
 - Food
 - GameState
 - model.animals (sub-package)
 - Animal
- io (package)
 - FileManager
 - GameLogger
 - SoundManager (win/lose sound playback)
- exceptions (package)
 - InvalidMoveException (custom exception for rule-violating moves)
- main (package)
 - Main

The project is split into layers. The GUI part shows the grid and takes user input. The logic part runs the rules and turn order. The game world is made of Grid and Cell, and things on the grid are Entities (like Animal, Food). The io part does file stuff: load food chain, save/load game, write logs, play sounds. If rules are broken (like wrong moves), custom exceptions are thrown and the GUI shows a warning.

2.1 model package

This package keeps the main game objects. These classes show what is on the board and hold the state for both the engine and the UI.

- Entity: Base class for anything placed on the grid. Stores x, y coordinates and has position update methods.
- Food: A simple subclass of Entity. Represents items that can be eaten. The name depends on the loaded food chain.
- Animal (inside model.animals): Active type in the game. Holds info like name, type, era, score, alive status, and movement/ability behavior. Includes:
 - Move type check (checkMoveType)
 - Ability geometry rules (isValidAbilityGeometry)
 - Eating rules (canEat)
 - Cooldown handling (triggerAbilityCooldown, reduceCooldown, isAbilityAvailable)
- GameState: A class that represents the whole snapshot of the game (round, mode, grid, entities). Used mainly for saving/loading with FileManager in text format.

```
public Animal(String name, String type, String era, int x, int y) {  
    super(x, y, name.charAt(0));
```

```

        this.name = name;
        this.type = type;
        this.era = era;
        this.position = new Point(x, y);
        this.score = 0;
        this.isAlive = true;

        assignProperties();

        this.cooldown = this.maxAbilityCooldown;
    }
}

```

Since Prey, Predator, and Apex don't have big differences apart from their special abilities, I didn't create separate classes for each. Instead, I preferred to pass their type as a parameter in the Animal constructor.

```
public abstract class Entity implements Serializable
```

I used the Entity class as an abstract class. This way, I didn't have to implement position handling separately for Food and all Animals.

2.2 io package

This package handles all the input/output stuff: reading config files, saving/loading, logging, and sounds.

- FileManager: Reads food chain files (like past.txt, present.txt, future.txt). Picks a random line and gets names for Apex, Predator, Prey, Food. Also does Save/Load with savegame.txt

by writing game data and rebuilding it later.

- GameLogger: Writes down what happens in the game (spawns, moves, scores, rounds, game over) into game_log.txt with time stamps. It flushes right away so logs are saved instantly.
- SoundManager: Plays win/lose sounds at the end of the game using Java Sound API

```
    /**
 * public static String[] loadFoodChainNames(String mode) throws IOException {
 *     String fileName = mode.toLowerCase() + ".txt"; // e.g., past.txt
 *
 *     InputStream is = FileManager.class.getResourceAsStream("/" + fileName);
 *     if (is == null) {
 *         throw new IOException("File not found: /" + fileName + " (Must be in classpath)");
 *     }
 *
 *     List<String> chains = new ArrayList<>();
 *
 *     try (BufferedReader br = new BufferedReader(new InputStreamReader(is))) {
 *         String line;
 *         while ((line = br.readLine()) != null) {
 *             if (line.startsWith("Food Chain")) {
 *                 chains.add(line);
 *             }
 *         }
 *     }
 *
 *     if (chains.isEmpty()) {
 *         throw new IOException("No 'Food Chain' lines found in: " + fileName);
 *     }
 *
 *     SecureRandom rnd = new SecureRandom();
 *     String selected = chains.get(rnd.nextInt(chains.size()));
 *
 *     int colon = selected.indexOf(':');
 *     if (colon < 0) {
 *         throw new IOException("Invalid format (missing ':'): " + selected);
 *     }
 *
 *     String payload = selected.substring(colon + 1).trim();
 *     String[] parts = payload.split(",");
 *
 *     if (parts.length != 4) {
 *         throw new IOException("Expected 4 names (Apex,Predator,Prey,Food), found: " + selected);
 *     }
 *
 *     return new String[] {
 *         parts[0].trim(),
 *         parts[1].trim(),
 *         parts[2].trim(),
 *         parts[3].trim()
 *     };
 * }
```

In the FileManager's loadFoodChainNames method, a random line is picked. If the line is not in the correct format, it is handled with try-catch blocks. The valid line is then split, and the needed name values are returned as a string array.

```

>   public static void saveGame(GameEngine engine) {
>     try (PrintWriter out = new PrintWriter(new FileWriter(SAVE_FILE))) {
>       out.println("MODE:" + engine.getCurrentMode());
>       out.println("GRID_SIZE:" + engine.getGrid().getSize());
>       out.println("ROUND:" + engine.getCurrentRound());
>       out.println("MAX_ROUNDS:" + engine.getMaxRounds());
>
>       List<Entity> entities = engine.getGrid().getEntities();
>
>       for (Entity e : entities) {
>         if (e instanceof Animal) {
>           Animal a = (Animal) e;
>           out.println(String.format("ENTITY:%s,%s,%d,%d,%d",
>             a.getType(), a.getName(), a.getX(), a.getY(), a.getAbilityCooldown()));
>         } else if (e instanceof Food) {
>           Food f = (Food) e;
>           out.println(String.format("ENTITY:FOOD,%s,%d,%d",
>             f.getName(), f.getX(), f.getY()));
>         }
>       }
>       out.flush();
>       System.out.println("Game saved successfully.");
>     } catch (IOException e) {
>       System.err.println("Save failed: " + e.getMessage());
>     }
>   }

```

In the FileManager class, the saveGame method writes all the necessary information so the game can later continue safely from where it was left off.

```

public static GameEngine loadGame() throws IOException {
  File file = new File(SAVE_FILE);
  if (!file.exists()) {
    throw new FileNotFoundException("Save file not found.");
  }

  String mode = "Present";
  int round = 0;
  int maxRounds = 30;
  int gridSize = 20; // Default fallback for legacy saves
  List<String> entityLines = new ArrayList<>();

  try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
    String line;
    while ((line = reader.readLine()) != null) {
      if (line.startsWith("MODE:")) {
        mode = line.split(":")[1].trim();
      } else if (line.startsWith("ROUND:")) {
        round = Integer.parseInt(line.split(":")[1].trim());
      } else if (line.startsWith("MAX_ROUNDS:")) {
        maxRounds = Integer.parseInt(line.split(":")[1].trim());
      } else if (line.startsWith("GRID_SIZE:")) {
        gridSize = Integer.parseInt(line.split(":")[1].trim());
      } else if (line.startsWith("ENTITY:")) {
        entityLines.add(line.substring(7)); // Remove "ENTITY:" prefix
      }
    }
  }

  GameEngine engine = new GameEngine(gridSize, maxRounds, mode);
  engine.setCurrentRound(round);
}

```

In the FileManager class, the loadGame method takes the data previously saved in **savegame.txt** and uses it to restore the game so it can continue from where it left off.

```

public static void log(String message) {%
>   if (!isInitialized) {%
>     System.out.println("[Log (No File)]: " + message);%
>   }%
>%
>   try {%
>     String timestamp = getCurrentTime();%
>     String formattedMessage = String.format("[%s] %s", timestamp, message);%
>%
>     writer.write(formattedMessage);%
>     writer.newLine();%
>     writer.flush();%
>   } catch (IOException e) {%
>     System.err.println("Error writing to log: " + e.getMessage());%
>   }%
}

private static void playSound(String filePath) {%
>   try {%
>     File soundfile = new File(filePath);%
>%
>     if (soundfile.exists()) {%
>       AudioInputStream audioInput = AudioSystem.getAudioInputStream(soundfile);%
>       Clip clip = AudioSystem.getClip();%
>       clip.open(audioInput);%
>       clip.start();%
>     } else {%
>       System.err.println("Sound file not found: " + filePath);%
>     }%
>   } catch (UnsupportedAudioFileException e) {%
>     System.err.println("Unsupported audio format! Please use .WAV files.");%
>   } catch (TDEException | LineUnavailableException e) {%
>     e.printStackTrace();%
>   }%
}

```

"In the GameLogger class, the log method is called whenever something needs to be written into game_log.txt. To add some flavor to the game, I implemented a SoundManager class where the playSound method plays music from the given path.

2.3 exceptions package

The exceptions package contains custom exceptions used for rule enforcement.

- **InvalidMoveException** is thrown when a move violates constraints such as out-of-bound targets, illegal ability usage, or era-specific rule restrictions. This enables clean separation of validation and UI feedback (GUI catches and shows warning dialogs).

```

1 package exceptions;%
2 %
3 /**
4  * Custom exception class thrown when a move violates game rules.%
5  * (e.g., Going out of grid bounds, entering a blocked path, etc.)%
6 */
7 public class InvalidMoveException extends Exception {%
8 %
9   private static final long serialVersionUID = 1L;%
10 %
11   public InvalidMoveException(String message) {%
12     super(message);%
13   }%
14 }%
15

```

Besides that, I used try-catch blocks a lot. This helped me catch small specific errors, for example when numeric values are entered in the StartPanel or when a file path cannot be found during file handling.

```

public static void init() {
    try {
        writer = new BufferedWriter(new FileWriter("game_log.txt"));
        isInitialized = true;
    } catch (IOException e) {
        System.err.println("Failed to create log file: " + e.getMessage());
    }
}

try {
    String timestamp = getCurrentTime();
    String formattedMessage = String.format("[%s] %s", timestamp, message);
    writer.write(formattedMessage);
    writer.newLine();
    writer.flush();
} catch (IOException e) {
    System.err.println("Error writing to log: " + e.getMessage());
}
}

```

2.4 main package

The main package contains the entry point of the application.

- Main launches the GameFrame on the Swing Event Dispatch Thread (EDT) using SwingUtilities.invokeLater(...), ensuring thread safety and a responsive GUI.

It makes sure that the main runs in the correct order without confusion.

```

public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        @Override
        public void run() {
            try {
                GameFrame.frame = new GameFrame();
                ...frame.setVisible(true);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
}

```

2.5 logic package

This package runs the main game behavior. It handles turn order, checks move, runs actions (eat, score, respawn), and makes AI decisions.

- **GameEngine**: The main game loop controller. Keeps track of the Grid, round info, and active animals (Predator, Apex, Prey). It processes player moves, applies era rules, runs AI moves, and ends rounds by updating cooldowns, increasing round count, and checking if the game is finished.
- **Grid**: The board, built as a 2D array of Cells. Handles placing, moving, and removing entities. Has helper methods like isValidPosition(...) and getEntities().
- **Cell**: A small container that holds one Entity or nothing. Provides simple checks like isEmpty() and removeEntity().

- **AIController:** Logic for AI animals (Prey and Apex). Finds valid moves, scores them, and returns the chosen target for the engine to run.

```

public void initializeGame() {
    try {
        >> String[] names = FileManager.loadFoodChainNames(currentMode);
        >> String apexName = names[0];
        >> String predatorName = names[1];
        >> String preyName = names[2];
        >> String foodName = names[3];

        >> System.out.println("MODE=" + currentMode);
        >> System.out.println("Loaded names: " + apexName + ", " + predatorName + ", " + preyName + ", " + foodName);

        >> apex = new Animal(apexName, "Apex", currentMode, 0, 0);
        >> player = new Animal(predatorName, "Predator", currentMode, 0, 0);
        >> prey = new Animal(preyName, "Prey", currentMode, 0, 0);

        >> animals.clear();
        >> animals.add(apex);
        >> animals.add(player);
        >> animals.add(prey);

        >> spawnEntityRandomly(apex);
        >> spawnEntityRandomly(player);
        >> spawnEntityRandomly(prey);

        >> Food food = new Food(0, 0, foodName);
        >> spawnEntityRandomly(food);
    }
}

```

In the GameEngine, the initializeGame() method uses the loadFoodChainNames() method to create the Apex, Predator, Prey, and Food entities.

```

public void processPlayerMove(int targetX, int targetY) throws
InvalidMoveException {

```

I used this method also to process player move under different conditions. For different Era conditions and to make best choice to move this method makes all by helper methods.

```

    private void performAiMove(Animal actor, int[] moveCoords) {
        if (!actor.isAlive()) return;
        int fromX = actor.getX();
        int fromY = actor.getY();
        int targetX = moveCoords[0];
        int targetY = moveCoords[1];
        Point target = new Point(targetX, targetY);
        int moveType = actor.checkMoveType(target);

        if (moveType == 3) {
            GameLogger.log(String.format(
                "MOVE AI: actor=%s(%s) from=(%d,%d) to=(%d,%d) %",
                actor.getName(), actor.getType(), fromX, fromY, targetX, targetY));
            return;
        }

        if (moveType == 2 && currentMode.equals("Future") && actor.getType().equals("Prey")) {
            Cell tc = grid.getCell(targetX, targetY);
            if (tc != null && !tc.isEmpty() && tc.getEntity() instanceof Food) {
                return;
            }
        }

        moveActor(actor, targetX, targetY);
        GameLogger.log(String.format(
            "MOVE AI: actor=%s(%s) from=(%d,%d) to=(%d,%d) %",
            actor.getName(), actor.getType(), fromX, fromY, targetX, targetY));
    }

    if (moveType == 2) {
        actor.triggerAbilityCooldown();
        GameLogger.log(actor.getName() + " used " + actor.getAbilityName());
    }
}

```

This method was designed with the help of the AIController class to make the AI perform the required moves. In addition, I handle different situations here and also write logs for them.

```

private boolean isAdjacent(Entity e1, Entity e2) {
    if (e1 == null || e2 == null) return false;
    int dx = Math.abs(e1.getX() - e2.getX());
    int dy = Math.abs(e1.getY() - e2.getY());
    int distance = Math.max(dx, dy);
    return distance == 1;
}

public String getWinner() {
    if (!player.isAlive()) {
        return apex.getName() + " (Player Eliminated)";
    }
    int pScore = player.getScore();
    int aScore = apex.getScore();
    int prScore = prey.getScore();
    if (pScore > aScore && pScore > prScore) return player.getName();
    if (aScore > pScore && aScore > prScore) return apex.getName();
    if (prScore > pScore && prScore > aScore) return prey.getName();
    return "Draw";
}

```

One of the helper methods I used is `isAdjacent`. It provides a quick check in special ability situations. Another helper method I used is `getWinner`. With this method, I handled both win and draw situations.

```

score = (minDistToThreat * 3.0) - minDistToFood;¶
Cell targetCell = grid.getCell(mx, my);¶
if (!targetCell.isEmpty() && targetCell.getEntity() instanceof Food) {¶
    score += 50.0;¶
}

```

In the AIController class, I created the getNextMoveForPrey method with a scoring algorithm. The goal is to move the Prey toward food while keeping it safe from the Apex. Since the food score is three times higher than the being-eaten penalty, I add an extra 50 points to encourage eating. So the Prey's priority is: 'Stay as far as possible from the Predator, but if it's safe, move closer to food.'

```

for (int[] move : possibleMoves) {¶
    int distToTarget = calculateDistance(move[0], move[1], closestTarget.getX(), closestTarget.getY());¶
    if (distToTarget < minMoveDist) {¶
        minMoveDist = distToTarget;¶
        bestMove = move;¶
    } else if (distToTarget == minMoveDist) {¶
        if (random.nextBoolean()) bestMove = move;¶
    }
}
return bestMove;

```

In the AIController, the getNextMoveForApex method is much simpler. It just makes the Apex move toward the closest target, because the only thing that matters for it is eating!

2.6 gui package

This package has all the Swing UI classes. It shows the game visually and handles user input. The GUI talks to the GameEngine, never touches grid cells directly—it just calls engine methods and refreshes the view.

- GameFrame: The main window. Switches between Start Screen and Game Screen with CardLayout. Has menu actions (New Game, Save, Load, Exit). Sends all game operations to GameEngine (startGame, loadSavedGame, updateInfoLabels, etc.).
- StartPanel: The setup screen at launch. Lets you enter Grid Size, Round Count, Era (via text fields, combo boxes, buttons). After checking inputs, it calls GameFrame.startGame(. . .). Also enables/disables Resume depending on whether a save file exists.
- GamePanel: The main canvas. Overrides paintComponent(Graphics g) to draw the grid,

animals, food, and move highlights. Calculates cell size, centers the grid, and uses cached images for speed. Mouse clicks are turned into grid coordinates with getGridCoordinates(...), then passed to the engine via GameFrame.

- InfoPanel: Shows live game info—Round, Remaining rounds, Era, Scores, and Ability Cooldowns (Apex, Predator, Prey). Updated with updateStats(...) whenever moves happen or the state changes.

```
public void startGame(int size, int rounds, String mode) {  
    try {  
        this.currentGridSize = size;  
        GameEngine newEngine = new GameEngine(size, rounds, mode);  
        initGameGUI(newEngine);  
    } catch (Exception e) {  
        e.printStackTrace();  
        JOptionPane.showMessageDialog(this, "Exception: " + e.getMessage());  
    }  
}
```

For example, in the GameFrame class, the startGame method creates a new GameEngine instance and starts the game.

```
private void confirmAndExit() {  
    if (engine == null || engine.isGameOver()) {  
        GameLogger.close();  
        System.exit(0);  
        return;  
    }  
  
    int choice = JOptionPane.showConfirmDialog(  
        this, "  
        Do you want save before exit?",  
        "Exit Confirmation",  
        JOptionPane.YES_NO_CANCEL_OPTION  
    );  
  
    if (choice == JOptionPane.YES_OPTION) {  
        FileManager.saveGame(engine);  
        JOptionPane.showMessageDialog(this, "Saved. See you soon!");  
        System.exit(0);  
    } else if (choice == JOptionPane.NO_OPTION) {  
        System.exit(0);  
    }  
}
```

When the player wants to quit the game, the confirmAndExit method ensures that a save is triggered, preventing the possibility of leaving without saving.

```

>>   for (int x = 0; x < gridSize; x++) {¶
>>     for (int y = 0; y < gridSize; y++) {¶
>>       int px = xOffset + (x * cellSize); ¶
>>       int py = yOffset + (y * cellSize); ¶
>>       ¶
>>       if (specialMask[x][y]) {¶
>>         g2d.setColor(new Color(255, 215, 0, 90)); ¶
>>         g2d.fillRect(px + 1, py + 1, cellSize - 1, cellSize - 1); ¶
>>       } else if (normalMask[x][y]) {¶
>>         g2d.setColor(new Color(46, 204, 113, 90)); ¶
>>         g2d.fillRect(px + 1, py + 1, cellSize - 1, cellSize - 1); ¶
>>       } ¶
>>       ¶
>>       g2d.setColor(Color.LIGHT_GRAY); ¶
>>       g2d.drawRect(px, py, cellSize, cellSize); ¶

```

The `paintComponent()` method in `GamePanel.java` is basically my canvas. It's the place where I rebuild the visual world of the game from scratch every second (or whenever a repaint is requested). Inside this method, I color the background and the grids.

<pre> private BufferedImage getImageFor(Entity entity){¶ String key;¶ ¶ if(entity instanceof Food){¶ Food food = (Food) entity;¶ ¶ key = food.getName().toLowerCase().replaceAll("[\\s-]", "");¶ }else if(entity instanceof Animal){¶ Animal animal = (Animal) entity;¶ key = animal.getName().toLowerCase().replaceAll("[\\s-]", "");¶ }else{¶ return null;¶ }¶ ¶ if(imageCache.containsKey(key)){¶ return imageCache.get(key);¶ }¶ ¶ BufferedImage img = loadImage(key);¶ if(img != null){¶ imageCache.put(key, img);¶ }¶ return img;¶ } </pre>	<pre> private BufferedImage loadImage(String key){¶ String path = "/images/" + key + ".png"; ¶ ¶ try{¶ URL url = getClass().getResource(path);¶ if(url == null){¶ System.err.println("Image not found: " + path);¶ return null;¶ }¶ return ImageIO.read(url);¶ }catch(IOException e){¶ return null;¶ }¶ } </pre>
---	--

With these two methods, I perform the image lookup for all my entities. The `loadImage` method loads from the classpath, and the `getImageFor` method uses `loadImage` to find the image and add it to the `GamePanel`.

```

private JPanel createPlayerPanel() {
    JPanel panel = createStyledPanel(new Color(255, 220, 180), "PREDATOR");
    ...
    lblPlayerName = new JLabel("Name: -");
    lblPlayerScore = new JLabel("Score: -");
    lblPlayerCooldown = new JLabel("Ability Cooldown: -");

    styleLabel(lblPlayerName);
    styleLabel(lblPlayerScore);
    styleLabel(lblPlayerCooldown);

    panel.add(lblPlayerName);
    panel.add(lblPlayerScore);
    panel.add(lblPlayerCooldown);
    return panel;
}

```

In the **InfoPanel**, I drew the panel located on the right side of the game screen and tried to keep it constantly updated.

```

private void styleLabel(JLabel label) {
    label.setFont(new Font("Monospaced", Font.PLAIN, 12));
    label.setAlignmentX(Component.CENTER_ALIGNMENT);
}

public void updateStats(int round, int maxRounds, String era, Animal apex, Animal player, Animal prey) {
    ...
    lblRound.setText("Round: " + round);
    lblEra.setText("Era: " + era);

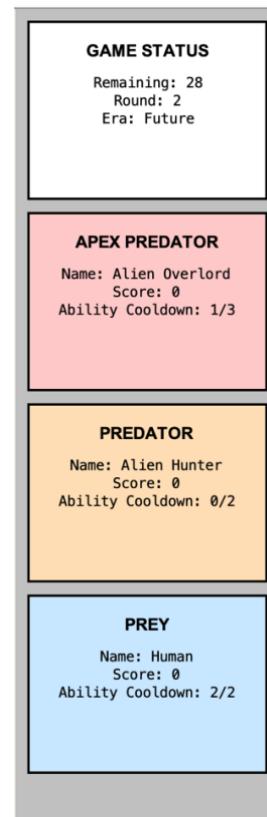
    int remaining = Math.max(0, maxRounds - round);
    lblRemaining.setText("Remaining: " + remaining);

    if (apex != null) {
        ...
        lblApexName.setText("Name: " + apex.getName());
        lblApexScore.setText("Score: " + apex.getScore());
        lblApexCooldown.setText(
            ...
            "Ability Cooldown: " + apex.getAbilityCooldown() + "/" + apex.getMaxAbilityCooldown()
        );
    } else {
        ...
        lblApexName.setText("Name: None");
        lblApexScore.setText("0");
        lblApexCooldown.setText("0");
    }

    ...
    if (player != null) {
        ...
        lblPlayerName.setText("Name: " + player.getName());
        lblPlayerScore.setText("Score: " + player.getScore());
        ...
        lblPlayerCooldown.setText(
            ...
            "Ability Cooldown: " + player.getAbilityCooldown() + "/" + player.getMaxAbilityCooldown()
        );
    } else {
        ...
        lblPlayerName.setText("Name: None");
        lblPlayerScore.setText("0");
        lblPlayerCooldown.setText("0");
    }

    if (prey != null) {
        ...
        lblPreyName.setText("Name: " + prey.getName());
        lblPreyScore.setText("Score: " + prey.getScore());
        ...
        lblPreyCooldown.setText(
            ...
            "Ability Cooldown: " + prey.getAbilityCooldown() + "/" + prey.getMaxAbilityCooldown()
        );
    } else {
        ...
        lblPreyName.setText("Name: None");
        lblPreyScore.setText("0");
        lblPreyCooldown.setText("0");
    }
}

```



With the `updateStats` method, I wrote the updated ability cooldowns and scores at the end of each round. In addition, I refreshed the Game Status by updating the round count and the remaining rounds.

```

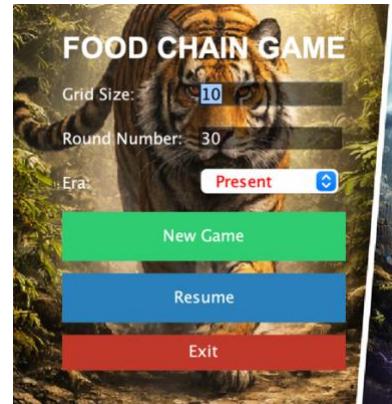
gbc.gridx++;|
gbc.gridx=0;|
JLabel.lblGrid = new JLabel("Grid Size:");|
lblGrid.setForeground(Color.WHITE);|
add(lblGrid, gbc);|

gbc.gridx = 1;|
txtGridSize = new JTextField("20");|
txtGridSize.setForeground(Color.WHITE);|
txtGridSize.setBackground(new Color(0, 0, 0, 140));|
txtGridSize.setCaretColor(Color.WHITE);|
add(txtGridSize, gbc);|

gbc.gridx++;|
gbc.gridx=0;|
JLabel.lblRounds = new JLabel("Round Number:");|
lblRounds.setForeground(Color.WHITE);|
add(lblRounds, gbc);|

gbc.gridx=1;|

```



Finally, in the StartPanel, I implemented the buttons on the start screen, handled input validation, and displayed error messages.

3. References

- <https://www.w3schools.com/java/default.asp>
- <https://www.geeksforgeeks.org/java/>
- <https://stackoverflow.com/questions>
- <https://www.tutorialspoint.com/java/index.htm>
- <https://youtu.be/Kmgo00avvEw>
- <https://youtu.be/ScUJx4aWRi0>
- <https://youtu.be/U28eKSLI7pw>
- <https://youtu.be/Su4-yQhzbSw>
- <https://youtu.be/ocb3x0TeoUw>
- <https://youtu.be/1XAfapkBQjk>
- <https://youtu.be/OIozDnGYqlU>

- https://youtu.be/_nmm0nZqIIY
- <https://youtu.be/Hiv3gwJC5kw>
- <https://youtube.com/playlist?list=PLyt2v1LVXYS0> -
[nE4ZMfJvhjDlx9kRqL&si=WoHBVTwLQTpo2hS](https://youtu.be/nE4ZMfJvhjDlx9kRqL&si=WoHBVTwLQTpo2hS)
- <https://youtu.be/QEF62Fm81h4>
- <https://www.geeksforgeeks.org/java-util-timer-class-java/>
- https://www.tutorialspoint.com/java/util/timer_delay.htm