1. How does each file work?

Game.java:

- This file serves as the entry point of the application. It creates the main window of the game where users can select the board size and start the game.

- It uses a `JComboBox` for board size selection and a `JButton` to start the game.

- When the "Start Game" button is clicked, the selected board size is passed to the `GameBoard` constructor to create a new game board, and the main window is disposed of.

MainMenu.java:

- This file provides the main menu interface for the game, allowing users to start a new game, view high scores, or exit the application.

- It creates a `JFrame`with options to select the board size and start the game, view high scores, or exit.

- The "Start Game" button launches a new game with the selected board size, the "High Scores" button displays the high scores, and the "Exit" button closes the application.

GameBoard.java:

- This is the core file that manages the game logic. It handles the game board's rendering, player and enemy movements, upgrades, and collision detection.

- The game board is initialized with a specific size, and various game elements (player, enemies, dots, and upgrades) are added to the board.

- It includes a game loop managed by a `Timer` that updates the game state and checks for collisions at regular intervals.

- The `populateDots` method adjusts the number of dots based on the board size.

- The `applyUpgrade` method handles the effects of collected upgrades (e.g., speed boost, invisibility).

HighScores.java:

- Manages loading and saving high scores to a file.

- The `loadHighScores` method reads high scores from a file and returns them as a list. If the file does not exist, it returns an empty list.

- The `saveHighScores` method writes high scores to a file.

- This file ensures that high scores are persisted across game sessions.

Player.java:

- Represents the player character in the game.
- It stores the player's position and handles movement.
- The `draw` method renders the player on the game board using graphics.

Enemy.java:

- Represents the enemies in the game.
- Stores the enemy's position and handles movement towards the player.
- The `setVisible` method controls the enemy's visibility, used for the invisibility upgrade.
- The `draw` method renders the enemy on the game board.

Upgrade.java:

- Represents the upgrades (power-ups) in the game.
- Stores the upgrade's position and type (e.g., speed, visibility).
- The `draw` method renders the upgrade on the game board.

Dot.java

- Represents the dots that the player collects for points.
- Stores the dot's position and value (small or big).
- The `draw` method renders the dot on the game board with different sizes based on its value.

2. What are the limitations of the program?

Scalability: The game may not scale well to very large board sizes or a high number of enemies and dots due to potential performance issues.

Graphics: The game uses simple graphics and may not have the visual appeal of more advanced games.

Thread Synchronization: While basic thread synchronization is implemented, more complex game logic might require more robust thread management.

High Score Management: High scores are managed per player but do not include features like a global leaderboard or online synchronization.

3. What are the known issues of the program?

Performance: As the board size increases, the number of dots and enemies also increases, which might lead to performance degradation.

4. How does the program work in conjunction with all classes?

- The program is structured with clear responsibilities for each class. `Game.java` initializes the game, `MainMenu.java` provides the main menu, and `GameBoard.java` manages the game logic.

- `Player`, `Enemy`, `Upgrade`, and `Dot` classes represent game elements and are managed by `GameBoard`.

- `HighScores.java` manages high score persistence.

- The main game loop in `GameBoard` updates the state of all game elements, checks for collisions, and applies game logic, ensuring smooth interaction between classes.

5. What's the logic of each code file, especially GameBoard?

Game.java: Starts the game and handles board size selection.

MainMenu.java: Provides options to start a new game, view high scores, and exit.

GameBoard.java: Handles the main game loop, updating game state, rendering elements, and managing collisions and upgrades.

- Initializes the game board based on the selected size.

- Populates the board with dots, enemies, and upgrades.

- Manages player and enemy movements and interactions.

- Applies upgrades and updates the score.

- Checks for game-over conditions and handles high score saving.

Player.java: Manages player position and rendering.

Enemy.java: Manages enemy position, movement, and rendering.

Upgrade.java: Manages upgrade position, type, and rendering.

Dot.java: Manages dot position, value, and rendering.

HighScores.java: Manages loading and saving high scores.

6. What parts of the code depict: inheritance, collections, interfaces or abstract classes, lambda expressions, Java Generics?

Inheritance: The `GameBoard` class extends `JFrame`, inheriting its properties and methods.

Collections: The program uses `ArrayList` to manage collections of players, enemies, upgrades, and dots.

Interfaces: The program uses interfaces such as `ActionListener` and `KeyListener` for event handling.

Lambda Expressions: The program uses lambda expressions in the `ActionListener`for the start game button and in the `Timer` for updating the game state.

Java Generics: The use of `ArrayList<Player>`, `ArrayList<Enemy>`, `ArrayList<Upgrade>`, and `ArrayList<Dot>` in `GameBoard` demonstrates Java Generics.

7. Parts of the code depict it's GUI components, Functional Interfaces, Serialization, Collections, Multithreading, File I/O.

GUI Components: The use of `JFrame`, `JButton`, `JLabel`, `JComboBox`, and `JPanel` for building the user interface.

Functional Interfaces: Usage of `ActionListener` and lambda expressions.

Serialization: Handled in `HighScores.java`for loading and saving high scores.

Collections: `ArrayList` is used to manage game elements.

Multithreading: The `Timer` class in `GameBoard.java`for updating the game state periodically.

File I/O: Reading and writing high scores in `HighScores.java`.