Supervision 2 Work

Github Repo: https://github.com/JamieLittle16/OOP-Supervision-2

1. Explain the differences between a class, an abstract class and an interface in Java (please provide example code to illustrate your understanding of each).

A standard class defines a complete blueprint for an object. The state and behaviour is complete, and an object can be directly instantiated of this class.

```
class Car {
	int year;

	public void drive() {
		System.out.println("Car is driving");
	}
}
```

An abstract class is an incomplete object blueprint. It can have functional methods and state, but can also have blank methods that child classes must fill it. An object instance of an abstract class cannot be made; it is designed to be inheritted by another class.

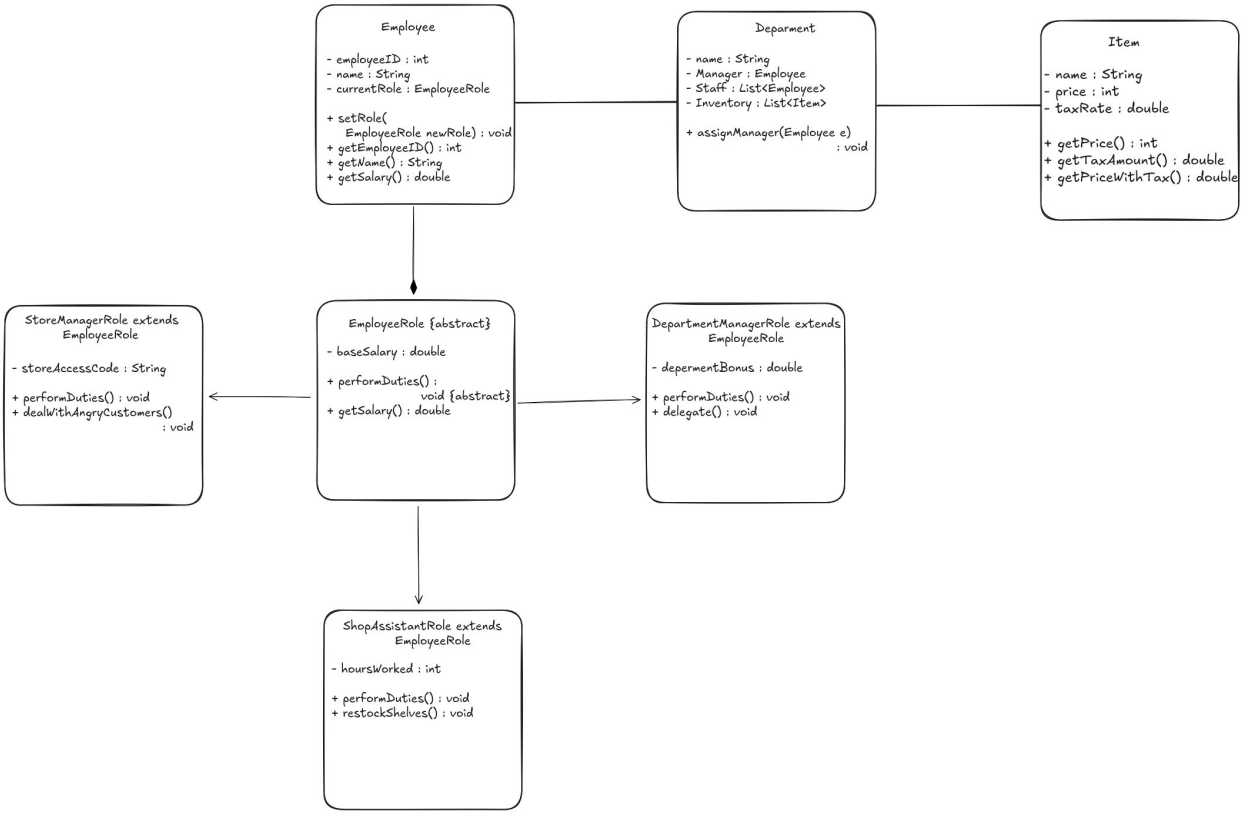
```
abstract class Vehicle {
	int year;

	public void drive() {
		System.out.println("Vechile is driving");
	}

	public abstract void refuel();
}
```

An interface defines a structure a child class must follow. It lays out a series of methods that child classes must implement without offering an implementation (default implementations can be made but are generally avoided). A class can implement multiple interfaces while a class can only inherit from a single parent class.

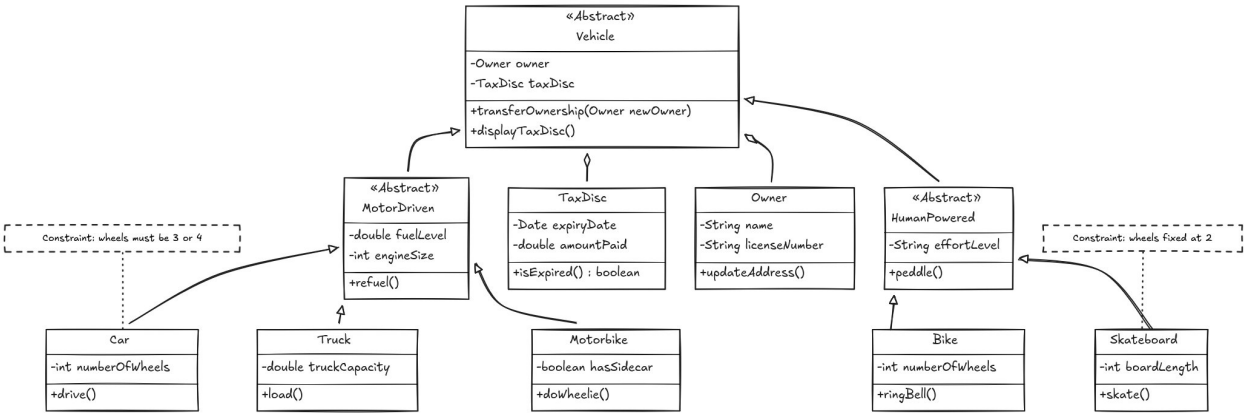
```
public interface Driveable {
	void drive();
}
```

2. Provide detailed UML class diagrams (identifying example fields and methods) for the following two scenarios:

a. A shop is composed of a series of departments, each with its own manager. There is also a store manager and many shop assistants. Each item sold has a price and a tax rate.



b. Vehicles are either motor-driven (cars, trucks, motorbikes) or human-powered (bikes, skateboards). All cars have 3 or 4 wheels and all bikes have two wheels. Every vehicle has an owner and a tax disc.



3. Explain the following concepts using examples from your UML diagrams in question 2 to illustrate your answer.

a. Modularity

Modularity is the process of breaking a large, complex problem into a series of smaller, self-contained parts (modules/classes) that each have their own distinct purpose and responsibility. Each module should be fully replacable (i.e. If the internal logic is updated, it should not effect the rest of the code).

For example, the TaxDisc class. Instead of cramming variables like taxExpiryDate and taxAmount directly into Vehicle or Car, a seperate independent class was created. If legislation is update changing how tax discs work, only this class needs to be changed.

b. Code Re-use

The practise of writing a piece of code just once, and reusing it in multiple places to reduce redundancy. This could be through inheritance (or importing a class into a different project).

For example, the Vehicle class; it is a superclass, so its state and behaviour (e.g. transferOwnerShip) is only contained within this class, but it is included within Car, Truck, Bike, Skateboard. The benefit of this is the methods do not need to be copied 4 seperate times, but can instead be inheritted. They only need to be redefined if there is some change of behaviour is required.

c. Encapsulation

Encapsulation is the idea of bundling data and methods into a single unit (class), and then restricted which parts of this can be accessed from outside. This practise ensures that other classes can only access it through set interfaces. This means the internal workings of the class can be obscured. As a result, the internal logic of this class can be overhauled without breaking any code using the class (linking to modularity) – so long as the interface methods are implemented. Furthering this, other classes cannot break the internal workings of the class by updating its internal variables.

For example the Item class makes the variable price private. This means that price can only be accessed through the getPrice() method. As a result the price cannot be updated directly outside of the class. If another class required setting an iterm price, we could add a setPrice() method. This method would ensure that only legal price values are set (i.e. They are not negative). This shifts control to the item class, allowing it to reject negative values that could break its internal logic.

4. Explain what is meant by (dynamic) polymorphism in OOP and explain why it is useful, illustrating your answer with an example.

Dynamic polymorphism is decided at run-time, and always uses the method in the child object (even if it is type cast as a parent object). However, since type is determined at run-time, type errors may cause run-time crashes, compared to static polymorphism, which is determined at compile time.

This may be useful, since different objects, all sharing a parent class, can be added to an array or linked list with type of the parent class. When accessing objects from this array, they will be

type cast to the type of the parent object. However since java uses dynamic polymorphism, if a method is accessed, the overrided (if defined) version in the child object will be used, instead of the parent implementation. This means that the group of objects can all be stored in a single array, and they can each have different implementations of the same method.

```
abstract class Vehicle() {
	public void drive() { System.out.println("Vehicle is driving"); }
}

class Car extends Vehicle {
	public void drive() { System.out.println("Car is driving");  }
}

class Truck extends Vehicle {
	public void drive() { System.out.println("Truck is driving");  }
}

class Demo {
	public static void main(String[] args) {
		Vehicle[] vehicles = new vehicles[] { new Car(), new Truck() };

		vehicles[0].drive();
		vehicles[1].drive();
	}
}
```

This code will output:

Car is driving

Truck is driving

This highlights how the individual implementations of child classes is executed despite the Car and Truck object being cast as a Vehicle object.

5. Imagine you have two classes: Employee (which represents being an employee) and Ninja (which represents being a Ninja). An Employee has both state and behaviour; a Ninja has only behaviour. You need to represent an employee who is also a ninja (a

common problem in the real world). By creating only one interface and only one class (NinjaEmployee), show how you can do this without having to copy method implementation code from either of the original classes.

Since Java does not have multiple inheritance, the NinjaEmployee class cannot simply just inherit both the Ninja and Employee classes. Instead we can use the delegation design pattern. Since employee has state and behaviour while Ninja only has behaviour, it makes sense to inherit from the Employee class for our base. To gain the behaviour of the Ninja class, an interface is created, which forces the Ninja methods to be implemented (this interface can then be implemented by both the Ninja and NinjaEmployee classes). To ensure that the NinjaEmployee class mimics the behaviour of a Ninja, an internal Ninja object is created within the class. When one of the ninja functions is called, we just call this function on the internal ninja object.

```
interface INinja {
  void throwShuriken();

  void sneak();
}

class Ninja implements INinja {
  public void throwShuriken() { System.out.println("Swish!"); }

  public void sneak() { System.out.println("..."); }
}

class Employee {
  String name;

  public void doOfficeWork() { System.out.println("Typing..."); }
}

class NinjaEmployee extends Employee implements INinja {

  private Ninja hiddenNinja = new Ninja();
```

```
  public void throwShuriken() { hiddenNinja.throwShuriken(); }

  public void sneak() { hiddenNinja.sneak(); }
}
```

6. Describe how garbage collection works in Java and the issue with finalizers.

New objects created in java will be placed into the Eden space. Garbage collection cycles happen more frequently here. If an object survives its first few garbage collection sweeps, it is moved into the Survivors space. In the survivor space garbage collection sweeps can happen less frequently (as it is unlikely the object was only created for some very short-lived purpose). If an object survives many garbage collection cycles, it is moved into the Tenured space. Objects in the Tenured space have been around for a very long time, so it is unlikely that they will be removed any time soon. As a result garbage collection sweeps are infrequent in the tenured space.

To carry out a garbage collection cycle, Java uses a “Root Reachibility” approach. It starts by looking at any local variables on the stack or static variables. From these variables it follows every refrence recursively, marking each object reached as alive. Once marking is complete, a sweep occurs, where all unmarked objects are deleted and the memory is reclaimed.

Finalizers are deprecated in recent versions of Java. While using a finalizer has a significant overhead, the main reason for their deprecation is unpredictability. The programmer has no control over the garbage collector. If an object becomes unreachable it possible that it will not be destoryed for a very long time. This means it cannot be accurately known when whatever cleanup operations within the finalizer will be run.

7. Explain (with an example) how you could change the output of the following code without changing

the testOutput() function:

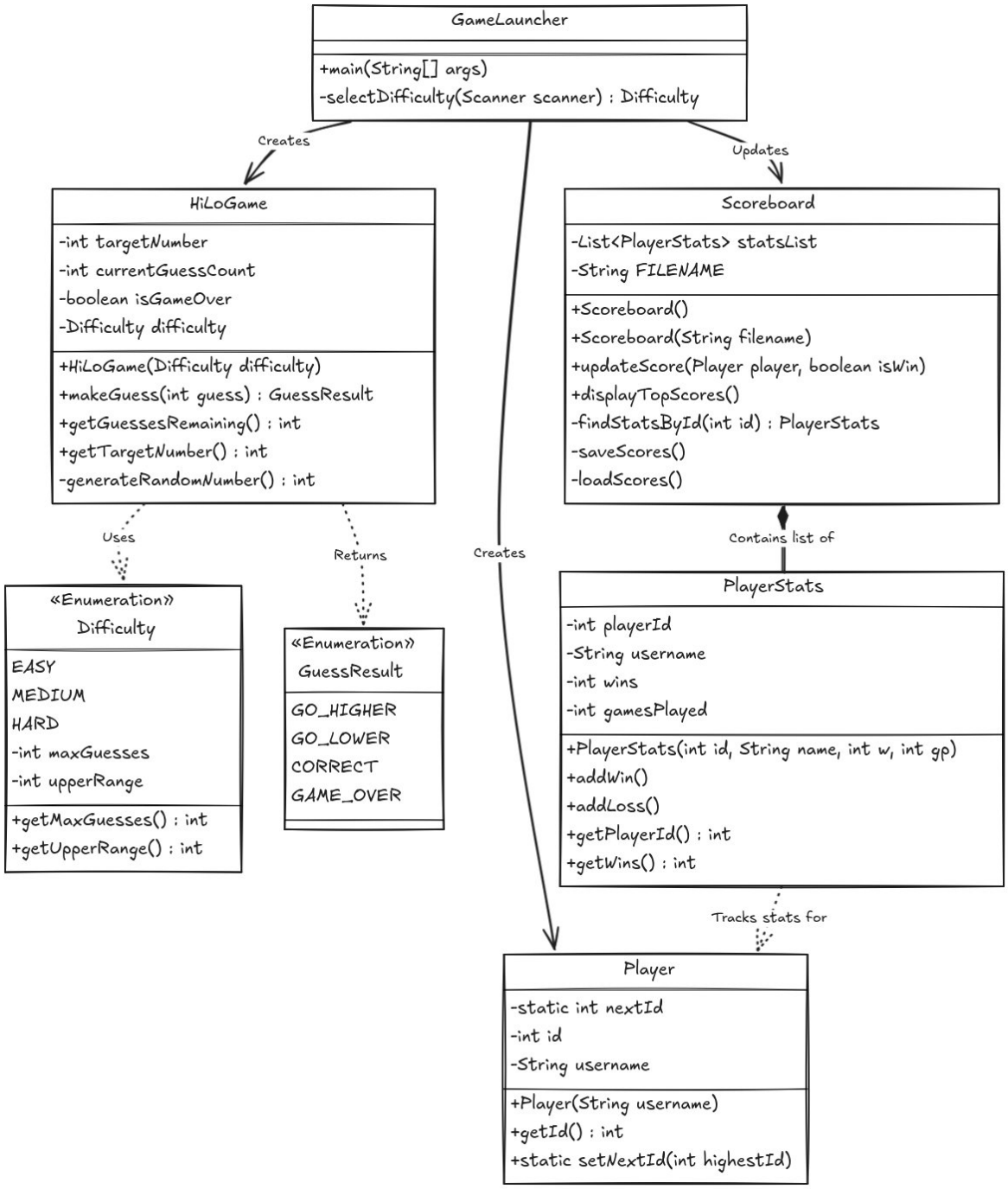
```
public void testOutput() {
        Person p = new Person(“Joe”, “Bloggs”);
        System.out.println(“Person details: “ + p);
}
```

When the “+” operator is used to concatenate an object to a String object, the `toString()` method is called on the object (This method is inheritted from the Object calss). By default this will output the hashcode of the object. However, within the Person class, the method can be overriden.

```
  @Override
  public String toString() {
```

```
        return firstName + " " + lastName;
    }
```

8. Design (using a UML class diagram) a HiLo guessing game. This game should select a random integer and the objective is for the user to guess the number that the computer has chosen. Each guess should result either in a response saying to “go higher”, “go lower” or “correct”. There should be a maximum number of guesses before the computer responds with a “Game over” message and prompts the user to have a rematch. There should be a scoreboard recording player information and game statistics. The game difficulty should also be configurable.



9. Implement your design from Q8 in Java. It can be as simple or complex as you like but should compile and run at the very least. (Be prepared to talk about/demo it during the supervision).

GameLauncher:

```
class GameLauncher {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    Scoreboard scoreboard = new Scoreboard();

    System.out.println("HiLo Guessing Game");

    boolean keepPlaying = true;
    while (keepPlaying) {
      System.out.println("Enter your username:");
      String username = scanner.next();
      Player player = new Player(username);

      Difficulty difficulty = selectDifficulty(scanner);

      HiloGame game = new HiloGame(difficulty);
      System.out.println("\nGame Started! Guess numbers between 1 and " +
difficulty.getUpperRange());
      System.out.println("You have " + difficulty.getMaxGuesses() + " guesses.");

      boolean gameRunning = true;
      while (gameRunning) {
        System.out.println("Enter your guess: ");

        while (!scanner.hasNextInt()) {
          System.out.println("Not a number, try again");
          scanner.next();
```

```
        }
        int guess = scanner.nextInt();

        GuessResult result = game.makeGuess(guess);

        switch (result) {
          case GO_HIGHER:
            System.out.println(" -> Go Higher!");
            System.out.println("   (Guesses left: " + game.getGuessesRemaining() +
")");
            break;

          case GO_LOWER:
            System.out.println(" -> Go Lower!");
            System.out.println("   (Guesses left: " + game.getGuessesRemaining() +
")");
            break;

          case CORRECT:
            System.out.println("\n CORRECT! You won!");
            scoreboard.updateScore(player, true); // Record Win
            gameRunning = false; // Exit loop
            break;

          case GAME_OVER:
            System.out.println("\n GAME OVER! You ran out of guesses.");
            System.out.println("The number was: " + game.getTargetNumber());
            scoreboard.updateScore(player, false); // Record Loss
            gameRunning = false; // Exit loop
            break;
        }
```

```
    }
    scoreboard.displayTopScores();

    System.out.println("Do you want to play again? (Y/N)");
    String answer = scanner.next();
    if (!answer.equalsIgnoreCase("Y")) {
      keepPlaying = false;
    }
  }
  System.out.println("Thanks for playing!");
  scanner.close();
}

private static Difficulty selectDifficulty(Scanner scanner) {
  System.out.println("\nSelect Difficulty:");
  System.out.println("1. Easy   (10 guesses, Range 1-20)");
  System.out.println("2. Medium (7 guesses,  Range 1-50)");
  System.out.println("3. Hard   (5 guesses,  Range 1-100)");
  System.out.print("Choice: ");

  int choice = 0;
  if (scanner.hasNextInt()) {
    choice = scanner.nextInt();
  } else {
    scanner.next(); // clear bad input
  }

  switch (choice) {
    case 1:
      return Difficulty.EASY;
    case 2:
```

```
        return Difficulty.MEDIUM;
      case 3:
        return Difficulty.HARD;
      default:
        System.out.println("Invalid choice. Defaulting to Medium.");
        return Difficulty.MEDIUM;
    }
  }
}
```

HiloGame:

```
class HiloGame {
  private int currentGuessCount;
  private final int targetNumber;
  private final Difficulty difficulty;
  private boolean isGameOver;

  public HiloGame(Difficulty difficulty) {
    this.difficulty = difficulty;
    this.currentGuessCount = 0;
    this.isGameOver = false;

    this.targetNumber = generateRandomNumber();
  }

  public GuessResult makeGuess(int guess) {
    if (isGameOver) {
      return GuessResult.GAME_OVER;
    }
    currentGuessCount++;
```

```
    if (guess == targetNumber) {
      return GuessResult.CORRECT;
    }
    if (currentGuessCount >= difficulty.getMaxGuesses()) {
      return GuessResult.GAME_OVER;
    }
    if (guess < targetNumber) {
      return GuessResult.GO_HIGHER;
    } else {
      return GuessResult.GO_LOWER;
    }
  }

  private int generateRandomNumber() {
    Random rand = new Random();
    return rand.nextInt(difficulty.getUpperRange()) + 1;
  }

  public int getTargetNumber() {
    return targetNumber;
  }

  public int getGuessesRemaining() {
    return difficulty.getMaxGuesses() - currentGuessCount;
  }
}
```

Difficulty:

```
public enum Difficulty {
  // Difficulty Levels
  EASY(10, 20), // 10 guesses, range 1-20
```

```
  MEDIUM(7, 50), // 7 guesses, range 1-50
  HARD(5, 100); // 5 guesses, range 1-100

  // Game settings
  private final int maxGuesses;
  private final int upperRange;

  Difficulty(int maxGuesses, int upperRange) {
    this.maxGuesses = maxGuesses;
    this.upperRange = upperRange;
  }

  public int getMaxGuesses() {
    return maxGuesses;
  }

  public int getUpperRange() {
    return upperRange;
  }
}
```

GuessResult:

```
public enum GuessResult {
  GO_HIGHER,
  GO_LOWER,
  CORRECT,
  GAME_OVER
}
```

Scoreboard:

```
class Scoreboard {
```

```
private List<PlayerStats> statsList;
private final String FILENAME;

public Scoreboard() {
  this.statsList = new ArrayList<>();
  FILENAME = "scoreboard.csv";
  loadScores();
}

public Scoreboard(String fileName) {
  this.statsList = new ArrayList<>();
  FILENAME = fileName;
  loadScores();
}

public void updateScore(Player player, boolean isWin) {
  PlayerStats stats = findStatsById(player.getId());

  if (stats == null) {
    stats = new PlayerStats(player.getId(), player.getUsername(), 0, 0);

    statsList.add(stats);
  }

  if (isWin) {
    stats.addWin();
  } else {
    stats.addLoss();
  }
  saveScores();
}
```

```
  public void displayTopScores() {
    statsList.sort((p1, p2) -> Integer.compare(p2.getWins(), p1.getWins()));

    System.out.println("\n============= HALL OF FAME =============");
    System.out.println(String.format("%-4s | %-15s | %-5s | %-5s", "ID", "Player",
"Wins", "Played"));
    System.out.println("---------------------------------------");

    for (PlayerStats p : statsList) {
      System.out.println(String.format("%-4d | %-15s | %-5d | %-5d",
          p.getPlayerId(),
          p.getUsername(),
          p.getWins(),
          p.getGamesPlayed()));
    }
    System.out.println("=========================================\n");
  }

  private PlayerStats findStatsById(int id) {
    for (PlayerStats ps : statsList) {
      if (ps.getPlayerId() == id) {
        return ps;
      }
    }
    return null;
  }

  private void saveScores() {
    try (PrintWriter writer = new PrintWriter(new FileWriter(FILENAME))) {
      for (PlayerStats ps : statsList) {
```

```
        // CSV Format: id,username,wins,total
        writer.println(ps.getPlayerId() + "," +
            ps.getUsername() + "," +
            ps.getWins() + "," +
            ps.getGamesPlayed());
      }
    } catch (IOException e) {
      System.err.println("Error saving scoreboard: " + e.getMessage());
    }
  }

  private void loadScores() {
    File file = new File(FILENAME);
    if (!file.exists())
      return;

    int highestIdFound = 0;

    try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
      String line;
      while ((line = reader.readLine()) != null) {
        String[] parts = line.split(",");
        if (parts.length == 4) {
          int id = Integer.parseInt(parts[0]);
          String name = parts[1];
          int wins = Integer.parseInt(parts[2]);
          int total = Integer.parseInt(parts[3]);

          statsList.add(new PlayerStats(id, name, wins, total));

          if (id > highestIdFound) {
```

```
            highestIdFound = id;
          }
        }
      }
      Player.setNextId(highestIdFound);

    } catch (IOException e) {
      System.err.println("Error loading scoreboard.");
    }
  }
}
```

PlayerStats:

```
class PlayerStats {
  private int playerId;
  private String username;
  private int wins;
  private int gamesPlayed;

  public PlayerStats(int playerId, String username, int wins, int gamesPlayed) {
    this.playerId = playerId;
    this.username = username;
    this.wins = wins;
    this.gamesPlayed = gamesPlayed;
  }

  public void addWin() {
    this.wins++;
    this.gamesPlayed++;
  }
```

```
  public void addLoss() {
    this.gamesPlayed++;
  }

  public int getPlayerId() {
    return playerId;
  }

  public String getUsername() {
    return username;
  }

  public int getWins() {
    return wins;
  }

  public int getGamesPlayed() {
    return gamesPlayed;
  }
}
```

Player:

```
class Player {
  private static int nextId = 1;

  private int id;
  private String username;

  public Player(String username) {
    this.username = username;
    this.id = nextId;
```

```
    nextId++;
  }

  public String getUsername() {
    return username;
  }

  public int getId() {
    return id;
  }

  public static void setNextId(int highestId) {
    nextId = highestId + 1;
  }
}
```