

```
import java.io.*;
import java.util.*;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

// Custom exception for game save operations
class GameSaveException extends Exception {
    public GameSaveException(String message) {
        super(message);
    }

    public GameSaveException(String message, Throwable cause) {
        super(message, cause);
    }
}

// Enum for achievement types
enum AchievementType {
    BRONZE, SILVER, GOLD, PLATINUM, SECRET
}

// Achievement class
class Achievement implements Serializable {
    private static final long serialVersionUID = 1L;

    private String id;
    private String name;
    private String description;
    private AchievementType type;
    private LocalDateTime unlockedDate;
    private boolean unlocked;
    private int points;

    public Achievement(String id, String name, String description, AchievementType type, int points) {
        this.id = id;
        this.name = name;
        this.description = description;
        this.type = type;
        this.points = points;
        this.unlocked = false;
        this.unlockedDate = null;
    }

    public void unlock() {
        if (!unlocked) {
            this.unlocked = true;
            this.unlockedDate = LocalDateTime.now();
        }
    }

    // Getters
    public String getId() { return id; }
    public String getName() { return name; }
    public String getDescription() { return description; }
    public AchievementType getType() { return type; }
    public int getPoints() { return points; }
    public boolean isUnlocked() { return unlocked; }
    public LocalDateTime getUnlockedDate() { return unlockedDate; }

    @Override
```

```

public String toString() {
    String status = unlocked ? "UNLOCKED" : "LOCKED";
    String dateStr = unlockedDate != null ?
        unlockedDate.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm")) : "N/A";
    return String.format("%s [%s] - %s (%d pts) - %s at %s",
                         name, type, description, points, status, dateStr);
}
}

// Game Item class
class GameItem implements Serializable {
    private static final long serialVersionUID = 1L;

    private String id;
    private String name;
    private String description;
    private ItemType type;
    private int quantity;
    private int maxStackSize;
    private boolean isEquipped;
    private int durability; // For equipment
    private int value; // Gold value

    public GameItem(String id, String name, String description, ItemType type,
                   int maxStackSize, int value) {
        this.id = id;
        this.name = name;
        this.description = description;
        this.type = type;
        this.maxStackSize = maxStackSize;
        this.value = value;
        this.quantity = 1;
        this.durability = 100;
        this.isEquipped = false;
    }

    public boolean combine(GameItem other) {
        if (this.id.equals(other.id) && this.quantity < maxStackSize) {
            int availableSpace = maxStackSize - this.quantity;
            int transferAmount = Math.min(availableSpace, other.quantity);
            this.quantity += transferAmount;
            other.quantity -= transferAmount;
            return other.quantity == 0;
        }
        return false;
    }

    public void use() {
        if (type == ItemType.CONSUMABLE && quantity > 0) {
            quantity--;
        } else if (type == ItemType.EQUIPMENT) {
            isEquipped = !isEquipped;
        }
    }

    // Getters and setters
    public String getId() { return id; }
    public String getName() { return name; }
    public String getDescription() { return description; }
    public ItemType getType() { return type; }
    public int getQuantity() { return quantity; }
    public void setQuantity(int quantity) { this.quantity = quantity; }
}

```

```
public int getValue() { return value; }
public boolean isEquipped() { return isEquipped; }
public int getDurability() { return durability; }
public void setDurability(int durability) { this.durability = durability; }

@Override
public String toString() {
    String equipped = isEquipped ? " [EQUIPPED]" : "";
    String durabilityStr = (type == ItemType.EQUIPMENT) ?
        String.format(" Durability: %d/100", durability) : "";
    return String.format("%s x%d%s - %s (Value: %d gold)",
        name, quantity, equipped, durabilityStr, description, value);
}

// Item types enum
enum ItemType {
    WEAPON, ARMOR, CONSUMABLE, MATERIAL, QUEST_ITEM, KEY_ITEM
}

// Player Inventory class
class PlayerInventory implements Serializable {
    private static final long serialVersionUID = 1L;

    private List<GameItem> items;
    private int maxCapacity;
    private int gold;

    public PlayerInventory(int maxCapacity) {
        this.items = new ArrayList<>();
        this.maxCapacity = maxCapacity;
        this.gold = 0;
    }

    public boolean addItem(GameItem newItem) {
        // Try to stack with existing items first
        for (GameItem item : items) {
            if (item.combine(newItem)) {
                return true;
            }
        }

        // If couldn't stack and has space, add new item
        if (items.size() < maxCapacity) {
            items.add(newItem);
            return true;
        }
    }

    return false; // Inventory full
}

public boolean removeItem(String itemId, int quantity) {
    Iterator<GameItem> iterator = items.iterator();
    while (iterator.hasNext()) {
        GameItem item = iterator.next();
        if (item.getId().equals(itemId)) {
            if (item.getQuantity() > quantity) {
                item.setQuantity(item.getQuantity() - quantity);
            } else {
                iterator.remove();
            }
        }
    }
    return true;
}
```

```

        }
    }
    return false;
}

public GameItem getItem(String itemId) {
    return items.stream()
        .filter(item -> item.getId().equals(itemId))
        .findFirst()
        .orElse(null);
}

public List<GameItem> getEquippedItems() {
    List<GameItem> equipped = new ArrayList<>();
    for (GameItem item : items) {
        if (item.isEquipped()) {
            equipped.add(item);
        }
    }
    return equipped;
}

// Getters and setters
public List<GameItem> getItems() { return new ArrayList<>(items); }
public int getGold() { return gold; }
public void setGold(int gold) { this.gold = gold; }
public void addGold(int amount) { this.gold += amount; }
public boolean removeGold(int amount) {
    if (gold >= amount) {
        gold -= amount;
        return true;
    }
    return false;
}
public int getMaxCapacity() { return maxCapacity; }
public int getUsedCapacity() { return items.size(); }

@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append(String.format("Inventory: %d/%d slots | Gold: %d\n",
                           getUsedCapacity(), maxCapacity, gold));
    sb.append("Items:\n");
    for (GameItem item : items) {
        sb.append("  - ").append(item).append("\n");
    }
    return sb.toString();
}
}

// Game Level class
class GameLevel implements Serializable {
    private static final long serialVersionUID = 1L;

    private int levelNumber;
    private String levelName;
    private boolean completed;
    private int score;
    private int maxScore;
    private LocalDateTime completionTime;
    private int deaths;
    private int secretsFound;
}

```

```

private int totalSecrets;

public GameLevel(int levelNumber, String levelName, int maxScore, int totalSecrets) {
    this.levelNumber = levelNumber;
    this.levelName = levelName;
    this.maxScore = maxScore;
    this.totalSecrets = totalSecrets;
    this.completed = false;
    this.score = 0;
    this.deaths = 0;
    this.secretsFound = 0;
}

public void completeLevel(int score, int secretsFound) {
    this.completed = true;
    this.score = Math.min(score, maxScore);
    this.secretsFound = Math.min(secretsFound, totalSecrets);
    this.completionTime = LocalDateTime.now();
}

public void incrementDeaths() {
    this.deaths++;
}

public double getCompletionPercentage() {
    return (score / (double) maxScore) * 100;
}

// Getters
public int getLevelNumber() { return levelNumber; }
public String getLevelName() { return levelName; }
public boolean isCompleted() { return completed; }
public int getScore() { return score; }
public int getMaxScore() { return maxScore; }
public int getDeaths() { return deaths; }
public int getSecretsFound() { return secretsFound; }
public int getTotalSecrets() { return totalSecrets; }
public LocalDateTime getCompletionTime() { return completionTime; }

@Override
public String toString() {
    String status = completed ? "COMPLETED" : "IN PROGRESS";
    String timeStr = completionTime != null ?
        completionTime.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm")) : "N/A";
    return String.format("Level %d: %s [%s]\n Score: %d/%d (%.1f%%) | Secrets: %d/%d | "
        + "Deaths: %d\n Completed: %s",
        levelNumber, levelName, status, score, maxScore,
        getCompletionPercentage(),
        secretsFound, totalSecrets, deaths, timeStr);
}

// Player Stats class
class PlayerStats implements Serializable {
    private static final long serialVersionUID = 1L;

    private int level;
    private int experience;
    private int health;
    private int maxHealth;
    private int mana;
    private int maxMana;
}

```

```
private int strength;
private int dexterity;
private int intelligence;
private LocalDateTime playTimeStart;
private long totalPlayTimeSeconds;

public PlayerStats() {
    this.level = 1;
    this.experience = 0;
    this.health = 100;
    this.maxHealth = 100;
    this.mana = 50;
    this.maxMana = 50;
    this.strength = 10;
    this.dexterity = 10;
    this.intelligence = 10;
    this.totalPlayTimeSeconds = 0;
    this.playTimeStart = LocalDateTime.now();
}

public void addExperience(int exp) {
    this.experience += exp;
    // Simple level up logic
    int expNeeded = level * 1000;
    while (experience >= expNeeded) {
        levelUp();
        expNeeded = level * 1000;
    }
}

private void levelUp() {
    level++;
    maxHealth += 20;
    health = maxHealth; // Full heal on level up
    maxMana += 10;
    mana = maxMana;
    strength += 2;
    dexterity += 2;
    intelligence += 2;
}

public void startPlaySession() {
    playTimeStart = LocalDateTime.now();
}

public void endPlaySession() {
    if (playTimeStart != null) {
        totalPlayTimeSeconds += java.time.Duration.between(playTimeStart,
LocalDateTime.now()).getSeconds();
        playTimeStart = null;
    }
}

public String getFormattedPlayTime() {
    long totalSeconds = totalPlayTimeSeconds;
    if (playTimeStart != null) {
        totalSeconds += java.time.Duration.between(playTimeStart,
LocalDateTime.now()).getSeconds();
    }

    long hours = totalSeconds / 3600;
    long minutes = (totalSeconds % 3600) / 60;
```

```

        long seconds = totalSeconds % 60;

        return String.format("%02d:%02d:%02d", hours, minutes, seconds);
    }

    // Getters and setters
    public int getLevel() { return level; }
    public int getExperience() { return experience; }
    public int getHealth() { return health; }
    public void setHealth(int health) { this.health = Math.min(health, maxHealth); }
    public int getMaxHealth() { return maxHealth; }
    public int getMana() { return mana; }
    public void setMana(int mana) { this.mana = Math.min(mana, maxMana); }
    public int getMaxMana() { return maxMana; }
    public int getStrength() { return strength; }
    public int getDexterity() { return dexterity; }
    public int getIntelligence() { return intelligence; }

    @Override
    public String toString() {
        return String.format(
            "Level: %d | EXP: %d/%d\n" +
            "Health: %d/%d | Mana: %d/%d\n" +
            "STR: %d | DEX: %d | INT: %d\n" +
            "Play Time: %s",
            level, experience, level * 1000, health, maxHealth, mana, maxMana,
            strength, dexterity, intelligence, getFormattedPlayTime()
        );
    }
}

// Main Game State class
class GameState implements Serializable {
    private static final long serialVersionUID = 1L;
    private static final String SAVE_FILE_PREFIX = "game_save_";
    private static final String SAVE_EXTENSION = ".sav";

    // Game state data
    private String playerName;
    private String saveSlotName;
    private LocalDateTime saveDate;
    private int gameVersion;

    // Game components
    private PlayerStats playerStats;
    private PlayerInventory inventory;
    private List<GameLevel> levels;
    private List<Achievement> achievements;
    private int currentLevelIndex;

    public GameState(String playerName, String saveSlotName) {
        this.playerName = playerName;
        this.saveSlotName = saveSlotName;
        this.saveDate = LocalDateTime.now();
        this.gameVersion = 1;

        // Initialize game components
        this.playerStats = new PlayerStats();
        this.inventory = new PlayerInventory(50); // 50 slot inventory
        this.levels = new ArrayList<>();
        this.achievements = new ArrayList<>();
        this.currentLevelIndex = 0;
    }
}

```

```

        initializeGameData();
    }

private void initializeGameData() {
    // Initialize levels
    levels.add(new GameLevel(1, "The Beginning", 1000, 3));
    levels.add(new GameLevel(2, "Forest of Whispers", 1500, 5));
    levels.add(new GameLevel(3, "Crystal Caverns", 2000, 7));
    levels.add(new GameLevel(4, "Dragon's Peak", 3000, 10));

    // Initialize achievements
    achievements.add(new Achievement("ACH_001", "First Steps", "Complete level 1",
AchievementType.BRONZE, 10));
    achievements.add(new Achievement("ACH_002", "Explorer", "Find 10 secrets",
AchievementType.SILVER, 25));
    achievements.add(new Achievement("ACH_003", "Dragon Slayer", "Defeat the dragon",
AchievementType.GOLD, 50));
    achievements.add(new Achievement("ACH_004", "Completionist", "Get 100% on all levels",
AchievementType.PLATINUM, 100));
    achievements.add(new Achievement("ACH_005", "Secret Finder", "Find the hidden treasure",
AchievementType.SECRET, 75));

    // Add starting items
    inventory.addItem(new GameItem("ITEM_001", "Health Potion", "Restores 50 HP",
ItemType.CONSUMABLE, 10, 25));
    inventory.addItem(new GameItem("ITEM_002", "Wooden Sword", "A basic sword",
ItemType.WEAPON, 1, 50));
    inventory.addItem(new GameItem("ITEM_003", "Leather Armor", "Basic protection",
ItemType.ARMOR, 1, 75));
    inventory.addGold(100);
}

// Save game method
public void saveGame() throws GameSaveException {
    this.saveDate = LocalDateTime.now();
    String filename = SAVE_FILE_PREFIX + saveSlotName.replace(" ", "_") + SAVE_EXTENSION;

    try (FileOutputStream fileOut = new FileOutputStream(filename);
        ObjectOutputStream objectOut = new ObjectOutputStream(fileOut)) {

        objectOut.writeObject(this);
        System.out.println("Game saved successfully: " + filename);

    } catch (IOException e) {
        throw new GameSaveException("Failed to save game: " + e.getMessage(), e);
    }
}

// Static method to load game
public static GameState loadGame(String saveSlotName) throws GameSaveException {
    String filename = SAVE_FILE_PREFIX + saveSlotName.replace(" ", "_") + SAVE_EXTENSION;

    try (FileInputStream fileIn = new FileInputStream(filename);
        ObjectInputStream objectIn = new ObjectInputStream(fileIn)) {

        GameState loadedState = (GameState) objectIn.readObject();
        System.out.println("Game loaded successfully: " + filename);
        return loadedState;

    } catch (FileNotFoundException e) {
        throw new GameSaveException("Save file not found: " + filename, e);
    }
}

```

```

        } catch (IOException | ClassNotFoundException e) {
            throw new GameSaveException("Failed to load game: " + e.getMessage(), e);
        }
    }

    // Get list of available save files
    public static List<String> getAvailableSaves() {
        List<String> saves = new ArrayList<>();
        File dir = new File(".");
        File[] files = dir.listFiles((d, name) -> name.startsWith(SAVE_FILE_PREFIX) &&
name.endsWith(SAVE_EXTENSION));

        if (files != null) {
            for (File file : files) {
                String name = file.getName();
                name = name.substring(SAVE_FILE_PREFIX.length(), name.length() -
SAVE_EXTENSION.length());
                name = name.replace("_", " ");
                saves.add(name);
            }
        }

        return saves;
    }

    // Game progression methods
    public void completeCurrentLevel(int score, int secretsFound) {
        if (currentLevelIndex < levels.size()) {
            GameLevel currentLevel = levels.get(currentLevelIndex);
            currentLevel.completeLevel(score, secretsFound);

            // Check for achievements
            checkAchievements();

            // Move to next level if available
            if (currentLevelIndex < levels.size() - 1) {
                currentLevelIndex++;
            }
        }
    }

    private void checkAchievements() {
        // Check level 1 completion
        if (levels.get(0).isCompleted()) {
            getAchievement("ACH_001").unlock();
        }

        // Check total secrets found
        int totalSecrets = levels.stream().mapToInt(GameLevel::getSecretsFound).sum();
        if (totalSecrets >= 10) {
            getAchievement("ACH_002").unlock();
        }

        // Check if all levels completed with 100%
        boolean allPerfect = levels.stream().allMatch(level -> level.getCompletionPercentage() ==
100);
        if (allPerfect) {
            getAchievement("ACH_004").unlock();
        }
    }

    private Achievement getAchievement(String id) {

```

```

        return achievements.stream()
            .filter(a -> a.getId().equals(id))
            .findFirst()
            .orElse(null);
    }

// Getters and setters
public String getPlayerName() { return playerName; }
public String getSaveSlotName() { return saveSlotName; }
public LocalDateTime getSaveDate() { return saveDate; }
public PlayerStats getPlayerStats() { return playerStats; }
public PlayerInventory getInventory() { return inventory; }
public List<GameLevel> getLevels() { return new ArrayList<>(levels); }
public List<Achievement> getAchievements() { return new ArrayList<>(achievements); }
public GameLevel getCurrentLevel() {
    return (currentLevelIndex < levels.size()) ? levels.get(currentLevelIndex) : null;
}
public int getCurrentLevelIndex() { return currentLevelIndex; }

public List<Achievement> getUnlockedAchievements() {
    List<Achievement> unlocked = new ArrayList<>();
    for (Achievement achievement : achievements) {
        if (achievement.isUnlocked()) {
            unlocked.add(achievement);
        }
    }
    return unlocked;
}

public int getTotalAchievementPoints() {
    return achievements.stream()
        .filter(Achievement::isUnlocked)
        .mapToInt(Achievement::getPoints)
        .sum();
}

@Override
public String toString() {
    return String.format(
        "Game State: %s (%s)\n" +
        "Saved: %s | Version: %d\n" +
        "Levels: %d/%d completed\n" +
        "Achievements: %d/%d (%d points)",
        playerName, saveSlotName,
        saveDate.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm")),
        gameVersion,
        levels.stream().filter(GameLevel::isCompleted).count(), levels.size(),
        getUnlockedAchievements().size(), achievements.size(), getTotalAchievementPoints()
    );
}

// Game Manager class
class GameManager {
    private GameState currentGameState;
    private Scanner scanner;

    public GameManager() {
        this.scanner = new Scanner(System.in);
    }

    public void start() {

```

```

System.out.println("Welcome to the Game Progress Management System! \n");

boolean running = true;
while (running) {
    displayMainMenu();
    int choice = getIntInput("Choose an option: ");

    switch (choice) {
        case 1:
            createNewGame();
            break;
        case 2:
            loadExistingGame();
            break;
        case 3:
            listSaveFiles();
            break;
        case 4:
            running = false;
            System.out.println("Thanks for playing! ");
            break;
        default:
            System.out.println("Invalid option! Please try again.");
    }

    if (currentGameState != null) {
        gameLoop();
    }
}

scanner.close();
}

private void displayMainMenu() {
    System.out.println("\n== MAIN MENU ==");
    System.out.println("1. New Game");
    System.out.println("2. Load Game");
    System.out.println("3. List Save Files");
    System.out.println("4. Exit");
}

private void createNewGame() {
    System.out.print("Enter player name: ");
    String playerName = scanner.nextLine();

    System.out.print("Enter save slot name: ");
    String saveSlotName = scanner.nextLine();

    currentGameState = new GameState(playerName, saveSlotName);
    System.out.println("... New game created for: " + playerName);
}

private void loadExistingGame() {
    List<String> saves = GameState.getAvailableSaves();
    if (saves.isEmpty()) {
        System.out.println("No save files found!");
        return;
    }

    System.out.println("\nAvailable Save Files:");
    for (int i = 0; i < saves.size(); i++) {
        System.out.println((i + 1) + ". " + saves.get(i));
    }
}

```

```

}

int choice = getIntInput("Select save file: ") - 1;
if (choice >= 0 && choice < saves.size()) {
    try {
        currentGameState = GameState.loadGame(saves.get(choice));
        System.out.println("... Game loaded: " + currentGameState.getPlayerName());
    } catch (GameSaveException e) {
        System.out.println("Error loading game: " + e.getMessage());
    }
} else {
    System.out.println("Invalid selection!");
}
}

private void listSaveFiles() {
    List<String> saves = GameState.getAvailableSaves();
    if (saves.isEmpty()) {
        System.out.println("No save files found.");
    } else {
        System.out.println("\nSave Files:");
        for (String save : saves) {
            System.out.println(" " + save);
        }
    }
}

private void gameLoop() {
    boolean inGame = true;

    while (inGame && currentGameState != null) {
        displayGameMenu();
        int choice = getIntInput("Choose an option: ");

        switch (choice) {
            case 1:
                playGame();
                break;
            case 2:
                viewPlayerStats();
                break;
            case 3:
                viewInventory();
                break;
            case 4:
                viewLevels();
                break;
            case 5:
                viewAchievements();
                break;
            case 6:
                saveGame();
                break;
            case 7:
                inGame = false;
                System.out.println("Returning to main menu...");
                break;
            default:
                System.out.println("Invalid option!");
        }
    }
}

```

```

// End play session when leaving game
if (currentGameState != null) {
    currentGameState.getPlayerStats().endPlaySession();
}
}

private void displayGameMenu() {
    System.out.println("\n==== GAME MENU ====");
    System.out.println("1. Play Game");
    System.out.println("2. View Player Stats");
    System.out.println("3. View Inventory");
    System.out.println("4. View Levels");
    System.out.println("5. View Achievements");
    System.out.println("6. Save Game");
    System.out.println("7. Return to Main Menu");
}

private void playGame() {
    currentGameState.getPlayerStats().startPlaySession();

    GameLevel currentLevel = currentGameState.getCurrentLevel();
    if (currentLevel == null) {
        System.out.println("🎉 Congratulations! You've completed all levels!");
        return;
    }

    System.out.println("\n==== PLAYING: " + currentLevel.getLevelName() + " ===");

    // Simulate gameplay
    Random random = new Random();
    int score = random.nextInt(currentLevel.getMaxScore() - 500) + 500;
    int secretsFound = random.nextInt(currentLevel.getTotalSecrets() + 1);
    int deaths = random.nextInt(5);

    // Apply gameplay effects
    for (int i = 0; i < deaths; i++) {
        currentLevel.incrementDeaths();
    }

    // Gain experience
    int expGained = score / 10;
    currentGameState.getPlayerStats().addExperience(expGained);

    // Find some items
    if (random.nextBoolean()) {
        GameItem newItem = new GameItem("ITEM_" + (100 + random.nextInt(900)),
                                         "Magic Potion", "Mysterious magical potion",
                                         ItemType.CONSUMABLE, 5, 100);
        currentGameState.getInventory().addItem(newItem);
        System.out.println("-found item: " + newItem.getName());
    }

    // Gain gold
    int goldFound = random.nextInt(100) + 50;
    currentGameState.getInventory().addGold(goldFound);

    // Complete level
    currentGameState.completeCurrentLevel(score, secretsFound);

    System.out.println("Level completed!");
    System.out.println("Score: " + score + "/" + currentLevel.getMaxScore());
    System.out.println("Secrets found: " + secretsFound + "/" +

```

```

currentLevel.getTotalSecrets());
    System.out.println("Deaths: " + deaths);
    System.out.println("Experience gained: " + expGained);
    System.out.println("Gold found: " + goldFound);

    currentGameState.getPlayerStats().endPlaySession();
}

private void viewPlayerStats() {
    System.out.println("\n==== PLAYER STATS ====");
    System.out.println(currentGameState.getPlayerStats());
}

private void viewInventory() {
    System.out.println("\n==== INVENTORY ====");
    System.out.println(currentGameState.getInventory());
}

private void viewLevels() {
    System.out.println("\n==== LEVEL PROGRESS ====");
    for (GameLevel level : currentGameState.getLevels()) {
        System.out.println(level);
        System.out.println();
    }
}

private void viewAchievements() {
    System.out.println("\n==== ACHIEVEMENTS ====");
    System.out.println("Total Points: " + currentGameState.getTotalAchievementPoints());
    System.out.println("Unlocked: " + currentGameState.getUnlockedAchievements().size() +
                       "/" + currentGameState.getAchievements().size());

    System.out.println("\nLocked Achievements:");
    for (Achievement achievement : currentGameState.getUnlockedAchievements()) {
        System.out.println("• " + achievement);
    }

    System.out.println("\nLocked Achievements:");
    for (Achievement achievement : currentGameState.getAchievements()) {
        if (!achievement.isUnlocked()) {
            System.out.println("• " + achievement.getName() + " - " +
achievement.getDescription());
        }
    }
}

private void saveGame() {
    try {
        currentGameState.saveGame();
    } catch (GameSaveException e) {
        System.out.println("Error saving game: " + e.getMessage());
    }
}

private int getIntInput(String prompt) {
    while (true) {
        System.out.print(prompt);
        try {
            int input = scanner.nextInt();
            scanner.nextLine(); // Consume newline
            return input;
        } catch (InputMismatchException e) {

```

```
        System.out.println("Please enter a valid number!");
        scanner.nextLine(); // Clear invalid input
    }
}
}

// Main class
public class GameProgressSystem {
    public static void main(String[] args) {
        GameManager gameManager = new GameManager();
        gameManager.start();
    }
}
```