MiniProjectAssessedLevel8

December 14, 2022

1 Miniproject Assessed Exercise

2 Level 8

2.1 Chou Chia-Yi

2.2 Student ID: 220271772

2.3 12 December 2022

2.4 Version 8

2.5 Summary of the Question

The program provides an option to save the state of the game into a file so the game can be quit and restarted at a later point.

2.6 Justification that the program passes this level

This is based on tick boxes at start of the question description for miniproject program level 7 - All the constructs, style and features above AND - BOTH file input AND file output - Excellent style over comments, indentation, variable usage, final variables, etc. - Program is fully composed into methods so excellent use of methods throughout - All variables are defined in methods and have minimum scope

2.7 The literate program development

2.7.1 Class

What it does Create a new type

Implementation (how it works) This creates a new type that can then be used in the program to declare different player.

[174]:
```
class CastleGamePlayer
{
    String playerName;
    int score;
    boolean leave;
}
```

2.7.2 Testing

[ ]:

2.7.3 Class

What it does Create a new type

Implementation (how it works) This creates a new type that can then be used in the program like existing types to declare variables.

[175]:
```
class CastleRoom
{
    String roomName;
    boolean hasTrap;
    boolean hasTreasure;
    int treasurePoints;
    int disableTrapPoints;
}
```

Testing

[ ]:

2.7.4 Check for bounds

What it does Check the if value between bounds

Implementation (how it works) Use try and catch to check if the value between bounds

[176]:
```
public static boolean checkForBounds(String input, int min, int max)
{
    int x;
    try
    {
        x = Integer.parseInt(input);
    }
    catch (Exception err)
    {
        return false;
    }
    if( x < min || x > max)
    {
        return false;
    }
    return true;
}
```

Testing

[ ]:

2.7.5 Return string

What it does Print the message

Implementation (how it works) Print the question and return the value as string

[177]:
```
public static String askQuestionStr(String question)
{
    String value;
    Scanner scanner = new Scanner(System.in);

    System.out.println(question);
    value = scanner.nextLine();

    return value;
}
```

Testing

[178]:
```
askQuestionStr("");
```

```
 d
```

[178]:
```
d
```

2.7.6 Return integer

What it does Print the message

Implementation (how it works) Print the message and return the value as integer

[179]:
```
public static int askQuestionInt(String question)
{
    int value;
    Scanner scanner = new Scanner(System.in);

    System.out.println(question);
    value = Integer.parseInt(scanner.nextLine());

    return value;
}
```

Testing

[180]:
```
askQuestionInt("");
```

```
 1
```

[180]: 1

2.7.7 Get player's name

What it does Stored the player's name record

Implementation (how it works) Given an player's name and return the name stored in the record

[181]:
```
public static String getCastleGamePlayerName(CastleGamePlayer player)
{
    return player.playerName;
}
```

Testing

[182]:
```
CastleGamePlayer player = new CastleGamePlayer();
player.playerName = "Lily";
getCastleGamePlayerName(player);
```

[182]: Lily

2.7.8 Get player's score

What it does Stored the player's score record

Implementation (how it works) Given a player's score and return the name stored in the record

[183]:
```
public static int getCastleGameScore(CastleGamePlayer player)
{
    return player.score;
}
```

Testing

[184]:
```
CastleGamePlayer player = new CastleGamePlayer();
player.score = 5;
getCastleGameScore(player);
```

[184]: 5

2.7.9 Get player's whether leave the game

What it does Stored the player's leaving record

Implementation (how it works) Given whether the player leave or not and return the value stored in the record

[185]:
```
public static boolean getCastleGameLeaving(CastleGamePlayer player)
{
    return player.leave;
}
```

Testing

[186]:
```
CastleGamePlayer player = new CastleGamePlayer();
player.leave = true;
getCastleGameLeaving(player);
```

[186]: true

2.7.10 Get room names

What it does Stored the room names record

Implementation (how it works) Given room names record and return the room names stored in the record

[187]:
```
public static String getCastleGameRoomName(CastleRoom room)
{
    return room.roomName;
}
```

Testing

[188]:
```
CastleRoom room = new CastleRoom();
room.roomName = "study";
getCastleGameRoomName(room);
```

[188]: study

2.7.11 Get room's trap record

What it does Stored the traps record

Implementation (how it works) Given the room has trap or not and return the boolean stored in the record

[189]:
```
public static boolean getCastleGameHasTrap(CastleRoom room)
{
```

```
    return room.hasTrap;
}
```

Testing

[190]:
```
CastleRoom room = new CastleRoom();
room.hasTrap = true;
getCastleGameHasTrap(room);
```

[190]:
```
true
```

2.7.12 Get the treasure

What it does Stored the treasures record

Implementation (how it works) Given whether the room has treasures and return the treasure stored in the record

[191]:
```
public static boolean getCastleGameHasTreasure(CastleRoom room)
{
    return room.hasTreasure;
}
```

Testing

[192]:
```
CastleRoom room = new CastleRoom();
room.hasTreasure = true;
getCastleGameHasTreasure(room);
```

[192]:
```
true
```

2.7.13 Get treasure points

What it does Stored the treasure points record

Implementation (how it works) Given a treasure points and return the treasure points stored in the record

[193]:
```
public static int getCastleGameTreasurePoints(CastleRoom room)
{
    return room.treasurePoints;
}
```

2.7.14 Testing

[194]:
```
CastleRoom room = new CastleRoom();
room.treasurePoints = 3;
getCastleGameTreasurePoints(room);
```

[194]: 3

2.7.15 Get disable trap points

What it does Stored the disable trap points record

Implementation (how it works) Given a disable trap points and return the points stored in the record

[195]:
```
public static int getCastleGameDisableTrapPoints(CastleRoom room)
{
    return room.disableTrapPoints;
}
```

Testing

[196]:
```
CastleRoom room = new CastleRoom();
room.disableTrapPoints = 3;
getCastleGameDisableTrapPoints(room);
```

[196]: 3

2.7.16 Set player's name

What it does Stored the player's name record

Implementation (how it works) Given an player's name, return the name and stored in the record then update the data in record

[197]:
```
public static void setCastleGamePlayerName(CastleGamePlayer player, String␣
 ↪playerName)
{
    player.playerName = playerName;
    return;
}
```

Testing

[198]:
```
CastleGamePlayer player = new CastleGamePlayer();
setCastleGamePlayerName(player, "Lily");
System.out.println(player.playerName);
```

```
Lily
```

2.7.17 Set player's score

What it does Stored the player's score record

Implementation (how it works) Given a player's score, return the score and stored in the record then update the data in record

[199]:
```
public static void setCastleGameScore(CastleGamePlayer player, int score)
{
    player.score = score;
    return;
}
```

Testing

[200]:
```
CastleGamePlayer player = new CastleGamePlayer();
setCastleGameScore(player, 3);
System.out.println(player.score);
```

```
3
```

2.7.18 Set player's leaving record

What it does Stored the player's leaving record

Implementation (how it works) Given a player's leaving record, return the leaving and stored in the record then update the data in record

[201]:
```
public static void setCastleGameLeave(CastleGamePlayer player, boolean leave)
{
    player.leave = leave;
    return;
}
```

Testing

[202]:
```
CastleGamePlayer player = new CastleGamePlayer();
setCastleGameLeave(player, true);
System.out.println(player.leave);
```

```
true
```

2.7.19 Set room name

What it does Stored the room's name record

Implementation (how it works) Given a room's name, return the name and stored in the record then update the data in record

[203]:
```
public static void setCastleGameRoomName(CastleRoom room, String roomName)
{
    room.roomName = roomName;
    return;
}
```

Testing

[204]:
```
CastleRoom room = new CastleRoom();
setCastleGameRoomName(room, "Study");
System.out.println(room.roomName);
```

```
Study
```

2.7.20 Set room's traps

What it does Stored the room's traps record

Implementation (how it works) Given a room's traps, return the traps and stored in the record then update the data in record

[205]:
```
public static void setCastleGameHasTrap(CastleRoom room, boolean hasTrap)
{
    room.hasTrap = hasTrap;
    return;
}
```

Testing

[206]:
```
CastleRoom room = new CastleRoom();
setCastleGameHasTrap(room, false);
System.out.println(room.hasTrap);
```

```
false
```

2.7.21 Set room's treasure

What it does Stored the treasure record

Implementation (how it works) Given the room has treasures or not, return the treasures and stored in the record then update the data in record

[207]:
```
public static void setCastleGameHasTreasure(CastleRoom room, boolean␣
↪hasTreasure)
{
    room.hasTreasure = hasTreasure;
    return;
}
```

Testing

[208]:
```
CastleRoom room = new CastleRoom();
setCastleGameHasTreasure(room, false);
System.out.println(room.hasTreasure);
```

```
false
```

2.7.22 Set treasure points

What it does Stored the treasure points record

Implementation (how it works) Given a treasure points, return the points and stored in the record then update the data in record

[209]:
```
public static void setCastleGameTreasurePoints(CastleRoom room, int␣
↪treasurePoints)
{
    room.treasurePoints = treasurePoints;
    return;
}
```

Testing

[210]:
```
CastleRoom room = new CastleRoom();
setCastleGameTreasurePoints(room, 3);
System.out.println(room.treasurePoints);
```

```
3
```

2.7.23 Set disable trap points

What it does Stored the disable trap points record

Implementation (how it works) Given a disable trap points, return the points and stored in the record then update the data in record

[211]:
```
public static void setCastleGameDisableTrapPoints(CastleRoom room, int␣
↪disableTrapPoints)
{
    room.disableTrapPoints = disableTrapPoints;
    return;
}
```

Testing

[212]:
```
CastleRoom room = new CastleRoom();
setCastleGameDisableTrapPoints(room, 3);
System.out.println(room.disableTrapPoints);
```

3

2.7.24 Ask room

What it does Ask player which room to go in.

Implementation (how it works) Give player a chance to choose a room to go in.

[249]:
```
public static String askRoom(String name, int len)
{
    String choice;

    System.out.println(name + ", it is your turn. You are in a giant hall.");
    choice = askQuestionStr("There are " + len + " rooms in front of you. Which␣
 ↪room do you want to go? (type 1-" + len + ", and q to quit, s to save): ");
    if (choice.equals("q") || choice.equals("s"))
    {
        return choice;
    }
    while (!checkForBounds(choice,1,len))
    {
        System.out.println("invalid input");
        choice = askQuestionStr("There are " + len + " rooms in front of you.␣
 ↪Which room do you want to go? (type 1-" + len + ", and q to quit, s to save):
 ↪ ");
        if (choice.equals('q')) return choice;
    }
    return choice;
}
```

Testing

[250]:
```
askRoom("Lily",6);
```

```
Lily, it is your turn. You are in a giant hall.
There are 6 rooms in front of you. Which room do you want to go? (type 1-6, and
q to quit, s to save):
```

```
 s
```

[250]:
```
s
```

2.7.25 Print message

What it does Prints the message saying "There is a gold ring on a table in the middle of the room.

Implementation (how it works) It just print a simple statement.

[215]:
```
public static void printSurroundings(CastleRoom room)
{
    System.out.println("You are in " + getCastleGameRoomName(room) + ".");
    return;
}
```

Testing

[216]:
```
CastleRoom room = new CastleRoom();
room.roomName = "study";
printSurroundings(room);
```

```
You are in study.
```

2.7.26 Get Point

What it does Give the player the points.

Implementation (how it works) Use a random number to create points.

[217]:
```
public static int getPoint(int max)
{
    Random bigdice = new Random();
    int diceroll = bigdice.nextInt(max) + 1;

    return diceroll;
}
```

Testing

[218]:
```
getPoint(3);
```

[218]: 3

2.7.27 Find Cup

What it does This method tells player how many score they get.

Implementation (how it works) Use the data above and print the statement.

[219]:
```
public static int findCup(CastleRoom room, int score)
{
    if (getCastleGameHasTrap(room) == false)
    {
        if (getCastleGameHasTreasure(room) == true)
        {
            int point = getCastleGameTreasurePoints(room);
```

```
            System.out.println("You find a cup and get " + point + " point(s).
 ↪");
            score = score + point;
            setCastleGameHasTreasure(room, false);
        }
        else
        {
            System.out.println("You didn't find anything.");
        }
    }
    else
    {
        System.out.println("It's a trap!");
        score = 0;
    }
    return score;
}
```

Testing

[220]:
```
CastleRoom room = new CastleRoom();
setCastleGameRoomName(room, "Study room");
setCastleGameHasTrap(room, true);
setCastleGameHasTreasure(room, true);
setCastleGameTreasurePoints(room, 3);
setCastleGameDisableTrapPoints(room, 1);
findCup(room,1);
```

```
It's a trap!
```

[220]:
```
0
```

2.7.28 Disable traps

What it does To see whether the room have traps

Implementation (how it works) Use boolean to see if player disarm the trap, if yes, they will lose points

[221]:
```
public static int disableTrap(CastleRoom room, int score)
{
    int point = getCastleGameDisableTrapPoints(room);
    System.out.println("You disarm the trap and it costs you " + point + "␣
 ↪point(s).");
    score = score - point;
    setCastleGameHasTrap(room, false);
    return score;
```

```
}
```

Testing

[222]:
```
CastleRoom room = new CastleRoom();
setCastleGameRoomName(room, "A");
setCastleGameHasTrap(room, true);
setCastleGameHasTreasure(room, false);
setCastleGameTreasurePoints(room, 0);
setCastleGameDisableTrapPoints(room, 1);
disableTrap(room,1);
```

```
You disarm the trap and it costs you 1 point(s).
```

[222]:
```
0
```

2.7.29 Print the score

What it does This method tells player how many score they get.

Implementation (how it works) Use the data above and print the statement.

[223]:
```
public static void printScore(int score)
{
    System.out.println("Your score is " + score + ".");
}
```

Testing

[224]:
```
printScore(3);
```

```
Your score is 3.
```

2.7.30 Ask Choice

What it does Ask whether to search the room, taking any object found or leave.

Implementation (how it works) Let player to make a choice, the action is passed the person's command as argument and returns the message to print.

[225]:
```
public static String askChoice()
{
    String choice;

    System.out.println("Do you what searching the room, taking any object found␣
 ↪or leaving the room?");
```

```
    choice = askQuestionStr("(type 1 for searching the room , type 2 for taking␣
 ↪any object found, type 3 for disarming a trap and type 4 for leaving the␣
 ↪room)");
    while (!checkForBounds(choice,1,4))
    {
        System.out.println("invalid input");
        choice = askQuestionStr("(type 1 for searching the room , type 2 for␣
 ↪taking any object found, type 3 for disarming a trap and type 4 for leaving␣
 ↪the room)");
    }
    return choice;

} // END askChoice
```

Testing

[226]:
```
askChoice();
```

```
Do you what searching the room, taking any object found or leaving the room?
(type 1 for searching the room , type 2 for taking any object found, type 3 for
disarming a trap and type 4 for leaving the room)

 4
```

[226]: 4

2.7.31 Create a new game

What it does Ask the player whether to start a new game

Implementation (how it works) Ask the player whether to start a new game

[227]:
```
public static String askNewGame()
{
    String choice;

    System.out.println("Do you want to start a new game or load a saved game?");
    choice = askQuestionStr("(type 1 for start a new game, or type 2 for load a␣
 ↪saved game): ");
    while (!checkForBounds(choice,1,2))
    {
        System.out.println("invalid input");
        choice = askQuestionStr("(type 1 for start a new game, or type 2 for␣
 ↪load a saved game): ");
    }

    return choice;
}
```

Testing

[228]:
```
askNewGame();
```

```
Do you want to start a new game or load a saved game?
(type 1 for start a new game, or type 2 for load a saved game):
 2
```

[228]: 2

2.7.32 Create a record

What it does Store the records of all rooms

Implementation (how it works) Store the records of all rooms

[229]:
```
public static CastleRoom createRoomRecord(String roomName, boolean hasTrap,␣
 ↪boolean hasTreasure, int treasurePoints, int disableTrapPoints)
{
    CastleRoom room = new CastleRoom();
    setCastleGameRoomName(room, roomName);
    setCastleGameHasTrap(room, hasTrap);
    setCastleGameHasTreasure(room, hasTreasure);
    setCastleGameTreasurePoints(room, treasurePoints);
    setCastleGameDisableTrapPoints(room, disableTrapPoints);
    return room;
}
```

2.7.33 Testing

[230]:
```
createRoomRecord("study",true,true,1,2);
```

[230]: REPL.$JShell$13E$CastleRoom@112db52e

2.7.34 Create a record

What it does Create a record for new players

Implementation (how it works) Create a record for new players

[231]:
```
public static CastleGamePlayer createPlayerRecord(String playerName, int score,␣
 ↪boolean leave)
{
    CastleGamePlayer player = new CastleGamePlayer();
    setCastleGamePlayerName(player, playerName);
    setCastleGameScore(player, score);
    setCastleGameLeave(player, leave);
```

```
    return player;
}
```

Testing

[232]:
```
createPlayerRecord("Lily",3,true);
```

[232]:
```
REPL.$JShell$12E$CastleGamePlayer@1d7093d4
```

2.7.35 Ask player's name

What it does Ask the player for their names.

Implementation (how it works) This method ask player's name and store their name.

[233]:
```
public static void askPlayers(CastleGamePlayer[] players, int len)
{
    String name;
    for (int i = 0; i < len; i++)
    {
        name = askQuestionStr("Player " + (i + 1) + ". What is your name? ");
        players[i] = createPlayerRecord(name, 0, false);
    }

    return;
}
```

Testing

[234]:
```
CastleGamePlayer [] players = new CastleGamePlayer [2];
askPlayers(players,2);
```

```
Player 1. What is your name?
 Lily
Player 2. What is your name?
 A
```

2.7.36 Create a record

What it does Store the records of all players

Implementation (how it works) Store the records of all players

[235]:
```
public static void createRooms(String [] roomNames, CastleRoom [] rooms, int␣
 ↪len)
{
    for (int i = 0; i < len; i++)
```

```
    {
        boolean hasTrap = false;
        boolean hasTreasure = false;
        int treasurePoints = getPoint(6);
        int disableTrapPoints = 1;
        if (getPoint(2) > 1) hasTrap = true;
        if (getPoint(2) > 1) hasTreasure = true;
        rooms[i] = createRoomRecord(roomNames[i], hasTrap, hasTreasure,␣
 ↪treasurePoints, disableTrapPoints);
    }
    return;
}
```

Testing

[236]:
```
String[] roomNames  = new String[] { "A", "B", "C" };
CastleRoom[] rooms = new CastleRoom[3];
createRooms(roomNames, rooms, 3);
```

2.7.37 Save the game

What it does To save the game

Implementation (how it works) Ask the player whether to save the game

[237]:
```
public static void saveGame(CastleGamePlayer[] players, int playerCount,␣
 ↪CastleRoom[] rooms, int roomCount, int currentPlayer) throws IOException
{
    PrintWriter outputStream = new PrintWriter(new FileWriter("game.txt"));
    int i;
    outputStream.println(playerCount);
    outputStream.println(roomCount);
    for (i = 0; i < playerCount; i++)
    {
        outputStream.println(getCastleGamePlayerName(players[i]));
        outputStream.println(getCastleGameScore(players[i]));
        if (getCastleGameLeaving(players[i]) == true)
        {
            outputStream.println("1");
        }
        else
        {
            outputStream.println("0");
        }
    }
    for (i = 0; i < roomCount; i++)
    {
```

```
        outputStream.println(getCastleGameRoomName(rooms[i]));
        outputStream.println(getCastleGameTreasurePoints(rooms[i]));
        outputStream.println(getCastleGameDisableTrapPoints(rooms[i]));
        if (getCastleGameHasTrap(rooms[i]) == true)
        {
            outputStream.println("1");
        }
        else
        {
            outputStream.println("0");
        }
        if (getCastleGameHasTreasure(rooms[i]) == true)
        {
            outputStream.println("1");
        }
        else
        {
            outputStream.println("0");
        }
    }
    outputStream.println(currentPlayer);
    outputStream.close();
}
```

Testing

[238]:
```
CastleGamePlayer[] players = new CastleGamePlayer[1];
players[0] = createPlayerRecord("Lily", 0, false);
String[] roomNames  = new String[] { "A", "B", "C" };
CastleRoom[] rooms = new CastleRoom[3];
createRooms(roomNames, rooms, 3);
saveGame(players, 1, rooms, 3, 0);
```

2.7.38 Load the game

What it does To load the game

Implementation (how it works) Load the game

[239]:
```
public static int loadGame(BufferedReader inputStream, CastleGamePlayer[]␣
 ↪players, int playerCount, CastleRoom[] rooms, int roomCount) throws␣
 ↪IOException
{
    int i;
    for (i = 0; i < playerCount; i++)
    {
        String playerName;
```

```
        int score;
        boolean leave;
        playerName = inputStream.readLine();
        score = Integer.parseInt(inputStream.readLine());
        if (Integer.parseInt(inputStream.readLine()) >= 1)
        {
            leave = true;
        }
        else
        {
            leave = false;
        }
        players[i] = createPlayerRecord(playerName, score, leave);
    }
    for (i = 0; i < roomCount; i++)
    {
        int treasurePoints;
        int disableTrapPoints;
        boolean hasTrap;
        boolean hasTreasure;
        String roomName;
        roomName = inputStream.readLine();
        treasurePoints = Integer.parseInt(inputStream.readLine());
        disableTrapPoints = Integer.parseInt(inputStream.readLine());
        if (Integer.parseInt(inputStream.readLine()) >= 1)
        {
            hasTrap = true;
        }
        else
        {
            hasTrap = false;
        }
        if (Integer.parseInt(inputStream.readLine()) >= 1)
        {
            hasTreasure = true;
        }
        else
        {
            hasTreasure = false;
        }
        rooms[i] = createRoomRecord(roomName, hasTrap, hasTreasure,␣
 ↪treasurePoints, disableTrapPoints);
    }
    int firstPlayer = Integer.parseInt(inputStream.readLine());
    return firstPlayer;
}
```

Testing

[240]:
```
BufferedReader inputStream = new BufferedReader(new FileReader("game.txt"));
int playerCount = Integer.parseInt(inputStream.readLine());
CastleGamePlayer[] players = new CastleGamePlayer[playerCount];
int roomCount = Integer.parseInt(inputStream.readLine());
CastleRoom[] rooms = new CastleRoom[roomCount];
loadGame(inputStream, players, playerCount, rooms, roomCount);
inputStream.close();
```

2.7.39 Loops

What it does Give player a choice to searching the room to find a new object, taking the object previously found, disarming a trap or leave.

Implementation (how it works) Repeating the question until they choose to leave the room.

[241]:
```
public static int roomAction(CastleGamePlayer[] players, int playerCount,␣
 ↪CastleRoom[] rooms, int roomCount, int firstPlayer)
{
    String choice;
    boolean hasPlayer = true;
    while (hasPlayer == true)
    {
        hasPlayer = false;
        for (int i = firstPlayer; i < playerCount; i++)
        {
            CastleGamePlayer player = players[i];
            if (getCastleGameLeaving(player) == false)
            {
                hasPlayer = true;
                choice = askRoom(getCastleGamePlayerName(player), roomCount);
                if (choice.equals("q"))
                {
                    System.out.println("You quit the game.");
                    setCastleGameLeave(player,true);
                }
                else if (choice.equals("s"))
                {
                    // Return the id of current player when save
                    return i;
                }
                else
                {
                    CastleRoom room = rooms[Integer.parseInt(choice) - 1];
                    System.out.println("You entered a room.");
                    printSurroundings(room);
                    boolean leaveRoom = false;
```

```
                    while (leaveRoom == false)
                    {
                        choice = askChoice();
                        if (choice.equals("1"))
                        {
                            printSurroundings(room);
                        }
                        else if (choice.equals("2"))
                        {
                            setCastleGameScore(player, findCup(room,␣
↪getCastleGameScore(player)));
                        }
                        else if (choice.equals("3"))
                        {
                            setCastleGameScore(player, disableTrap(room,␣
↪getCastleGameScore(player)));
                        }
                        else if (choice.equals("4"))
                        {
                            System.out.println("You leave the room safely");
                            leaveRoom = true;
                        }
                        else
                        {
                            System.out.println("Please input choice between 1-4!
↪");
                        }
                        printScore(getCastleGameScore(player));
                    }
                }
            }
        }
        // Reset first player for next round
        firstPlayer = 0;
    }
    // Return -1 when no player left
    return -1;
}
```

Testing

[242]:
```
CastleGamePlayer[] players = new CastleGamePlayer[1];
players[0] = createPlayerRecord("Lily", 0, false);
String[] roomNames  = new String[] { "A", "B", "C" };
CastleRoom[] rooms = new CastleRoom[3];
createRooms(roomNames, rooms, 3);
roomAction(players, 1, rooms, 3, 0);
```

```
Lily, it is your turn. You are in a giant hall.
There are 3 rooms in front of you. Which room do you want to go? (type 1-3, and
q to quit, s to save):

 q

You quit the game.
```

[242]:
```
-1
```

2.7.40 Print the score

What it does Print the score

Implementation (how it works) Print the final score when all players leave the game

[243]:
```
public static void printFinalScore(CastleGamePlayer[] players, int len)
{
    System.out.println("Final score:");
    for (int i = 0; i < len; i++)
    {
        System.out.println(players[i].playerName + ": " + players[i].score + "␣
 ↪points.");
    }
}
```

Testing

[244]:
```
CastleGamePlayer[] players = new CastleGamePlayer[1];
players[0] = createPlayerRecord("Lily", 50, false);
printFinalScore(players, 1);
```

```
Final score:
Lily: 50 points.
```

2.7.41 Main

What it does Store the names of rooms and see whether everyone quit the game

Implementation (how it works) Store the names of rooms and see whether everyone quit the game

[245]:
```
public static void outerMethod7() throws IOException
{
    int playerCount;
    int firstPlayer;
    int roomCount;
    CastleGamePlayer[] players;
    CastleRoom[] rooms;
```

```
    String choice = askNewGame();
    if (choice.equals("1"))
    {
        playerCount = askQuestionInt("How many players? ");
        players = new CastleGamePlayer[playerCount];
        askPlayers(players, playerCount);

        String[] roomNames = new String[] { "Lab", "Study Room", "Common Room",␣
↪"Restroom", "Accomadation", "Laundry" }; // Room names
        roomCount = askQuestionInt("How many rooms? (max " + roomNames.length +␣
↪"): ");
        rooms = new CastleRoom[roomCount];
        createRooms(roomNames, rooms, roomCount);

        firstPlayer = 0;
    }
    else
    {
        BufferedReader inputStream = new BufferedReader(new FileReader("game.
↪txt"));
        playerCount = Integer.parseInt(inputStream.readLine());
        players = new CastleGamePlayer[playerCount];
        roomCount = Integer.parseInt(inputStream.readLine());
        rooms = new CastleRoom[roomCount];
        firstPlayer = loadGame(inputStream, players, playerCount, rooms,␣
↪roomCount);
    }

    firstPlayer = roomAction(players, playerCount, rooms, roomCount,␣
↪firstPlayer);

    if (firstPlayer < 0)
    {
        System.out.println("Everyone quit the game and the game is over.");
        printFinalScore(players, playerCount);
    }
    else
    {
        saveGame(players, playerCount, rooms, roomCount, firstPlayer);
        System.out.println("The game is saved. You can load the game next time␣
↪and the game will continue from the player who saved the game.");
    }

    return;
}
```

Testing

[246]:
```
outerMethod7 ();
```

```
Do you want to start a new game or load a saved game?
(type 1 for start a new game, or type 2 for load a saved game):

 2

Lily, it is your turn. You are in a giant hall.
There are 3 rooms in front of you. Which room do you want to go? (type 1-3, and
q to quit, s to save):

 q

You quit the game.
Everyone quit the game and the game is over.
Final score:
Lily: 0 points.
```

2.7.42 Running the program

Run the following call to simulate running the complete program.

[252]:
```
outerMethod7 ();
```

```
Do you want to start a new game or load a saved game?
(type 1 for start a new game, or type 2 for load a saved game):

 2

lily, it is your turn. You are in a giant hall.
There are 3 rooms in front of you. Which room do you want to go? (type 1-3, and
q to quit, s to save):

 1

You entered a room.
You are in Lab.
Do you what searching the room, taking any object found or leaving the room?
(type 1 for searching the room , type 2 for taking any object found, type 3 for
disarming a trap and type 4 for leaving the room)

 1

You are in Lab.
Your score is 0.
Do you what searching the room, taking any object found or leaving the room?
(type 1 for searching the room , type 2 for taking any object found, type 3 for
disarming a trap and type 4 for leaving the room)

 2

It's a trap!
Your score is 0.
```

```
Do you what searching the room, taking any object found or leaving the room?
(type 1 for searching the room , type 2 for taking any object found, type 3 for
disarming a trap and type 4 for leaving the room)

 3

You disarm the trap and it costs you 1 point(s).
Your score is -1.
Do you what searching the room, taking any object found or leaving the room?
(type 1 for searching the room , type 2 for taking any object found, type 3 for
disarming a trap and type 4 for leaving the room)

 4

You leave the room safely
Your score is -1.
lily, it is your turn. You are in a giant hall.
There are 3 rooms in front of you. Which room do you want to go? (type 1-3, and
q to quit, s to save):

 q

You quit the game.
Everyone quit the game and the game is over.
Final score:
lily: -1 points.
```

2.8 The complete program

This version will only compile here. To run it copy it into a file called initials.java on your local computer and compile and run it there.

[89]:
```java
import java.util.Random;
import java.util.Scanner;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.FileWriter;
import java.io.FileReader;
import java.io.IOException;


//Create a record for players' names and scores
class CastleGamePlayer
{
    String playerName;
    int score;
    boolean leave;
}

//Create a record for rooms
class CastleRoom
```

```
{
    String roomName;
    boolean hasTrap;
    boolean hasTreasure;
    int treasurePoints;
    int disableTrapPoints;
}

class CastleGameLevel7
{
    public static void main(String[] args) throws IOException
    {
        outerMethod7();
    }

    //Return the value in string
    public static String askQuestionStr(String question)
    {
        String value;
        Scanner scanner = new Scanner(System.in);

        System.out.println(question);
        value = scanner.nextLine();

        return value;
    }

    //Return the value in integer
    public static int askQuestionInt(String question)
    {
        int value;
        Scanner scanner = new Scanner(System.in);

        System.out.print(question);
        value = Integer.parseInt(scanner.nextLine());

        return value;
    }

    //Check whether the value is between bounds
    public static boolean checkForBounds(String input, int min, int max)
    {
        int x;
        try
        {
            x = Integer.parseInt(input);
        }
```

```
        catch (Exception err)
        {
            return false;
        }
        if( x < min || x > max)
        {
            return false;
        }
        return true;
    }

    public static String getCastleGamePlayerName(CastleGamePlayer player)
    {
        return player.playerName;
    }

    public static int getCastleGameScore(CastleGamePlayer player)
    {
        return player.score;
    }

    public static boolean getCastleGameLeaving(CastleGamePlayer player)
    {
        return player.leave;
    }

    public static String getCastleGameRoomName(CastleRoom room)
    {
        return room.roomName;
    }

    public static boolean getCastleGameHasTrap(CastleRoom room)
    {
        return room.hasTrap;
    }

    public static boolean getCastleGameHasTreasure(CastleRoom room)
    {
        return room.hasTreasure;
    }

    public static int getCastleGameTreasurePoints(CastleRoom room)
    {
        return room.treasurePoints;
    }

    public static int getCastleGameDisableTrapPoints(CastleRoom room)
```

```
    {
        return room.disableTrapPoints;
    }

    public static void setCastleGamePlayerName(CastleGamePlayer player, String␣
↪playerName)
    {
        player.playerName = playerName;
        return;
    }

    public static void setCastleGameScore(CastleGamePlayer player, int score)
    {
        player.score = score;
        return;
    }

    public static void setCastleGameLeave(CastleGamePlayer player, boolean␣
↪leave)
    {
        player.leave = leave;
        return;
    }

    public static void setCastleGameRoomName(CastleRoom room, String roomName)
    {
        room.roomName = roomName;
        return;
    }

    public static void setCastleGameHasTrap(CastleRoom room, boolean hasTrap)
    {
        room.hasTrap = hasTrap;
        return;
    }

    public static void setCastleGameHasTreasure(CastleRoom room, boolean␣
↪hasTreasure)
    {
        room.hasTreasure = hasTreasure;
        return;
    }

    public static void setCastleGameTreasurePoints(CastleRoom room, int␣
↪treasurePoints)
    {
        room.treasurePoints = treasurePoints;
```

```
        return;
    }

    public static void setCastleGameDisableTrapPoints(CastleRoom room, int␣
↪disableTrapPoints)
    {
        room.disableTrapPoints = disableTrapPoints;
        return;
    }

    //To ask what to do in the room
    public static String askRoom(String name, int len)
    {
        String choice;

        System.out.println(name + ", it is your turn. You are in a giant hall.
↪");
        choice = askQuestionStr("There are " + len + " rooms in front of you.␣
↪Which room do you want to go? (type 1-" + len + ", and q to quit, s to save):
↪ ");
        if (choice.equals("q") || choice.equals("s"))
        {
            return choice;
        }
        while (!checkForBounds(choice,1,len))
        {
            System.out.println("invalid input");
            choice = askQuestionStr("There are " + len + " rooms in front of␣
↪you. Which room do you want to go? (type 1-" + len + ", and q to quit, s to␣
↪save): ");
            if (choice.equals('q')) return choice;
        }
        return choice;
    }

    //Print out which room the player are
    public static void printSurroundings(CastleRoom room)
    {
        System.out.println("You are in " + getCastleGameRoomName(room) + ".");
        return;
    }

    //Give the player a score randomly
    public static int getPoint(int max)
    {
        Random bigdice = new Random();
        int diceroll = bigdice.nextInt(max) + 1;
```

```
        return diceroll;
    }

    //Tell the player whether they find the cup or touch the trap
    public static int findCup(CastleRoom room, int score)
    {
        if (getCastleGameHasTrap(room) == false)
        {
            if (getCastleGameHasTreasure(room) == true)
            {
                int point = getCastleGameTreasurePoints(room);
                System.out.println("You find a cup and get " + point + "␣
↪point(s).");
                score = score + point;
                setCastleGameHasTreasure(room, false);
            }
            else
            {
                System.out.println("You didn't find anything.");
            }
            }
        else
        {
            System.out.println("It's a trap!");
            score = 0;
        }
        return score;
    }


    //If the player choose to disarm the trap, they will lose points
    public static int disableTrap(CastleRoom room, int score)
    {
        int point = getCastleGameDisableTrapPoints(room);
        System.out.println("You disarm the trap and it costs you " + point + "␣
↪point(s).");
        score = score - point;
        setCastleGameHasTrap(room, false);
        return score;
    }

    //Print how many score they have
    public static void printScore(int score)
    {
        System.out.println("Your score is " + score + ".");
        return;
    }
```

```
   //Ask what to do in the room
   public static String askRoom(String name, int len)
   {
       String choice;

       System.out.println(name + ", it is your turn. You are in a giant hall.
↪");
       choice = askQuestionStr("There are " + len + " rooms in front of you.␣
↪Which room do you want to go? (type 1-" + len + ", and q to quit, s to save):
↪ ");
       if (choice.equals("q") || choice.equals("s"))
       {
           return choice;
       }
       while (!checkForBounds(choice,1,len))
       {
           System.out.println("invalid input");
           choice = askQuestionStr("There are " + len + " rooms in front of␣
↪you. Which room do you want to go? (type 1-" + len + ", and q to quit, s to␣
↪save): ");
           if (choice.equals('q')) return choice;
       }
       return choice;
   }

   //Ask the player whether to save the game or load the game
   public static String askNewGame()
   {
       String choice;

       System.out.println("Do you want to start a new game or load a saved␣
↪game?");
       choice = askQuestionStr("(type 1 for start a new game, or type 2 for␣
↪load a saved game): ");
       while (!checkForBounds(choice,1,2))
       {
           System.out.println("invalid input");
           choice = askQuestionStr("(type 1 for start a new game, or type 2␣
↪for load a saved game): ");
       }

       return choice;
   }

   //Store the records of the rooms
```

```
    public static CastleRoom createRoomRecord(String roomName, boolean hasTrap,␣
↪boolean hasTreasure, int treasurePoints, int disableTrapPoints)
    {
        CastleRoom room = new CastleRoom();
        setCastleGameRoomName(room, roomName);
        setCastleGameHasTrap(room, hasTrap);
        setCastleGameHasTreasure(room, hasTreasure);
        setCastleGameTreasurePoints(room, treasurePoints);
        setCastleGameDisableTrapPoints(room, disableTrapPoints);

        return room;
    }

    //Store the records of players
    public static CastleGamePlayer createPlayerRecord(String playerName, int␣
↪score, boolean leave)
    {
        CastleGamePlayer player = new CastleGamePlayer();
        setCastleGamePlayerName(player, playerName);
        setCastleGameScore(player, score);
        setCastleGameLeave(player, leave);
        return player;
    }

    //Ask for the players name
    public static void askPlayers(CastleGamePlayer[] players, int len)
    {
        String name;
        for (int i = 0; i < len; i++)
        {
            name = askQuestionStr("Player " + (i + 1) + ". What is your name?␣
↪");
            players[i] = createPlayerRecord(name, 0, false);
        }
    }

    //To create the records for the castle, whether the room have traps or␣
↪treasures or given randomly
    public static void createRooms(String[] roomNames, CastleRoom[] rooms, int␣
↪len)
    {
        for (int i = 0; i < len; i++)
        {
            boolean hasTrap = false;
            boolean hasTreasure = false;
            int treasurePoints = getPoint(6);
            int disableTrapPoints = 1;
```

```
            if (getPoint(2) > 1) hasTrap = true;
            if (getPoint(2) > 1) hasTreasure = true;
            rooms[i] = createRoomRecord(roomNames[i], hasTrap, hasTreasure,␣
↪treasurePoints, disableTrapPoints);
        }
    }

    //To save the game
    public static void saveGame(CastleGamePlayer[] players, int playerCount,␣
↪CastleRoom[] rooms, int roomCount, int currentPlayer) throws IOException
    {
        PrintWriter outputStream = new PrintWriter(new FileWriter("game.txt"));
        int i;
        outputStream.println(playerCount);
        outputStream.println(roomCount);
        for (i = 0; i < playerCount; i++)
        {
            outputStream.println(getCastleGamePlayerName(players[i]));
            outputStream.println(getCastleGameScore(players[i]));
            if (getCastleGameLeaving(players[i]) == true)
            {
                outputStream.println("1");
            }
            else
            {
                outputStream.println("0");
            }
        }
        for (i = 0; i < roomCount; i++)
        {
            outputStream.println(getCastleGameRoomName(rooms[i]));
            outputStream.println(getCastleGameTreasurePoints(rooms[i]));
            outputStream.println(getCastleGameDisableTrapPoints(rooms[i]));
            if (getCastleGameHasTrap(rooms[i]) == true)
            {
                outputStream.println("1");
            }
            else
            {
                outputStream.println("0");
            }
            if (getCastleGameHasTreasure(rooms[i]) == true)
            {
                outputStream.println("1");
            }
            else
            {
```

```
                outputStream.println("0");
            }
        }
        outputStream.println(currentPlayer);
        outputStream.close();
    }

    //To load the game
    public static int loadGame(BufferedReader inputStream, CastleGamePlayer[]␣
↪players, int playerCount, CastleRoom[] rooms, int roomCount) throws␣
↪IOException
    {
        int i;
        for (i = 0; i < playerCount; i++)
        {
            String playerName;
            int score;
            boolean leave;
            playerName = inputStream.readLine();
            score = Integer.parseInt(inputStream.readLine());
            if (Integer.parseInt(inputStream.readLine()) >= 1)
            {
                leave = true;
            }
            else
            {
                leave = false;
            }
            players[i] = createPlayerRecord(playerName, score, leave);
        }
        for (i = 0; i < roomCount; i++)
        {
            int treasurePoints;
            int disableTrapPoints;
            boolean hasTrap;
            boolean hasTreasure;
            String roomName;
            roomName = inputStream.readLine();
            treasurePoints = Integer.parseInt(inputStream.readLine());
            disableTrapPoints = Integer.parseInt(inputStream.readLine());
            if (Integer.parseInt(inputStream.readLine()) >= 1)
            {
                hasTrap = true;
            }
            else
            {
                hasTrap = false;
```

```
            }
            if (Integer.parseInt(inputStream.readLine()) >= 1)
            {
                hasTreasure = true;
            }
            else
            {
                hasTreasure = false;
            }
            rooms[i] = createRoomRecord(roomName, hasTrap, hasTreasure,␣
↪treasurePoints, disableTrapPoints);
        }
        int firstPlayer = Integer.parseInt(inputStream.readLine());
        return firstPlayer;
    }

    // Return the id of first player to play next time (after save/load).␣
↪Return -1 if there's no player left
    public static int roomAction(CastleGamePlayer[] players, int playerCount,␣
↪CastleRoom[] rooms, int roomCount, int firstPlayer)
    {
        String choice;
        boolean hasPlayer = true;
        while (hasPlayer == true)
        {
            hasPlayer = false;
            for (int i = firstPlayer; i < playerCount; i++)
            {
                CastleGamePlayer player = players[i];
                if (getCastleGameLeaving(player) == false)
                {
                    hasPlayer = true;
                    choice = askRoom(getCastleGamePlayerName(player),␣
↪roomCount);
                    if (choice.equals("q"))
                    {
                        System.out.println("You quit the game.");
                        setCastleGameLeave(player,true);
                    }
                    else if (choice.equals("s"))
                    {
                        // Return the id of current player when save
                        return i;
                    }
                    else
                    {
                        CastleRoom room = rooms[Integer.parseInt(choice) - 1];
```

```
                        System.out.println("You entered a room.");
                        printSurroundings(room);
                        boolean leaveRoom = false;
                        while (leaveRoom == false)
                        {
                            choice = askChoice();
                            if (choice.equals("1"))
                            {
                                printSurroundings(room);
                            }
                            else if (choice.equals("2"))
                            {
                                setCastleGameScore(player, findCup(room,␣
↪getCastleGameScore(player)));
                            }
                            else if (choice.equals("3"))
                            {
                                setCastleGameScore(player, disableTrap(room,␣
↪getCastleGameScore(player)));
                            }
                            else if (choice.equals("4"))
                            {
                                System.out.println("You leave the room safely");
                                leaveRoom = true;
                            }
                            else
                            {
                                System.out.println("Please input choice between␣
↪1-4!");
                            }
                            printScore(getCastleGameScore(player));
                        }
                    }
                }
            }
            // Reset first player for next round
            firstPlayer = 0;
        }
        // Return -1 when no player left
        return -1;
    }

    //Print the final score
    public static void printScore(CastleGamePlayer[] players, int len)
    {
        System.out.println("Final score:");
        for (int i = 0; i < len; i++)
```

```
        {
            System.out.println(players[i].playerName + ": " + players[i].score␣
↪+ " points.");
        }
    }

    //Ask whehter to save the game or load the game
    public static void outerMethod7() throws IOException
    {
        int playerCount;
        int firstPlayer;
        int roomCount;
        CastleGamePlayer[] players;
        CastleRoom[] rooms;

        String choice = askNewGame();
        if (choice.equals("1"))
        {
            playerCount = askQuestionInt("How many players? ");
            players = new CastleGamePlayer[playerCount];
            askPlayers(players, playerCount);

            String[] roomNames = new String[] { "Lab", "Study Room", "Common␣
↪Room", "Restroom", "Accomadation", "Laundry" }; // Room names
            roomCount = askQuestionInt("How many rooms? (max " + roomNames.
↪length + "): ");
            rooms = new CastleRoom[roomCount];
            createRooms(roomNames, rooms, roomCount);

            firstPlayer = 0;
        }
        else
        {
            BufferedReader inputStream = new BufferedReader(new␣
↪FileReader("game.txt"));
            playerCount = Integer.parseInt(inputStream.readLine());
            players = new CastleGamePlayer[playerCount];
            roomCount = Integer.parseInt(inputStream.readLine());
            rooms = new CastleRoom[roomCount];
            firstPlayer = loadGame(inputStream, players, playerCount, rooms,␣
↪roomCount);
            inputStream.close();
        }

        firstPlayer = roomAction(players, playerCount, rooms, roomCount,␣
↪firstPlayer);
```

```
        if (firstPlayer < 0)
        {
            System.out.println("Everyone quit the game and the game is over.");
            printFinalScore(players, playerCount);
        }
        else
        {
            saveGame(players, playerCount, rooms, roomCount, firstPlayer);
            System.out.println("The game is saved. You can load the game next␣
 ↪time and the game will continue from the player who saved the game.");
        }

        return;
    }
}
```

END OF LITERATE DOCUMENT

[ ]: