Lab Exercise 3 – Week 5 and Week 6

You may begin working on the lab exercise as soon as it is released to the class – you can start working at home and you don't have to wait until your scheduled lab time. We recommend that you read all of the instructions until the end before writing any code.

You can come in during scheduled lab times for CSCI 235 to ask questions and get help from the lab instructors and teaching assistants. You may discuss the tasks with your classmates and friends, but you are NOT allowed to share code, or even show your code to one another.

When you are done and ready to submit your lab to Moodle (on the lecture page, not the lab page), find your **src** folder within the project directory, zip it up, and submit the resulting archive file. Any other submissions, such as code copied and passed into text files will not be accepted.

Submission deadline for Part 1 is Friday, September 16, at 11:00 pm.

Submission deadline for Part 2 is Friday, September 23, at 11:00 pm.

The maximum grade for two parts of the lab is **2 points**. To get the maximum, all of the following tasks should be completed, and your code should compile and run properly. Late submissions will receive 0 points.

Lab Tasks

Part 1: Converting UML diagram to code.

1. In this lab, you are required to implement an object-oriented model for the game described below.
2. Your implementation must be consistent with the UML class diagram you are provided, but you will need to decide upon any additional fields, methods, classes, associations, etc. that are needed to meet the requirements.
3. You must ensure your classes are named in accordance with the specification provided.

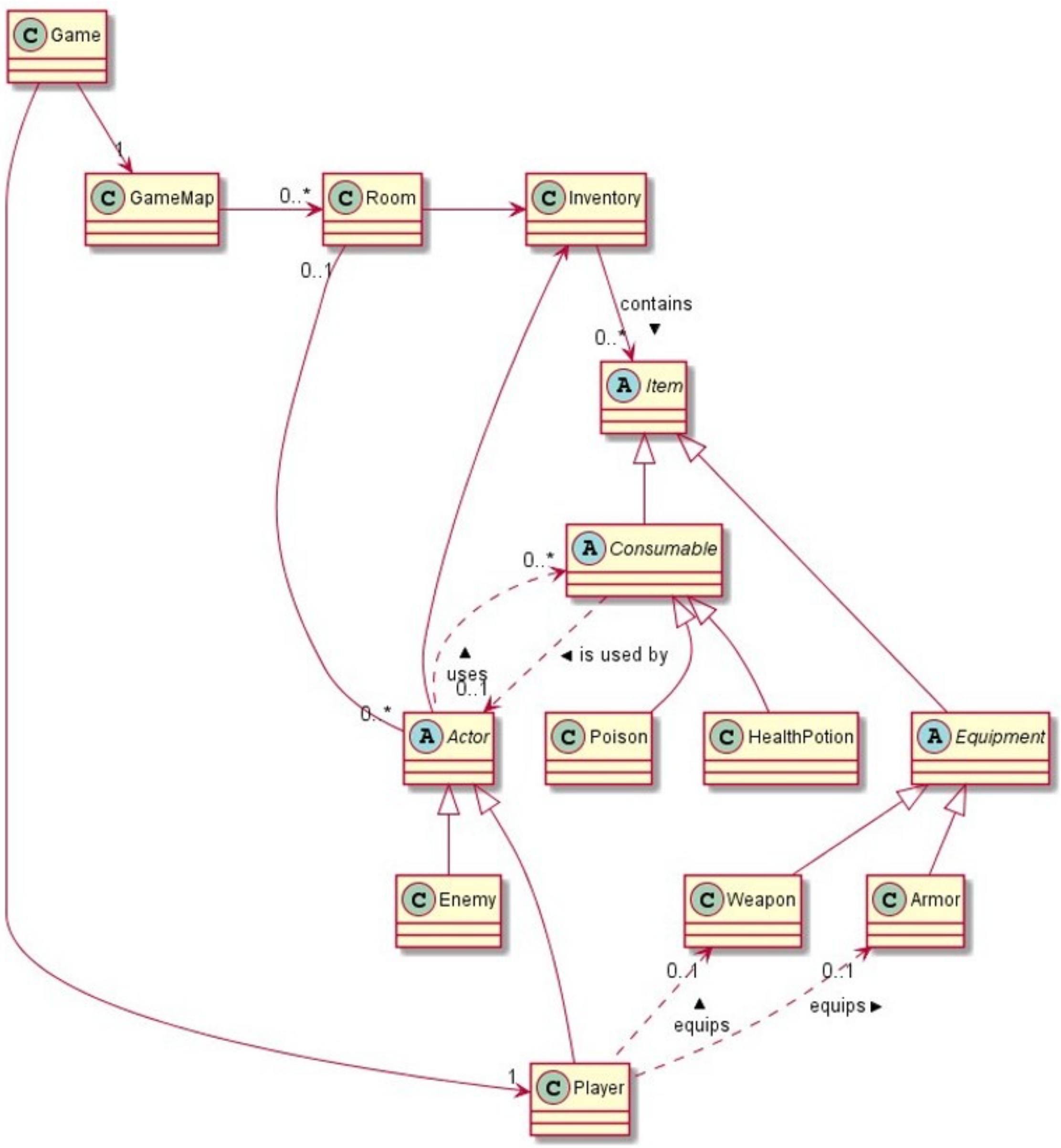
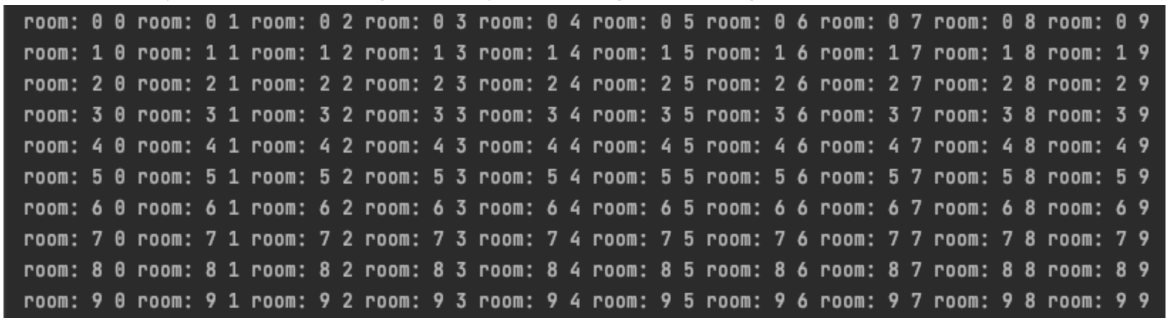


Figure 1 A UML Class Diagram, showing the required classes and associations between them.

Implementation:

1. Based on the given UML diagram in Figure 1, you should do these steps:
    a. Create three packages for *actors, game, inventory and the Main.java file outside of these pakcages.*
    b. Package *actors* consists of superclass Actor and two subclasses Enemy and Player
    c. Package *game* consists of GameMap and Room. Where GameMap consists of n*n grid of Rooms. The size (i.e. n) of the GameMap can be determined by the user during the creation (start) of the game.
    d. Package *inventory* consists of Inventory, Item, Consumable, Poison, HealthPotion, Equipment, Weapon, Armor. Where Inventory consists of several items (Item). Item is the superclass of Consumable and Equipment. Where Consumable is the superclass of Poison and HealthPotion classes. Equipment is the superclass of Weapon and Armor classes.
    e. The class Game should be outside of any packages.

f. Besides you need to have a main class, where you can start the game and print the map of the game. For that you need:
   - toString of Room, which prints“room: ” with i and j indices (where i – is rows’ index and j – is columns’ index)
   - printMap method in GameMap class, which is called by printMap method in Game classes.
   - For the first time you should get the output something like this, when for example, we defined a game map consisting of 10*10 grid of rooms.



```
room: 0 0 room: 0 1 room: 0 2 room: 0 3 room: 0 4 room: 0 5 room: 0 6 room: 0 7 room: 0 8 room: 0 9
room: 1 0 room: 1 1 room: 1 2 room: 1 3 room: 1 4 room: 1 5 room: 1 6 room: 1 7 room: 1 8 room: 1 9
room: 2 0 room: 2 1 room: 2 2 room: 2 3 room: 2 4 room: 2 5 room: 2 6 room: 2 7 room: 2 8 room: 2 9
room: 3 0 room: 3 1 room: 3 2 room: 3 3 room: 3 4 room: 3 5 room: 3 6 room: 3 7 room: 3 8 room: 3 9
room: 4 0 room: 4 1 room: 4 2 room: 4 3 room: 4 4 room: 4 5 room: 4 6 room: 4 7 room: 4 8 room: 4 9
room: 5 0 room: 5 1 room: 5 2 room: 5 3 room: 5 4 room: 5 5 room: 5 6 room: 5 7 room: 5 8 room: 5 9
room: 6 0 room: 6 1 room: 6 2 room: 6 3 room: 6 4 room: 6 5 room: 6 6 room: 6 7 room: 6 8 room: 6 9
room: 7 0 room: 7 1 room: 7 2 room: 7 3 room: 7 4 room: 7 5 room: 7 6 room: 7 7 room: 7 8 room: 7 9
room: 8 0 room: 8 1 room: 8 2 room: 8 3 room: 8 4 room: 8 5 room: 8 6 room: 8 7 room: 8 8 room: 8 9
room: 9 0 room: 9 1 room: 9 2 room: 9 3 room: 9 4 room: 9 5 room: 9 6 room: 9 7 room: 9 8 room: 9 9
```

g. From the UML diagram in Figure 1 you should notice that:
   i. some classes are marked as **abstract.** Please change the classes accordingly.
   ii. some classes have an **inheritance hierarchy**. Please change the classes accordingly.
   iii. Some classes which have association of more than one (i.e. “0..*”) should have a generic collection of that type. For example, Room should have a collection of Actors, Inventory class should have a collection of Items and so on…., i.e. you can use ArrayList or any other types of collections if you like.

Part 2: Main Part of the Implementation – Will be given for the next week, Week 6 with Deadline 23 September at 23:00

System Description

You are required to implement the classes required for a simple single-player game, which adheres to the provided UML class diagram (see UML for details). The game comprises a *player*, a number of *enemies*, a number of *items*, and a number of *rooms*. Rooms can contain a player, a number of enemies, and a number of items. The relative locations of rooms are set when a *map* is generated. The following game rules apply:

- **Weapon**, **Armor**, **Poison**, **HealthPotion** classes should have this:
  - o Each of the class have correspondingly one the powers: Weapon has attack Power, Armor has Defense Power, Poison has Destroy Power and HealthPotion has Healing Power.
  - o toString methods for printing the type of Item and its power.
  - o You might have additional fields and methods, if you like.
- **Inventory** should have this:
  - o For storing more than one Item, you need to create some collection for Items,
  - o Since Inventory is possessed by the Room and Actor, where the types and number of items could be different you can create a method which generates and adds those items to the list of items in this class using the Random generator. You can even create several methods for that.
  - o toString method for printing all items in the list.

    - You might have additional fields and methods, if you like. For example adding random items in a random amount to the room with random powers.
- **Room** class should have this:
    - For storing more than one Actor, you need to have some collection for Actors,
    - For storing inventory, one instance of inventory that will be passed to constructor should be enough. Since the inventory has own list of different Items.
    - toString method for printing inventory and all actors which were located in that room.
    - You might have additional fields and methods, if you like. For example for adding inventories and adding actors, that will be called by GameMap class later.
- **Actor** class should have this:
    - For storing inventory, one instance of inventory that will be passed to constructor should be enough. Since the inventory has own list of different Items.
    - You might have additional fields and methods, if you like. For example for adding inventories that will be called by GameMap class later or some getters.
- **Enemy** class should have this:
    - toString method for printing the type of Actor and its inventory items.
- **Player** class should have this:
    - toString method for printing the type of Actor and its inventory items.
- **GameMap** class should have this:
    - Once the game has created the gameMap with width*height rooms should be allocated first. (Hint: In the class GameMap you need to create a 2D array of type Room); (This part probably done in Part 1).
    - You might need a field for maximum number of enemies passed to GameMap constructor, so that we could know what is the limit of enemies that can be allocated to the GameMap.
    - GameMap should have a method which goes through every room in the map and adds random number of items and enemies to random rooms. Where room can have multiple items and enemies at the same time. This methods will be called by Game class later.
    - GameMap should have a method which allocates a player of the game to some random room to start the game.
    - GameMap should have a method to print every room with their containing items and actors. This methods will be called by Game class later.
    - You might have additional fields and methods, if you like.
- **Game** class should have this:
    - Since the size of the GameMap can be resizable, you should have a constructor, which takes width, height, maximum number of enemies and the player itself. (You should extend the constructor of the Game class from Part 1.)
    - In order to begin the game you will need a method, which allows to fill the GameMap with items, enemies and even player.
    - As well as print this game map with all the rooms and the content of each room. (This method just does the method call from GameMap and done in Part 1).
- Inside the **Main** class:

    - You need to have an instances of the Player and of the Game to begin the game with printing only the Game Map with all the rooms and their contained items.

Part 3: Advanced Level. Whole Game: - not required for the grade, only for those who is interested and enthusiastic about Java and OOP.

- Each **room** can have up to 4 neighboring rooms – one to the north, one to the south, one to the east, and one to the west.
- An **actor** can move from the room they currently occupy to any neighboring room.
- An **actor** can pick up any item that is in the room they currently occupy. Once the item has been picked up it enters the actor's inventory and is removed from the room.
- A player's inventory has a fixed capacity indicating the number of items it can hold. When a player's inventory is full, any attempt by that player to pick up an item should fail (i.e. return false).
- An actor can drop any item that is in their inventory. Once the item has been dropped it enters the inventory of the currently occupied room, and it is removed from the actor's inventory.
- An actor can use any item that is in its inventory. It can only use an item on an actor that occupies the same room. Once a consumable item has been used it is removed from the users' inventory.
- A player can equip any weapon that is in its inventory. A player can have only one weapon equipped at a time. The equipped weapon remains in the player's inventory.
- A player can equip any piece of armor that is in its inventory. A player can have only one piece of armor equipped at a time. The equipped armor remains in the player's inventory.
- An actor can attack any other actor that occupies the same room.
    - The power of the attack is determined by the actor's basic attack power plus the attack power associated with any equipped weapon.
    - The defensive power of the attacked actor is determined by its basic defensive power plus the defensive power associated with any equipped armor.
- When an actor is attacked there is one of two outcomes:
    - If the actor's defensive power is greater than the attacker's attack power, the actor will successfully defend against the attack and lose no health
    - If the actor's defensive power is less than or equal to the attacker's attack power the actor will fail to defend against the attack and its health will be reduced by the value of the attack power used against it.
- Poisons reduce the health of the actor they are used on by the value of the poison's power. Defensive power and armor provide no protection against poison
- Health increase the health of the actor they are used on by the value of the potion's healing power.
- If an actor's health goes to zero or below, it is dead. All items in its inventory should be dropped in the room it occupies and the actor should be removed from the map.

- If the player dies, the game is over, and the player loses.
- If the player kills all its enemies, the game is over, and the player wins.