CS 101 - Algorithms & Programming I

Fall 2025 - Lab 9

Due: Week of December 15, 2025

*Remember the **honor code** for your programming assignments.*
*For all labs, your solutions must conform to the CS101 style **guidelines**!*
All data and results should be stored in variables (or constants where appropriate) with meaningful names.

The objective of this lab is to understand and apply the object-oriented concepts, including abstract classes, interfaces, inheritance, and polymorphism, through an interactive game. Before starting the implementation, you are encouraged to think about how the game elements interact and organize them on paper as classes with clearly defined responsibilities and inheritance relations. Also, make sure where and how polymorphism is to be used.

0. Setup Workspace

Start VSC and open the previously created workspace named `labs_ws`. Now, under the `labs` folder, create a new folder named `lab9`.

In this lab, you are to create Java classes/files (under `labs/lab9` folder) as described below. You are expected to submit 10 files for your initial solution as described in a UML class diagram in the section below. For the revision, you may submit additional files, adding the `"Rev"` suffix to each revision file (e.g., `GameMgrRev`). **Please submit the files without compressing them, and ensure that no other or previous lab solutions are included.**

Output/gameplay is shared as **animated GIFs through Google Drive.** When possible, outputs of sample runs are shown in **brown,** whereas the user inputs are shown in **blue**.

Your code must match the names and types of variables and the interface of the methods specified in this document or any initial code provided.

1. Escape the Maze

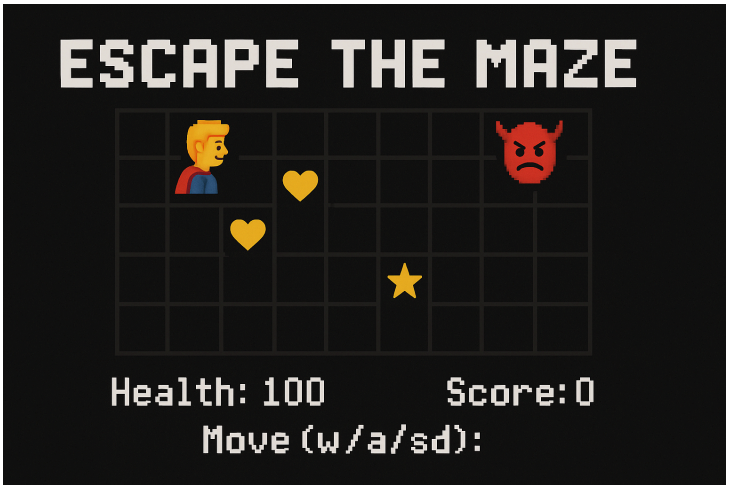
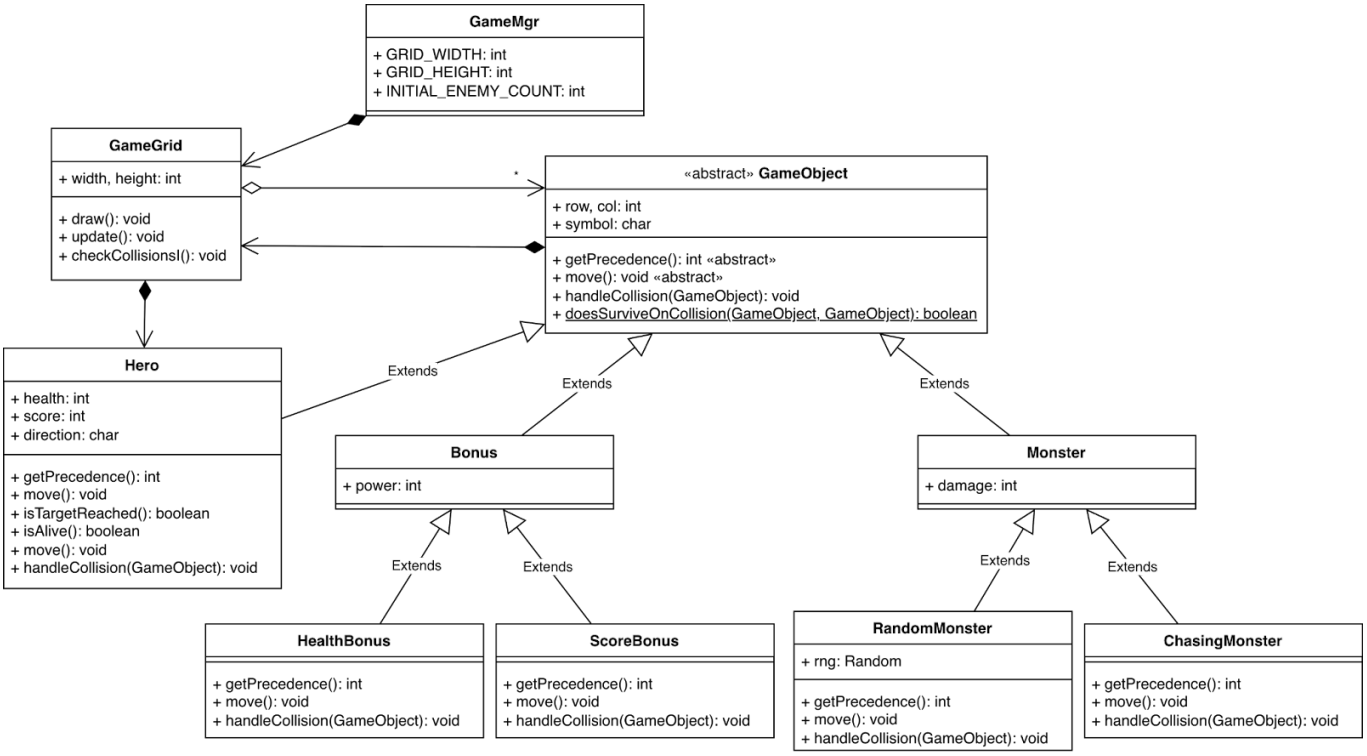


Image created by GPT

In this game, the hero starts from the top-left of a maze and aims to reach the bottom-right of the maze without getting killed, while scoring as many points as possible. The obstacles include monsters: one called a “random monster”, which moves randomly and damages the hero’s health (initially 100) by a certain amount. The other monster, called a “chasing monster”, is more dangerous as it not only damages the hero’s health by a certain amount but also actively chases the hero and survives upon collision with the hero.

Not everything the hero meets on their way to the target is dangerous, though. In fact, there are two kinds of bonuses placed and popping up in random locations. One is a health bonus, which increases the player’s health by a certain amount, and the other is a score bonus that increases their score by a certain amount.

The objects needed for implementing the game in an object-oriented manner can be described as follows. A UML class diagram summarizing the classes and their inheritance and association relations is shown below. Notice that this is not complete, and you may need additional data and functionality associated with each class. However, pay attention to polymorphic functionality with certain methods defined in a base class and overridden in subclasses.



A UML class diagram showing the classes and their inheritance and association relations

There are a number of different game objects that are placed on the maze, including the hero, enemies (random and chasing monsters), and bonuses of two kinds (health and score bonuses). So, you should implement classes for each of these types of game objects, making sure to use inheritance (with abstract classes and interfaces as needed) and polymorphism.

A game grid of size 10 by 6 (10 columns and 6 rows, where columns are numbered from 0 to 9 and rows are numbered from 0 to 5) representing the maze is needed as well. The hero starts out from (0,0) and targets to reach (5,9) in the maze. This class should maintain a list of all game objects. A separate reference to the hero should come in handy. Since the number of game objects during gameplay is expected to be a lot less than the total locations in the maze, you will maintain your game objects (including the hero) list as a list (an ArrayList). For convenience, the hero will always be the first game object in this list.

Finally, a game manager class that puts it all together is needed. This class would instantiate a game grid, create the hero, as well as an initial number of enemies and bonuses (initially spawn 5 game objects, with the probability of the game object being a bonus to be the same as being an enemy; similarly, if it's a bonus, a health bonus should be created with the same probability as a score bonus, and same goes for monsters). The game manager is also responsible for the game logic. Note that the game ends when either the target location is reached or the hero is dead, as their health goes down to zero or less. Alternatively, the user may quit by just typing 'q' as a direction. In each round of the game,

- The current state is displayed (i.e., printed out),
- The player inputs a direction for the hero (one of *awsdq, 'q'* is for quitting),
- All moving game objects, including the hero, are updated/moved,

- Collisions are checked for and handled properly, and
- A new game object is spawned in a random location, where its type is determined using the same rules as those where initial game objects.

The hero (and other moving game objects) move one row or column at a time, and they are not supposed to go out of the maze boundaries. Hence, when a game object is going left, for instance, its column location cannot be less than 0. In such cases, we should keep them in column 0. The hero's direction is controlled with the following keys/characters: *a*: left, *w*: up, *s*: down, and *d*: right.

The random monster moves in one of four directions randomly at each round, whereas the chasing monster moves towards the hero two-thirds of the time (with 67% probability) and stays put at other times. When moving towards the hero, whether to move closer to the hero by one location horizontally or vertically is decided randomly with an equal probability.

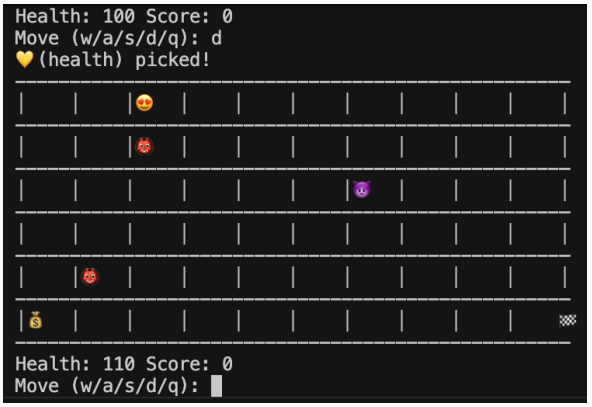
There are two types of collisions to handle: one is the collision of the hero with another game object, and the other is between two game objects that are neither the hero. Here is a list of what happens when a hero meets/collides with various types of game objects:

- Health bonus: hero's health is increased by 10, and the bonus is destroyed.
- Score bonus: hero's score increases by 1, and the bonus is destroyed.
- Random monster: the hero loses 10 of its health, and the monster is destroyed.
- Chasing monster: the hero loses 10 of its health, *but* the monster stays alive.

When non-hero game objects meet, we use a precedence rule to decide which object gets destroyed and which one survives. Precedence of game objects is unique, and from *highest to lowest* is as follows: chasing monster, health bonus, random monster, and score bonus. For instance, when a health bonus collides with a random monster, the random monster dies, and the health bonus survives.

Here are the icons used for the game objects and a sample screenshot from the gameplay:

- Player/Hero: 😍
- Random monster: 😈
- Chasing monster: 👹
- Health bonus: 🧡
- Score bonus: 💰



<u>Assume that no more than 3 game objects can be at the same grid position at a time when planning your draw method!</u>

Inheritance & Polymorphism

Notice that there is a very good opportunity to make use of inheritance in this implementation. A game object is the base class for not only the hero but also the bonuses and enemies. As game objects behave differently (e.g., some are stationary, whereas others move) and the action that needs to take place on the collision of two game objects varies highly depending on the game object types, a full-blown hierarchy of all game objects should be implemented. In addition, you should take advantage of polymorphism, especially in the following cases:

- When moving/updating the location of a game object
- When handling the collision of two game objects

Methods handling these should be defined as abstract at the base class (the game object class) and need to be overridden in subclasses as appropriate.

Class structures are partially provided, along with some partial implementations. You are expected to complete the remaining parts of the code, and you may add helper methods as needed.

Below are descriptions of some classes. You are expected to figure out and design the rest yourselves.

`GameMgr` Class:

This class serves as the entry point for the game. It creates an instance of `GameGrid`, spawns a number of monsters and bonuses in random locations of the grid, and starts the main game loop that manages user interaction and overall gameplay.`

Methods:

- `main`: Launches the game by creating a new `GameGrid` object, creating a random number of monsters and bonuses, and starting the game loop.

`GameGrid` Class:

This class implements the game board/grid, where the hero, as well as any bonuses and monsters, are located.

Instance Data Members:

- `width, height`: integer width and height of the game board/grid
- `objects`: game objects that are currently on the game grid (hero is always at index 0)
- `queuedObjectsForRemoval`: objects that are queued for removal at the end of this game round

Methods:

- `draw()`: renders the game grid with all game objects
- `update()`: updates all game object positions
- `checkAndHandleCollisions()`: check for and handle all game object collisions

`GameObject` Class:

This is the base class for all game objects, including the hero, the monsters, and the bonuses.

Instance Data Members:

- `row, column`: Integer row and column numbers to represent where this game object is
- `symbol`: A symbol that represents this game object on the grid/maze

Methods:

- `abstract getPrecendece()`: Each type of game object has a unique precedence (an integer) used to decide which survives and which gets destroyed upon a collision.
- `abstract move()`: Moves the game object in every game loop iteration. Notice that some game objects are stationary (e.g., bonuses), while others move (monsters).

- `handleCollisions(GameObject obj)`: Handles the collision of the input game object with this one (default behavior is implemented here).
- `doesSurviveOnCollision(GameObject obj, GameObject obj2)`: A static method that checks which of two non-hero game objects survives on collision; returns true if obj survives, false otherwise.

`Hero` Class:

This class implements the game player.

Instance Data Members:

- `health`: An integer variable tracking the hero's health, which starts from 100
- `score`: An integer variable tracking the hero's score, which increases each time a score bonus is destroyed
- `direction`: The direction in which the hero moves in the current round

Methods:

- `isAlive()`: Returns whether or not the current health level is greater than zero
- `isTargetReached()`: Returns whether or not the target position is reached (and the game is over)
- `takeDamage(int dmg)`: Takes damage of specified amount (reduces the health by this amount)
- `addHealth(int hlth)`: Increases the health by the given amount
- `increaseScore(int scr)`: Increases the score by the given amount

3. Gameplay

You can view a sample gameplay here.
Additionally, you can access the sample output as a text file here.