CS1112 Spring 2015 Project 6 Part A **Due Wednesday 5/6 at 11pm**

You must work either on your own or with one partner. If you work with a partner you must first register as a group in CMS and then submit your work as a group. Adhere to the Code of Academic Integrity. For a group, you below refers to your group. You may discuss background issues and general strategies with others and seek help from the course staff, but the work that you submit must be your own. In particular, you may discuss general ideas with others but you may not work out the detailed solutions with others. It is not OK for you to see or hear another student's code and it is certainly not OK to copy code from another person or from published/Internet sources. If you feel that you cannot complete the assignment on you own, seek help from the course staff.

Objectives

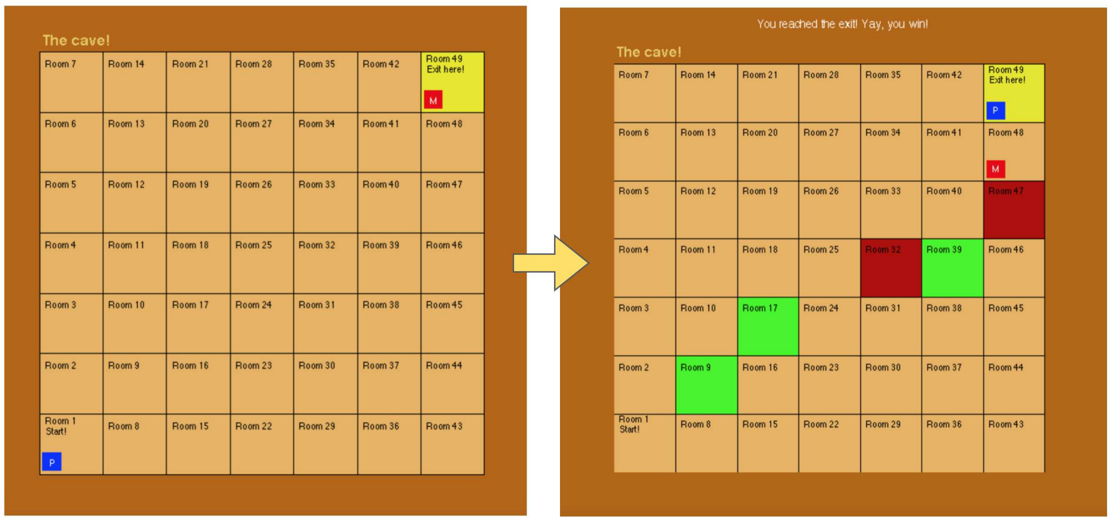
Part A of this project involves the completion of a game and will solidify your understanding of object-oriented programming, as well as give you practice on developing code incrementally—one class (or even one method) at a time. You will also be able to complete read the code of a graphical user interface (GUI) that reflects the game that you will be implementing.

1 A Game: Escape the Cave!

The objective of the player (marked "P" in the figure below) is to escape from a multi-storey cave that contains several rooms. Somewhere within the cave, however, is a malicious monster (marked "M" below) who will attempt to impede the player's progress! The player will begin in a room located in the bottom-left corner of the cave and will have to move through the rooms (while avoiding the monster) to reach the exit, which is at the top-right corner of the cave.

The player also has a finite number of health points, which may decrease along the player's journey to the exit depending on the kinds of rooms it enters. Normal rooms do not cause the player to lose health points. Poisonous rooms (shown in green below) damage the player's health and cause the player to lose points on each move afterwards until the player has been "unpoisoned" (which happens with a certain probability). There are also trap rooms (shown in dark red below) that simply cause the player's health points to decrease just once by some fixed amount. The player's quest will end abruptly if the monster catches it by moving into the same room, or if its health points are fully depleted.

The player makes two moves on each turn while the monster moves just once. If the player enters a poisonous or trap room, the room will change color so that you, the player, will know to avoid it from then on!



Game interface: The left image shows what the game looks like at the begining, and the right image shows the board after the player has moved to the exit and won!

1.1 Object-Oriented Design

We are providing you with the design of the classes whose methods you will need to implement in this project. In further sections of this document are detailed descriptions of each class.

Thinking about the various nouns and verbs you see when reading about this project may help you understand the object-oriented design. There are a few nouns used broadly in the game description, including *player*, *monster*, and *room*. These are different entities that interact with each other in the game, and can therefore be thought of as *classes* in our design. Some nouns can be organized into broader categories; for example, while a player and a monster are different, they are still both *characters* in our game and share a few characteristics. So, we may construct a *character* class and then have two subclasses *player* and *monster*.

Our design involves five classes: `Game`, `Character`, `Player` (subclass of `Character`), `Monster` (another subclass of `Character`), and `Room`. Following is a summary diagram of these classes.

Game	**Room**	**Character**	**Player** is a **Character**	**Monster** is a **Character**
Properties:	*Properties:*	*Properties:*	*Properties:*	*Methods:*
· roomArr	· id	· room	· health	· Monster
· player	· xCoord	*Methods:*	· poisoned	· moveToAttack
· monster	· yCoord	· Character	· poisonHit	· moveToProtect
· numRooms	· exit	· moveCharacter	· poisonEscape	· draw
Methods:	· playerInRoom	· disp	*Methods:*	· removeDrawing
· Game	· monsterInRoom		· Player	
· getMonstLoc	· hazardID		· getHealth	
· getPlayerLoc	· playerVisited		· decreaseHealth	
· getMonstDist	· hazardAmount		· checkPoison	
· draw	*Methods:*		· poison	
· updateGraphics	· Room		· move	
· run	· getLoc		· draw	
	· getID		· removeDrawing	
	· isHazardous			
	· applyHazard			
	· draw			

Which class should you work on first? Try working on the most independent class first, i.e. the one that doesn't depend much on other classes. You may wish to start working on the `Room` class first, as only one method (`applyHazard`) in this class requires that you refer to an object of another type, which is `Player`. Next you could work on the `Character` class as well as its subclasses, `Player` and `Monster`, since these classes have a `Room` as an instance variable. Then you can check that the `Room` and `Character` classes (and their subclasses) work together. Then you'll be ready to use the provided `Game` class to play!

In the following sections, you will find for each class an explanation of each of the properties and a list of the methods that you will implement. In the given MATLAB files, the methods that you need to implement are marked `% TO DO` and you should remove or change the statements inside those method bodies. Do not modify any of the properties or function headers in any of the classes. *Be sure to always use the available methods to accomplish a task instead of (re)writing code unnecessarily.*

2 Classes

2.1 Class `Room`

Properties This class has seven public properties, one private property, and one public constant:

- `xCoord` and `yCoord`: these are public numerical properties representing the coordinates of the *bottom-left* corner of a room. The room in which the player begins, at the bottom-left of the cave in the previous images, has the coordinates (1,1)—but that's just for your information since the provided `Game` class takes care of setting the room coordinates to match the coordinates in the room array.

- `exit`: this public property can either be 0 or 1; it is 0 if this room does not contain the exit from which the player can escape from the cave, and 1 if the room does have the exit. Note that only one room in the game will have an exit. In the `Game` class, we set the exit room to be the top-right room in the cave by default, so that is the only room whose `exit` property is 1.
- `playerInRoom`: like `exit`, this public property can either be 0 or 1; it is 1 if a `Player` object has this room as its current room and 0 otherwise. You will not need to worry too much about this property until you work on the `Player` class.
- `monsterInRoom`: this public property is similar to `playerInRoom`; it is set to 1 if a `Monster` object has this room as its current room and 0 otherwise. You will not need to worry about this property until you work on the `Monster` class.
- `hazardID`: this public property determines whether or not a room is hazardous, and if so, what kind of hazard it applies to a `Player` that enters it. If `hazardID` is 0, the room is a *normal* room that is not hazardous. If it is 1, the room is a *trap* room. If it is 2, the player is poisoned upon entering the room.
- `id`: this is a private property that gaves a numerical ID number to a room object. But because it is private, it cannot be accessed directly by other classes; it can only be accessed within the `Room` class. This is why you will need to implement a simple `getID()` function (described below).
- `playerVisited`: this public property has the value 0 or 1. It is 1 if this room has been visited by the player and 0 otherwise.
- `hazardAmount`: this is a public *constant*, meaning that its value cannot be changed. This is the fixed amount by which the player's health is decreased if the player enters a trap room. A constant may be accessed by the syntax *classname.constantname* and since this constant is public, you can access it anywhere (inside or outside of class `Room`) as `Room.hazardAmount`. *Use this constant (name) instead of just using its value in your code.*

Methods You will implement the following methods:

- `Room()`: this is the constructor for the room class. It has seven parameters corresponding to the first seven properties described above. If all seven arguments are provided, the properties should be set to those arguments. Be sure to use `nargin` appropriately to check the number of arguments passed. If there are not seven arguments provided, you should set the seven properties to be some reasonable default values. The property `playerVisited` should be set to 0.
- `getLoc()`: this method returns the room's x- and y-coordinates.
- `getID()`: this method returns the value of the private property `id`.
- `isHazardous()`: this method returns 0 if the room is not hazardous, and 1 if it is.
- `applyHazard()`: this method takes in a `Player` object[1] as an argument. If the room has a `hazardID` of 1, the player's `decreaseHealth()` method should be called to damage the player's health points by the amount `Room.hazardAmount`. If the room has a `hazardID` of 2, call the player's `poison` method (to poison the player). You may want to implement this method after you've finished the `Player` class.

Do not modify the `draw()` function. At this point, test this class' implementation by creating a few `Room` objects and calling their methods the Command Window. In fact, throughout the development of the remaining classes and methods, as much as possible *test them one by one* by instantiating an object of that class and calling the newly implemented methods.

[1]When we say "a `Player` object", we really mean "the handle to a `Player` object". We will use this shorthand throughout this document.

2.2 Class `Game`

The `Game` class has been fully implemented for you. This class is in charge of creating a 2-d array of `Rooms` that we call "the cave" for the game, instantiating a `Player` and a `Monster` object, getting and processing user input (clicks on the cave) and running the game algorithm, and dealing with graphics.

Although you don't need to write any code for this class, it is helpful to read the given code so that you know how the classes that you are implementing will be used. The provided code also gives you hints and syntax reminders.

Properties This class has six public properties:

- `roomArr`: a *square* matrix of `Room` objects
- `player`: a `Player` object. A game has only one player.
- `monster`: a `Monster` object. A game has only one monster.
- `exitRoom`: a `Room` object that is set to be the room in `roomArr` that is designated as the exit room; that is, it is the one room in the cave whose `exit` property is set to 1.
- `numRooms`: a numerical value representing the number of rooms in the cave. This number is always a *perfect square* so that the grid of rooms always has the same number or rooms in the vertical and horizontal directions.
- `type`: an integer value representing the monster's strategy. *[Revised 5/1:] If* `type` *is 1, the monster will always move using its* `moveToAttack` *method. If it is 2, the monster will always move using its* `moveToProtect` *method.* If it is any other value, the monster will choose randomly on each turn whether it will `moveToAttack` or `moveToProtect` on that turn. The methods `moveToAttack` and `moveToProtect` will be explained in the `Monster` class.

For now, read only the constructor of `Game` and note in particular how the nested loops create the 2-d array of `Rooms` for the game and takes care of setting all the coordinates and room ids:

```
g.roomArr(x,y) = Room(x,y,0,i,0,0,hazard);
```

`g` is the handle to the `Game` object that is being instantiated. The above statement says that the `Room` object `g.roomArr(x,y)` has the x- and y-coordinates `x` and `y` respectively. This simplifies our life! Later, when we need to move the player from one room to another, you just need to calculate the x- and y-coordinates and you will have the indices for the 2-d `roomArr` array.

Also shown in the constructor are that the player's `startRoom` is always set to the bottom-left room in the grid, the exit room is always set to be the top-right room, and the exit room's `hazardID` is always set to 0 (we wouldn't want the exit room to be a hazardous one!), and the monster always begins at the exit room.

2.3 Class `Character`

Properties This class has just one property, `room`, which is a `Room` object. Note the difference between `room` and `Room`: `room`, with a lowercase 'r', is the *name of a property* that belongs to a `Character` object. `Room`, with a capital 'R', is the *name of a class.* Property `room` is the (handle to the) `Room` object that the character currently occupies, so `room` refers to one of the `Room` objects in the 2-d array that makes up the cave of the game.

Methods You will implement the following methods:

- `Character()`: this is the constructor for the `Character` class. It takes in one argument, called `startRoom`, which is a `Room` *object.* The `room` property of the class, which is described above, should be set to `startRoom`. If no argument is given, then the `room` property should just be set to a "blank" `Room` object. You can create a blank `Room` object by writing `Room()`. (Do you know what this does? Think about the constructor in the `Room` class and what happens when we give that constructor fewer than seven arguments.)

- `moveCharacter()`: this method "moves" a `Character` object from one room to another by setting the `room` property to another `Room` in the cave. It takes in three arguments: an array of `Rooms` called `roomArr` and two integers `dx` and `dy`. `dx` and `dy` will each be one of three values: $-1, 0, 1$. `dx` and `dy` are the amounts by which the x- and y- coordinates of the room that the character is in will be changing. The characters (player and monster) can only move to a room that is adjacent horizontally, vertically, or diagonally. Hence the values $-1, 0, 1$. Happily the given methods in the `Game` class perform error checking (e.g., clicking outside the grid or not the adjacent rooms), so you just need to add `dx` and `dy` to the original coordinates to get the new coordinates. Note that in addition to setting the `room` property, this method should return the updated room.

Now that you have implemented the `moveCharacter` method, you may want to find out how the method is used and about the game algorithm ... Scan the `updateGraphics` method in class `Game`. You'll see how the game deals with user clicks (valid and invalid), how the different scenarios are checked using conditionals in order to produce different game messages, etc. Note that in our game the player makes two moves in one turn, so the player can win by entering the room occupied by the monster in step 1 and then reaching the exit in step 2.

2.4 Class `Player`

Properties This class has three public properties and one private property:

- `health`: this private property is the number of health points that the player has. The health points will decrease if the player encounters a hazardous room, that is, a room whose `hazardID` is greater than 0.
- `poisoned`: this public property is set to 0 if the player is not currently poisoned, and set to 1 if it is poisoned. Initially this property should be set to 0, and then switched to 1 if the player's current `room` has a `hazardID` of 2. Also, if the player is poisoned, its health will be reduced by `poisonHit` points each time it moves until it has been unpoisoned. Only after the player is unpoisoned can this property be reset to 0.
- `poisonHit`: this public property is the amount of points that will be deducted from the player's health points if the player is poisoned.
- `poisonEscape`: this public property is a real value between 0 and 1 that represents the probability that a player will become unpoisoned on its next move after having been poisoned. Unpoisoning the player should happen in the `move` method described below.

Methods You will implement the following methods:

- `Player()`: this is the constructor for the `Player` class. It takes in four arguments: `startRoom`, `startHealth`, `poisonHit`, and `poisonEscape`. `startRoom` is a `Room` object, and the `room` property that is inherited from the `Character` superclass should be set to this `startRoom`. Be sure to invoke the parent `Character` class; recall that the syntax for doing so is objectName@superclassName(Parameters). Then you should set the properties that are specific to the `Player` class. Further, if there are not four arguments provided, you should set the the `health`, `poisonHit`, and `poisonEscape` properties to reasonable default values of your choosing.
- `getHealth()`: this method simply returns the player's health points.
- `decreaseHealth()`: this method takes in a numerical argument called `damage`. The method should subtract `damage` from the player's `health` property. This method does not return anything.
- *[Revised 5/1:]* `poison`*: this method sets the player's* `poisoned` *property to 1.*
- `checkPoison`: this method returns the value of the player's `poisoned` property, 0 or 1.
- `move`: this method makes the player "move" by changing its `room`. First, the method will need to check if the player's health is greater than 0 (or else it cannot move!) and then set the `playerInRoom` property of the player's `room` to be 0, to indicate that the player is no longer in its old room. Then you should call

the `moveCharacter()` method from the parent class to actually change the player's `room`. After this is done, you will need to check if the player has entered a hazardous room or not and change its properties accordingly. If the player's new room is hazardous, simply call the `applyHazard()` method from the room class. If the player is poisoned, you will need to call the `decreaseHealth()` method because, as previously mentioned, each move when poisoned damages the player's health. Then, you can generate a random number to decide whether or not to unpoison the player (remember that `poisonEscape` is the probability of being unpoisoned). Therefore it is possible for the player both to get poisoned and instantly unpoisoned in the same move.

Do not modify the `draw()` and `removeDrawing()` methods.

2.5 Class `Monster`

Properties The monster does not have any additional properties; that is, its only property is the `room` property that is inherited from the parent `Character` class.

Methods You will implement the following methods:

- `Monster()`: this is the constructor for the `Monster` class. The only argument it takes in is a room object called `startRoom`. The use of this property is exactly the same as for the constructor in the `Player` class. Like in the player class, this constructor will need to call the superclass constructor to set the `room` property.
- `moveToAttack()`: this method moves the monster towards the player. It takes in one argument: a `Game` object, called `game`. You will use this object to access the matrix of `Room` objects in the game. This method should get the coordinates of the monster's current `room`, calculate the distance between itself and the player (the player's location can be obtained by accessing the `player` property of the `game` object), and then set two variables `dx` and `dy` (which are used in the same way as they are in the `moveCharacter` method) to move to the most appropriate location such that the distance between the monster and player is minimized. Then you can use `moveCharacter()` to actually cause the move to happen. Recall the formula for calculating the distance between two points, given their coordinates: distance = $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.
- `moveToProtect()`: this method makes the monster move closer to the exit room as a way of protecting that room. This method works quite similarly to `moveToAttack()`, except that here you should calculate the distance between the monster and the exit room and attempt to minimize it. The coordinates of the exit room can be accessed through the `game` argument.

Do not modify the `draw()` and `removeDrawing()` methods.

3 Playing the Game

The careful program development and testing should pay off now!

```
% Create a Game object called g, with 49 rooms, monster movement type of 1
% (meaning the monster will always moveToAttack), and probability .3 that
% a room will be hazardous.
g = Game(49,1,0.3)

% Start the graphics and let you play the game
g.run()
```

Try other game parameters! In particular, try other movement types of the monster. Good luck and have fun! And of course, submit your files `Room.m`, `Character.m`, `Player.m`, and `Monster.m` on CMS.

Part B will appear in a separate document. Both parts have the same due date.