DUE DATE: MARCH 9, 2018 AT 11:59PM

Instructions:

- You must complete the "**Blanket Honesty Declaration**" checklist on the course website before you can submit any assignment.
- Only submit the **java files**. Do ***not*** submit any other files. **Do not submit a ZIP file!**
- To submit the assignment, upload the specified files to the **Assignment 3** folder on the course website.
- Assignments must follow the **programming standards** document published on UMLearn.
- After the due date and time, assignments may be submitted but will lose **2% of the marks per hour** late. The last file submitted determines the late penalty for the entire assignment.
- If you submit a question multiple times, only the **most recent version** will be marked.
- These assignments are your chance to learn the material for the exams. Code your assignments independently. We use software to compare all submitted assignments to each other, and **pursue academic dishonesty vigorously**.
- To be eligible to earn full marks, your Java programs must compile and run upon download, without requiring any modifications.

Assignment overview

In this assignment, you will implement a text-based game that is heavily inspired by the game NetHack. NetNack is a classic open-source console-based game, and has been ported to most operating systems. See their homepage here, or play online here. NetHack is a very deep game, and we are only borrowing basic elements from it.

The key parts we are keeping are:

- There is a board of size m by n.
- Each element of the board may be a wall or an empty space.
- Each empty space can be occupied by one character or item: either the player, an enemy, a health potion or a trap.
- Commands are given by typing "up", "down", "left" or "right" (or just 'u', 'd', 'l', or 'r'). The player will only move in that direction if the space is unoccupied, and not a wall.
- If the space is occupied by a different character, the player attacks that character.

Keep all of your methods short and simple. In a properly-written object-oriented program, the work is distributed among many small methods, which call each other.

There are many files to hand in with this assignment, most of which are only 1 or 2 lines long. Only a few will need 20 lines of code or more. Writing good code that uses inheritance means writing the logic once, and using that logic in the subclasses.

Phase 1: The board

First, implement the abstract class **`Tile`**. This class represents the elements of the two-dimensional array that is the board (a **`Tile[][]`** array). The tiles will hold the game pieces or a wall. We will add more to this class later. To start, a **`Tile`** should have:

- Two instance variables: a **`String`** which is the symbol that should be displayed by this tile, and a **`boolean`** which will indicate if this tile is a tile that the player can move onto (is "passable").
- A constructor that has two parameters – the **`String`** and the **`boolean`**, in that order.
- Method **`String getSymbol()`** which fetches the symbol that should be displayed.
- Method **`boolean isPassable()`** which indicates if the tile is passable.

Now implement two subclasses of the **`Tile`** class: **`Wall`**, and **`OpenSpace`**.

- Class **`Wall`** is not passable, and the symbol is a hash: **#**

- Class **OpenSpace** is an empty space which is passable, and the symbol is a period: **.**

These subclasses will only contain constructors. All the other logic in them is inherited from class **Tile**.

You can test your classes using the supplied test program **TestPhase1.java**. You should get the output shown below.

```
####
#..#
#..#
####
```

Phase 2: Content

Next, implement the **abstract Content** class, which represents items that are sitting on the Tiles.

The **Content** class should have:

- One instance variable: a **String** that represents the symbol that will be drawn on the map.
- A constructor that accepts only one parameter (the **String**).
- A **String getSymbol()**method that returns the symbol for this item.

Implement an abstract subclass of **Contents** named **Item**. Subclasses of **Item** will be objects in the game that the player can touch to interact with. **Item** should have:

- A constructor with only a **String** parameter specifying the symbol.
- An **int getEffect()** method that returns how much of the player's health ("hit points") are added, or removed by touching this item. The default value to return is 0.

Now write three subclasses of **Item**:

- **Amulet** : which is marked by the character "Y" on the map. This is the "Amulet of Yendor". Picking up this item will result in winning the game. This item has no effect on hit points. This will be implemented later. In addition to returning a 0, the **getEffect()** method should also write "You picked up the Amulet of Yendor!" to the console.
- **HealthPotion** : Represented by an "h" on the map, a health potion has the effect of adding 5 hit points (health) to the player. In addition to returning a 5, the **getEffect()** method should also write "You picked up a health potion!" to the console.
- **Trap**: Represented by a "^" on the map, the player will lose 5 hit points when touching this item. In addition to returning a -5, the **getEffect()** method should also write "You set off a trap!" to the console.

These three classes only contain a no-parameters **constructor**, and a **getEffect()** method.

Now make the following modifications to class **Tile** and its subclasses:

- Add an instance variable of type **Content** to **Tile**. When there is no content to a tile, this variable should be **null**.
- Add methods **removeContent()**, which will set your **Content** instance variable to **null**, and a **getContent()** method which will return the contents of this tile, and a **setContent(Content)** method which will set the content of this tile.
- Change the **getSymbol()** method to return the symbol of the content, if there is a content in the tile. Otherwise, return the symbol of the tile itself.
- Add a second contructor to the **Tile** and **OpenSpace** classes, that accepts an additional parameter of type **Content**. The signature should be: **public Tile(String, boolean, Content)**. The **Wall** class should not have a constructor of this kind, since a wall cannot have any content.

You can test your class with **`TestPhase2.java`**. You should get the output shown below.

```
####
#^Y#
#^h#
####
```

Phase 3: Combatants

Combatants are the characters on the board. There is always exactly one player, and zero or more enemies. Both have "hit points" (the amount of health the combatant has). When the hit points become zero or lower, the combatant is removed from the board. Health potions can increase the player's hit points, while traps lower the player's hit points.

Combatants have attacks which vary in strength. There are minimum and maximum attack values. Each time an attack is made, a new random attack value between the minimum and maximum is returned. This amount of health is removed from the combatant that is being attacked.

Define an **abstract `Combatant`** class that is a subclass of **`Content`**. It should have:

- Three instance variables: **`int`** values for the health, and minimum and maximum attack values.
- A constructor which accepts parameters of type **`(String, int, int, int)`** which are the symbol, health, minimum, and maximum attack points, in that order.
- Three methods:
  - **`int getHP()`** – returns the current number of hit points (health)
  - **`int doAttack()`** – return a random number between the minimum and maximum attack values, inclusive. Investigate **`Math.random()`** to do this.
  - **`void changeHP()`** – add this amount to the combatant's hit points. The number could be negative.

Define three subclasses of **`Combatant`**:

- **`Player`**: represented by an "@" on the board, this is the playing piece the user will be able to move. The player should start with 100 hit points, and have a minimum attack value of 5, and a maximum of 10.
- **`Troll`**: represented by a "t" on the board. Trolls should start with 10 hit points, and have a minimum attack value of 1, and a maximum of 10.
- **`GreaterTroll`**, which is a ***subclass*** of Troll: Greater Trolls have double the hit points of **`Troll`**s, attack twice not once, and are represented by a "T" on the board. The constructor of **`GreaterTroll`** should call the constructor of **`Troll`**, then modify the hit points. The **`doAttack()`** method of **`GreaterTroll`** should call the **`doAttack()`** method in **`Troll`** twice, and return the total damage.
- These three subclasses should have only a no-parameters constructor.

Add **`toString()`** methods to the **`Combatant`** class, and all of its subclasses, that simply returns the class name.

You can test your class with **`TestPhase3.java`**. You should get the output shown below.

```
######
#^Y.t#
#^h@.#
######
```

Phase 4 Gameboard:

Create a **`Gameboard`** class that will hold and manage a 2D array of tiles that will represent the game state. To initialize the game board, data will be read in from a text file. The text file has the following format. The first row contains two integers that specify how many rows and columns are in the rectangular game board. The rest of the file gives the initial state of the game board. It has the same format as the output that we have been generating to this point: "Y" is the amulet, "@" is the player, and so on.

The **Gameboard** class, for this phase, only needs one instance variable of type **Tile[][]**.

Write a constructor that accepts a file name as a parameter (a **String**). Open that file, read the contents, and initialize your two dimensional array of tiles based on the file. It can be assumed that the file will be in the same folder as your code.

Write a **toString()** method that returns the game board as a multi-line **String**. The first line should give the current health of the player, and the remaining lines should show the board. See the examples below.

You can test your class with **TestPhase4.java**. You should get the output shown below. Make sure the supplied test data files (**phase4GameBoard1.txt** and **phase4GameBoard2.txt**) are in the same folder as the rest of your files.

```
Health: 100
#######
#^Y.t.#
#^h@.T#
#.h..h#
#######

Health: 100
#############
#^Y.t.......#
#^h..T..^...#
#.h..h.....@#
#############
```

Phase 5: The Game

Extend the **Gameboard** class to play a simple game with the game board we have created. Add:

- Static variables to define the four possible directions. You can set these to whatever values you like:
  - **public static final int UP**
  - **public static final int DOWN**
  - **public static final int LEFT**
  - **public static final int RIGHT**
- Two **boolean** instance variables: one indicating whether or not the game is over, and one indicating whether or not the player has won the game. Add accessor methods **getWon()** and **getDone()** for these instance variables.
- **void doRound(int)**: This method contains all the logic to control one move in the game. The parameter specifies the direction of the move: up, down, left or right. This will be a fairly large method.
  - If the player tries to move into a wall, the player should not move.
  - If the player moves into an empty space, the player should move to that space.
  - If the player moves onto a space where there is an **Item**, call **getEffect()** on the item, and change the user's hit points appropriately with **changeHP()**. If the item is the Amulet of Yendor (Y), the game is won. Set the **boolean** variables appropriately.
  - If the player attempts to move into a space occupied by a different combatant:
    - The player attacks that combatant. Use **doAttack()** to generate how many hit points the attack removes from the combatant. If the other combatant is reduced to 0 or fewer hit points, that combatant is removed from the board, and the player moves into that square.
    - If the combatant is not at 0 or fewer hit points, the player does not move, and the combatant attacks back, reducing the health of the player. If the player is reduced to 0 or fewer hit points, the player dies and the game is over.

- Print out to the console the number of hit points both combatants have after the attacks are complete.
- Print out to the console "[enemy type] is vanquished!" when a combatant is reduced to 0 or fewer hit points.

You can test your class with **`TestPhase5.java`**. Sample output is shown below, which shows fighting a combatant, and setting off a trap.

```
Welcome to NetHack, 1020 edition.
Health: 100
#############
#^Y.t.......#
#^h..T..^.^t#
#.h..h.....@#
#############

Which way would you like to move?
Valid commands are up, down, left, right.
up
Player attacks the Troll for 8
Troll attacks the player for 3
Player health:97 Troll health 2
Health: 97
#############
#^Y.t.......#
#^h..T..^.^t#
#.h..h.....@#
#############

Which way would you like to move?
Valid commands are up, down, left, right.
up
Player attacks the Troll for 7
Troll is vanquished!
Health: 97
#############
#^Y.t.......#
#^h..T..^.^@#
#.h..h......#
#############

Which way would you like to move?
Valid commands are up, down, left, right.
left
You set off a trap!
Health: 92
#############
#^Y.t.......#
#^h..T..^.@.#
#.h..h......#
#############

Which way would you like to move?
Valid commands are up, down, left, right.
left
Health: 92
```

```
#############
#^Y.t.......#
#^h..T..^@..#
#.h..h......#
#############

Which way would you like to move?
Valid commands are up, down, left, right.
```

Hand in

Submit your thirteen Java files. ***Do not submit .class or .java~ files! Do not submit a .zip file!*** You do ***not*** need to submit the `TestPhaseN.java` files that were given to you. If you did not complete all five phases of the assignment, use the Comments field when you hand in the assignment to tell the marker which phases were completed, so that only the appropriate tests can be run. For example, if you say that you completed Phases 1-2, then the marker will compile your files and also TestPhase2.java, and then run the main method in TestPhase2. If it fails to compile and run, you will lose ***all*** of the marks for the test runs. The marker will ***not*** try to run anything else, and will ***not*** edit your files in any way. (***Make sure none of your files specify a package at the top!)***