CSE 341, Spring 2011, Assignment 7
Due: Saturday, June 4, 11:00 PM

No late assignments will be accepted after that time, even if you have late days remaining.

Warm-Ups

Write the following top-level Ruby functions (not inside of any class/module). Use these when possible in later problems, and they may also be needed in order for the provided code to run properly.

1. Write a function `print_indented` whose argument is an `Enumerable`. It should return `nil`, and as a side-effect, print each element of the argument on a new line with two spaces of indentation. The lines should be sorted by the elements' `<=>` method. If the argument is `nil`, no side-effects should take place. Note: To see this function's usage, look at the provided `RoomState` class.

2. Write a function `when_detected` whose arguments are an `Enumerable enum` and an object `o`, and has an associated block when it is called. The function should look for the first object in `enum` that has a `to_s` equal to `o`'s `to_s`. If such an object exists, the block should be executed with the object as its parameter, and the value of the function should be whatever value the block returns. If no such object exists (or `enum` is `nil`), `when_detected` should return `nil` and the block should not be executed. Note: don't worry about `enum` containing `nil` as a element. To see this function's usage, look at the provided `RoomState` class.

The Adventure Game

> "Welcome to the 341 adventure game! I'm your host, the HomeworkSpec!" In this game, which you will be authoring, a player is put into the role of an adventurer cast into maze-like world. It's their job to navigate this maze of rooms in order to get to the "final room of victory." It will soon be brought to their attention that this may not always be so simple, though. There will be many challenging battles along the way and the smart adventurer should first acquire some items to better protect themselves if they ever want to succeed...

So this is your stage for this assignment. You will be creating a text-based version of the type of adventure described above in pure Ruby code. However, unlike the adventurer, you won't be alone since we've set up the basic infrastructure for the game. Our game will be played through a simple REPL[1]-like environment where the player will type commands and the game world will respond accordingly (exactly like sending messages to objects) using Ruby's reflection capabilities to find the correct command to execute. This is all done for you in the provided `Adventure#play` method.

To represent the game world's changing states (i.e. being in a room, in a fight, at the main menu, winning the game, etc.), our implementation will have a notion of game States. The game will always be in a single State at any time. Each State will then hold the current commands that the player can use and will be implemented by a Ruby class filled with methods prefixed with "`cmd_`" that will each individually act as a available command. To show this concept, we have provided the initial State that the game will be in, the `MainMenuState`.

This architecture allows for a large amount of flexibility when extending the game. For instance, if the developer wants to add a new command to the game, they only have to add a single method to a State class. In most languages, this may be bad if we wanted to add the same command for many different States (duplication of code). Since we're in Ruby, though, we can use the power of mixins the share commands (code) very robustly. This is where our provided `SystemState` module comes into play. The `SystemState` module contains the commands (such the `help` command) that are available for the player during all parts of the game. This module should be included into all of your State classes just like it is included in all of our provided State classes.

To complete this assignment, you will be required to write one new State, the `FightState`, representing representing when a player is in a battle against a group of foes. Also, you will need to write a mixin of commands related to dealing with the player in the `PlayerState` module. These player-related commands will be included in both the `FightState` and the provided `RoomState`. The following is a summary of unique commands for each of these (more detailed explanations follow).

Required commands summary:

`FightState`
- attack
- defend
- enemies

`PlayerState`
- inventory
- health

[1] Read-Eval-Print Loop

The format for defining the worlds that the player will be spelunking in is a simple nested hash structure. The reason for this design choice (as opposed to a class hierarchy) is to make writing new worlds simple and quick much like our philosophy on creating new commands with States. As such, there is a hash stored as a class variable within the `MainMenuState`, `@@worlds`, containing all of the possible worlds that the player can choose to play. Each individual world has the following basic format:

```
{:long_name => "World name",
 :desc => "World description",
 :health => 20,  # The player's starting health in this world
 :start_room => :room_key, # The room to first place the player
 :final_room => :room2, # The goal room for the player to reach
 :rooms => { # A hash from room id's to their details
   :room_key => { # A room with :room_key as its id
     :name => "Room Name",
     :desc => "Room description",
     :items => [...],
     :enemies => [...],
     :exits => {:north => :room_key,
                :south => :room2}},
   :room2 => ...}}
```

When a `play` command is entered, the named world is retrieved from `@@worlds` and used to create the new game.

There is an example world given in the provided code (at the bottom of the file) to show the structure more concretely. In order for the States to work properly, these mutable hash structures will be passed around to represent the state of the world as the player interacts with it.

The `:items` and `:enemies` keys in the world hash will be `Enumerable`s of class instances. You will be implementing the entire class structure for these classes by using the provided `Entity` class as a root for the hierarchy. An `Entity` is basically defined as something that exists in the world with a unique tag that can be used to identify it by the player.

Now that you know the basics, here are more of the specifics on what you should be implementing and a basic ordering to do it all in:

1. Extend the provided `SystemState` mixin module to include a private function `move_to_room` that, given a valid world hash (as defined above) and a valid room key in `world[:rooms]`, returns the next State as a result of moving the player to the specified room. Most of the time, the returned object is a new `RoomState`. However, it can also be a new `VictoryState` (which is provided and represents a good ending to the player's adventure) if a player moves into the final room, as configured by `world[:final_room]`, or it can be a new `FightState` if there are any enemies present in the given room. Note: This shouldn't alter any provided code. Instead, you should use Ruby's ability to add to modules after their initial definition. Also, you may first want to look at `RoomState`'s provided constructor and the specification for `FightState`'s constructor.

2. Write the `PlayerState` mixin module. This module handles basic commands having to do with the player and their state. All of this mixin's commands should work well with the structure defined in the provided `RoomState` for storing the player's current world/room (i.e. it uses `@world` and `@room` correctly. Likewise, this mixin should also later work with `FightState`'s structure. Specifically, this module should have the following behavior and commands:

   - The `inventory` command which prints out the current contents of the player's inventory/pockets (located in a world's `:inventory` key as an enum or nil) with each item's `to_s` on its own line and indented by two spaces. The lines should be sorted by the `to_s`'s natural ordering.
   - The `health` command which displays the player's current health.

3. Design and write a class hierarchy with its root being the provided `Entity` class. You should design a way to represent both items that can be taken from a room and enemies that can be battled with using those items. In particular, each enemy should at least have a number to represent its health, attack_power, and defense_power while each item should have attributes that at least represent an attack rating and a defense rating. What you do with attributes within a `FightState` should be non-trivial. You should have at least three distinct items and three distinct enemies with useful subclassing relationships for your solution. Notice that the provided code shows how to place these entities in the worlds through TestGame1's :room2 and :room3. You may write the `GoblinEnemy` and `ClownHammerWeapon` as implied by the example or not. It's up to you!

4. Write a `FightState` class that represents the player being in a fight with the group of enemies within a room. This State has the following commands and behavior:

   - A constructor that takes a world hash (something from `@@worlds`) and a room, storing these for later use in the class. The room should be assumed to have a non-empty group of enemies in it. Also, the message "You've been ambushed!" should be printed.
   - All the commands as specified for `PlayerState`.
   - The `enemies` command which is similar to the previously specified `inventory` command in `PlayerState` but lists the current room's enemies instead.
   - The `attack` command which uses the player's inventory items to lower a given enemy's health points. The enemy should be specified by its tag, as listed by the `enemies` command. If no enemy tag is given, this command should print "Attack what?" and then list the enemies for the player. After the player's attack has been concluded, all of the enemies in the room should (one-by-one) be made to counter-attack the player, lowering their health points. If at any point the player's health is zero, the returned State should be a new `FailureState` (which is provided and represents a bad ending to the player's adventure). If an enemy's health reaches zero, they should be removed from the enemies in the room. If all enemies have been removed, the returned State should be a new `RoomState` for the current room (no need to use `move_to_room` here).
   - The `defend` command which is like `attack` but only allows the enemies to perform their counter-attacks while the player's defense (from their inventory items) somehow lowers the enemies' attack strength (it's up to you on the semantics of this). This command should also heal the player (when and how is up to you).

Special Notes

- See the given given compiled jar executable for specifics on formatting. Your game should fit to the format of its output.
- If the write-up does not state an explicit return for a command, it should return `self` to indicate no change in the game State.
- If a command should do something when not provided with an argument, that command should use Ruby's ability to define default values to parameters to its advantage.
- There are cases where certain keys in various world hashes are not present but are usually `Enumerable`s. These cases should handled the same as if an empty `Enumerable` is present instead of nil.
- All State classes should have the `SystemState` commands even if not explicitly specified.
- To test your code, you should probably write/design a few more test worlds that what we have given you to test all of the cases.
- There are certain cases that haven't been specified in the write-up such as what happens when a player attacks without any inventory items (or even the rules on having multiple items while fighting). These are left for the implementer (you) to make some sane decisions about.
- To provide the help text for a command (as seen in the example solution jar), you may use the `set_cmd_help` method as seen in the provided State classes.

Bonuses

By virtue of our game's programming architecture, and it being written in Ruby, it is fairly easy to extend or add gameplay elements. For instance, you can easily add new commands to any of the States (like a `runaway` command in the `FightState`), extend your `Entity` hierarchy, or even extend the world format to include new things like doors, traps, or merchant shops without ever touching the provided code. For this assignment, we welcome your creative extensions (after you finish the basics, of course)! Students that decide to implement any non-trivial extensions will even receive a small amount of bonus points. When turning in the assignment, be sure to include a `README.txt` file describing all of the extensions to your game or we will assume you made an unextended version while grading.

Note: Only touch your own code while making extensions, not any of the provided code. If desired, you may use Ruby's ability to extend previosly defined modules/classes instead.

Partners

You may work with a partner on this assignment. If you do work with someone, the two of you should create and turn in a single assignment, and both partners will receive equal credit. Whether or not you include any additional bonus features in your game, you should include in a `README.txt` file a brief description of how you and your partner divided the work.

Assessment

Your solutions should be correct functionally as well as in good style. Good style issues include (but are not limited to):

- consistent/idiomatic indentation and line breaking
- good, idiomatic use of Ruby and its many built-in functions

Turn-in Instructions

- Create and turn in a text file containing a transcript of your travels through a game world at least as complex as the `TestGame1` sample. In Linux systems you can usually use select all/copy/paste to copy the contents of a terminal window if a more convenient print command is not available, or you can use the `script` command to capture a terminal session in a file.
- Put your solution in one file, `hw7.rb`, that includes both the provided code as well as your own code.
- The first line of your `.rb` file should be a Ruby comment with your name and, if you worked with a partner, your partner's name, and the phrase `Homework 7`.
- If you implement any bonus extensions or work with a partner (or both), you must also upload a `README.txt` file describing all of your extensions and/or how you and your partner divided the work, as appropriate.
- Turn in your files using the Catalyst dropbox link on the course website. If you work with a partner, only one of you should submit an assignment with both of your names on it.