CSE/ISE 337: Scripting Languages

Stony Brook University

Programming Assignment #2

Fall 2020

Assignment Due: Monday, Oct 19th, 2020, by 11:59 PM (EST)

Learning Outcomes

After completion of this programming project, you should be able:

- Design and implement scripts in Ruby
- Extend existing Ruby classes with your own functionality
- Design and implement a detailed object-oriented system

Instructions

- Take time read the problem descriptions carefully. The devil is in the details.
- Pay attention to the function names and class names. Incorrect names may lead to heavy penalties; could also result in your submission not being graded
- Create one file for each problem.
    - Name the first file **array.rb**
    - Name the second file **rgrep.rb**
    - Name the third file **hunt-treasure.rb**
- Zip all files and submit the zipped distribution
- Use the sample cases test cases to understand the problem and to test your code. However, we will use additional test cases to test your code.
- Create your own test cases to thoroughly test your code.
- All code must be written in Ruby.

Problem 1 (10 + 10 = 20 points)

The *Array* class in Ruby has numerous methods. Of these methods, two methods – *[]* and *map*, are of particular interest to us.

The *[]* method is used to obtain the value at a particular position in an array. For example, if *a* is an array such that a = [1,2,34,5], then *a[0] = 1, a[1] = 2,* and so on. If we call the *[]* method with an index that is out of bounds, then *[]* returns the *nil* value.

The *map* method when invoked on an instance of class *Array*, applies the code block associated with *map* to every element in the array, and then returns a new array. For example, if *a* is an array such that *a = [1,2,34,5]* and we call *map* on array *a* with the code block *{ |x| x.to_f }*, then *a.map { |x| x.to_f }* will result in a new array, a' *= [1.0,2.0,34.0,5.0]*.

We want to change the behavior of the *[]* and *map* methods in the *Array* class. For the *[]* method, we want to return a default value '\0' instead of *nil* when an out of bounds index is passed to the *[]* method. If an index that is not out of bounds is passed as argument to *[]*, then the method should return the value at that index. For example, for an array *a = [1,2,34,5]*, *a[2]* and *a[-2]* should return 34 since both indices are in bound. However, *a[5]* or *a[-6]* should return the default value '\0' since both of those indices are out of bound.

For the *map* method, we want to change its behavior such that it can be called with an optional sequence argument. When the sequence argument is provided, *map* should apply the associated code block to only those elements that belong to the indices in the sequence. If an invalid sequence is provided, then an empty array should be returned. If the sequence of indices is not provided, then the *map* method should default to it original behavior, that is, apply the associated code block to all elements in the array. Consider the following example:

```
b = ["cat","bat","mat","sat"]
b.map(2..4) { |x| x[0].upcase + x[1,x.length] }
b.map { |x| x[0].upcase + x[1,x.length] }
```

In the above example, the first call to *map* will result in the array `["Mat", "Sat"]` because the the associated code block is only applied to elements in positions 2,3, and 4 of array b. On the other hand, the second call to *map* will result in the array `["Cat", "Bat", "Mat", "Sat"]` since no sequence was provided. Hence the original *map* function, which applies the associated code block to all elements in the array was called.

If the sequence provided to the *map* method contains indices in the array that are out of bounds, then the new array should not contain any value for these indices. For example, if we provide the sequence (2..10) to our *map* method, as shown below, the new array should be `["Mat", "Sat"]`.

```
b = ["cat","bat","mat","sat"]
b.map(2..10) { |x| x[0].upcase + x[1,x.length] }
```

Sample Test Cases

```
a = [1,2,34,5]
a[1] = 2
a[10] = '\0'
a.map(2..4) { |i| i.to_f} = [34.0, 5.0]
a.map { |i| i.to_f} = [1.0, 2.0, 34.0, 5.0]

b = ["cat","bat","mat","sat"]
b[-1] = "sat"
b[5] = '\0'
b.map(2..10) { |x| x[0].upcase + x[1,x.length] } = ["Mat", "Sat"]
b.map(2..4) { |x| x[0].upcase + x[1,x.length] } = ["Mat", "Sat"]
b.map(-3..-1) { |x| x[0].upcase + x[1,x.length] } = ["Bat", "Mat", “Sat”]
b.map { |x| x[0].upcase + x[1,x.length] } = ["Cat", "Bat", "Mat", "Sat"]
```

Reference

https://ruby-doc.org/core-2.4.2/Array.html

Problem 2 (30 points)

grep is a command line utility tool for searching plain-text data sets for lines that match a regular expression. We want to develop a grep-like utility using Ruby. Our grep-like utility will be called **rgrep**. On a unix-based system we should be able to use **rgrep** by using the following command:

$ path/to/rgrep/rgrep.rb

On a non unix-based system (e.g., Windows) we will use the normal ruby command to call our utility

$ ruby path/to/rgrep/rgrep.rb

Just like the **grep** utility takes a filename and offers numerous options to search the filename, our **rgrep** utility will also takes a filename and based on the provided option combinations will return the search result. Following are the options it supports:

- **-w** <pattern>: treats <pattern> as a word and looks for the word in the filename. It returns all lines in filename that contains the word.
- **-p** <pattern>: treats <pattern> as a regular expression and searches the filename based on the regular expression. It returns all lines that matches the regular expression.
- **-v** <pattern>: treats <pattern> as a regular expression and searches the filename based on the regular expression. It returns all lines that *do not* match the regular expression.
- **-c** <pattern>: can only be used in conjunction with options **–w**, **-p**, or **–v**. For each conjunction, it returns the number of lines that match the pattern.
- **-m** <pattern>: can only be used in conjunction with options **–w**, or **-p**. For each conjunction, it returns the matched part of each line that match the pattern.

Usage

A user of **rgrep** should provide a file name (fully qualified path if not in current directory) followed by an option or a valid combination of options, and the pattern that will be used to search in the provided file.

Any other combination of options other than the ones listed above, should result in an error message “Invalid combination of options”

If any option other than the ones listed above is provided, an error message “Invalid option” should be reported.

If the combination of options is valid, then the order of the options does not matter.

The default option is **-p.**

Sample Test Cases

Consider a file "test.txt" that contains the following lines:
101 broad road
101 broad lane
102 high road
234 Johnson Street
Lyndhurst Pl 224

$ ./rgrep.rb
Missing required arguments

$ ./rgrep.rb test.txt
Missing required arguments

$ ./rgrep.rb test.txt -f
Invalid option

$ ./rgrep.rb test.txt –v –m '\d'
Invalid combination of options

$ ./rgrep.rb test.txt –w road
101 broad road
102 high road

$ ./rgrep.rb test.txt –w –m road
road
road

$ ./rgrep.rb test.txt –w –c road
2

$ ./rgrep.rb test.txt –p '\d\d'
101 broad road
101 broad lane
102 high road
234 Johnson Street

Lyndhurst Pl 224

$ ./rgrep.rb test.txt –p –c '\d\d'
5

$ ./rgrep.rb test.txt –v '^\d\d'
Lyndhurst Pl 224

$ ./rgrep.rb test.txt –v –c '\d\d'
1

$ ./rgrep.rb test.txt '\d\d'
101 broad road
101 broad lane
102 high road
234 Johnson Street
Lyndhurst Pl 224

Additional Notes

If you want to change your ruby file to an executable add #!/usr/bin/env to the first line of your file. Then, change the permissions of your file with the following command:

$ chmod +x rgrep.rb

However, this is not required. It is perfectly ok to run your program normally.

Problem 3 (50 points)

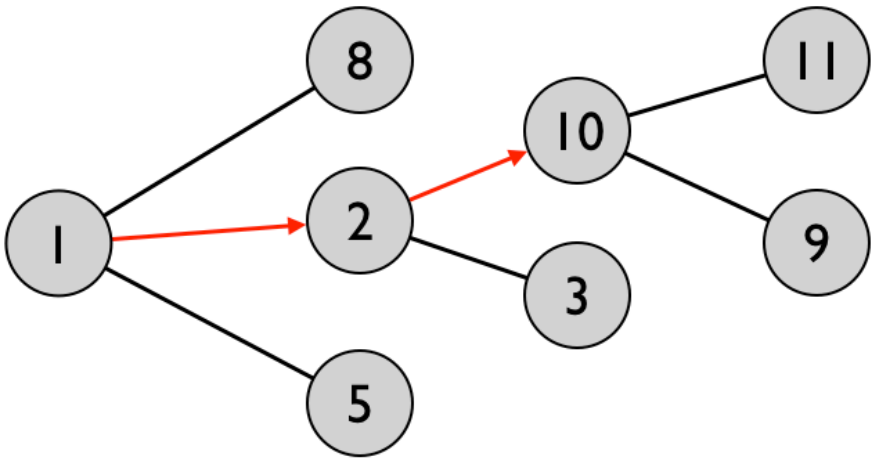
Treasure Hunt is a game that takes place in an underground cave network full of interconnected rooms. To win the game, a player need to locate the treasure hidden in one of the rooms while avoiding different hazards.

Game Demonstration

A player in this game can take only to two actions: move to adjacent rooms, or to shoot arrows into nearby rooms in an attempt to kill the guard protecting the treasure. Until the player knows where the treasure is, most of the time a player will end up moving from room to room to understand the cave's layout. Here is an example of how the game may proceed.

```
You are in room 1.

Exits go to: 2, 8, 5

---------------------------------------

What do you want to do? (m)ove or (s)hoot? m

Where? 2

---------------------------------------

You are in room 2.

Exits go to: 1, 10, 3

---------------------------------------

What do you want to do? (m)ove or (s)hoot? m

Where? 10

---------------------------------------

You are in room 10.

Exits go to: 2, 11, 9
```

After the player takes a couple of actions, she can begin to understand the topography of the cave.



Play continues in this fashion till the player encounters a hazard

```
What do you want to do? (m)ove or (s)hoot? m

Where? 11

----------------------------------------

You are in room 11.

Exits go to: 10, 8, 20

----------------------------------------

What do you want to do? (m)ove or (s)hoot? m

Where? 20

----------------------------------------

You are in room 20.

You feel a cold wind blowing from a nearby cavern.

Exits go to: 11, 19, 17
```

In this case, the player has managed to get close to a room with a bottomless pit, which is detected by the presence of cold wind emanating from one of the adjacent rooms.

Since hazards are sensed indirectly, the player needs a sensing mechanism to detect the rooms with hazards. Based on the topography of the cave known so far, the only two rooms with potential hazards are rooms 17 and 19. One of them might be safe or both might have hazards.

At this point, the player might guess a safe room. However, that would be too risky. The wise thing to do would be to backtrack and try another route:

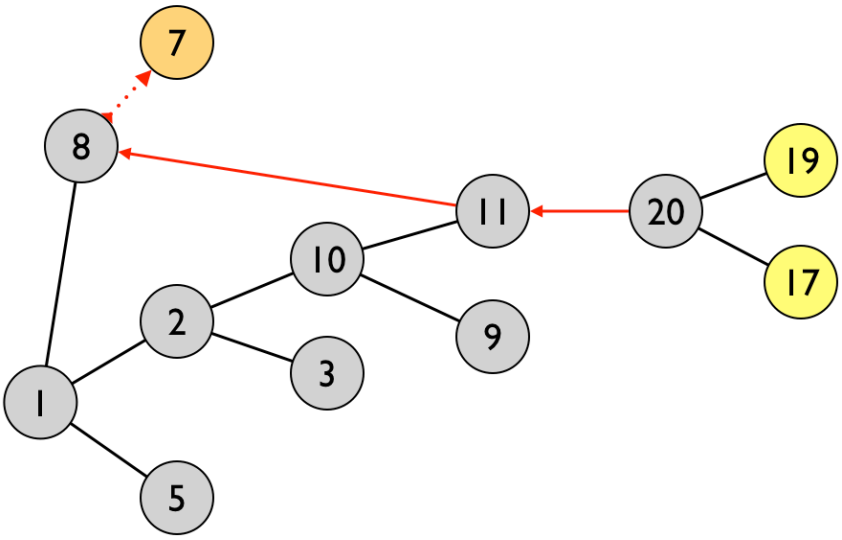
```
What do you want to do? (m)ove or (s)hoot? m

Where? 11

----------------------------------------
```

```
You are in room 11.

Exits go to: 10, 8, 20

----------------------------------------

What do you want to do? (m)ove or (s)hoot? m

Where? 8

----------------------------------------

You are in room 8.

You smell something terrible nearby

Exits go to: 11, 1, 7
```

Changing directions worked! On entering room 8, the terrible smell suggests that the guard protecting the treasure is nearby. Spending years in the cave has given the guard a peculiar stench. Luckily, the player has already visited rooms 1 and 11. Hence, the only other adjacent room 7 must contain the treasure:

```
What do you want to do? (m)ove or (s)hoot? s

Where? 7

----------------------------------------

YOU KILLED THE GUARD! GOOD JOB, BUDDY!!!
```

The game ends there. At this point the player's map would look like:



The player could have encountered other hazards in a room such as giant bats, which would have moved the player to a random room. Since such factors are randomized, every time this game is played a new cave map would be encountered.

We will discuss more about the details of the game as and when we describe the the details required to implement this game.

Implementation Details

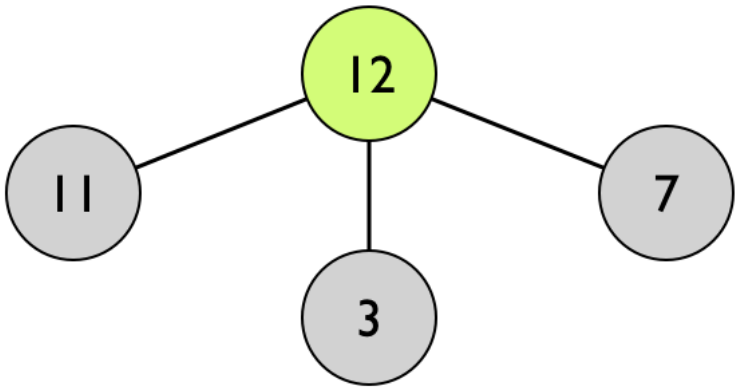
You need to implement this game by defining the following classes:

1. A *Room* class to manage hazards and connections between rooms
2. A *Cave* class to manage the overall topography of the cave
3. A *Player* class that handles sensing and encountering hazards

The following sections will describe the design and implementation details of each class along with the game rules. There is another class called the *Narrator*. Treat this class like a black box. You do not need to implement it, but it will help you understand the game. More on *Narrator* later.

Modeling a room

Structurally, rooms and their connections form a simple undirected graph:



Our *Room* class will manage these connections, and also make it easy to query and manipulate the hazards (bats, pits, and guard) that can be found in a room. The *Room* class should have the following attributes and behaviors:

- A room should have a *number* to identify the room
- A room *may* contain hazards
- A room *may* have two-way connections neighbors. For example, if a room R has a neighbor N, then R will be connected to N and vice-versa.
- A room knows the numbers of all neighboring rooms
- A room can choose a neighbor randomly
- A room is not safe if it has hazards
- A room is not safe if its neighbors have hazards
- A room is safe if it and its neighbors have no hazards

Let us walk through each of the above requirements and the tests that will be used to verify the requirements.

1. Every room has an identifying number that can be used by the player to keep track of where she is in the cave. Hence, we should be able to do the following:

```
room = Room.new(12)
room.number == 12   # true
```

2. Rooms may contain hazards, which can be added or removed as the game progresses. Rooms can also be checked for the presence or absence of hazards. Hence, we should be able to the following:

```
room.empty?          # true; initially room has no hazards

room.add(:guard)     # hazards can be added by add method
room.add(:bats)

room.empty?          # false, since room has hazards

room.has?(:guard)    # true; since :guard was added
room.has?(:wall)     # false; since :wall was never added

room.remove(:bats)   # hazards can be removed by remove method
room.has?(:bats)     # false; :bats were removed
```

3. Each room can be connected to other rooms in the cave. Hence, we should be able to do the following:

```
exit_numbers = [11, 3, 7]
exit_numbers.each { |i|
       room.connect(Room.new(i))
}
```

4. One-way paths are not allowed, that is, all connections between rooms are bi-directional. Hence, we should be able to do the following:

```
exit_numbers.each { |i|
       room.neighbor(i).number == i
       room.neighbor(i).neighbor(room.number) == room

}
```

5. Each room knows all of its exits, which consist of all neighboring room numbers. Hence, we should be able to do the following:

```
room.exits == exit_numbers # true
```

6. Neighboring rooms can be selected at random, which is needed for certain game events. Hence, we should be able to do the following:

```
exit_numbers.include?(room.random_neighbor.number)
```

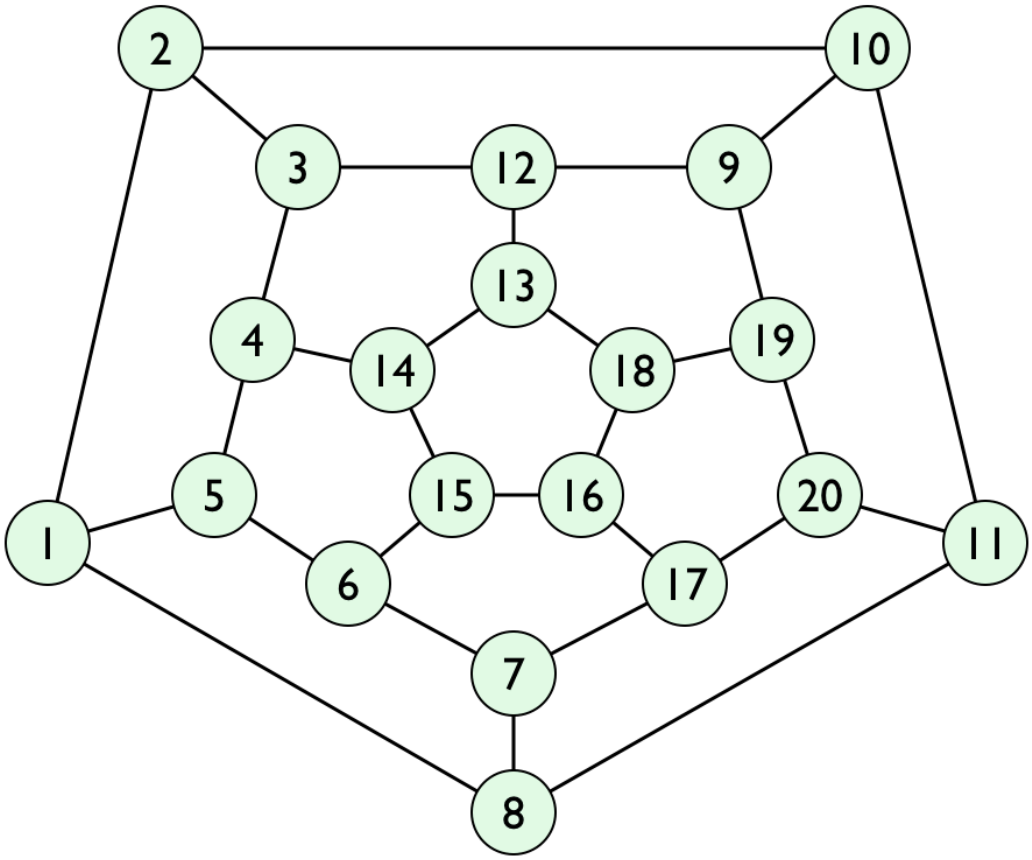
7. A room is considered safe if there are no hazards within it or any of its neighbors. Hence, we should be able to do the following:

```
room.add(:guard)
room.safe?           # false; since room has :guard
```

```
room
.random_neighbor
.add(:bats)
room.safe?          # false; since neighbor has :bats

room = Room.new(9)
room.safe?    # true; since room or neighbors have no hazards
```

Modeling the cave

Although this game can be played with any arbitrary cave layout, we will make things easier so we can focus on designing the steps of the game. Hence, we will assume a dodecahedron cave layout. In this layout, a room is placed at each vertex, and the edges form the connections between rooms. We can visualize the layout as follows:



To traverse the cave structure and to manipulate it, we will design a *Cave* class such that an instance of *Cave* has the following properties:

- has 20 rooms that each connect to *exactly* three other rooms
- can select rooms at random
- can move hazards from one room to another
- can add hazards at random to a specific number of rooms
- can find a room with a particular hazard
- can find a safe room to serve as an entrance

We will now understand each of these features with the following test cases:

1. The cave has 20 rooms, and each room is connected to exactly three other rooms. Hence, we should be able to verify this requirement as follows:

```
cave = Cave.dodecahedron
rooms = (1..20).each { |i| cave.room(i)}
rooms.each do |room|
   room.neighbors.count == 3
   room.neighbors.each { |i|
       i.neighbors.include?(room) == true
   }
```

2. Rooms in the cave can be selected randomly. Hence, we should be able do the following:

```
new_room = cave.random_room
```

3. Hazards can be moved from one room to a *different* room. Hence, we should be able to do the following:

```
room = cave.random_room
new_room = room.neighbors[1]
room.has?(:pit) == true
new_room.has?(:pit) == false
cave.move(:pit, room, new_room)
room.has?(:pit) == false
new_room.has?(:pit) == true
```

   The above code shows that a hazard is being moved from a room to one of its neighbors. However, a hazard can be moved from one to room any other room, doesn't have to be a neighbor.

4. Hazards can be randomly distributed throughout the cave. Hence, we should be able to do the following:

```
cave.add_hazard(:bats, 3)
rooms_with_bats = rooms.select { |e|
```

```
        e.has?(:bats)
}
rooms_with_bats.count == 3
```

Since there is no point adding a hazard to a room that already has that hazard, take care to implement *Cave#add_hazard* in a way that adds a hazard to a room only if the hazard doesn't already exist in the room.

5. Rooms can be looked up based on the hazards they contain. So, we should be able to do the following:

```
cave.add_hazard(:guard, 1)
cave.room_with(:guard).has?(:guard) == true
```

6. A safe entrance can be located. Hence, we should be able to do the following:

```
cave.add_hazard(:guard, 1)
cave.add_hazard(:pit, 3)
cave.add_hazard(:bats, 3)

entrance = cave.entrance

entrance.safe? == true
```

This is where the method *Room#safe?* is useful. Picking any room that passes the condition is enough to get the job done.

Modelling a player

Most events in this treasure hunt game are triggered by conditions based on the player's location. For example, imagine that the player is in Room 1 and the neighboring rooms are rooms 2,3, and 4. Also, room 2 has the guard, room 3 has bats, and room 4 is empty. With this setup, the player would sense the nearby hazards, resulting in the following:

```
You are in room 1.

You hear a rustling sound nearby

You smell something terrible nearby

Exits go to: 2, 3, 4
```

In this example, the player's possible outcomes would be as follows:

- The player will encounter the guard upon moving to room 2.
- The player will encounter bats upon moving to room 3.
- The player will not encounter any hazards in room 4.
- The player can shoot into room to kill the guard and access the treasure.
- The player will miss by shooting into rooms 3 or 4.

By now, you must have realized that the player's events can be easily generalized into the following:

1. The player can *sense* hazards
2. The player can *encounter* hazards
3. The player can perform *actions* on neighboring rooms

With these requirements, we can now model the *Player* class as an event-driven object that handles each event type or requirement listed above. The only state it explicitly needs to maintain is a reference to the room currently being explored; everything else can be managed externally via callbacks. This will become clearer when we look at how the game rules are implemented. For now, let us look at the tests that will help understand the behavior of the *Player* class.

Assuming that the *Player* class and its behavior is correctly implemented. *We* should be able to setup the player's environment as described in the example above. The following shows each step in the setup:

1. We will create an instance of the *Player* class

```
player = Player.new
```

2. We will create the rooms

```
empty_room = Room.new(1)
guard_room = Room.new(2)
bats_room = Room.new(3)
room4 = Room.new(4)
```

3. We will register the events that will be triggered based on the conditions in the player's location. We will do this via dummy callbacks meant to serve as stand-ins for actual game logic

```
sensed = Set.new
encountered = Set.new
```

```
empty_room.connect(guard_room)
empty_room.connect(bats_room)

player.sense(:bats) do
    sensed.add("You hear a rustling")
end
player.sense(:guard) do
    sensed.add("You smell something terrible")
end

player.encounter(:guard) do
    sensed.add("The guard killed you")
end
player.encounter(:bats) do
    sensed.add("The bats whisked you away")
end

player.action(:move) do |destination|
    player.enter(destination)
end
```

4. Once this setup is done, we should be able to test the behavior of the *Player* class as follows:

```
player.enter(empty_room)
player.explore_room
sensed == Set["You hear a rustling", "You smell something
terrible"]

encountered.empty? == true

player = Player.new
player.enter(bat_room)
encountered == Set["The bats whisked you away"]

sensed.empty? == true

# perform actions on neighboring rooms
player = Player.new
player.act("move, guard_room)
player.room.number == guard_room.number
encountered == Set["The guard killed you"]
sensed.empty? == true
```

These test cases will help verify if the right callbacks are being called by the *Player* class. The real use case of the *Player* class is to trigger operations on game

objects. This will become clearer in the next section where we succinctly define the game rules.

Defining Game Rules

The first thing we will need is a cave.

```
cave = Cave.dodecahedron
```

The cave will contain three pits, three giant bats, and the harshest and cruelest guard whose task is to protect the treasure.

```
cave.add_hazard(:guard, 1)

cave.add_hazard(:pit, 3)

cave.add_hazard(:bats, 3)
```

We will need a player to navigate the cave, and a narrator to regale us with tales of adventure. Consider the narrator as a black box. You can ignore the details of how the narrator is implemented since it is not necessary for your tasks.

```
player = Player.new
narrator = Narrator.new
```

Whenever a player senses a hazard, the narrator will give us a hint of what kind of trouble lurks around the corner. This will be done as follows:

```
player.sense(:bats) {
    narrator.say("You hear a rustling")
}

player.sense(:guard) {
   narrator.say("You smell something terrible")
}

player.sense(:pit) {
   narrator.say("You feel a cold wind blowing")
}
```

If upon entering, the player encounters the guard, the guard will become startled. We'll discuss the detailed consequences of this later, but the basic idea is that it will cause the guard to either run away to an adjacent room, or to gobble the player up:

```
player.encounter(:guard) {
    player.act(:startle_guard, player.room)
}
```

When bats are encountered, the narrator will inform us of the event, then a random room will be selected to drop the player off in. If any hazards are encountered in that room, the effects will be applied immediately, possibly leading to the player's demise.
But assuming that the player managed to survive the flight, the bats will take up residence in the new location. This can make navigation very complicated,

because stumbling back into that room will cause the player to be moved to yet another random location:

```
player.encounter(:bats) {
    narrator.say "Giant bats whisk you away to a new cavern"

    old_room = player.room
    new_room = cave.random_room

    player.enter(new_room)

    cave.move("bats, old_room, new_room)
}
```

If the player happens to come across a bottomless pit, the story ends immediately, even though the player's journey will probably go on forever:

```
player.encounter(:pit) {
    narrator.finish_story "You fell into a bottomless pit"
}
```

The player's actions are what ultimately ends up triggering game events. The movement action is straightforward: it simply updates the player's current location and then fires callbacks for any hazards encountered:

```
player.action(:move) { |destination|
    player.enter(destination)
}
```

Shooting is more complicated, If the player shoots into the room that the gurad is in, the guard is slayed and the player becomes rich because of the treasure. If instead the player shoots into the wrong room, then no matter where the guard is in the cave, it will be startled by the sound.

```
player.action(:shoot) { |destination|
    if destination.has?(:guard)
        narrator.finish_story "You killed the guard! Good job"
```

```
    else
        narrator.say "Your arrow missed"

        player.act(:startle_guard, cave.room_with(:guard))
    end
}
```

When the guard is startled, it will either stay where it is or move into one of its neighboring rooms. The player will be able to hear the guard move anywhere in the cave, even if it is not in a nearby room.
If the guard is in the same room as the player at the end of this process, it will gobble the player up and the game will end in sadness and tears:

```
player.action(:startle_guard) { |old_guard_room|
    if [:move, :stay].sample == :move # randomly select action
        new_guard_room = old_guard_room.random_neighbor
        cave.move(:guard, old_guard_room, new_guard_room)
    end

    if player.room.has?(:guard)
        narrator.finish_story "You woke up the guard and he
killed you"
    end
}
```

Reference

Use the *Array#sample* (https://ruby-doc.org/core-2.4.1/Array.html#method-i-sample) method in Ruby to randomly select a number from an array of numbers.