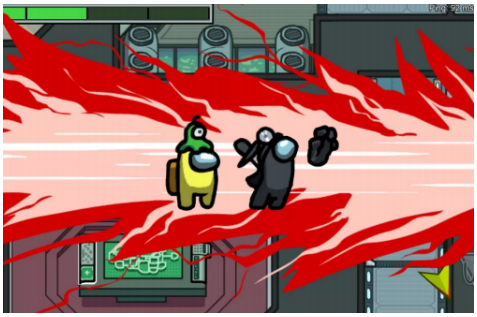
National University of Singapore
School of Computing
CS1010S: Programming Methodology

Extra Practice 8 Solutions

Question 1: Among Us

In a game of "Among Us", there are two groups of characters - normal crewmates and impostors.



The rules for our version are:

- Everyone starts off without a location and has to move to a location first (we'll assume this is not going to be a problem)
- Only the impostors can kill crewmates, and can only kill a crewmate if they are in the same location
- If a crewmate dies, he becomes a ghost and can still move around like a usual crewmate and do tasks
- Both impostors and crewmate can report a dead body, but must be in the same location as the dead body
- Impostors can pretend to do tasks as well

Define a class `Place` that is initialized with a name. It should have the following methods:

- `get_name()` that returns the name of the place
- `get_people()` that returns the names of all the people (dead or alive) in the location currently in a string `"[Person A], [Person B], [Person C] in [name of place]"`. If there is no one, return `"Nobody here"`

Define another class `Person` that is initialized with a name. It should have the following methods:

- `get_name()` that returns the name of the place
- `get_state()` that returns `"Alive"` if the person is still alive and `"Killed by [Murderer's name]"` if he/she is dead
- `do_task(task)` that takes in a task as an input string and returns the following sentence `"[Name of person] does [Name of task]"`
- `move(location)` that takes in a place object and moves the person from his current place to that new location

  - At the start of the game when this is called, return `"[Name of person] moves to [Name of place]"`
  - If the location is his current location, return `"Already here"`
  - Otherwise, return `"[Name of person] moves from [Current location name] to [New location name]"`
- `report(person)` that takes in a person object and tries to report it
  - If the person reporting is already dead, return `"Ghosts cannot report"`
  - If the person he is trying to report is oneself, return `"Cannot report oneself"`
  - If the person he is trying to report is not in the same location as he is in, return `"[Name of other person] is in [Name of location]"`
  - If the person he is trying to report is still alive, return `"[Name of person] is still alive"`
  - Otherwise, return `"[Name of person] reports [Name of other person]..."`
    He also immediately suspects everyone alive in the room at that moment and additionally returns the sentence `"... and suspects [Person A's name], [Person B's name] ... [Person D's name]"`
    If there is nobody else that he can suspect, the sentence `"... and nobody to suspect"` is returned

We now want to define another class `Impostor` that is a subclass of `Person`, and is initialized with two inputs - name and weapon. It should have the following additional methods:

- `kill(person)` that takes in a person object and attempts to kill it
  - If the person is oneself, return `"Cannot kill oneself"`
  - If the person is not in the same room as him, return `"[Victim's name] is in [Name of location victim is in]. Cannot kill"`
  - If the person is a fellow impostor, return `"Cannot kill another impostor"`
  - If the person is already dead, return `"Already killed"`
  - Otherwise, kill that person and return `"[Name] killed [name of victim] with a [name of weapon]"`
- `victim_list()` that returns the names of all the people he killed in alphabetical order `"[Name] killed [Person A], [Person B] ... [Person D]"`
  - If he has not killed anyone yet, return `"Killed nobody"`

Sample Execution:

```
>>> ravn = Impostor("Ravn", "Pistol")
>>> brendan = Impostor("Brendan", "Knife")
>>> daryl = Person("Daryl")
>>> tze = Person("Tze Lynn")
>>> ryan = Person("Ryan")
>>> clifton = Person("Clifton")
>>> daniel = Person("Daniel")
>>> room = Place("classroom")
>>> toilet = Place("toilet")
>>> hall = Place("hall")

>>> daniel.get_state()
'Alive'
>>> daniel.move(toilet)
'Daniel moves to toilet'
>>> daryl.move(room)
'Daryl moves to classroom'
>>> tze.move(hall)
'Tze Lynn moves to hall'
>>> ryan.move(toilet)
'Ryan moves to toilet'
>>> brendan.move(toilet)
'Brendan moves to toilet'
>>> ravn.move(room)
'Ravn moves to classroom'
>>> clifton.move(room)
'Clifton moves to classroom'
>>> room.get_people()
'Daryl, Ravn, Clifton in classroom'
>>> ravn.kill(ravn)
'Cannot kill oneself'
>>> ravn.victim_list()
'Killed nobody'
>>> tze.do_task("wiring")
'Tze Lynn does wiring'
>>> brendan.kill(clifton)
'Clifton is in classroom. Cannot kill'
>>> ravn.kill(clifton)
'Ravn killed Clifton with a Pistol'
>>> ravn.report(tze)
'Tze Lynn is in hall'
>>> ryan.report(brendan)
'Brendan is still alive'
>>> daniel.move(room)
'Daniel moves from toilet to classroom'
>>> ravn.report(clifton)
'Ravn reports Clifton and suspects Daryl, Daniel'
>>> ravn.victim_list()
'Ravn killed Clifton'
```

```
>>> clifton.get_state()
'Killed by Ravn'
>>> brendan.kill(ryan)
'Brendan killed Ryan with a Knife'
>>> brendan.move(room)
'Brendan moves from toilet to classroom'
>>> brendan.kill(ravn)
'Cannot kill another impostor'
>>> brendan.do_task("swipe card")
'Brendan does swipe card'
>>> tze.move(toilet)
'Tze Lynn moves from hall to toilet'
>>> tze.report(ryan)
'Tze Lynn reports Ryan and nobody to suspect'
>>> hall.get_people()
'Nobody here'
>>> room.get_people()    # order does not matter here
'Daryl, Ravn, Clifton, Daniel, Brendan in classroom'
>>> brendan.kill(daniel)
'Brendan killed Daniel with a Knife'
>>> ravn.report(daniel)
'Ravn reports Daniel and suspects Daryl, Brendan'
>>> brendan.victim_list()
'Brendan killed Daniel, Ryan'   # must be in alphabetical order
>>> brendan.kill(daniel)
'Already killed'
```

Solution:
Here's a possible implementation. Feel free to think of an alternative implemenation.

Disclaimer:
By right, any use or query of the **`name`** attribute inside functions other than the constructor method and the **`get_name()`** method should be replaced by the **`get_name()`** method due to abstraction.
For example, **`x.name`** should be replaced by **`x.get_name()`**.

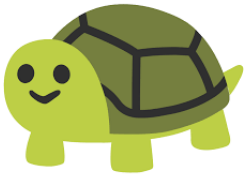
I will replace them by directly accessing the name attribute just because of space constraints, but you shouldn't during the actual PE or finals :)

Likewise, if you define another getter functions that returns the attribute as is, for example, **`get_place(self)`** that simply returns **`self.place`**, then you should do the same as the **`get_name()`** method. Otherwise, you will be considered breaking abstraction.

```
class Place:
    def __init__(self, name):
        self.name = name
        self.people = []

    def get_name(self):
        return self.name

    def get_people(self):
        if not self.people:
            return "Nobody here"
        return ", ".join(list(map(lambda x: x.name,
                                  self.people))) + f" in {self.name}"

class Person:
    def __init__(self, name):
        self.name = name
        self.place = None
        self.dead = False
        self.murderer = None

    def get_name(self):
        return self.name

    def get_state(self):
        if self.dead:
            return f"Killed by {self.murderer.name}"
        return "Alive"

    def do_task(self, task):
        return f"{self.name} does {task}"

    def move(self, location):
        if self.place == None:
            message = f"{self.name} moves to {location.name}"
```

```
        elif self.place == location:
            return "Already here"
        else:
            self.place.people.remove(self)
            message = f"{self.name} moves from {self.place.name} " + \
                      f"to {location.name}"

        location.people.append(self)
        self.place = location
        return message

    def report(self, person):
        if self.dead:
            return "Ghosts cannot report"
        elif self == person:
            return "Cannot report oneself"
        elif person.place != self.place:
            return f"{person.name} is in {person.place.name}"
        elif not person.dead:
            return f"{person.name} is still alive"
        else:
            msg = f"{self.name} reports {person.name} and "
            suspects = list(map(lambda x: x.name,
                                filter(lambda x: x != self and not x.dead,
                                       self.place.people)))
            if suspects:
                msg += "suspects " + ", ".join(suspects)
            else:
                msg += "nobody to suspect"

            return msg

class Impostor(Person):
    def __init__(self, name, weapon):
        super().__init__(name)
        self.weapon = weapon
        self.victims = []

    def kill(self, person):
        if self == person:
            return "Cannot kill oneself"
        elif self.place != person.place:
            return f"{person.name} is in {person.place.name}. " + \
                   "Cannot kill"
        elif isinstance(person, Impostor):
            return "Cannot kill another impostor"
        elif person.dead:
            return "Already killed"
        else:
            person.dead = True
```

```
            person.murderer = self
            self.victims.append(person)
            return f"{self.name} killed {person.name} with a " + \
                   f"{self.weapon}"

    def victim_list(self):
        if self.victims:
            return f"{self.name} killed " + \
                   ", ".join(sorted(map(lambda x: x.name,
                                        self.victims)))
        return "Killed nobody"
```



Carl-bot, possibly the most powerful turtle in the whole world

Question 2: Discord Muting Saga

Background (very okay to skip):
Discord is a VoIP, instant messaging and digital distribution platform designed for creating communities. Users communicate with voice calls, video calls, text messaging, media and files in private chats or as part of communities called "servers". Servers are a collection of persistent chat rooms and voice chat channels.

Discord communities are organized into discrete collections of channels called servers. Servers are referred to as "guilds" in the developer documentation. Users can create servers for free, manage their public visibility and create both channels and channel categories up to 250.

Channels may be either used for voice chat and streaming or for instant messaging and file sharing. The visibility and access to channels can be customized to limit access from certain users. Text channels support some rich text via a subset of the Markdown syntax. Code blocks with language-specific highlighting can also be used.

Source: Wikipedia

The question itself
In our Discord server, you might have noticed the existence of Carl-bot, a bot that has the ability to manage autoroles and automoderations on a server. With this bot, you get to be assigned to your respective tutorial channels.

However, Carl-bot has a feature that enables anyone with the server-managing permission to mute other members for any duration with any kinds of reasons. This is useful to moderate the server, but also dangerous as moderators can abuse their powers with the role they currently have.

In this case, we decided to not care if this feature is dangerous or not. Instead, we can simulate how we can **abuse Carl-bot's muting capability.**

In our "server", there are a few types of Discord users that we can implement, which is a regular user, a student, a tutor, or the owner of the server. There are also several channels which has a restricted access only for a list of roles. There are a few ground rules:

- A user can only send messages in a channel if the user has joined the channel and is not muted.
- A user can only join channels if the user is not muted.
- Only the owner and the tutors can mute any user. However, if the tutor is not a

server owner, he/she is not allowed to mute a fellow tutor. The same rule applies for unmuting.

- A muted user cannot do anything. Either joining a channel, muting someone else, sending a message, or unmuting someone else.
- There is no option to leave the server. Have fun in this chaos!

Define a class `Channel` that is initialized with a name and a list of strings representing the permitted roles. However, the initialized name has no hashtag in front of it, so we have to add it when we initialize a new channel. For example, from `"name"` to `"#name"`.
It should have the following methods:

- `get_name()` that returns the name of the channel
- `get_permitted_roles()` that returns the list of all permitted roles
- `get_members()` that returns the list of all members who joined the channel, sorted alphabetically
- `hall_of_mute()` that returns the list of all muted members who joined the channel, sorted alphabetically

Now define a class `User` that is initialized with a name. Once a `User` object is created, it has no role. It should have the following methods:

- `get_role()` returns `"[Name] is a [role]"` if the user has a role and `"[Name] has no role"` otherwise. However, if the user is the owner, return `"[Name] is the owner!"`
- `get_channels()` returns the list of all channel names the user joined, sorted alphabetically
- `join(channel)` that takes in a channel object and tries to join it
  - If the user has already joined the channel, return `"[Name] has already joined [Channel name]"`
  - If the user is muted, return `"[Name] is muted!"`. This is because a muted user cannot join a new channel
  - If either the user is the owner of the server or the user's role is in the channel's permitted roles, return `"[Name] joins [Channel name]"`
  - Else, the user has no permission, return `"[Name] has no permission to join [Channel name]"`
- `message(channel, msg)` that takes in a channel object and a message, then tries to send a message inside the channel
  - If the user has not joined the channel, return `"[Name] has not joined [Channel name]"`
  - If the user is muted, return `"[Name] is not allowed to send messages in [Channel name]"`
  - Else, return a notification in the format of `"[Channel name] - [Name]: [Message]"`
- `mute(other, time, reason)` that takes in:
  - `other`: Another `User` object that wants to be muted
  - `time`: The duration of the muting. If unspecified, it should be `None`
  - `reason`: The reason of the muting, which will be a string

and tries to mute the user with the given details:

- If the target user is the same as the muter, return `"Cannot mute oneself"`
- If the muter is muted, return `"[Name] is not allowed to send messages!"`
- If the target user is already muted, return `"[Other user's name] is muted!"`
- If the muter is the owner, mute the target user with the following rules:
  - If the time is unspecified, return `"[Name] muted [Other user's name] indefinitely. Reason: [Reason]"`
  - Else, return `"[Name] muted [Other user's name] for [Time] minutes. Reason: [Reason]"`
- If the muter is a tutor, the following rules apply:
  - If the target user is a fellow tutor, return `"Cannot mute a fellow tutor"`
  - If the target user is the owner, the same thing will happen because the owner is also a tutor in this case
  - Other than that, mute the target user with the same rule that applies when the owner is the muter
- If none of the above applies, the user has no permission to mute other users, so return `"[Name] doesn't have a permission to mute another user"`

- `unmute(other)` that takes in a `User` object and tries to unmute him/her
  - If the unmuter tries to unmute oneself while being muted, return `"[Name] is not allowed to send messages!"`
  - If the target user is not muted, return `"[Other user's name] is not muted :)"`
  - If the unmuter is the owner, always unmute the target user and return `"[Name] unmuted [Other user's name]!"`
  - If the unmuter is a tutor, the following rules apply:
    - If the target user is also a tutor, return `"Cannot unmute a fellow tutor"`
    - Else, unmute the target user and return `"[Name] unmuted [Other user's name]!"`
  - Else, the user has no permission to unmute other users, so return `"[Name] doesn't have a permission to unmute another user"`

Next, define a class `Tutor` that is a subclass of `User`. The only difference between `Tutor` and `User` is that when being initialized, `Tutor` will have the role `"Tutor"` instead of `None`. You may override the methods defined in the `User` class if needed.

Similarly, define a class `Student` that is a subclass of `User`. The only difference between `Student` and `User` is that when being initialized, `Tutor` will have the role `"Student"` instead of `None`. You may override the methods defined in the `User` class if needed.

Finally, define a class `Owner` that is also a `Tutor`. However, the owner's role will be a list of three roles, `["Owner", "Tutor", "Student"]`. You may override the methods defined in the `User` class if needed.

Sample Execution (a very very long one):

```
>>> general = Channel("general-helpdesk", ["Owner", "Tutor", "Student"])
>>> announcements = Channel("announcements", ["Owner", "Tutor"])
>>> secret = Channel("foobar", ["Owner"])

>>> general.get_name()
'#general-helpdesk'
>>> secret.get_name()
'#foobar'

>>> russell = Owner("Russell")
>>> clifton = Tutor("Clifton")
>>> aeron = Student("Aeron")
>>> kenghwee = User("Keng Hwee")

>>> def display_hall_of_mute(): # don't mind this function
        channels = [general, announcements, secret]
        print()
        print("HALL OF MUTE")
        for channel in channels:
            print(f"{channel.get_name()}: {channel.hall_of_mute()}")
        print()

>>> russell.get_role()
'Russell is the owner!'
>>> clifton.get_role()
'Clifton is a Tutor'
>>> aeron.get_role()
'Aeron is a Student'
>>> kenghwee.get_role()
'Keng Hwee has no role'

>>> russell.join(general)
'Russell joins #general-helpdesk'
>>> russell.join(general)
'Russell has already joined #general-helpdesk'
>>> clifton.join(general)
'Clifton joins #general-helpdesk'
>>> russell.mute(aeron, None, None)
'Russell muted Aeron indefinitely. Reason: None'
>>> aeron.join(general)
'Aeron is muted!' # therefore cannot join
>>> russell.unmute(aeron)
'Russell unmuted Aeron!'
>>> aeron.join(general)
'Aeron joins #general-helpdesk'
>>> kenghwee.join(general)
'Keng Hwee has no permission to join #general-helpdesk'
>>> general.get_members()
['Aeron', 'Clifton', 'Russell']
```

```
>>> russell.join(announcements)
'Russell joins #announcements'
>>> clifton.join(announcements)
'Clifton joins #announcements'
>>> aeron.join(announcements)
'Aeron has no permission to join #announcements'
>>> kenghwee.join(announcements)
'Keng Hwee has no permission to join #announcements'
>>> announcements.get_members()
['Clifton', 'Russell']

>>> russell.join(secret)
'Russell joins #foobar'
>>> clifton.join(secret)
'Clifton has no permission to join #foobar'
>>> aeron.join(secret)
'Aeron has no permission to join #foobar'
>>> kenghwee.join(secret)
'Keng Hwee has no permission to join #foobar'
>>> secret.get_members()
['Russell']

>>> russell.get_channels()
['announcements', 'foobar', 'general-helpdesk']
>>> clifton.get_channels()
['announcements', 'general-helpdesk']
>>> aeron.get_channels()
['general-helpdesk']
>>> kenghwee.get_channels()
[]

>>> russell.message(announcements, "Tutorial is canceled!")
'#announcements --- Russell: Tutorial is canceled!'
>>> clifton.message(general, "Hooray!")
'#general-helpdesk --- Clifton: Hooray!'
>>> aeron.message(announcements, "Tutorial is canceled!")
'Aeron has not joined #announcements'
>>> kenghwee.message(announcements, "Tutorial is canceled!")
'Keng Hwee has not joined #announcements'
>>> russell.message(secret, "I am alone!")
'#foobar --- Russell: I am alone!'
>>> clifton.message(secret, "WHAT")
'Clifton has not joined #foobar'
```

```
# Everything changes when the Mute Nation attacked...
>>> russell.mute(kenghwee, None, "Testing")
'Russell muted Keng Hwee indefinitely. Reason: Testing'
>>> clifton.mute(russell, 10, "Revenge")
'Cannot mute a fellow tutor'
>>> kenghwee.mute(russell, 100, "Revenge")
'Keng Hwee is not allowed to send messages!' # because he's muted
>>> kenghwee.message(announcements, "Tutorial is canceled!")
'Keng Hwee has not joined #announcements'
>>> clifton.unmute(kenghwee)
'Clifton unmuted Keng Hwee!'
>>> kenghwee.mute(russell, 100, "Revenge")
"Keng Hwee doesn't have a permission to mute another user"
>>> aeron.mute(clifton, 3, None)
"Aeron doesn't have a permission to mute another user"
>>> clifton.mute(aeron, 5, "Why would you try to mute me?")
'Clifton muted Aeron for 5 minutes. Reason: Why would you try to mute me?'
>>> kenghwee.mute(kenghwee, 10, "No idea")
'Cannot mute oneself'
>>> russell.mute(russell, 10, "Same here")
'Cannot mute oneself'
>>> russell.mute(aeron, 3, "Spam")
'Aeron is muted!'
>>> display_hall_of_mute()
'''
HALL OF MUTE
#general-helpdesk: ['Aeron']
#announcements: []
#foobar: []
'''

>>> aeron.message(secret, "Hello")
'Aeron has not joined #foobar'
>>> aeron.message(general, "Yoooo")
'Aeron is not allowed to send messages in #general-helpdesk'
>>> aeron.message(announcements, "Test")
'Aeron has not joined #announcements'
>>> clifton.mute(russell, None, None)
'Cannot mute a fellow tutor'
>>> russell.mute(clifton, None, "Muting a fellow tutor is a can")
'Russell muted Clifton indefinitely. Reason: Muting a fellow tutor is a can'
>>> display_hall_of_mute()
'''
HALL OF MUTE
#general-helpdesk: ['Aeron', 'Clifton']
#announcements: ['Clifton']
#foobar: []
'''
```

```
>>> clifton.mute(aeron, 3, "Spam?")
'Clifton is not allowed to send messages!'
>>> russell.unmute(clifton)
'Russell unmuted Clifton!'
>>> display_hall_of_mute()
'''
HALL OF MUTE
#general-helpdesk: ['Aeron']
#announcements: []
#foobar: []
'''

>>> clifton.mute(aeron, 3, "Spam?")
'Aeron is muted!'
>>> aeron.message(general, "Yoooo")
'Aeron is not allowed to send messages in #general-helpdesk'
# since he's muted
>>> aeron.mute(kenghwee, 2, "Lol")
'Aeron is not allowed to send messages!'
# again, because he's still muted
>>> aeron.unmute(kenghwee)
'Keng Hwee is not muted :)'
>>> clifton.unmute(aeron)
'Clifton unmuted Aeron!'
>>> display_hall_of_mute()
'''
HALL OF MUTE
#general-helpdesk: []
#announcements: []
#foobar: []
'''

>>> kenghwee.mute(russell, 10, None)
"Keng Hwee doesn't have a permission to mute another user"
>>> kenghwee.message(general, "Hi guys I'm unmuted")
'Keng Hwee has not joined #general-helpdesk'
>>> russell.message(general, "Hello")
'#general-helpdesk --- Russell: Hello'
>>> clifton.message(general, "Hello!")
'#general-helpdesk --- Clifton: Hello!'
>>> aeron.message(general, "I'm so happy!")
"#general-helpdesk --- Aeron: I'm so happy!"
>>> aeron.message(announcements, "Test")
'Aeron has not joined #announcements'
>>> kenghwee.join(general)
'Keng Hwee has no permission to join #general-helpdesk'
>>> russell.unmute(kenghwee)
'Keng Hwee is not muted :)'
>>> russell.unmute(russell)
'Russell is not muted :)'
```

Solution:
Here's a possible implementation. Feel free to think of an alternative implemenation.

Disclaimer:
By right, any use or query of the Channel class' **`name`** and **`permitted_roles`** attribute inside functions other than the constructor method and the respective getter methods should be replaced by the **`get_name()`** and **`get_permitted_roles()`** method themselves due to abstraction.
For example, **`channel.permitted_roles`** should be replaced by **`channel.get_permitted_roles()`**.

I will replace them by directly accessing the name attribute just because of space constraints, but you shouldn't during the actual PE or finals :)

Likewise, if you define another getter functions that returns the attribute as is, then you should do the same as the **`get_name()`** and and **`get_permitted_roles()`** method. Otherwise, you will be considered breaking abstraction.

```
class Channel:
    def __init__(self, name, roles):
        self.name = f"#{name}"
        self.permitted_roles = roles
        self.members = []

    def get_name(self):
        return self.name

    def get_permitted_roles(self):
        return self.permitted_roles

    def get_members(self):
        return sorted(map(lambda x: x.name, self.members))

    def hall_of_mute(self):
        return sorted(map(lambda x: x.name,
               filter(lambda x: x.muted, self.members)))

class User: # @everyone
    def __init__(self, name):
        self.name = name # for now, we don't use Discord tags
        self.role = None
        self.channels = []
        self.muted = False

    def get_role(self):
        if self.role:
            return f"{self.name} is a {self.role}"
        else:
            return f"{self.name} has no role"

    def get_channels(self):
        return sorted(map(lambda x: x.name, self.channels))
```

```
    def join(self, channel):
        if channel in self.channels:
            return f"{self.name} has already joined {channel.name}"
        elif self.muted:
            return f"{self.name} is muted!"
        else:
            if isinstance(self, Owner) or self.role in channel.permitted_roles:
                channel.members.append(self)
                self.channels.append(channel)
                return f"{self.name} joins {channel.name}"
            else:
                return f"{self.name} has no permission to join {channel.name}"

    def message(self, channel, msg):
        if channel not in self.channels:
            return f"{self.name} has not joined {channel.name}"
        elif self.muted:
            return f"{self.name} is not allowed to send messages " + \
                   f"in {channel.name}"
        else:
            return f"{channel.name} --- {self.name}: {msg}"

    def mute(self, other, time, reason):
        if other == self:
            return "Cannot mute oneself"
        elif self.muted:
            return f"{self.name} is not allowed to send messages!"
        elif other.muted:
            return f"{other.name} is muted!"
        elif isinstance(self, Owner):
            other.muted = True
            if time == None:
                return f"{self.name} muted {other.name} indefinitely. " + \
                       f"Reason: {reason}"
            return f"{self.name} muted {other.name} for {time} minutes. " + \
                   f"Reason: {reason}"
        elif isinstance(self, Tutor):
            if isinstance(other, Tutor):
                return "Cannot mute a fellow tutor"
            else:
                other.muted = True
                if time == None:
                    return f"{self.name} muted {other.name} indefinitely." + \
                           f" Reason: {reason}"
                return f"{self.name} muted {other.name} for {time} " + \
                       f"minutes. Reason: {reason}"
        else:
            return f"{self.name} doesn't have a permission to mute " + \
                   "another user"
```

```
    def unmute(self, other):
        if other == self and self.muted:
            return f"{self.name} is not allowed to send messages!"
        elif not other.muted:
            return f"{other.name} is not muted :)"
        elif isinstance(self, Owner):
            other.muted = False
            return f"{self.name} unmuted {other.name}!"
        elif isinstance(self, Tutor):
            if isinstance(other, Tutor):
                return "Cannot unmute a fellow tutor"
            else:
                other.muted = False
                return f"{self.name} unmuted {other.name}!"
        else:
            return f"{self.name} doesn't have a permission " + \
                   "to unmute another user"

class Tutor(User):
    def __init__(self, name):
        super().__init__(name)
        self.role = "Tutor"

class Student(User):
    def __init__(self, name):
        super().__init__(name)
        self.role = "Student"

class Owner(Tutor):
    def __init__(self, name):
        super().__init__(name)
        self.role = ["Owner", "Tutor", "Student"]

    def get_role(self):
        return f"{self.name} is the owner!"
```

Solution compiled by Russell Saerang.