Homework due Aug 6, 2022 19:07 EDT

Problem Description

Hello! Please make sure to read all parts of this document carefully.

You and your group of astronaut friends are on a spaceship! Fun! However, you are not completely sure you can trust some of them... kinda sus... There seems to be a few Impostors onboard, trying to sabotage you and your crew. Your job is to analyze the players and find the impostors before it is too late! The crewmates win if all of the impostors are frozen, and the impostors win if they hold a majority of the unfrozen players (greater than or equal to half). In this assignment, you will be given a partially filled `Player.java` (see below for more details), and will create `Impostor.java`, `Crewmate.java`, `RedAstronaut.java`, and `BlueAstronaut.java`. These classes will simulate the gameplay. In addition, you will create a `Gameplay.java` file that will not be turned in, serving as a driver that can be used to test your code and check its output. To complete this assignment, you will use your knowledge of abstract classes and interfaces.

Solution Description

You will create two interfaces, one abstract class, and two concrete subclasses. You will be creating a number of fields and methods for each file. Based on the description given for each variable and method, you will have to decide whether or not the variables/method should be static, and whether it should be private or public. To make these decisions, you should carefully follow the guidelines on these keywords as taught in the modules.

Hint: A lot of the code you will write for this assignment can be reused. Try to think of what keywords you can use that will help you! You should be able to put @Override on the line before the method header for any methods you override.

`Impostor.java`

This file defines an **interface** with the name Impostor. Impostors will be able to mess with Players through sabotage and freeze tagging them.

Methods

- `freeze(Player p)`
  - Abstract method that takes in a Player object and does not return anything
  - (Note: any class that implements Impostor must provide a method definition for this method)
- `sabotage(Player p)`
  - Abstract method that takes in a Player object and does not return anything

- (Note: any class that implements Impostor must provide a method definition for this method)

`Crewmate.java`

This file defines an **interface** with the name Crewmate.

Methods

- `completeTask()`
  - Abstract method that does not take in anything and does not return anything
  - (Note: any class that implements Crewmate must provide a method definition for this method)

`Player.java`

This class represents a player in the game. You must not be able to create an instance of this class (**Hint:** there is a keyword that prevents us from creating instances of a class). **Players implement the Comparable interface with the proper type parameter.**

A portion of this class was given to you. There are 2 methods (as follows) that you need to implement. Make sure to code where there are comments that say YOUR CODE HERE. Javadocs have been provided to guide you.

Variables (These have been provided to you)

- `String name` – The Player's name as a String.
- `int susLevel` – This represents how suspicious a Player is; the higher, the more suspicious.
- `boolean frozen` – This represents if a Player is frozen, which will affect their ability to participate in some methods below.
- `static Player[] players` – This is the array of Player objects that have been created. It is always guaranteed to be **full** by the constructor (no null spaces), so if a Player is created, players becomes larger by one. Frozen players will NOT be removed from this array! (Note: It's ok move around elements in this array, but don't add or delete any elements)

Methods

Do not create any other methods than those specified. Any extra methods will result in point deductions. All methods must have the proper visibility to be used where it is specified they are used.

- `emergencyMeeting()`
  - Any concrete class that extends the Player class must provide a method definition for emergencyMeeting()
  - An abstract method for a Player to call an emergency meeting to vote on which Player to freeze
  - Does not return anything

- `compareTo(Player p)`
  - Override the compareTo method (You should be able to put @Override on the line before the method header)
  - Takes in a Player object and returns an int, adhering to the API contract (Comparable Interface)
  - The method body should compare two Player objects based on the `susLevel` attribute. If the current Player instance's `susLevel` is less than the Player passed in, return a negative number. If it is greater, return a positive number. If their `susLevel` attributes are equal, return 0

`RedAstronaut.java`

This file defines a RedAstronaut, **which is a Player** and should have all attributes of one. **Have RedAstronaut implement the Impostor interface.**

Variables

All variables must be not allowed to be directly modified outside the class in which they are declared, unless otherwise stated in the description of the variable. **Hint:** there is a specific visibility modifier that can do this!

The Red class must have these variables. Do **NOT** re-declare any of the instance variables declared in Player class:

- `skill` - a String that represents skill of the Red crewmate a String value of either inexperienced, experienced, or expert.

Constructors

- A constructor that takes in the `name`, `susLevel`, and `skill` and sets all fields accordingly. It must accept the variables in the specified order. Assume that the passed in parameter for skill will be one of the three values, although it may have different capitalization.
  - Hint: There is a specified keyword in L12 to access the superclass's constructor.
- A constructor that takes in just a `name` and assigns the following default values:
  - `susLevel: 15`
  - `skill: experienced`

Methods

Do not create any other methods than those specified. Any extra methods will result in point deductions. All methods must have the proper visibility to be used where it is specified they are used.

- `emergencyMeeting()`
  - **A Player that is frozen cannot call an emergency meeting.**
  - Holds a meeting and votes out (freezes) the most suspicious Player, only considering Players that

are not frozen

- The player that has the highest `susLevel` (**that is NOT the current impostor calling the meeting**) will be accused of being the impostor and will be voted off
- If two players have the same highest `susLevel`, no player will be voted off.
- **Hint:** think of an easy way to do this without having to iterate through the entire array. Check the Java API for Arrays for a method you can use

- Make sure to change the `frozen` variable of the player to true when voting off players (don't call the freeze method!)
- At the end of the vote, check if the game is over using the provided method in Player.java
- Does not return anything

- `freeze(Player p)`
  - Implements the method provided in the Impostor interface.
  - It is not possible to freeze another Impostor, and an Impostor that is frozen cannot attempt to freeze. If the passed in Player is an Impostor, the method should end. Freezing an already frozen Player should also do nothing.
  - A freeze is successful if the RedAstronaut's `susLevel` is less than the Player's
  - If the freeze is unsuccessful, the RedAstronaut's `susLevel` doubles (multiply the current `susLevel` by 2)
  - Remember to change the `frozen` boolean value for the Crewmate as needed.
  - After the freeze attempt, check if the game is over using the provided method in Player.java
  - Does not return anything
- `sabotage(Player p)`
  - It is not possible to sabotage another Impostor, and an Impostor that is frozen cannot sabotage. Also, sabotaging a frozen Player should do nothing.
  - If the Impostor's `susLevel` is under 20, through shifty maneuvers and cunning words, they are able to increase the Crewmate's `susLevel` by 50%
  - Otherwise, they can only manage to increase the Crewmate's `susLevel` by 25%
  - (Note: In both cases, the the Crewmate's `susLevel` is rounded down to the nearest int value)
  - Does not return anything
- `equals(Object o)`
  - Two Red are equal if they both have the same `name`, `frozen`, `susLevel`, and `skill`
  - Returns a boolean

- `toString()` - returns a String describing RedAstronaut as follows:
  (Note: replace the values in brackets [] with the actual value)
  - `"My name is [name], and I have a suslevel of [susLevel]. I am currently (frozen / not frozen). I am an [skill] player!”`
    - If `susLevel` is greater than 15, return the String in all capital letters.
  - You must use the toString() method from the Player class to receive full credit.
- Getters and Setters as necessary.

`BlueAstronaut.java`

This file defines a BlueAstronaut, **which is a Player** and should have all attributes of one. **Have BlueAstronaut implement the Crewmate interface.**

Variables

All variables must be not allowed to be directly modified outside the class in which they are declared, unless otherwise stated in the description of the variable. **Hint:** there is a specific visibility modifier that can do this!

The `Blue` class must have these variables. Do **NOT** redeclare any instance variables created in the Player class

- `numTasks` - the number of tasks that needs to be completed as an integer number
- `taskSpeed` - the speed at which the astronaut is completing each task as a positive, nonzero integer number

Constructors

- A constructor that takes in the `name`, `susLevel`, `numTasks`, and `taskSpeed` and sets all fields accordingly. It must accept the variables in the specified order.
- A constructor that takes in just a `name` and assigns the following default values:
  - `susLevel: 15`
  - `numTasks: 6`
  - `taskSpeed: 10`

Methods

Do not create any other methods than those specified. Any extra methods will result in point deductions. All methods must have the proper visibility to be used where it is specified they are used.

- `emergencyMeeting()`
  - A Player that is frozen cannot call an emergency meeting.

- Holds a meeting and votes out (freezes) the most suspicious individual of the Player objects, only considering Players that are not frozen
  - The player that has the highest `susLevel` will be accused of being the impostor and will be voted off (This could be them!)
  - If two players have the same highest `susLevel`, no player will be voted off.
  - **Hint:** think of an easy way to do this without having to iterate through the entire array. Check the Java API for Arrays for a method you can use.
- Make sure to change the `frozen` variable of the player to true when voting off players (don't call freeze!)
- At the end of the vote, check if the game is over using the provided method in Player.java
- Does not return anything

- `completeTask()`
  - A BlueAstronaut that is frozen cannot complete tasks.
  - If `taskSpeed` is greater than 20, subtract 2 from `numTasks`. Otherwise, subtract 1 from `numTasks`.
    - If `numTasks` falls below 0, set it to 0
  - After BlueAstronaut is done with their tasks, meaning `numTasks` is equal to 0 (only for the first time),
    - Print out "`I have completed all my tasks`"
    - Then reduce BlueAstronaut's `susLevel` by 50% (round down)
  - Does not return anything.

- `equals(Object o)`
  - Two BlueAstronauts are equal if they both have the same `name`, `frozen`, `susLevel`, `numTasks`, and `taskSpeed`
  - Returns a boolean

- `toString()` - returns a String describing BlueAstronaut as follows:
  - "`My name is [name], and I have a suslevel of [susLevel]. I am currently (frozen / not frozen). I have [numTasks] left over.`"
    - If `susLevel` is greater than 15, return the String in all capital letters.
  - (Note: replace the values in brackets [] with the actual value)
  - You must use the toString() method from the Player class to receive full credit.

- Getters and Setters as necessary.

`Gameplay.java`

This Java file is a driver, meaning it will run the simulation. You can also use it to test your code. Here are some basic tests to get you started with Amidst Us. These tests are my no means comprehensive, so be sure to create your own!

This is just to show an example of object implementation to see how objects interact with each other. Feel free to play around with different values and method callings, especially using toString to check the values after every step! This will NOT be turned in.

Create a BlueAstronaut with the following fields:

- `name` = “Bob”, `susLevel` = 20, `numTasks` = 6, `taskSpeed` = 30

Create a BlueAstronaut with the following fields:

- `name` = “Heath”, `susLevel` = 30, `numTasks` = 3, `taskSpeed` = 21

Create a BlueAstronaut with the following fields:

- `name` = “Albert”, `susLevel` = 44, `numTasks` = 2, `taskSpeed` = 0

Create a BlueAstronaut with the following fields:

- `name` = “Angel”, `susLevel` = 0, `numTasks` = 1, `taskSpeed` = 0

Create a RedAstronaut with the following fields:

- `name` = “Liam”, `susLevel` = 19, `skill` = "experienced"

Create a RedAstronaut with the following fields:

- `name` = “Suspicious Person”, `susLevel` = 100, `skill` = "expert"

Have the objects do the following:

1. Have RedAstronaut Liam sabotage BlueAstronaut Bob. After the sabotage:
   - Bob should have: `susLevel` = 30, `frozen` = false
2. Have RedAstronaut Liam freeze RedAstronaut Suspicious Person:
   - Nothing should happen

3. Have RedAstronaut Liam freeze BlueAstronaut Albert. After the freeze:
    - Liam should have: `susLevel` = 19
    - Albert is now frozen
4. Have BlueAstronaut Albert call an emergency meeting:
    - Nothing should happen since he is frozen
5. Have RedAstronaut Suspicious Person call an emergency meeting:
    - This will result in a tie between Bob and Heath, so nothing should happen
6. Have BlueAstronaut Bob call an emergency meeting:
    - Suspicious Person should have: `frozen` = true
7. Have BlueAstronaut Heath complete tasks:
    - Heath should have: `numTasks` = 1
8. Have BlueAstronaut Heath complete tasks:
    - "`I have completed all my tasks`" should be printed to console
    - Heath should have: `numTasks` = 0, `susLevel` = 15
9. Have BlueAstronaut Heath complete tasks:
    - Nothing should happen
10. Have RedAstronaut Liam freeze Angel:
    - Angel should have: `frozen` = false
    - Liam should have: `susLevel` = 38
11. Have RedAstronaut Liam sabotage Bob twice:
    - Bob should have: `susLevel` = 46 (30 -> 37 -> 46)
12. Have RedAstronaut Liam freeze Bob:
    - Bob should have: `frozen` = true

Now there are two options going forward

13. Have BlueAstronaut Angel call emergency meeting:
    - Liam should have: `frozen` = true
    - "`Crewmates win!`" should be printed to console

Or

14. Have RedAstronaut Liam call sabotage on Heath 5 times:
    - Heath should have: `susLevel` = 41 (15->18->22->27->33->41)
15. Have RedAstronaut Liam freeze Heath:
    - Heath should have: `frozen` = true
    - `"Impostors win!"` should be printed to console

Reuse your code when possible. Certain methods can be reused using certain keywords.

Allowed Imports

To prevent trivialization of the assignment, you are only allowed to import `java.util.Arrays`. You are *not* allowed to import any other classes or packages.

Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or break our auto grader. For that reason, do not use any of the following in your final submission:

- `var` (the reserved keyword)
- `System.exit`

Grading

Homeworks are graded in an "all or nothing" manner. If your code is correct, you receive a 100 for the assignment; if it isn't, you receive a 0.

Allowed Collaboration

When completing homeworks for CS1331 you may talk with other students about:

- What general strategies or algorithms you used to solve problems in the homeworks
- Parts of the homework you are unsure of and need more explanation
- Online resources that helped you find a solution
- Key course concepts and Java language features used in your solution

You may **not** discuss, show, or share by other means the specifics of your code, including screenshots, file sharing, or showing someone else the code on your computer, or use code shared by others.

The Vocareum (code editor) interface has six main components:

- The **Drop-Down** in the top left. This lets you choose from multiple available files. Note that this drop-down will only be visible in assignments that require multiple files.
- The **Build / Run** button. For all assignments in this course, the build and run button will perform the same action: compile your code and run a file scan. Building and running your code will not count towards your total allowed submission attempts, therefore you are free to build / run as many times as needed.
- The **Submit** button. This will compile your code, run a file scan, grade your assignment, and output results to console. Note that for most assignments in this class, you will only be allowed a limited number of submissions. A submission is counted when the submit button is clicked, regardless of whether or not your code is able to compile or if there are any file issues. Therefore, **we highly recommend that you build or run your code before submitting to ensure that there are no issues that will prevent your code from being graded and that every submission attempt will generate meaningful results**.
- The **Reset** button. This will revert all your changes and reset your code to the default code template.
- The **Code Window**. This is where you will write your code. Again, We highly recommend copying the starter code and working in your preferred IDE.
- The **Output Window**. This window will appear whenever you build, run, or submit your code and will display the results for you to view.

Vocareum Troubleshooting

We acknowledge that the Vocareum integration has some issues when submitting programs with multiple files. That is, the Vocareum interface may hide both the current file name and the combo box you need to select a file to edit. Unfortunately, this is beyond our control. Recall that the instructor recommends that you code and debug on your local JVM and submit in Vocareum mainly for grading. However, if you do need to edit in Vocareum, we have found the following workaround that you might try if you experience the stated problem.

1. Close any other homework from this course or another that uses the Vocareum integration.
2. Reload the webpage.
3. Open Vocareum in a new window. Keep the edX homework open: Vocareum won't visualize the

homework description with the same formatting.

4. Select the course CS1331 and follow the instructions to start or continue working on the homework.
5. Look for where these four checkboxes appear: Files, README, Terminal, Source. Then, click the checkbox "Files." The README checkbox should uncheck automatically; if it doesn't, uncheck it.
6. You can now edit the files in the folder "work." Do not edit any files the extension "class" in the folder. Do not edit any file in any other folder or subfolder (such as "resource," "Submissions," or "workspace"). Do not create any additional files or folders.

For additional help, please visit the Vocareum information page located in the course information module!

For learners unable to access the Vocareum environment, this is the provided Player.java file:

Player.java