Creative Project 1: Survivor Challenge

Specification

Thanks to Miya Natsuhara and Sam Wolfson for help with inspiration and brainstorming for this assignment.

Background

The popular TV show *Survivor*, like many other competition or "reality" shows, often puts its contestants through challenges consisting of a series of tasks that must be completed to earn a prize or reward. The challenges can take many different forms and require a wide range of skills. In this assignment, you will implement several classes to represent different types of tasks for a simulation of these types of challenges.

Learning Objectives

By completing this assignment, students will demonstrate their ability to:

- Define relationships between Java classes using inheritance, abstract classes, and references
- Write a Java class that extends a given abstract class
- Produce clear and effective documentation to improve comprehension and maintainability of classes
- Write classes that are readable and maintainable, and that conform to provided guidelines for style, and implementation

Simulation Structure

In our simulation, a challenge will be represented by a list of tasks as defined by the provided `Task` class. You will be responsible for implementing several specific types of tasks, as follows:

- **`EnduranceTask`** - tasks that test endurance by requiring contestants to repeat a single action many times in a row
  - e.g. jumping over a series of hurdles, hanging from a wall for an extended period of time, swimming a number of laps in a pool or lake
- **`PrecisionTask`** - tasks that test skill and accuracy by requiring contestants to take a particular set of actions in a particular order
  - e.g. completing an obstacle course, pushing a series of buttons in a particular order

- `PuzzleTask` - tasks that test intelligence or memory by requiring constants to solve a puzzle or riddle
    - e.g. solving a math problem
- ***One additional task type of your choice that extends one of the above task types***
    - e.g. a strength task, a speed task, a memory task, or any other task you can think of!

Each type of task will be represented by a single class, which should (directly or indirectly) extend the `Task` class specified in `Task.java`. See the source file for the required methods.

Sample Simulation

Here is an example of how the simulation might run when the required classes are fully implemented. This output was produced using the provided `Client.java` and a reference solution for all subclasses of the `Task` class. You are not required to exactly match this output format and text, but you *must not modify* `Client.java` (see below). In the below example, user input is **bold and underlined:**

```
Current task: A set of three hurdles.
Previous actions: []
Possible actions: [jump, run, swim, crawl, climb]
Your action? jump
  Action successful!

Current task: A set of three hurdles.
Previous actions: [jump]
Possible actions: [jump, run, swim, crawl, climb]
Your action? jump
  Action successful!

Current task: A set of three hurdles.
Previous actions: [jump, jump]
Possible actions: [jump, run, swim, crawl, climb]
Your action? swim
  Action not successful.

Current task: A set of three hurdles.
Previous actions: [jump, jump]
Possible actions: [jump, run, swim, crawl, climb]
Your action? jump
  Action successful!
  Task completed!

Current task: A small lake.
Previous actions: []
Possible actions: [jump, run, swim, crawl, climb]
Your action? fly
  **Invalid action: fly**

Current task: A small lake.
```

```
Previous actions: []
Possible actions: [jump, run, swim, crawl, climb]
Your action? run
  Action not successful.

Current task: A small lake.
Previous actions: []
Possible actions: [jump, run, swim, crawl, climb]
Your action? swim
  Action successful!
  Task completed!

Current task: A low crawl net, then a wall with a rope, then a dash to the end.
Previous actions: []
Possible actions: [jump, run, swim, crawl, climb]
Your action? crawl
  Action successful!

Current task: A low crawl net, then a wall with a rope, then a dash to the end.
Previous actions: [crawl]
Possible actions: [jump, run, swim, crawl, climb]
Your action? crawl
  Action not successful.

Current task: A low crawl net, then a wall with a rope, then a dash to the end.
Previous actions: [crawl]
Possible actions: [jump, run, swim, crawl, climb]
Your action? climb
  Action successful!

Current task: A low crawl net, then a wall with a rope, then a dash to the end.
Previous actions: [climb, crawl]
Possible actions: [jump, run, swim, crawl, climb]
Your action? run
  Action successful!
  Task completed!

Current task: What is 2 + 2?
Previous actions: []
Possible actions: [hint, solve <solution>]
Your action? hint
  Action successful!

Current task: What is 2 + 2?
  HINT: It's 4.
Previous actions: [hint]
Possible actions: [hint, solve <solution>]
Your action? solve 22
  Action not successful.

Current task: What is 2 + 2?
  HINT: It's 4.
Previous actions: [hint]
Possible actions: [hint, solve <solution>]
```

```
Your action? solve 4
  Action successful!
  Task completed!

Challenge complete. Congratulations!!!
```

Implementation Requirements

Make sure to read the specification very carefully and thoroughly! It includes important cases to take into mind and account for!

Each type of task should be represented by a class that extends the `Task` class. You should **not** modify `Task`. You should utilize inheritance to capture common behavior among similar task types and eliminate as much redundancy between classes as possible.

Your classes should be implemented so that the client program in `Client.java` works as written. In particular, your implementations must not rely on any public methods beyond those specified in the `Task` interface. (You are welcome to add additional public or private helper methods, but your classes must be able to be utilized by the client without calling these methods directly.)

Required Constructors

Each of the required classes should have the following constructors (you may include additional constructors if you wish):

```
public EnduranceTask(String type, int duration, String description)
```

- `String type` - the type of task, which is also the action required to complete the task
  - This must be one of the valid actions for this task type
- `int duration` - the number of times the action must be taken to complete the task
- `String description` - a text description of the task

```
public PrecisionTask(List<String> requiredActions, String description)
```

- `List<String> requiredActions` - the sequence of actions that are required to complete the task
  - Each action must be one of the valid actions for this task type
  - Actions need to be completed *in order* to complete the task
- `String description` - a text description of the task

```
public PuzzleTask(String solution, List<String> hints, String description)
```

- `String solution` - the expected solution for this task
  - The solution will be provided with the "solve" action

- `List<String> hints` - an ordered list of hints to be provided to the client when taking the "hint" action
    - May be empty if there are no hints
    - If all hints have already been given (including if there are no hints), `takeAction()` should return `false` for the `hint` action.
- `String description` - a text description of the task
    - Includes the most recent hint given.

Finally, all implementations of a task **must throw** an `IllegalArgumentException` in `takeAction()` in the event of an invalid action.

Your classes **must not directly produce any console output**-- all output must be produced by the client program.

HINT 1: You may override the `getDescription()` to reflect changes in the task's state if you would like, but you should do so sparingly.
HINT 2: Recall that subclasses can't directly access or change their superclass's fields. The subclasses, however, *do* have access to their superclass's methods, which then *can* directly access or change the superclass's fields.

You may add one or more lines of code after line 11 of `Client.java` to create additional tasks (**including at least one instance of your custom task type**), and you may create any additional variables or data to pass to constructors as parameters. But you should not otherwise modify the client, and in particular, you should not modify the `attemptChallenge` method. Implement your classes so that the client works **as written**. You should also **not** modify the `Task.java` file.

Assignment Requirements

For this assignment, you should follow the Code Quality guide when writing your code to ensure it is readable and maintainable. In particular, you should focus on the following requirements:

- You should make all of your fields private and you should reduce the number of fields only to those that are necessary for solving the problem.
- Each of your fields should be initialized inside of your constructor(s).
- You should comment your code following the Commenting Guide. You should write comments with basic info (a header comment at the top of your file), a class comment for every class, and a comment for every method other than main.
    - Make sure to avoid including *implementation details* in your comments. In particular, for your object class, a *client* should be able to understand how to use your object effectively by only reading your class and method comments, but your comments should maintain *abstraction* by avoiding implementation details.

Survivor Challenge

This code slide does not have a description.

Reflection

The following questions will ask you practice **metacognition** to reflect on the topics covered on this assignment and your experience completing it. For each question, focus on your plan and/or process for working through the assignment along with the CS concepts. Think about things like how you organized your working time, what sorts of things tended to go wrong, and how you dealt with those errors or mistakes.

Please answer all questions.

Question 1

Describe the custom task type you implemented. What sort of task does it represent? What class did you decide it should extend? Why did you choose to extend that class?

No response

Question 2

Describe how you went about testing your implementation. What specific situations and/or test cases did you consider? Why were those cases important?

No response

Question 3

What skills did you learn and/or practice with working on this assignment?

No response

Question 4

What did you struggle with most on this assignment?

No response

Question 5

What questions do you still have about the concepts and skills you used in this assignment?

No response

Question 6

About how long (in hours) did you spend on this assignment? (Feel free to estimate, but try to be close.)

No response

Question 7

Was any part of the specification or requirements unclear? If so, which part(s), how was it unclear, and how could it have been made more clear?

No response

Question 8

[OPTIONAL] Do you have any other feedback, questions, or comments about this assignment?

(Note that we may not be able to respond to questions here, so please post on the message board if you would like a response!)

No response

Testing Comprehension

Question 1

Consider the following method spec:

```
//pre: x represents a number (i.e. not NaN)
//post: returns the square root of x, or ComplexNumberException if x is negative
public double squareRoot(double x)
```

Which of the following would be appropriate tests for `squareRoot`? Assume that no weird floating point stuff will happen (everything will be perfectly mathy)

- [ ] Testing that `squareRoot(4.0)` returns 2.0
- [ ] Testing that `squareRoot(0.0)` returns 0.0
- [ ] Testing that `squareRoot(-1.0)` returns `i`
- [ ] Testing that `squareRoot(-1.0)` returns 1.0
- [ ] Testing that `squareRoot(Double.NaN)` throws an IllegalArgumentException
- [ ] Testing that `squareRoot(-1.0)` throws an `IllegalArgumentException`
- [ ] Testing that `squareRoot(-1.0)` throws an `ComplexNumberException`

Question 2

What does each part of `assertEquals` and `assertTrue` represent?

```
assertTrue(a, b)
assertEquals(x, y, z)
```

Order the items as:

a

b

x

y

z

The tested value

The error message, if x does not equal y

The expected value

The error message, if a is not true

The tested boolean value

Question 3

Consider the following unit test for the method `squareRoot`. Arrange where each item should go in alphabetical order (ex. the first item in order will go where `A` is, etc.)

```
@Test
@DisplayName("squareRoot Correct")
public void testSquareRoot() {
    assertEquals(4.0, A, B);
    assertEquals(C, squareRoot(0.0), D);
    assertThrows(C, () -> squareRoot(-1.0), "Does not throw correct exception for negative input!)
}
```

"Does not return correct answer for positive number"

ComplexNumberException.class

0.0

"Does not return correct answer for zero"

squareRoot(16)

Question 4

Consider the following unit test for `PuzzleTask` in Survivor Challenge. Arrange where each item should go in alphabetical order (ex. the first item in order will go where `A` is, etc.)

```
@Test
@DisplayName("Puzzle Answer Correct")
public void testPuzzleAnswer() {
    Task t = new PuzzleTask("answer", new ArrayList<String>(), "test puzzle");
    assertTrue(A, "isComplete returned true incorrectly");
    assertTrue(B, "Solving task did not function as expected.");
    assertTrue(C, D);
}
```

Fun Fact: this is one of the actual tests that runs when you hit "Mark"!

t.isComplete()

"Task is not considered complete correctly."

t.takeAction("solve answer")

!t.isComplete()

□ Final Submission □

□ Final Submission□

Fill out the box below and click "Submit" in the upper-right corner of the window to submit your work.

Question

I attest that the work I am about to submit is my own and was completed according to the course Academic Honesty and Collaboration policy. If I collaborated with any other students or utilized any outside resources, they are allowed and have been properly cited. If I have any concerns about this policy, I will reach out to the course staff to discuss *before* submitting.

(Type "yes" as your response.)

No response