CodeCrunch

CS2030/S PE2 In-Lab Submissions (Hidden)

Tags & Categories

Tags:

Categories:

Related Tutorials

Task Content

Zork!!

This PE is worth 20% of your grade. You must work on it alone, without discussing with anyone (whether online or offline).

Problem Description

If you think that a computer game requires stunning graphics, or least, cute ones, to be captivating, then you haven't played Zork. Created in the late 1970s, before sound cards or graphics cards were invented, Zork was a text-only adventure game famed for its storytelling and advanced text parser. In it, you play the role of an explorer plonked in front of a house, surrounded by a forest, with no hint of what your goal might be: whether to rescue an imprisoned princess, or to find hidden treasure.

You interact by typing commands, such as "GO NORTH", or "HIT THE TROLL WITH THE ELVISH SWORD", and the game responds with textual descriptions, such as "Phosphorescent mosses, fed by a trickle of water from some unseen source above, make the crystal grotto glow and sparkle with every color of the rainbow". These words spark one's imagination of mystery, fear, and adventure in ways that graphics and sound cannot.

Its commercial launch was a great success, spawning a trilogy (because otherwise the entire game could not fit into computer memory) that in total sold over 800,000 copies, and keeping it among the "Top 50 Games of All Time" even twenty years later. Zork foreshadowed, and influenced, the development of modern AI chatbots.

In this Practical Assessment (PA), you will create the basic game elements in Zork. While keeping it simple to fit the needs of a PA, there is great scope for future development into a much richer project for educational purpose (*read: talk to the Profs if you are interested*).

Task

In this task, you are only required to work with three things: a candle, a sword and a troll. These things are placed at possibly different rooms. The objective is to simulate and test different scenarios of game play. Take note of the following:

- all Java objects constructed are to be *immutable*;
- for simplicity, at no time will there be multiple things of the same type

This task is divided into several levels. You need to complete all levels. Read through all the levels to see how the different levels are related.

Remember to:

- write each class/interface/enum in a separate `.java` file
- always compile your program files first before using `jshell` to test your program
- declare object properties starting with `private final` or `protected final`
- you may import any Java package; however do not use the wild card `*` in your import statements, else CodeCrunch will render your program uncompilable
- all tests use valid arguments; you need not check for validity of arguments; `null` will not be used for the tests

Level 1

Write a class `Room` to represent a room in the game. Things can be placed in a room. In this level, we shall place a `Candle` in the foyer. To simulate the passing of time, a method `tick()` in class `Room` is called.

With respect to the candle, every tick causes the candle to change state.The following are the state changes of the candle.

- `Candle flickers.`
- `Candle is getting shorter.`
- `Candle is about to burn out.`
- `Candle has burned out.`

Note that any `ticks` beyond the last state of the candle causes it to remain in it's final state. Also, make sure that `Room` does not sub-class from any other class (apart from `Object` of course).

```
$ javac your_java_files
$ jshell -q your_java_files_in_bottom-up_dependency_order < test1.jsh
```

```
jshell> new Room("foyer");
$.. ==> @foyer
jshell> new Room("foyer").add(new Candle());
$.. ==> @foyer
Candle flickers.
jshell> new Room("foyer").add(new Candle()).tick()
$.. ==> @foyer
Candle is getting shorter.
jshell> new Room("foyer").add(new Candle()).tick().tick()
$.. ==> @foyer
Candle is about to burn out.
jshell> new Room("foyer").add(new Candle()).tick().tick().tick()
$.. ==> @foyer
Candle has burned out.
jshell> new Room("foyer").add(new Candle()).tick().tick().tick().tick()
$.. ==> @foyer
Candle has burned out.
jshell> /exit
```

Level 2

Let's now include two more things, `Troll` and `Sword`, into our room. Just like the candle, each of these things have their own states:

- For the Troll:
  - `Troll lurks in the shadows.`
  - `Troll is getting hungry.`
  - `Troll is VERY hungry.`
  - `Troll is SUPER HUNGRY and is about to ATTACK!`
  - `Troll attacks!`
- For the Sword:
  - `Sword is shimmering.`

Things are output in order of which they are added into the room. Also note that your program should be flexible enough for other things to be included in future.

```
$ javac your_java_files
$ jshell -q your_java_files_in_bottom-up_dependency_order < test2.jsh
jshell> Room foyer = new Room("foyer").add(new Candle()).add(new Troll())
jshell> Stream.iterate(foyer, x -> x.tick()).limit(6).forEach(System.out::println)
@foyer
Candle flickers.
Troll lurks in the shadows.
@foyer
Candle is getting shorter.
Troll is getting hungry.
@foyer
Candle is about to burn out.
Troll is VERY hungry.
@foyer
Candle has burned out.
Troll is SUPER HUNGRY and is about to ATTACK!
@foyer
Candle has burned out.
Troll attacks!
@foyer
Candle has burned out.
Troll attacks!
jshell> foyer = foyer.add(new Sword())
jshell> Stream.iterate(foyer, x -> x.tick()).limit(3).forEach(System.out::println)
@foyer
Candle flickers.
Troll lurks in the shadows.
Sword is shimmering.
@foyer
Candle is getting shorter.
Troll is getting hungry.
Sword is shimmering.
@foyer
Candle is about to burn out.
Troll is VERY hungry.
Sword is shimmering.
jshell> foyer
foyer ==> @foyer
Candle flickers.
Troll lurks in the shadows.
Sword is shimmering.
jshell> /exit
```

Level 3

Now, we are ready to interact with the objects. An interaction occurs by passing an action into an overloaded `tick` method that takes one argument. The action is in the form of a lambda that describes what actions to take on one or more things.

As an example, we can pass a dummy action `x -> x` (the identity lambda) into `tick`. The test will take the following form:

```
jshell> new Room("foyer").add(new Candle()).add(new Sword()).tick(x -> x)
$.. ==> @foyer
Candle is getting shorter.
Sword is shimmering.
```

As you can see, applying the dummy action above results in the same behaviour as

```
jshell> new Room("foyer").add(new Candle()).add(new Sword()).tick()
$.. ==> @foyer
Candle is getting shorter.
Sword is shimmering.
```

In addition, write the action `takeSword` into the file `actions.jsh`. Note that this will result in three different outputs (see sample run below) according to the state of things in the room. These output (prepended with "`-->`") must be specified within `actions.jsh` and not in the individual classes. You may choose any suitable types for the lambda.

```
$ javac your_java_files
$ jshell -q your_java_files_in_bottom-up_dependency_order actions.jsh < test3.jsh
jshell> new Room("foyer").add(new Candle()).add(new Sword()).tick(x -> x)
$.. ==> @foyer
Candle is getting shorter.
Sword is shimmering.
jshell> new Room("foyer").add(new Candle()).add(new Sword()).tick(x -> x).tick(x -> x)
$.. ==> @foyer
Candle is about to burn out.
Sword is shimmering.
jshell> new Room("foyer").add(new Sword()).tick(takeSword)
--> You have taken sword.
$.. ==> @foyer
Sword is shimmering.
jshell> new Room("foyer").add(new Sword()).tick(takeSword).tick(takeSword)
--> You have taken sword.
--> You already have sword.
$.. ==> @foyer
Sword is shimmering.
jshell> new Room("foyer").tick(takeSword)
--> There is no sword.
$.. ==> @foyer
jshell> new Room("foyer").add(new Candle()).add(new Troll()).add(new Sword()).tick()
$.. ==> @foyer
Candle is getting shorter.
Troll is getting hungry.
Sword is shimmering.
jshell> new Room("foyer").add(new Candle()).add(new Troll()).add(new Sword()).tick(x -> x)
$.. ==> @foyer
Candle is getting shorter.
Troll is getting hungry.
Sword is shimmering.
jshell> new Room("foyer").add(new Candle()).add(new Troll()).add(new Sword()).tick().tick(x -> x)
$.. ==> @foyer
Candle is about to burn out.
Troll is VERY hungry.
Sword is shimmering.
jshell> /exit
```

Level 4

Now add another action `killTroll` into `actions.jsh`. Look at the sample runs below for the behaviour of the action. Once again, the output (prepended with "`-->`") must be specified in `actions.jsh` and not in the respective class files. Also note that a killed troll will vanish from the room.

```
$ javac your_java_files
$ jshell -q your_java_files_in_bottom-up_dependency_order actions.jsh < test4.jsh
jshell> new Room("foyer").add(new Sword()).tick(killTroll)
--> There is no troll
$.. ==> @foyer
Sword is shimmering.
jshell> new Room("foyer").add(new Sword()).add(new Troll()).tick(killTroll)
--> You have no sword.
$.. ==> @foyer
Sword is shimmering.
Troll is getting hungry.
jshell> new Room("foyer").add(new Sword()).add(new Troll()).tick(takeSword).tick(killTroll)
--> You have taken sword.
--> Troll is killed.
$.. ==> @foyer
Sword is shimmering.
jshell> new Room("foyer").add(new Candle()).add(new Troll()).add(new Sword()).tick().tick(takeSword)
--> You have taken sword.
$.. ==> @foyer
Candle is about to burn out.
Troll is VERY hungry.
Sword is shimmering.
jshell> new Room("foyer").add(new Candle()).add(new Troll()).add(new Sword()).tick().tick(takeSword).tick(killTroll)
--> You have taken sword.
--> Troll is killed.
```

```
$.. ==> @foyer
Candle has burned out.
Sword is shimmering.
jshell> new Room("foyer").add(new Candle()).add(new Troll()).add(new Sword()).tick().tick(killTroll)
--> You have no sword.
$.. ==> @foyer
Candle is about to burn out.
Troll is VERY hungry.
Sword is shimmering.
jshell> new Room("foyer").add(new Candle()).add(new Troll()).add(new Sword()).tick().tick(killTroll).tick(takeSword)
--> You have no sword.
--> You have taken sword.
$.. ==> @foyer
Candle has burned out.
Troll is SUPER HUNGRY and is about to ATTACK!
Sword is shimmering.
jshell> new Room("foyer").add(new Candle()).add(new Troll()).add(new Sword()).tick().tick(killTroll).tick(takeSword).tick(killTro
--> You have no sword.
--> You have taken sword.
--> Troll is killed.
$.. ==> @foyer
Candle has burned out.
Sword is shimmering.
jshell> new Room("foyer").add(new Candle()).add(new Troll()).tick(killTroll)
--> You have no sword.
$.. ==> @foyer
Candle is getting shorter.
Troll is getting hungry.
jshell> /exit
```

Level 5

We are now ready to venture into other rooms. Going into a room requires that a room be created first. This is done via the go method that takes in another lambda of the form `x -> new Room...`. As an example, we can represent *deja vu* with the following test:

```
jshell> new Room("dining").add(new Candle()).add(new Sword())
$.. ==> @dining
Candle flickers.
Sword is shimmering.
jshell> new Room("dining").add(new Candle()).add(new Sword()).go(x -> new Room("mystery", x))
$.. ==> @mystery
Candle flickers.
Sword is shimmering.
```

Notice that this new mystery room that we just entered looks like the same room before! Not only that we can also bring an item that we picked into a new room.

```
jshell> new Room("foyer").add(new Sword()).tick(takeSword).go(x -> new Room("dining").add(new Candle()))
--> You have taken sword.
$.. ==> @dining
Sword is shimmering.
Candle flickers.
```

Note that things brought into a new room are listed first.

```
$ javac your_java_files
$ jshell -q your_java_files_in_bottom-up_dependency_order actions.jsh < test5.jsh
jshell> new Room("dining").add(new Candle()).add(new Sword())
$.. ==> @dining
Candle flickers.
Sword is shimmering.
jshell> new Room("dining").add(new Candle()).add(new Sword()).go(x -> new Room("mystery", x))
$.. ==> @mystery
Candle flickers.
Sword is shimmering.
jshell> new Room("dining").add(new Candle()).tick().add(new Sword()).go(x -> new Room("mystery", x))
$.. ==> @mystery
Candle is getting shorter.
Sword is shimmering.
jshell> new Room("foyer").add(new Sword()).tick(takeSword).go(x -> new Room("dining").add(new Candle()))
--> You have taken sword.
$.. ==> @dining
Sword is shimmering.
Candle flickers.
jshell> Room r1 = new Room("foyer").add(new Candle())
jshell> Room r2 = r1.go(x -> new Room("library").add(new Sword()))
jshell> Room r3 = r2.go(x -> new Room("dining").add(new Troll()))
jshell> r3.tick(killTroll)
--> You have no sword.
$.. ==> @dining
Troll is getting hungry.
jshell> r2.tick(takeSword).go(x -> new Room("dining").add(new Candle()).add(new Troll()))
--> You have taken sword.
$.. ==> @dining
Sword is shimmering.
```

```
Candle flickers.
Troll lurks in the shadows.
jshell> r2.tick(takeSword).go(x -> new Room("dining").add(new Candle()).add(new Troll())).tick(killTroll)
--> You have taken sword.
--> Troll is killed.
$.. ==> @dining
Sword is shimmering.
Candle is getting shorter.
jshell> r1.go(x -> new Room("library").
   ...> add(new Sword()).
   ...> tick(takeSword).
   ...> go(y -> new Room("dining").add(new Candle()).add(new Troll()))).tick(killTroll)
--> You have taken sword.
--> Troll is killed.
$.. ==> @dining
Sword is shimmering.
Candle is getting shorter.
jshell> /exit
```

Level 6

Finally, we would like to go back to the previous rooms. For simplicity, once we come back from a room, the room that we came back from vanishes, so we can't go into it again. Note that when you return from room B to room A, things in room A are output first, followed by B (this is to be consistent with the output in the previous level dealing with `go`). That is to say, things in the room that is created earlier are always listed first. Also note that once we enter into a new room, and return from it, time in the original room ticks only once.

Lastly, add an action `dropSword` to drop the sword.

```
$ javac your_java_files
$ jshell -q your_java_files_in_bottom-up_dependency_order actions.jsh < test6.jsh
jshell> Room r1 = new Room("foyer").add(new Candle())
jshell> Room r2 = r1.go(x -> new Room("dining").add(new Troll()))
jshell> Room r3 = r2.go(x -> new Room("library").add(new Sword()))
jshell> r1
r1 ==> @foyer
Candle flickers.
jshell> r2
r2 ==> @dining
Troll lurks in the shadows.
jshell> r2.back()
$.. ==> @foyer
Candle is getting shorter.
jshell> r2.tick().tick().tick()
$.. ==> @dining
Troll is SUPER HUNGRY and is about to ATTACK!
jshell> r2.tick().tick().tick().back()
$.. ==> @foyer
Candle is getting shorter.
jshell> r3
r3 ==> @library
Sword is shimmering.
jshell> r3.back()
$.. ==> @dining
Troll is getting hungry.
jshell> r3.back().back()
$.. ==> @foyer
Candle is getting shorter.
jshell> r3.tick(takeSword).back().tick(killTroll).back()
--> You have taken sword.
--> Troll is killed.
$.. ==> @foyer
Candle is getting shorter.
Sword is shimmering.
jshell> r3.tick(takeSword).back().tick(killTroll).tick(dropSword)
--> You have taken sword.
--> Troll is killed.
--> You have dropped sword.
$.. ==> @dining
Sword is shimmering.
jshell> r3.tick(takeSword).back().tick(killTroll).tick(dropSword).back()
--> You have taken sword.
--> Troll is killed.
--> You have dropped sword.
$.. ==> @foyer
Candle is getting shorter.
jshell> r1.go(x -> new Room("dining").add(new Troll())).
   ...> tick().
   ...> go(x -> new Room("library").add(new Sword())).
   ...> tick().
   ...> tick(takeSword).
   ...> back().
   ...> tick().
   ...> tick(killTroll)
--> You have taken sword.
--> Troll is killed.
```

```
$.. ==> @dining
Sword is shimmering.
jshell> /exit
```

Submission (Practice)

Your Files: BROWSE

SUBMIT (only .java, .c, .cpp, .h, .jsh, and .py extensions allowed)

To submit multiple files, click on the Browse button, then select one or more files. The selected file(s) will be added to the upload queue. You can repeat this step to add more files. Check that you have all the files needed for your submission. Then click on the Submit button to upload your submission.