Homework 02 – Trick or Treat

Authors: Rachna, Nick, Calvin, Sooraj, Allan
Topics: Javadocs, inheritance, copy constructors, static variables, wrapper classes

Problem Description

Please make sure to read all parts of this document carefully.
As Halloween approaches, you decide to go trick or treating with friends, and want to document your memories using your knowledge of object-oriented programming. Some of your friends decide to be witches, and some decide to be ghosts. All of you are trick or treaters, and all of you want to keep track of the total amount of candy you get. Use the following instructions to write your classes.

Solution Description

You'll be creating the 5 classes in this assignment: TrickOrTreater, Ghost, Witch, BlackCat, and HappyHalloween. Each of these has constructors, methods, and variables that are described below. A lot of the code will be reused through inheritance, so make sure you pay attention to the hints given below!

TrickOrTreater.java

This file defines a `TrickOrTreater` object.

Variables:

The `TrickOrTreater` class must have these variables:

- `name` – the name of the trick or treater. This should be a String. It should be visible to subclasses of `TrickOrTreater`.
- `neighborhood`– the neighborhood the trick or treater lives in. This should be a String. It should be visible to subclasses of `TrickOrTreater`.
- `numCandy`– the integer amount of candy this `TrickOrTreater` has. It should be visible to subclasses of `TrickOrTreater`.
- `totalCandy` – The total integer amount of candy acquired by all trick or treaters. Think about what reserved word you need to use for this variable. It should be visible to subclasses of `TrickOrTreater`.

Constructor(s):

- A constructor that takes in the `name`, `neighborhood`, and `numCandy`.
- A constructor that takes in no parameters. `name` should be set to "Agnes". `neighborhood` should be set to "Halloweentown." `numCandy` should be set to 0.
- Assume that inputs to the constructor are valid inputs.
- You **must** minimize the amount of code written and maximize code reuse when writing these constructors. Hint: what keyword have we learned that does this?

Methods:

The following methods should be public.

- `seekCandy`
  - o takes in an `int luck` and calculates the number of candy gained to be 3*luck. This amount of candy should be added to `numCandy` and `totalCandy`. The method should not return anything.
- Getters and setters for each of the instance variables.
  - o Remember, if `numCandy` is updated, `totalCandy` should also be updated.
  - o `numCandy` and `totalCandy` should only be able to increase.

Ghost.java

This file defines a `Ghost` object, and should be a subclass of `TrickOrTreater`

Variables:

In addition to the variables it inherits from its superclass, Ghost should also have the following variables:

- `transparency`– this should be an int that represents how transparent the Ghost is. This should not be visible outside the Ghost class.

Constructor(s):

- A constructor that takes in the `name`, `neighborhood`, `numCandy`. and `transparency`.
- A constructor that takes in `transparency`. (`name`, `neighborhood`, and `numCandy` should follow the same default values as in `TrickOrTreater`. How can you implement this using constructor chaining?)
- A constructor that takes in another `Ghost other` and creates a copy based on that `Ghost`.
- Assume that inputs to the constructor are valid inputs.

Methods:

The following methods should be public.

- `spook`
  - o This should print out:
  - o "Very spooky" if the transparency level is at 10 or higher
  - o "Boo!" if the transparency level is between 3 and 9, inclusive
  - o "Not very spooky" if the transparency level is at 2 or lower.
- Getters and setters for each of the instance variables

Witch.java

This file defines a Witch object and should be a subclass of TrickOrTreater.

Variables:
In addition to the variables it inherits from its superclass, Witch should also have the following variables:

- `signatureSpell` – this should be a String that represents the Witch's favorite spell. This should not be visible outside the Witch class.
- `companion`– this should be an object of type BlackCat that accompanies this witch. (More on that class below). This should not be visible outside the Witch class.

Constructor(s):

- A constructor that takes in the `name`, `neighborhood`, `numCandy`, `signatureSpell`, and `companion`.
- A constructor that takes in `name`, `signatureSpell`, and `companion`. `neighborhood` should be set to "Godric's Hollow" and `numCandy` should be set to 13; This constructor should chain to Witch's other constructor to maximize code reuse.
- A constructor that takes in another `Witch other` and creates a **deep** copy based on that Witch. (Hint: should changes made to the other witch's companion affect this witch's companion?)
- Assume that inputs to the constructor are valid inputs.

Methods:
The following methods should be public.

- `castSpell`
  - o This method should print out, "<name> casts <`signatureSpell`>!". It should also double the witch's candy. (This should also update `totalCandy`. Hint: How can you use methods you already wrote to do this?) This method should take in no parameters and does not return anything.
- Getters and setters for each of the instance variables

BlackCat.java

This file defines a BlackCat object. This is an object that a trick-or-treaters dressed as witches should be accompanied by.

Variables:
The BlackCat class must have the following variables:

- `name`– A String that represents the name of the cat. This should not be visible outside the `BlackCat` class.
- `familiar` – A boolean representing if the cat is a familiar (a supernatural creature that may accompany a witch) or not (just a regular cat). This should not be visible outside the `BlackCat` class.

Constructor(s):

- A constructor that takes in name and familiar.

Methods:
The following methods should be public.

- `meow`
    - This should print out "<name> is a familiar" or "<name> is not a familiar" based on the value of `familiar`.
- Getters and setters for `name` and `familiar`.

Testing

HappyHalloween.java

This Java file is a driver, meaning it will contain and run `TrickOrTreater, Witch, Ghost, and BlackCat`. Use this to test your code. Here are some tips to help you get started:

- Create at least 2 Witches, Ghosts, and Blackcoats.
- Use the copy constructors at least once. Try modifying the objects to see if you have deep copied properly.
- Call `seekCandy()` using each of your TrickOrTreaters.
- Print out the amount of `totalCandy` they all get.
- Reuse your code when possible. These tests and the ones on Gradescope are by no means comprehensive, so be sure to create your own!

Checkstyle

You must run checkstyle on your submission (To learn more about Checkstyle, check out cs1331-style-guide.pdf under CheckStyle Resources in the Modules section of Canvas.) **The Checkstyle cap for this assignment is 15 points.** This means there is a maximum point deduction of 15. If you don't have Checkstyle yet, download it from Canvas -> Modules -> CheckStyle Resources -> checkstyle-8.28.jar. Place it in the same folder as the files you want to run Checkstyle on. Run checkstyle on your code like so:

```
$ java -jar checkstyle-8.28.jar yourFileName.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. If you had any errors, they would show up above this message, and the number at the end would be the points we would take off (limited by the checkstyle cap mentioned in the Rubric section). In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

Additionally, you must Javadoc your code.

Run the following to only check your Javadocs:

```
$ java -jar checkstyle-8.28.jar -j yourFileName.java
```

Run the following to check both Javadocs and Checkstyle:

```
$ java -jar checkstyle-8.28.jar -a yourFileName.java
```

For additional help with Checkstyle see the CS 1331 Style Guide.

Turn-In Procedure

Submission

To submit, upload the files listed below to the corresponding assignment on Gradescope:

- `TrickOrTreater.java`
- `Ghost.java`
- `Witch.java`
- `BlackCat.java`
- `HappyHalloween.java`

Make sure you see the message stating the assignment was submitted successfully. From this point, Gradescope will run a basic autograder on your submission as discussed in the next section. **Any autograder test are provided as a courtesy to help "sanity check" your work and you may not see all the test cases used to grade your work.** You are responsible for thoroughly testing your submission on your own to ensure you have fulfilled the requirements of this assignment. If you have questions about the requirements given, reach out to a TA or Professor via Piazza for clarification.

You can submit as many times as you want before the deadline, so feel free to resubmit as you make substantial progress on the homework. We will only grade your latest submission. **Be sure to submit every file each time you resubmit**.

Gradescope Autograder

If an autograder is enabled for this assignment, you may be able to see the results of a few basic test cases on your code. Typically, tests will correspond to a rubric item, and the score returned represents the performance of your code on those rubric items only. If you fail a test, you can look at the output to determine what went wrong and resubmit once you have fixed the issue.

The Gradescope tests serve two main purposes:

- Prevent upload mistakes (e.g. non-compiling code)
- Provide basic formatting and usage validation

In other words, the test cases on Gradescope are by no means comprehensive. Be sure to thoroughly test your code by considering edge cases and writing your own test files. You also should avoid using Gradescope to compile, run, or Checkstyle your code; you can do that locally on your machine.

Other portions of your assignment can also be graded by a TA once the submission deadline has passed, so the output on Gradescope may not necessarily reflect your grade for the assignment.

Allowed Imports

To prevent trivialization of the assignment, you are not allowed to import any 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`
- `System.arraycopy`

Collaboration

Only discussion of the Homework (HW) at a conceptual high level is allowed. You can discuss course concepts and HW assignments broadly, that is, at a conceptual level to increase your understanding. If you find yourself dropping to a level where specific Java code is being discussed, that is going too far. Those discussions should be reserved for the instructor and TAs. To be clear, you should never exchange code related to an assignment with anyone other than the instructor and TAs.

Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items
- Do not submit `.class` files
- Test your code in addition to the basic checks on Gradescope
- Submit every file each time you resubmit
- Read the "Allowed Imports" and "Restricted Features" to avoid losing points
- **Check on Ed Discussion for a note containing all official clarifications and sample outputs**

It is expected that everyone will follow the Student-Faculty Expectations document, and the Student Code of Conduct. The professor expects a **positive, respectful, and engaged academic environment** inside the classroom, outside the classroom, in all electronic communications, on all file submissions, and on any document submitted throughout the duration of the course. No inappropriate language is to be used, and any assignment, deemed by the professor, to contain inappropriate, offensive language or threats will get a zero. You are to use professionalism in your work. Violations of this conduct policy will be turned over to the Office of Student Integrity for misconduct.