Homework 04 –*Pirates and Plunder*

Authors: Will, Nathaniel, Faris, Brandon, Matthew, Allan
Topics: polymorphism, dynamic binding, casting

Problem Description

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

Ahoy matey,

The year be 1700. A group of pirates recently acquired a computer from the British Navy (just go with it) in a surprise raid on a British fort. Despite their affinity for tea, the British had recently installed Java on the computer before it was plundered by the pirates. The pirates, being business savvy as they are, have offered a contract worth one-thousand doubloons to you to help modernize their logistical systems. You choose to ignore the ethical issues of piracy for those sweet, sweet doubloons and the opportunity to apply your **polymorphism, dynamic binding,** and **casting** knowledge to a real-world problem.

Solution Description

Create files Loot.java, Coin.java, Sugar.java, Plunderable.java, and Ship.java. You will be creating several fields and methods for each file. Some classes will inherit from other classes, and some will implement an interface. Follow the instructions, paying close attention to keywords, to determine what extends and implements what.

Loot.java

This class represents the spoils gained from plundering a ship or fort. Loot is an abstract class.

Variables:

All variables must not be allowed to be directly modified outside the class in which they are declared, unless stated otherwise:

- `double value` – the monetary value of an item.

Constructor(s):

- Constructor that takes in a `value` and ensures that the value of the item is greater than 0.
- No-param constructor that sets `value` to 0.

Methods:

All methods in this class should be public, unless stated otherwise:

- Getter and setter for `value`. We want both to be public, since the value of a piece of loot might change based on supply and demand.
- `toString()`
  - This method should properly override `Object`'s `toString()` method
  - This method should return "`A piece of loot worth {value}`"

- `equals()`
  - This method should properly override `Object`'s `equals()` method
  - If two Loot objects have the same `value` we say they are equal

Coin.java

This class represents a piece of loot in the form of a coin.

Variables:

All variables must not be allowed to be directly modified outside the class in which they are declared, unless otherwise stated in the description of the variable. Coin inherits all variables from Loot.

- `boolean heads` – represents whether or not the coin is heads up or tails up.
- `int yearMade` – represents the year the coin was manufactured in. `yearMade` must be positive
- `String material` – represents the material the Coin is made of, such as gold, silver, etc.

Constructor(s):

- A constructor that takes in `value`, `heads`, `yearMade`, and `material`
  - Reuse any code if you can
  - Ensure `yearMade` is valid, in the range [0, 1700]. If not, set it to 1700.
- A constructor that takes in `heads` and `yearMade`
  - Use constructor chaining if possible
  - For value, set it to `(3000 - yearMade) / 100`
  - Will use a default material of `"Gold"`

Methods:

All methods must have the proper visibility to be used where it is specified they are used.

- `toString()`
  - This method should properly override Object's and Loot's `toString()` methods
  - It should print the following:
    `"A {material} coin made in {yearMade}. Heads side is up: {heads}.`
- `equals()`
  - This method should properly override Object's `equals()` method
  - If two `Coin` objects have the same `year, value, and material`, they are equal
  - Reuse code. Hint: how can you use Loot's `equals()` method?

Sugar.java

This class represents the valuable cash crop of sugar as loot, a frequently pirated commodity.

Variables:

All variables must not be allowed to be directly modified outside the class in which they are declared, unless otherwise stated in the description of the variable. Sugar inherits all variables from Loot.

- `double amount` – unlike a coin, we don't have "one sugar". Instead we have an amount represented by a double
- `double sweetness` – This value represents the sweetness of the sugar. It must be in range 0 to 100 inclusive - [0, 100]

Constructor(s):

- A constructor that takes in `amount and sweetness`
  - Ensure `sweetness` value is within acceptable range. If not, set to 0.
  - After determining the appropriate `sweetness` above, the `value` of the sugar should be determined by its amount times its sweetness

Methods:

All methods must have the proper visibility to be used where it is specified they are used.

- `toString()`
  - This method should properly override `Object`'s `toString()` method
  - It should print the following:
    "`A pile of sugar of size {amount} and sweetness {sweetness}.`"
- `equals()`
  - This method should properly override `Object`'s `equals()` method
  - If two `Sugar` objects have the same `amount`, `value`, and `sweetness`, they are equal
  - Reuse code. Hint: how can you use Loot's `equals()` method?

Plunderable.java

Plunderable is an interface which is implemented when an object can be plundered.

Methods:

Plunder only has one method, `bePlundered`.

- `bePlundered()`
  - This method should not be implemented in plunderable. It should, when implemented, return an array of Loot objects, representing what the object plundered had on hand.

Ship.java

This class represents a ship on the high seas with cargo on board.

Ship should implement Plunderable.java

Variables:

All variables must not be allowed to be directly modified outside the class in which they are declared, unless otherwise stated in the description of the variable.

- `Loot[] cargo` – This is the cargo a ship has aboard. Initially, this array should be of length 10.
- `double totalCargoValue` – This is the total value of the cargo on the ship. You should continually update this whenever new cargo is added or removed.
- `String name` – Every ship needs a good name, these are no exception

Constructor(s):

- A one-param constructor which takes in a `name` and sets the initializes `cargo`'s length to the value specified is required.
- A no-parameter constructor chained to the one above, with default name `Black Pearl`

Methods:

All methods must have the proper visibility to be used where it is specified they are used.

- `toString()`
  - This method should properly override `Object`'s `toString()` method
  - It should print the following:
    "`A ship called {name} with cargo {cargo[0]}, {cargo[1]}..., which has a total value of {totalCargoValue}.`"
  - Only slots in the array that have something in them should have their values displayed
  - Round `totalCargoValue` to 2 decimal places in the string
- `addCargo(Loot newItem)`
  - This method should be responsible for the logic of adding cargo to a ship. This new cargo should take the first available slot in the array. Additionally, update the `totalCargoValue` of the ship.
  - If there are **no slots left in the array,** create a new array with twice the current one's length, copy over the old one's cargo, and add the new cargo according to the previous rule.
- `removeCargo()`
  - This method will set the first filled cargo slot in the array to null, then return that cargo item.
  - It should also update `totalCargoValue`.
  - If the array is empty, return `null`
- `removeCargo(Loot cargo)`
  - This method will iterate through the array, checking if that cargo is on the ship. If it is, perform the operation specified in `removeCargo()` on that item. Else, return null. Note: check for object equality, **not** reference equality when determining if an item is in the array.
- `bePlundered`()
  - Return an array of the Loot objects in the ship's cargo, then set all the items in the old array to `null`. E.g. if a ship with 3 coins is plundered, you should return an array of 3 coins and that ship's cargo array should be empty.
  - This method should also account for the `totalCargoValue`'s changes.

Testing

Reuse your code when possible. Certain methods can be reused using certain keywords. 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 25 points.** This means there is a maximum point deduction of 20. 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:

- `Loot.java`
- `Coin.java`
- `Sugar.java`
- `Plunderable.java`
- `Ship.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.