Lab Exercise 6: Abstract Classes and Interfaces
CS 2334

September 29, 2016

Introduction

In this lab, you will experiment with using inheritance in Java through the use of abstract classes and interfaces. Continuing the fantasy game theme of Lab 2, you will implement a set of classes that represent an inventory of items of different types. In addition, your implementation will facilitate the comparison of Item objects, even when they are different types of items.

Learning Objectives

By the end of this laboratory exercise, you should be able to:

1. Create and extend abstract classes and methods
2. Use interfaces to define standard behavior across multiple classes

Proper Academic Conduct

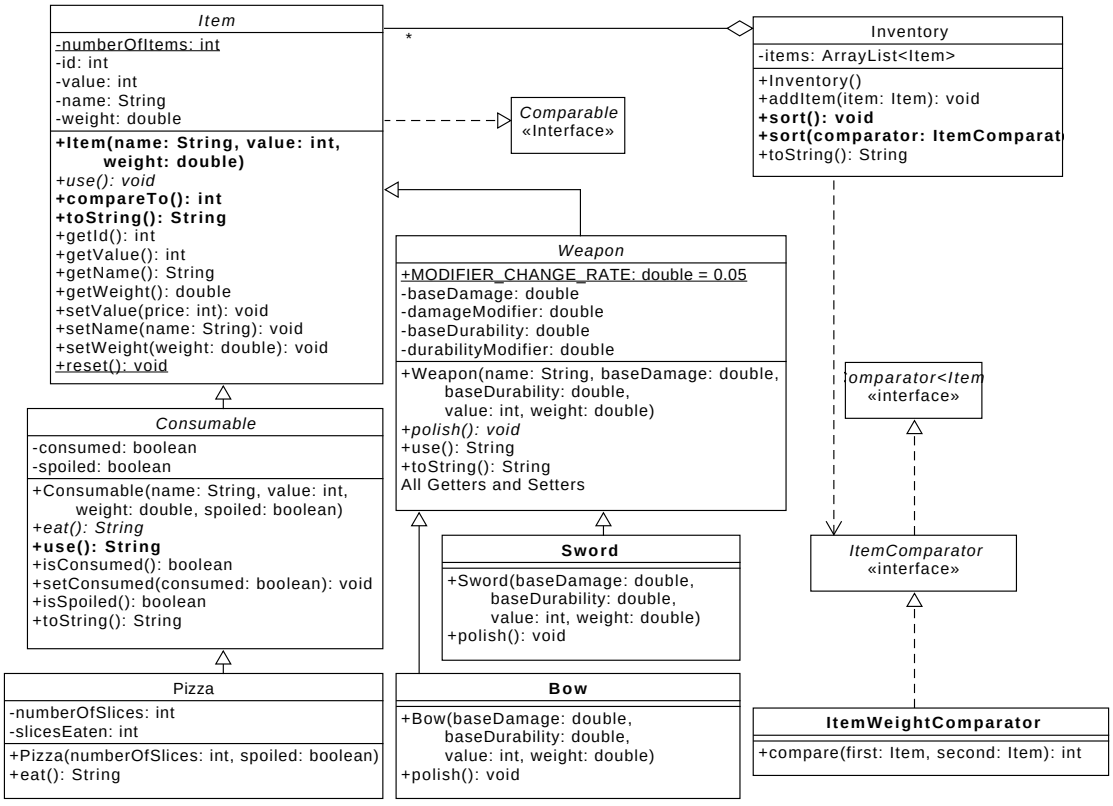
This lab is to be done individually. Do not look at or discuss solutions with anyone other than the instructor or the TAs. Do not copy or look at specific solutions from the net.

Preparation

1. Import the existing lab6 implementation into your eclipse workspace:
   `http://www.cs.ou.edu/~fagg/classes/cs2334/labs/lab6/lab6.zip`

Representing an Inventory of Items

Below is the UML representation of a set of classes that represent the inventory and various types of items. Your task will be to implement this set of classes and an associated set of JUnit test procedures. Methods and classes where you are providing an implementation are shown in bold.



The classes in italics represent abstract classes or interfaces. The concrete child classes must implement all methods from the abstract parent classes. In this lab, **Item**, **Consumable**, and **Weapon** are the abstract classes.

The line from **Item** to **Comparable** indicates that **Item** must implement the **Comparable** interface. Similarly, the **ItemWeightComparator** class must implement the **ItemComparator** interface, which extends the **Comparator** interface. **ItemWeightComparator** compares items based on their weight.

Specific Steps

The **Item** class is the common ancestor to the various types of items that can exist in this fantasy game.

- All instances of **Item** are given a unique int *id*. These are to be assigned by the **Item** constructor. The first instance of an item is assigned an *id* of 0 (zero); the next is assigned 1, etc. Note that you have available a class variable that will help with the implementation of the constructor (and there is a static *reset()* method that facilitates writing unit tests).

- *compareTo(Item other)*: The **Item** class implements the **Comparable** interface. This requires adding the *compareTo(Item other)* method to the class. The *compareTo(Item other)* method takes in another instance of **Item** and compares it to the current instance. If the current instance's *value* field is greater than *other*'s *value* field then the method should return a positive integer (convention is 1). If the current instance's *value* field is less than *other*'s *value* field then the method should return a negative integer (convention is -1). If both items are equal, then compare the *name* field of the items lexicographically (meaning, compare each character in the strings based on its value, ignoring case. i.e. A == a), returning the appropriate value. *Hint: you might find something helpful for this in the API of the String class.*

- *Item.toString()*: for an **Item** with the *name* of "ring", a *value* of 3000, and a *weight* of 0.013, the method must return a String in the following format (excluding the quotes):

```
"ring -- Value: 3000, Weight: 0.01"
```

The **ItemWeightComparator** class implements the **ItemComparator** interface, meaning instances of it can be passed to methods requiring a comparator for objects of type **Item**.

- The *compare(Item first, Item second)* method of **ItemWeightComparator** should function similarly to the *compareTo(Item other)* method of the **Item** class, but for the *weight* field of the **Item**s. If the weights are equal, this method should call the *compareTo(Item other)* method of the *first* **Item** and return the resulting value.

The **Weapon** class is an abstract implementation of **Item** and describes items that can deal damage and break from use. The implementation of this class is provided for you. All instances of **Weapon** have a base damage value *baseDamage* and a modifier to that value *damageModifier*. The sum of these two values determines the *effective damage* that this Weapon can do on a single use. In addition, **Weapon**s have a base durability value *baseDurability*, and a modifier to that value *durabilityModifier*. The sum of these two values determines the *effective durability* of the **Weapon**. When this sum reaches zero or less, the effective durability is zero and the *Weapon* is considered to be *broken* and cannot be used.

We provide several implemented methods that include:

- *Weapon.getDamage():* Returns the effective damage of the **Weapon**.
- *Weapon.getDurability():* Returns the effective durability of the **Weapon**.
- *Weapon.toString()*: for a **Weapon** with the *name* of "hammer", a *value* of 300, a *weight* of 2.032, a *baseDamage* value of 30.4219, a *damageModifier* of 0.05, a *baseDurability* of 0.7893, and a *durabilityModifier* of 0.05, the method returns a String in the following format:

```
"hammer -- Value: 300, Weight: 2.03, Damage: 30.47, Durability: 83.93%"
```

- *Weapon.use()*: This method returns a String describing what happens when a **Weapon** is used. For a **Weapon** with the *name* of "hammer", and an effective damage of 30.4725, the method should return the following:

```
"You use the hammer, dealing 30.47 points of damage."
```

"Using" a **Weapon** lowers (subtracts) its effective durability by *Weapon.MODIFIER_CHANGE_RATE*. If the effective durability of the **Weapon** hits or drops below 0, the **Weapon** will "break". If the **Weapon** "breaks", the method should output the previous String, but additionally with a newline character and the additional text "The hammer breaks.":

```
"You use the hammer, dealing 34.05 points of damage.
The hammer breaks."
```

For a **Weapon** with the *name* of "hammer", if it is "broken" (The effective durability is 0 or less), calling its *use()* method returns the following:

```
"You can't use the hammer, it is broken."
```

In this case, there is no change to *durabilityModifier*.

The **Sword** class is a concrete implementation of **Weapon** that you must provide.

- All instances of the Sword class have the *name* "sword".
- *Sword.polish():* This method increases the instance's **damageModifier** by adding *Weapon.MODIFIER_CHANGE_RATE* each time *polish()* is called, up to 25% of the *baseDamage* value. If the base damage of a sword were to be 100, then the maximum that the effective damage could be increased to would be 125.

The **Bow** class is a concrete implementation of **Weapon** that you must provide.

- All instances of the Bow class have the *name* "bow".
- *Bow.polish():* This method increases the instance's **durabilityModifier** by adding *Weapon.MODIFIER_CHANGE_RATE*. Any changes are capped such that **effective durability** is no larger than one (1).

Much like in Lab 2, the **Inventory** class is a container for items in this fantasy game. This class has been partially implemented already, but you will need to add the following methods:

- *Inventory.sort():* This sorts the items in the **Inventory** instance based on their *value*.
- *Inventory.sort(ItemComparator comparator):* This sorts the items in the **Inventory** instance based on their *weight*.

The **Consumable** class describes those items that can be eaten by the player. Consumables can be marked as consumed, and can be spoiled. These properties are stored in the instance variables *consumed* and *spoiled*, respectively. A newly-created **Consumable** object should have its *consumed* field set to false.

- *Consumable.use():* If a **Consumable** is not spoiled and is not consumed, calling this simply returns the value from a call to *Consumable.eat()*. For a **Consumable** with the *name* of "bread" that has already been consumed, this method returns the following:

```
"There is nothing left of the bread to consume."
```

Assuming for this **Consumable** named "bread" that the value returned by a call to its **eat()** method is the following:

```
"You eat the bread."
```

If this "bread" were to be spoiled, the method returns this String, appended with a newline and the text "You feel sick."

```
"You eat the bread.
You feel sick."
```

Lab 6: Specific Instructions

Start from the class files that are provided in lab6.zip.

1. Modify the **Item**, **Consumable**, and **Inventory** classes.
2. Create the **Sword**, **Bow** and **ItemWeightComparator** classes.
   - Be sure that the class names are exactly as shown
   - You must use the default package, meaning that the package field must be left blank
3. Use a **Driver** class for quick tests, as needed
4. Create JUnit test classes for the methods that you have created or modified.
   - You need to convince yourself that everything is working properly
   - Make sure that you cover the methods in question while creating your tests. Keep in mind that we have our own tests that we will use for grading.

Final Steps

1. Generate Javadoc using Eclipse.
   - Select *Project/Generate Javadoc...*
   - Make sure that your project and all of its classes are selected.
   - Select *Private* visibility
   - Use the default destination folder
   - Click *Finish*
2. Open the *lab6/doc/index.html* file using your favorite web browser or Eclipse (double clicking in the package explorer will open the web page). Check to make sure that that both of your classes are listed and that all of your documented methods have the necessary documentation.
3. If you complete the above instructions during lab, you may have your implementation checked by one of the TAs.

Submission Instructions

- All required components (source code and compiled documentation) are due at 11:59pm on Friday, September 30th.
- Method 1: Submit through Eclipse
   1. From the *Window* menu, select *Preferences/Configured Assignment*.
   2. Select your project.
   3. From the Project menu, select *Submit Assignment*.
   4. Under *Select the assignment to submit*, select *Lab 6: Lab 6: Abstract Classes and Interfaces*.
   5. Click *Change Username or Password....* Enter your Web-Cat username and password. Click *OK*. You should only need to do this step once per session.
   6. Click *Finish*.

7. Your browser should automatically open a Web-Cat page that shows your submission being graded. After a short wait, the page will show a report of your submission. See the main class web page for a link that describes the Web-Cat output.

- Method 2: Submit directly to the Web-Cat server

1. From the File menu, select *Export*.
2. Select *Java/JAR File*. Click *Next*.
3. Select and expand your project folder.
4. Select your *src* and *doc* folders.
5. Select *Export Java source files and resources*.
6. Select an export destination location (e.g., your *Documents* folder/directory). This file should end in *.jar*
7. Select *Add directory entries*.
8. Click *Finish*.
9. In your web browser, login to the Web-Cat server.
10. Click the *Submit* button.
11. Browse to your jar file.
12. Click the *Upload Submission* button.
13. The next page will give you a list of all files that you are uploading. If you selected the correct jar file, then click the *Confirm* button.
14. Your browser will then open a Web-Cat page that shows your submission being graded. After a short wait, the page will show a report of your submission. See the main class web page for a link that describes the Web-Cat output.

Rubric

The project will be graded out of 100 points. The distribution is as follows:

Correctness/Testing: 45 points

The Web-Cat server will grade this automatically upon submission. Your code will be compiled against a set of tests (called *Unit Tests*). These unit tests will not be visible to you, but the Web-Cat server will inform you as to which tests your code passed/failed. This grade component is proportional to the fraction of tests that your code passes (so 22.5 points means that your code passed half of the tests)

Style/Coding: 20 points

The Web-Cat server will grade this automatically upon submission. Every violation of the *Program Formatting* standard described in Lab 1 will result in a subtraction of a small number of points (usually two points). Looking at your submission report on the Web-Cat server, you will be able to see a notation for each violation that describes the nature of the problem and the number of subtracted points.

Design/Readability: 35 points

This element will be assessed by a grader (typically sometime after the lab deadline). Any *errors* in your program will be noted in the code stored on the Web-Cat server, and two points will be deducted for each. Possible errors include:

- Non-descriptive or inappropriate project- or method-level documentation (up to 10 points)
- Missing or inappropriate inline documentation (2 points per violation; up to 10 points)
- Inappropriate choice of variable or method names (2 points per violation; up to 10 points)
- Inefficient implementation of an algorithm (minor errors: 2 points each; up to 10 points)
- Incorrect implementation of an algorithm (minor errors: 2 points each; up to 10 points)

- Incomplete coverage of your Unit Tests. We expect that your unit tests will test all lines of your code (up to 15 points)

If you do not submit compiled Javadoc for your lab, 5 points will be deducted from this part of your score.

Note that the grader may also give *warnings* or other feedback. Although no points will be deducted, the issues should be addressed in future submissions (where points may be deducted).

Bonus: up to 5 points

You will earn one bonus point for every two hours that your assignment is submitted early.

Penalties: up to 100 points

You will lose ten points for every minute that your assignment is submitted late.