CSCI-605 Advanced Object-Oriented Programming Concepts
Homework 4: Toy Store

1 Introduction

For this homework, you will be implementing a simulation of a toy store named *Al's Toy Barn*. Each morning, the Toy Store restocks its shelves with brand new, mint condition toys ordered from the *Fly By Night Toy Factory*. Unfortunately, Fly-By-Night is somewhat legendary for producing shoddy products that break after only being used a few times; the condition of their toys degrades each time that it is played with. That doesn't seem to matter to the eager customers that begin lining up before the store even opens each day. Each customer quickly buys the first toy that they see and plays with it. But Al's customers have very short attention spans, and they quickly become bored with their toys. After playing with them only once, they sell them back to Al at a reduced price. If a toy is broken, Al discards it rather than restocking it. Once the store runs out of toys in stock (because they are all broken), Al closes for the day.

1.1 Goals

This homework will help students to gain experience working with:

- Inheritance and Abstract Classes
- Interfaces
- Polymorphism
- Factory Method Design Pattern
- Access modifiers: public vs protected vs private
- Method overriding
- Superclass method calling

2 Starter Code

To get the starter code, click on this GitHub classroom link. There are four directories in the starter code.

Note: You cannot modify any of these files except for the ToyFactory class.

- **output**: The outputs of the main program when running with 10, 50, and 100 toys.
- **src.test**: The folder containing JUnit tests that will help you develop and verify your toys.
- **src.store**: This folder contains the toy store and the simulation program.
  - `AlsToyBarn` - This is the implementation of the toy store that is used in the simulation.
- **src.toy**: The folder containing all the classes about the toys.
  1. `BatteryType` - An `enum` that defines the differet battery types that a toy may use (*AA*, *AAA*, *AAAA*, *C*, *D*, and *9-Volt*).
  2. `Condition` - An `enum` that defines the conditions that a toy may be in (*mint*, *near mint*, *very good*, *good*, *fair*, *poor*, and *broken*). A toy's condition is determine by its level of degradation. In addition to defining the possible value's for a toy's condition, the `Condition enum` also defines the following methods:
     (a) `getMultiplier()` - Each possible condition has an associated multiplier that can be accessed using the condition's `getMultiplier` method. The resale value of a toy is determined by multiplying its MSRP[1] by this multiplier. For example, a toy that is in *very good* condition has a resale value of 75% of its MSRP.
     (b) `get(int degradationLevel)` - Each condition has also associated a range of degradation level (e.g. the range of mint condition is between 0 and 10 (exclusive)). The `get` method will return the toy's condition based on its current degradation level. For example, a toy with 5% level of degradation is considered to be in mint condition.
  3. `IToy` - This is the interface that must be implemented by all toys. It defines the following methods:
     - Accessors for the various common toy attributes (i.e. *name*, *product code*, *MSRP*, and *condition*).
     - `getResaleValue()` - A toy's resale value is determined by its MSRP multiplied by its current condition's multiplier (see above).
     - `play()` - Called whenever a toy is played with. This will degrade the condition of the toy and print a message, (e.g. `"After play, Barbie's condition is NEAR_MINT"`).
  4. `ToyFactory` - A partial implemented toy factory.

1. Manufacturer's Suggested Retail Price

3 Implementation

For this homework you must complete three main programming tasks:

1. Implement a separate class for each of the five varieties of toys: *scooters*, *dolls*, *action figures*, *RC cars*, and *robots*. To earn full credit for this task, it will not be sufficient to simply create 5 classes. Instead, you must make effective use of inheritance and maximize your code reuse.
2. Implement the `ToyFactory`, which provides a `makeToy` factory method to create a toy.
3. Implement the `ToyStore` which runs the simulation.

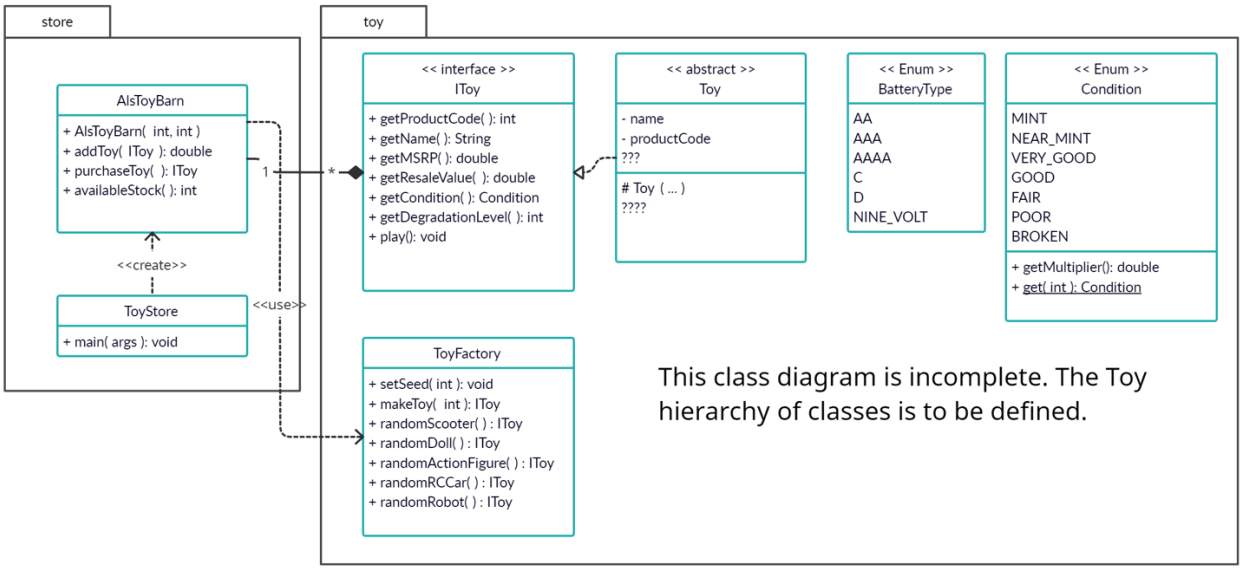
3.1 Design

You have been provided with some starter code. The diagram below depicts the design that you must follow for this homework. **Remember that you cannot modify the provided code.**

The class diagram excludes the private state and behavior. It also omits the entire Toy hierarchy of classes. You are responsible for defining the Toy hierarchy using good inheritance design principles. For example, a data member shared by all subclasses should be defined in the shared superclass. While working on the design, try to answer the following questions:

1. What are the attributes common to all toys?
2. What behaviors should all toys be able to do?
3. What attributes and behaviors are unique to a particular toy?
4. Which toys will need to customize the way they do something different from other toys, and what are those customizations?

Note: You do not need to submit the answer to all the questions above. Those questions are meant to help you designing your program.



3.2 Activity 1: The Toy Implementations

You will be required to implement a class for each of the required variety of toys. Remember, *all* of the code that you write should be in the `toy` package. To earn full credit for this task, you must make effective use of inheritance.

You are not allowed to use the `instanceof` operator nor type casting for this lab.

Your implementations must meet the specifications below:

- All toys have a unique 7-digit product code, a name, an MSRP, a condition, and a degradation level. All toys start in mint condition and zero level of degradation.
  The product code is a sequential number that will increment by 1 every time a toy is created. Each type of toy has a different product code. For example, the Scooter's product code starts at 9000000 while the Doll's product code starts at 3000000.
- **Scooter**
  - The 7-digit product code always starts with a "9" and is automatically assigned when the scooter is manufactured.
  - Must have a color (string).
  - Must have 2 or 3 wheels.
  - Must have an odometer that keeps track of the total miles that the scooter has been used.
  - The MSRP must be a random value between 39.99 and 160.99.
  - The string representation must match the following example:
    `"Razor A [product code=9000000, MSRP=89.60, degradation level=90%, condition=BROKEN, resale value=0.00, color=Red, wheels=2, odometer=36]"`.
  - Each time the `play` method is called, two miles are added to the scooter's odometer and its degradation level is increased by 5%. It should produce output that matches the following example:
    `"After play, Razor A's condition is VERY_GOOD"`.
- **Doll**
  - The 7-digit product code always starts with a "3" and is automatically assigned when the doll is manufactured.
  - Must have a hair color (string).
  - Must have an eye color (string).
  - The MSRP must be a random value between 12.99 and 60.00.
  - The string representation must match the following example:
    `"Babs [product code=3000001, MSRP=48.80, degradation level=100%, condition=BROKEN, resale value=0.00, hair color=Blond, eye color=Brown]"`.
  - Each time the `play` method is called its degradation level is increased by 17%. It should produce output that matches the following example:
    `"After play, Babs's condition is VERY_GOOD"`.
- **Action Figure**
  - The 7-digit product code always starts with a "5" and is automatically assigned when the action figure is manufactured.
  - Must have a hair color (string).

- Must have an eye color (string).
- May or may not have Kung-Fu Grip™.
- The MSRP must be a random value between 9.99 and 24.99.
- The string representation must match the following example:
  `G.I. Barbie [product code=5000000, MSRP=24.87, degradation level=0%, condition=MINT, resale value=24.87, hair color=Red, eye color=Green, kung-fu grip=true]"`.
- Each time the `play` method is called its degradation level is increased by 10%. It should produce output that matches the following example:
  `"After play, G.I. Barbie's condition is VERY_GOOD"`.

- **RC Car**
  - The 7-digit product code always starts with a "6" and is automatically assigned when the RC Car is manufactured.
  - Must have a battery type.
  - Must have a number of batteries (a random number between 1 and 6).
  - Must have a scale speed (in MPH). This must be a random value between 100 and 300.
  - The MSRP must be a random value between 19.99 and 159.99.
  - The string representation must match the following example:
    `"METAKOO RC [product code=6000002, MSRP=74.21, degradation level=15%, condition=NEAR MINT, resale value=66.78, battery type=NINE VOLT, number of batteries=2, battery level=70%, speed=133]`.
  - Each time the `play` method is called, its battery gets depleted 30% and its degradation level is increased by 15%. If the RC Car's battery level reaches 0, it cannot be played with and the batteries must be replaced. The `play` method should work as follows:

    ```
    Before play, check the batteries and replace them if needed
    Play with the toy and deplete the batteries
    ```

    It should produce output that matches the following example:
    `"METAKOO RC races in circles at 133 mph!`
    `After play, METAKOO RC's condition is NEAR_MINT"`.

- **Robot**
  - The 7-digit product code always starts with a "7" and is automatically assigned when the RC Car is manufactured.
  - Must have a battery type.
  - Must have a number of batteries (a random number between 1 and 6).
  - Must have a sound that it makes when played with.
  - The MSRP must be a random value between 25.99 and 699.99.
  - The string representation must match the following example:
    `Reginald [product code=7000000, MSRP=73.83, degradation level=20%, condition=VERY_GOOD, resale value=55.37, battery type=AA, number of batteries=1, battery level=75%, sound=I am not a gun!]"`.

– Each time the `play` method is called, its battery gets depleted 25% and its degradation level is increased by 20%. If the Robot's battery level reaches 0, it cannot be played with and the batteries must be replaced. The `play` method should work as follows:

```
Before play, check the batteries and replace them if needed
Play with the toy and deplete the batteries
```

It should produce output that matches the following example:
`"Roomba goes 'Eject!'`
`After play, Roomba's condition is POOR".`

3.3 Activity 2: `ToyFactory`

We want to design our program in such way that the class `AlsToyBarn` can create different type of toys without worrying about the concrete classes must instantiate. A popular pattern in object oriented programming to do that is the **Factory Method**.

The constructors of all the classes created in Activity 1 must not be accessible from the `store` package. The `ToyFactory` class provides a public static method, `makeToy`. This allows a client to create a new toy of any type without worrying about the construction details. The `makeToy` will create a random toy by using the `randomInt` function. `randomInt` will generate a random number in the range received as parameter, for example:

```
// generates a random integer between 1 and 5 (both inclusive)
int type = randomInt(1, 5);
```

From here, the `ToyFactory.makeToy` factory method would invoke the method that creates a toy of that type with random values. That method will create that toy by calling its constructor and return that as a `IToy` reference.

```
if ( type == SCOOTER ) {
    return randomScooter();
}
```

Note that none of the toys' constructor must accept a product code as a parameter. Each toy class is responsible for assigning its own, unique product codes to each toy as it is created.

The `ToyFactory` class contains different lists with names, colors, and sounds for creating the toys. It also provides several utility methods that allow you to generate random integer and double values, and even select a random element from a list. For example, calling `randomString(SCOOTER_NAMES)` will select a random name for a `Scooter` toy.

3.4 Activity 3: ToyStore

For this last activity you will implement the `ToyStore` class which runs the entire simulation. This class must be created in the `store` package.

3.4.1 Command line

The program runs on the command line like this:

```
$ java ToyStore #_toys #_seed
```

If the number of arguments is correct, it is guaranteed to be a valid integers. `#_toys` will be used to restock the Toy store's shelves, and `#_seed` will be used as the seed to the random number generator. The generator will be used to create toys of different type randomly.

If the number of arguments is incorrect, display a usage message and exit the program, i.e.:

```
$ Usage: java ToyStore #_toys #_seed
```

3.4.2 Toy Store simulation

When the program runs it will create an instance of Al's toy store with the given number of toys and seed. Then, the simulation will repeat the following steps until the store runs out of stock:

1. A customer buys the first toy they see
2. Plays with it only once
3. Sells the toy back to Al's toy store at a reduced price
4. Al restocks the toy if not broken (you do not need to implement this step, see the `AlsToyBarn.addToy` method)

4 Testing

Because we are providing a complete JUnit set of tests for your homework, it is suggested you follow the software development process referred as Test Driven Development. Following this approach, each time you develop a new toy class, you should run the test cases and ensure that the module you just developed, plus the previous ones, all work correctly.

If you do not remember how to set up and run JUnit tests, please refer to homework 2.

5 Submission

You will need to submit all of your code to the MyCourses assignment before the due date. You must submit your solution as a ZIP archive named "hw4.zip" (if you submit another format, such as 7-Zip, WinRAR, or Tar, you will not receive credit for this homework).

6 Grading

The grade breakdown is as follows:

- Implementation: 75%
  - Activity 1: 50%
  - Activity 2: 15%
  - Activity 3: 10%
- Testing: 15%
- Code Documentation & Style: 10%