Lab 11: More Inheritance with Polymorphism
CSE-021-01, Spr 2018 Points: 20

Overview

We will work more with inheritance, extending the `ArrayList` class (which is a class that works like an array, but provides more flexibility). Additionally, you will work on a 2-player game, called Pig, to put some concepts into practice.

Before you get started, read chapters 10.5, 10.6 and 10.7. Answer activities 10.5.1, 10.6.1 and 10.7.1 in a Word or text file.

Getting Started

After following the import instructions in the assignment page, you should have a Java project in Eclipse titled Lab 21_11. This PDF document is included in the project in the **doc** directory. The Java files you will use in this lab are in the **src/pig** and **src/arraylist** directories.

If you have missing links then in the project properties (right click on "Lab 21_11" in the package explorer and select properties), edit the "Java Build Path"

- Add JUnit4 (click "add library")
- Add the eduride testing jar (click "add jar", and locate it in "Lab 21_11/lib/eduride-library.jar")

Part 1: ArrayList and Inheritance

The `java.util` class library contains a variety of useful classes. One is the `ArrayList` class, which represents a list of elements that can be accessed by position as with an array. An `ArrayList`, however, can only contain objects from a specific class. To specify the class that an `ArrayList` holds, you use `<` and `>` to enclose the class name, like:

```
ArrayList<String>     //can contain a list of strings
```

Can an `ArrayList` contain `int` data? Not directly, no. An `ArrayList` can only contain objects from a particular class, and an `int` is not an object but a *primitive*: `int`s don't have methods, can't be extended through inheritance, etc. So,

```
ArrayList<int>        //error!
```

will cause a compile error in Java.

Can one `ArrayList` contain both `ModNCounter` and `SeasonCounter` objects? Remember, an `ArrayList` contains only objects from a single class. However, because of inheritance both objects are members of `Counter`, and can be contained in

```
ArrayList<Counter>    //OK
```

The `ArrayList` class has numerous methods; we will use several in this lab.

- `boolean add(`*`Object`*` obj)`: Appends the specified element to the end of this list. You don't have to worry about the current size of the list, it will expand as you add elements, as necessary. Returns `true`, since an `ArrayList` always adds when asked to.
- `boolean remove(Object o)`: Removes a single instance of the specified element from this list, if it is present. Returns `true` if an element was removed.
- `int size()`: Returns the number of elements in this list.

The Object class whose object obj is a parameter to add() should be the class that you defined the ArrayList with. For instance, an ArrayListdeclared with

```
   ArrayList<String> myList = new ArrayList<String>();
```

would be able to

```
   myList.add("a string")        // OK
```

but not (for instance)

```
   myList.add(new Counter())     // this will fail with myList declaration above
   myList.add(5)                 // this will fail with myList declaration above
```

Task 1

Consider the ArrayListRunner class, where an ArrayList initialized as follows:

```
   ArrayList<String> words = new ArrayList<String>();
```

What is the result of executing the code below? Show all the results after every line of code.

```
   words.add("a");
   words.add("b");
   words.add("c");
   words.remove("b");
   words.add("d");
   words.remove("a");
   words.add("e");
   words.remove("b");
   words.add("d");
   words.remove("c");
   words.remove("d");
```

Task 2

Using inheritance, define a class TrackedArrayList that behaves just like the ArrayList class except that it has an extra method:

```
   public int maxSizeSoFar();
```

which returns the maximum number of elements in this list at any time since the list was constructed. For example, if maxSizeSoFar were called immediately after each call in the code sequence in Task 1 above, it would return the following values:

```
   1
   2
   3
   3
   3
   3
   3
   3
   4
   4
   4
```

Use the framework code in `arraylist/TrackedArrayList.java` to get started. You will see the special format to use when extending `ArrayList`, in order to refer to the class of the elements that it can contain.

Use the class in `ArrayListRunner.java` to test your work as you go, if you need to. You may find that the EduRide feedback tool gives you enough information to solve this task; if so, you'll need to comment on the specific feedback that you found useful.

Take advantage of the `ArrayList` methods and variables as much as possible, using calls to super, rather than reinventing the wheel.

Part 2: The game of Pig

Consider the game of 2-player Pig, described in Wikipedia as follows:

Each turn, a player repeatedly rolls a die until either a 1 is rolled or the player decides to "hold":

- If the player rolls a 1, he or she scores nothing, and it becomes the next player's turn.
- If the player rolls any other number, it is added to his or her turn total and the player's turn continues.
- If a player chooses to "hold", the player's turn total is added to his or her score, and it becomes the next player's turn.

The first player to score 100 or more points wins.

For example, the first player, Ann, begins a turn with a roll of 5. Ann could hold and score 5 points, but chooses to roll again. Ann rolls a 2, and could hold with a turn total of 7 points, but chooses to roll again. Ann rolls a 1, and must end her turn without scoring. The next player, Bob, rolls the sequence 4-5-3-5-5, after which he chooses to hold, and adds his turn total of 22 points to his score.

The program `PigSimulator.java` contains code to *simulate* playing of several games of Pig.
The main method initializes variables to keep track of how many games each player won. Then it repeatedly calls the `play1game` method to play a single game, alternating between the two players, stopping when one of the players reaches `100` points. The `play1turn` method simulates the roll of the die, and calls a method on the `Player` named `tallyRoll` to implement the various events listed above.

The `Player` class keeps track of the die rolls in a turn, and contains a method named `throwAgain` that represents a *strategy* for playing the game. By supplying subclasses for `Player` that override the `throwAgain` method, one can test different strategies against one another. A strategy may be based on a player's total score, the number of rolls made so far in the turn, or the total accumulated so far in the turn.

Task 3
The current program refers to `ConservativePlayer` and `RiskyPlayer` classes. Write these two classes, such that a `ConservativePlayer` will always choose to hold and a `RiskyPlayer` will never hold.

If you need to, create a class `PlayerRunner` in which you test your `ConservativePlayer` and `RiskyPlayer` classes.

Task 4
Think about which of these players will play a better game of Pig. Simulate several games (by executing `main` of `PigSimulator.java`) pitting them against each other, and report what happened. Was your prediction correct?

Task 5
Think of two other strategies and design players (by extending `Player`) that implement them. Create a class `PlayerRunner` (or augment the class if you created it for task 3) that tests your implementations.

Task 6

Test all four players against each other, which will take several simulations. Explain what happened. How did your strategies do? Did you expect to see the results you saw?

Part 3: (Assessment) Logic Check and Level of Understanding

Consider the following classes in a Java program, with the methods defined in each:

- `Pet`
  `eat()`
  `sleep()`
  - `Dog` *extends* `Pet`
    `goForAWalk()`
    `bark()`
    - `Pomeranian` *extends* `Dog`
      `yap()`
    - `GreatDane` *extends* `Dog`
      `woof()`
  - `Cat` *extends* `Pet`
    `meow()`
    - `Siamese` *extends* `Cat`
      `ignoreYou()`

For each of the questions below, answer with an `ArrayList` declaration with the most specific type, if any, that can work. For instance, the `ArrayList` that could contain `Cat` objects is `ArrayList<Cat>`

1. contain `Pomeranian` objects
2. contain `Pomeranian` and `Cat` objects
3. contain objects of any of the above types that you can call the `eat()` method on.
4. contain `GreatDane` objects that you will only call the `sleep()` method on
5. contain `Pomeranian` and `Siamese` objects
6. contain `Pomeranian` and `Siamese` objects that you will call `yap()` and `ignoreYou()`, as appropriate

What to hand in

When you are done with this lab assignment, submit all your work through CatCourses.

Before you submit, make sure you have done the following:

- Verified your solution with your TA or instructor.
- Included output for code in Lines 19 – 29 of `ArrayListRunner.java` in a Word or text file named `Part1`.
- Attached filled in `ArrayListRunner.java`, if you used it to test. If you didn't, describe what parts of the EduRide feedback tool was useful to you in writing `TrackedArrayList.java` in the file `Part1` you created for Task1.
- Included answers to questions and explanations for Tasks 4 and 6 in a Word or text file named `Part2`.
- Included answers to activities 10.5.1, 10.6.1 and 10.7.1, and Assessment questions (1 – 6) in a Word document or text file named `Part3`.
- Attached filled-in `TrackedArrayList.java` file, `ConservativePlayer.java`, `RiskyPlayer.java`, `PlayerRunner.java`, the source file(s) containing the two classes created for Task 5, `Part1`, `Part2` and `Part3` files
- Filled in your collaborator's name (if any) in the "Comments…" text-box at the submission page.