CS 61B Discussion 5: Inheritance II Fall 2015

1 Reduce

We'd like to write a method `reduce`, which uses a binary function to accumulate the values of a `List` of integers into a single value. `reduce` will need to take in an object that can operate (through a method) on two integer arguments and return a single integer. Note that `reduce` must work with a range of binary functions (addition and multiplication, for example). Fill in `reduce` and `main`, and define types for `add` and `mult` in the space provided.

```
import java.util.ArrayList;
import java.util.List;
public class ListUtils {
    /** Apply a function of two arguments cumulatively to the
    *    elements of list and return a single accumulated value. */
    static int reduce(BinaryFunction func, List<Integer> list) {
        if(list.size() == 0){
            return 0;
        }
        int soFar = list.get(0);
        for(int i = 1; i < list.size(); i++){
            soFar = func.apply(soFar, list.get(i));
        }
        return soFar;
    }
    public static void main(String[] args) {
        ArrayList<Integer> integers = new ArrayList<>();
        integers.add(2); integers.add(3); integers.add(4);
        Adder add = new Adder();
        Multiplier mult = new Multiplier();
        reduce(add, integers); //Should evaluate to 9
        reduce(mult, integers); //Should evaluate to 24
    }
}

//Add additional classes and interfaces below:
interface BinaryFunction {
    int apply(int x, int y);
}

class Adder implements BinaryFunction {
    public int apply(int x, int y){
        return x + y;
    }
}

class Multiplier implements BinaryFunction {
    public int apply(int x, int y){
        return x * y;
    }
}
```

We declare an interface `BinaryFunction` which our `Adder` and `Multiplier` classes can implement. Writing a common interface is important, because it allows us to write a `reduce` function that is capable of accepting many kinds of functions. Note that interface methods are `public` by default, so `apply` must be `public` in `Adder` and `Multiplier`.

2 Exception Handling

Below is an implementation of a `Farm` class. Its only field is an `ArrayList` of animals, and it has three methods: `getAnimal`, `addAnimal`, and `animalCount`. `getAnimal` takes an integer `i` as an argument and returns the i^{th} element of the farm's list of animals.

This implementation produces an `IndexOutOfBoundsException` when we try to get an animal at an index outside of the bounds of our internal `ArrayList`. This could be confusing to a user with no knowledge of our implementation of the Farm class. Instead, rewrite the `getAnimal` method so that it catches `IndexOutOfBoundsExceptions` and throws a more descriptive `IllegalArgumentException`.

```
1  import java.util.ArrayList;
2  public class Farm{
3      private ArrayList<Animal> animals = new ArrayList<>();
4
5      /** Adds an animal toAdd to the farm. */
6      void addAnimal(Animal toAdd){
7          animals.add(toAdd);
8      }
9
10     /** Takes an index between 0 and animalCount() - 1 (inclusive)
11      *  and returns the animal at that index. */
12     Animal getAnimal(int index){
13         return animals.get(index);
14     }
15
16     /** Returns the number of animals on the farm. */
17     int animalCount(){
18         return animals.size();
19     }
20 }
```

```
Animal getAnimal(int index){
    try {
        return animals.get(index);
    } catch(IndexOutOfBoundsException e){
        throw new IllegalArgumentException
            ("Must pass in an index between 0 and animalCount() - 1");
    }
}
```

When writing code that handles exceptions, the first step is to wrap the code that could cause an exception in a `try` clause. In this case, we wrap `return animals.get(index)` in a `try`, because it is at risk of throwing an `IndexOutOfBoundsException`. We associate exception handlers with a `try` block by following it with a `catch` block that handles whatever exception is specified by its argument. We specify that we are only looking to catch `IllegalArgumentExceptions`

so that we can still see other kinds of exceptions (what would happen if we caught `Exception e`?) Finally, we raise a custom exception with the `throw` keyword.

3 Comparator

We'd like to sort an `ArrayList` of animals into ascending order, by age. We can accomplish this using `Collections.sort(List<T> list, Comparator<? super T> c)`. Because instances of the `Animal` class (reproduced below) have no natural ordering, `sort` requires that we write an implementation of the `Comparator` interface that can provide an ordering for us. Note that an implementation of `Comparator` only needs to support pairwise comparison (see the `compare` method). Remember that we would like to sort in ascending order of age, so an `Animal` that is 3 years old should be considered "less than" one that is 5 years old.

```
public interface Comparator<T> {
    /** Compares its two arguments for order.
     *  Returns a negative integer, zero, or a positive integer if the first
     *  argument is less than, equal to, or greater than the second. */
    int compare(T o1, T o2);

    /** Indicates whether some other object is "equal to" this
    *   comparator. */
    boolean equals(Object obj);
}
```

```
import java.util.ArrayList;
import java.util.Collections;
public class Animal {
    protected String name, noise;
    protected int age;
    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
        this.noise = "Huh?";
    }
    /** Returns this animal's age. */
    public int getAge() {
        return this.age;
    }
    public static void main(String[] args) {
        ArrayList<Animal> animals = new ArrayList<>();
        animals.add(new Cat("Garfield", 4));
        animals.add(new Dog("Biscuit", 2));
        AnimalComparator c = new AnimalComparator(); //Initialize comparator
        Collections.sort(animals, c);
    }
}
```

```
import java.util.Comparator;
public class AnimalComparator implements Comparator<Animal> {
    public int compare(Animal o1, Animal o2) {
        return o1.getAge() - o2.getAge();
    }
}
```

We want to implement `Comparator<Animal>` because we are concerned with comparing objects of type `Animal`. Similarly, `compare` should take objects of type `Animal`. We would like younger animals to be considered "less than" older animals, so in `compare` we can simply return `o1.getAge() - o2.getAge()` (this way, we return a negative integer if o1 is younger than o2, zero if the two animals are the same age, and a positive integer if o2 is younger than o1). `Collections.sort`'s second argument is a `Comparator`, so we initialize our custom implementation on line `21` and pass it in on `22`.