CS 61B Discussion 5: Test Review Fall 2019

1 Inheritance Practice

```
public class Q {
   public void a() {
      System.out.println("Q.a");
   }
   public void b() {
      a();
   }
   public void c() {
      e();
   }
   public void d() {
      e();
   }
   public static void e() {
      System.out.println("Q.e");
   }
}
public class R extends Q {
   public void a() {
      System.out.println("R.a");
   }
   public void d() {
      e();
   }
   public static void e() {
      System.out.println("R.e");
   }
}
```

```
public class S {
   public static void main(String[]
      args) {
      R aR = new R();
      run(aR);
   }
   public static void run(Q x) {
      x.a();        /* R.a */
      x.b();        /* R.a */
      x.c();        /* Q.e */
      ((R)x).c();   /* Q.e */
      x.d();        /* R.e */
      ((R)x).d();   /* R.e */
   }
}
```

In `run`, write what gets printed next to each line.

`x.a()` will call the `a()` according to the variable's dynamic type.

`x.b()`, because `b()` is not overridden, will use the `b()` in `Q`. Then, `b()` selects which `a()` to run based on the variable's dynamic type.

`x.c()` runs `Q.c()`, which runs `Q.e()`. Note that e() is a static method, so it uses the static type to look up which function to call.

`((R)x).c()` makes the same series of calls. Again, e() is a static method, so it uses the static type to look up which function to call.

`x.d()` runs `R.d()`, which runs `this.e()`, this has a static type of R in R.d() so `R.e()` is ran.

`((R)x).d()` makes the same series of calls.

2 Reduce

We'd like to write a method `reduce`, which uses a `BinaryFunction` interface to accumulate the values of a `List` of integers into a single value. `BinaryFunction` can operate (through the `apply` method) on two integer arguments and return a single integer. Note that `reduce` can now work with a range of binary functions (addition and multiplication, for example). Write two classes `Adder` and `Multiplier` that implement `BinaryFunction`. Then, 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 {
    /** If the list is empty, return 0; if its one element, return it
     *  Otherwise, apply a function of two arguments cumulatively to the
     *  elements of list and return a single accumulated value. */
    public 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
    }
}

interface BinaryFunction {
    int apply(int x, int y);
}
//Add additional classes and interfaces below:
public class Adder implements BinaryFunction {
    public int apply(int x, int y) {
        return x + y;
    }
}
public 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`.

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.
Note: for this question, you do not need to worry about implementing `equals`.

```
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 {
    private String name;
    private int age;
    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
    }
    /** 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 Animal("Garfield", 4));
        animals.add(new Animal("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 `18` and pass it in on `19`.
For this question, you do not need to worry about implementing `equals`, as the `equals` method on a Comparator allows you to indicate that one comparator provides the same ordering as another comparator - an extremely rarely needed functionality.

4 Midterm Practice

```
public class PasswordChecker {
    /**
     * Returns true if the password is correct for the user
     */
    public boolean authenticate(String a, String b) {
        // Does something secret
    }

}

public class User {
    private String username;
    private String password;

    public void login(PasswordChecker p) {
        p.authenticate(username, password);
    }
}
```

Write a class containing a method `public String extractPassword(User u)` which returns the password of a given user `u`. You may not alter the provided classes. Note the access modifiers of instance variables.

```
public class PasswordExtractor extends PasswordChecker {
    String extractedPassword;
    public String extractPassword(User u) {
        u.login(this);
        return extractedPassword;
    }
    public boolean authenticate(String a, String b) {
        extractedPassword = b;
        return true; // or false. Just need to return something.
    }
}
```

By letting us subclass `PasswordChecker`, we can overwrite the `authenticate` method to capture the password in a local variable. By calling a user's `login` method and passing ourselves in, we can force the user to provide its password. Finally, we can return the extracted password. We could fix this security hole by making `PasswordChecker` no longer a public class.