CS 61B
Spring 2020

Small Group Tutoring
Section 4: Comparables, Exceptions, and MT1

Worksheet 6

1 String Comparisons

Suppose you are a freelance writer, and one of your clients asks you to write them a short thought piece about unusual applications of computer science. They have quite generously offered to pay you by the word, with the limitation that you may not exceed 10 pages. Obviously, you would now like to find the shortest words in the English language so that you can use as many of them as possible. To this end, you decide to write a little Java.

```
public interface StringComparator {
    /** Returns a negative number if s1 is 'less than' s2, 0 if 'equal,'
      * and a positive number otherwise. Null is considered less than
      * any String. If both inputs are null, return 0. */
    public int compare(String s1, String s2);
}
```

1. Write a new class `LengthComparator` that implements the `StringComparator` interface provided above. The `LengthComparator` should compare strings based on their lengths.
   Solution:

```
public class LengthComparator implements StringComparator {
    public int compare(String s1, String s2) {
        if ((s1 == null) && (s2 == null)) {
            return 0;
        }
        if (s1 == null) {
            return -1;
        }
        if (s2 == null) {
            return 1;
        }
        return s1.length() - s2.length();
    }
}
```

Notes:
1. We know that a negative number means the length of s1 is less than the length of s2, a positive number means the length of s1 is greater than the length of s2, and 0 signifies both strings are of the same length. So it's okay to just return the difference in length if both strings are not null. We do not have to explicitly return -1, 0, or 1.

2. Now suppose your client says that they will also pay you more for using words that occur later in the dictionary. Complete `DoubleComparator` to order strings first by length, and then in reverse alphabetical order. *Hint:* Use the `LengthComparator` class and Java Strings' `compareTo` method.
   Solution:

```
public class DoubleComparator implements StringComparator {
    public int compare(String s1, String s2) {
        LengthComparator lc = new LengthComparator();
        if (lc.compare(s1, s2) == 0) {
            return s2.compareTo(s1);
        } else {
            return lc.compare(s1, s2);
        }
    }
}
```

   Notes:
   1. We may have to order a list of many strings, but DoubleComparator is a tool to help us do this. This is why we implement DoubleComparator to compare two strings at a time, and not the whole list.

2 Exceptions Logged

One common software engineering strategy is to create log files that can be manually examined if something goes wrong. In the code below, the `writeToLog` method writes the given argument to some log.

```
public class LogFile {
    public static void printTenth(int[] a) {
        try {
            writeToLog(a[10]);
            if (a[10] == 11) {
                throw(new Exception("eleven"));
            }
        } catch (IndexOutOfBoundsException e) {
            writeToLog("No tenth item!");
            throw(e);
        } catch (Exception e) {
            writeToLog("The tenth item was an eleven!");
        }
        writeToLog("done");
    }
    public static void main(String[] args) {
        printTenth(new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10});
        printTenth(new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11});
        printTenth(new int[]{0, 1, 2, 3, 4});
        printTenth(new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10});
    }
}
```

What does the method `printTenth` do? What are the contents of the log file after the code is executed?

Solution: printTenth will attempt to write the element at the 10th index of the given array to the log. Otherwise, it will write a message and throw an exception if the number at the 10th index is equal to 11 or if there is no 10th element. It then writes ”done” to the log at the end.

Written to log:

10

done

11

The tenth item was an eleven!

done

No tenth item!

Exceptions thrown: `ArrayIndexOutOfBoundsException` // Note: This is caught by the first catch block since ArrayIndexOutOfBoundsException is a subclass of the class IndexOutOfBoundsException. Attempting to access an index out of bounds will throw an ArrayIndexOutOfBoundsException.

Notice “done” gets logged unless you preemptively return from the function, for example by throwing an

`Exception`.

Notes:
1. The `writeToLog` function is not explicitly defined anywhere, but we are told what it does in the problem statement.
2. The last call to `printTenth` never gets executed because there was `ArrayIndexOutOfBoundsException` thrown in the previous call to `printTenth`.
3. The third call to `printTenth` which throws `ArrayIndexOutOfBoundsException` is not caught in the second catch statement because catch blocks don't support a "waterfall effect." Once a catch block is entered, no other catch blocks are executed.
4. The third catch block actually handles our exception, so even though we got an error, we can continue in the method.

3 Access Modifiers (MT1 Review)

1. Complete the table below for where methods/variables with different access modifiers can be used.
Solution:

	Same Class	Same Package	Subclass	Other Package
`public`	Yes	Yes	Yes	Yes
`protected`	Yes	Yes	Yes	No
No modifier (Package protected)	Yes	Yes	No	No
`private`	Yes	No	No	No

2. Choosing the the correct access modifiers for your methods and variables is an important part of coding in Java. When should you use `public`? When should you use `private`?
Solution: You should consider making your methods and variables public if you want users of your API to be able to call and use the variables and methods. If you dont want users of your API to be able to do this, and you would like these methods and variables to only be used internally by you while coding then keep the access modifiers as private.

3. Now consider the following contrived example to see how access modifiers behave in practice. Assume all of the classes exist in the same package. For each of the lines marked with a comment determine whether the line will or will not work.

```
public class Beverage {
    private String drinkName;
    private int size;
    public class DrinkPromoter {
        public void promote() {
            System.out.println(drinkName + " is the best!"); // 1
        }
    }
}
public class Horchata extends Beverage {
    public int size;
```

```
        public String getDrinkName() {
            return drinkName; } // 2
        public int getSize() {
            return size;
        }
    }
    public class Drinks {
        public static void main(String[] args) {
            Beverage theBest = new Horchata();
            Horchata soGood = (Horchata) theBest;
            System.out.println(theBest.drinkName); // 3
            System.out.println(theBest.size); // 4
            System.out.println(soGood.size); // 5
        }
    }
```

Solution:

1. This works. Although `drinkName` is a private variable, `DrinkPromoter` is a inner class within `Beverage`, so this is not a problem.
2. This does not work. Although `Horchata` is a subclass of `Beverage`, `drinkName` is a private instance variable. For this to work `drinkName` must not be specified as private.
3. This does not work. `Drinks` is just another class, so accessing the private instance variable `drinkName` is not allowed.
4. This does not work. `theBest`'s static type is `Beverage` and its dynamic type is `Horchata`. We see that `Horchata` has a public instance variable also named `size`, but this will not compile as its static type `Beverage` does not.
5. This works as here the static and dynamic type of `soGood` is `Horchata`, so the compiler sees that `soGood` has a public instance variable `size`.

4 Dogs Woof

```
class Dog {
    void bark(Dog d) {
        System.out.println("bark");
    }
}

class Poodle extends Dog {
    void bark(Dog d) {
        System.out.println("woof");
    }

    void bark(Poodle p) {
        System.out.println("yap");
    }

    void play(Dog d) {
        System.out.println("no");
    }

    void play(Poodle p) {
        System.out.println("bowwow");
    }

    public static void main(String[] args) {
        Dog dan = new Poodle();
        Poodle pym = new Poodle();
```

```
1. dan.play(dan)

2. dan.play(pym)

3. pym.play(dan)

4. pym.play(pym)

5. pym.bark(dan)

6. pym.bark(pym)

7. dan.bark(dan)

8. dan.bark(pym)
```

Solution:

1) dan.play(dan) // Compile-error

2) dan.play(pym) // Compile-error

3) pym.play(dan) // no

4) pym.play(pym) // bowwow

5) pym.bark(dan) // woof

6) pym.bark(pym) // yap

7) dan.bark(dan) // woof

8) dan.bark(pym) // woof

5 Pizza Iterator (MT1 Review)

Artichoke's is overwhelmed by the number of hungry students in line at 12am. To make things more efficient, the owner has asked you to build a custom iterator that will aggregate all orders and print out the number of slices that should made for each kind of pizza.

The static menu array declared inside `MenuIterator` contains the three types of pizza offered that night.

```
static String[] menu = {"Artichoke", "Margherita", "Meatball"};
```

The input array passed into the constructor contains the list of orders.

```
int [] orders = { 0 , 2 , 1 , 0 , 1 , 0 };
```

Each order is represented by an integer that corresponds to the pizza's index in the menu array. For example, 0 represents an order of Artichoke pizza.

Fill in the code for `MenuIterator`, an iterator that takes in an `int[]` array representing orders at the restaurant and iterates over the aggregated results.

Given the input above, calls to `next()` would eventually return "Artichoke 3", "Margherita 2", "Meatball 1". Make sure your iterator adheres to standard iterator rules.

Solution:

```
public class MenuIterator implements Iterator {
    private static String[] menu = {"Artichoke", "Margherita", "Meatball"};
    private int[] order_counts = new int[3];
    private int index;

    public MenuIterator(Integer[] orders){
        //Initialize index and order_counts.
        for (int i=0; i < orders.length; i++){
            order_counts[orders[i]] += 1;
        }
        index = 0;
    }

    public boolean hasNext() {
        return index < menu.length;
    }

    public String next() {
        //Should return a string in the format "Artichoke 3".
        String order = menu[index] + " " + order_counts[index];
        index += 1;
        return order;
    }
}
```

6 IntSkip (MT1 Review)

Write a method intSkip that takes in an IntList L and changes every IntList's tail to point to an IntList n steps away from it, where n is the head attribute (the number stored inside) of each IntList.

Traverse starting from the first IntList (given by the pointer L) to the last IntList. If there is no valid IntList at n steps away from the current one, then make its tail the last IntList in L. At the last IntList, do not make any changes since the tail is null.

Solution:

```
public static void intSkip(IntList L) {
    IntList curr = L;
    Intlist next;
    while (curr != null) {
        next = curr.tail;
        int count = 0;
        IntList newTail = curr;
        while (count < curr.head && newTail.tail != null) {
            count += 1;
            newTail = newTail.tail;
        }
        curr.tail = newTail;
        curr = next;
    }
}
```

7 Supersize Me (MT1 Review)

Suppose we have the following class:

```
public class Restaurant {
    public String name;
    public String style;
    public Restaurant(String name, String style) {
         this.name = name;
         this.style = style;
    }
    public void order(String food) {
         System.out.println("One " + food + " coming right up!");
    }
}
```

Implement the class McDonalds such that it has the following specifications:

1. Is a subclass of the the Restaurant class.
2. Has a one argument constructor that takes in the location of the restaurant and saves it.
3. All instances should have the name ”McDonalds” and style ”Fast Food”.

4. Each instance has its own franchise number. Franchise numbers start at 1 and increase by 1.

5. Has an order method that uses Restaurant's order method but always prints out ”Would you like fries with that?” at the end as well.

Do not hide fields.

Solution:

```
public class McDonalds extends Restaurant {
    public String location;
    public int franchiseNumber;
    public static int lastFranchiseNumber = 0;

    public McDonalds(String location) {
        super("McDonalds", "Fast Food");
        this.location = location;
        lastFranchiseNumber += 1;
        franchiseNumber = lastFranchiseNumber;
    }

    public void order(String food) {
        super.order(food);
        System.out.println("Would you like fries with that?");
    }
}
```