Tabassum Fabiha
tf2478

1.1 - Enums

- compareTo()
  - Compares the order of the two objects
  - For for A and B of the Letter enum
    - A.compareTo(B) will return -1 because A will be written before B in the order
    - A.compareTo(A) will return 0 because A was written in the same position in the order of constants in Letter as itself
    - B.compareTo(A) will return 1 because B will be written after A in the order
- name() and toString() are initially the same, both returning a String representation of the name of the constant. However they differ in that toString() can be overwritten to give some other value while name() is final and cannot be overwritten.
- values() is a feature that returns an array of all the constants declared in an enum. The varargs construct of passing parameters and values() are similar in that the values that they contain are iterated in the same way.
  - In an enum Letter the values would be iterated in the following way

```
for (Letter letter : Letter.values()) {
   System.out.println(letter);
}
```

  - When passing parameters through varargs the iteration is done in the same way

```
public static void print(String ... letters) {
        for (String letter: letters)
                System.out.println(letter);
}
```

1.2 - Patterns: Builder

- The Law of Demeter says to only talk with you friends, which means you can only call the methods of objects that are global variables, your instance variables, passed to you as a parameter, or locally initialized. In most cases that would mean you should only call object.method(), thus using only one dot.
- Builder is not an exception to the Law of Demeter, but it is an exception to the “use only one dot” rule. It doesn’t violate the Law of Demeter because at no point does it call the method of an object that you did not create locally. Take the following Builder pattern example

```
new MainClass
        .Builder()
        .setA()
        .setB()
        .done()
```

When you call new MainClass.Builder() you get an instance of the class Builder. setA() is a method of the Builder class and it returns the same Builder instance that the method was called with. Same happens when you call setB(). It's called from the instance of Builder that was passed from setA() and returns that same instance again. Finally done() is called once again from this same instance of Builder and returns an instance of MainClass. Because we locally initialized that instance of the Builder class and keep calling different methods that it itself has, we've never called the methods of more than one object, thus not violating the Law of Demeter.

1.3 - Patterns: Iterator

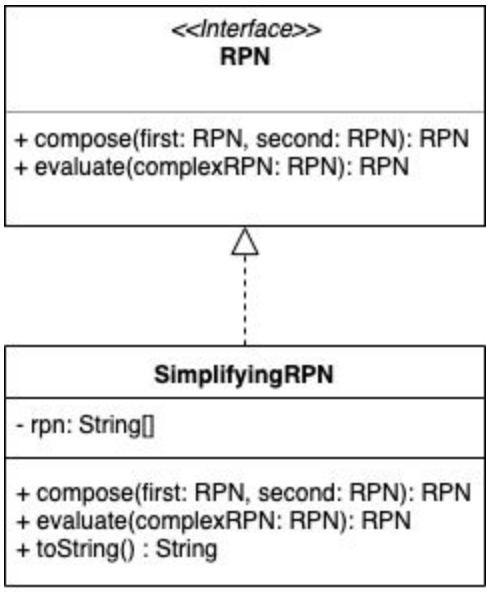
- This method will give you the closest index that fits the criteria of being twice from the end of the list as it is from the start of the list, since it's better to give an approximation than nothing at all if the conditions don't exactly match.
    - If list.size() % 3 == 0 then the index returned will be list.size() / 3 - 1.
    - If list.size() % 3 == 1 then the index returned will be list.size() / 3, exactly one third away from the beginning of the list.
    - If list.size() % 3 == 2 then the index returned will be list.size() / 3.
    - If list.size < 3 then we can never find one third of it and so we return -1.

```
public int oneThird(LinkedList<String> list) {
        ListIterator<String> iterForward = list.listIterator();
        ListIterator<String> iterBackward = list.listIterator( list.size() );

        while( iterForward.hasNext() ) {
                String ret = "";
                ret += iterForward.next();

                for ( int i = 0; i < 2; i++ ) {
                        if ( iterBackward.hasPrevious() ) {
                                if ( iterBackward.previousIndex() ==
                                iterForward.previousIndex() ) {
                                        System.out.println(ret);
                                        break;
                                }
                                else {
                                        ret += iterBackward.previous();

                                }
                        }
                }

                System.out.println(ret);
```

```
        }
    }
```

1.4 - Patterns: Composite

```
    public interface RPN {
        public RPN compose(RPN first, RPN second, String operation);
        public RPN evaluate(RPN complexRPN);
    }
```




```
    public RPN compose( RPN first, RPN second, String operation ) {
        RPN ret = new SimplifyingRPN( first.rpn.size() + second.rpn.size() + 1 );
        for (int i = 0; i < first.rpn.size(); i++)
            ret.rpn[i] = first.rpn[i];
        for (int i = 0; i < second.rpn.size(); i++)
            ret.rpn[ first.size() + i ] = second.rpn[i];
        return ret;
    }

    public RPN evaluate ( RPN complexRPN ) {
        int operIndex = -1;
        for ( int i = 0; i < complexRPN.rpn.size(); i++ ) {
            operIndex = "+-/*".indexOf(complexRPN.rpn[i]);
            if (operIndex >= 0)
                break;
        }
```

```
        if (operIndex == -1)
                return complexRPN;

        String first = complexRPN.rpn[operIndex - 2];
        String second = complexRPN.rpn[operIndex - 1;
        String operation = complexRPN.rpn[operIndex];

        String result = "";

        if (operation.equals("+"))
                result += Integer.parseInt(first) + Integer.parseInt(second);
        else if (operation.equals("-"))
                result += Integer.parseInt(first) - Integer.parseInt(second);
        else if (operation.equals("/"))
                result += Integer.parseInt(first) / Integer.parseInt(second);
        else
                result += Integer.parseInt(first) * Integer.parseInt(second);

        RPN ret = new SimplifyingRPN( complexRPN.rpn.size() - 2 );

        for (int i = 0; i < operIndex - 2; i++)
                ret.rpn[i] = complexRPN.rpn[i];
        ret.rpn[operIndex - 2] = result;
        for (int i = operIndex - 1; i < complexRPN.rpn.size() - 2; i++)
                ret.rpn[i] = complexRPN.rpn[i + 2];
        return evaluate(ret);
    }
```

1.5 - GUI

- Missing Constructors
    - Rectangle(Point p, int width, in height)
    - Rectangle(int x, int y, Dimension d)
- Point refers to a specific idea, in which the coordinates it contains relate to each other to provide a location. The contents of Dimension, width and height, on the other hand, have less of a correlation to each other.
- Dimension has the ability to take in doubles for the width and height through the method setSize(), so making the return type of the getters a double helps keep uniformity.
- Yes, the Builder Pattern would have been a better design choice for Rectangle since the constructors have different numbers and types of parameters depending on the situation and there is no natural progression for the parameters. For example when you're taking in four ints for the constructor how do you know for sure what each parameter represents? It leaves room for users to create bugs. Adding a Builder Pattern will allow this situation to be handled so that the location parameters and dimension parameters

will be taken in with different method calls that explicitly explain what is being done to the given parameters.

1.6 - UML