1. Immutable List Framework

Given an *IList* of Integer, write an *IListAlgo* called SumN to produce a new *IList* whose n-th element is the sum of the first n elements in the original *IList*. For examples,

- SumN over the empty list is the empty list.
- SumN over the list (3) is the list (3).
- SumN over the list (3 -5) is the list (3 -2).

Be sure to use *IListFactory* to manufacture *IList* objects and write helper visitors as anonymous inner classes. If you do not know how to write anonymous inner classes, then write the helpers as ordinary top-level classes and get at most 90% of the grade in this case.

```
public class SumN implements IListAlgo {

    private IListFactory _fac;

    /**
     * @param lf a non null IListFactory object.
     */
    public SumN(IListFactory lf) {
        _fac = lf;
    }

    public Object emptyCase(IEmptyList host, Object... nu) {
        return host;

    }


    public Object nonEmptyCase(INEList host, Object... nu) {
        return _fac.makeNEList(host.getFirst(),
(IList)host.getRest().execute(new IListAlgo() {
            // acc[0] is the accumulated sum of the preceding list.
            public Object emptyCase(IEmptyList h, Object... acc) {
                return h;             }

            // acc[0] is the accumulated sum of the preceding list.
            public Object nonEmptyCase(INEList h, Object... acc) {
                int newSum = (Integer)acc[0] + (Integer)h.getFirst();
                return _fac.makeNEList(newSum,
                                       (IList)h.getRest().execute(this, newSum));
            }
        }, host.getFirst()));       }
}
```

2. Polynomials

A polynomial of one variable, $p(x)$, is an expression of the form

$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$,

where n is a non-negative integer, a_k (k = 0.. n) are numbers, and a_n is a number not equal to 0.

n is called the degree (or order) of $p(x)$, a_k (k = 0.. n) are called the coefficients of $p(x)$, and a_n is called the leading coefficient of $p(x)$.

For examples:

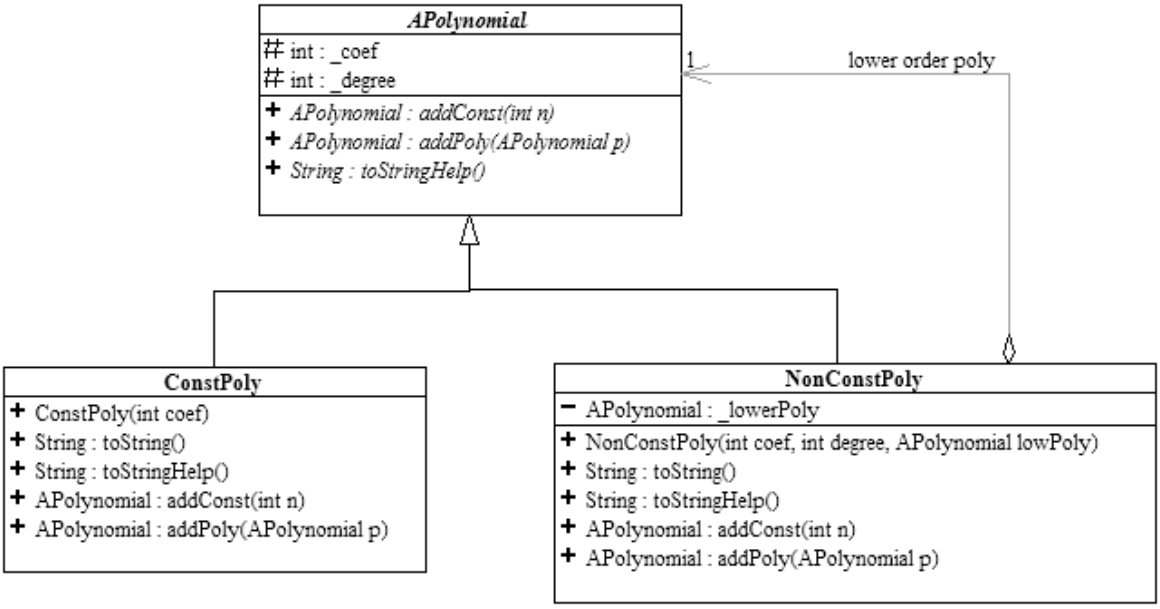
- $p(x) = 12x^5 - 3x^2 + 7$ is a polynomial of degree (order) 5 with leading coefficient 12. $-3x^2 + 7$ is called the lower order polynomial for $p(x)$.
- $p(x) = 7$ is a polynomial of degree 0 with leading coefficient 7. This is an example of a *constant* polynomial. It has no lower order polynomial.

We can describe polynomials in the following manner.

- There is an abstraction called polynomial.
- A constant polynomial is a polynomial with a leading coefficient and degree (order) 0.
- A non-constant polynomial is a polynomial with a leading coefficient, a positive degree, and a polynomial of lower order.

Polynomials can be viewed as having an **immutable** linear recursive structure and can be implemented using the composite design pattern. The operations on polynomial such as addition, multiplication, can be implemented as methods of the polynomial composite structure using the interpreter pattern. For the sake of simplicity, we shall only work with polynomials whose coefficients are integers.

Here is the UML class diagram showing the design of polynomials with integer coefficients.



Write the following methods for the polynomial classes. Feel free to add helper methods as needed.

a) Override the `toString()` method for the polynomial classes to compute and return the String representation of the host as described by the following.

- For constant polynomials, return the String representation of the coefficient (you may use `Integer.toString (int n)` to convert an int to a String).
- For non-constant polynomials, use ^ to denote exponentiation, and if the constant term is zero, do not show zero. For examples, $12x^5 - 3x^2 + 7$ prints as 12x^5 + -3x^2 + 7, and $12x^5 - 3x^2 + 0$ prints as 12x^5 + -3x^2.

<< students to complete >>

```
class ConstPoly extends APolynomial {

    public String toString() {
        return _coef + "";

    }

    public String toStringHelp() {
        if (_coef == 0) {
            return "";
        }
        else {
            return " + " + _coef;
        }

    }
/// other methods elided
}
```

```
Class NonConstPoly extends APolynomial {

    public String toString() {
        return _coef + "x^" + _degree + _lowerPoly.toStringHelp();

    }

    public String toStringHelp() {
       return " + " + _coef +  "x^" + _degree + _lowerPoly.toStringHelp();

         // or
         // return " + " + toString();

    }
/// other methods elided
}
```

b) Write the method `addConst(int n)` that adds a constant int to the target (i.e. receiver) polynomial and return the resulting polynomial.

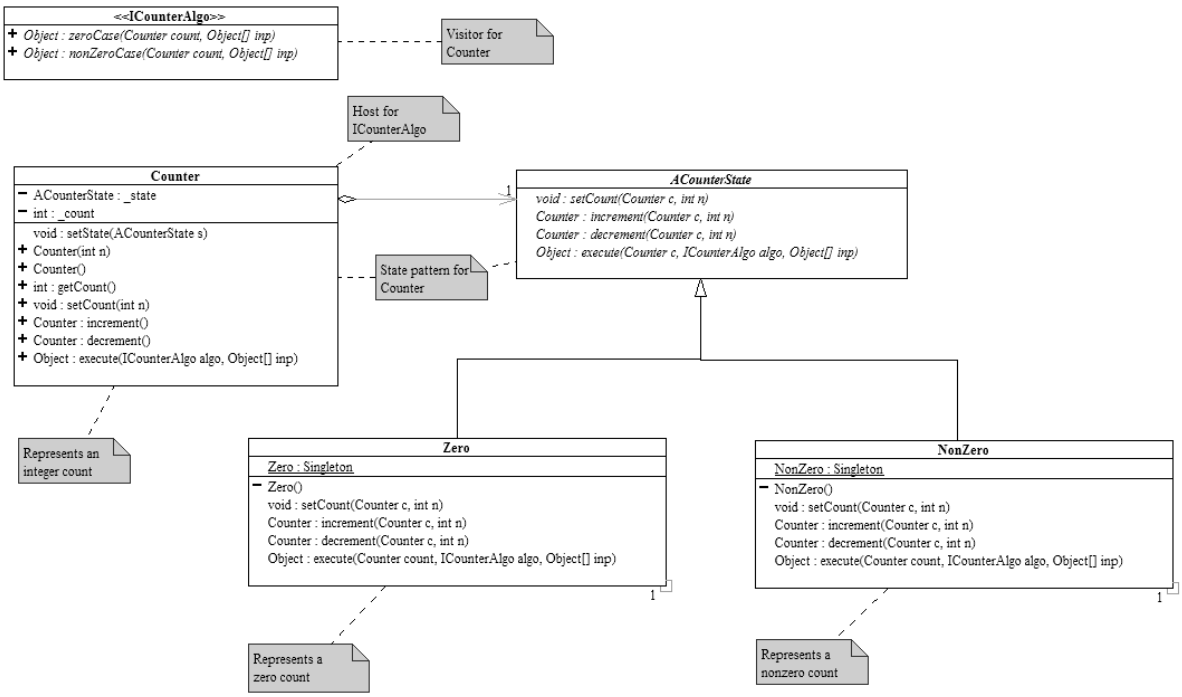
```
class ConstPoly:

    public APolynomial addConst(int n) {
        return new ConstPoly(_coef + n);

    }

class NonConstPoly:

    public APolynomial addConst(int n) {
        return new NonConstPoly (_coef, _degree, _lowerPoly.addConst(n));

    }
```

c) Write the method `add(APolynomial p)` that adds a polynomial p to the target (i.e. receiver) polynomial and return the resulting polynomial. Remember when you add polynomials, you can only add like terms, that is the terms with the same degree.

For example, if $p(x) = 12x^5 - 3x^2 + 7$ and $q(x) = 9x^7 - 4x^5 + 3x^2 - x$, then $p(x) + q(x) = 9x^7 + 8x^5 - x + 7$

```
Class ConstPoly:

    public APolynomial addPoly(APolynomial p) {
        return p.addConst(_coef);

    }

Class NonsConstPoly:

public APolynomial addPoly(APolynomial p) {
    // Check for the degrees of the operands and do the sum:
    if (p._degree < _degree) {
        return new NonConstPoly(_coef, _degree, _lowerPoly.addPoly(p));
    }
    else if (p._degree == _degree) {
        int newCoef = _coef + p._coef;
        APolynomial lowerSum = _lowerPoly.addPoly(((NonConstPoly)p)._lowerPoly);
        return 0 == newCoef? // IMPORTANT!
                   lowerSum:
                   new NonConstPoly(newCoef, _degree, lowerSum);
    }
    else {
        return new NonConstPoly(p._coef, p._degree, ((NonConstPoly)p)._lowerPoly.addPoly(this));
            // or the following:
            // return p.addPoly(this);

    }
}
```

3. Using an integer counter to keep track of the number of times we perform some computation is a very common programming technique. For example, to remove the last `n` elements from an `LRStruct`, we can perform the `RemLast` visitor (shown in the lecture) `n` times. We start with a counter n and each time we remove the last element from the `LRStruct`, we decrement n by one. When the counter reaches 0, we stop. Such an integer counter can be modeled as a class with two states, zero state and non-zero state, as shown in the UML class diagram below. This is an example of the state design pattern. Computations that make use of such a counter object are modeled as visitors on the counter object (see `ICounterAlgo` in the UML diagram).



Here is the code for the Counter class and its states.

```
public class Counter {

    private ACounterState _state;

    private int _count;  // the value of the Counter

    void setState(ACounterState s) {
        _state = s;
    }

    public Counter(int n) {
        _count = n;
        if (0 == n) {
            _state = Zero.Singleton;
        }
        else {
            _state = NonZero.Singleton;
        }
    }

    public Counter() { // Counter with initial count 0.
        this(0); // this is how to call the constructor Counter(n).
    }

    public int getCount() {
        return _count;
```

```
    }

    public void setCount(int n) {
        _count = n;
        _state.setCount(this, n);
    }

    public Counter increment() {
        _count++;
        return _state.increment(this, _count);
    }

    public Counter decrement() {
        _count--;
        return _state.decrement(this, _count);
    }

    public Object execute(ICounterAlgo algo, Object... inp) {
        return _state.execute(this, algo, inp);
    }
}

abstract class ACounterState {

    abstract void setCount(Counter c, int n);

    abstract Counter increment(Counter c, int n);

    abstract Counter decrement(Counter c, int n);

    abstract Object execute(Counter  c, ICounterAlgo algo, Object... inp);
}


class Zero extends ACounterState {
    final static Zero Singleton = new Zero();

    private Zero() {
    }

    void setCount(Counter c, int n) {
        if (0 != n) {
            c.setState(NonZero.Singleton);
        }
    }

    Counter increment(Counter c, int n) {
        c.setState(NonZero.Singleton);
        return c;
    }

    Counter decrement(Counter c, int n) {
        c.setState(NonZero.Singleton);
        return c;
    }

    Object execute(Counter count, ICounterAlgo algo, Object... inp) {
        return algo.zeroCase(count, inp);
    }
}

class NonZero extends ACounterState {
    static NonZero Singleton = new NonZero();

    private NonZero() {
    }

    void setCount(Counter c, int n) {
        if (0 == n) {
            c.setState(Zero.Singleton);
        }
```

```
    }

    Counter increment(Counter c, int n) {
        if (0 == n) {
            c.setState(Zero.Singleton);
        }
        return c;
    }

    Counter decrement(Counter c, int n) {
        if (0 == n) {
            c.setState(Zero.Singleton);
        }
        return c;
    }

    Object execute(Counter count, ICounterAlgo algo, Object... inp) {
        return algo.nonZeroCase(count, inp);
    }
}


public abstract interface ICounterAlgo {  // the visitor to Counter

    public abstract Object zeroCase(Counter count, Object... inp);

    public abstract Object nonZeroCase(Counter count, Object... inp);
}
```

a) Write an `ICounterAlgo` called `RemLastNCounterAlgo` that takes as input an `LRStruct` and applies the `RemLast` visitor for `LRStruct` `n` times to remove the last `n` elements of the `LRStruct`. Removing 0 elements from the list leaves the list unchanged. If `n` is greater than the number of elements in the `LRStruct`, an exception should be thrown. Recall from the lecture that `RemLast` is an `LRStruct` visitor that removes the last element of the `LRStruct` host. If the host is empty then it throws an exception.

```
class RemLastNCounterAlgo implements ICounterAlgo {
    public static final RemLastNCounterAlgo Singleton = new RemLastNCounterAlgo();
    private RemLastNCounterAlgo() {
    }

    /**
     * @param count is zero
     * @param inp[0] LRStruct
     */
    public Object zeroCase(Counter count, Object... inp) {
        return inp[0];

    }

    /**
     * @param count is not zero.
     * @param inp[0] LRStruct
     */
    public Object nonZeroCase(Counter count, Object... inp) {
        // hint: remove the last element, decrement the count and repeat recursively.
        LRStruct list = (LRStruct)inp[0];
        list.execute(RemLast.Singleton); // there is a side-effect here on list.
        return count.decrement().execute(this, list);
    }
}
```

b) Write an LRStruct visitor called RemLastN that takes as input an integer n and removes the last n elements from the host LRStruct. This visitor should make use of the RemLastNCounterAlgo done in part a) above. If n is greater than the number of elements in the LRStruct, an exception should be thrown. NOTE: this part is a visitor to LRStruct in contrast with part a) which is a visitor to Counter.

```
/**
 * Removes the last n element of a list.
 * Removing 0 elements from the list means leave the list unchanged
 * Uses an ICounterAlgo to apply the RemLast algo on LRStruct n times.
 */
public class RemLastN implements IAlgo {

    public static final RemLastN Singleton = new RemLastN();
    private RemLastN() {
    }

    /**
     * @param host is empty
     * @param n Integer >= 0 the number of elements to remove.
     * @return host.
     */
    public Object emptyCase(LRStruct host, Object... n) {

        return new Counter((Integer)n[0]).execute(RemLastNCounterAlgo.Singleton, host);
    }

    /**
     * @param host non-empty
     * @param n Integer >= 0
     * @return host.
     */
    public Object nonEmptyCase(LRStruct host, Object... n) {

        return new Counter((Integer)n[0]).execute(RemLastNCounterAlgo.Singleton, host);
    }
}
```

4. There is an abstract notion of a robot. Robots are manufactured by a factory. Each robot remembers its current location, its factory of manufacture and has a unique serial number. (In other words, no two robots, regardless of factory have the same serial number.)
   - Robots have a behavior "go to a location", as well as "getter" methods to return their location, their factory of manufacture and their serial number.
   - A factory has a location and knows all of the robots it has made. Factories have two behaviors: "make a robot" and "recall robots".

   Occasionally, the government is given a failed robot and discovers a design defect in that robot. At which point, the government requests the factory that made the robot to recall all robots ever manufactured by that factory. A recall is defined as directing all robots made by that factory to go to the location of the factory. Your task is to produce an object-oriented design simulating robots and factories. Express your design in Java code. Add comments only where you feel necessary to explain your code. Do not bother to comment getter methods. Note that a factory may need additional data structures to keep track of its robots.