```
/**
  * This is my code! Its goal is to create a term, containing a coefficient
  * and an exponent
  * CS 312 - Assignment 5
  * @author Mari Sisco
  */

public class Term implements Comparable<Term>
{
  protected double coefficient;
  protected Integer exponent;

 /*
  * purpose: construct a Term
  * input: coefficient and exponent
  * result: intialized Term
  */
  public Term(double coefficient, int exponent)
  {
    this.coefficient = coefficient;
    this.exponent = exponent;
  }

 /*
  * purpose: compares the exponent of two terms
  * input: term we want to compare this term to
  * result: -1, 0 or 1, depending on which term is bigger, or if they are
  *         equal.
  */
  @Override
  public int compareTo(Term newTerm)
  {
    return this.exponent.compareTo(newTerm.exponent);
  }

  /*
   * purpose: produce a human-readable Term
   * input: none
   * result: String with representation of this Term
   */
  public String toString()
  {
    if(exponent == 1)
      return coefficient + "x";
    if(exponent == 0)
      return coefficient + "";

    return coefficient + "x^" + exponent;
  }

}
```



```
/**
  * This is my code! Its goal is to create a polynomial, where addition and
  * multiplication may occur
  * CS 312 - Assignment 5
  * @author Mari Sisco
  */

import java.util.Deque;
import java.util.ArrayDeque;
import java.util.List;
import java.util.LinkedList;
import java.util.Iterator;
import java.lang.Iterable;

public class Poly
{
  //protected Deque<Term> polynomial;
  protected List<Term> polynomial;

  /*
   * purpose: construct a polynomial, an array deque or a linked list of terms
   * input: none
   * result: initialized polynomial
   */
  public Poly()
  {
    //polynomial = new ArrayDeque<>();
    polynomial = new LinkedList<>();
  }

  /*
   * purpose: insert a Term to this polynomial at the end
   * input: none
   * result: updated polynomial
   */
  public void insertAtEnd(Term t)
  {
    polynomial.add(t);
  }

  /*
   * purpose: sorts polynomial from Term with the highest exponent to term with
   *          the lowest exponent
   * input: none
   * result: sorted polynomial
   */
  public Poly sortPoly()
  {
     Poly result = new Poly();
     for (Integer i = degree(); i>= 0 ; i--)
     {
       for (Term t : polynomial)
       {
         if (t.exponent.compareTo(i) == 0)
           result.insertAtEnd(t);
       }
     }
     return result;
  }

  /*
   * purpose: adds the terms of this polynomial with terms of another polynomial
   * input: polynomial we want to add
   * result: a polynomial, result, of addition
   */
  public Poly add(Poly newPoly)
  {
    Poly result = new Poly();
```

```
    Poly polynomial = sortPoly();
    newPoly = newPoly.sortPoly();

    Iterator<Term> i1 = this.polynomial.iterator();
    Iterator<Term> i2 = newPoly.polynomial.iterator();

    if (polynomial == null)
      return newPoly;

    else if (newPoly == null)
        return polynomial;
    else
    {
      while(i1.hasNext() && i2.hasNext())
      {
        Term t1 = i1.next();
        Term t2 = i2.next();

        if( t1.exponent.compareTo(t2.exponent)  == 0)
        {
      Term t3 = new Term (t1.coefficient + t2.coefficient, t1.exponent);
          result.insertAtEnd(t3);
        }
        else if( t1.exponent.compareTo(t2.exponent) == 1)
        {
          result.insertAtEnd(t1);
      result.insertAtEnd(t2);
        }
        else if (t1.exponent.compareTo(t2.exponent) == -1)
        {
          result.insertAtEnd(t2);
      result.insertAtEnd(t1);
        }
      }

      //inserting remaining Terms
      if (i1.hasNext())
      {
        while(i1.hasNext())
        {
          result.insertAtEnd(i1.next());
        }
      }

      else if(i2.hasNext())
      {
        while(i2.hasNext())
        {
          result.insertAtEnd(i2.next());
        }
      }
    }

    result = result.sortPoly();
    result = result.addingLikeTerms();

    return result;
  }

  /*
   * purpose: combining terms with the same exponent together
   * input: none
   * output: a polynomial with all like terms together
   */
  public Poly addingLikeTerms()
  {
    Poly result = new Poly();

    Iterator<Term> i1 = this.polynomial.iterator();
```



```
    Iterator<Term> i2 = this.polynomial.iterator();

    i2.next();

    while(i1.hasNext() && i2.hasNext())
    {
      Term t1 = i1.next();
      Term t2 = i2.next();

      if(t1.exponent.compareTo(t2.exponent) == 0)
      {
        result.insertAtEnd(new Term(t1.coefficient + t2.coefficient,
                t1.exponent));
        if(i1.hasNext())
    {
      i1.next();
    }
    if(i2.hasNext())
    {
          i2.next();
    }
      }
      else
        result.insertAtEnd(t1);
    }

    if(i1.hasNext())
    {
      result.insertAtEnd(i1.next());
    }
    else if (i2.hasNext())
    {
      result.insertAtEnd(i2.next());
    }

    return result;
  }


  /*
   * purpose: multiply two polynomials together
   * input: polynomial we want to multiply by
   * result: the product of this polynomial and input polynomial
   */
  public Poly multiply(Poly newPoly)
  {
    Poly polinomial = sortPoly();
    newPoly = newPoly.sortPoly();

    Poly result = new Poly();
    Poly finalResult = new Poly();

    for (Term t1 : newPoly.polynomial)
    {
      Iterator<Term> itr = this.polynomial.iterator();
      while(itr.hasNext())
      {
    Term t2 = itr.next();
    Term product = new Term(t1.coefficient * t2.coefficient ,
            t1.exponent + t2.exponent);
    result.insertAtEnd(product);
      }

      finalResult = finalResult.add(result);
      result.polynomial.clear();
    }

     finalResult = finalResult.sortPoly();
     finalResult = finalResult.addingLikeTerms();
```

```
    return finalResult;
  }

  /*
   * purpose: find the highest degree of all terms in polynomial
   * input: none
   * result: int with the highest degree
   */
  public int degree()
  {
    int highestDegree = 0;
    for( Term t : polynomial)
    {
      if(t.exponent > highestDegree)
        highestDegree = t.exponent;
    }
    return highestDegree;
  }

  /*
   * purpose: produce a human-readable polynomial
   * input: none
   * result: String with representation of this polynomial
   */
  @Override
  public String toString()
  {
    String s = "";
    for(Term t: polynomial)
    {
      s += " " + t + "+";
    }
    if (s.length() == 0)
      return s;
    else
      return s.substring(0, s.length()  - 1);
  }

  /*
   * purpose: print a polynomial
   * input: none
   * result: String from toString()
   */
  public String print()
  {
    return toString();
  }
}
```



```
/**
  * This is Binkley's code!  It's goal is to test the polynomial code.
  * CS 312 - Assignment 5
  * @author Binkley
  * @version 1.2  10/15/2022
  */

// Squidward's test driver for Assignment 5 "Poly want a nomial?"

/*
 * purpose: test the polynomial class (including a stress test)
 * input:   nada
 * result:  output from a series of polynomial additions and multiplications
 */
class Tester
{
  public static void main(String [] args)
  {
    Poly poly1 = new Poly();
    System.out.println("poly = " + poly1);

    poly1.insertAtEnd(new Term(5,2));
    System.out.println("poly = " + poly1);

    poly1.insertAtEnd(new Term(8,0));
    System.out.println("poly = " + poly1);

    System.out.println("poly^2 = " + poly1.multiply(poly1));

    Poly poly2 = new Poly();
    poly2.insertAtEnd(new Term(3,1));
    poly1 = poly1.add(poly2);
    System.out.println("poly = poly + 3x = " + poly1);

    Poly p = poly1.add(poly1);
    System.out.println("p = poly + poly = " + p);

    Poly p2 = new Poly();
    p2.insertAtEnd(new Term(1,5));
    System.out.println("p2 = " + p2);
    System.out.println("p + p2 = " + p.add(p2));
    System.out.println("p2 + p = " + p2.add(p));

    Term t = new Term(2, 1);
    System.out.println("t = " + t);
    Poly testing = new Poly();
    testing.insertAtEnd(t);
    Poly pp = p.multiply(testing);
    System.out.println("p*t = " + p + " * " + t + " = " + pp);

    long startTime, time, memoryUsed;

    startTime = System.currentTimeMillis();
    time = startTime;
    memoryUsed = Runtime.getRuntime().totalMemory()
                 - Runtime.getRuntime().freeMemory();
    System.out.println("start stress test starts at " + startTime +
      "ms, using " + memoryUsed/(1024*1024) + "Mb");

    // stress test!
    int K1 = 16000000;
    int K2 = 19000;
    int K3 = 9;

    // at this point you will get better data if you only perform one of
    // the following stress tests at a time!
    // consider adding support for command-line selection:
    //  System.out.println("stress test usage: java Tester add | term | poly");
```

```
   /*
    for(int i=0; i<K1; i++)
      pp = pp.add(pp);

    time = System.currentTimeMillis();
    memoryUsed = Runtime.getRuntime().totalMemory()
                 - Runtime.getRuntime().freeMemory();
    System.out.println("post add test took " + (time - startTime)
      + "ms, using " + memoryUsed/(1024*1024) + "Mb");
    System.out.println("pp.degree = " + pp.degree() );
*/

    /*
    startTime = time;
    pp = p.multiply(testing);
    for(int i=0; i<K2; i++)
       pp = pp.multiply(testing);

    time = System.currentTimeMillis();
    memoryUsed = Runtime.getRuntime().totalMemory()
                 - Runtime.getRuntime().freeMemory();
    System.out.println("post multiply term test took " + (time - startTime)
      + "ms, using " + memoryUsed/(1024*1024) + "Mb");
    System.out.println("pp.degree = " + pp.degree() );
   */

    startTime = time;
    pp = p.multiply(testing);
    for(int i=0; i<K3; i++)
      pp = pp.multiply(pp);

    time = System.currentTimeMillis();
    memoryUsed = Runtime.getRuntime().totalMemory()
                 - Runtime.getRuntime().freeMemory();

    System.out.println("post multiply poly test took " + (time - startTime)
      + "ms, using " + memoryUsed/(1024*1024) + "Mb");
    System.out.println("pp.degree = " + pp.degree() );

  }
}
```