Feb 19, 04 18:09 **problem1.txt** Page 1/1

```
# problem1.txt
# Dan R. K. Ports <drkp@mit.edu>
# 6.170 PS2-1, 2004/02/12
# $Id: problem1.txt,v 1.1 2004/02/19 23:09:40 drkp Exp $

1. The one-line comments clarify the math being used by the
   computations that generate the return values of these functions.

2. A NaN is represented by a denominator of zero. RatNum.add and
   RatNum.mul generate new RatNums that have denominator
   (this.denom * arg.denom), so if either number is NaN, then the
   denominator of the returned value will be zero, i.e. NaN. This is
   not true of a div call, so it is necessary to explicitly check
   whether the argument is NaN.

3. RatNum.parse is static because it generates a new object from only
   a string; it does not depend on the previous state of a RatNum
   object. This could also have been implemented as a constructor.

4. At the beginning of the constructor, the fields have not been
   initialized, so they should not be expected to satisfy the runtime
   invariant.

5. The two-argument constructor would be simpler: it wouldn't need to
   reduce the input.
   The checkRep method would be simplified to meet the weaker
   invariant.
   unparse() would be more complicated because it would need to
   reduce the number before outputting it.
   equals() and hashCode() would need to be made more complex: they
   would need to take into account that two rationals may be equal
   but in different unreduced forms.

6. The add/sub/mult/div calls need to return a new RatNum
   object. They are not allowed to modify the existing (immutable)
   object.
```

+2 +2 +4 +2 +2 +2 +4 +4 +2 +2 free

24/24 nice, clear answers!

Feb 19, 04 17:50 **problem4.txt** Page 1/1

```
# problem4.txt
# Dan Ports <drkp@mit.edu>
# 6.170 PS2-4, 2004/02/19
# $Id: problem4.txt,v 1.1 2004/02/19 22:50:46 drkp Exp $

0
1
x
x^2
```

Overall: 99/100

```
# problem5.txt
# Dan Ports <drkp@mit.edu>
# 6.170 PS2-5, 2004/02/19
# $Id: problem5.txt,v 1.1 2004/02/19 23:05:51 drkp Exp $

The provided implementation of BasicList was tested using the
BasicListTest test suite. The following errors were observed:

testEqualsFalse() failed:
  The comparison (with equals()) of two non-equal BasicLists returned
  true. equals() operated correctly when tested on test cases that
  should have returned true, and comparisons to null and objects of
  different runtime types. Perhaps equals() may be returning true for
  all comparisons of two BasicLists.

testContains() failed:
  The contains() method does not correctly return whether an object is
  contained in the list.

testIndexOfNull() failed:
  indexOf() threw a NullPointerException when checking the index of
  null in a BasicList; it should have returned a correct value.

testRemoveReturnValue() failed:
  remove() correctly removed objects from the BasicList, but it did
  not return the objects it removed. It returned null instead.

testToArrayOversize() failed:
  When called with an array as argument that has more elements than
  the list, toArray() should return the same array with the elements
  from the list placed in the first spaces followed by a null
  element. However, when a BasicList of length 3 was placed in an
  array of length 5 using toArray, the returned array had length 3 --
  the length of the BasicList, which is not correct.
```

5/5

staff tests: 25/25 excellent!


```
package ps2;

import junit.framework.TestCase;
import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests extends TestCase {
  public static Test suite() {
    TestSuite suite = new TestSuite(RatNumTest.class);
    suite.addTest(RatPolyTest.suite());
    suite.addTest(RatPolyStackTest.suite());
    suite.addTest(new TestSuite(BasicListTest.class));
    return suite;
  }
}
```

```
package ps2;

import junit.framework.*;
import java.lang.Character;
import java.util.ArrayList;
import java.util.List;
import java.util.Iterator;


/** This class contains a set of test cases that can be used to
    test the implementation of the BasicList class.
    <p>
*/
public class BasicListTest extends TestCase {

    BasicList emptyList = null;

    public void setUp() {
        //this method can be used to initialize variables that can be
        //re-used for other tests
        emptyList = new BasicList();
    }

    // make an empty BasicList
    private BasicList makeList()
    {
        return new BasicList();
    }

    // make a BasicList containing the characters in a string
    // Based on the helper functions from RatPolyStackTest
    // Requires the add() method to work.
    private BasicList makeList(String str)
    {
        BasicList l = new BasicList();

        for (int i = 0; i < str.length() ; i++)
        {
            l.add(new Character(str.charAt(i)));
        }

        return l;
    }


    // compare a BasicList of characters to a string listing the
    // expected characters. Requires the get() and size() methods
    // to work.
    private void assertListIs(BasicList l, String s)
    {
        assertTrue(l.size() == s.length());

        for (int i = 0; i < s.length(); i++)
        {
            Object o = l.get(i);
            assertTrue(o instanceof Character);
            char c1 = ((Character) o).charValue();
            char c2 = s.charAt(i);
            String errStr = "Elem(" + i + ") was " + c1
                + " not " + c2 + " (" + s + ")";

            assertTrue(errStr, c1 == c2);
        }
    }


    public void testEqualsNullAndOtherType() {
        assertTrue(emptyList.equals(new BasicList()));
```



```
        //test that null objects and objects of other run-time types
        //return false
        assertTrue(!emptyList.equals(null));
        assertTrue(!emptyList.equals("listString"));
    }

    public void testEqualsTrue()
    {
        BasicList l = makeList("abcde");
        assertTrue(l.equals(l));
        assertTrue(l.equals(makeList("abcde")));
    }

    public void testEqualsFalse()
    {
        BasicList l = makeList("abcde");
        assertTrue(!l.equals(emptyList));
        assertTrue(!l.equals(makeList("abcd")));
        assertTrue(!l.equals(makeList("abcdef")));
        assertTrue(!l.equals(makeList("xyzzy")));
    }

    public void testHashCode()
    {
        BasicList l = makeList("abcde");
        assertEquals(l.hashCode(), (makeList("abcde")).hashCode());

        assertTrue(l.hashCode() != (makeList("abcd")).hashCode());
        assertTrue(l.hashCode() != (makeList("abcdef")).hashCode());
        assertTrue(l.hashCode() != (makeList("xyzzy")).hashCode());
        assertTrue(l.hashCode() != emptyList.hashCode());
    }


    // zero-arg constructor is already tested by testEquals(); we
    // don't need to test it explicitly

    public void testOneArgConstructor()
    {
        List source = new ArrayList();
        BasicList l;

        l = new BasicList(source);
        assertTrue(l.equals(emptyList));

        source.add(new Character('a'));
        source.add(new Character('b'));
        source.add(new Character('c'));

        l = new BasicList(source);

        assertListIs(l, "abc");

        try
        {
            l = new BasicList(null);
            assertTrue(false);

        }
        catch (NullPointerException e)
        {

        } // end of try-catch

    }

    public void testAdd()
    {
        BasicList l = makeList("abc");
```

```
        assertListIs(l, "abc");

        assertTrue(l.add(new Character('d')));
        assertListIs(l, "abcd");
    }

    public void testAddNull()
    {
        BasicList l = makeList("abcd");
        l.add(null);
        assertTrue(l.get(4) == null);

    }

    public void testAddIndexed()
    {
        BasicList l = makeList("abcd");

        l.add(2, new Character('e'));
        assertListIs(l, "abecd");
        l.add(0, new Character('f'));
        l.add(6, new Character('g'));
        assertListIs(l, "fabecdg");


        try
        {
            l.add(-1, new Character('z'));
            fail("Should throw an IndexOutOfBoundsException");
        }
        catch (IndexOutOfBoundsException e)
        {
            assertListIs(l, "fabecdg");
        } // end of try-catch

        try
        {
            l.add(9, new Character('z'));
            fail("Should throw an IndexOutOfBoundsException");
        }
        catch (IndexOutOfBoundsException e)
        {
            assertListIs(l, "fabecdg");
        } // end of try-catch

    }


    public void testContains()
    {
        BasicList l = new BasicList();
        Character a = new Character('a');
        Character b = new Character('b');
        Character c = new Character('c');
        Character d = new Character('d');

        assertTrue(!l.contains(a));

        l.add(a);
        l.add(b);
        assertListIs(l, "ab");
        assertTrue(l.contains(a));
        assertTrue(l.contains(b));
        assertTrue(!l.contains(c));
        assertTrue(!l.contains(d));
        assertTrue(!l.contains(new String("test")));

        l.add(c);
        l.add(d);
```



```
        assertListIs(l, "abcd");
        assertTrue(l.contains(a));
        assertTrue(l.contains(b));
        assertTrue(l.contains(c));
        assertTrue(l.contains(d));
        assertTrue(!l.contains(new String("test")));

        l.add(a);
        l.add(b);
        assertListIs(l, "abcdab");
        assertTrue(l.contains(a));
        assertTrue(l.contains(b));
        assertTrue(l.contains(c));
        assertTrue(l.contains(d));
        assertTrue(!l.contains(new String("test")));
    }


    public void testGet()
    {
        // assertListIs uses get(), so we can use it for
        // testing

        BasicList l = new BasicList();
        assertListIs(l, "");
        l = makeList("abcde");
        assertListIs(l, "abcde");

        try
        {
            l.get(-1);
            fail("Should throw an IndexOutOfBoundsException");
        }
        catch (IndexOutOfBoundsException e)
        {
            assertListIs(l, "abcde");
        } // end of try-catch

        try
        {
            l.get(5);
            fail("Should throw an IndexOutOfBoundsException");
        }
        catch (IndexOutOfBoundsException e)
        {
            assertListIs(l, "abcde");
        } // end of try-catch
    }


    public void testIndexOf()
    {
        BasicList l = makeList("abc");
        assertEquals(l.indexOf(new Character('a')), 0);
        assertEquals(l.indexOf(new Character('b')), 1);
        assertEquals(l.indexOf(new Character('c')), 2);
        assertEquals(l.indexOf(new Character('d')), -1);
        assertEquals(l.indexOf(new String("test")), -1);
    }

    public void testIndexOfNull()
    {
        BasicList l = makeList("abc");
        assertEquals(l.indexOf(null), -1);
        l.add(null);
        assertEquals(l.indexOf(null), 3);
    }
```

```
public void testIsEmpty()
{
    BasicList l = new BasicList();

    assertTrue(l.isEmpty());
    l.add(null);
    assertTrue(!l.isEmpty());
    l = makeList("abc");
    assertTrue(!l.isEmpty());
}


public void testRemoveEffects()
{
    BasicList l = new BasicList();
    Character a = new Character('a');
    Character b = new Character('b');
    Character c = new Character('c');

    l.add(a);
    l.add(b);
    l.add(c);
    assertListIs(l, "abc");

    l.remove(1);
    assertListIs(l, "ac");
    l.remove(1);
    assertListIs(l, "a");
    l.remove(0);
    assertListIs(l, "");
}


public void testRemoveReturnValue()
{
    BasicList l = new BasicList();
    Character a = new Character('a');
    Character b = new Character('b');
    Character c = new Character('c');

    l.add(a);
    l.add(b);
    l.add(c);
    assertListIs(l, "abc");

    assertSame(b, l.remove(1));
    assertSame(c, l.remove(1));
    assertSame(a, l.remove(0));
}


public void testRemoveOOB()
{
    BasicList l = new BasicList();
    Character a = new Character('a');
    Character b = new Character('b');
    Character c = new Character('c');

    l.add(a);
    l.add(b);
    l.add(c);
    assertListIs(l, "abc");

    try
    {
        l.remove(-1);
        fail("Should throw an IndexOutOfBoundsException");
    }
    catch (IndexOutOfBoundsException e)
```



```
    {
        assertListIs(l, "abc");
    } // end of try-catch

    try
    {
        l.remove(3);
        fail("Should throw an IndexOutOfBoundsException");
    }
    catch (IndexOutOfBoundsException e)
    {
        assertListIs(l, "abc");
    } // end of try-catch
}


public void testSize()
{
    BasicList l = makeList();
    assertEquals(l.size(), 0);
    l = makeList("abc");
    assertEquals(l.size(), 3);
    l.add(new Character('a'));
    assertEquals(l.size(), 4);
    l.add(null);
    assertEquals(l.size(), 5);
    l.remove(3);
    assertEquals(l.size(), 4);
}


public BasicListTest(String name) {
    super(name);
}


public void testIterator()
{
    // The spec does not specify whether the returned
    // iterators support the remove() operation, so we
    // don't test that.

    BasicList l = new BasicList();
    Character a = new Character('a');
    Character b = new Character('b');
    Character c = new Character('c');

    l.add(a);
    l.add(b);
    l.add(c);
    assertListIs(l, "abc");

    Iterator iter = l.iterator();

    assertTrue(iter.hasNext());
    assertSame(a, iter.next());
    assertTrue(iter.hasNext());
    assertSame(b, iter.next());
    assertTrue(iter.hasNext());
    assertSame(c, iter.next());
    assertTrue(!iter.hasNext());
}


public void testToArrayNoArgs()
{
    BasicList l = new BasicList();
    Character a = new Character('a');
```

```
        Character b = new Character('b');
        Character c = new Character('c');

        Object[] arr = l.toArray();
        assertEquals(0, arr.length);


        l.add(a);
        l.add(b);
        l.add(c);
        assertListIs(l, "abc");

        arr = l.toArray();
        assertEquals(3, arr.length);
        assertSame(a, arr[0]);
        assertSame(b, arr[1]);
        assertSame(c, arr[2]);
    }


    public void testToArraySameSize()
    {
        BasicList l = new BasicList();
        Character a = new Character('a');
        Character b = new Character('b');
        Character c = new Character('c');

        Character[] arr = new Character[3];

        l.add(a);
        l.add(b);
        l.add(c);
        assertListIs(l, "abc");

        arr = (Character[]) l.toArray(arr);
        assertEquals(3, arr.length);
        assertSame(a, arr[0]);
        assertSame(b, arr[1]);
        assertSame(c, arr[2]);
    }


    public void testToArrayUndersize()
    {
        BasicList l = new BasicList();
        Character a = new Character('a');
        Character b = new Character('b');
        Character c = new Character('c');

        Character[] arr = new Character[2];

        l.add(a);
        l.add(b);
        l.add(c);
        assertListIs(l, "abc");

        arr = (Character[]) l.toArray(arr);
        assertEquals(3, arr.length);
        assertSame(a, arr[0]);
        assertSame(b, arr[1]);
        assertSame(c, arr[2]);
    }


    public void testToArrayOversize()
    {
        BasicList l = new BasicList();
        Character a = new Character('a');
        Character b = new Character('b');
```



```
        Character c = new Character('c');

        Character[] arr = new Character[5];

        l.add(a);
        l.add(b);
        l.add(c);
        assertListIs(l, "abc");

        arr = (Character[]) l.toArray(arr);
        assertEquals(5, arr.length);
        assertSame(a, arr[0]);
        assertSame(b, arr[1]);
        assertSame(c, arr[2]);
        assertSame(null, arr[3]);

    }


    public void testToArrayBadType()
    {
        BasicList l = new BasicList();
        Character a = new Character('a');
        Character b = new Character('b');
        Character c = new Character('c');

        String[] arr = new String[3];

        l.add(a);
        l.add(b);
        l.add(c);
        assertListIs(l, "abc");

        try
        {
            arr = (String[]) l.toArray(arr);
            fail("Expected an ArrayStoreException to be thrown");
        }
        catch (ArrayStoreException e)
        {

        } // end of try-catch
    }


    // Tell JUnit what order to run the tests in
    public static Test suite() {
        TestSuite suite = new TestSuite();

        suite.addTest(new BasicListTest("testGet"));
        suite.addTest(new BasicListTest("testAdd"));
        suite.addTest(new BasicListTest("testAddNull"));
        suite.addTest(new BasicListTest("testAddIndexed"));
        suite.addTest(new BasicListTest("testEqualsNullAndOtherType"));
        suite.addTest(new BasicListTest("testEqualsFalse"));
        suite.addTest(new BasicListTest("testEqualsTrue"));
        suite.addTest(new BasicListTest("testHashCode"));
        suite.addTest(new BasicListTest("testOneArgConstructor"));
        suite.addTest(new BasicListTest("testContains"));
        suite.addTest(new BasicListTest("testIndexOf"));
        suite.addTest(new BasicListTest("testIndexOfNull"));
        suite.addTest(new BasicListTest("testIsEmpty"));
        suite.addTest(new BasicListTest("testRemoveEffects"));
        suite.addTest(new BasicListTest("testRemoveReturnValue"));
        suite.addTest(new BasicListTest("testRemoveOOB"));
        suite.addTest(new BasicListTest("testSize"));
        suite.addTest(new BasicListTest("testIterator"));
        suite.addTest(new BasicListTest("testToArrayNoArgs"));
        suite.addTest(new BasicListTest("testToArraySameSize"));
```

```
        suite.addTest(new BasicListTest("testToArrayUndersize"));
        suite.addTest(new BasicListTest("testToArrayOversize"));
        suite.addTest(new BasicListTest("testToArrayBadType"));

        // ////////////
        //add other method names here
        // ////////////

        return suite;
    }

}
```



```
package ps2;

import java.lang.Double;

/** RatNum represents an immutable rational number.
    It includes all of the elements of the set of rationals, as well
    as the special "NaN" (not-a-number) element that results from
    division by zero.
    <p>
    The "NaN" element is special in many ways.  Any arithmetic
    operation (such as addition) involving "NaN" will return "NaN".
    With respect to comparison operations, such as less-than, "NaN" is
    considered equal to itself, and larger than all other rationals.
    <p>
    Examples of RatNums include "-1/13", "53/7", "4", "NaN", and "0".
*/
public class RatNum {

    private int numer;
    private int denom;

    // Abstraction Function:
    // A RatNum r is NaN if r.denom = 0, (r.numer / r.denom) otherwise.
    // (An abstraction function explains what the state of the fields in a
    // RatNum represents.  In this case, a rational number can be
    // understood as the result of dividing two integers, or not-a-number
    // if we would be dividing by zero.)

    // Representation invariant for every RatNum r:
    //  (r.denom >= 0) &&
    //  (r.denom > 0 ==> there does not exist integer i > 1 such that
    //                   r.numer mod i = 0 and r.denom mod i = 0)
    // (A representation invariant tells us something that is true for all
    // instances of a RatNum; in this case, that the denominator is always
    // greater than zero and that if the denominator is non-zero, the
    // fraction represented is in reduced form.)

    /** @effects Constructs a new RatNum = n. */
    public RatNum(int n) {
        numer = n;
        denom = 1;
        checkRep();
    }

    /** @effects If d = 0, constructs a new RatNum = NaN.  Else
    constructs a new RatNum = (n / d).
    */
    public RatNum(int n, int d) {
        // special case for zero denominator; gcd(n,d) requires d != 0
        if (d == 0) {
            numer = n;
            denom = 0;
            checkRep();
            return;
        }

        // reduce ratio to lowest terms
        int g = gcd(n, d);
        numer = n / g;
        denom = d / g;

        if (denom < 0) {
            numer = -numer;
            denom = -denom;
            checkRep();
        }
    }

    /** Checks to see if the representation invariant is being violated and if s
```

```
o, throws RuntimeException
    @throws RuntimeException if representation invariant is violated
    */
    private void checkRep() throws RuntimeException {
        if (denom < 0)
            throw new RuntimeException("Denominator of a RatNum cannot be less than zero");

        if (denom > 0) {
            int thisGcd = gcd(numer, denom);
            if (thisGcd != 1 && thisGcd != -1) {
                throw new RuntimeException("RatNum not in lowest form");
            }
        }
    }

    /** @return true iff this is NaN (not-a-number) */
    public boolean isNaN() {
        checkRep();
        return (denom == 0);
    }

    /** @return true iff this < 0. */
    public boolean isNegative() {
        checkRep();
        return (compareTo(new RatNum(0)) < 0);
    }

    /** @return true iff this > 0. */
    public boolean isPositive() {
        checkRep();
        return (compareTo(new RatNum(0)) > 0);
    }

    /** @requires rn != null
    @return a negative number if this < rn,
    0 if this = rn,
    a positive number if this > rn.
    */
    public int compareTo(RatNum rn) {
        checkRep();
        if (this.isNaN() && rn.isNaN()) {
            checkRep();
            return 0;
        } else if (this.isNaN()) {
            checkRep();
            return 1;
        } else if (rn.isNaN()) {
            checkRep();
            return -1;
        } else {
            RatNum diff = this.sub(rn);
            checkRep();
            return diff.numer;
        }
    }

    /** Approximates the value of this rational.
    @return a double approximation for this.  Note that "NaN" is
    mapped to {@link Double#NaN}, and the {@link Double#NaN} value
    is treated in a special manner by several arithmetic operations,
    such as the comparison and equality operators.  See the
    <a href="http://java.sun.com/docs/books/jls/second_edition/html/typesValues.
doc.html#9208">
    Java Language Specification, section 4.2.3</a>, for more details.
    */
    public double approx() {
        checkRep();
        if (isNaN()) {
            checkRep();
```



```
            return Double.NaN;
        } else {
            // convert int values to doubles before dividing.
            checkRep();
            return ((double) numer) / ((double) denom);
        }
    }

    /** @return a String representing this, in reduced terms.
    The returned string will either be "NaN", or it will take on
    either of the forms "N" or "N/M", where N and M are both
    integers in decimal notation and M != 0.
    */
    public String unparse() {
        // using '+' as String concatenation operator in this method
        checkRep();
        if (isNaN()) {
            checkRep();
            return "NaN";
        } else if (denom != 1) {
            checkRep();
            return numer + "/" + denom;
        } else {
            checkRep();
            return Integer.toString(numer);
        }
    }

    // in the implementation comments for the following methods, <this>
    // is notated as "a/b" and <arg> likewise as "x/y"

    /** @return a new Rational equal to (0 - this). */
    public RatNum negate() {
        checkRep();
        return new RatNum(-this.numer, this.denom);
    }

    /** @requires arg != null
    @return a new RatNum equal to (this + arg).
    If either argument is NaN, then returns NaN.
    */
    public RatNum add(RatNum arg) {
        // a/b + x/y = ay/by + bx/by = (ay + bx)/by
        checkRep();
        return new RatNum(
            this.numer * arg.denom + arg.numer * this.denom,
            this.denom * arg.denom);
    }

    /** @requires arg != null
    @return a new RatNum equal to (this - arg).
    If either argument is NaN, then returns NaN.
    */
    public RatNum sub(RatNum arg) {
        // a/b - x/y = a/b + -x/y
        checkRep();
        return this.add(arg.negate());
    }

    /** @requires arg != null
    @return a new RatNum equal to (this * arg).
    If either argument is NaN, then returns NaN.
    */
    public RatNum mul(RatNum arg) {
        // (a/b) * (x/y) = ax/by
        checkRep();
        return new RatNum(this.numer * arg.numer, this.denom * arg.denom);
    }
```

```
    /** @requires arg != null
    @return a new RatNum equal to (this / arg).
    If arg is zero, or if either argument is NaN, then returns NaN.
    */
    public RatNum div(RatNum arg) {
        // (a/b) / (x/y) = ay/bx
        checkRep();
        if (arg.isNaN()) {
            checkRep();
            return arg;
        } else {
            checkRep();
            return new RatNum(this.numer * arg.denom, this.denom * arg.numer);
        }
    }

    /** Returns the greatest common divisor of 'a' and 'b'.
    @requires b != 0
    @return d such that a % d = 0 and b % d = 0
    */
    private static int gcd(int a, int b) {
        // Euclid's method
        if (b == 0)
            return 0;
        while (b != 0) {
            int tmp = b;
            b = a % b;
            a = tmp;
        }
        return a;
    }

    /** Standard hashCode function.
    @return an int that all objects equal to this will also
    return.
    */
    public int hashCode() {
        checkRep();
        // all instances that are NaN must return the same hashcode;
        if (this.isNaN()) {
            return 0;
        }
        return this.numer * 2 + this.denom * 3;
    }

    /** Standard equality operation.
    @return true if and only if 'obj' is an instance of a RatNum
    and 'this' = 'obj'.  Note that NaN = NaN for RatNums.
    */
    public boolean equals(Object obj) {
        checkRep();
        if (obj instanceof RatNum) {
            RatNum rn = (RatNum) obj;

            // special case: check if both are NaN
            if (this.isNaN() && rn.isNaN()) {
                checkRep();
                return true;
            } else {
                checkRep();
                return this.numer == rn.numer && this.denom == rn.denom;
            }
        } else {
            checkRep();
            return false;
        }
    }

    /** @return implementation specific debugging string. */
```



```
    public String debugPrint() {
        checkRep();
        return "RatNum<numer:" + this.numer + " denom:" + this.denom + ">";
    }

    // When debugging, or when interfacing with other programs that have
    // specific I/O requirements, you might change this.
    public String toString() {
        checkRep();
        return debugPrint();
    }

    /** Makes a RatNum from a string describing it.
    @requires 'ratStr' is an instance of a string, with no spaces,
    of the form: <UL>
    <LI> "NaN"
    <LI> "N/M", where N and M are both integers in
    decimal notation, and M != 0, or
    <LI> "N", where N is an integer in decimal
    notation.
    </UL>
        @returns NaN if ratStr = "NaN".  Else returns a
    RatNum r = ( N / M ), letting M be 1 in the case
    where only "N" is passed in.
    */
    public static RatNum parse(String ratStr) {
        int slashLoc = ratStr.indexOf('/');
        if (ratStr.equals("NaN")) {
            return new RatNum(1, 0);
        } else if (slashLoc == -1) {
            // not NaN, and no slash, must be an Integer
            return new RatNum(Integer.parseInt(ratStr));
        } else {
            // slash, need to parse the two parts seperately
            int n = Integer.parseInt(ratStr.substring(0, slashLoc));
            int d =
                Integer.parseInt(
                    ratStr.substring(slashLoc + 1, ratStr.length()));
            return new RatNum(n, d);
        }
    }
}
```

```
package ps2;

import junit.framework.TestCase;

public class RatNumTest extends TestCase {

    // naming convention used throughout class: spell out number in
    // variable as its constructive form.  Unary minus is notated with
    // the prefix "neg", and the solidus is notated with an 'I'
    // character.  Thus, "1 + 2/3" becomes one_plus_two_I_three

    // some simple base RatNums
    private RatNum zero = new RatNum(0);
    private RatNum one = new RatNum(1);
    private RatNum negOne = new RatNum(-1);
    private RatNum two = new RatNum(2);
    private RatNum three = new RatNum(3);

    private RatNum one_I_two = new RatNum(1, 2);
    private RatNum one_I_three = new RatNum(1, 3);
    private RatNum one_I_four = new RatNum(1, 4);
    private RatNum two_I_three = new RatNum(2, 3);
    private RatNum three_I_four = new RatNum(3, 4);

    private RatNum negOne_I_two = new RatNum(-1, 2);

    // improper fraction
    private RatNum three_I_two = new RatNum(3, 2);

    // NaNs
    private RatNum one_I_zero = new RatNum(1, 0);
    private RatNum negOne_I_zero = new RatNum(-1, 0);
    private RatNum hundred_I_zero = new RatNum(100, 0);

    // ratnums: Set of varied ratnums (includes NaNs)
    // set is { 0, 1, -1, 2, 1/2, 3/2, 1/0, -1/0, 100/0 }
    private RatNum[] ratnums =
        new RatNum[] {
            zero,
            one,
            negOne,
            two,
            one_I_two,
            negOne_I_two,
            three_I_two,
        /* NaNs */
        one_I_zero, negOne_I_zero, hundred_I_zero };

    // ratnans: Set of varied NaNs
    // set is { 1/0, -1/0, 100/0 }
    private RatNum[] ratnans =
        new RatNum[] { one_I_zero, negOne_I_zero, hundred_I_zero };

    // ratnanans: Set of varied non-NaN ratnums
    // set is ratnums - ratnans
    private RatNum[] ratnanans =
        new RatNum[] { zero, one, negOne, two, one_I_two, three_I_two };

    public RatNumTest(String name) {
        super(name);
    }

    // self-explanatory helper function
    private void eq(RatNum ratNum, String rep) {
        assertEquals(rep, ratNum.unparse());
    }

    public void testOneArgCtor() {
        eq(zero, "0");
```



```
        eq(one, "1");

        RatNum four = new RatNum(4);
        eq(four, "4");

        eq(negOne, "-1");

        RatNum negFive = new RatNum(-5);
        eq(negFive, "-5");

        RatNum negZero = new RatNum(-0);
        eq(negZero, "0");
    }

    public void testTwoArgCtor() {
        RatNum one_I_two = new RatNum(1, 2);
        eq(one_I_two, "1/2");

        RatNum two_I_one = new RatNum(2, 1);
        eq(two_I_one, "2");

        RatNum three_I_two = new RatNum(3, 2);
        eq(three_I_two, "3/2");

        RatNum negOne_I_thirteen = new RatNum(-1, 13);
        eq(negOne_I_thirteen, "-1/13");

        RatNum fiftyThree_I_seven = new RatNum(53, 7);
        eq(fiftyThree_I_seven, "53/7");

        RatNum zero_I_one = new RatNum(0, 1);
        eq(zero_I_one, "0");
    }

    public void testTwoArgCtorOnNaN() {
        RatNum one_I_zero = new RatNum(1, 0);
        eq(one_I_zero, "NaN");

        RatNum two_I_zero = new RatNum(2, 0);
        eq(two_I_zero, "NaN");

        RatNum negOne_I_zero = new RatNum(-1, 0);
        eq(negOne_I_zero, "NaN");

        RatNum zero_I_zero = new RatNum(0, 0);
        eq(zero_I_zero, "NaN");

        RatNum negHundred_I_zero = new RatNum(-100, 0);
        eq(negHundred_I_zero, "NaN");
    }

    public void testReduction() {
        RatNum negOne_I_negTwo = new RatNum(-1, -2);
        eq(negOne_I_negTwo, "1/2");

        RatNum two_I_four = new RatNum(2, 4);
        eq(two_I_four, "1/2");

        RatNum six_I_four = new RatNum(6, 4);
        eq(six_I_four, "3/2");

        RatNum twentySeven_I_thirteen = new RatNum(27, 13);
        eq(twentySeven_I_thirteen, "27/13");

        RatNum negHundred_I_negHundred = new RatNum(-100, -100);
        eq(negHundred_I_negHundred, "1");
    }
```

```
    // self-explanatory helper function
    private void approxEq(double d1, double d2) {
        assertTrue(Math.abs(d1 - d2) < .0000001);
    }
    public void testApprox() {
        approxEq(zero.approx(), 0.0);
        approxEq(one.approx(), 1.0);
        approxEq(negOne.approx(), -1.0);
        approxEq(two.approx(), 2.0);
        approxEq(one_I_two.approx(), 0.5);
        approxEq(two_I_three.approx(), 2. / 3.);
        approxEq(three_I_four.approx(), 0.75);

        // cannot test that one_I_zero.approx() approxEq Double.NaN,
        // because it WON'T!!!  Instead, construct corresponding
        // instance of Double and use .equals(..) method
        assertTrue(
            (new Double(Double.NaN)).equals(new Double(one_I_zero.approx())));

        // use left-shift operator "<<" to create integer for 2^30
        RatNum one_I_twoToThirty = new RatNum(1, (1 << 30));
        double quiteSmall = 1. / Math.pow(2, 30);
        approxEq(one_I_twoToThirty.approx(), quiteSmall);
    }

    public void testAddSimple() {
        eq(zero.add(zero), "0");
        eq(zero.add(one), "1");
        eq(one.add(zero), "1");
        eq(one.add(one), "2");
        eq(one.add(negOne), "0");
        eq(one.add(two), "3");
        eq(two.add(two), "4");
    }

    public void testAddComplex() {
        eq(one_I_two.add(zero), "1/2");
        eq(one_I_two.add(one), "3/2");
        eq(one_I_two.add(one_I_two), "1");
        eq(one_I_two.add(one_I_three), "5/6");
        eq(one_I_two.add(negOne), "-1/2");
        eq(one_I_two.add(two), "5/2");
        eq(one_I_two.add(two_I_three), "7/6");
        eq(one_I_two.add(three_I_four), "5/4");

        eq(one_I_three.add(zero), "1/3");
        eq(one_I_three.add(two_I_three), "1");
        eq(one_I_three.add(three_I_four), "13/12");
    }

    public void testAddImproper() {
        eq(three_I_two.add(one_I_two), "2");
        eq(three_I_two.add(one_I_three), "11/6");
        eq(three_I_four.add(three_I_four), "3/2");

        eq(three_I_two.add(three_I_two), "3");
    }

    public void testAddOnNaN() {
        // each test case (addend, augend) drawn from the set
        // ratnums x ratnans

        for (int i = 0; i < ratnums.length; i++) {
            for (int j = 0; j < ratnans.length; j++) {
                eq(ratnums[i].add(ratnans[j]), "NaN");
                eq(ratnans[j].add(ratnums[i]), "NaN");
            }
        }
```



```
    }

    public void testAddTransitively() {
        eq(one.add(one).add(one), "3");
        eq(one.add(one.add(one)), "3");
        eq(zero.add(zero).add(zero), "0");
        eq(zero.add(zero.add(zero)), "0");
        eq(one.add(two).add(three), "6");
        eq(one.add(two.add(three)), "6");

        eq(one_I_three.add(one_I_three).add(one_I_three), "1");
        eq(one_I_three.add(one_I_three.add(one_I_three)), "1");

        eq(one_I_zero.add(one_I_zero).add(one_I_zero), "NaN");
        eq(one_I_zero.add(one_I_zero.add(one_I_zero)), "NaN");

        eq(one_I_two.add(one_I_three).add(one_I_four), "13/12");
        eq(one_I_two.add(one_I_three.add(one_I_four)), "13/12");
    }

    public void testSubSimple() {
        eq(zero.sub(one), "-1");
        eq(zero.sub(zero), "0");
        eq(one.sub(zero), "1");
        eq(one.sub(one), "0");
        eq(two.sub(one), "1");
        eq(one.sub(negOne), "2");
        eq(one.sub(two), "-1");
        eq(one.sub(three), "-2");
    }

    public void testSubComplex() {
        eq(one.sub(one_I_two), "1/2");
        eq(one_I_two.sub(one), "-1/2");
        eq(one_I_two.sub(zero), "1/2");
        eq(one_I_two.sub(two_I_three), "-1/6");
        eq(one_I_two.sub(three_I_four), "-1/4");
    }

    public void testSubImproper() {
        eq(three_I_two.sub(one_I_two), "1");
        eq(three_I_two.sub(one_I_three), "7/6");
    }

    public void testSubOnNaN() {
        // analogous to testAddOnNaN()

        for (int i = 0; i < ratnums.length; i++) {
            for (int j = 0; j < ratnans.length; j++) {
                eq(ratnums[i].sub(ratnans[j]), "NaN");
                eq(ratnans[j].sub(ratnums[i]), "NaN");
            }
        }
    }

    public void testSubTransitively() {
        // subtraction is not transitive; testing that operation is
        // correct when *applied transitivitely*, not that it obeys
        // the transitive property

        eq(one.sub(one).sub(one), "-1");
        eq(one.sub(one.sub(one)), "1");
        eq(zero.sub(zero).sub(zero), "0");
        eq(zero.sub(zero.sub(zero)), "0");
        eq(one.sub(two).sub(three), "-4");
        eq(one.sub(two.sub(three)), "2");

        eq(one_I_three.sub(one_I_three).sub(one_I_three), "-1/3");
        eq(one_I_three.sub(one_I_three.sub(one_I_three)), "1/3");
```

```
        eq(one_I_zero.sub(one_I_zero).sub(one_I_zero), "NaN");
        eq(one_I_zero.sub(one_I_zero.sub(one_I_zero)), "NaN");

        eq(one_I_two.sub(one_I_three).sub(one_I_four), "-1/12");
        eq(one_I_two.sub(one_I_three.sub(one_I_four)), "5/12");
    }

    public void testMulProperties() {
        // zero property
        for (int i = 0; i < ratnanans.length; i++) {
            eq(zero.mul(ratnanans[i]), "0");
            eq(ratnanans[i].mul(zero), "0");
        }

        // one property
        for (int i = 0; i < ratnanans.length; i++) {
            eq(one.mul(ratnanans[i]), ratnanans[i].unparse());
            eq(ratnanans[i].mul(one), ratnanans[i].unparse());
        }

        // negOne property
        for (int i = 0; i < ratnanans.length; i++) {
            eq(negOne.mul(ratnanans[i]), ratnanans[i].negate().unparse());
            eq(ratnanans[i].mul(negOne), ratnanans[i].negate().unparse());
        }
    }

    public void testMulSimple() {
        eq(two.mul(two), "4");
        eq(two.mul(three), "6");
        eq(three.mul(two), "6");
    }

    public void testMulComplex() {
        eq(one_I_two.mul(two), "1");
        eq(two.mul(one_I_two), "1");
        eq(one_I_two.mul(one_I_two), "1/4");
        eq(one_I_two.mul(one_I_three), "1/6");
        eq(one_I_three.mul(one_I_two), "1/6");
    }

    public void testMulImproper() {
        eq(three_I_two.mul(one_I_two), "3/4");
        eq(three_I_two.mul(one_I_three), "1/2");
        eq(three_I_two.mul(three_I_four), "9/8");
        eq(three_I_two.mul(three_I_two), "9/4");
    }

    public void testMulOnNaN() {
        // analogous to testAddOnNaN()

        for (int i = 0; i < ratnums.length; i++) {
            for (int j = 0; j < ratnans.length; j++) {
                eq(ratnums[i].mul(ratnans[j]), "NaN");
                eq(ratnans[j].mul(ratnums[i]), "NaN");
            }
        }
    }

    public void testMulTransitively() {
        eq(one.mul(one).mul(one), "1");
        eq(one.mul(one.mul(one)), "1");
        eq(zero.mul(zero).mul(zero), "0");
        eq(zero.mul(zero.mul(zero)), "0");
        eq(one.mul(two).mul(three), "6");
        eq(one.mul(two.mul(three)), "6");

        eq(one_I_three.mul(one_I_three).mul(one_I_three), "1/27");
```



```
        eq(one_I_three.mul(one_I_three.mul(one_I_three)), "1/27");

        eq(one_I_zero.mul(one_I_zero).mul(one_I_zero), "NaN");
        eq(one_I_zero.mul(one_I_zero.mul(one_I_zero)), "NaN");

        eq(one_I_two.mul(one_I_three).mul(one_I_four), "1/24");
        eq(one_I_two.mul(one_I_three.mul(one_I_four)), "1/24");
    }

    public void testDivSimple() {
        eq(zero.div(zero), "NaN");
        eq(zero.div(one), "0");
        eq(one.div(zero), "NaN");
        eq(one.div(one), "1");
        eq(one.div(negOne), "-1");
        eq(one.div(two), "1/2");
        eq(two.div(two), "1");
    }

    public void testDivComplex() {
        eq(one_I_two.div(zero), "NaN");
        eq(one_I_two.div(one), "1/2");
        eq(one_I_two.div(one_I_two), "1");
        eq(one_I_two.div(one_I_three), "3/2");
        eq(one_I_two.div(negOne), "-1/2");
        eq(one_I_two.div(two), "1/4");
        eq(one_I_two.div(two_I_three), "3/4");
        eq(one_I_two.div(three_I_four), "2/3");

        eq(one_I_three.div(zero), "NaN");
        eq(one_I_three.div(two_I_three), "1/2");
        eq(one_I_three.div(three_I_four), "4/9");
    }

    public void testDivImproper() {
        eq(three_I_two.div(one_I_two), "3");
        eq(three_I_two.div(one_I_three), "9/2");
        eq(three_I_two.div(three_I_two), "1");
    }

    public void testDivOnNaN() {
        // each test case (addend, augend) drawn from the set
        // ratnums x ratnans

        for (int i = 0; i < ratnums.length; i++) {
            for (int j = 0; j < ratnans.length; j++) {
                eq(ratnums[i].div(ratnans[j]), "NaN");
                eq(ratnans[j].div(ratnums[i]), "NaN");
            }
        }

    }

    public void testDivTransitively() {
        // (same note as in testSubTransitively re: transitivity property)

        eq(one.div(one).div(one), "1");
        eq(one.div(one.div(one)), "1");
        eq(zero.div(zero).div(zero), "NaN");
        eq(zero.div(zero.div(zero)), "NaN");
        eq(one.div(two).div(three), "1/6");
        eq(one.div(two.div(three)), "3/2");

        eq(one_I_three.div(one_I_three).div(one_I_three), "3");
        eq(one_I_three.div(one_I_three.div(one_I_three)), "1/3");

        eq(one_I_zero.div(one_I_zero).div(one_I_zero), "NaN");
        eq(one_I_zero.div(one_I_zero.div(one_I_zero)), "NaN");
```

```
        eq(one_I_two.div(one_I_three).div(one_I_four), "6");
        eq(one_I_two.div(one_I_three.div(one_I_four)), "3/8");

    }

    public void testNegate() {
        eq(zero.negate(), "0");
        eq(one.negate(), "-1");
        eq(negOne.negate(), "1");
        eq(two.negate(), "-2");
        eq(three.negate(), "-3");

        eq(one_I_two.negate(), "-1/2");
        eq(one_I_three.negate(), "-1/3");
        eq(one_I_four.negate(), "-1/4");
        eq(two_I_three.negate(), "-2/3");
        eq(three_I_four.negate(), "-3/4");

        eq(three_I_two.negate(), "-3/2");

        eq(one_I_zero.negate(), "NaN");
        eq(negOne_I_zero.negate(), "NaN");
        eq(hundred_I_zero.negate(), "NaN");
    }

    // helper function, "decode-and-check"
    private void decChk(String s, RatNum expected) {
        eq(RatNum.parse(s), expected.unparse());
    }
    public void testParse() {
        decChk("0", zero);

        decChk("1", one);
        decChk("1/1", one);
        decChk("2/2", one);
        decChk("-1/-1", one);

        decChk("-1", negOne);
        decChk("1/-1", negOne);
        decChk("-3/3", negOne);

        decChk("2", two);
        decChk("2/1", two);
        decChk("-4/-2", two);

        decChk("1/2", one_I_two);
        decChk("2/4", one_I_two);

        decChk("3/2", three_I_two);
        decChk("-6/-4", three_I_two);

        decChk("NaN", one_I_zero);
        decChk("NaN", negOne_I_zero);
    }

    public void testEqualsReflexive() {
        for (int i = 0; i < ratnums.length; i++) {
            assertTrue(ratnums[i].equals(ratnums[i]));
        }
    }
    public void testEquals() {
        assertTrue(one.equals(one));
        assertTrue(one.add(one).equals(two));
        assertTrue(one_I_two.equals(one.div(two)));
        assertTrue(three_I_two.equals(three.div(two)));

        assertTrue(one_I_zero.equals(one_I_zero));
        assertTrue(one_I_zero.equals(negOne_I_zero));
        assertTrue(one_I_zero.equals(hundred_I_zero));
```



```
        assertTrue(!one.equals(zero));
        assertTrue(!zero.equals(one));
        assertTrue(!one.equals(two));
        assertTrue(!two.equals(one));
        assertTrue(!one.equals(negOne));
        assertTrue(!negOne.equals(one));

        assertTrue(!one.equals(one_I_two));
        assertTrue(!one_I_two.equals(one));
        assertTrue(!one.equals(three_I_two));
        assertTrue(!three_I_two.equals(one));
    }

    private void assertGreater(RatNum larger, RatNum smaller) {
        assertTrue(larger.compareTo(smaller) > 0);
        assertTrue(smaller.compareTo(larger) < 0);
    }
    public void testCompareToReflexive() {
        // reflexivitiy
        for (int i = 0; i < ratnums.length; i++) {
            assertTrue(ratnums[i].compareTo(ratnums[i]) == 0);
        }
    }
    public void testCompareToNonFract() {
        assertGreater(one, zero);
        assertGreater(one, negOne);
        assertGreater(two, one);
        assertGreater(two, zero);
        assertGreater(zero, negOne);
    }
    public void testCompareToFract() {
        assertGreater(one, one_I_two);
        assertGreater(two, one_I_three);
        assertGreater(one, two_I_three);
        assertGreater(two, two_I_three);
        assertGreater(one_I_two, zero);
        assertGreater(one_I_two, negOne);
        assertGreater(one_I_two, negOne_I_two);
        assertGreater(zero, negOne_I_two);
    }
    public void testCompareToNaNs() {
        for (int i = 0; i < ratnans.length; i++) {
            for (int j = 0; j < ratnans.length; j++) {
                assertTrue(ratnans[i].compareTo(ratnans[j]) == 0);
            }
            for (int j = 0; j < ratnanans.length; j++) {
                assertGreater(ratnans[i], ratnanans[j]);
            }
        }
    }

    private void assertPos(RatNum n) {
        assertTrue(n.isPositive());
        assertTrue(!n.isNegative());
    }
    private void assertNeg(RatNum n) {
        assertTrue(n.isNegative());
        assertTrue(!n.isPositive());
    }
    public void testIsPosAndIsNeg() {
        assertTrue(!zero.isPositive());
        assertTrue(!zero.isNegative());

        assertPos(one);
        assertNeg(negOne);
        assertPos(two);
        assertPos(three);
```

```
        assertPos(one_I_two);
        assertPos(one_I_three);
        assertPos(one_I_four);
        assertPos(two_I_three);
        assertPos(three_I_four);

        assertNeg(negOne_I_two);

        assertPos(three_I_two);

        assertPos(one_I_zero);
        assertPos(negOne_I_zero); // non-intuitive; see spec
        assertPos(hundred_I_zero);
    }

    public void testIsNaN() {
        for (int i = 0; i < ratnans.length; i++) {
            assertTrue(ratnans[i].isNaN());
        }
        for (int i = 0; i < ratnanans.length; i++) {
            assertTrue(!ratnanans[i].isNaN());
        }
    }

}
```



```
package ps2;

import java.util.StringTokenizer;

/** <b>RatPoly</b> represents an immutable single-variate polynomial
    expression.  RatPolys have RatNum coefficients and integer
    exponents.
    <p>

    Examples of RatPolys include "0", "x-10", and "x^3-2*x^2+5/3*x+3",
    and "NaN".

    *** RatPoly was tested against the provided RatPolyTest today, and
    *** all tests succeeded. -- drkp 2004/02/19

*/
public class RatPoly {

    // holds terms of this
    private RatTermVec terms;

    // convenient zero and one constants
    private static final RatNum ZERO = new RatNum(0);
    private static final RatNum ONE = new RatNum(1);

    // convenient way to get a RatPoly that isNaN
    private static RatPoly nanPoly() {
        RatPoly a = new RatPoly();
        a.terms.addElement(new RatTerm(new RatNum(1, 0), 0));
        return a;
    }

    // Definitions:
    //   For a RatPoly p, let C(p,i) be "p.terms.get(i).coeff" and
    //                        E(p,i) be "p.terms.get(i).expt"
    //                        length(p) be "p.terms.size()"
    // (These are helper functions that will make it easier for us
    // to write the remainder of the specifications.  They are not
    // executable code; they just represent complex expressions in a
    // concise manner, so that we can stress the important parts of
    // other expressions in the spec rather than get bogged down in
    // the details of how we extract the coefficient for the 2nd term
    // or the exponent for the 5th term.  So when you see C(p,i),
    // think "coefficient for the ith term in p".)

    // Abstraction Function:
    //   A RatPoly p is the Sum, from i=0 to length(p), of C(p,i)*x^E(p,i)
    // (This explains what the state of the fields in a RatPoly
    // represents: it is the sum of a series of terms, forming an
    // expression like "C_0 + C_1*x^1 + C_2*x^2 + ...".  If there are no
    // terms, then the RatPoly represents the zero polynomial.)

    // Representation Invariant for every RatPoly p:
    //   terms != null &&
    //   forall i such that (0 <= i < length(p), C(p,i) != 0 &&
    //   forall i such that (0 <= i < length(p), E(p,i) >= 0 &&
    //   forall i such that (0 <= i < length(p) - 1), E(p,i) > E(p, i+1)
    //   (This tells us four important facts about every RatPoly:
    //   * the terms field always points to some usable object,
    //   * no term in a RatPoly has a zero coefficient,
    //   * no term in a RatPoly has a negative exponent, and
    //   * the terms in a RatPoly are sorted in descending exponent order.)

    /** @effects Constructs a new Poly, "0".
     */
    public RatPoly() {
        terms = new RatTermVec();
    }
```

29/30 but there are

Dan – your impl is good, some style issues:
- prefer {} to whitespace
- 80 chars per line
- make sure your editor does indents correctly – what are you using?

```
    /** @requires e >= 0
    @effects Constructs a new Poly equal to "c * x^e".
    If c is zero, constructs a "0" polynomial.
    */
    public RatPoly(int c, int e) {
        terms = new RatTermVec();
        if (c != 0)
            terms.addElement(new RatTerm(new RatNum(c), e));
        checkRep();
    }

    /** @requires 'rt' satisfies clauses given in rep. invariant
    @effects Constructs a new Poly using 'rt' as part of the
    representation. The method does not make a copy of 'rt'.
    */
    private RatPoly(RatTermVec rt) {
        terms = rt;
        //The specification says that the method does *not* make a copy
        //of 'rt', so we can simply make the assignment.  Because this
        //constructor is private, we do not necessarily need to
        //defensively copy the argument.
    }

    /** Returns the degree of this.
    @requires !this.isNaN()
    @return the largest exponent with a non-zero coefficient, or 0
    if this is "0".
    */
    public int degree() {
        checkRep();

        if (terms.size() == 0)
            return 0;
        else
            return terms.get(0).expt;

    }
    /** @requires !this.isNaN()
    @return the coefficient associated with term of degree 'deg'.
        If there is no term of degree 'deg' in this poly, then returns
        zero.
    */
    public RatNum coeff(int deg) {
        checkRep();

        for (int i = 0; i < terms.size(); i++)
        {
            if (terms.get(i).expt == deg)
                return terms.get(i).coeff;
        }
        return ZERO;
    }

    /** @return true if and only if this has some coefficient = "NaN".
     */
    public boolean isNaN() {
        checkRep();

        for (int i = 0; i < terms.size(); i++)
        {
            if (terms.get(i).coeff.isNaN())
                return true;
        }
        return false;
    }

    /** Returns a string representation of this.
    @return a String representation of the expression represented
```



```
    by this, with the terms sorted in order of degree from highest
    to lowest.
    <p>
    There is no whitespace in the returned string.
    <p>
    Terms with zero coefficients do not appear in the returned
    string, unless the polynomial is itself zero, in which case
    the returned string will just be "0".
    <p>
    If this.isNaN(), then the returned string will be just "NaN"
    <p>
    The string for a non-zero, non-NaN poly is in the form
    "(-)T(+|-)T(+|-)...",where for each
    term, T takes the form "C*x^E" or "C*x", UNLESS:
    (1) the exponent E is zero, in which case T takes the form "C", or
    (2) the coefficient C is one, in which case T takes the form
    "x^E" or "x"
    <p>
    Note that this format allows only the first term to be output
    as a negative expression.
    <p>
    Valid example outputs include "x^17-3/2*x^2+1", "-x+1", "-1/2",
    and "0".
    <p>
    */
    public String unparse() {
        String str = "";

        checkRep();

        if (terms.size() == 0)
            return "0";

        if (isNaN())
                return "NaN";

        for (int i = 0; i < terms.size(); i++)
        {
            if (terms.get(i).coeff.isNegative())
                str += "-";
            else if (i != 0) // don't include + for first term
                str += "+";

            RatNum coeffAbs = terms.get(i).coeff.isNegative() ?
                terms.get(i).coeff.negate() :
                terms.get(i).coeff;

            if (terms.get(i).expt == 0)
            {
                str += coeffAbs.unparse();
            }
            else
            {
                if (!coeffAbs.equals(ONE))
                {
                    str += coeffAbs.unparse();
                    str += "*";
                }
                str += "x";

                if (terms.get(i).expt != 1)
                {
                    str += "^";
                    str += terms.get(i).expt;
                }
            }
        }

        return str;
```

```
    }

    /** Scales coefficients within 'vec' by 'scalar' (helper procedure).
    @requires vec, scalar != null
    @modifies vec
    @effects Forall i s.t. 0 <= i < vec.size(),
    if (C . E) = vec.get(i)
    then vec_post.get(i) = (C*scalar . E)
    @see ps2.RatTerm regarding (C . E) notation
    */
    private static void scaleCoeff(RatTermVec vec, RatNum scalar) {
        for (int i = 0; i < vec.size(); i++) {
            vec.set(new RatTerm(vec.get(i).coeff.mul(scalar),
                    vec.get(i).expt), i);
        } // end of for ()
    }

    /** Increments exponents within 'vec' by 'degree' (helper procedure).
    @requires vec != null
    @modifies vec
    @effects Forall i s.t. 0 <= i < vec.size(),
    if (C . E) = vec.get(i)
    then vec_post.get(i) = (C . E+degree)
    @see ps2.RatTerm regarding (C . E) notation
    */
    private static void incremExpt(RatTermVec vec, int degree) {
        for (int i = 0; i < vec.size(); i++) {
            vec.set(new RatTerm(vec.get(i).coeff,
                    vec.get(i).expt + degree), i);
        } // end of for ()

    }

    /** Merges a term into a sequence of terms, preserving the
    sorted nature of the sequence (helper procedure).

    Definitions:
    Let a "Sorted RatTermVec" be a RatTermVec V such that
    [1] V is sorted in descending exponent order &&
    [2] there are no two RatTerms with the same exponent in V &&
    [3] there is no RatTerm in V with a coefficient equal to zero

    For a Sorted(RatTermVec) V and integer e, let cofind(V, e)
    be either the coefficient for a RatTerm rt in V whose
    exponent is e, or zero if there does not exist any such
    RatTerm in V.  (This is like the coeff function of RatPoly.)

    @requires vec != null && sorted(vec)
    @modifies vec
    @effects sorted(vec_post) &&
    cofind(vec_post,newTerm.expt)
    = cofind(vec,newTerm.expt) + newTerm.coeff
    */
    private static void sortedAdd(RatTermVec vec, RatTerm newTerm) {
        for (int i = 0; i < vec.size(); i++) {
            RatTerm term = vec.get(i);
            if (newTerm.expt > term.expt) {
                vec.insert(newTerm, i);
                return;
            }
            if (newTerm.expt == term.expt) {
                RatNum newCoeff = newTerm.coeff.add(term.coeff);
                vec.remove(i);
                if (!newCoeff.equals(ZERO))
                    vec.insert(new RatTerm(newCoeff, term.expt), i);
                return;
            }
        }
```



```
        vec.addElement(newTerm);
    }

    /** @return a new Poly equal to "0 - this";
    if this.isNaN(), returns some r such that r.isNaN()
    */
    public RatPoly negate() {
        checkRep();

        if (isNaN()) {
            return nanPoly();
        } // end of if ()

        RatTermVec newTerms = terms.copy();
        scaleCoeff(newTerms, new RatNum(-1));

        checkRep();

        return new RatPoly(newTerms);
    }

    /** Addition operation.
    @requires p != null
    @return a new RatPoly, r, such that r = "this + p";
    if this.isNaN() or p.isNaN(), returns some r such that r.isNaN()
    */
    public RatPoly add(RatPoly p) {
        checkRep();

        if (isNaN() || p.isNaN())
        {
            return nanPoly();
        } // end of if ()

        RatTermVec newTerms = terms.copy();

        for (int i = 0; i < p.terms.size(); i++)
        {
            sortedAdd(newTerms, p.terms.get(i));
        }

        return new RatPoly(newTerms);
    }

    /** Subtraction operation.
    @requires p != null
    @return a new RatPoly, r, such that r = "this - p";
    if this.isNaN() or p.isNaN(), returns some r such that r.isNaN()
    */
    public RatPoly sub(RatPoly p) {
        checkRep();

        return add(p.negate());
    }

    /** Multiplication operation.
    @requires p != null
    @return a new RatPoly, r, such that r = "this * p";
    if this.isNaN() or p.isNaN(), returns some r such that r.isNaN()
    */
    public RatPoly mul(RatPoly p) {
        checkRep();

        if (this.isNaN() || p.isNaN())
        {
            return nanPoly();

        } // end of if ()
```

```
        RatPoly partialProd = new RatPoly();

        for (int i = 0; i < p.terms.size(); i++)
        {
            RatTermVec vec = terms.copy();
            scaleCoeff(vec, p.terms.get(i).coeff);
            incremExpt(vec, p.terms.get(i).expt);
            partialProd = partialProd.add(new RatPoly(vec));
        } // end of for

        return partialProd;
    }

    /** Division operation (truncating).
    @requires p != null
    @return a new RatPoly, q, such that q = "this / p";
    if p = 0 or this.isNaN() or p.isNaN(),
    returns some q such that q.isNaN()
    <p>

    Division of polynomials is defined as follows:
    Given two polynomials u and v, with v != "0", we can divide u by
    v to obtain a quotient polynomial q and a remainder polynomial
    r satisfying the condition u = "q * v + r", where
    the degree of r is strictly less than the degree of v,
    the degree of q is no greater than the degree of u, and
    r and q have no negative exponents.
    <p>

    For the purposes of this class, the operation "u / v" returns
    q as defined above.
    <p>

    Thus, "x^3-2*x+3" / "3*x^2" = "1/3*x" (with the corresponding
    r = "-2*x+3"), and "x^2+2*x+15 / 2*x^3" = "0" (with the
    corresponding r = "x^2+2*x+15").
    <p>

    Note that this truncating behavior is similar to the behavior
    of integer division on computers.
    */
    public RatPoly div(RatPoly p) {
        RatPoly result = new RatPoly();
        if (p.isNaN() || this.isNaN() || p.terms.size() == 0) {
            return nanPoly();
        }

        RatPoly thisCopy = new RatPoly(this.terms.copy());

        while (thisCopy.degree() >= p.degree() && thisCopy.terms.size() > 0) {
            RatTerm tempTerm =
                new RatTerm(
                    thisCopy.terms.get(0).coeff.div(p.terms.get(0).coeff),
                    thisCopy.terms.get(0).expt - p.terms.get(0).expt);
            sortedAdd(result.terms, tempTerm);
            RatPoly tempPoly = new RatPoly();
            tempPoly.terms.addElement(tempTerm);
            thisCopy = thisCopy.sub(p.mul(tempPoly));
        }

        return result;
    }

    /** @return a new RatPoly that, q, such that q = dy/dx,
    where this == y.  In other words, q is the
    derivative of this.  If this.isNaN(), then return
    some q such that q.isNaN()
```



```
    <p>The derivative of a polynomial is the sum of the
    derivative of each term.

    <p>Given a term, a*x^b, the derivative of the term is:
    (a*b)*x^(b-1)  for b > 0 and 0 for b == 0
    */
    public RatPoly differentiate() {

        if (isNaN())
        {
            return nanPoly();
        } // end of if ()

        RatTermVec qTerms = new RatTermVec();

        for (int i = 0; i < terms.size(); i++)
        {
            if (terms.get(i).expt > 0)
            {
                qTerms.addElement(new RatTerm(terms.get(i).coeff.mul( new RatNum
(terms.get(i).expt)), (terms.get(i).expt)-1));
            } // end of if ()
        } // end of for ()

        return new RatPoly(qTerms);
    }

    /** @requires integrationConstant != null
    @return a new RatPoly that, q, such that dq/dx = this
    and the constant of integration is "integrationConstant"
    In other words, q is the antiderivative of this.
    If this.isNaN() or integrationConstant.isNaN(),
    then return some q such that q.isNaN()

    <p>The antiderivative of a polynomial is the sum of the
    antiderivative of each term plus some constant.

    <p>Given a term, a*x^b,  (where b >= 0)
    the antiderivative of the term is:  a/(b+1)*x^(b+1)
    */
    public RatPoly antiDifferentiate(RatNum integrationConstant) {
        if (isNaN() || integrationConstant.isNaN())
        {
            return nanPoly();
        } // end of if ()

        RatTermVec qTerms = new RatTermVec();

        for (int i = 0; i < terms.size(); i++)
        {
            if (terms.get(i).expt >= 0)
            {
                qTerms.addElement(new RatTerm(terms.get(i).coeff.div( new RatNum
((terms.get(i).expt)+1)), (terms.get(i).expt)+1));
            } // end of if ()
        } // end of for ()

        if (!integrationConstant.equals(ZERO))
        {
            qTerms.addElement(new RatTerm(integrationConstant, 0));
        } // end of if ()


        return new RatPoly(qTerms);
    }

    /** @return a double that is the definite integral of this with
    bounds of integration between lowerBound and upperBound.
```

```
    <p>  The Fundamental Theorem of Calculus states that the definite integral
    of f(x) with bounds a to b is F(b) - F(a) where dF/dx = f(x)
    NOTE: Remember that the lowerBound can be higher than the upperBound
    */
    public double integrate(double lowerBound, double upperBound) {
        RatPoly antiDeriv = antiDifferentiate(ZERO);

        return (antiDeriv.eval(upperBound) - antiDeriv.eval(lowerBound));
    }

    /** @return the value of this polynomial when evaluated at 'd'.
    For example, "x+2" evaluated at 3 is 5, and "x^2-x"
    evaluated at 3 is 6.
    if (this.isNaN() == true), return Double.NaN
    */
    public double eval(double d) {
        if (isNaN())
        {
            return Double.NaN;
        } // end of if ()

        double result = 0;

        for (int i = 0; i < terms.size(); i++)
        {
            result += (terms.get(i).coeff.approx())
                * Math.pow(d,(terms.get(i).expt));
        } // end of for ()

        return result;
    }

    /** @requires 'polyStr' is an instance of a string with no spaces
    that expresses a poly in the form defined in the
    unparse() method.
    @return a RatPoly p such that p.unparse() = polyStr
    */
    public static RatPoly parse(String polyStr) {

        RatPoly result = new RatPoly();

        // First we decompose the polyStr into its component terms;
        // third arg orders "+" and "-" to be returned as tokens.
        StringTokenizer termStrings = new StringTokenizer(polyStr, "+-", true);

        boolean nextTermIsNegative = false;
        while (termStrings.hasMoreTokens()) {
            String termToken = termStrings.nextToken();

            if (termToken.equals("-")) {
                nextTermIsNegative = true;
            } else if (termToken.equals("+")) {
                nextTermIsNegative = false;
            } else {
                // Not "+" or "-"; must be a term

                // Term is: "R" or "R*x" or "R*x^N" or "x^N" or "x",
                // where R is a rational num and N is a natural num.

                // Decompose the term into its component parts.
                // third arg orders '*' and '^' to act purely as delimiters.
                StringTokenizer numberStrings =
                    new StringTokenizer(termToken, "*^", false);

                RatNum coeff;
                int expt;

                String c1 = numberStrings.nextToken();
                if (c1.equals("x")) {
```



```
                    // ==> "x" or "x^N"
                    coeff = new RatNum(1);

                    if (numberStrings.hasMoreTokens()) {
                        // ==> "x^N"
                        String N = numberStrings.nextToken();
                        expt = Integer.parseInt(N);

                    } else {
                        // ==> "x"
                        expt = 1;
                    }
                } else {
                    // ==> "R" or "R*x" or "R*x^N"
                    String R = c1;
                    coeff = RatNum.parse(R);

                    if (numberStrings.hasMoreTokens()) {
                        // ==> "R*x" or "R*x^N"
                        String x = numberStrings.nextToken();

                        if (numberStrings.hasMoreTokens()) {
                            // ==> "R*x^N"
                            String N = numberStrings.nextToken();
                            expt = Integer.parseInt(N);
                        } else {
                            // ==> "R*x"
                            expt = 1;
                        }

                    } else {
                        // ==> "R"
                        expt = 0;
                    }
                }

                // at this point, coeff and expt are initialized.
                // Need to fix coeff if it was preceeded by a '-'
                if (nextTermIsNegative) {
                    coeff = coeff.negate();
                }

                // accumulate terms of polynomial in 'result'
                if (!coeff.equals(ZERO)) {
                    result.terms.addElement(new RatTerm(coeff, expt));
                }
            }
        }
        result.checkRep();
        return result;
    }

    /** Checks to see if the representation invariant is being violated and if s
o, throws RuntimeException
    @throws RuntimeException if representation invariant is violated
    */
    private void checkRep() throws RuntimeException {
        if (terms == null) {
            throw new RuntimeException("terms == null!");
        }
        for (int i = 0; i < terms.size(); i++) {
            if ((terms.get(i).coeff.compareTo(ZERO)) == 0) {
                throw new RuntimeException("zero coefficient!");
            }
            if (terms.get(i).expt < 0) {
                throw new RuntimeException("negative exponent!");
            }
            if (i < terms.size() - 1) {
                if (terms.get(i + 1).expt >= terms.get(i).expt) {
```

```
                throw new RuntimeException("terms out of order!");
            }
        }
    }
}
```



```
package ps2;

/** Cons is a simple cons cell record type. */
class Cons {
    RatPoly head;
    Cons tail;
    Cons(RatPoly h, Cons t) {
        head = h;
        tail = t;
    }
}

/** <b>RatPolyStack</B> is a mutable finite sequence of RatPoly objects.
    <p>
    Each RatPolyStack can be described by [p1, p2, ... ], where [] is
    an empty stack, [p1] is a one element stack containing the Poly
    'p1', and so on.  RatPolyStacks can also be described
    constructively, with the append operation, ':'. such that [p1]:S
    is the result of putting p1 at the front of the RatPolyStack S.
    <p>
    A finite sequence has an associated size, corresponding to the
    number of elements in the sequence.  Thus the size of [] is 0, the
    size of [p1] is 1, the size of [p1, p1] is 2, and so on.
    <p>
    Note that RatPolyStack is similar to <i>vectors</i> like {@link
    RatTermVec} with respect to its abstract state (a finite
    sequence), but is quite different in terms of intended usage.  A
    stack typically only needs to support operations around its top
    efficiently, while a vector is expected to be able to retrieve
    objects at any index in amortized constant time.  Thus it is
    acceptable for a stack to require O(n) time to retrieve an element
    at some arbitrary depth, but pushing and popping elements should
    be O(1) time.

    *** RatPolyStack was tested against the provided RatPolyStackTest
    *** today; all tests were successful. --drkp 2004/02/19

 */
public class RatPolyStack {

    private Cons polys; // head of list
    private int size; // redundantly-stored list length

    // Definitions:
    //   For a Cons c, let Seq(c) be [] if c == null,
    //                                [c.head]:Seq(c.tail) otherwise
    //                      Count(c) be 0 if c == null,
    //                                  1 + Count(c.tail) otherwise
    //
    // (These are helper functions that will make it easier for us
    // to write the remainder of the specifications.  They are
    // seperated out because the nature of this representation lends
    // itself to analysis by recursive functions.)

    // Abstraction Function:
    //   RatPolyStack s models Seq(s.polys)
    // (This explains how we can understand what a Stack is from its
    // 'polys' field.  (Though in truth, the real understanding comes
    // from grokking the Seq helper function).)

    // RepInvariant:
    //   s.size == Count(s.polys)
    // (This defines how the 'size' field relates to the 'polys'
    // field.  Notice that s.polys != null is *not* a given Invariant;
    // this class, unlike the RatPoly class, allows for one of its
    // fields to reference null, and thus your method implementations
    // should not assume that the 'polys' field will be non-null on
    // entry to the method, unless some other aspect of the method
    // will enforce this condition.)
```

15/15 nice, clean code

```
    /** @effects Constructs a new RatPolyStack, [].
     */
    public RatPolyStack() {
        size = 0;
        polys = null;
    }

    /** Pushes a RatPoly onto the top of this.
    @requires p != null
    @modifies this
    @effects this_post = [p]:this
    */
    public void push(RatPoly p) {
        polys = new Cons(p, polys);
        size++;
    }

    /** Returns the top RatPoly.
    @requires this.size() > 0
    @modifies this
    @effects If this = [p]:S
    then this_post = S && returns p
    */
    public RatPoly pop() {
        Cons popCons = polys;

        checkRep();
        polys = polys.tail;
        size--;

        checkRep();

        return popCons.head;
    }

    /** Duplicates the top RatPoly on this.
    @requires this.size() > 0
    @modifies this
    @effects If this = [p]:S
    then this_post = [p, p]:S
    */
    public void dup() {
        checkRep();

        polys = new Cons(polys.head, polys);
        size++;

        checkRep();
    }

    /** Swaps the top two elements of this.
    @requires this.size() >= 2
    @modifies this
    @effects If this = [p1, p2]:S
    then this_post = [p2, p1]:S
    */
    public void swap() {
        checkRep();

        RatPoly a, b;

        a = pop();
        b = pop();
        push(a);
        push(b);

        checkRep();
    }
```



```
    /** Clears the stack.
    @modifies this
    @effects this_post = []
    */
    public void clear() {
        size = 0;
        polys = null;
    }

    /** Returns the RatPoly that is 'index' elements from the top of
    the stack.
    @requires index >= 0 && index < this.size()
    @effects If this = S:[p]:T where S.size() = index, then
    returns p.
    */
    public RatPoly get(int index) {
        Cons p = polys;

        for (int i = 0; i < index; i++)
        {
            p = p.tail;
        } // end of for ()

        return p.head;
    }

    /** Adds the top two elements of this, placing the result on top
    of the stack.
        @requires this.size() >= 2
    @modifies this
    @effects  If this = [p1, p2]:S
    then this_post = [p3]:S
    where p3 = p1 + p2
    */
    public void add() {
        checkRep();

        RatPoly a = pop();
        RatPoly b = pop();

        push(a.add(b));

        checkRep();
    }

    /** Subtracts the top poly from the next from top poly, placing
    the result on top of the stack.
        @requires this.size() >= 2
    @modifies this
    @effects  If this = [p1, p2]:S
    then this_post = [p3]:S
    where p3 = p2 - p1
    */
    public void sub() {
        checkRep();

        RatPoly a = pop();
        RatPoly b = pop();

        push(b.sub(a));

        checkRep();
    }

    /** Multiplies top two elements of this, placing the result on
    top of the stack.
    @requires this.size() >= 2
    @modifies this
```

```
    @effects  If this = [p1, p2]:S
    then this_post = [p3]:S
    where p3 = p1 * p2
    */
    public void mul() {
        checkRep();

        RatPoly a = pop();
        RatPoly b = pop();

        push(a.mul(b));

        checkRep();
    }

    /** Divides the next from top poly by the top poly, placing the
    result on top of the stack.
    @requires this.size() >= 2
    @modifies this
    @effects  If this = [p1, p2]:S
    then this_post = [p3]:S
    where p3 = p2 / p1
    */
    public void div() {
        checkRep();

        RatPoly a = pop();
        RatPoly b = pop();

        push(b.div(a));

        checkRep();
    }

    /** Integrates the top element of this, placing the result on top
    of the stack.
    @requires this.size() >= 1
    @modifies this
    @effects  If this = [p1]:S
    then this_post = [p2]:S
    where p2 = indefinite integral of p1 with integration constant 0
    */
    public void integrate() {
        checkRep();

        RatPoly a = pop();

        push(a.antiDifferentiate(new RatNum(0)));

        checkRep();
    }

    /** Differentiates the top element of this, placing the result on top
    of the stack.
    @requires this.size() >= 1
    @modifies this
    @effects  If this = [p1]:S
    then this_post = [p2]:S
    where p2 = derivative of p1
    */
    public void differentiate() {
        checkRep();

        RatPoly a = pop();

        push(a.differentiate());

        checkRep();
    }
```



```
    /** @return the size of this sequence.
     */
    public int size() {
        checkRep();

        return size;
    }

    /** Checks to see if the representation invariant is being violated and if s
o, throws RuntimeException
    @throws RuntimeException if representation invariant is violated
    */
    private void checkRep() throws RuntimeException {

        if (polys == null) {
            if (size != 0)
                throw new RuntimeException("size field should be equal to zero when polys is null s
ine stack is empty");
        } else {

            int countResult = 0;
            RatPoly headPoly = polys.head;
            Cons nextCons = polys;

            if (headPoly != null) {
                for (int i = 1;; i++) {
                    if (nextCons != null) {
                        countResult = i;
                        nextCons = nextCons.tail;
                    } else
                        break;
                }
            }
            if (countResult != size)
                throw new RuntimeException(
                    "size field is not equal to Count(s.polys). Size constant is "
                        + size
                        + " Cons cells have length "
                        + countResult);
        }
    }
}
```

```
package ps2;

import junit.framework.*;

public class RatPolyStackTest extends TestCase {
    // create a new poly that is a constant (doesn't depend on x)
    private RatPoly constantPoly(int constant) {
        return new RatPoly(constant, 0);
    }

    // create a new poly that is a constant (doesn't depend on x)
    // taking a char allows us to represent stacks as strings
    private RatPoly constantPoly(char constant) {
        return constantPoly(Integer.valueOf("" + constant).intValue());
    }

    //* @return a new RatPolyStack instance
    private RatPolyStack stack() {
        return new RatPolyStack();
    }

    // create stack of single-digit constant polys
    private RatPolyStack stack(String desc) {
        RatPolyStack s = new RatPolyStack();

        // go backwards to leave first element in desc on _top_ of stack
        for (int i = desc.length() - 1; i >= 0; i--) {
            char c = desc.charAt(i);
            s.push(constantPoly(c));
        }
        return s;
    }

    // RatPoly equality check
    // (getting around non-definition of RatPoly.equals)
    private boolean eqv(RatPoly p1, RatPoly p2) {
        return p1.unparse().equals(p2.unparse());
    }

    // compares 's' to a string describing its values
    // thus stack123 = "123".  desc MUST be a sequence of
    // decimal number chars
    //
    // NOTE: THIS CAN FAIL WITH A WORKING STACK IF RatPoly.unparse IS BROKEN!
    private void assertStackIs(RatPolyStack s, String desc) {
        assertTrue(s.size() == desc.length());

        for (int i = 0; i < desc.length(); i++) {
            RatPoly p = s.get(i);
            char c = desc.charAt(i);
            String asstr =
                "Elem("
                + i
                + "): "
                + p.unparse()
                + ", Expected "
                + c
                + ", (Expected Stack:"
                + desc
                + ")";

            assertTrue(asstr, eqv(p, constantPoly(c)));
        }
    }

    public RatPolyStackTest(String name) {
        super(name);
    }
```



```
    public void testCtor() {
        RatPolyStack stk1 = stack();
        assertTrue(stk1.size() == 0);
    }

    public void testPush() {
        RatPolyStack stk1 = stack();
        stk1.push(constantPoly(0));

        assertStackIs(stk1, "0");

        stk1.push(constantPoly(0));
        assertStackIs(stk1, "00");

        stk1.push(constantPoly(1));
        assertStackIs(stk1, "100");

        stk1 = stack("3");
        assertStackIs(stk1, "3");

        stk1 = stack("23");
        assertStackIs(stk1, "23");

        stk1 = stack("123");
        assertStackIs(stk1, "123");
    }

    public void testPushCheckForSharingTwixtStacks() {
        RatPolyStack stk1 = stack();
        RatPolyStack stk2 = stack("123");
        assertStackIs(stk1, "");
        assertStackIs(stk2, "123");

        stk1.push(constantPoly(0));
        assertStackIs(stk1, "0");
        assertStackIs(stk2, "123");

        stk1.push(constantPoly(0));
        assertStackIs(stk1, "00");
        assertStackIs(stk2, "123");

        stk1.push(constantPoly(1));
        assertStackIs(stk1, "100");
        assertStackIs(stk2, "123");

        stk2.push(constantPoly(8));
        assertStackIs(stk1, "100");
        assertStackIs(stk2, "8123");
    }

    public void testPop() {
        RatPolyStack stk1 = stack("123");

        RatPoly poly = stk1.pop();
        assertTrue(eqv(poly, constantPoly(1)));
        assertStackIs(stk1, "23");

        poly = stk1.pop();
        assertTrue(eqv(poly, constantPoly(2)));
        assertStackIs(stk1, "3");

        poly = stk1.pop();
        assertStackIs(stk1, "");
    }

    public void testDup() {
        RatPolyStack stk1 = stack("3");
        stk1.dup();
        assertStackIs(stk1, "33");
```

```
        stk1 = stack("23");
        stk1.dup();
        assertStackIs(stk1, "223");
        assertTrue(stk1.size() == 3);
        assertTrue(eqv(stk1.get(0), constantPoly(2)));
        assertTrue(eqv(stk1.get(1), constantPoly(2)));
        assertTrue(eqv(stk1.get(2), constantPoly(3)));

        stk1 = stack("123");
        stk1.dup();
        assertStackIs(stk1, "1123");

    }

    public void testSwap() {
        RatPolyStack stk1 = stack("23");
        stk1.swap();
        assertStackIs(stk1, "32");

        stk1 = stack("123");
        stk1.swap();
        assertStackIs(stk1, "213");

        stk1 = stack("112");
        stk1.swap();
        assertStackIs(stk1, "112");
    }

    public void testClear() {
        RatPolyStack stk1 = stack("123");
        stk1.clear();
        assertStackIs(stk1, "");
        RatPolyStack stk2 = stack("112");
        stk2.clear();
        assertStackIs(stk2, "");
    }

    public void testAdd() {
        RatPolyStack stk1 = stack("123");
        stk1.add();
        assertStackIs(stk1, "33");
        stk1.add();
        assertStackIs(stk1, "6");

        stk1 = stack("112");
        stk1.add();
        assertStackIs(stk1, "22");
        stk1.add();
        assertStackIs(stk1, "4");
        stk1.push(constantPoly(5));
        assertStackIs(stk1, "54");
        stk1.add();
        assertStackIs(stk1, "9");

    }

    public void testSub() {
        RatPolyStack stk1 = stack("123");
        stk1.sub();
        assertStackIs(stk1, "13");
        stk1.sub();
        assertStackIs(stk1, "2");

        stk1 = stack("5723");
        stk1.sub();
        assertStackIs(stk1, "223");
        stk1.sub();
        assertStackIs(stk1, "03");
```



```
        stk1.sub();
        assertStackIs(stk1, "3");
    }

    public void testMul() {
        RatPolyStack stk1 = stack("123");
        stk1.mul();
        assertStackIs(stk1, "23");
        stk1.mul();
        assertStackIs(stk1, "6");

        stk1 = stack("112");
        stk1.mul();
        assertStackIs(stk1, "12");
        stk1.mul();
        assertStackIs(stk1, "2");
        stk1.push(constantPoly(4));
        assertStackIs(stk1, "42");
        stk1.mul();
        assertStackIs(stk1, "8");
    }

    public void testDiv() {
        RatPolyStack stk1 = stack("123");
        stk1.div();
        assertStackIs(stk1, "23");
    }

    public void testDifferentiate() {
        RatPolyStack stk1 = stack("123");
        stk1.differentiate();
        stk1.differentiate();
        stk1.differentiate();
        stk1.differentiate();
        assertTrue ("Test if stack size changes", stk1.size() == 3);
        assertStackIs (stk1, "023");

        RatPoly rp1 = new RatPoly(3, 5);
        RatPoly rp2 = new RatPoly(7, 0);
        RatPoly rp3 = new RatPoly (4,1);
        stk1.push (rp1);
        stk1.push (rp2);
        stk1.push (rp3);

        stk1.differentiate();
        assertTrue ("Test simple differentiate1", stk1.pop().unparse().equals ("4"));
        stk1.differentiate();
        assertTrue ("Test simple differentiate2", stk1.pop().unparse().equals ("0"));
        stk1.differentiate();
        assertTrue ("Test simple differentiate3", stk1.pop().unparse().equals ("15*x^4"))
;
    }

    public void testIntegrate() {
        RatPolyStack stk1 = stack("123");
        stk1.integrate();
        stk1.integrate();
        stk1.integrate();
        stk1.integrate();
        assertTrue ("Test if stack size changes", stk1.size() == 3);
        assertTrue ("Test simple integrate1", stk1.pop().unparse().equals ("1/24*x^4"));
        RatPoly rp1 = new RatPoly(15, 4);
        RatPoly rp2 = new RatPoly(7, 0);
        RatPoly rp3 = new RatPoly (4,0);
        stk1.push (rp1);
        stk1.push (rp2);
        stk1.push (rp3);

        stk1.integrate();
```

```
        assertTrue ("Test simple integrate1", stk1.pop().unparse().equals ("4*x"));
        stk1.integrate();
        assertTrue ("Test simple integrate2", stk1.pop().unparse().equals ("7*x"));
        stk1.integrate();
        assertTrue ("Test simple integrate3", stk1.pop().unparse().equals ("3*x^5"));
    }


    // Tell JUnit what order to run the tests in
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(new RatPolyStackTest("testCtor"));
        suite.addTest(new RatPolyStackTest("testPush"));
        suite.addTest(
                                    new RatPolyStackTest("testPushCheckForSharingTwixt
Stacks"));
        suite.addTest(new RatPolyStackTest("testPop"));
        suite.addTest(new RatPolyStackTest("testDup"));
        suite.addTest(new RatPolyStackTest("testSwap"));
        suite.addTest(new RatPolyStackTest("testClear"));
        suite.addTest(new RatPolyStackTest("testAdd"));
        suite.addTest(new RatPolyStackTest("testSub"));
        suite.addTest(new RatPolyStackTest("testMul"));
        suite.addTest(new RatPolyStackTest("testDiv"));
        suite.addTest(new RatPolyStackTest("testDifferentiate"));
        suite.addTest(new RatPolyStackTest("testIntegrate"));
        return suite;
    }
}
```



```
package ps2;

import junit.framework.*;

/** This class contains a set of test cases that can be used to
          test the implementation of the RatPoly class.
                  <p>
*/
public class RatPolyTest extends TestCase {
    // get a RatNum for an integer
    private RatNum num(int i) {
        return new RatNum(i);
    }

    private RatNum nanNum = (new RatNum(1)).div(new RatNum(0));

    // convenient way to make a RatPoly
    private RatPoly poly(int coef, int expt) {
        return new RatPoly(coef, expt);
    }

    // Convenient way to make a quadratic polynomial, arguments
    // are just the coefficients, highest degree term to lowest
    private RatPoly quadPoly(int x2, int x1, int x0) {
        RatPoly ratPoly = new RatPoly(x2, 2);
        return ratPoly.add(poly(x1, 1)).add(poly(x0, 0));
    }

    // convenience for parse
    private RatPoly parse(String s) {
        return RatPoly.parse(s);
    }

    // convenience for zero RatPoly
    private RatPoly zero() {
        return new RatPoly();
    }

    public RatPolyTest(String name) {
        super(name);
    }

    // only unparse is tested here
    private void eq(RatPoly p, String target) {
        String t = p.unparse();
        assertEquals(target, t);
    }

    private void eq(RatPoly p, String target, String message) {
        String t = p.unparse();
        assertEquals(message, target, t);
    }

    // parses s into p, and then checks that it is as anticipated
    // forall i, parse(s).coeff(anticipDegree - i) = anticipCoeffForExpts(i)
    //   (anticipDegree - i) means that we expect coeffs to be expressed
    //   corresponding to decreasing expts
    private void eqP(String s, int anticipDegree, RatNum[] anticipCoeffs) {
        RatPoly p = parse(s);
        assertTrue(p.degree() == anticipDegree);
        for (int i = 0; i <= anticipDegree; i++) {
            assertTrue(
                                    "wrong coeff; \n"
                                    + "anticipated: "
                                    + anticipCoeffs[i]
                                    + "; received: "
                                    + p.coeff(anticipDegree - i)
                                    + "\n"
                                    + " received: "
```

```
                                + p
                                + " anticipated:"
                                + s,
                                p.coeff(anticipDegree - i).equals(anticipCoeffs
[i]));
        }
    }

    // added convenience: express coeffs as ints
    private void eqP(String s, int anticipDegree, int[] intCoeffs) {
        RatNum[] coeffs = new RatNum[intCoeffs.length];
        for (int i = 0; i < coeffs.length; i++) {
            coeffs[i] = num(intCoeffs[i]);
        }
        eqP(s, anticipDegree, coeffs);
    }

    // make sure that unparsing a parsed string yields the string itself
    private void assertUnparseWorks(String s) {
        assertEquals(s, parse(s).unparse());
    }

    public void testParseSimple() {
        eqP("0", 0, new int[] { 0 });
        eqP("x", 1, new int[] { 1, 0 });
        eqP("x^2", 2, new int[] { 1, 0, 0 });
    }

    public void testParseMultTerms() {
        eqP("x^3+x^2", 3, new int[] { 1, 1, 0, 0 });
        eqP("x^3-x^2", 3, new int[] { 1, -1, 0, 0 });
        eqP("x^10+x^2", 10, new int[] { 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 });
    }

    public void testParseLeadingNeg() {
        eqP("-x^2", 2, new int[] { -1, 0, 0 });
        eqP("-x^2+1", 2, new int[] { -1, 0, 1 });
        eqP("-x^2+x", 2, new int[] { -1, 1, 0 });
    }

    public void testParseLeadingConstants() {
        eqP("10*x", 1, new int[] { 10, 0 });

        eqP("10*x^4+x^2", 4, new int[] { 10, 0, 1, 0, 0 });

        eqP("10*x^4+100*x^2", 4, new int[] { 10, 0, 100, 0, 0 });

        eqP("-10*x^4+100*x^2", 4, new int[] { -10, 0, 100, 0, 0 });
    }

    public void testParseRationals() {
        eqP("1/2", 0, new RatNum[] { num(1).div(num(2)) });
        eqP("1/2*x", 1, new RatNum[] { num(1).div(num(2)), num(0) });
        eqP("x+1/3", 1, new RatNum[] { num(1), num(1).div(num(3)) });
        eqP(
                "1/2*x+1/3",
                1,
                new RatNum[] { num(1).div(num(2)), num(1).div(num(3)) });
        eqP(
                "1/2*x+3/2",
                1,
                new RatNum[] { num(1).div(num(2)), num(3).div(num(2)) });
        eqP(
                "1/2*x^3+3/2",
                3,
                new RatNum[] {
                    num(1).div(num(2)),
                    num(0),
                    num(0),
```



```
                    num(3).div(num(2)) });
        eqP(
                "1/2*x^3+3/2*x^2+1",
                3,
                new RatNum[] {
                    num(1).div(num(2)),
                    num(3).div(num(2)),
                    num(0),
                    num(1) });
    }

    public void testParseNaN() {
        assertTrue(parse("NaN").isNaN());
    }

    public void testUnparseSimple() {
        assertUnparseWorks("0");
        assertUnparseWorks("x");
        assertUnparseWorks("x^2");
    }

    public void testUnparseMultTerms() {
        assertUnparseWorks("x^3+x^2");
        assertUnparseWorks("x^3-x^2");
        assertUnparseWorks("x^100+x^2");
    }

    public void testUnparseLeadingNeg() {
        assertUnparseWorks("-x^2");
        assertUnparseWorks("-x^2+1");
        assertUnparseWorks("-x^2+x");
    }

    public void testUnparseLeadingConstants() {
        assertUnparseWorks("10*x");
        assertUnparseWorks("10*x^100+x^2");
        assertUnparseWorks("10*x^100+100*x^2");
        assertUnparseWorks("-10*x^100+100*x^2");
    }

    public void testUnparseRationals() {
        assertUnparseWorks("1/2");
        assertUnparseWorks("1/2*x");
        assertUnparseWorks("x+1/3");
        assertUnparseWorks("1/2*x+1/3");
        assertUnparseWorks("1/2*x+3/2");
        assertUnparseWorks("1/2*x^10+3/2");
        assertUnparseWorks("1/2*x^10+3/2*x^2+1");
    }

    public void testUnparseNaN() {
        assertUnparseWorks("NaN");
    }

    public void testNoArgCtor() {
        eq(new RatPoly(), "0");
    }

    public void testTwoArgCtor() {
        eq(poly(0, 0), "0");
        eq(poly(0, 1), "0");
        eq(poly(1, 0), "1");
        eq(poly(-1, 0), "-1");
        eq(poly(1, 1), "x");
        eq(poly(1, 2), "x^2");
        eq(poly(2, 2), "2*x^2");
        eq(poly(2, 3), "2*x^3");
        eq(poly(-2, 3), "-2*x^3");
        eq(poly(-1, 1), "-x");
```

```
        eq(poly(-1, 3), "-x^3");
    }

    public void testDegree() {
        assertTrue("x^0 degree 0", poly(1, 0).degree() == 0);
        assertTrue("x^1 degree 1", poly(1, 1).degree() == 1);
        assertTrue("x^100 degree 100", poly(1, 100).degree() == 100);
        assertTrue("0*x^100 degree 0", poly(0, 100).degree() == 0);
        assertTrue("0*x^0 degree 0", poly(0, 0).degree() == 0);
    }

    public void testAdd() {
        RatPoly _XSqPlus2X = poly(1, 2).add(poly(1, 1)).add(poly(1, 1));
        RatPoly _2XSqPlusX = poly(1, 2).add(poly(1, 2)).add(poly(1, 1));

        eq(poly(1, 0).add(poly(1, 0)), "2");
        eq(poly(1, 0).add(poly(5, 0)), "6");
        eq(poly(1, 1).add(poly(1, 1)), "2*x");
        eq(poly(1, 2).add(poly(1, 2)), "2*x^2");
        eq(poly(1, 2).add(poly(1, 1)), "x^2+x");
        eq(_XSqPlus2X, "x^2+2*x");
        eq(_2XSqPlusX, "2*x^2+x");
        eq(poly(1, 3).add(poly(1, 1)), "x^3+x");
    }

    public void testSub() {
        eq(poly(1, 1).sub(poly(1, 0)), "x-1");
        eq(poly(1, 1).add(poly(1, 0)), "x+1");
    }

    public void testMul() {
        eq(poly(0, 0).mul(poly(0, 0)), "0");
        eq(poly(1, 0).mul(poly(1, 0)), "1");
        eq(poly(1, 0).mul(poly(2, 0)), "2");
        eq(poly(2, 0).mul(poly(2, 0)), "4");
        eq(poly(1, 0).mul(poly(1, 1)), "x");
        eq(poly(1, 1).mul(poly(1, 1)), "x^2");
        eq(poly(1, 1).sub(poly(1, 0)).mul(poly(1, 1).add(poly(1, 0))), "x^2-1");
    }

    public void testOpsWithNaN(RatPoly p) {
        RatPoly nan = RatPoly.parse("NaN");
        eq(p.add(nan), "NaN");
        eq(nan.add(p), "NaN");
        eq(p.sub(nan), "NaN");
        eq(nan.sub(p), "NaN");
        eq(p.mul(nan), "NaN");
        eq(nan.mul(p), "NaN");
        eq(p.div(nan), "NaN");
        eq(nan.div(p), "NaN");
    }

    public void testOpsWithNaN() {
        testOpsWithNaN(poly(0, 0));
        testOpsWithNaN(poly(0, 1));
        testOpsWithNaN(poly(1, 0));
        testOpsWithNaN(poly(1, 1));
        testOpsWithNaN(poly(2, 0));
        testOpsWithNaN(poly(2, 1));
        testOpsWithNaN(poly(0, 2));
        testOpsWithNaN(poly(1, 2));
    }

    public void testImmutabilityOfOperations() {
        // not the most thorough test possible, but hopefully will
        // catch the easy cases early on...
        RatPoly one = poly(1, 0);
        RatPoly two = poly(2, 0);
```



```
        one.degree();
        two.degree();
        eq(one, "1", "Degree mutates receiver!");
        eq(two, "2", "Degree mutates receiver!");

        one.coeff(0);
        two.coeff(0);
        eq(one, "1", "Coeff mutates receiver!");
        eq(two, "2", "Coeff mutates receiver!");

        one.isNaN();
        two.isNaN();
        eq(one, "1", "isNaN mutates receiver!");
        eq(two, "2", "isNaN mutates receiver!");

        one.eval(0.0);
        two.eval(0.0);
        eq(one, "1", "eval mutates receiver!");
        eq(two, "2", "eval mutates receiver!");

        one.negate();
        two.negate();
        eq(one, "1", "Negate mutates receiver!");
        eq(two, "2", "Negate mutates receiver!");

        one.add(two);
        eq(one, "1", "Add mutates receiver!");
        eq(two, "2", "Add mutates argument!");

        one.sub(two);
        eq(one, "1", "Sub mutates receiver!");
        eq(two, "2", "Sub mutates argument!");

        one.mul(two);
        eq(one, "1", "Mul mutates receiver!");
        eq(two, "2", "Mul mutates argument!");

        one.div(two);
        eq(one, "1", "Div mutates receiver!");
        eq(two, "2", "Div mutates argument!");
    }

    public void testEval() {
        RatPoly zero = new RatPoly();
        RatPoly one = new RatPoly(1, 0);
        RatPoly _X = new RatPoly(1, 1);
        RatPoly _2X = new RatPoly(2, 1);
        RatPoly _XSq = new RatPoly(1, 2);

        assertEquals(" 0 at 0 ", 0.0, zero.eval(0.0), 0.0001);
        assertEquals(" 0 at 1 ", 0.0, zero.eval(1.0), 0.0001);
        assertEquals(" 0 at 2 ", 0.0, zero.eval(2.0), 0.0001);
        assertEquals(" 1 at 0 ", 1.0, one.eval(0.0), 0.0001);
        assertEquals(" 1 at 1 ", 1.0, one.eval(1.0), 0.0001);
        assertEquals(" 1 at 1 ", 1.0, one.eval(2.0), 0.0001);

        assertEquals(" x at 0 ", 0.0, _X.eval(0.0), 0.0001);
        assertEquals(" x at 1 ", 1.0, _X.eval(1.0), 0.0001);
        assertEquals(" x at 2 ", 2.0, _X.eval(2.0), 0.0001);

        assertEquals(" 2*x at 0 ", 0.0, _2X.eval(0.0), 0.0001);
        assertEquals(" 2*x at 1 ", 2.0, _2X.eval(1.0), 0.0001);
        assertEquals(" 2*x at 2 ", 4.0, _2X.eval(2.0), 0.0001);

        assertEquals(" x^2 at 0 ", 0.0, _XSq.eval(0.0), 0.0001);
        assertEquals(" x^2 at 1 ", 1.0, _XSq.eval(1.0), 0.0001);
        assertEquals(" x^2 at 2 ", 4.0, _XSq.eval(2.0), 0.0001);

        RatPoly _XSq_minus_2X = _XSq.sub(_2X);
```

```
        assertEquals("x^2-2*x at 0", 0.0, _XSq_minus_2X.eval(0.0), 0.0001);
        assertEquals("x^2-2*x at 1", -1.0, _XSq_minus_2X.eval(1.0), 0.0001);
        assertEquals("x^2-2*x at 2", 0.0, _XSq_minus_2X.eval(2.0), 0.0001);
        assertEquals("x^2-2*x at 3", 3.0, _XSq_minus_2X.eval(3.0), 0.0001);
    }

    public void testCoeff() {
        // coeff already gets some grunt testing in eqP; checking an interesting
        // input here...
        RatPoly _XSqPlus2X = poly(1, 2).add(poly(1, 1)).add(poly(1, 1));
        RatPoly _2XSqPlusX = poly(1, 2).add(poly(1, 2)).add(poly(1, 1));

        assertTrue(_XSqPlus2X.coeff(-1).equals(num(0)));
        assertTrue(_XSqPlus2X.coeff(-10).equals(num(0)));
        assertTrue(_2XSqPlusX.coeff(-1).equals(num(0)));
        assertTrue(_2XSqPlusX.coeff(-10).equals(num(0)));
        assertTrue(zero().coeff(-10).equals(num(0)));
        assertTrue(zero().coeff(-1).equals(num(0)));
    }

    public void testDiv() {
        // 0/x = 0
        eq(poly(0, 1).div(poly(1, 1)), "0");

        // x/x = 1
        eq(poly(1, 1).div(poly(1, 1)), "1");

        // -x/x = -1
        eq(poly(-1, 1).div(poly(1, 1)), "-1");

        // x/-x = -1
        eq(poly(1, 1).div(poly(-1, 1)), "-1");

        // -x/-x = 1
        eq(poly(-1, 1).div(poly(-1, 1)), "1");

        // -x^2/x = -x
        eq(poly(-1, 2).div(poly(1, 1)), "-x");

        // x^100/x^1000 = 0
        eq(poly(1, 100).div(poly(1, 1000)), "0");

        // x^100/x = x^99
        eq(poly(1, 100).div(poly(1, 1)), "x^99");

        // x^99/x^98 = x
        eq(poly(1, 99).div(poly(1, 98)), "x");

        // x^10 / x = x^9 (r: 0)
        eq(poly(1, 10).div(poly(1, 1)), "x^9");

        // x^10 / x^3+x^2 = x^7-x^6+x^5-x^4+x^3-x^2+x-1  (r: -x^2)
        eq(
            poly(1, 10).div(poly(1, 3).add(poly(1, 2))),
            "x^7-x^6+x^5-x^4+x^3-x^2+x-1");

        // x^10 / x^3+x^2+x = x^7-x^6+x^4-x^3+x-1 (r: -x)
        eq(
            poly(1, 10).div(poly(1, 3).add(poly(1, 2).add(poly(1, 1)))),
            "x^7-x^6+x^4-x^3+x-1");

        // x^10+x^5 / x = x^9+x^4 (r: 0)
        eq(poly(1, 10).add(poly(1, 5)).div(poly(1, 1)), "x^9+x^4");

        // x^10+x^5 / x^3 = x^7+x^2 (r: 0)
        eq(poly(1, 10).add(poly(1, 5)).div(poly(1, 3)), "x^7+x^2");

        // x^10+x^5 / x^3+x+3 = x^7-x^5-3*x^4+x^3+7*x^2+8*x-10 (r: 29*x^2+14*x-3
```



```
0)
        eq(
            poly(1, 10).add(poly(1, 5)).div(
                                                                        pol
y(1, 3).add(poly(1, 1)).add(poly(3, 0))),
            "x^7-x^5-3*x^4+x^3+7*x^2+8*x-10");
    }

    public void testDivComplexI() {
        // (x+1)*(x+1) = x^2+2*x+1
        eq(
            poly(1, 2).add(poly(2, 1)).add(poly(1, 0)).div(

                    poly(1, 1).add(poly(1, 0))),
            "x+1");

        // (x-1)*(x+1) = x^2-1
        eq(poly(1, 2).add(poly(-1, 0)).div(poly(1, 1).add(poly(1, 0))), "x-1");
    }

    public void testDivComplexII() {
        // x^8+2*x^6+8*x^5+2*x^4+17*x^3+11*x^2+8*x+3 =
        // (x^3+2*x+1) * (x^5+7*x^2+2*x+3)
        RatPoly 1arge =
            poly(1, 8)
            .add(poly(2, 6))
            .add(poly(8, 5))
            .add(poly(2, 4))
            .add(poly(17, 3))
            .add(poly(11, 2))
            .add(poly(8, 1))
            .add(poly(3, 0));

        // x^3+2*x+1
        RatPoly sub1 = poly(1, 3).add(poly(2, 1)).add(poly(1, 0));
        // x^5+7*x^2+2*x+3
        RatPoly sub2 =
            poly(1, 5).add(poly(7, 2)).add(poly(2, 1)).add(poly(3, 0));

        // just a last minute typo check...
        eq(sub1.mul(sub2), large.unparse());
        eq(sub2.mul(sub1), large.unparse());

        eq(large.div(sub2), "x^3+2*x+1");
        eq(large.div(sub1), "x^5+7*x^2+2*x+3");
    }

    public void testDivExamplesFromSpec() {
        // seperated this test case out because it has a dependency on
        // both "parse" and "div" functioning properly

        // example 1 from spec
        eq(parse("x^3-2*x+3").div(parse("3*x^2")), "1/3*x");
        // example 2 from spec
        eq(parse("x^2+2*x+15").div(parse("2*x^3")), "0");
    }

    public void testDivExampleFromPset() {
        eq(
            parse("x^8+x^6+10*x^4+10*x^3+8*x^2+2*x+8").div(

                    parse("3*x^6+5*x^4+9*x^2+4*x+8")),
            "1/3*x^2-2/9");
    }

    private void assertIsNaNanswer(RatPoly nanAnswer) {
        eq(nanAnswer, "NaN");
    }
    public void testDivByZero() {
```

```
        RatPoly nanAnswer;
        nanAnswer = poly(1, 0).div(zero());
        assertIsNaNanswer(nanAnswer);

        nanAnswer = poly(1, 1).div(zero());
        assertIsNaNanswer(nanAnswer);
    }

    public void testDivByPolyWithNaN() {
        RatPoly nan_x2 = poly(1, 2).mul(poly(1, 1).div(zero()));
        RatPoly one_x1 = new RatPoly(1, 1);

        assertIsNaNanswer(nan_x2.div(one_x1));
        assertIsNaNanswer(one_x1.div(nan_x2));
        assertIsNaNanswer(nan_x2.div(zero()));
        assertIsNaNanswer(zero().div(nan_x2));
        assertIsNaNanswer(nan_x2.div(nan_x2));
    }

    public void testIsNaN() {
        assertTrue(RatPoly.parse("NaN").isNaN());
        assertTrue(!RatPoly.parse("1").isNaN());
        assertTrue(!RatPoly.parse("1/2").isNaN());
        assertTrue(!RatPoly.parse("x+1").isNaN());
        assertTrue(!RatPoly.parse("x^2+x+1").isNaN());
    }

    public void testDifferentiate() {
        eq(poly(1, 1).differentiate(), "1");
        eq(quadPoly(7, 5, 99).differentiate(), "14*x+5");
        eq(quadPoly(3, 2, 1).differentiate(), "6*x+2");
        eq(quadPoly(1, 0, 1).differentiate(), "2*x");
        assertIsNaNanswer(RatPoly.parse("NaN").differentiate());
    }

    public void testAntiDifferentiate() {
        eq(poly(1, 0).antiDifferentiate(new RatNum(1)), "x+1");
        eq(poly(2, 1).antiDifferentiate(new RatNum(1)), "x^2+1");
        eq(quadPoly(0, 6, 2).antiDifferentiate(new RatNum(1)), "3*x^2+2*x+1");
        eq(quadPoly(4, 6, 2).antiDifferentiate(new RatNum(0)), "4/3*x^3+3*x^2+2*x")
;

        assertIsNaNanswer(RatPoly.parse("NaN").antiDifferentiate(new RatNum(1))
);
        assertIsNaNanswer(poly(1, 0).antiDifferentiate(new RatNum(1, 0)));
    }

    public void testIntegrate() {
        assertEquals("one from 0 to 1", 1.0, poly(1,0).integrate(0,1), 0.0001 );
        assertEquals("2x from 1 to -2", 3.0, poly(2,1).integrate(1,-2), 0.0001 );
        assertEquals("7*x^2+6*x+2 from 1 to 5", 369.33333333, quadPoly(7,6,2).integra
te(1,5), 0.0001 );
    }

    // Tell JUnit what order to run the tests in
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(new RatPolyTest("testParseSimple"));
        suite.addTest(new RatPolyTest("testParseMultTerms"));
        suite.addTest(new RatPolyTest("testParseLeadingNeg"));
        suite.addTest(new RatPolyTest("testParseLeadingConstants"));
        suite.addTest(new RatPolyTest("testParseRationals"));
        suite.addTest(new RatPolyTest("testParseNaN"));
        suite.addTest(new RatPolyTest("testUnparseSimple"));
        suite.addTest(new RatPolyTest("testUnparseMultTerms"));
        suite.addTest(new RatPolyTest("testUnparseLeadingNeg"));
        suite.addTest(new RatPolyTest("testUnparseLeadingConstants"));
        suite.addTest(new RatPolyTest("testUnparseRationals"));
        suite.addTest(new RatPolyTest("testUnparseNaN"));
```



```
        suite.addTest(new RatPolyTest("testNoArgCtor"));
        suite.addTest(new RatPolyTest("testTwoArgCtor"));
        suite.addTest(new RatPolyTest("testDegree"));
        suite.addTest(new RatPolyTest("testAdd"));
        suite.addTest(new RatPolyTest("testSub"));
        suite.addTest(new RatPolyTest("testMul"));
        suite.addTest(new RatPolyTest("testOpsWithNaN"));
        suite.addTest(new RatPolyTest("testCoeff"));
        suite.addTest(new RatPolyTest("testDiv"));
        suite.addTest(new RatPolyTest("testDivComplexI"));
        suite.addTest(new RatPolyTest("testDivComplexII"));
        suite.addTest(new RatPolyTest("testDivExamplesFromSpec"));
        suite.addTest(new RatPolyTest("testDivExampleFromPset"));
        suite.addTest(new RatPolyTest("testDivByZero"));
        suite.addTest(new RatPolyTest("testDivByPolyWithNaN"));
        suite.addTest(new RatPolyTest("testIsNaN"));
        suite.addTest(new RatPolyTest("testEval"));
        suite.addTest(new RatPolyTest("testImmutabilityOfOperations"));
        suite.addTest(new RatPolyTest("testDifferentiate"));
        suite.addTest(new RatPolyTest("testAntiDifferentiate"));
        suite.addTest(new RatPolyTest("testIntegrate"));
        return suite;
    }
}
```