0.1 API

```
package com.ncalathus.sample.ring.api;

public interface IAbstractRing {

    public interface CLASS<REP extends INST> {

        REP zero();

        REP unit();

        REP neg_unit();
   }

    public interface INST<REP extends INST> {

        CLASS<REP> CLASS();

        REP add(REP elem);

        REP subst(REP elem);

        REP mult(REP elem);
    }
}
```

```
package com.ncalathus.sample.ring.api;

public interface INumericRing {
    interface CLASS<N extends Number, REP extends INST<N, REP>> extends IAbst
        REP create(N n);
    }

    interface INST<N extends Number, REP extends INST<N, REP>> extends IAbstra

        @Override
        CLASS<N, REP> CLASS();
    }
}
```

```
package com.ncalathus.sample.ring.api;

import java.util.Map;

public interface IPolynomialRing {
    interface CLASS<
            C_INST extends IAbstractRing.INST<C_INST>, C_CLASS extends IAbstra
            REP extends INST<C_INST, C_CLASS, REP>
        > extends IAbstractRing.CLASS<REP> {

        C_CLASS coefficientRing();

        REP variable(int degree);

        REP create(int degree, C_INST c);

        REP create(final Map<Integer, C_INST> terms);

        REP create(final C_INST... coeffs);

    }

    interface INST<
            C_INST extends IAbstractRing.INST<C_INST>, C_CLASS extends IAbstra
            REP extends INST<C_INST, C_CLASS, REP>
        > extends IAbstractRing.INST<REP> {

        @Override
        CLASS<C_INST, C_CLASS, REP> CLASS();

        int degree();

        C_INST coefficient(int degree);

        REP r_mult(C_INST coeff);
    }

}
```

0.2 Representation

```
package com.ncalathus.sample.ring.rep;

import com.ncalathus.sample.ring.api.*;

// this class allows the all elements of NumericRing to have the same class w
// Meta class must be used to share the same generic type for static and insto
// Also these static/instance has recuresive references, so outer classes mus
// (see NumricRing in the INumericRing_Static<N, NumericRing> for NumericRing
public class NumericRing<N extends Number> implements IAbstractRing {

    protected final Class<N> _nCls;
    protected final CLASS _class;

    protected NumericRing(final Class<N> nCls) {
        this._nCls = nCls;
        this._class = new CLASS();
    }

    // this is an entry point.
    // all instance of NumericRing are cretated from NumericRingStatic
    // this gurantee, those elements has the same element type.
    public CLASS getCLASS() {
        return _class;
    }

    public static <N extends Number> NumericRing<N>.CLASS CLASS(Class<N> cls)
        return new NumericRing<N>(cls).getCLASS();
    }

    //
    public class CLASS implements INumericRing.CLASS<N, INST> {
        private final INST _zero;
        private final INST _unit;
        private final INST _neg_unit;

        CLASS() {
            this._zero = new INST(create_zero(_nCls));
            this._unit = new INST(create_unit(_nCls));
            this._neg_unit = new INST(create_neg_unit(_nCls));
        }

        public INST zero() { return _zero; }
        public INST unit() { return _unit; }
        public INST neg_unit() { return _neg_unit; }

        public INST create(N n) {
```

```
    return new INST(n);
}

final N create_zero(Class<N> nCls) {
    if (nCls.equals(Short.class)) {
        return (N)new Short((short)0);
    } else if (nCls.equals(Integer.class)) {
        return (N)new Integer(0);
    } else if (nCls.equals(Long.class)) {
        return (N)new Long(0);
    } else if (nCls.equals(Float.class)) {
        return (N)new Float(0);
    } else if (nCls.equals(Double.class)) {
        return (N)new Double(0);
    } else {
        throw new RuntimeException("bug");
    }
}
private N create_unit(Class<N> nCls) {
    if (nCls.equals(Short.class)) {
        return (N)new Short((short)1);
    } else if (nCls.equals(Integer.class)) {
        return (N)new Integer(1);
    } else if (nCls.equals(Long.class)) {
        return (N)new Long(1);
    } else if (nCls.equals(Float.class)) {
        return (N)new Float(1);
    } else if (nCls.equals(Double.class)) {
        return (N)new Double(1);
    } else {
        throw new RuntimeException("bug");
    }
}
private N create_neg_unit(Class<N> nCls) {
    if (nCls.equals(Short.class)) {
        return (N)new Short((short)(-1));
    } else if (nCls.equals(Integer.class)) {
        return (N)new Integer(-1);
    } else if (nCls.equals(Long.class)) {
        return (N)new Long(-1);
    } else if (nCls.equals(Float.class)) {
        return (N)new Float(-1);
    } else if (nCls.equals(Double.class)) {
        return (N)new Double(-1);
    } else {
        throw new RuntimeException("bug");
```

```
        }
    }
}

public class INST implements INumericRing.INST<N, INST> {
    // this is to enforce the interface requirement.
    @Override
    public CLASS CLASS() {
        return _class;
    }

    private final N i;

    INST(final N i) {
        this.i = i;
    }

    @Override
    public INST add(final INST elem) {
        return new INST(add(i, elem.i));
    }
    @Override
    public INST subst(final INST elem) {
        return new INST(subst(i, elem.i));
    }
    @Override
    public INST mult(final INST elem) {
        return new INST(mult(i, elem.i));
    }

    @Override
    public String toString() {
        return ""+i;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof NumericRing.INST) {
            INST inst = (INST)obj;
            return i.equals(inst.i);
        } else {
            return false;
        }
    }

    @Override
```

```
public int hashCode() {
    return i.hashCode();
}

//
private N add(final N n1, final N n2) {
    if (n1 == null || n2 == null) {
        throw new RuntimeException("add for null is not supported");
    }
    if (n1 instanceof Short) {
        final short i1 = (Short)n1;
        final short i2 = (Short)n2;
        return (N)new Short((short)(i1+i2));
    } else if (n1 instanceof Integer) {
        final int i1 = (Integer)n1;
        final int i2 = (Integer)n2;
        return (N)new Integer(i1+i2);
    } else if (n1 instanceof Long) {
        final long i1 = (Long)n1;
        final long i2 = (Long)n2;
        return (N)new Long(i1+i2);
    } else if (n1 instanceof Long) {
        final long i1 = (Long)n1;
        final long i2 = (Long)n2;
        return (N)new Long(i1+i2);
    } else if (n1 instanceof Float) {
        final float i1 = (Float)n1;
        final float i2 = (Float)n2;
        return (N)new Float(i1+i2);
    } else if (n1 instanceof Double) {
        final double i1 = (Double)n1;
        final double i2 = (Double)n2;
        return (N)new Double(i1+i2);
    } else {
        throw new RuntimeException("bug");
    }
}
private N subst(final N n1, final N n2) {
    if (n1 == null || n2 == null) {
        throw new RuntimeException("add for null is not supported");
    }
    if (n1 instanceof Short) {
        final short i1 = (Short)n1;
        final short i2 = (Short)n2;
        return (N)new Short((short)(i1-i2));
    } else if (n1 instanceof Integer) {
```

```
            final int i1 = (Integer)n1;
            final int i2 = (Integer)n2;
            return (N)new Integer(i1-i2);
        } else if (n1 instanceof Long) {
            final long i1 = (Long)n1;
            final long i2 = (Long)n2;
            return (N)new Long(i1-i2);
        } else if (n1 instanceof Long) {
            final long i1 = (Long)n1;
            final long i2 = (Long)n2;
            return (N)new Long(i1-i2);
        } else if (n1 instanceof Float) {
            final float i1 = (Float)n1;
            final float i2 = (Float)n2;
            return (N)new Float(i1-i2);
        } else if (n1 instanceof Double) {
            final double i1 = (Double)n1;
            final double i2 = (Double)n2;
            return (N)new Double(i1-i2);
        } else {
            throw new RuntimeException("bug");
        }
    }
    private N mult(final N n1, final N n2) {
        if (n1 == null || n2 == null) {
            throw new RuntimeException("add for null is not supported");
        }
        if (n1 instanceof Short) {
            final short i1 = (Short)n1;
            final short i2 = (Short)n2;
            return (N)new Short((short)(i1*i2));
        } else if (n1 instanceof Integer) {
            final int i1 = (Integer)n1;
            final int i2 = (Integer)n2;
            return (N)new Integer(i1*i2);
        } else if (n1 instanceof Long) {
            final long i1 = (Long)n1;
            final long i2 = (Long)n2;
            return (N)new Long(i1*i2);
        } else if (n1 instanceof Long) {
            final long i1 = (Long)n1;
            final long i2 = (Long)n2;
            return (N)new Long(i1*i2);
        } else if (n1 instanceof Float) {
            final float i1 = (Float)n1;
            final float i2 = (Float)n2;
```

```
                return (N)new Float(i1*i2);
            } else if (n1 instanceof Double) {
                final double i1 = (Double)n1;
                final double i2 = (Double)n2;
                return (N)new Double(i1*i2);
            } else {
                throw new RuntimeException("bug");
            }
        }
    }

    public static void main(String[] args) {
        //NumericRing<Integer> integerRing = new NumericRing<Integer>(Integer
        //final NumericRing<Integer>.CLASS IntegerRing = new NumericRing<Inte
        final NumericRing<Integer>.CLASS IntegerRing = NumericRing.CLASS(Inte
        final NumericRing<Integer>.INST nr1 = IntegerRing.create(7);
        final NumericRing<Integer>.INST nr2 = IntegerRing.create(9);
        final NumericRing<Integer>.INST nr3 = nr1.mult(nr2);
        System.out.println(">> nr3: "+nr3);
    }
}
```

```
package com.ncalathus.sample.ring.rep;

import com.ncalathus.sample.ring.api.IAbstractRing;
import com.ncalathus.sample.ring.api.IPolynomialRing;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class PolynomialRing<C_INST extends IAbstractRing.INST<C_INST>, C_CLAS

    protected final C_CLASS _coeffCls;
    protected CLASS _class;
    protected final String variable;

    protected PolynomialRing(final C_CLASS coeffCls, final String variable) {
        this._coeffCls = coeffCls;
        this.variable = variable;
        initCLASS();
    }

    // this should be overl;oaded by subclass if CLASS is redefined thee.
    protected void initCLASS() {
        this._class = new CLASS();
    }

    // this is an entry point.
    // all instance of NumericRing are cretated from NumericRingStatic
    // this gurantee, those elements has the same element type.
    //
    // this should be overl;oaded by subclass if CLASS is redefined thee.
    public CLASS getCLASS() {
        return _class;
    }

    public static <C_INST extends IAbstractRing.INST<C_INST>, C_CLASS extends
        return new PolynomialRing<C_INST, C_CLASS>(cls, variable).getCLASS();
    }

    public class Term {
        final int degree;
        final C_INST c;
        Term(final int degree, final C_INST c) {
            this.degree = degree;
```

```
        this.c = c;
    }

    Term r_mult(final C_INST c1) {
        final C_INST c2 = c.mult(c1);
        return new Term(degree, c2);
    }
    Term mult(final Term term) {
        return new Term(degree+term.degree, c.mult(term.c));
    }
    @Override
    public String toString() {
        return "("+c+")"+termString();
    }
    public String termString() {
        if (degree == 0) {
            return "";
        } else if (degree == 1) {
            return variable;
        } else {
            return variable+"^"+degree;
        }
    }
    public String toString(final String vars) {
        return "("+c+")"+termString(vars);
    }
    public String termString(final String vars) {
        if (degree == 0) {
            return vars;
        } else {
            return termString()+((vars.isEmpty())?"":("*"+vars));
        }
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof PolynomialRing.Term) {
            Term term = (Term)obj;
            return degree == term.degree && c == term.c;
        } else {
            return false;
        }
    }
    @Override
    public int hashCode() {
        return degree+137*c.hashCode();
```

```
    }
}

//
public class CLASS implements IPolynomialRing.CLASS<C_INST, C_CLASS, INST>
    private final INST _zero;
    private final INST _unit;
    private final INST _neg_unit;

    CLASS() {
        this._zero = create(0, _coeffCls.zero());
        this._unit = create(0, _coeffCls.unit());
        this._neg_unit = create(0, _coeffCls.neg_unit());
    }

    public INST zero() { return _zero; }
    public INST unit() { return _unit; }
    public INST neg_unit() { return _neg_unit; }

    @Override
    public C_CLASS coefficientRing() {
        return _coeffCls;
    }

    @Override
    public INST variable(int degree) {
        return new INST(degree, _coeffCls.unit());
    }

    @Override
    public final INST create(int degree, C_INST c) {
        return new INST(degree, c);
    }

    @Override
    public final INST create(final Map<Integer, C_INST> terms) {
        return new INST(new ArrayList<Term>() {{
            for (final Map.Entry<Integer, C_INST> term: terms.entrySet())
                add(new Term(term.getKey(), term.getValue()));
            }
        }});
    }

    @Override
    public final INST create(final C_INST... coeffs) {
        return new INST(new ArrayList<Term>() {{
```

```
            int degree = coeffs.length-1;
            for (int i = degree; i >= 0; i--) {
                final C_INST c = coeffs[i];
                if (!c.equals(_coeffCls.zero())) {
                    add(new Term(degree-i, c));
                }
            }
        }});
    }
}

public class INST implements IPolynomialRing.INST<C_INST, C_CLASS, INST>
    // this is to enforce the interface requirement.
    @Override
    public CLASS CLASS() {
        return _class;
    }

    private final List<Term> terms;

    protected INST(final int degree, final C_INST c) {
        this.terms = new ArrayList<Term>();
        terms.add(new Term(degree, c));
    }
    protected INST(final List<Term> terms) {
        this.terms = normalize(terms);
    }
    protected INST() {
        this.terms = new ArrayList<Term>();
    }

    private List<Term> normalize(List<Term> terms) {
        final Set<Integer> degrees = new HashSet<Integer>();
        for (final Term term: terms) {
            degrees.add(term.degree);
        }
        final List<Integer> sorted_degrees = new ArrayList<Integer>(degre
        Collections.sort(sorted_degrees, Collections.reverseOrder());
        final List<Term> norm = new ArrayList<Term>();
        for (final Integer degree: sorted_degrees) {
           final C_INST c0 = sumCoeff(degree, terms);
           if (!c0.equals(_coeffCls.zero())) {
               norm.add(new Term(degree, c0));
           }
        }
        return norm;
```

```
}
private C_INST sumCoeff(int degree, List<Term> terms) {
    C_INST c = _coeffCls.zero();
    for (final Term term: terms) {
        if (term.degree == degree) {
            c = c.add(term.c);
        }
    }
    return c;
}

@Override
public int degree() {
    if (terms.isEmpty()) {
        return 0;
    }
    return terms.get(0).degree;
}
@Override
public C_INST coefficient(int degree) {
    for (final Term term: terms) {
        if (term.degree == degree) {
            return term.c;
        }
    }
    return _coeffCls.zero();
}

@Override
public INST add(final INST elem) {
    return new INST(new ArrayList<Term>(){{
        addAll(INST.this.terms);
        addAll(elem.terms);
    }});
}
@Override
public INST subst(final INST elem) {
    return add(elem.r_mult(_coeffCls.neg_unit()));
}
@Override
public INST mult(final INST elem) {
    return new INST(new ArrayList<Term>(){{
        for (final Term term1: terms) {
            for (final Term term2: elem.terms) {
                add(term1.mult(term2));
            }
```

```
            }
        }});
    }

    @Override
    public INST r_mult(final C_INST c) {
        return new INST(new ArrayList<Term>(){{
            for (final Term term: terms) {
                add(term.r_mult(c));
            }
        }});
    }

    @Override
    public String toString() {
        final StringBuilder sb = new StringBuilder();
        return toString(sb, "");
    }

    public String toString(final StringBuilder sb, final String vars) {
        //final StringBuilder sb = new StringBuilder();
        boolean is_first = true;
        for (final Term term: terms) {
            if (is_first) {
                is_first = false;
            } else {
                sb.append("+");
            }

            if (term.c instanceof PolynomialRing.INST) {
                final INST inst = (PolynomialRing.INST)term.c;
                final String term0 = term.termString();
                final String vars0 = (vars.isEmpty())?term0:(vars+"*"+tern
                inst.toString(sb, vars0);
            } else {
                sb.append(term.toString(vars));
            }
        }
        return sb.toString();
    }


    @Override
    public boolean equals(Object obj) {
        if (obj instanceof PolynomialRing.INST) {
            INST inst = (INST)obj;
```

```
            return equalTerms(inst.terms);
        } else {
            return false;
        }
    }
    @Override
    public int hashCode() {
        int i = 0;
        for (Term term: terms) {
            i += term.hashCode();
        }
        return i;
    }
    private boolean equalTerms(List<Term> terms0) {
        final int size = terms.size();
        if (terms0.size() != size) {
            return false;
        }
        for (int i = 0; i < size; i++) {
            if (!terms.get(i).equals(terms0.get(i))) {
                return false;
            }
        }
        return true;
    }
}

public static void main(String[] args) {
    {
        //final PolynomialRing<NIntegerRing.INST, NIntegerRing.CLASS>.CLA
        final PolynomialRing<NIntegerRing.INST, NIntegerRing.CLASS>.CLASS
        final NIntegerRing.CLASS coeffCls = IntPolynomialRing.coefficientF
        final PolynomialRing<NIntegerRing.INST, NIntegerRing.CLASS>.INST
        final PolynomialRing<NIntegerRing.INST, NIntegerRing.CLASS>.INST
        final PolynomialRing<NIntegerRing.INST, NIntegerRing.CLASS>.INST
        System.out.println(">> nr3: "+nr3);
    }
}
}
```

```
package com.ncalathus.sample.ring.rep;

import com.ncalathus.sample.ring.api.IPolynomialRing;
import java.util.HashMap;

public class NumericPolynomialRing<N extends Number> extends PolynomialRing<N
    protected NumericPolynomialRing(final NumericRing<N>.CLASS coeffCls, fina
        super(coeffCls, variable);
    }
    public class CLASS extends PolynomialRing<NumericRing<N>.INST, NumericRing
        public final INST create(final N... coeffs) {
            return create(new HashMap<Integer, NumericRing<N>.INST>() {{
                int degree = coeffs.length-1;
                for (int i = 0; i <= degree; i++) {
                    final NumericRing<N>.INST c = _coeffCls.create(coeffs[i])
                    if (!c.equals(_coeffCls.zero())) {
                        put(degree-i, c);
                    }
                }
            }});
        }
    }
    @Override
    protected void initCLASS() {
        this._class = new CLASS();
    }
    @Override
    public CLASS getCLASS() {
        return (CLASS)_class;
    }

    public static <N extends Number> NumericPolynomialRing<N>.CLASS CLASS(fin
        return new NumericPolynomialRing(nCls, variable).getCLASS();
    }

    //
    //
    //
    static void test_int_1() {
        System.out.println("----------_test_int_1_----------");
        final NumericPolynomialRing<Integer>.CLASS NIntPolynomialRing = Numer
        final NumericRing<Integer>.CLASS coeffCls = NIntPolynomialRing.coeffic
        {
            final NumericPolynomialRing<Integer>.INST nr1 = NIntPolynomialRing
            final NumericPolynomialRing<Integer>.INST nr2 = NIntPolynomialRing
            final NumericPolynomialRing<Integer>.INST nr3 = nr1.mult(nr2);
```

```
            System.out.println(">> nr1: "+nr1);
            System.out.println(">> nr2: "+nr2);
            System.out.println(">> nr3: "+nr3);
        }
        // this looks strange, but it is OK, since subclass deosn ot define C
same inner class in byte code level.
        {
            final IntPolyRing.INST nr1 = NIntPolynomialRing.create(7, coeffCls
            final IntPolyRing.INST nr2 = NIntPolynomialRing.create(5, coeffCls
            final IntPolyRing.INST nr3 = nr1.mult(nr2);
            System.out.println(">> nr1: "+nr1);
            System.out.println(">> nr2: "+nr2);
            System.out.println(">>> nr3: "+nr3);
        }
    }

    static void test_int_2() {
        System.out.println("---------- test_int_2 ----------");
        final NumericPolynomialRing<Integer>.CLASS NIntPolynomialRing = Numer
        final NumericRing<Integer>.CLASS coeffCls = NIntPolynomialRing.coeffic
        {
            /*
            final NumericPolynomialRing<Integer> npr = new NumericPolynomialR
            final NumericPolynomialRing<Integer>.INST nr1 = npr.new INST() {{
                final Random rnd = new Random(10);
                for (int i = 0; i < 10; i++) {
                    //int degree =
                    //add();
                }
            }};
            */
            final NumericPolynomialRing<Integer>.INST nr1 = NIntPolynomialRing
            final NumericPolynomialRing<Integer>.INST nr2 = NIntPolynomialRing
            final NumericPolynomialRing<Integer>.INST nr3 = nr1.mult(nr2);
            System.out.println(">> nr1: "+nr1);
            System.out.println(">> nr2: "+nr2);
            System.out.println(">> nr3: "+nr3);
        }
    }
    // (compiler) error case
    /*
    static void test_int_1a() {
        final NumericPolynomialRing<Integer>.CLASS NIntPolynomialRing = Nume
        final NumericRing<Float>.CLASS coeffCls = NIntPolynomialRing.coefficie
        {
            final NumericPolynomialRing<Integer>.INST nr1 = NIntPolynomialRin
```

```
        final NumericPolynomialRing<Integer>.INST nr2 = NIntPolynomialRin
        final NumericPolynomialRing<Integer>.INST nr3 = nr1.mult(nr2);
        System.out.println(">> nr3: "+nr3);
    }
    // strange..
    {
        final IntPolyRing.INST nr1 = NIntPolynomialRing.create(7, coeffCl
        final IntPolyRing.INST nr2 = NIntPolynomialRing.create(5, coeffCl
        final IntPolyRing.INST nr3 = nr1.mult(nr2);
        System.out.println(">>> nr3: "+nr3);
    }
}
*/

// this looks type checking is not working as expected..
static void test_int_1b() {
    System.out.println("----------test_int_1b----------");
    final NumericPolynomialRing<Integer>.CLASS NIntPolynomialRing = Numer
    final NIntegerRing.CLASS coeffCls = NIntPolynomialRing.coefficientRing
    {
        final NumericPolynomialRing<Integer>.INST nr1 = NIntPolynomialRing
        final NumericPolynomialRing<Integer>.INST nr2 = NIntPolynomialRing
        final NumericPolynomialRing<Integer>.INST nr3 = nr1.mult(nr2);
        System.out.println(">> nr1: "+nr1);
        System.out.println(">> nr2: "+nr2);
        System.out.println(">> nr3: "+nr3);
    }
    // strange..
    {
        final IntPolyRing.INST nr1 = NIntPolynomialRing.create(7, coeffCls
        final IntPolyRing.INST nr2 = NIntPolynomialRing.create(5, coeffCls
        final IntPolyRing.INST nr3 = nr1.mult(nr2);
        System.out.println(">> nr1: "+nr1);
        System.out.println(">> nr2: "+nr2);
        System.out.println(">>> nr3: "+nr3);
    }
}

static void test_float_1() {
    System.out.println("----------test_float_1----------");
    final NumericPolynomialRing<Float>.CLASS NFloatPolynomialRing = Numer
    final NumericRing<Float>.CLASS coeffCls = NFloatPolynomialRing.coeffic
    {
        final NumericPolynomialRing<Float>.INST nr1 = NFloatPolynomialRing
        final NumericPolynomialRing<Float>.INST nr2 = NFloatPolynomialRing
        final NumericPolynomialRing<Float>.INST nr3 = nr1.mult(nr2);
```

```
        System.out.println(">> nr1: "+nr1);
        System.out.println(">> nr2: "+nr2);
        System.out.println(">> nr3: "+nr3);
    }
}

static void test_float_2() {
    System.out.println("---------- test_float_2 ----------");
    final NumericRing<Float>.CLASS FloatNumeral = NumericRing.CLASS(Float
    final NumericPolynomialRing<Float>.CLASS NFloatPolynomialRing = Numer
    {
        final NumericPolynomialRing<Float>.INST nr0 = NFloatPolynomialRing
        final NumericPolynomialRing<Float>.INST nr1 = NFloatPolynomialRing
        final NumericPolynomialRing<Float>.INST nr2 = NFloatPolynomialRing
        final NumericPolynomialRing<Float>.INST nr3 = NFloatPolynomialRing
        final NumericPolynomialRing<Float>.INST nr4 = NFloatPolynomialRing
        final NumericPolynomialRing<Float>.INST nr5 = NFloatPolynomialRing
        final NumericPolynomialRing<Float>.INST nr6 = nr1.mult(nr2);
        System.out.println(">> nr1: "+nr1);
        System.out.println(">> nr2: "+nr2);
        System.out.println(">> nr3: "+nr3);
        System.out.println(">> nr4: "+nr4);
        System.out.println(">> nr5: "+nr5);
        System.out.println(">> nr6: "+nr6);

        final PolynomialRing<NumericPolynomialRing<Float>.INST, NumericPo

        //final PolynomialRing<NumericPolynomialRing<Float>.INST, Numeric
        //final PolynomialRing<NumericPolynomialRing<Float>.INST, Numeric
        //final PolynomialRing<NumericPolynomialRing<Float>.INST, Numeric

        final PolynomialRing<NumericPolynomialRing<Float>.INST, NumericPo
        final PolynomialRing<NumericPolynomialRing<Float>.INST, NumericPo
        final PolynomialRing<NumericPolynomialRing<Float>.INST, NumericPo
        System.out.println(">> nr1a: "+nr1a);
        System.out.println(">> nr2a: "+nr2a);
        System.out.println(">> nr3a: "+nr3a);
    }

}

public static void main(String[] args) {
    /*
    test_int_1();
    test_int_1b();
    test_float_1();
```

```
        */
        //test_int_2();
        test_float_2();     }
}
```