

*Open Source Solutions to Java Development Problems*



# Jakarta Commons Cookbook™

O'REILLY®

*Timothy M. O'Brien*

## 8.0 Introduction

In recent years, Java has lost its reputation as a language suffering from serious performance problems. Although the debate still rages on and various benchmarks show conflicting results, improvements to the JVM, the compiler, and a more intelligent garbage collector have boosted performance to levels on par with C++. Java has never been the traditional language-of-choice for scientific computing, numerical methods, or high-performance computing, but, as performance improves, there are fewer reasons to avoid using Java for numerical computing. It is no longer inconceivable to consider Java when implementing systems involving complex mathematics, and Jakarta Commons contains two projects that provide some very basic math capabilities: Commons Lang and Commons Math.

The first four recipes in this chapter deal with math utilities found in Commons Lang. This includes a class that represents fractions, finding the minimum and maximum values in an array, representing a range of numbers, and convenient ways to retrieve different random variables. Commons Lang was introduced in Chapter 1, and instructions for downloading and installing Commons Lang can be found in Recipe 1.1. The remainder of this chapter deals with Jakarta Commons Math. Recipes involving Commons Math deal with complex numbers, the calculation of univariate statistics, solving a system of linear equations, and establishing a relationship between two independent variables. Instructions for downloading and installing Commons Math can be found in Recipe 8.5.

## 8.1 Using Fractions

### Problem

You need to work with fractions supplied by the user, such as  $3\frac{4}{5}$  and  $134/21$ . Your application needs to parse, multiply, and reduce fractions.

## Solution

Use Jakarta Commons Lang's Fraction class to parse and manipulate fractions. The following code demonstrates the parsing of a String containing a fraction:

```
import org.apache.commons.lang.math.Fraction;

String userInput = "23 31/37";
Fraction fraction = Fraction.getFraction( userInput );
double value = fraction.doubleValue();
```

The String "23 31/37" is converted to a double value of 23.837837. A Fraction object is created by calling the Fraction.getFraction() method, and double value of the Fraction object is obtained with fraction.doubleValue().

## Discussion

The Fraction class provides a number of operations that can be used to simplify the following expression to an improper fraction. The following code evaluates the expression in Figure 8-1 using Fraction:

```
import org.apache.commons.lang.math.Fraction;

Fraction numer1 = Fraction.getFraction( 3, 4 );
Fraction numer2 = Fraction.getFraction( 51, 3509 );

Fraction numerator = numer1.multiplyBy( numer2 );
Fraction denominator = Fraction.getFraction( 41, 59 );

Fraction fraction = numerator.divideBy( denominator );
Fraction result = fraction.reduce();

System.out.println( "as Fraction: " + result.reduce().toString() );
System.out.println( "as double: " + result.doubleValue() );
```

$$\left( \frac{\frac{3}{4} \times \frac{51}{3509}}{\frac{41}{59}} \right)$$

Figure 8-1. Expression to be evaluated with Fraction

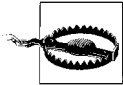
The previous example creates an instance of Fraction by calling the static getFraction(int numerator, int denominator) method. Fraction objects are then multiplied and divided with the multiplyBy() and divideBy() methods of Fraction. And, the final call to reduce() reduces the Fraction to the smallest possible denominator. This example executes and prints the following output to the console:

```
Expression as Fraction: 9027/575476
Expression as double: 0.015686145034719084
```

An improper fraction is a fraction such that  $X/Y > 1$  (i.e., “135/23” or “3/2”). Fraction provides the ability to convert improper fractions to proper fractions as demonstrated in the following example:

```
import org.apache.commons.lang.math.Fraction;

String userInput = "101/99";
String properString = Fraction.getFraction(userInput).toProperString();
// properString is now "1 2/99"
```



Fraction does not automatically reduce contents, and it is important to call `reduce()` before performing any arithmetic with the `Fraction` class to reduce the risk of overflow. For example, `Fraction.getFraction( 10000, 100000 ).pow( 6 )` should equal `1.0E-6`, but, because `Fraction` simply multiplies each numerator and denominator without reducing the fraction, the result of this statement will be `1.0`. When raised to the power of 6, the `Fraction` object quickly becomes `Fraction.getFraction(Integer.MAX_VALUE, Integer.MAX_VALUE)` or `1.0`. Call `reduce()` liberally or you may have occasion to curse this `Fraction` class.

Table 8-1 lists a sampling of methods available on the `Fraction` class.

Table 8-1. *Methods on Commons Lang Fraction*

Method	Description
<code>abs()</code>	Returns the absolute value of a <code>Fraction</code>
<code>add(Fraction fraction)</code>	Adds two <code>Fraction</code> objects together
<code>subtract(Fraction fraction)</code>	Subtracts the parameter from the current <code>Fraction</code>
<code>multiplyBy(Fraction fraction)</code>	Multiplies the parameter by the current <code>Fraction</code>
<code>divideBy(Fraction fraction)</code>	Divides the current <code>Fraction</code> by the parameter
<code>reduce()</code>	Reduces the <code>Fraction</code> to the smallest denominator
<code>negate()</code>	Returns $-1 * \text{Fraction}$
<code>invert()</code>	Swaps the numerator and denominator
<code>getNumerator()</code>	Returns the numerator
<code>getDenominator()</code>	Returns the denominator
<code>getProperNumerator()</code>	Returns the proper numerator
<code>getProperWhole()</code>	Returns the proper whole number
<code>pow()</code>	Raises a <code>Fraction</code> to the specified power

## See Also

For more information about downloading Commons Lang, see Recipe 1.1.

## 8.2 Finding the Maximum and Minimum in an Array

### Problem

You need to retrieve the maximum and minimum values from a `double[]`, `float[]`, `long[]`, `int[]`, `short[]`, or `byte[]`.

### Solution

Use Commons Lang `NumberUtils.max()` and `NumberUtils.min()` to retrieve the minimum or maximum values from an array of primitives. The following code retrieves the minimum and maximum values from a `double[]`:

```
import org.apache.commons.lang.math.NumberUtils;

double[] array = {0.2, 0.4, 0.5, -3.0, 4.223, 4.226};

double max = NumberUtils.max( array ); // returns 4.226
double min = NumberUtils.min( array ); // returns -3.0
```

### Discussion

`NumberUtils.min()` and `NumberUtils.max()` both accept `double[]`, `float[]`, `long[]`, `int[]`, `short[]`, and `byte[]`. If the array is empty or null, both `NumberUtils.min()` and `NumberUtils.max()` will return an `IllegalArgumentException`.

Jakarta Commons Math also contains a class that can find the minimum and maximum value in a `double[]`. The following example uses the `Max` and `Min` classes from Commons Math to evaluate a `double[]`:

```
import org.apache.commons.math.stat.univariate.rank.Max;
import org.apache.commons.math.stat.univariate.rank.Min;

double[] array = {0.2, 0.4, 0.5, -3.0, 4.223, 4.226};

Max maximum = new Max();
Min minimum = new Min();

double max = maximum.evaluate( array, 0, array.length );
double min = minimum.evaluate( array, 0, array.length );
```

## 8.3 Using Number Ranges

### Problem

You need to define a range of acceptable values for a variable, and test to see if that variable is within those boundaries.

## Solution

Use an implementation of `Range`, an interface that defines a simple numerical range. There are a number of different implementations for different types: `NumberRange`, `DoubleRange`, `FloatRange`, `IntRange`, and `LongRange`. The following example demonstrates the use of `DoubleRange` to verify that a variable is within a valid range. A `DoubleRange` is created with minimum and maximum values, and a value is tested by `DoubleRange` using a method named `containsDouble()`:

```
import org.apache.commons.lang.math.DoubleRange;
import org.apache.commons.lang.math.Range;

Range safeSpeed = new DoubleRange( 0.0, 65.0 );

double currentSpeed = getCurrentSpeed();
if( !safeSpeed.containsDouble( currentSpeed ) ) {
    System.out.println( "Warning, current speed is unsafe." );
}
```

## Discussion

Additionally, one can also test to see if another `Range` is contained within a `Range`, or if a `Range` overlaps another `Range`. The following example demonstrates the use of `containsRange()` to determine if a `Range` is entirely contained within another `Range`:

```
import org.apache.commons.lang.math.Range;
import org.apache.commons.lang.math.IntRange;
import org.apache.commons.lang.math.NumberUtils;

double recordHigh = getRecordHigh();
double recordLow = getRecordLow();

IntRange recordRange = new IntRange( recordLow, recordHigh );

int todayTemp = getTodaysMaxTemp();
IntRange daysRange = new IntRange( NumberUtils.min( todayTemp ),
                                   NumberUtils.max( todayTemp ) );

if( !recordRange.containsRange( todayTemp ) ) {
    System.out.println( "Today is a record temperature day!" );
}
```

The previous code creates a `Range`, `recordRange`, from the record high and low temperatures. It then creates `daysRange`, which is a `Range` of the current day's high and low temperatures. If `daysRange` is not entirely contained within the `recordRange`, then the current day contains a record temperature and `recordRange.containsRange(daysRange)` will return `false`. `containsRange()` returns `true` if every value in the containing range occurs in the contained range, and `overlapsRange()` returns `true` if two `Range` objects

share any common value. `NumberUtils` is used to retrieve the maximum and minimum values from the `todayTemp` array.

In another example, a `Range` object is used to ascertain the state of an element from a temperature measurement. Elemental Gold (Au) melts at 1337.33 Kelvin and boils at 3129.15 Kelvin. The following code is used to read the temperature from a thermometer and print the current state of the element:

```
import org.apache.commons.lang.math.Range;
import org.apache.commons.lang.math.DoubleRange;

double melting = 1337.33;
double boiling = 3129.15

// State ranges for element Au
Object[] stateRanges =
    new Object[][]{{"solid" , new DoubleRange( 0.0, melting )},
                  {"liquid", new DoubleRange( melting, boiling )},
                  {"gas", new DoubleRange( boiling,Double.INFINITY )};

// Read measurement from thermometer
double temp = thermometer.getReading();
String state = "unknown";

// Test the state
for( int i = 0; i < stateRanges.length; i++ ) {
    DoubleRange stateRange = (DoubleRange) stateRanges[i][1];
    if( stateRange.contains( temp ) ) {
        state = (String) stateRanges[i][0];
    }
}

System.out.println( "The substance is in a " + state + " state." );

// If the temperature is a temperate 293 K, this line would print
// "The Gold is in a solid state."
```

The ranges in this example overlap; the solid range ends at melting, and the liquid range begins at melting. Because each `Range` is tested in a defined order, each `Range` object in this example is lower- and upper-bound inclusive. If the `temp` variable has the same value as melting, the program will indicate solid state, and if the `temp` variable has the same value as boiling, this program will signify the liquid state.

## See Also

For more information about downloading Commons Lang, see [Recipe 1.1](#).

## 8.4 Generating Random Variables

### Problem

J2SE 1.4 includes a `java.lang.Math` class that provides a mechanism to get a random double value between 0.0 and 1.0, but you need to create random boolean values, or random int variables between zero and a specified number.

### Solution

Generate random variables with Commons Lang `RandomUtils`, which provides a mechanism to generate random int, long, float, double, and boolean variables. The following code generates a random integer between zero and the value specified in the parameter to `nextInt()`:

```
import org.apache.commons.lang.math.RandomUtils;

// Create a random integer between 0 and 30
int maxVal = 30;
int randomInt = RandomUtils.nextInt( maxVal );
```

Or, if your application needs a random boolean variable, create one with a call to the static method `nextBoolean()`:

```
import org.apache.commons.lang.math.RandomUtils;

boolean randomBool = RandomUtils.nextBoolean();
```

### Discussion

A frequent argument for not using a utility like `RandomUtils` is that the same task can be achieved with only one line of code. For example, if you need to retrieve a random integer between 0 and 32, you could write the following code:

```
int randomInt = (int) Math.floor( (Math.random() * (double) maxVal) );
```

While this statement may seem straightforward, it does contain a conceptual complexity not present in `RandomUtils.nextInt(maxVal)`. `RandomUtils.nextInt(maxVal)` is a simple statement: “I need a random integer between 0 and `maxVal`”; the statement without `RandomUtils` is translated to a more complex statement:

I’m going to take a random double between 0.0 and 1.0, and multiply this number by `maxVal`, which has been cast to a `double`. This result should be a random double between 0.0 and `maxVal`, which I will then pass to `Math.floor()` and cast to an `int`.

While the previous statement does achieve the same task as `RandomUtils`, it does so by rolling-up multiple statements into a single line of code: two casts, a call to `floor()`, a call to `random()`, and a multiplication. You may be able to instantly recognize this pattern as code that retrieves a random integer, but someone else may have a completely different approach. When you start to use some of the smaller utilities from

Jakarta Commons systemwide, an application will tend toward greater readability; these small reductions in conceptual complexity quickly add up.

## 8.5 Obtaining Commons Math

### Problem

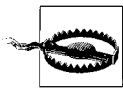
You need to use Jakarta Commons Math to work with complex numbers, matrices, statistics, or linear equations.

### Solution

Download Jakarta Commons Math and put the necessary JAR files in your classpath. Because Commons Math has not yet been released, you will need to download the latest nightly snapshot of Commons Math from <http://cvs.apache.org/builds/jakarta-commons/nightly/commons-math/>. Once you have downloaded the latest nightly build, uncompress the distribution, and place the *commons-math-1.0-RC1.jar* file in your classpath.

### Discussion

Jakarta Commons Math was created to provide some more advanced mathematical capabilities under an Apache-style license. Commons Math provides classes to work with complex numbers, utilities to calculate statistics, a matrix implementation, special functions, continued fractions, root-finding, interpolation, and bivariate regression. Commons Math depends on Commons Collections 3.0, Commons Lang 2.0, and Commons Logging 1.0.3. To obtain these dependencies, see Recipes 1.1, 5.1, and 7.9.



By the time this book is published, Commons Math 1.0 may be released. Make sure you look at the Commons Math project page (<http://jakarta.apache.org/commons/math>) and check for a release before downloading.

While the previous four recipes demonstrated classes and utilities available in Jakarta Commons Lang, the next five recipes demonstrate classes and utilities from Jakarta Commons Math. Simpler math utilities, which have wide application, will frequently be included in Commons Lang, and more complex utilities will be added to Commons Math. As both components continue to evolve, you may notice some overlap between the feature-set of Commons Lang and Commons Math.

### See Also

Commons Math is a relatively new math library primarily motivated by licensing and community issues; all established math libraries were covered under a GPL or LGPL

license. The Jakarta Commons community saw an opportunity to create a library, and through the hard work of a few dedicated developers in Commons, a very useful math library will soon be released. That being said, if you need a more mature math library, and your project can use LGPL components, look at the Colt Distribution from CERN (<http://hoschek.home.cern.ch/hoschek/colt/>).

For the authoritative cookbook of mathematics, pick up a copy of *Numerical Recipes in C++* or *Numerical Recipes in C* (Cambridge University Press). These classic tomes contain a huge library of code and examples, but be forewarned, the mathematics will quickly intimidate the faint of math. More information about this indispensable text can be found at the Numerical Recipes website (<http://www.nr.com/>). Unlike all the components described throughout this book, the code and examples from both of these books is covered under a very restrictive license described at <http://www.numerical-recipes.com/infotop.html#distinfo>.

## 8.6 Calculating Simple Univariate Statistics

### Problem

You need to calculate univariate statistics such as mean, median, variance, minimum and maximum.

### Solution

Use Commons Math's `StatUtils` to calculate simple univariate statistics. The following example uses `StatUtils` to calculate simple statistics for a `double[]`:

```
import org.apache.commons.math.stat.StatUtils;

double[] values = new double[] { 2.3, 5.4, 6.2, 7.3, 23.3 };

System.out.println( "min: " + StatUtils.min( values ) );
System.out.println( "max: " + StatUtils.max( values ) );
System.out.println( "mean: " + StatUtils.mean( values ) );
System.out.println( "product: " + StatUtils.product( values ) );
System.out.println( "sum: " + StatUtils.sum( values ) );
System.out.println( "variance: " + StatUtils.variance( values ) );
```

This code executes and prints a few simple statistics to the console, as follows:

```
min: 2.3
max: 23.3
mean: 8.9
product: 13097.61036
sum: 44.5
variance: 68.25500000000001
```

## Discussion

StatUtils delegates these calculations to functors in the `org.apache.commons.math.stat.univariate.moment`, `org.apache.commons.math.stat.univariate.rank`, and `org.apache.commons.math.stat.univariate.summary` packages. The following example uses the individual classes from these packages to recreate the previous example, and it adds some measures not available in StatUtil:

```
import org.apache.commons.math.stat.univariate.moment.*;
import org.apache.commons.math.stat.univariate.rank.*;
import org.apache.commons.math.stat.univariate.summary.*;

// Measures from previous example
Min min = new Min();
Max max = new Max();
Mean mean = new Mean();
Product product = new Product();
Sum sum = new Sum();
Variance variance = new Variance();

System.out.println( "min: " + min.evaluate( values ) );
System.out.println( "max: " + max.evaluate( values ) );
System.out.println( "mean: " + mean.evaluate( values ) );
System.out.println( "product: " + product.evaluate( values ) );
System.out.println( "sum: " + sum.evaluate( values ) );
System.out.println( "variance: " + variance.evaluate( values ) );

// New measures
Percentile percentile = new Percentile();
GeometricMean geoMean = new GeometricMean();
Skewness skewness = new Skewness();
Kurtosis kurtosis = new Kurtosis();

System.out.println( "80 percentile value: " +
    percentile.evaluate( values, 80.0 ) );
System.out.println( "geometric mean: " + geoMean.evaluate( values ) );
System.out.println( "skewness: " + skewness.evaluate( values ) );
System.out.println( "kurtosis: " + kurtosis.evaluate( values ) );
```

The previous example adds percentile, geometric mean, standard deviation, skewness, and kurtosis to the available univariate statistics. The previous example produces the following output:

```
min: 2.3
max: 23.3
mean: 8.9
product: 13097.61036
sum: 44.5
variance: 68.25500000000001
80 percentile value: 20.099999999999998
geometric mean: 6.659450778469037
standard dev: 8.261658429153314
skewness: 1.9446683453691376
kurtosis: 4.102348153299074
```

## See Also

If you need a formal definition of a specific moment, rank, or summary, see MathWorld (<http://mathworld.wolfram.com>), an invaluable mathematical reference site from Wolfram, the makers of Mathematica.

## 8.7 Solving a System of Linear Equations

### Problem

You need to find the values of  $x$ ,  $y$ , and  $z$  that satisfy the system of linear equations shown in Figure 8-2.

$$\begin{aligned}3x + 20y + 89z &= 1324 \\4x + 40y + 298z &= 2999 \\7x - 21y + 0.42z &= 2039\end{aligned}$$

Figure 8-2. A system of linear equations

### Solution

Use the `RealMatrix` and `RealMatrixImpl` from Commons Math. Represent this system of linear equations as matrices in the  $Ax=B$  form, as shown in Figure 8-3. Place the coefficients of  $A$  in a `RealMatrix`, and put  $B$  in a `double[]`. Call the `solve()` method on `RealMatrix` to retrieve a `double[]` of values for  $x$ ,  $y$ , and  $z$  that satisfy this system of equations.

$$\begin{bmatrix} 3 & 20 & 89 \\ 4 & 40 & 298 \\ 7 & 21 & 0.42 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1324 \\ 2999 \\ 2039 \end{bmatrix}$$

Figure 8-3. System of linear equations in  $Ax=B$  form

The following example takes the coefficients and constants from Figure 8-3 and uses a `RealMatrix` to solve this system:

```
import org.apache.commons.math.linear.RealMatrix;
import org.apache.commons.math.linear.RealMatrixImpl;
import org.apache.commons.lang.ArrayUtils;

double[][] coefficients = { { 3.0, 20.0, 89.0 },
                             { 4.0, 40.0, 298.0 },
                             { 7.0, 21.0, 0.42 } };
double[] values = { 1324, 2999, 2039 };
```

```

RealMatrix matrix = new RealMatrixImpl();
matrix.setData( coefficients );

double[] answers = matrix.solve( values );

System.out.println( "Answers: " + ArrayUtils.toString( answers ) );

```

This example solves this system of equations and prints out the values of  $x$ ,  $y$ , and  $z$  using Commons Lang `ArrayUtils` to print a `double[]`:

```
Answers: {400.4839095455532, -36.59139305646149, 9.599731825759218}
```

Using Commons Math, we find that the following values satisfy this system of equations:  $x = 400$ ,  $y = -36$ , and  $z = 9.6$ .

## Discussion

To solve these equations, a `double[][]` of coefficients is created to represent a  $3 \times 3$  matrix, and a `double[]` of constants is created. The `RealMatrix` interface is implemented by `RealMatrixImpl`, which stores a matrix as a `double[][]`; to populate this `double[][]`, pass the `double[][]`, `coefficients`, to the `setData()` method of `RealMatrixImpl`. To solve the system, the values `double[]` is passed to `matrix.solve()`, and a `double[]` containing  $x$ ,  $y$ , and  $z$  is returned.

This method will not work for every matrix; there are systems of linear equations that are unsolvable. For example, if one attempts to find values for the system of equations from Figure 8-4, an `InvalidMatrixException` will be thrown stating that the matrix is singular. Additionally, if the number of rows in  $B$  does not equal the number of columns in  $A$ , `solve()` will throw an `InvalidMatrixException`.

$x + y + z = 1$
$x + y - z = 35$
$x - y + z = 23$

Figure 8-4. An unsolvable system of equations

## See Also

For more information about solving systems of linear equations (or, for that matter, information about anything), see Wikipedia ([http://en.wikipedia.org/wiki/System\\_of\\_linear\\_equations](http://en.wikipedia.org/wiki/System_of_linear_equations)). `RealMatrixImpl` uses a process known as LU decomposition to solve this system of equations. For more information about LU decomposition, see the JavaDoc for the `org.apache.commons.math.linear` package (<http://jakarta.apache.org/commons/math/apidocs/index.html>).

## 8.8 Arithmetic with Complex Numbers

### Problem

You need to perform arithmetic with complex numbers. For example, given the complex numbers  $A$ ,  $B$ ,  $C$ , and  $E$  and two equations shown in Figure 8-5, you need to find the real part of  $F$  and the imaginary part of  $D$ .

$$(A + B) / C = D$$
$$D * x E = F$$
$$\text{Answer} = \text{Re}(F) / \text{Im}(D)$$

Figure 8-5. Expressions evaluated with the Complex object

### Solution

Use Commons Math `Complex` and `ComplexMath` classes to represent complex numbers and perform arithmetic using complex numbers. Use the `ComplexFormat` class to print the real and imaginary parts of a complex number. The following example demonstrates the use of the `Complex` class to calculate  $D$  and  $F$  from Figure 8-5 using arbitrary values for  $A$ ,  $B$ ,  $C$ , and  $E$ :

```
import org.apache.commons.math.complex.Complex;
import org.apache.commons.math.complex.ComplexFormat;

Complex a = new Complex(2, 3);
Complex b = new Complex(4, 5);
Complex c = new Complex(0.3, 2);
Complex e = new Complex(4, 4);

Complex sum = a.add( b );
Complex d = c.divide( sum );
Complex f = e.multiply( d.conjugate() );

System.out.println( "D is: " + ComplexFormat.formatComplex( d ) );
System.out.println( "F is: " + ComplexFormat.formatComplex( f ) );

double realF = f.getReal();
double imD = d.getImaginary();
double answer = realF / imD;

System.out.println( "Answer Re(F)/Im(D): " +
    NumberFormat.getInstance().format( answer ) );
```

The variables  $a$ ,  $b$ ,  $c$ , and  $e$  are created using arbitrary values, and an intermediate `Complex` object `sum` is calculated by adding  $b$  to  $a$ —`a.add(b)`.  $d$  is calculated by dividing this intermediate sum by  $c$ : `c.divide(sum)`.  $f$  is calculated by multiplying  $e$  times the complex conjugate of  $d$ : `e.multiply( d.conjugate() )`. The final answer is calculated by taking the real part of  $f$  (`f.getReal()`) and dividing that by the imaginary

part of `d`: `d.getImaginary()`. The previous example performs complex arithmetic and prints the following to the console:

```
D is: 0.18 + 0.1i
F is: 1.1 + 0.33i
Answer Re(F)/Im(D): 11.417
```

## Discussion

The previous example used the `ComplexFormat` class to create a `String` representation of a `Complex` object. This class allows you to print out the complex number  $N$  in the format  $Re(N) + Im(N)i$ . This class also has a constructor that takes a `String` to use instead of “i.” In electrical engineering, where “i” is frequently used to refer to current, complex impedance is represented using a “j” instead of an “i.” To print a complex number using a “j”, write the following code:

```
Complex impedance = new Complex( 1.0, 2.0 );
ComplexFormat format = new ComplexFormat("j");
System.out.println( "Impedance: " + format.format( impedance ) );
```

The previous code prints the following output to the console:

```
Impedance: 1.0 + 2.0j
```

The `Complex` object contains simple arithmetic methods such as `add()`, `subtract()`, `multiply()`, `divide()`, `conjugate()`, and `negate()`. More advanced methods are available as static methods on the `ComplexMath` class. `ComplexMath` includes trigonometric methods such as `sin()`, `sinh()`, `cos()`, and `tan()`, as well as methods to calculate logarithms and to take the square root of a `Complex` object.

## See also

For more information about the `ComplexMath` utility, see the Commons Math JavaDoc at <http://jakarta.apache.org/commons/math/apidocs/index.html>.

# 8.9 Establishing Relationships Between Variables

## Problem

You need to establish a relationship between two independent variables. These variables could be temperature versus energy use or the number of news channels versus stress-related ailments; you need to measure the correlation between two variables.

## Solution

Add data points to an instance of Commons Math `SimpleRegression`. This class will calculate the slope, slope confidence, and a measure of relatedness known as R-square. The `SimpleRegression` class performs a least squares regression with one

independent variable; adding data points to this model refines parameters to the equation  $y = ax + b$ . The following code uses `SimpleRegression` to find a relationship between two series of values  $[0, 1, 2, 3, 4, 5]$  and  $[0, 1.2, 2.6, 3.2, 4, 5]$ :

```
import org.apache.commons.math.stat.multivariate.SimpleRegression;

SimpleRegression sr = new SimpleRegression();

// Add data points
sr.addData( 0, 0 );
sr.addData( 1, 1.2 );
sr.addData( 2, 2.6 );
sr.addData( 3, 3.2 );
sr.addData( 4, 4 );
sr.addData( 5, 5 );

// Print the value of y when line intersects the y axis
System.out.println( "Intercept: " + sr.getIntercept() );

// Print the number of data points
System.out.println( "N: " + sr.getN() );

// Print the Slope and the Slop Confidence
System.out.println( "Slope: " + sr.getSlope() );
System.out.println( "Slope Confidence: " + sr.getSlopeConfidenceInterval() );

// Print RSquare a measure of relatedness
System.out.println( "RSquare: " + sr.getRSquare() );
```

This example passes six data points to `SimpleRegression` and prints the slope, number of data points, and R-square from `SimpleRegression`:

```
Intercept: 0.238
N: 6
Slope: 0.971
Slope Confidence: 0.169
RSquare: 0.985
```

## Discussion

R-square is the square of something called the Pearson's product moment correlation coefficient, which can be obtained by calling `getR()` on `SimpleRegression`. R-square is a determination of correlation between two series of numbers. The parameters to the `addData()` method of `SimpleRegression` are a corresponding x and y value in two sets of data. If R-square is 1.0, the model shows that as x increases linearly, y increases linearly. In the previous example, R-square is 0.98, and this demonstrates that the (x,y) data points added to `SimpleRegression` have a strong linear relationship.

If R-square is  $-1.0$ ,  $x$  increases linearly as  $y$  decreases linearly. A value of  $0.0$  shows that the relationship between  $x$  and  $y$  is not linear. The following example demonstrates two series of numbers with no relationship:

```
import org.apache.commons.math.stat.multivariate.SimpleRegression;

SimpleRegression sr = new SimpleRegression();
sr.addData( 400, 100 );
sr.addData( 300, 105 );
sr.addData( 350, 70 );
sr.addData( 200, 50 );
sr.addData( 150, 300 );
sr.addData( 50, 500 );

// Print RSquare a measure of relatedness
System.out.println( "RSquare: " + sr.getRSquare() );
```

The data points added to this `SimpleRegression` are all over the map;  $x$  and  $y$  are unrelated, and the R-square value for this set of data points is very close to zero:

```
Intercept: 77.736
N: 12
Slope: 0.142
Slope Confidence: 0.699
RSquare: 0.02
```

The  $(x,y)$  data points supplied to the previous example have no linear correlation. This doesn't prove that there is no relationship between  $x$  and  $y$ , but it does prove that the relationship is not linear.

## See Also

For more information about least squares, the technique used by `SimpleRegression`, see Wikipedia ([http://en.wikipedia.org/wiki/Least\\_squares](http://en.wikipedia.org/wiki/Least_squares)). More information about R and R-square can also be found on Wikipedia ([http://en.wikipedia.org/wiki/Pearson\\_product-moment\\_correlation\\_coefficient](http://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient)).

## 8.10 Estimating the Amount of Time Left in a Process

### Problem

You are running a program that takes a long time to execute, and you need to present the user with an estimated time until completion.

### Solution

Use Commons Math's `SimpleRegression` and Commons Lang's `StopWatch` to create a `ProcessEstimator` class that can be used to predict when a particular program will be

finished. Your program needs to process a number of records, and this program could take a few hours to finish. You would like to provide some feedback, and, if you are confident that each record will take roughly the same amount of time, you can use `SimpleRegression`'s slope and intercept to estimate the time when all records will be processed. Example 8-1 defines the `ProcessEstimator` class that combines the power of `StopWatch` and `ProcessEstimator` to estimate the time remaining in a process.

*Example 8-1. ProcessEstimator to estimate time of program execution*

```
package com.discursive.jccook.math.timeestimate;

import org.apache.commons.lang.time.StopWatch;
import org.apache.commons.math.stat.multivariate.SimpleRegression;

public class ProcessEstimator {

    private SimpleRegression regression = new SimpleRegression();
    private StopWatch stopWatch = new StopWatch();

    // Total number of units
    private int units = 0;

    // Number of units completed
    private int completed = 0;

    // Sample rate for regression
    private int sampleRate = 1;

    public ProcessEstimator( int numUnits, int sampleRate ) {
        this.units = numUnits;
        this.sampleRate = sampleRate;
    }

    public void start() {
        stopWatch.start();
    }

    public void stop() {
        stopWatch.stop();
    }

    public void unitCompleted() {
        completed++;

        if( completed % sampleRate == 0 ) {
            long now = System.currentTimeMillis();
            regression.addData( units - completed, stopWatch.getTime());
        }
    }
}
```

*Example 8-1. ProcessEstimator to estimate time of program execution (continued)*

```
public long projectedFinish() {
    return (long) regression.getIntercept();
}

public long getTimeSpent() {
    return stopWatch.getTime();
}

public long projectedTimeRemaining() {
    long timeRemaining = projectedFinish() - getTimeSpent();
    return timeRemaining;
}

public int getUnits() {
    return units;
}

public int getCompleted() {
    return completed;
}
}
```

ProcessEstimator has a constructor that takes the number of records to process and the sample rate to measure progress. With 10,000 records to process and a sample of 100, the SimpleRegression will add a data point of units remaining versus time elapsed after every 100 records. As the program continues to execute, projectedTimeRemaining() will return an updated estimation of time remaining by retrieving the y-intercept from SimpleRegression and subtracting the time already spent in execution. The y-intercept from SimpleRegression represents the y value when x equals zero, where x is the number of records remaining; as x decreases, y increases, and y represents the total time elapsed to process all records.

The ProcessEstimationExample in Example 8-2 uses the ProcessEstimator to estimate the time remaining while calling the performLengthyProcess() method 10,000 times.

*Example 8-2. An example using the ProcessEstimator*

```
package com.discursive.jccook.math.timeestimate;

import org.apache.commons.lang.math.RandomUtils;

public class ProcessEstimationExample {

    private ProcessEstimator estimate;

    public static void main(String[] args) {
        ProcessEstimationExample example = new ProcessEstimationExample();
        example.begin();
    }
}
```

Example 8-2. An example using the *ProcessEstimator* (continued)

```
public void begin() {
    estimate = new ProcessEstimator( 10000, 100 );
    estimate.start();

    for( int i = 0; i < 10000; i++ ) {
        // Print status every 1000 items
        printStatus(i);
        performLengthyProcess();
        estimate.unitCompleted();
    }

    estimate.stop();

    System.out.println( "Completed " + estimate.getUnits() + " in " +
        Math.round( estimate.getTimeSpent() / 1000 ) + " seconds." );
}

private void printStatus(int i) {
    if( i % 1000 == 0 ) {
        System.out.println( "Completed: " + estimate.getCompleted() +
            " of " + estimate.getUnits() );

        System.out.println( "\tTime Spent: " +
            Math.round( estimate.getTimeSpent() / 1000 ) +
            " sec" + ", Time Remaining: " +
            Math.round( estimate.projectedTimeRemaining() / 1000 ) +
            " sec" );
    }
}

private void performLengthyProcess() {
    try {
        Thread.sleep(RandomUtils.nextInt(10));
    } catch( Exception e ) {}
}
}
```

After each call to `performLengthyProcess()`, the `unitCompleted()` method on `ProcessEstimator` is invoked. Every 100th call to `unitComplete()` causes `ProcessEstimator` to update `SimpleRegression` with the number of records remaining and the amount of time spent so far. After every 1000th call to `performLengthyProcess()`, a status message is printed to the console as follows:

```
Completed: 0 of 10000
    Time Spent: 0 sec, Time Remaining: 0 sec
Completed: 1000 of 10000
    Time Spent: 4 sec, Time Remaining: 42 sec
Completed: 2000 of 10000
    Time Spent: 9 sec, Time Remaining: 38 sec
Completed: 3000 of 10000
    Time Spent: 14 sec, Time Remaining: 33 sec
Completed: 4000 of 10000
```

```
Time Spent: 18 sec, Time Remaining: 28 sec
Completed: 5000 of 10000
Time Spent: 24 sec, Time Remaining: 23 sec
Completed: 6000 of 10000
Time Spent: 28 sec, Time Remaining: 19 sec
Completed: 7000 of 10000
Time Spent: 33 sec, Time Remaining: 14 sec
Completed: 8000 of 10000
Time Spent: 38 sec, Time Remaining: 9 sec
Completed: 9000 of 10000
Time Spent: 43 sec, Time Remaining: 4 sec
Completed 10000 in 47 seconds.
```

As shown above, the output periodically displays the amount of time you can expect the program to continue executing. Initially, there is no data to make a prediction with, so the `ProcessEstimator` returns zero seconds, but, as the program executes the `performLengthyProcess()` method 10,000 times, a meaningful time remaining is produced.

## Discussion

The previous example used a method that sleeps for a random number of milliseconds between 1 and 10, and this value is selected using the `RandomUtils` class described in Recipe 8.4. It is easy to predict how long this process is going to take because, on average, each method call is going to sleep for five milliseconds. The `ProcessEstimator` is inaccurate when the amount of time to process each record takes a steadily increasing or decreasing amount of time, or if there is a block of records that takes substantially more or less time to process. If the amount of time to process each record does not remain constant, then the relationship between records processed and time elapsed is not linear. Because the `ProcessEstimator` uses a linear model, `SimpleRegression`, a nonconstant execution time will produce inaccurate predictions for time remaining. If you are using the `ProcessEstimator`, make sure that it takes roughly the same amount of time to process each record.

## See Also

This recipe refers to the `StopWatch` class from Commons Lang. For more information about the `StopWatch` class, see Recipe 1.19.