Course Title: Software Testing, Reliability, and Quality
Course Code: SENG 438
Assignment #: 3
Student Names:

Aashik Ilangovan (30085993)
Emmanuel Omari-Osei (30092729)
Gibran Akmal (30094918)
Priyanka Gautam (30091244)

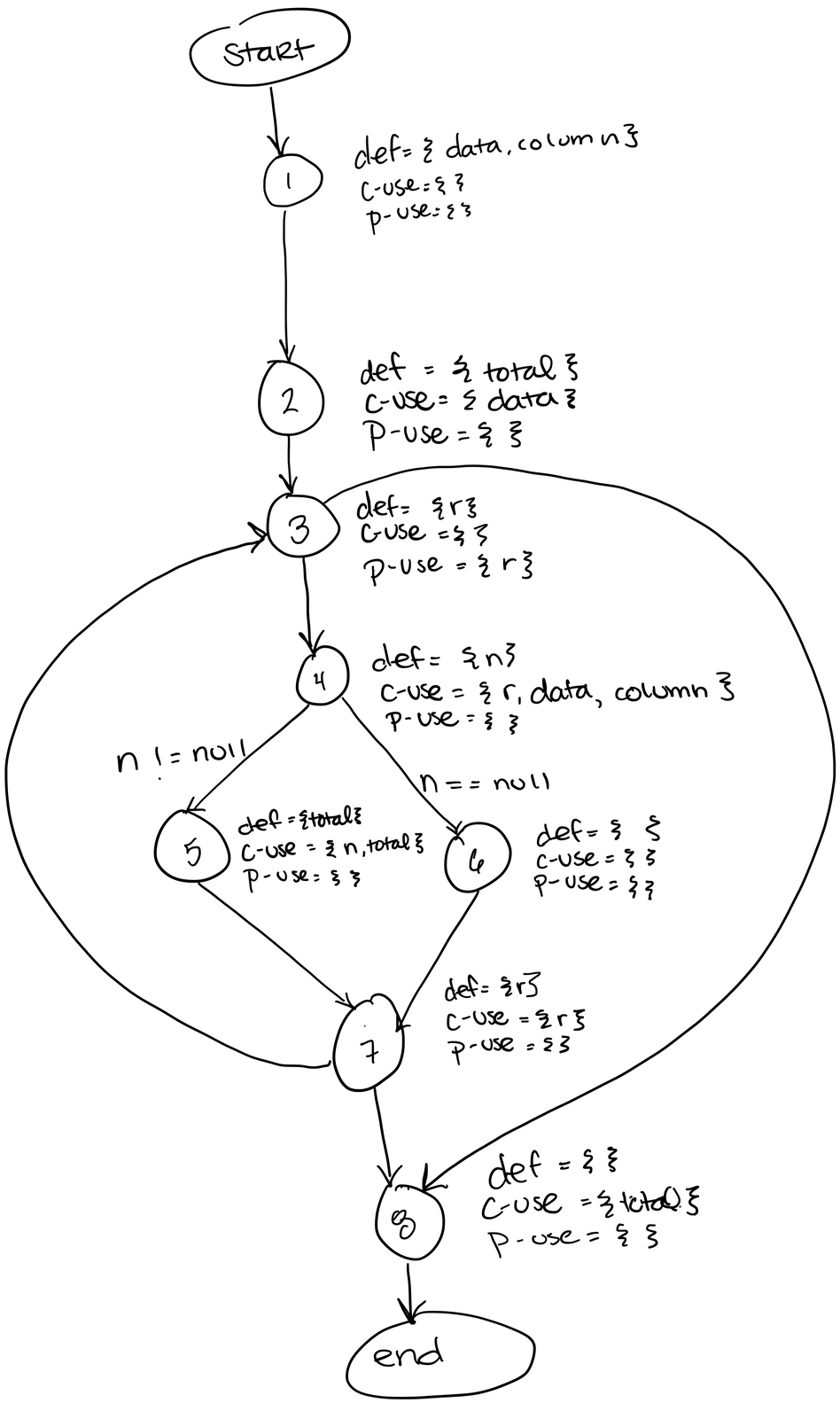
Group Number: 31
Submission Date: 4/03/2022

• Data flow graph for DataUtilities

Saturday, January 22, 2022 8:45 PM

MEASURE DATA FLOW COVERAGE MANUALLY

public static double calculateColumnTotal (Values2D data, int column)



du pairs for variables:

total: (2,5), (5,5), (2,8), (5,8)

data: (1,2), (1,4)

column: (1,4)

r: (3,3), (3,4), (3,7), (7,3), (7,4), (7,7)

n: (4,5)

- For all the test cases designed for this method, all DU pairs were covered.

- DU pair coverage:

$$\frac{(CU_c + PU_c)}{((CU + PU) - (CU_f + PU_f))}$$

$$= \frac{8+1}{[(8+1)-(0+0)]}$$

$$= \frac{9}{9}$$

$$= 1 * 100\%$$

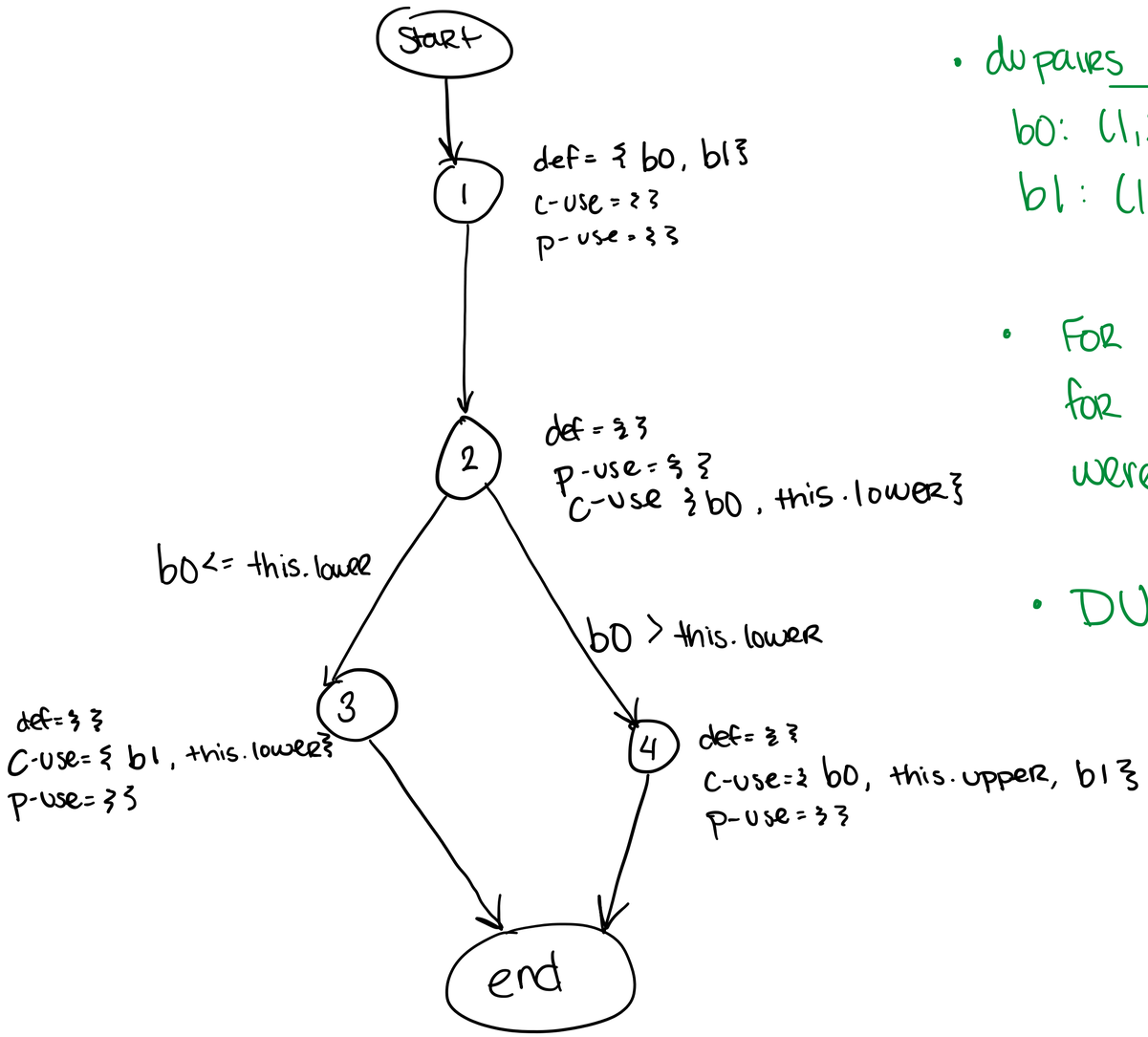
DU pair coverage = 100 %

Data flow graph for Range

Sunday, February 27, 2022 5:25 PM

MEASURE DATA FLOW COVERAGE MANUALLY

public boolean intersects (double b0, double b1)



- du pairs per variable:

  b0: (1,2) , (1,4)

  b1: (1,3) , (1,4)

- For all the test cases designed for this method, all DU pairs were covered.

- DU pair coverage:

$$\frac{(CU_c + PU_c)}{((CU + PU) - (CU_f + PU_f))}$$

(7 over CU_c, 0 over PU_c; 7 over CU, 0 over PU, 0 over CU_f, 0 over PU_f)

$$= \frac{7+0}{[(7+0)-(0+0)]}$$

$$= \frac{7}{7}$$

$$= 1 * 100\%$$

DU pair coverage = 100 %

A high level description of five selected test cases you have designed using coverage information, and how they have increased code coverage:

Method 1

expandToInclude()

```
    public static Range expandToInclude(Range range, double value) {
        if (range == null) {
            return new Range(value, value);
        }
        if (value < range.getLowerBound()) {
            return new Range(value, range.getUpperBound());
        }
        else if (value > range.getUpperBound()) {
            return new Range(range.getLowerBound(), value);
        }
        else {
            return range;
        }
    }
```

Method 2

expand()

```
316     /**
317      * Creates a new range by adding margins to an existing range.
318      *
319      * @param range  the range (<code>null</code> not permitted).
320      * @param lowerMargin  the lower margin (expressed as a percentage of the
321      *                     range length).
322      * @param upperMargin  the upper margin (expressed as a percentage of the
323      *                     range length).
324      *
325      * @return The expanded range.
326      */
327     public static Range expand(Range range,
328                                double lowerMargin, double upperMargin) {
329         ParamChecks.nullNotPermitted(range, "range");
330         double length = range.getLength();
331         double lower = range.getLowerBound() - length * lowerMargin;
332         double upper = range.getUpperBound() + length * upperMargin;
333         if (lower > upper) {
334             lower = lower / 2.0 + upper / 2.0;
335             upper = lower;
336         }
337         return new Range(lower, upper);
338     }
```

Method 3

shift()

```
/**
 * Shifts the range by the specified amount.
 *
 * @param base  the base range (<code>null</code> not permitted).
 * @param delta  the shift amount.
 *
 * @return A new range.
 */
public static Range shift(Range base, double delta) {
    return shift(base, delta, false);
}

/**
 * Shifts the range by the specified amount.
 *
 * @param base  the base range (<code>null</code> not permitted).
 * @param delta  the shift amount.
 * @param allowZeroCrossing  a flag that determines whether or not the
 *                           bounds of the range are allowed to cross
 *                           zero after adjustment.
 *
 * @return A new range.
 */
public static Range shift(Range base, double delta,
                          boolean allowZeroCrossing) {
    ParamChecks.nullNotPermitted(base, "base");
    if (allowZeroCrossing) {
        return new Range(base.getLowerBound() + delta,
                base.getUpperBound() + delta);
    }
    else {
        return new Range(shiftWithNoZeroCrossing(base.getLowerBound(),
                delta), shiftWithNoZeroCrossing(base.getUpperBound(),
                delta));
    }
}
```

Method 4

CalculateRowTotal()

```
/**
 * Returns the total of the values in one row of the supplied data
 * table by taking only the column numbers in the array into account.
 *
 * @param data  the table of values (<code>null</code> not permitted).
 * @param row  the row index (zero-based).
 * @param validCols the array with valid cols (zero-based).
 *
 * @return The total of the valid values in the specified row.
 *
 * @since 1.0.13
 */
public static double calculateRowTotal(Values2D data, int row,
        int[] validCols) {
    ParamChecks.nullNotPermitted(data, "data");
    double total = 0.0;
    int colCount = data.getColumnCount();
    for (int v = 0; v < validCols.length; v++) {
        int col = validCols[v];
        if (col < colCount) {
            Number n = data.getValue(row, col);
            if (n != null) {
                total += n.doubleValue();
            }
        }
    }
    return total;
}
```

Method 5

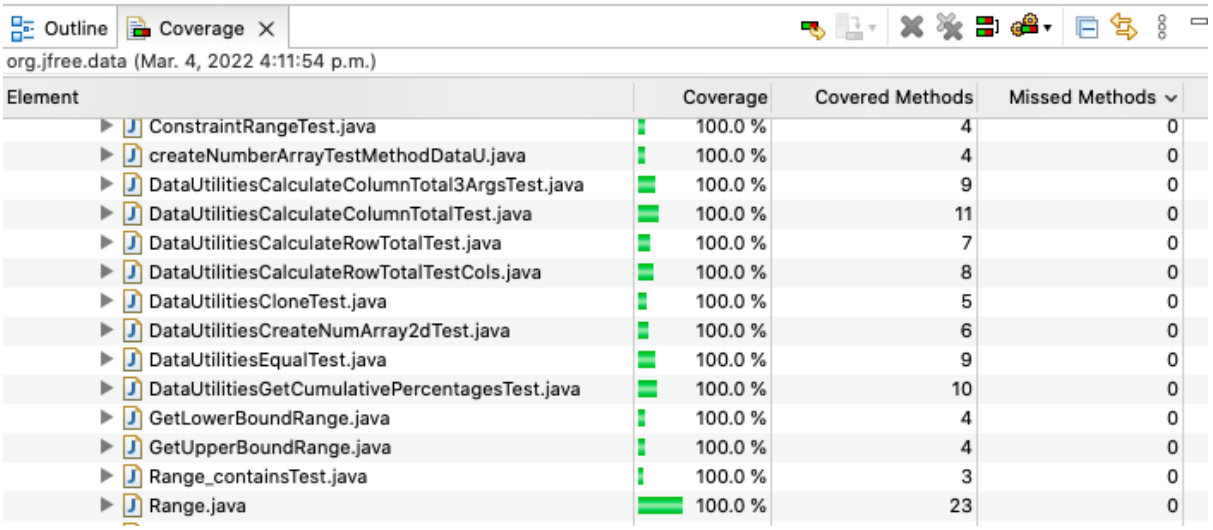
clone()

```
public static double[][] clone(double[][] source) {
    ParamChecks.nullNotPermitted(source, "source");
    double[][] clone = new double[source.length][];
    for (int i = 0; i < source.length; i++) {
        if (source[i] != null) {
            double[] row = new double[source[i].length];
            System.arraycopy(source[i], 0, row, 0, source[i].length);
            clone[i] = row;
        }
    }
    return clone;
}
```

A detailed report of the coverage achieved of each class and method (a screen shot from the code cover results in green and red colour would suffice)

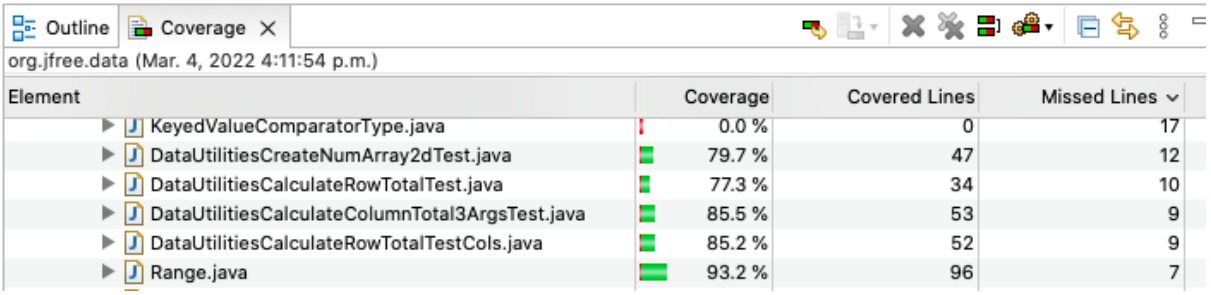
Range

Method Counter ~ 100.0%



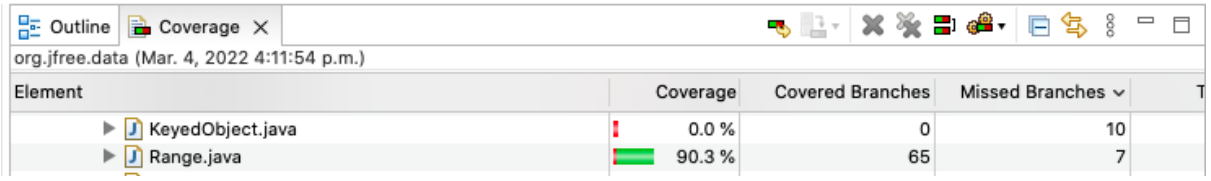
Element	Coverage	Covered Methods	Missed Methods
ConstraintRangeTest.java	100.0 %	4	0
createNumberArrayTestMethodDataU.java	100.0 %	4	0
DataUtilitiesCalculateColumnTotal3ArgsTest.java	100.0 %	9	0
DataUtilitiesCalculateColumnTotalTest.java	100.0 %	11	0
DataUtilitiesCalculateRowTotalTest.java	100.0 %	7	0
DataUtilitiesCalculateRowTotalTestCols.java	100.0 %	8	0
DataUtilitiesCloneTest.java	100.0 %	5	0
DataUtilitiesCreateNumArray2dTest.java	100.0 %	6	0
DataUtilitiesEqualTest.java	100.0 %	9	0
DataUtilitiesGetCumulativePercentagesTest.java	100.0 %	10	0
GetLowerBoundRange.java	100.0 %	4	0
GetUpperBoundRange.java	100.0 %	4	0
Range_containsTest.java	100.0 %	3	0
Range.java	100.0 %	23	0

Line Counter ~ 93.2%



Element	Coverage	Covered Lines	Missed Lines
KeyedValueComparatorType.java	0.0 %	0	17
DataUtilitiesCreateNumArray2dTest.java	79.7 %	47	12
DataUtilitiesCalculateRowTotalTest.java	77.3 %	34	10
DataUtilitiesCalculateColumnTotal3ArgsTest.java	85.5 %	53	9
DataUtilitiesCalculateRowTotalTestCols.java	85.2 %	52	9
Range.java	93.2 %	96	7

Branch Counter ~ 90.3%



Element	Coverage	Covered Branches	Missed Branches
KeyedObject.java	0.0 %	0	10
Range.java	90.3 %	65	7

```
 99     /**
100      * Returns the lower bound for the range.
101      *
102      * @return The lower bound.
103      */
104     public double getLowerBound() {
105         return this.lower;
106     }
107
108     /**
109      * Returns the upper bound for the range.
110      *
111      * @return The upper bound.
112      */
113     public double getUpperBound() {
114         return this.upper;
115     }
116
117     /**
118      * Returns the length of the range.
119      *
120      * @return The length.
121      */
122     public double getLength() {
123         return this.upper - this.lower;
124     }
125
126     /**
127      * Returns the central value for the range.
128      *
129      * @return The central value.
130      */
131     public double getCentralValue() {
132         return this.lower / 2.0 + this.upper / 2.0;
133     }
```

```
135     /**
136      * Returns <code>true</code> if the range contains the specified value and
137      * <code>false</code> otherwise.
138      *
139      * @param value  the value to lookup.
140      *
141      * @return <code>true</code> if the range contains the specified value.
142      */
143     public boolean contains(double value) {
144         return (value >= this.lower && value <= this.upper);
145     }
146
147     /**
148      * Returns <code>true</code> if the range intersects with the specified
149      * range, and <code>false</code> otherwise.
150      *
151      * @param b0  the lower bound (should be &lt;= b1).
152      * @param b1  the upper bound (should be &gt;= b0).
153      *
154      * @return A boolean.
155      */
156     public boolean intersects(double b0, double b1) {
157         if (b0 <= this.lower) {
158             return (b1 > this.lower);
159         }
160         else {
161             return (b0 < this.upper && b1 >= b0);
162         }
163     }
164
165     /**
166      * Returns <code>true</code> if the range intersects with the specified
167      * range, and <code>false</code> otherwise.
168      *
169      * @param range  another range (<code>null</code> not permitted).
170      *
171      * @return A boolean.
172      *
173      * @since 1.0.9
174      */
175     public boolean intersects(Range range) {
176         return intersects(range.getLowerBound(), range.getUpperBound());
177     }
```

```
178
179     /**
180      * Returns the value within the range that is closest to the specified
181      * value.
182      *
183      * @param value  the value.
184      *
185      * @return The constrained value.
186      */
187     public double constrain(double value) {
188         double result = value;
189         if (!contains(value)) {
190             if (value > this.upper) {
191                 result = this.upper;
192             }
193             else if (value < this.lower) {
194                 result = this.lower;
195             }
196         }
197         return result;
198     }
199
200     /**
201      * Creates a new range by combining two existing ranges.
202      * <P>
203      * Note that:
204      * <ul>
205      *   <li>either range can be <code>null</code>, in which case the other
206      *       range is returned;</li>
207      *   <li>if both ranges are <code>null</code> the return value is
208      *       <code>null</code>.</li>
209      * </ul>
210      *
211      * @param range1  the first range (<code>null</code> permitted).
212      * @param range2  the second range (<code>null</code> permitted).
213      *
214      * @return A new range (possibly <code>null</code>).
215      */
216     public static Range combine(Range range1, Range range2) {
217         if (range1 == null) {
218             return range2;
219         }
220         if (range2 == null) {
221             return range1;
222         }
223         double l = Math.min(range1.getLowerBound(), range2.getLowerBound());
224         double u = Math.max(range1.getUpperBound(), range2.getUpperBound());
225         return new Range(l, u);
226     }
228     /**
229      * Returns a new range that spans both <code>range1</code> and
230      * <code>range2</code>.  This method has a special handling to ignore
231      * Double.NaN values.
232      *
233      * @param range1  the first range (<code>null</code> permitted).
234      * @param range2  the second range (<code>null</code> permitted).
235      *
236      * @return A new range (possibly <code>null</code>).
237      *
238      * @since 1.0.15
239      */
240     public static Range combineIgnoringNaN(Range range1, Range range2) {
241         if (range1 == null) {
242             if (range2 != null && range2.isNaNRange()) {
243                 return null;
244             }
245             return range2;
246         }
247         if (range2 == null) {
248             if (range1.isNaNRange()) {
249                 return null;
250             }
251             return range1;
252         }
253         double l = min(range1.getLowerBound(), range2.getLowerBound());
254         double u = max(range1.getUpperBound(), range2.getUpperBound());
255         if (Double.isNaN(l) && Double.isNaN(u)) {
256             return null;
257         }
258         return new Range(l, u);
259     }
```

```
280     private static double max(double d1, double d2) {
281         if (Double.isNaN(d1)) {
282             return d2;
283         }
284         if (Double.isNaN(d2)) {
285             return d1;
286         }
287         return Math.max(d1, d2);
288     }
289
290     /**
291      * Returns a range that includes all the values in the specified
292      * <code>range</code> AND the specified <code>value</code>.
293      *
294      * @param range  the range (<code>null</code> permitted).
295      * @param value  the value that must be included.
296      *
297      * @return A range.
298      *
299      * @since 1.0.1
300      */
301     public static Range expandToInclude(Range range, double value) {
302         if (range == null) {
303             return new Range(value, value);
304         }
305         if (value < range.getLowerBound()) {
306             return new Range(value, range.getUpperBound());
307         }
308         else if (value > range.getUpperBound()) {
309             return new Range(range.getLowerBound(), value);
310         }
311         else {
312             return range;
313         }
314     }

316     /**
317      * Creates a new range by adding margins to an existing range.
318      *
319      * @param range  the range (<code>null</code> not permitted).
320      * @param lowerMargin  the lower margin (expressed as a percentage of the
321      *                     range length).
322      * @param upperMargin  the upper margin (expressed as a percentage of the
323      *                     range length).
324      *
325      * @return The expanded range.
326      */
327     public static Range expand(Range range,
328                                double lowerMargin, double upperMargin) {
329         ParamChecks.nullNotPermitted(range, "range");
330         double length = range.getLength();
331         double lower = range.getLowerBound() - length * lowerMargin;
332         double upper = range.getUpperBound() + length * upperMargin;
333         if (lower > upper) {
334             lower = lower / 2.0 + upper / 2.0;
335             upper = lower;
336         }
337         return new Range(lower, upper);
338     }
339
340     /**
341      * Shifts the range by the specified amount.
342      *
343      * @param base  the base range (<code>null</code> not permitted).
344      * @param delta  the shift amount.
345      *
346      * @return A new range.
347      */
348     public static Range shift(Range base, double delta) {
349         return shift(base, delta, false);
350     }
```

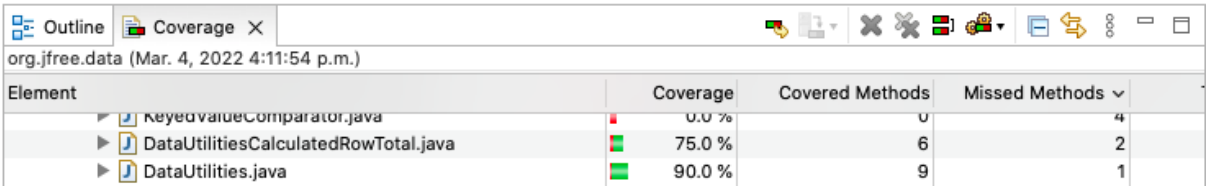
```
351
352     /**
353      * Shifts the range by the specified amount.
354      *
355      * @param base  the base range (<code>null</code> not permitted).
356      * @param delta  the shift amount.
357      * @param allowZeroCrossing  a flag that determines whether or not the
358      *                           bounds of the range are allowed to cross
359      *                           zero after adjustment.
360      *
361      * @return A new range.
362      */
363     public static Range shift(Range base, double delta,
364                               boolean allowZeroCrossing) {
365         ParamChecks.nullNotPermitted(base, "base");
366         if (allowZeroCrossing) {
367             return new Range(base.getLowerBound() + delta,
368                     base.getUpperBound() + delta);
369         }
370         else {
371             return new Range(shiftWithNoZeroCrossing(base.getLowerBound(),
372                     delta), shiftWithNoZeroCrossing(base.getUpperBound(),
373                     delta));
374         }
375     }
```

```
398     /**
399      * Scales the range by the specified factor.
400      *
401      * @param base the base range (<code>null</code> not permitted).
402      * @param factor the scaling factor (must be non-negative).
403      *
404      * @return A new range.
405      *
406      * @since 1.0.9
407      */
408     public static Range scale(Range base, double factor) {
409         ParamChecks.nullNotPermitted(base, "base");
410         if (factor < 0) {
411             throw new IllegalArgumentException("Negative 'factor' argument.");
412         }
413         return new Range(base.getLowerBound() * factor,
414                 base.getUpperBound() * factor);
415     }
416
417     /**
418      * Tests this object for equality with an arbitrary object.
419      *
420      * @param obj  the object to test against (<code>null</code> permitted).
421      *
422      * @return A boolean.
423      */
424     @Override
425     public boolean equals(Object obj) {
426         if (!(obj instanceof Range)) {
427             return false;
428         }
429         Range range = (Range) obj;
430         if (!(this.lower == range.lower)) {
431             return false;
432         }
433         if (!(this.upper == range.upper)) {
434             return false;
435         }
436         return true;
437     }
```

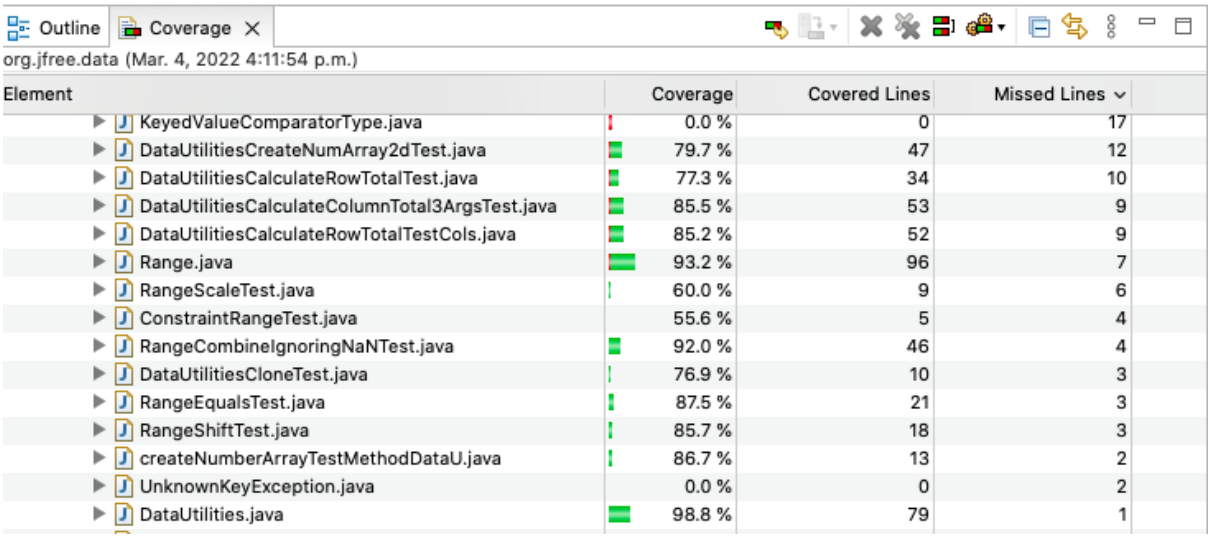
```
456     @Override
457     public int hashCode() {
458         int result;
459         long temp;
460         temp = Double.doubleToLongBits(this.lower);
461         result = (int) (temp ^ (temp >>> 32));
462         temp = Double.doubleToLongBits(this.upper);
463         result = 29 * result + (int) (temp ^ (temp >>> 32));
464         return result;
465     }
466
467     /**
468      * Returns a string representation of this Range.
469      *
470      * @return A String "Range[lower,upper]" where lower=lower range and
471      *         upper=upper range.
472      */
473     @Override
474     public String toString() {
475         return ("Range[" + this.lower + "," + this.upper + "]");
476     }
477
478 }
```

DataUtilities

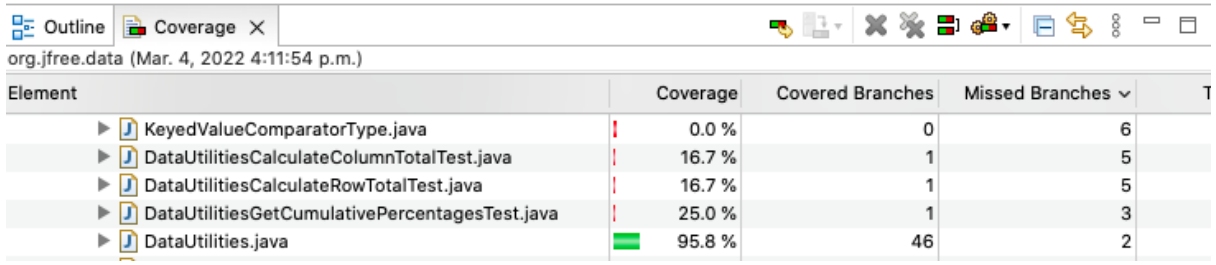
Method Counter ~ 90.0%



Line Counter ~ 98.8%



Branch Counter ~ 95.8%




```
61      /**
62       * Tests two arrays for equality.  To be considered equal, the arrays must
63       * have exactly the same dimensions, and the values in each array must also
64       * match (two values that are both NaN or both INF are considered equal
65       * in this test).
66       *
67       * @param a  the first array (<code>null</code> permitted).
68       * @param b  the second array (<code>null</code> permitted).
69       *
70       * @return A boolean.
71       *
72       * @since 1.0.13
73       */
74      public static boolean equal(double[][] a, double[][] b) {
75          if (a == null) {
76              return (b == null);
77          }
78          if (b == null) {
79              return false;  // already know 'a' isn't null
80          }
81          if (a.length != b.length) {
82              return false;
83          }
84          for (int i = 0; i < a.length; i++) {
85              if (!Arrays.equals(a[i], b[i])) {
86                  return false;
87              }
88          }
89          return true;
90      }
91
```

```
92      /**
93       * Returns a clone of the specified array.
94       *
95       * @param source  the source array (<code>null</code> not permitted).
96       *
97       * @return A clone of the array.
98       *
99       * @since 1.0.13
100      */
101     public static double[][] clone(double[][] source) {
102         ParamChecks.nullNotPermitted(source, "source");
103         double[][] clone = new double[source.length][];
104         for (int i = 0; i < source.length; i++) {
105             if (source[i] != null) {
106                 double[] row = new double[source[i].length];
107                 System.arraycopy(source[i], 0, row, 0, source[i].length);
108                 clone[i] = row;
109             }
110         }
111         return clone;
112     }
113
114     /**
115      * Returns the total of the values in one column of the supplied data
116      * table.
117      *
118      * @param data  the table of values (<code>null</code> not permitted).
119      * @param column  the column index (zero-based).
120      *
121      * @return The total of the values in the specified column.
122      */
123     public static double calculateColumnTotal(Values2D data, int column) {
124         ParamChecks.nullNotPermitted(data, "data");
125         double total = 0.0;
126         int rowCount = data.getRowCount();
127         for (int r = 0; r < rowCount; r++) {
128             Number n = data.getValue(r, column);
129             if (n != null) {
130                 total += n.doubleValue();
131             }
132         }
133         return total;
134     }
```

```
136     /**
137      * Returns the total of the values in one column of the supplied data
138      * table by taking only the row numbers in the array into account.
139      *
140      * @param data  the table of values (<code>null</code> not permitted).
141      * @param column  the column index (zero-based).
142      * @param validRows the array with valid rows (zero-based).
143      *
144      * @return The total of the valid values in the specified column.
145      *
146      * @since 1.0.13
147      */
148     public static double calculateColumnTotal(Values2D data, int column,
149             int[] validRows) {
150         ParamChecks.nullNotPermitted(data, "data");
151         double total = 0.0;
152         int rowCount = data.getRowCount();
153         for (int v = 0; v < validRows.length; v++) {
154             int row = validRows[v];
155             if (row < rowCount) {
156                 Number n = data.getValue(row, column);
157                 if (n != null) {
158                     total += n.doubleValue();
159                 }
160             }
161         }
162         return total;
163     }
164
165     /**
166      * Returns the total of the values in one row of the supplied data
167      * table.
168      *
169      * @param data  the table of values (<code>null</code> not permitted).
170      * @param row  the row index (zero-based).
171      *
172      * @return The total of the values in the specified row.
173      */
174     public static double calculateRowTotal(Values2D data, int row) {
175         ParamChecks.nullNotPermitted(data, "data");
176         double total = 0.0;
177         int columnCount = data.getColumnCount();
178         for (int c = 0; c < columnCount; c++) {
179             Number n = data.getValue(row, c);
180             if (n != null) {
181                 total += n.doubleValue();
182             }
183         }
184         return total;
185     }
```

```
187     /**
188      * Returns the total of the values in one row of the supplied data
189      * table by taking only the column numbers in the array into account.
190      *
191      * @param data  the table of values (<code>null</code> not permitted).
192      * @param row  the row index (zero-based).
193      * @param validCols the array with valid cols (zero-based).
194      *
195      * @return The total of the valid values in the specified row.
196      *
197      * @since 1.0.13
198      */
199     public static double calculateRowTotal(Values2D data, int row,
200             int[] validCols) {
201         ParamChecks.nullNotPermitted(data, "data");
202         double total = 0.0;
203         int colCount = data.getColumnCount();
204         for (int v = 0; v < validCols.length; v++) {
205             int col = validCols[v];
206             if (col < colCount) {
207                 Number n = data.getValue(row, col);
208                 if (n != null) {
209                     total += n.doubleValue();
210                 }
211             }
212         }
213         return total;
214     }
215
216     /**
217      * Constructs an array of <code>Number</code> objects from an array of
218      * <code>double</code> primitives.
219      *
220      * @param data  the data (<code>null</code> not permitted).
221      *
222      * @return An array of <code>Double</code>.
223      */
224     public static Number[] createNumberArray(double[] data) {
225         ParamChecks.nullNotPermitted(data, "data");
226         Number[] result = new Number[data.length];
227         for (int i = 0; i < data.length; i++) {
228             result[i] = new Double(data[i]);
229         }
230         return result;
231     }
```

```
233     /**
234      * Constructs an array of arrays of <code>Number</code> objects from a
235      * corresponding structure containing <code>double</code> primitives.
236      *
237      * @param data  the data (<code>null</code> not permitted).
238      *
239      * @return An array of <code>Double</code>.
240      */
241     public static Number[][] createNumberArray2D(double[][] data) {
242         ParamChecks.nullNotPermitted(data, "data");
243         int l1 = data.length;
244         Number[][] result = new Number[l1][];
245         for (int i = 0; i < l1; i++) {
246             result[i] = createNumberArray(data[i]);
247         }
248         return result;
249     }
250
251     /**
252      * Returns a {@link KeyedValues} instance that contains the cumulative
253      * percentage values for the data in another {@link KeyedValues} instance.
254      * <p>
255      * The percentages are values between 0.0 and 1.0 (where 1.0 = 100%).
256      *
257      * @param data  the data (<code>null</code> not permitted).
258      *
259      * @return The cumulative percentages.
260      */
261     public static KeyedValues getCumulativePercentages(KeyedValues data) {
262         ParamChecks.nullNotPermitted(data, "data");
263         DefaultKeyedValues result = new DefaultKeyedValues();
264         double total = 0.0;
265         for (int i = 0; i < data.getItemCount(); i++) {
266             Number v = data.getValue(i);
267             if (v != null) {
268                 total = total + v.doubleValue();
269             }
270         }
271         double runningTotal = 0.0;
272         for (int i = 0; i < data.getItemCount(); i++) {
273             Number v = data.getValue(i);
274             if (v != null) {
275                 runningTotal = runningTotal + v.doubleValue();
276             }
277             result.addValue(data.getKey(i), new Double(runningTotal / total));
278         }
279         return result;
280     }
281
282 }
```