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)

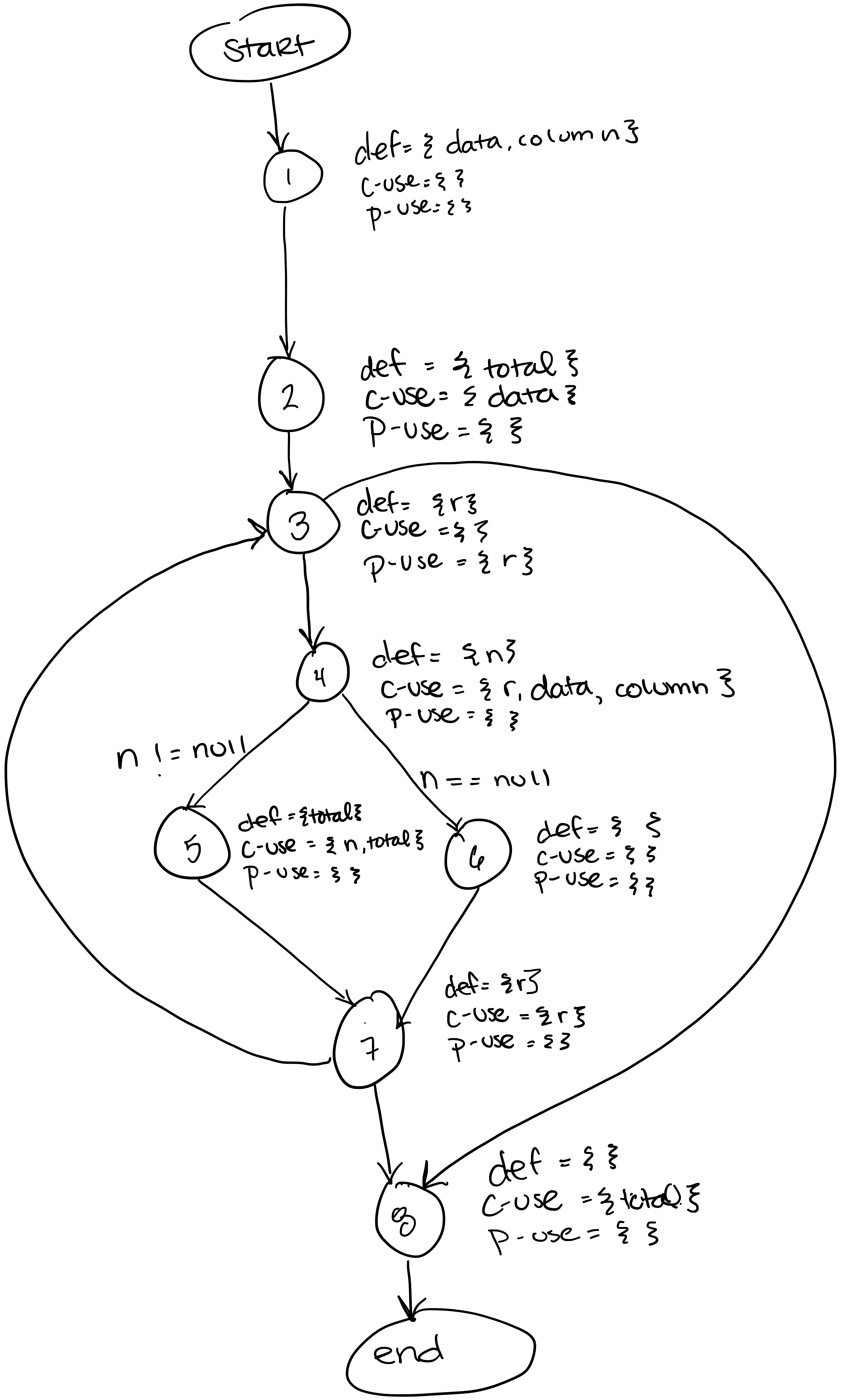
Group Number: 31
Submission Date: 4/03/2022

- Data flow graph for DataUtilities

Saturday, January 22, 2022 8:45 PM

MEASURE DATA FLOW COVERAGE MANUALLY

public static double calculateColumnTotal (Values 2D 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:

$$\frac{(\cancelto{8}{CU_c} + \cancelto{1}{PU_c})}{((\cancelto{8}{CU} + \cancelto{1}{PU}) - (\cancelto{0}{CU_f} + \cancelto{0}{PU_f}))}$$

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

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

$= 1 * 100\%$

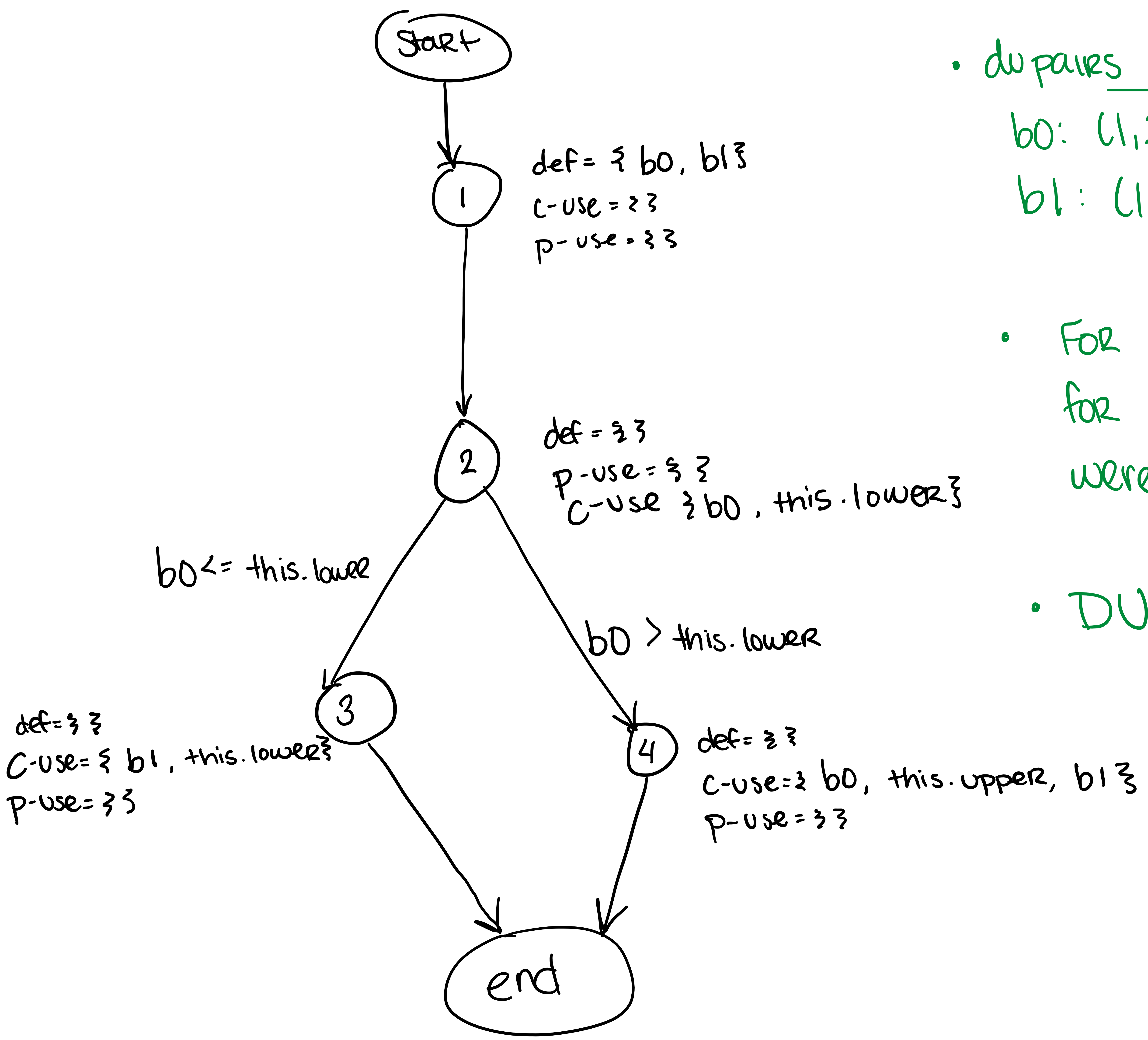
DU pair coverage = 100 %

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:

$$\frac{(\overset{7}{\cancel{CU_c}} + \overset{0}{\cancel{PU_c}})}{((\overset{7}{\cancel{CU}} + \overset{0}{\cancel{PU}}) - (\overset{0}{\cancel{CU_f}} + \overset{0}{\cancel{PU_f}}))}$$

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

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

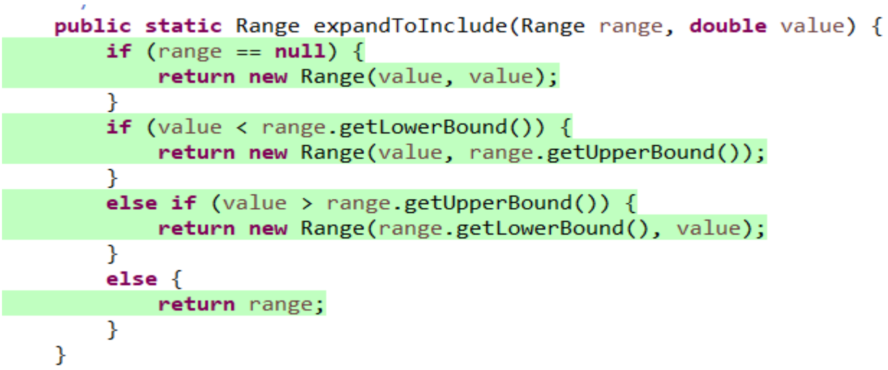
$$= 1 * 100\%$$

$$\boxed{\text{DU pair coverage} = 100\ \%}$$

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

Method 1

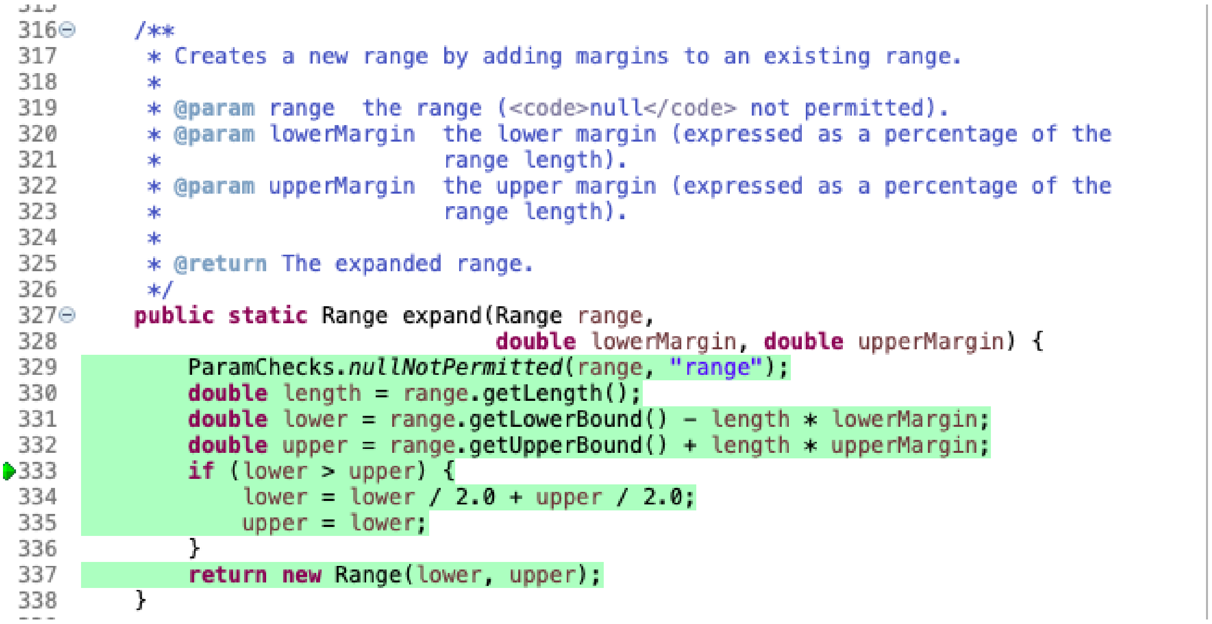
expandToInclude()



```
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

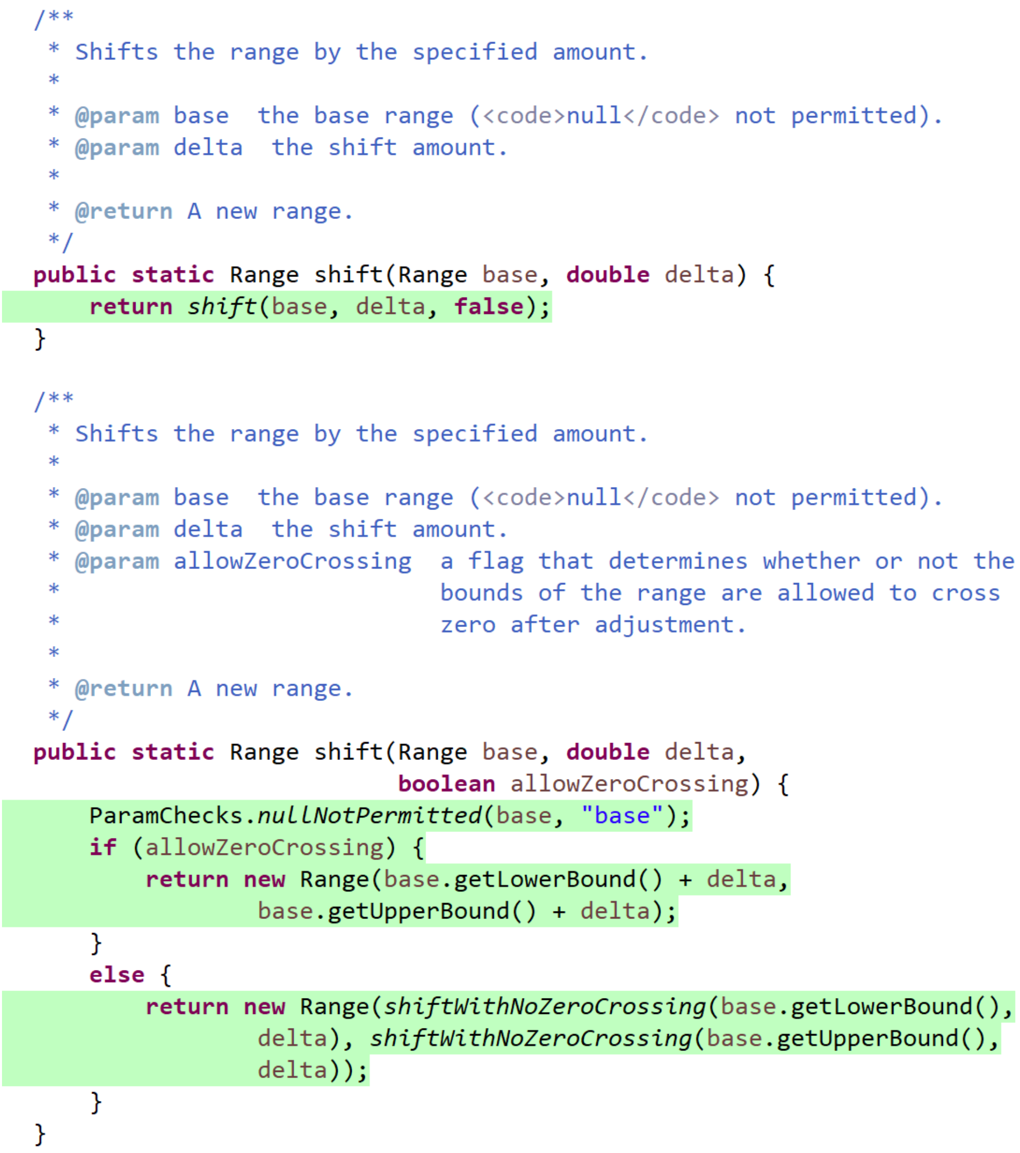
expand()



```
    /**
     * Creates a new range by adding margins to an existing range.
     *
     * @param range  the range (<code>null</code> not permitted).
     * @param lowerMargin  the lower margin (expressed as a percentage of the
     *                     range length).
     * @param upperMargin  the upper margin (expressed as a percentage of the
     *                     range length).
     *
     * @return The expanded range.
     */
    public static Range expand(Range range,
                               double lowerMargin, double upperMargin) {
        ParamChecks.nullNotPermitted(range, "range");
        double length = range.getLength();
        double lower = range.getLowerBound() - length * lowerMargin;
        double upper = range.getUpperBound() + length * upperMargin;
        if (lower > upper) {
            lower = lower / 2.0 + upper / 2.0;
            upper = lower;
        }
        return new Range(lower, upper);
    }
```

Method 3

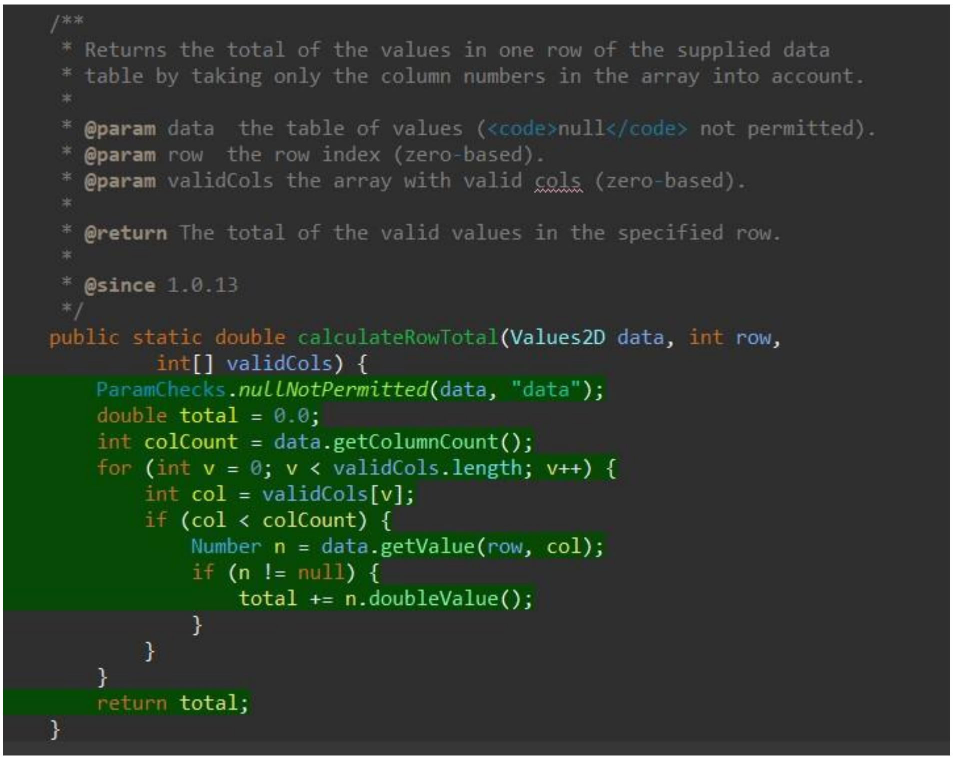
shift()



```
/**
 * 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

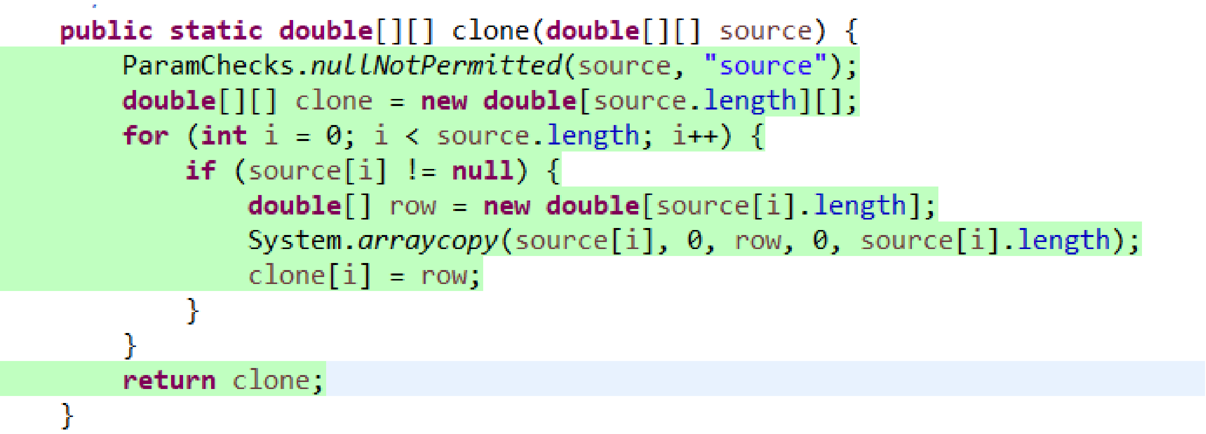
CalculateRowTotal()



```
/**
 * 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()

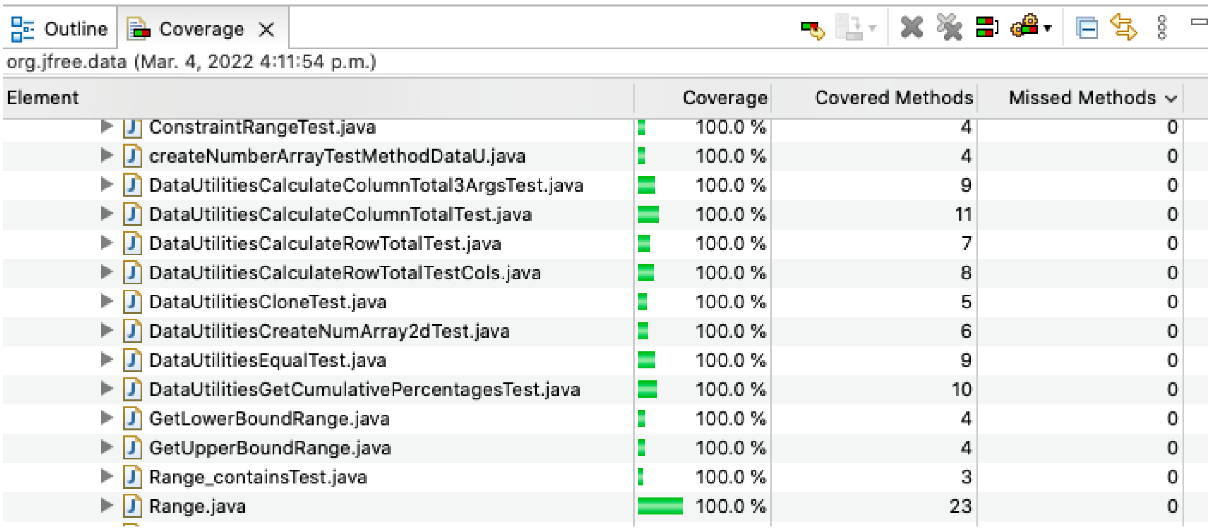


```
public static double[][] clone(double[][] source) {
    ParamChecks.nullNotPermitted(source, "source");
    double[][] clone = new double[source.length][];
    for (int i = 0; i < source.length; i++) {
        if (source[i] != null) {
            double[] row = new double[source[i].length];
    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

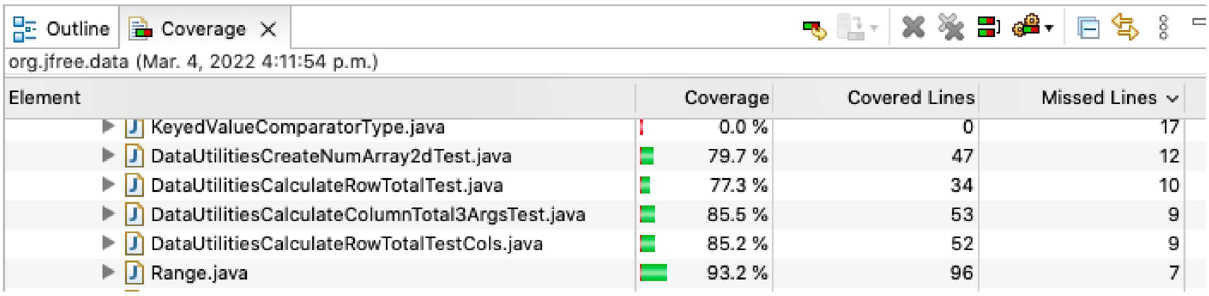
Method Counter ~ 100.0%



org.jfree.data (Mar. 4, 2022 4:11:54 p.m.)

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

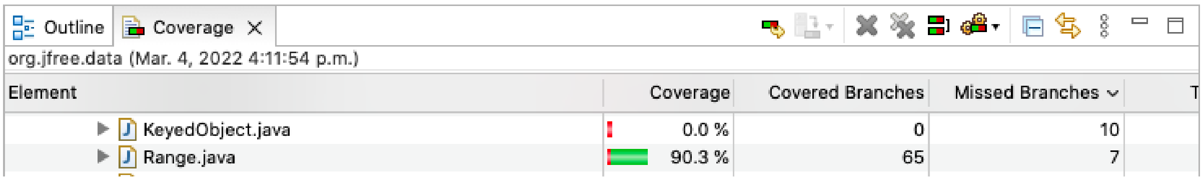
Line Counter ~ 93.2%



org.jfree.data (Mar. 4, 2022 4:11:54 p.m.)

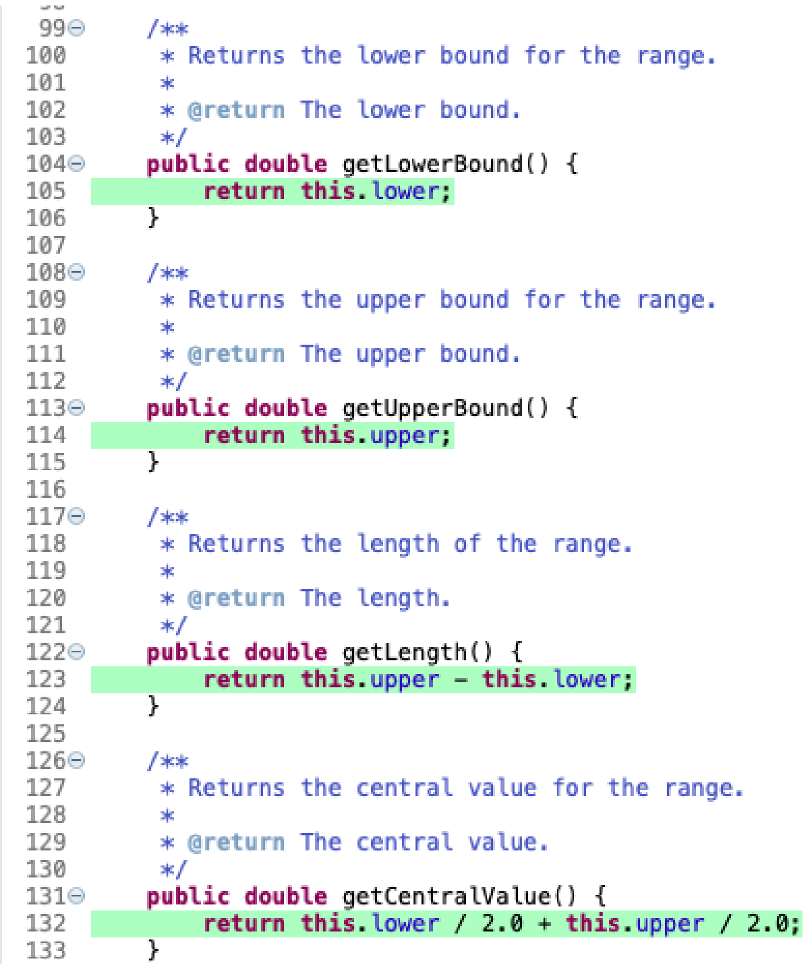
Element	Coverage	Covered Lines	Missed Lines
KeyedValueComparatorType.java	🟥 0.0 %	0	17
DataUtilitiesCreateNumArray2dTest.java	🟩 79.7 %	47	12
DataUtilitiesCalculateRowTotalTest.java	🟩 77.3 %	34	10
DataUtilitiesCalculateColumnTotal3ArgsTest.java	🟩 85.5 %	53	9
DataUtilitiesCalculateRowTotalTestCols.java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

Branch Counter ~ 90.3%

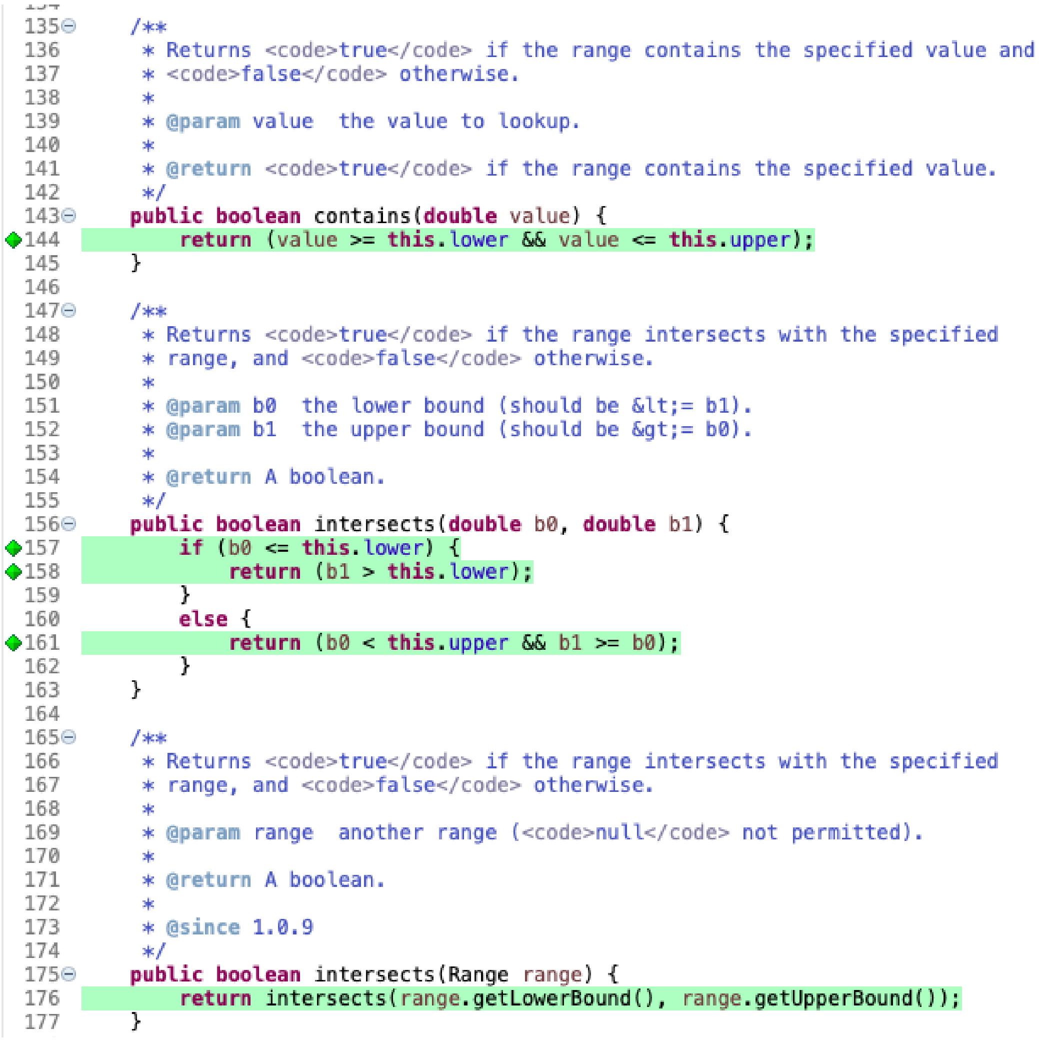


org.jfree.data (Mar. 4, 2022 4:11:54 p.m.)

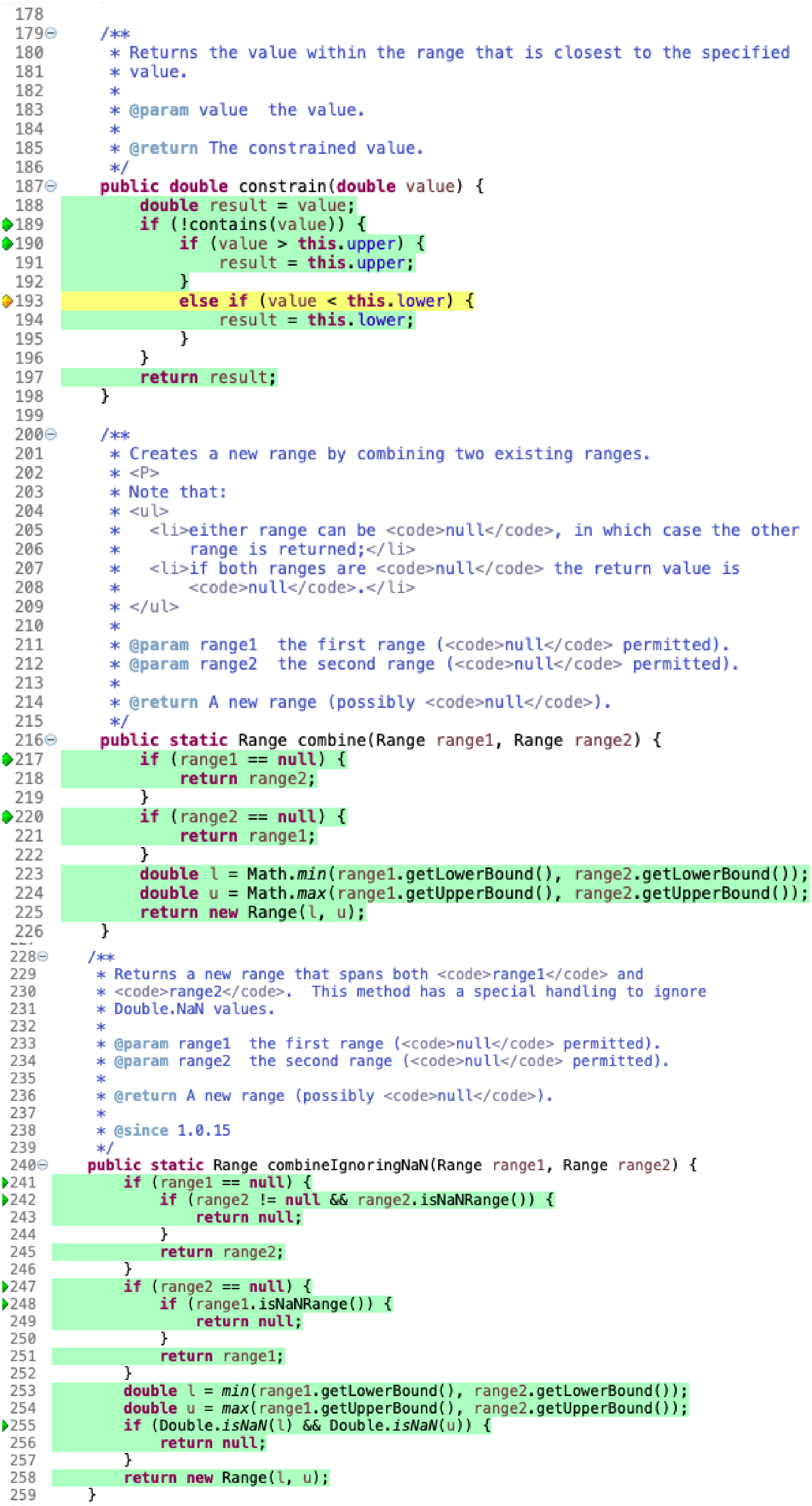
Element	Coverage	Covered Branches	Missed Branches	T
KeyedObject.java	🟥 0.0 %	0	10	
Range.java	🟥🟩 90.3 %	65	7	



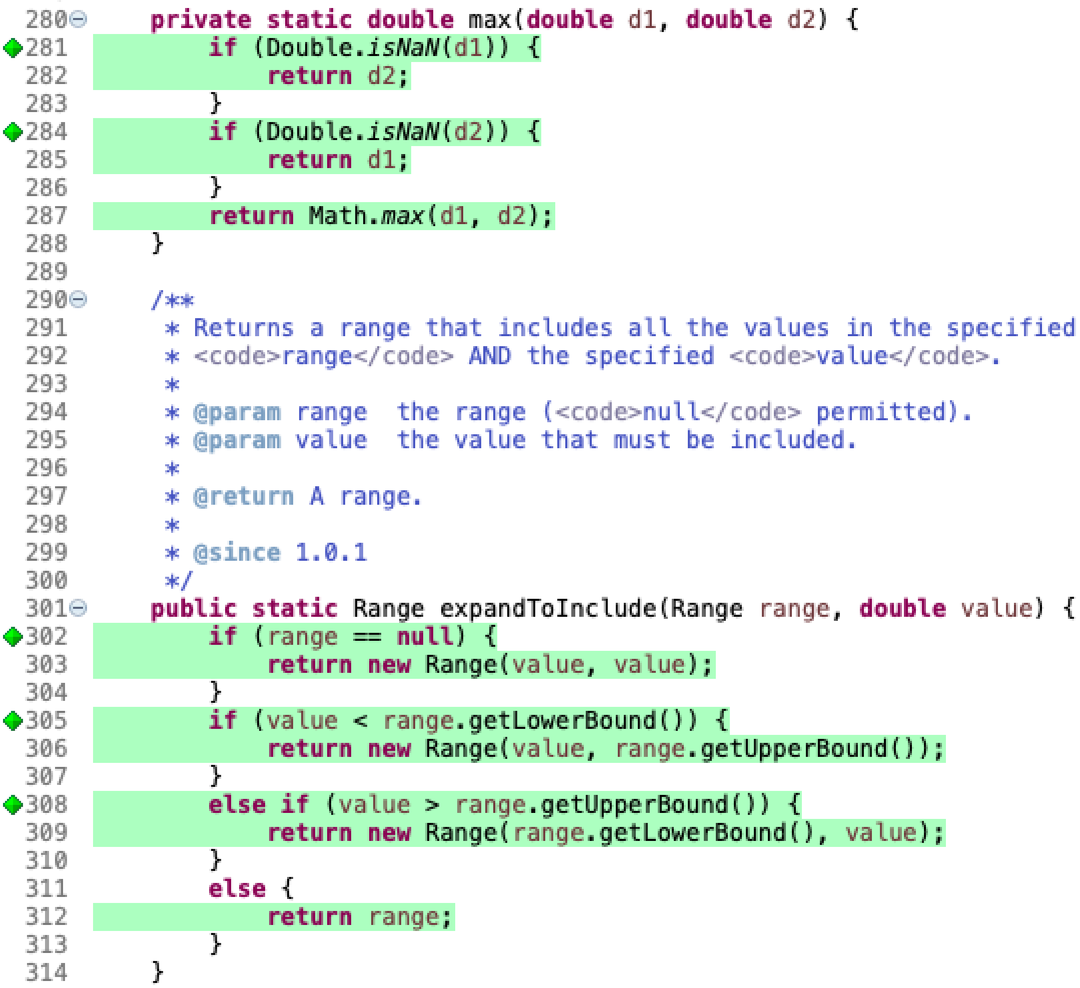
```
    /**
     * Returns the lower bound for the range.
     *
     * @return The lower bound.
     */
    public double getLowerBound() {
        return this.lower;
    }

    /**
     * Returns the upper bound for the range.
     *
     * @return The upper bound.
     */
    public double getUpperBound() {
        return this.upper;
    }

    /**
     * Returns the length of the range.
     *
     * @return The length.
     */
    public double getLength() {
        return this.upper - this.lower;
    }

    /**
     * Returns the central value for the range.
     *
     * @return The central value.
     */
    public double getCentralValue() {
        return this.lower / 2.0 + this.upper / 2.0;
    }
```



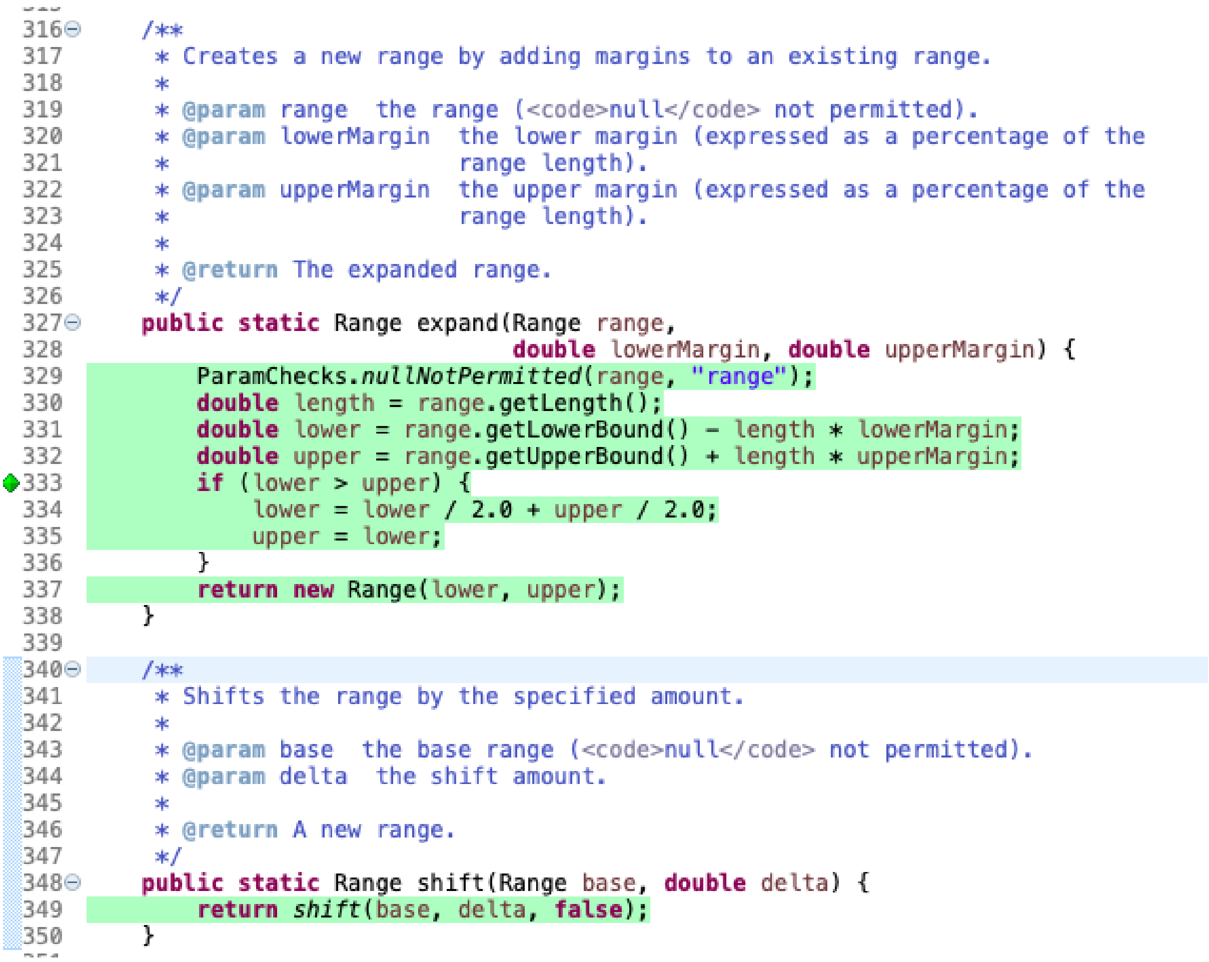
```
    /**
     * Returns <code>true</code> if the range contains the specified value and
     * <code>false</code> otherwise.
     *
     * @param value  the value to lookup.
     *
     * @return <code>true</code> if the range contains the specified value.
     */
    public boolean contains(double value) {
        return (value >= this.lower && value <= this.upper);
    }

    /**
     * Returns <code>true</code> if the range intersects with the specified
     * range, and <code>false</code> otherwise.
     *
     * @param b0  the lower bound (should be &lt;= b1).
     * @param b1  the upper bound (should be &gt;= b0).
     *
     * @return A boolean.
     */
    public boolean intersects(double b0, double b1) {
        if (b0 <= this.lower) {
            return (b1 > this.lower);
        }
        else {
            return (b0 < this.upper && b1 >= b0);
        }
    }

    /**
     * Returns <code>true</code> if the range intersects with the specified
     * range, and <code>false</code> otherwise.
     *
     * @param range  another range (<code>null</code> not permitted).
     *
     * @return A boolean.
     *
     * @since 1.0.9
     */
    public boolean intersects(Range range) {
        return intersects(range.getLowerBound(), range.getUpperBound());
    }
```



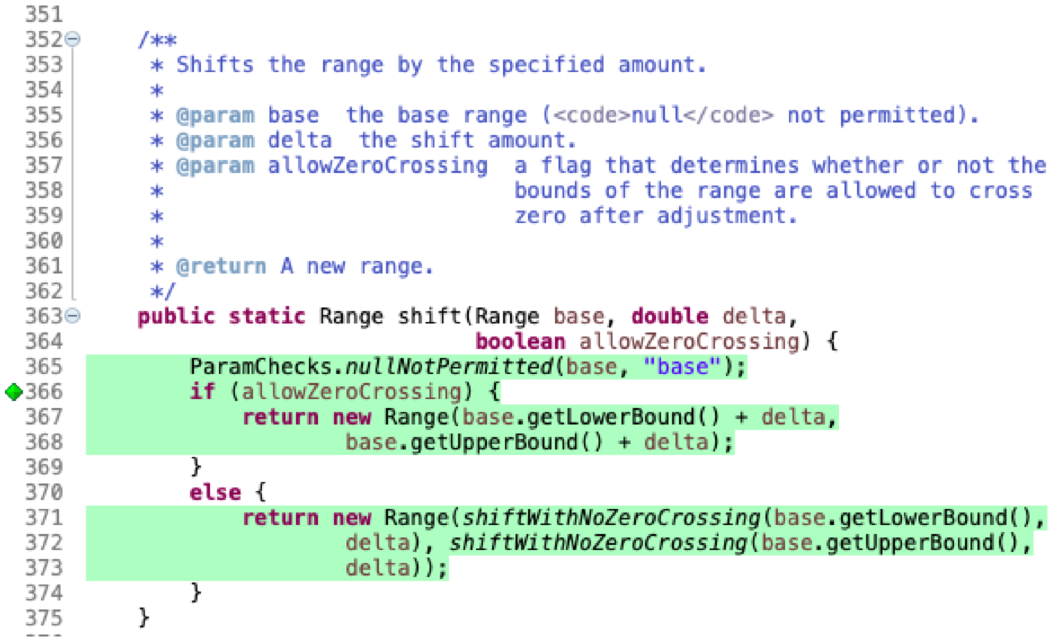
```

    /**
     * Returns the value within the range that is closest to the specified
     * value.
     *
     * @param value  the value.
     *
     * @return The constrained value.
     */
    public double constrain(double value) {
        double result = value;
        if (!contains(value)) {
            if (value > this.upper) {
                result = this.upper;
            }
            else if (value < this.lower) {
                result = this.lower;
            }
        }
        return result;
    }

    /**
     * Creates a new range by combining two existing ranges.
     * <P>
     * Note that:
     * <ul>
     *   <li>either range can be <code>null</code>, in which case the other
     *       range is returned;</li>
     *   <li>if both ranges are <code>null</code> the return value is
     *       <code>null</code>.</li>
     * </ul>
     *
     * @param range1  the first range (<code>null</code> permitted).
     * @param range2  the second range (<code>null</code> permitted).
     *
     * @return A new range (possibly <code>null</code>).
     */
    public static Range combine(Range range1, Range range2) {
        if (range1 == null) {
            return range2;
        }
        if (range2 == null) {
            return range1;
        }
        double l = Math.min(range1.getLowerBound(), range2.getLowerBound());
        double u = Math.max(range1.getUpperBound(), range2.getUpperBound());
        return new Range(l, u);
    }
    /**
     * Returns a new range that spans both <code>range1</code> and
     * <code>range2</code>.  This method has a special handling to ignore
     * Double.NaN values.
     *
     * @param range1  the first range (<code>null</code> permitted).
     * @param range2  the second range (<code>null</code> permitted).
     *
     * @return A new range (possibly <code>null</code>).
     *
     * @since 1.0.15
     */
    public static Range combineIgnoringNaN(Range range1, Range range2) {
        if (range1 == null) {
            if (range2 != null && range2.isNaNRange()) {
                return null;
            }
            return range2;
        }
        if (range2 == null) {
            if (range1.isNaNRange()) {
                return null;
            }
            return range1;
        }
        double l = min(range1.getLowerBound(), range2.getLowerBound());
        double u = max(range1.getUpperBound(), range2.getUpperBound());
        if (Double.isNaN(l) && Double.isNaN(u)) {
            return null;
        }
        return new Range(l, u);
    }
```



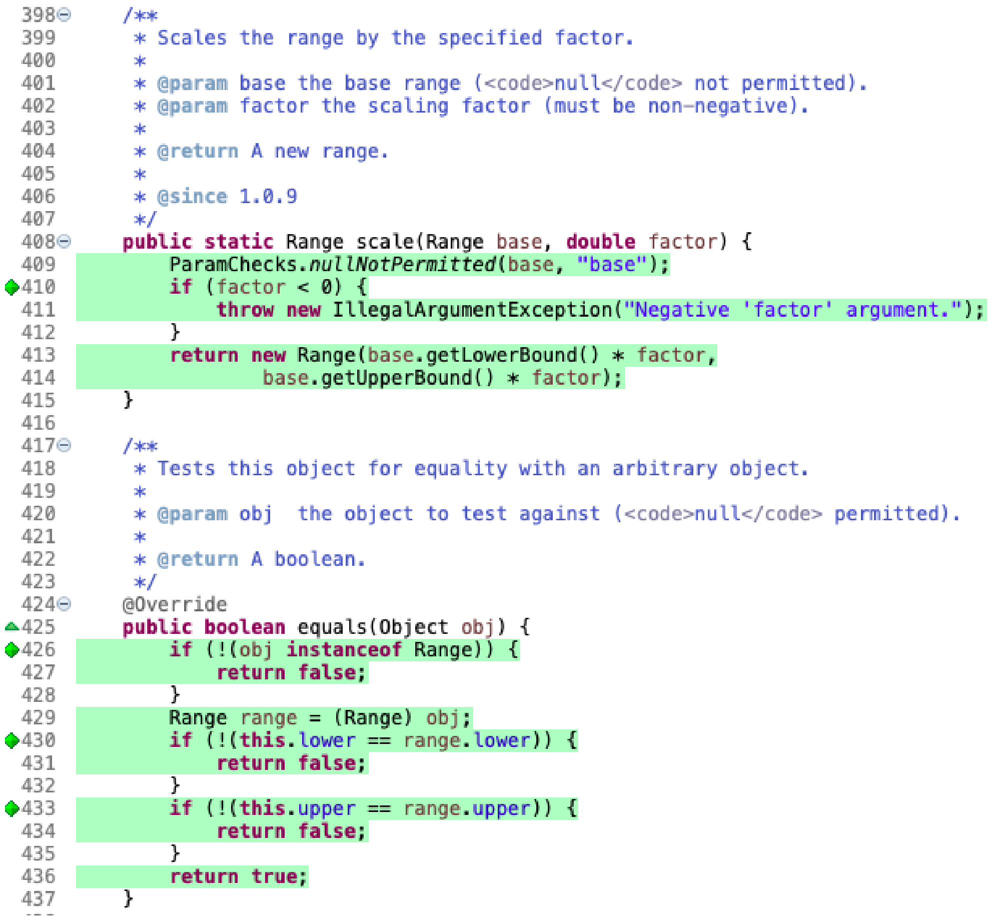
```java
    private static double max(double d1, double d2) {
        if (Double.isNaN(d1)) {
            return d2;
        }
        if (Double.isNaN(d2)) {
            return d1;
        }
        return Math.max(d1, d2);
    }

    /**
     * Returns a range that includes all the values in the specified
     * <code>range</code> AND the specified <code>value</code>.
     *
     * @param range  the range (<code>null</code> permitted).
     * @param value  the value that must be included.
     *
     * @return A range.
     *
     * @since 1.0.1
     */
    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;
        }
    }
```



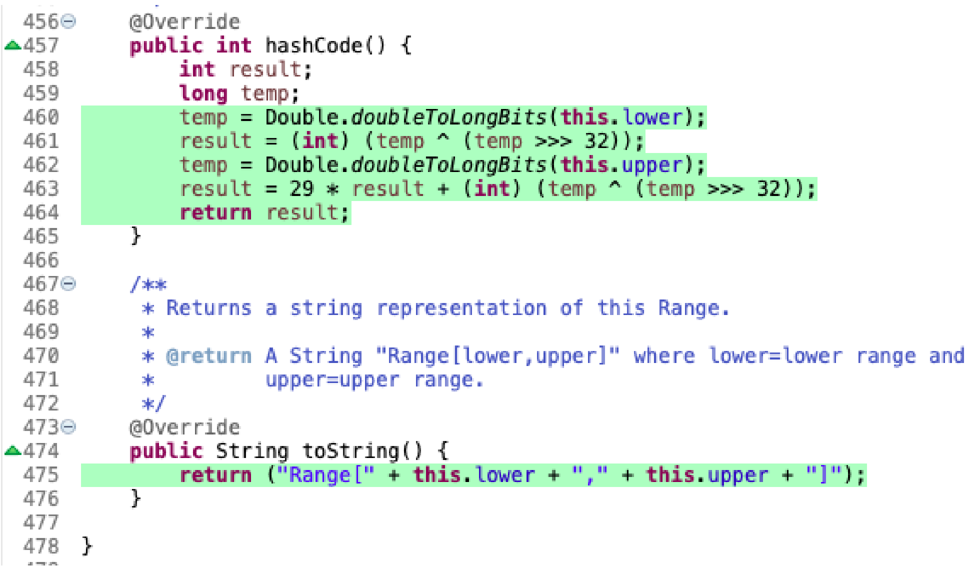
```java
    /**
     * Creates a new range by adding margins to an existing range.
     *
     * @param range  the range (<code>null</code> not permitted).
     * @param lowerMargin  the lower margin (expressed as a percentage of the
     *                     range length).
     * @param upperMargin  the upper margin (expressed as a percentage of the
     *                     range length).
     *
     * @return The expanded range.
     */
    public static Range expand(Range range,
                               double lowerMargin, double upperMargin) {
        ParamChecks.nullNotPermitted(range, "range");
        double length = range.getLength();
        double lower = range.getLowerBound() - length * lowerMargin;
        double upper = range.getUpperBound() + length * upperMargin;
        if (lower > upper) {
            lower = lower / 2.0 + upper / 2.0;
            upper = lower;
        }
        return new Range(lower, upper);
    }

    /**
     * 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);
    }
```



```java

    /**
     * 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));
        }
    }
```



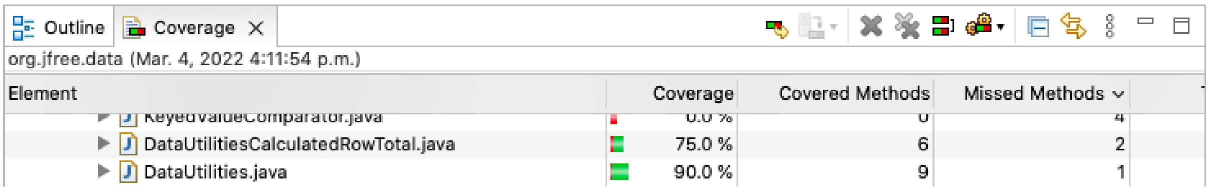
```java
    /**
     * Scales the range by the specified factor.
     *
     * @param base the base range (<code>null</code> not permitted).
     * @param factor the scaling factor (must be non-negative).
     *
     * @return A new range.
     *
     * @since 1.0.9
     */
    public static Range scale(Range base, double factor) {
        ParamChecks.nullNotPermitted(base, "base");
        if (factor < 0) {
            throw new IllegalArgumentException("Negative 'factor' argument.");
        }
        return new Range(base.getLowerBound() * factor,
                base.getUpperBound() * factor);
    }

    /**
     * Tests this object for equality with an arbitrary object.
     *
     * @param obj  the object to test against (<code>null</code> permitted).
     *
     * @return A boolean.
     */
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Range)) {
            return false;
        }
        Range range = (Range) obj;
        if (!(this.lower == range.lower)) {
            return false;
        }
        if (!(this.upper == range.upper)) {
            return false;
        }
        return true;
    }
```



```
    @Override
    public int hashCode() {
        int result;
        long temp;
        temp = Double.doubleToLongBits(this.lower);
        result = (int) (temp ^ (temp >>> 32));
        temp = Double.doubleToLongBits(this.upper);
        result = 29 * result + (int) (temp ^ (temp >>> 32));
        return result;
    }

    /**
     * Returns a string representation of this Range.
     *
     * @return A String "Range[lower,upper]" where lower=lower range and
     *         upper=upper range.
     */
    @Override
    public String toString() {
        return ("Range[" + this.lower + "," + this.upper + "]");
    }

}
```

DataUtilities

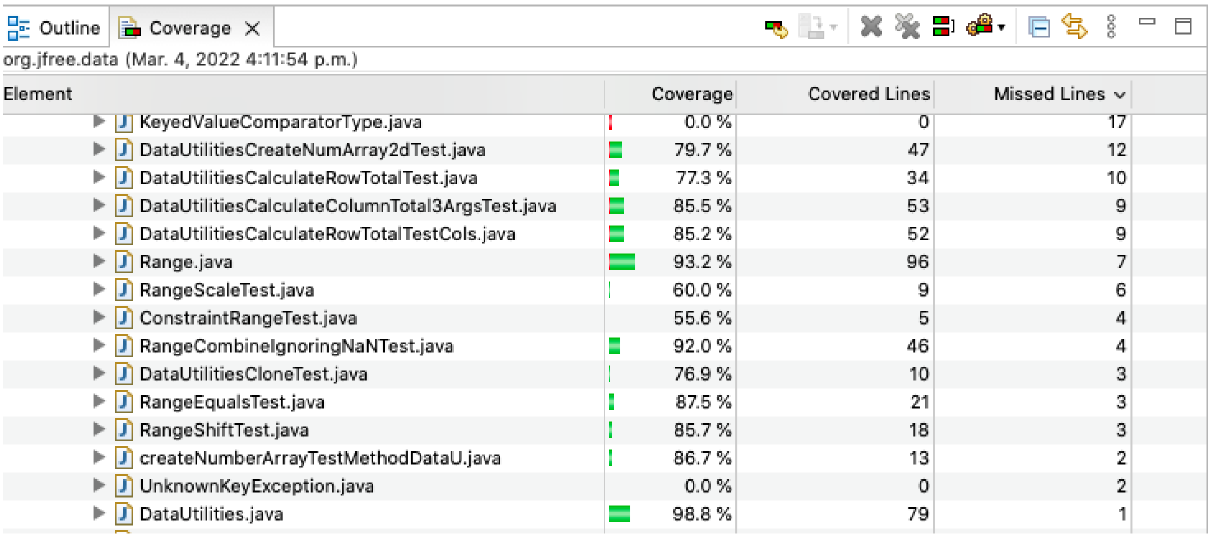
Method Counter ~ 90.0%



org.jfree.data (Mar. 4, 2022 4:11:54 p.m.)

Element	Coverage	Covered Methods	Missed Methods
KeyedValueComparator.java	0.0 %	0	4
DataUtilitiesCalculatedRowTotal.java	75.0 %	6	2
DataUtilities.java	90.0 %	9	1

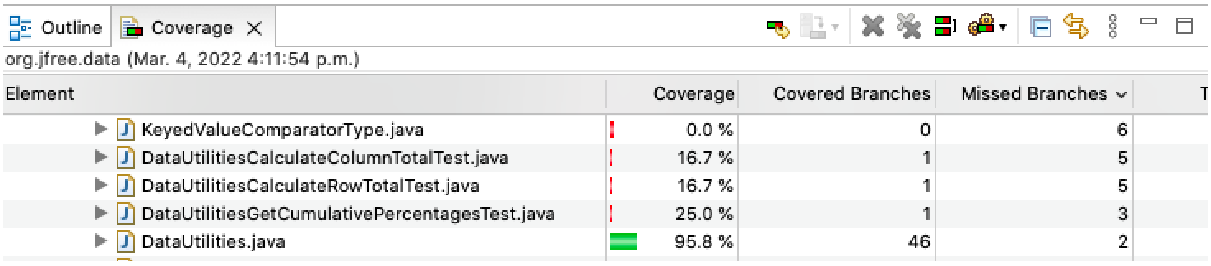
Line Counter ~ 98.8%



org.jfree.data (Mar. 4, 2022 4:11:54 p.m.)

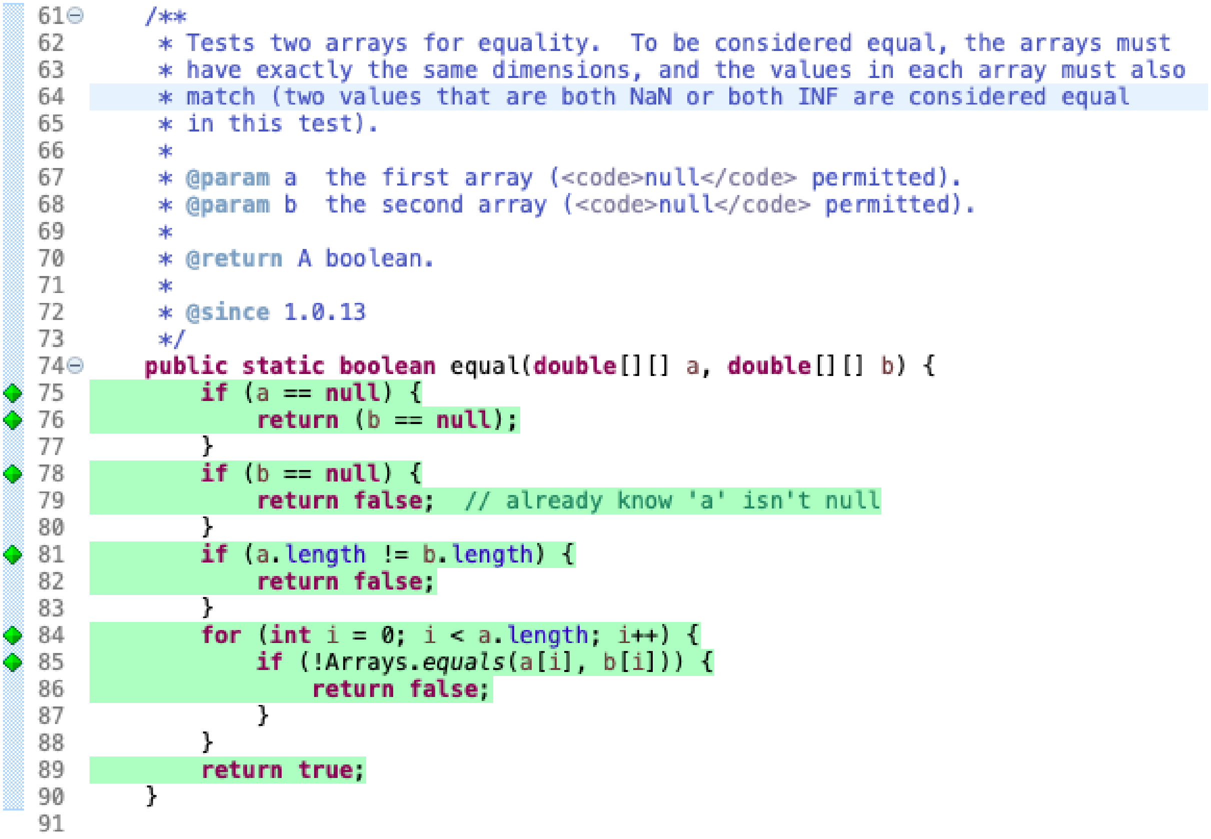
Element	Coverage	Covered Lines	Missed Lines
KeyedValueComparatorType.java	0.0 %	0	17
DataUtilitiesCreateNumArray2dTest.java	79.7 %	47	12
DataUtilitiesCalculateRowTotalTest.java	77.3 %	34	10
DataUtilitiesCalculateColumnTotal3ArgsTest.java	85.5 %	53	9
DataUtilitiesCalculateRowTotalTestCols.java	85.2 %	52	9
Range.java	93.2 %	96	7
RangeScaleTest.java	60.0 %	9	6
ConstraintRangeTest.java	55.6 %	5	4
RangeCombineIgnoringNaNTest.java	92.0 %	46	4
DataUtilitiesCloneTest.java	76.9 %	10	3
RangeEqualsTest.java	87.5 %	21	3
RangeShiftTest.java	85.7 %	18	3
createNumberArrayTestMethodDataU.java	86.7 %	13	2
UnknownKeyException.java	0.0 %	0	2
DataUtilities.java	98.8 %	79	1

Branch Counter ~ 95.8%

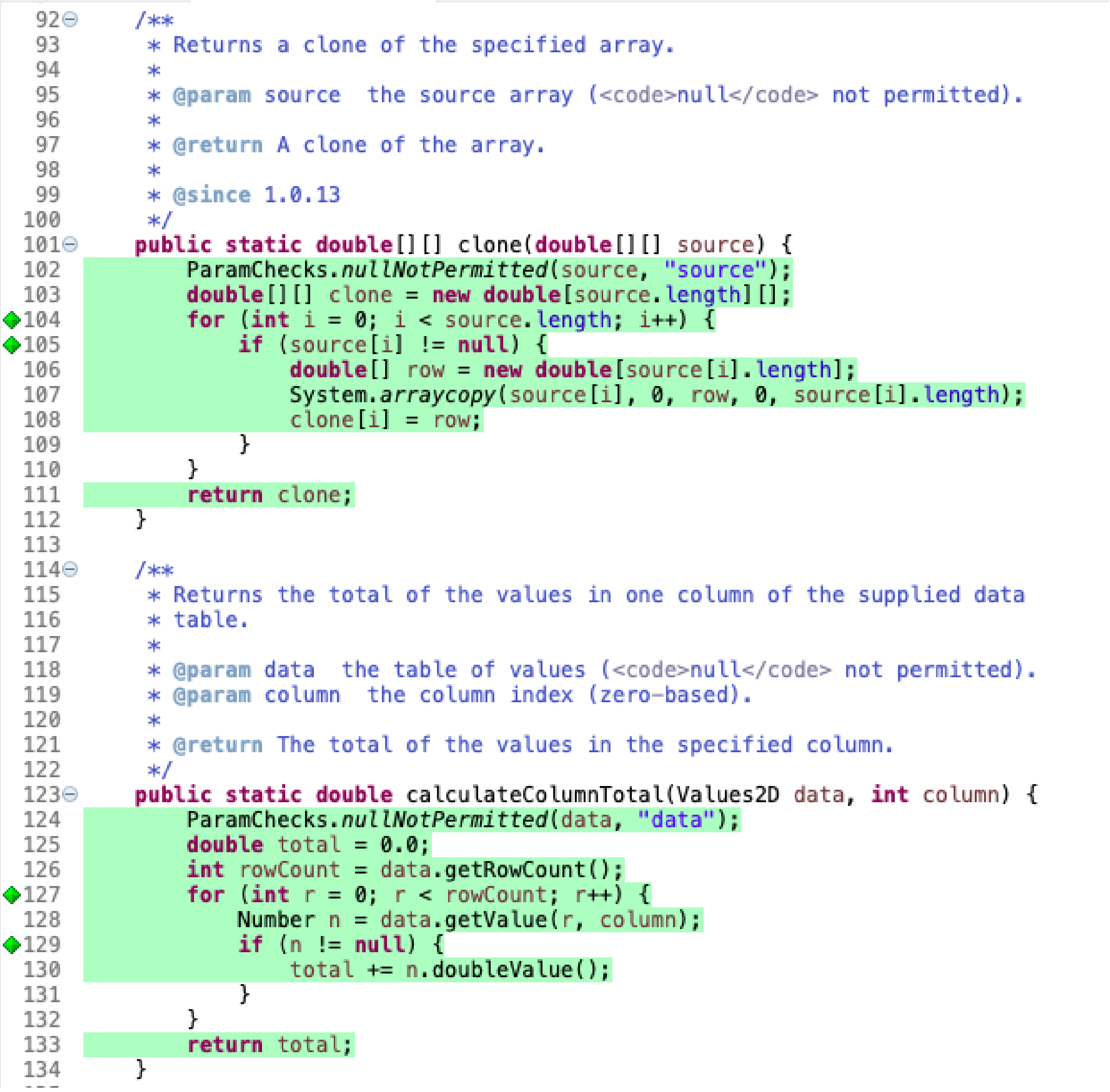


org.jfree.data (Mar. 4, 2022 4:11:54 p.m.)

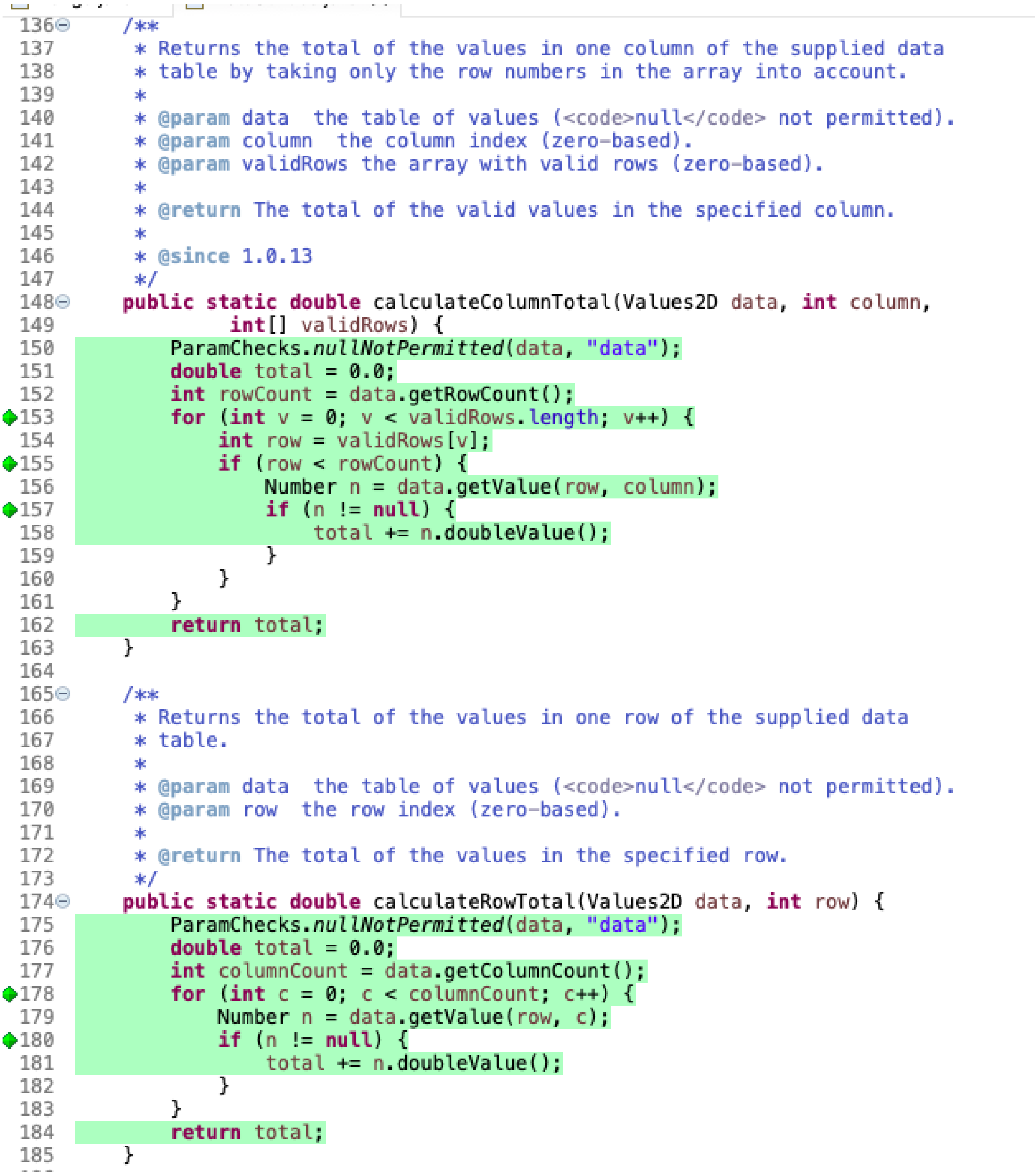
Element	Coverage	Covered Branches	Missed Branches	T
KeyedValueComparatorType.java	0.0 %	0	6	
DataUtilitiesCalculateColumnTotalTest.java	16.7 %	1	5	
DataUtilitiesCalculateRowTotalTest.java	16.7 %	1	5	
DataUtilitiesGetCumulativePercentagesTest.java	25.0 %	1	3	
DataUtilities.java	95.8 %	46	2	



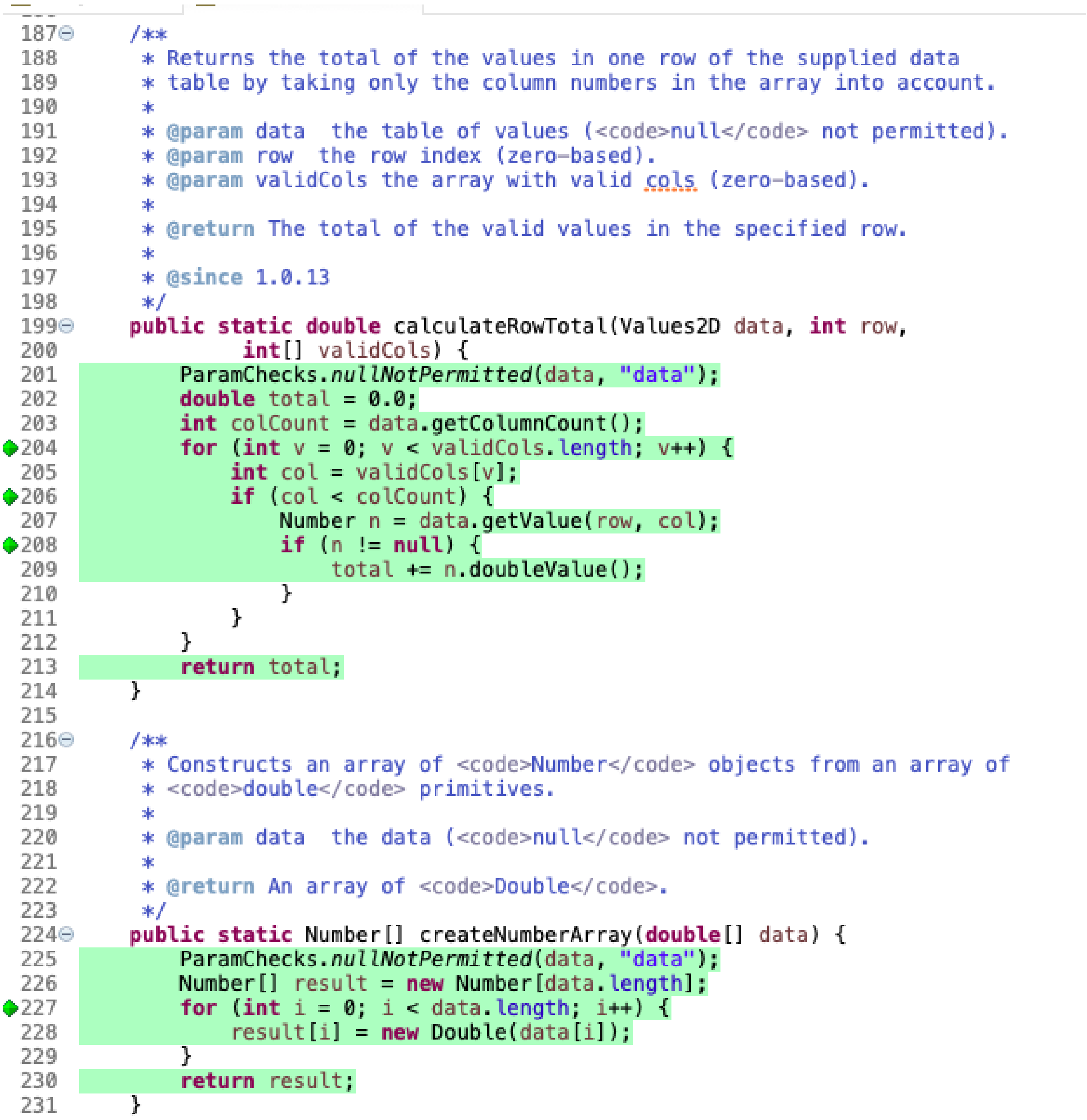
```
    /**
     * Tests two arrays for equality.  To be considered equal, the arrays must
     * have exactly the same dimensions, and the values in each array must also
     * match (two values that are both NaN or both INF are considered equal
     * in this test).
     *
     * @param a  the first array (<code>null</code> permitted).
     * @param b  the second array (<code>null</code> permitted).
     *
     * @return A boolean.
     *
     * @since 1.0.13
     */
    public static boolean equal(double[][] a, double[][] b) {
        if (a == null) {
            return (b == null);
        }
        if (b == null) {
            return false;  // already know 'a' isn't null
        }
        if (a.length != b.length) {
            return false;
        }
        for (int i = 0; i < a.length; i++) {
            if (!Arrays.equals(a[i], b[i])) {
                return false;
            }
        }
        return true;
    }
```



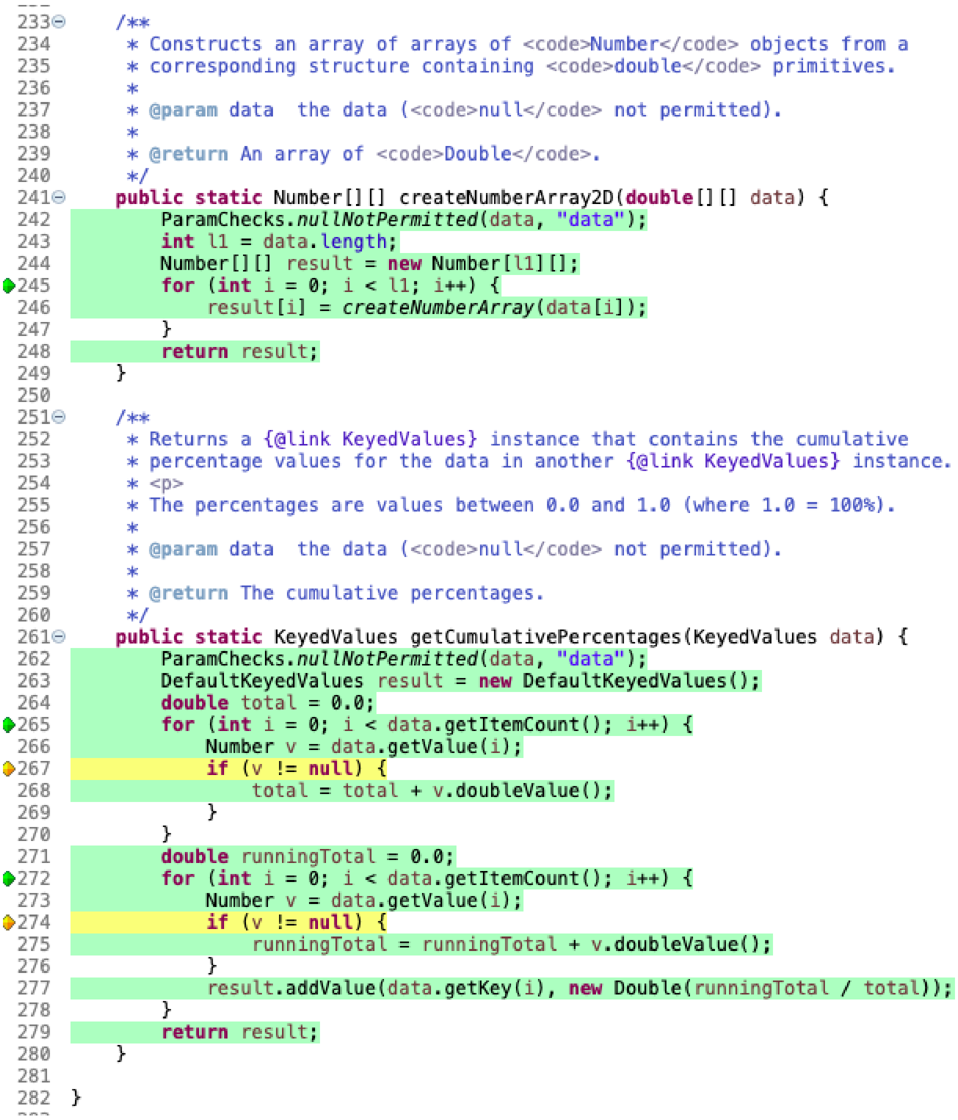
```java
    /**
     * Returns a clone of the specified array.
     *
     * @param source  the source array (<code>null</code> not permitted).
     *
     * @return A clone of the array.
     *
     * @since 1.0.13
     */
    public static double[][] clone(double[][] source) {
        ParamChecks.nullNotPermitted(source, "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;
    }

    /**
     * Returns the total of the values in one column of the supplied data
     * table.
     *
     * @param data  the table of values (<code>null</code> not permitted).
     * @param column  the column index (zero-based).
     *
     * @return The total of the values in the specified column.
     */
    public static double calculateColumnTotal(Values2D data, int column) {
        ParamChecks.nullNotPermitted(data, "data");
        double total = 0.0;
        int rowCount = data.getRowCount();
        for (int r = 0; r < rowCount; r++) {
            Number n = data.getValue(r, column);
            if (n != null) {
                total += n.doubleValue();
            }
        }
        return total;
    }
```



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

    /**
     * Returns the total of the values in one row of the supplied data
     * table.
     *
     * @param data  the table of values (<code>null</code> not permitted).
     * @param row  the row index (zero-based).
     *
     * @return The total of the values in the specified row.
     */
    public static double calculateRowTotal(Values2D data, int row) {
        ParamChecks.nullNotPermitted(data, "data");
        double total = 0.0;
        int columnCount = data.getColumnCount();
        for (int c = 0; c < columnCount; c++) {
            Number n = data.getValue(row, c);
            if (n != null) {
                total += n.doubleValue();
            }
        }
        return total;
    }
```



```java
    /**
     * 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;
    }

    /**
     * Constructs an array of <code>Number</code> objects from an array of
     * <code>double</code> primitives.
     *
     * @param data  the data (<code>null</code> not permitted).
     *
     * @return An array of <code>Double</code>.
     */
    public static Number[] createNumberArray(double[] data) {
        ParamChecks.nullNotPermitted(data, "data");
        Number[] result = new Number[data.length];
        for (int i = 0; i < data.length; i++) {
            result[i] = new Double(data[i]);
        }
        return result;
    }
```



```java
    /**
     * Constructs an array of arrays of <code>Number</code> objects from a
     * corresponding structure containing <code>double</code> primitives.
     *
     * @param data  the data (<code>null</code> not permitted).
     *
     * @return An array of <code>Double</code>.
     */
    public static Number[][] createNumberArray2D(double[][] data) {
        ParamChecks.nullNotPermitted(data, "data");
        int l1 = data.length;
        Number[][] result = new Number[l1][];
        for (int i = 0; i < l1; i++) {
            result[i] = createNumberArray(data[i]);
        }
        return result;
    }

    /**
     * Returns a {@link KeyedValues} instance that contains the cumulative
     * percentage values for the data in another {@link KeyedValues} instance.
     * <p>
     * The percentages are values between 0.0 and 1.0 (where 1.0 = 100%).
     *
     * @param data  the data (<code>null</code> not permitted).
     *
     * @return The cumulative percentages.
     */
    public static KeyedValues getCumulativePercentages(KeyedValues data) {
        ParamChecks.nullNotPermitted(data, "data");
        DefaultKeyedValues result = new DefaultKeyedValues();
        double total = 0.0;
        for (int i = 0; i < data.getItemCount(); i++) {
            Number v = data.getValue(i);
            if (v != null) {
                total = total + v.doubleValue();
            }
        }
        double runningTotal = 0.0;
        for (int i = 0; i < data.getItemCount(); i++) {
            Number v = data.getValue(i);
            if (v != null) {
                runningTotal = runningTotal + v.doubleValue();
            }
            result.addValue(data.getKey(i), new Double(runningTotal / total));
        }
        return result;
    }

}
```