SENG 438 - Software Testing, Reliability, and Quality

Lab. Report #3 – Code Coverage, Adequacy Criteria and Test Case Correlation

Group #: 2
Student Names:
Daniel Picazo
Oliver Molina
Quentin Jennings
Nouruddine Jaffan

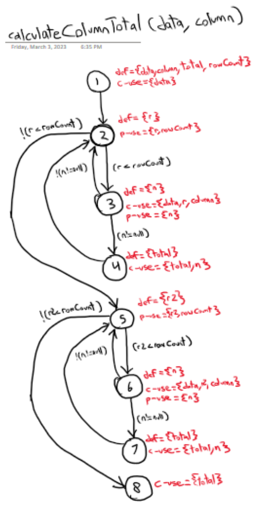
1 Introduction

Previously in Assignment 2 we were tasked with performing black-box testing on a modified JFreeChart system. While this allowed us to gain a level of insight white-box testing is required in order to dive deeper and become certain that our tests have covered all parts of the program. White-box testing is when you test based on an understanding of the internal workings of the code, which gives a much more complete image when combined with black-box testing. In this assignment, we were tasked to create unit tests based on what the source code of the method/class is based on fulfilling coverage criterias. In this assignment we investigate both control flow coverage and data flow coverage. Control flow coverage measures how much of the program's code and logic is covered by the program and uses metrics such as statement (line) coverage and branch coverage, while data flow coverage measures how much of the program's variable definitions and uses are covered and uses Def-Use (DU) coverage as a metric. In this lab we were tasked to add on to our previous unit test and verify its integrity using the previously discussed metrics and following in this report will be our methodology and results.

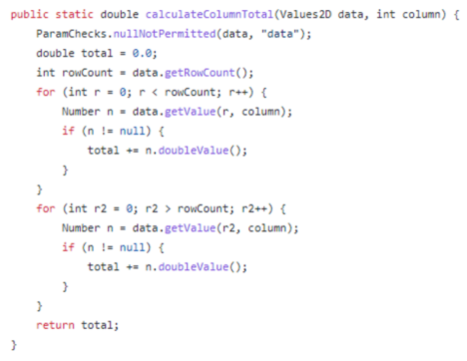
2 Manual data-flow coverage calculations for DataUtilities.CalculateColumnTotal() and Range.Contains() methods

For the two methods we were tasked with creating its data flow graph and def-use sets per statement given the source code. Once completed we were able to list the DU-pairs per variable and compare each test case to see which pairs were covered and which weren't. Using that we were able to calculate the DU-pair coverage.

Below is the manual data-flow coverage for Data Utilities CalculateColumnTotal method, split into two overloaded methods:



Def-use per statement:



def = {data, column}	123	`public static double calculateColumnTotal(Values2D data, int column) {`
c-use = {data}	124	`    ParamChecks.nullNotPermitted(data, "data");`
def = {total}	125	`    double total = 0.0;`
def = {rowCount} c-use = {data}	126	`    int rowCount = data.getRowCount();`
def = {r} p-use = {r, rowCount} c-use = {r}	127	`    for (int r = 0; r < rowCount; r++) {`
def = {n} c-use = {data, r, column}	128	`        Number n = data.getValue(r, column);`
p-use = {n}	129	`        if (n != null) {`
def = {total} c-use = {total, n}	130	`            total += n.doubleValue();`
	131	`        }`
	132	`    }`
def = {r2} p-use = {r2, rowCount} c-use = {r2}	133	`    for (int r2 = 0; r2 > rowCount; r2++) {`
def = {n} c-use = {data, r2, column}	134	`        Number n = data.getValue(r2, column);`
p-use = {n}	135	`        if (n != null) {`
def = {total} c-use = {total, n}	136	`            total += n.doubleValue();`
	137	`        }`
	138	`    }`
c-use = {total}	139	`    return total;`
	140	`}`

Def-use pairs per variable:

data: {(1, 1), (1, 3), (1, 6)}
column: {(1, 3), (1, 6)}
total: {(1, 4), (1, 7), (1, 8),
(4, 4), (4, 7), (4, 8),
(7, 7), (7, 8)}
rowCount: {(1, 2), (1, 5)}
r: {(3, 3), (3, 4)}
r2: {(5, 5), (5, 6)}
n: {(3, 3), (3, 4),
(6, 6), (6, 7)}

Test DU Pairs

calculateColumnTotalForTwoValues

data: {(1, 1), (1, 3)}
column: {(1, 3)}
total: {(1, 4), (4, 4), (4, 8)}
rowCount: {(1, 2), (1, 5)}
r: {(3, 3), (3, 4)}
n: {(3, 3), (3, 4)}

calculateColumnTotalForInvalidColumn

data: {(1, 1), (1, 3)}
column: {(1, 3)}
total: {(1, 4), (1, 8)}
rowCount: {(1, 2), (1, 5)}
r: {(3, 3)}
n: {(3, 3)}

calculateColumnTotalForNegativeColumn

data: {(1, 1), (1, 3)}
column: {(1, 3)}
total: {(1, 4), (1, 8)}
rowCount: {(1, 2), (1, 5)}
r: {(3, 3)}
n: {(3, 3)}

calculateColumnTotalForNullTable

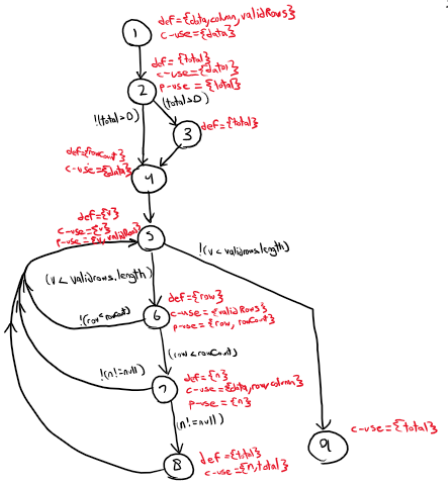
data: {1, 1}

DU Coverage

$\text{Coverage} = \frac{\text{covered}}{\text{total}} = \frac{13}{23} \cdot 100\% = \underline{57\%}$

calculateColumnTotal(data, column, validRows)

Friday, March 3, 2023 2:43 PM



Def-Use per Statement:

```
def={data,column,validRows}                 154  public static double calculateColumnTotal(Values2D data, int column,
                                            155          int[] validRows) {
c-use={data}                                156      ParamChecks.nullNotPermitted(data, "data");
def={total}                                 157      double total = 0.0;
p-use={total}                               158      if (total > 0){
def={total}                                 159          total = 100;
                                            160      }
                                            161
def={rowCount} c-use-{data}                 162      int rowCount = data.getRowCount();
def={v} c-use={v,validRows}                 163      for (int v = 0; v < validRows.length; v++) {
def={row} c-use={validRows}                 164          int row = validRows[v];
p-use={row,rowCount}                        165          if (row < rowCount) {
def={n} c-use={data,row,column}             166              Number n = data.getValue(row, column);
p-use={n}                                   167              if (n != null) {
def={total} c-use={total,n}                 168                  total += n.doubleValue();
                                            169              }
                                            170          }
                                            171      }
c-use={total}                               172      return total;
                                            173  }
```

Def-Use Pairs per variable:

data: {(1,1), (1,2), (1,4), (1,7)}

column: {(1,7)}

validRows: {(1,5), (1,6)}

total: {(2,2), (2,8), (2,9), (3,8), (3,9), (8,8), (8,9)}

rowCount: {(4,6)}

v: {(5,5)}

row: {(6,6), (6,7)}

n: {(7,7), (7,8)}

Test DU Pairs

calcColumnTotalValidRows

highlighted in yellow

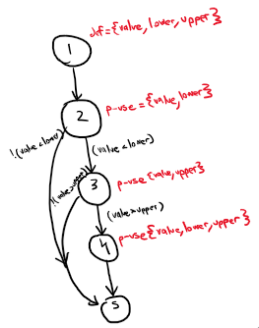
DU coverage

$$\text{Coverage} = \frac{\text{covered}}{\text{total}} = \frac{18}{20} \cdot 100\% = \underline{90\%}$$

Below is the manual data-flow coverage for Range contains method:

contains()

Friday, March 3, 2023 4:27 PM



Def-use pairs per variable:

value: {(1,2),(1,3),(1,4)}

lower: {(1,2),(1,4)}

upper: {(1,3),(1,4)}

Def-use per statement:

def={value, lower, upper}

p-use={value, lower}

p-use={value, upper}

p-use={value, lower, upper}

```
public boolean contains(double value) {
    if (value < this.lower) {
        return false;
    }
    if (value  > this.upper) {
        return false;
    }
    return (value >= this.lower && value <= this.upper);
}
```

Test Name — (value, lower, upper)

ContainsReturnsTrue() — (2, 1, 10)
- value: (1,2), (1,3), (1,4)
- lower: (1,2), (1,4)
- upper: (1,3), (1,4)

ContainsReturnsFalse() — (-5, 1, 10)
- value: (1,2)
- lower: (1,2)

ContainsLowerBound() — (1, 1, 10)
- value: (1,2), (1,3), (1,4)
- lower: (1,2), (1,4)
- upper: (1,3), (1,4)

ContainsUpperBound() — (10, 1, 10)
- value: (1,2), (1,3), (1,4)
- lower: (1,2), (1,4)
- upper: (1,3), (1,4)

DU Coverage

$\text{Coverage} = \frac{\text{covered}}{\text{total}} = \frac{7}{7} \cdot 100\% = 100\%$

-DU-pair coverage = 100% since all DU pairs are evidently covered!

3 A detailed description of the testing strategy for the new unit test

Now that we can look at the internal code we can see what parts of the code we missed and deduce a testing strategy aimed at maximizing test coverage. In specific, we were required to have 90% statement coverage, 70% branch coverage and 60% condition coverage by adding to our test suite. Using the data flow charts and statement coverages as a guideline we iteratively created tests, specifically designing them to cover uncovered blocks of code. The built-in coverage tool within JUnit was incredibly useful for this purpose as it made our workflow easy and efficient.

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

CHOOSE 5 TEST CASES:

RangeTest - RangeConstructorForInvalidRange():

- In this, we are testing to see whether or not the Range constructor outputs the proper exception when the upper bound argument is less than the lower bound argument. To test this condition we send in arguments that are flipped and expect to see an IllegalArgumentException.

RangeTest - EqualsReturnsFalseForNonRange():

- The coverage metrics from Eclemma showed that we were not reaching the condition where we check if we deal with a non Range object. To reach this condition and increase our coverage we made a test case sending in a Double into the Range.equals() method and expect to get a false result.

RangeTest - EqualsReturnsFalseForNonRange():

- The equals() function from Range is meant to return a boolean true if two ranges are equal. When testing this function, we added the standard test cases such as identical ranges and different ranges; however, thanks to white-box testing and seeing that the function also covers for when a range is not an instance of Range object, it will return false. This allowed us to increase coverage by accounting for a branch we would have otherwise missed.

DataUtilitiesTest - cloneNullTest():

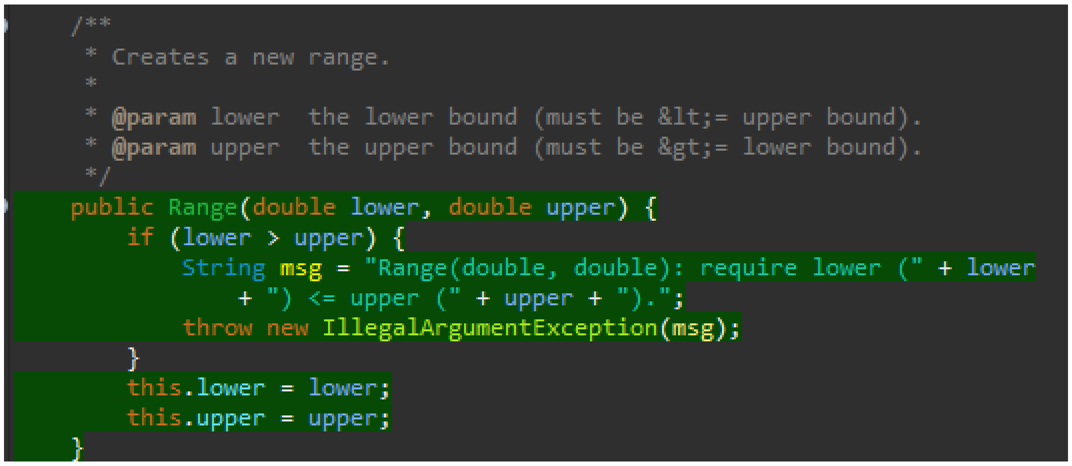
- In this test, we were testing the clone() command from DataUtilities. Normally, we would have made tests to test cloning arrays for true, false, and for nulls; however, after taking a look at the source code, we noticed it had branches for indices of the 2D array (that is, the arrays within the array) being null. Thus, we made a test where we tried to clone {{1,2,3},null,{7,8,9}} and it passed.

DataUtilitiesTest - negativeColCountForRowTotal():

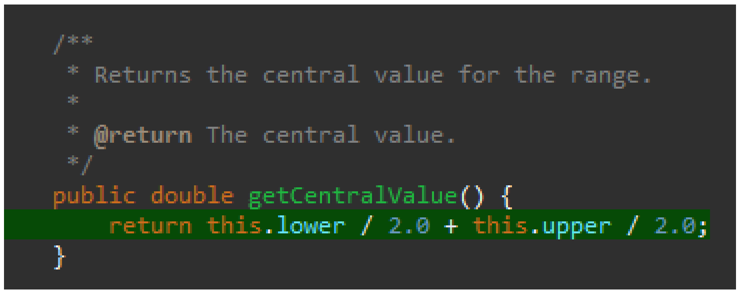
- If this were black-box testing, we would have never tested for the possibility of negative row or column counts, assuming it to be dead code. After taking a look at the function calculateRowTotal(Values2D data, int row, int[] validCols), we noticed that it did in fact account for negative column count, and would thus return 0 as the total.

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

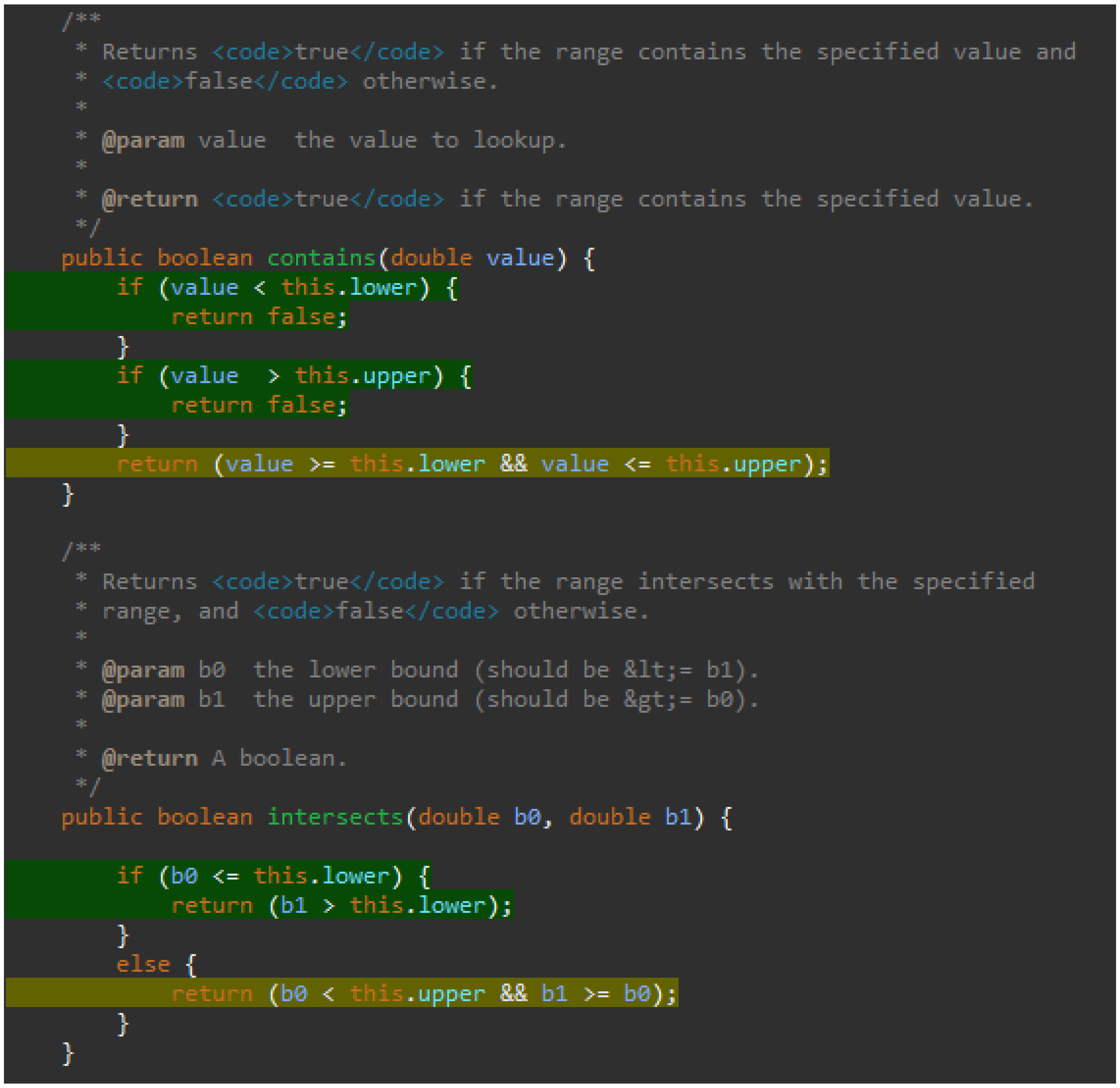
Range.java:



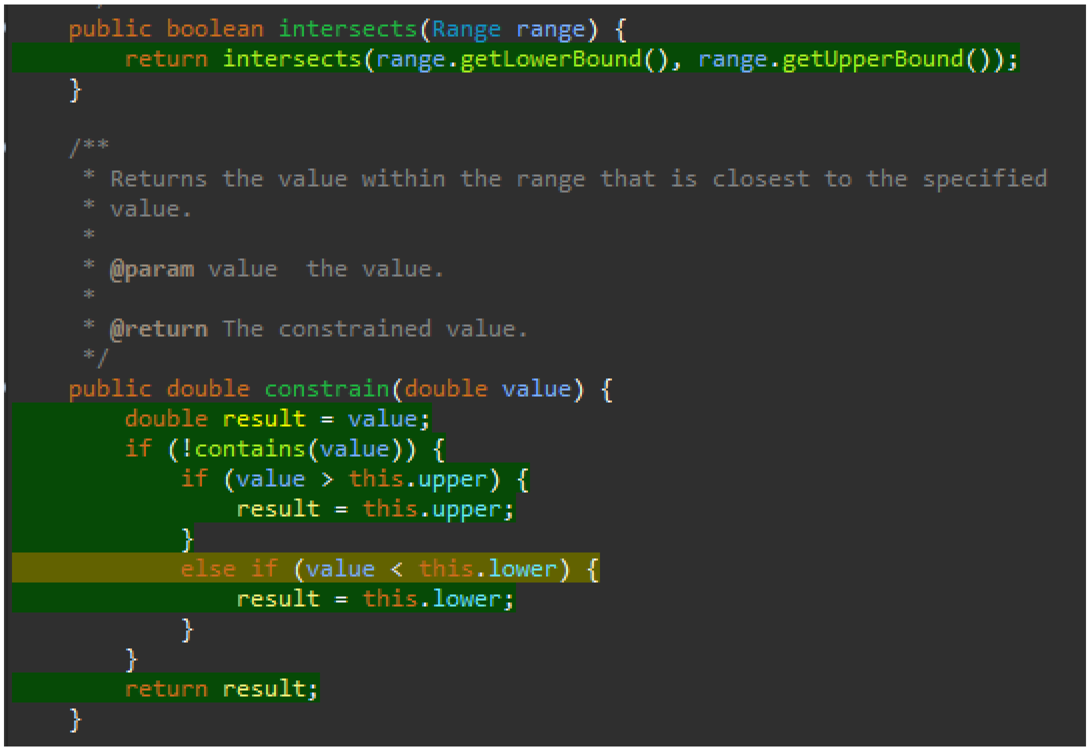
```
    /**
     * Creates a new range.
     *
     * @param lower  the lower bound (must be &lt;= upper bound).
     * @param upper  the upper bound (must be &gt;= lower bound).
     */
    public Range(double lower, double upper) {
        if (lower > upper) {
            String msg = "Range(double, double): require lower (" + lower
                + ") <= upper (" + upper + ").";
            throw new IllegalArgumentException(msg);
        }
        this.lower = lower;
        this.upper = upper;
    }
```



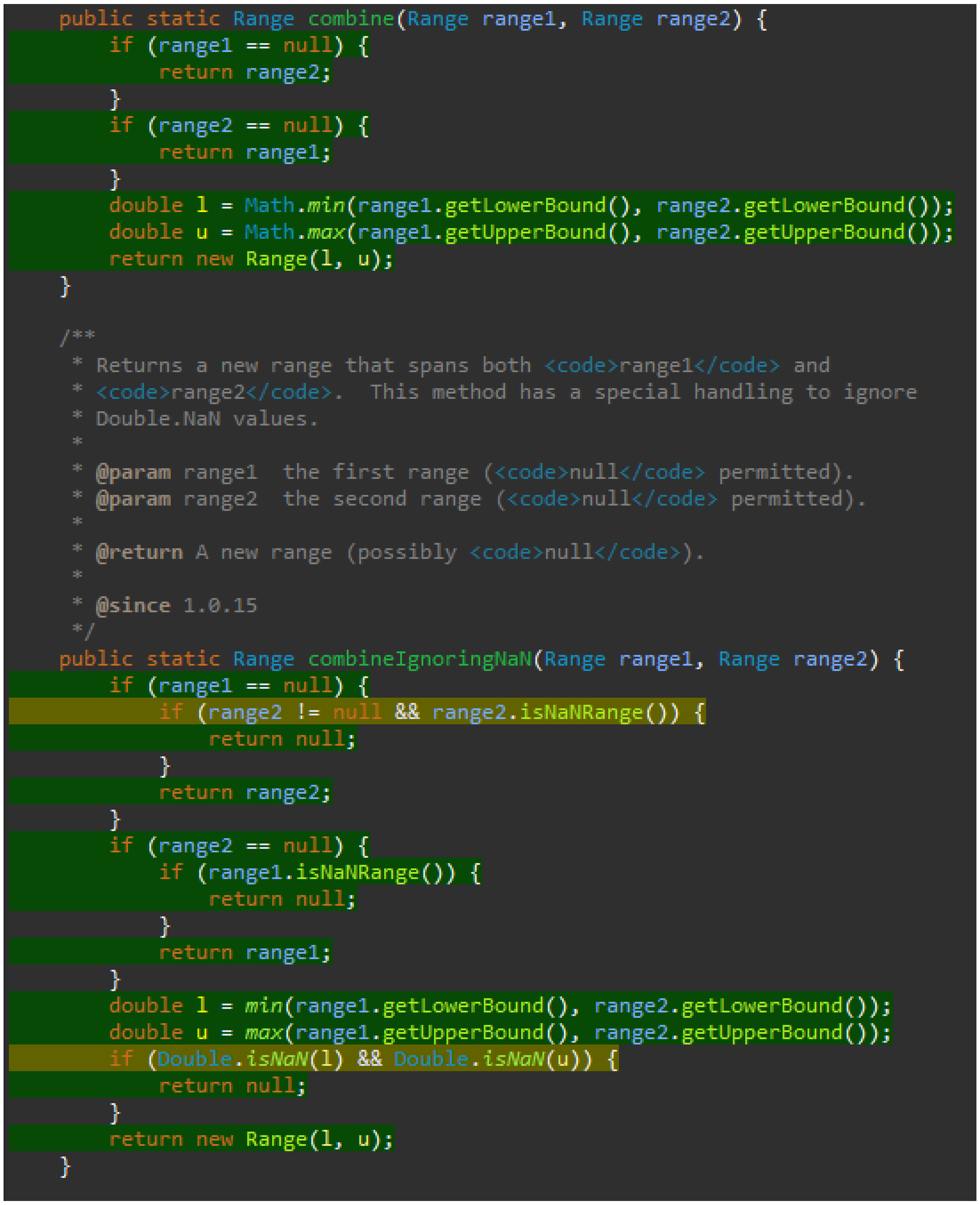
```
    /**
     * Returns the central value for the range.
     *
     * @return The central value.
     */
    public double getCentralValue() {
        return this.lower / 2.0 + this.upper / 2.0;
    }
```



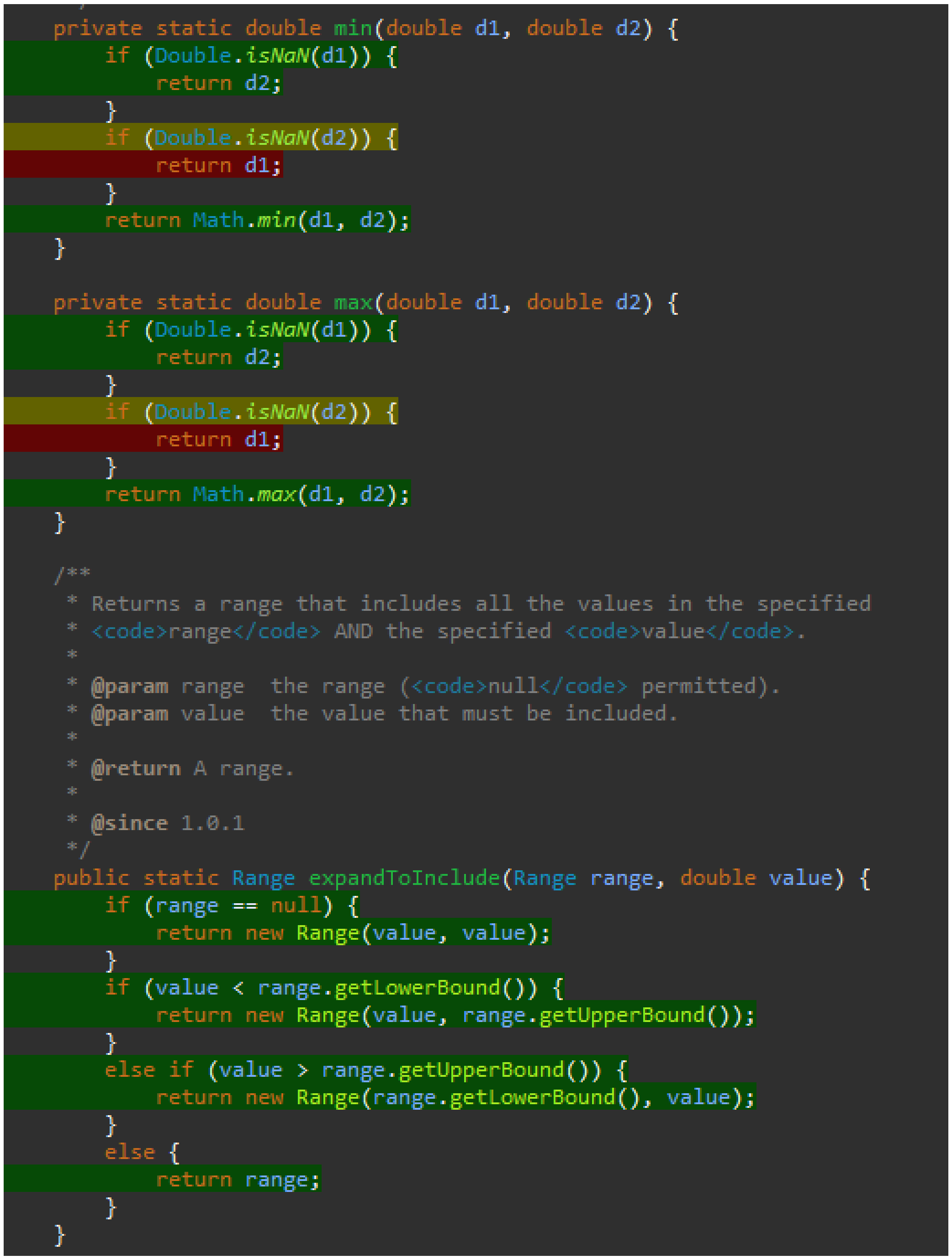
```
    /**
     * Returns <code>true</code> if the range contains the specified value and
     * <code>false</code> otherwise.
     *
     * @param value  the value to lookup.
     *
     * @return <code>true</code> if the range contains the specified value.
     */
    public boolean contains(double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



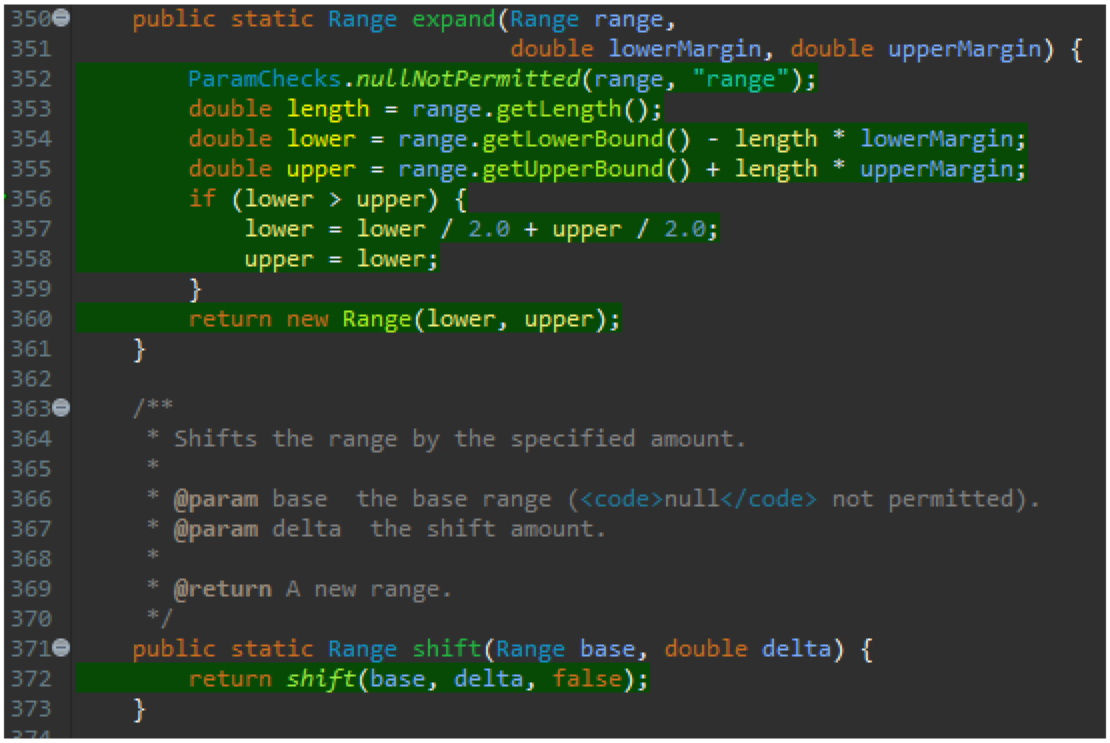
```
    public boolean intersects(Range range) {
        return intersects(range.getLowerBound(), range.getUpperBound());
    }

    /**
     * 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;
    }
```



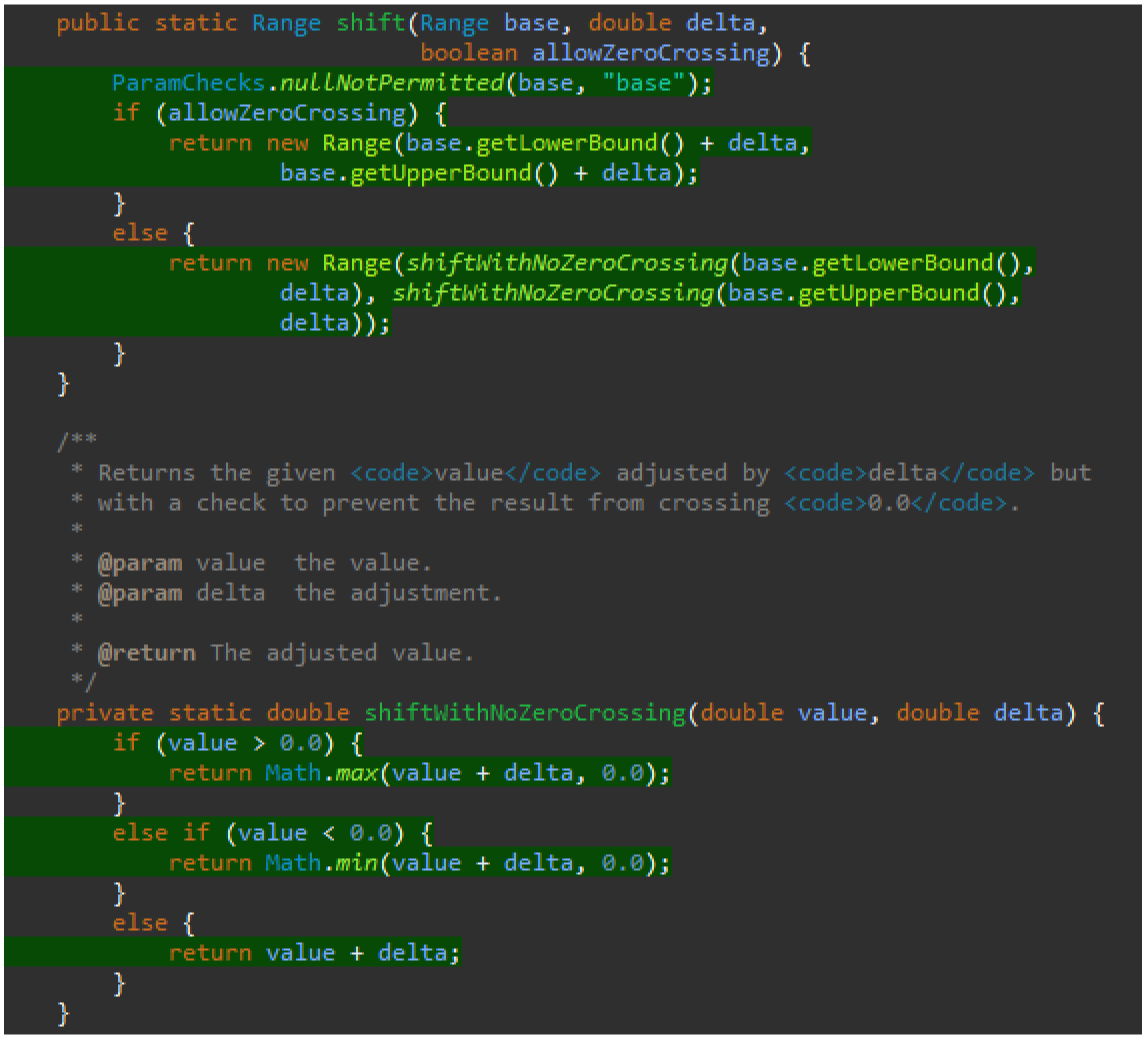
```
    public static Range combine(Range range1, Range range2) {
        if (range1 == null) {
            return range2;
        }
        if (range2 == null) {
            return range1;
        }
        double l = Math.min(range1.getLowerBound(), range2.getLowerBound());
        double u = Math.max(range1.getUpperBound(), 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);
    }
```



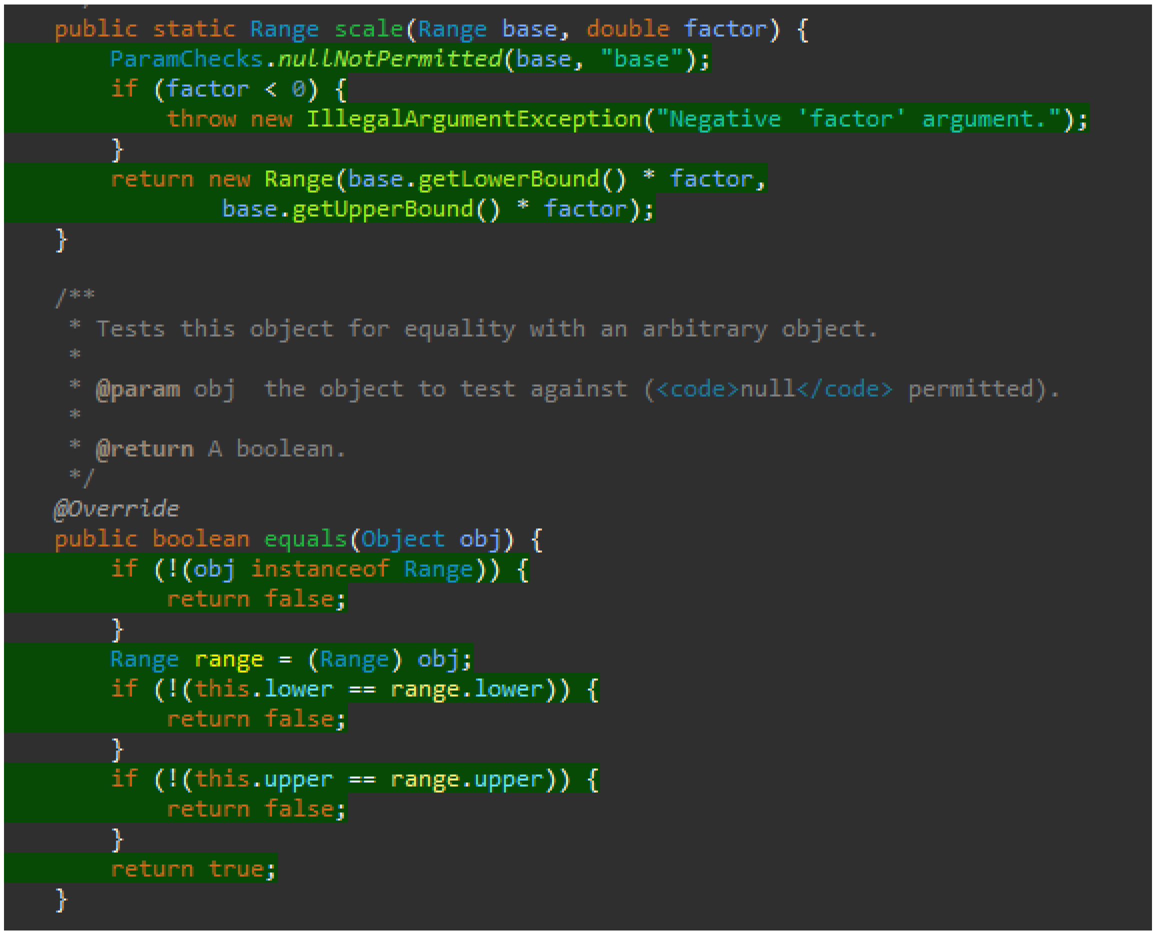
```
    private static double min(double d1, double d2) {
        if (Double.isNaN(d1)) {
            return d2;
        }
        if (Double.isNaN(d2)) {
            return d1;
        }
        return Math.min(d1, d2);
    }

    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;
        }
    }
```



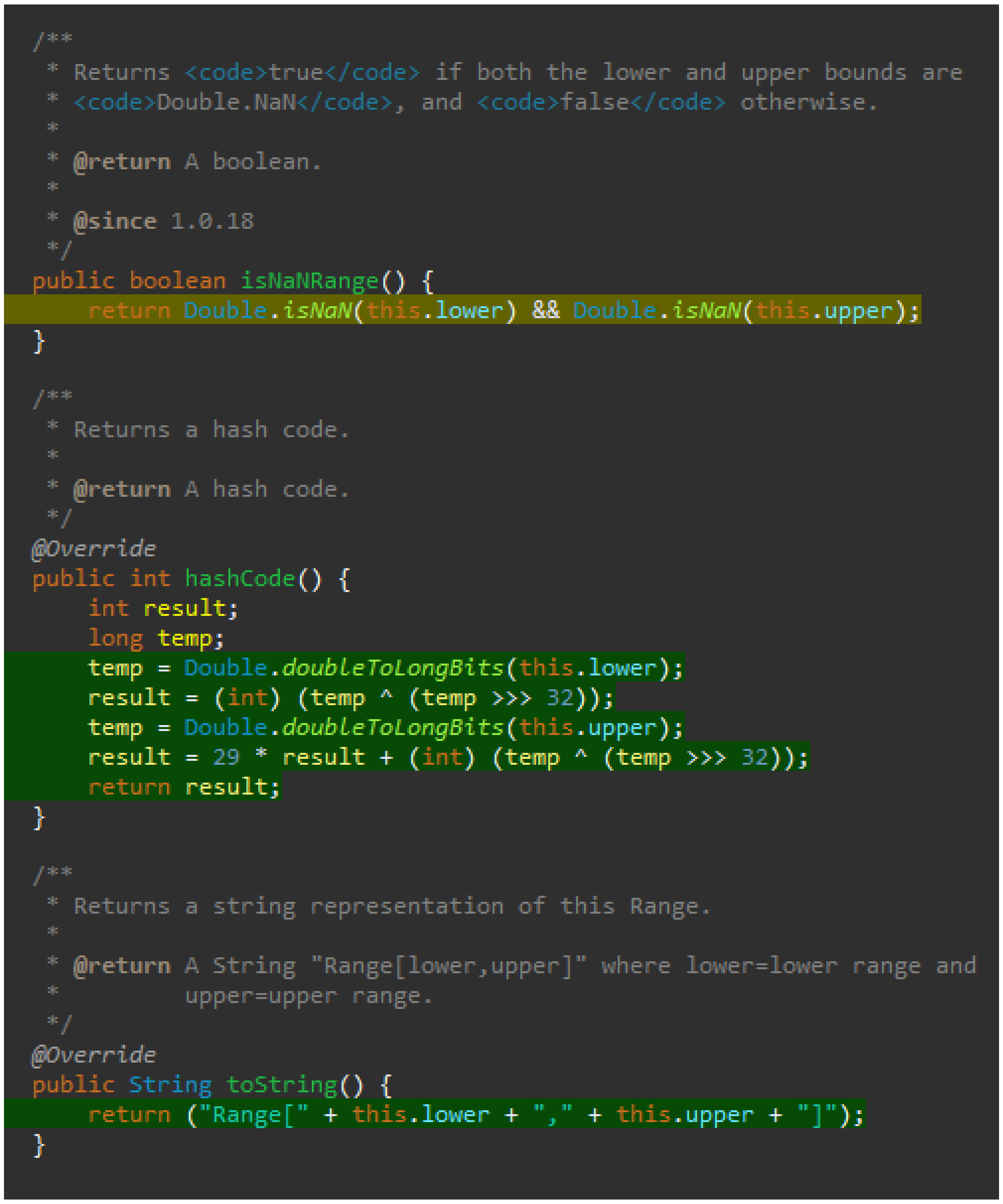
```
    public static Range expand(Range range,
                               double lowerMargin, double upperMargin) {
        ParamChecks.nullNotPermitted(range, "range");
        double length = range.getLength();
        double lower = range.getLowerBound() - length * lowerMargin;
        double 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);
    }
```



```
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));
    }
}

/**
 * Returns the given <code>value</code> adjusted by <code>delta</code> but
 * with a check to prevent the result from crossing <code>0.0</code>.
 *
 * @param value  the value.
 * @param delta  the adjustment.
 *
 * @return The adjusted value.
 */
private static double shiftWithNoZeroCrossing(double value, double delta) {
    if (value > 0.0) {
        return Math.max(value + delta, 0.0);
    }
    else if (value < 0.0) {
        return Math.min(value + delta, 0.0);
    }
    else {
        return value + delta;
    }
}
```

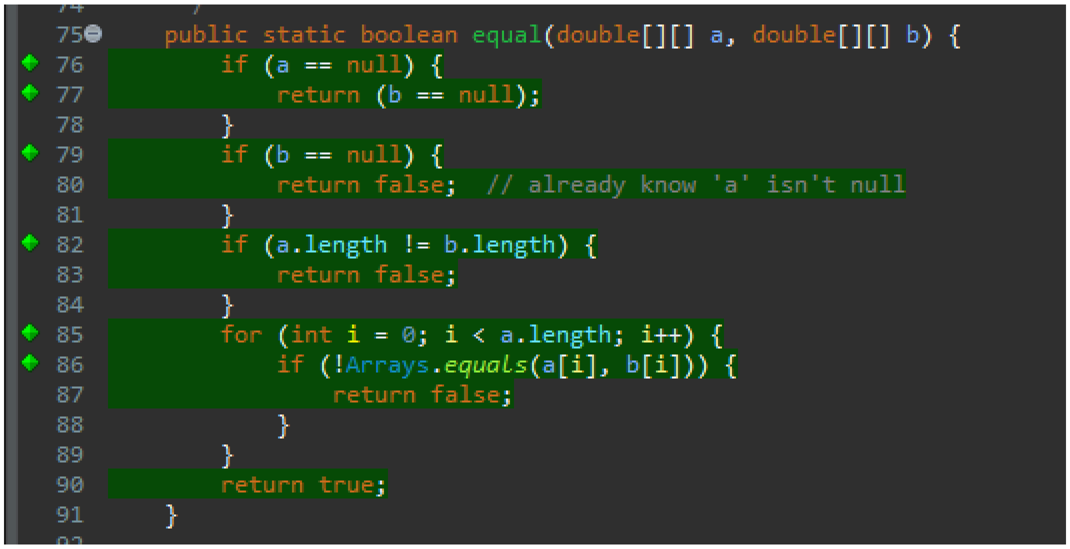


```
    public static Range scale(Range base, double factor) {
        ParamChecks.nullNotPermitted(base, "base");
        if (factor < 0) {
            throw new IllegalArgumentException("Negative 'factor' argument.");
        }
        return new Range(base.getLowerBound() * factor,
                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;
    }
```

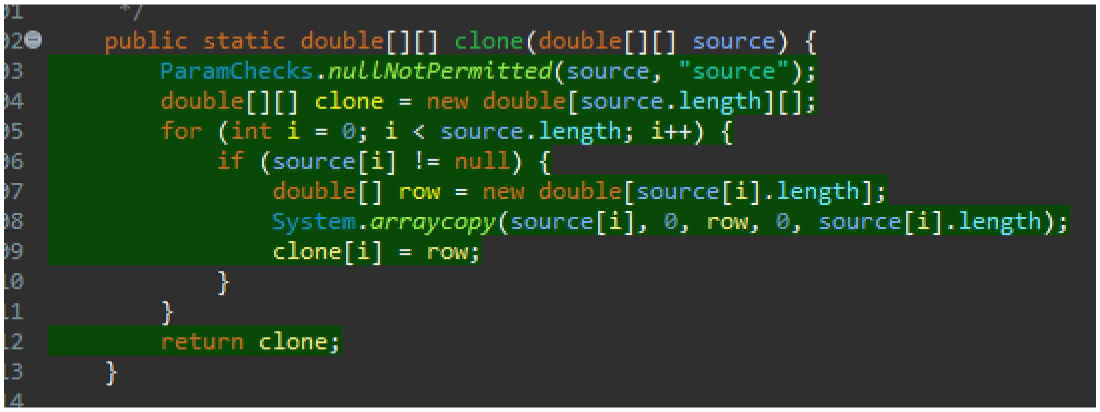


```
/**
 * Returns <code>true</code> if both the lower and upper bounds are
 * <code>Double.NaN</code>, and <code>false</code> otherwise.
 *
 * @return A boolean.
 *
 * @since 1.0.18
 */
public boolean isNaNRange() {
    return Double.isNaN(this.lower) && Double.isNaN(this.upper);
}

/**
 * Returns a hash code.
 *
 * @return A hash code.
 */
@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 + "]");
}
```

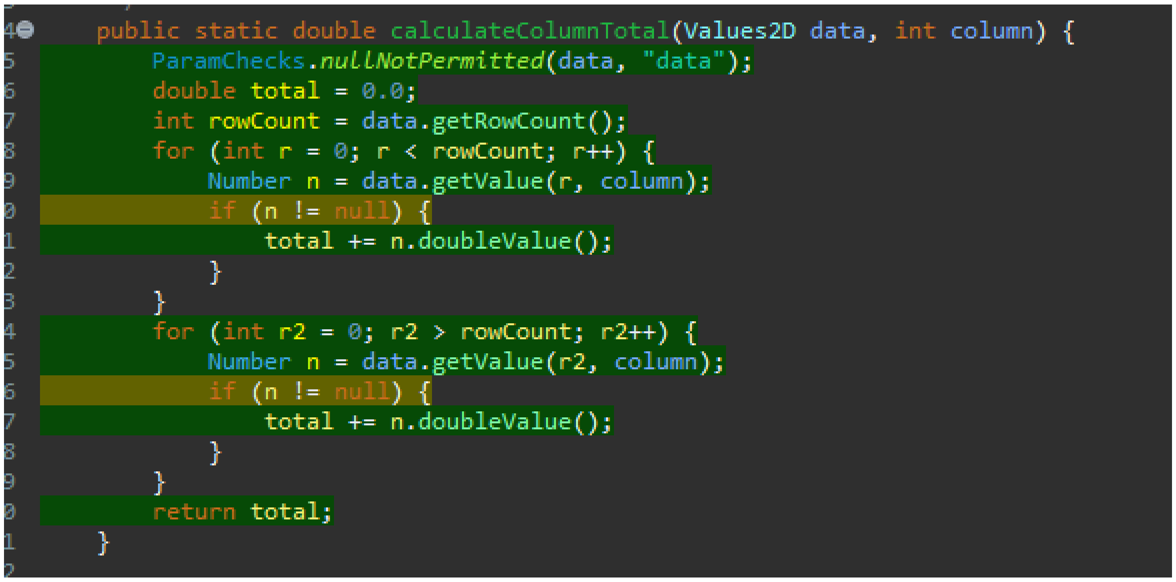
DataUtilties.java:



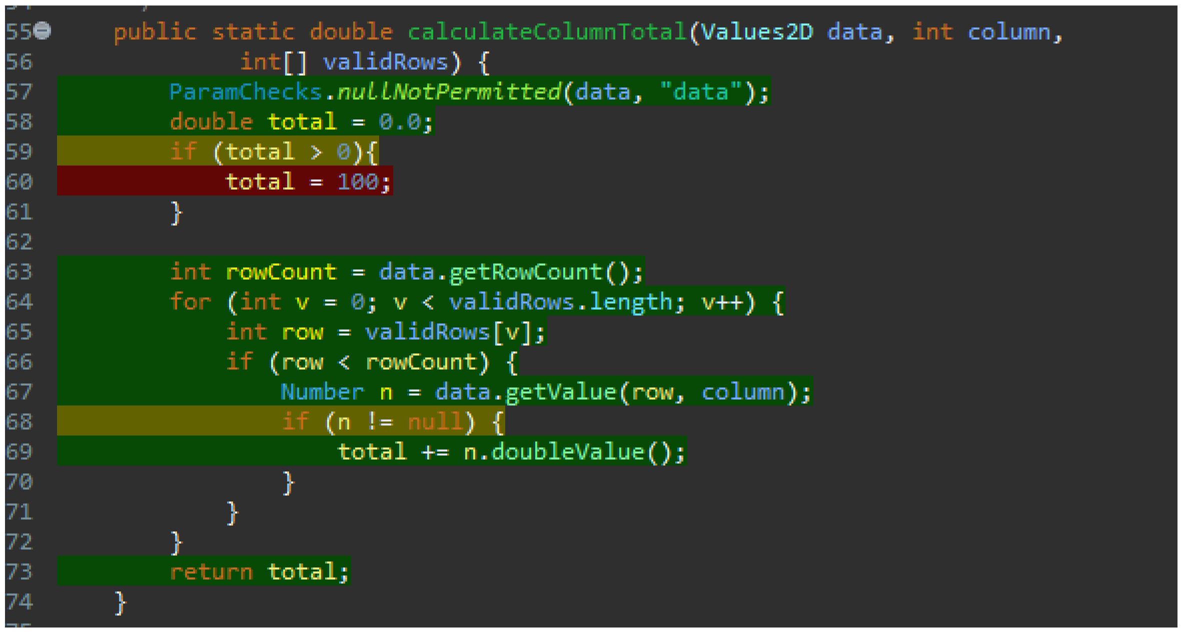
```java
    public static boolean equal(double[][] a, double[][] b) {
        if (a == null) {
            return (b == null);
        }
        if (b == null) {
            return false;  // already know 'a' isn't null
        }
        if (a.length != b.length) {
            return false;
        }
        for (int i = 0; i < a.length; i++) {
            if (!Arrays.equals(a[i], b[i])) {
                return false;
            }
        }
        return true;
    }
```



```java
    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;
    }
```

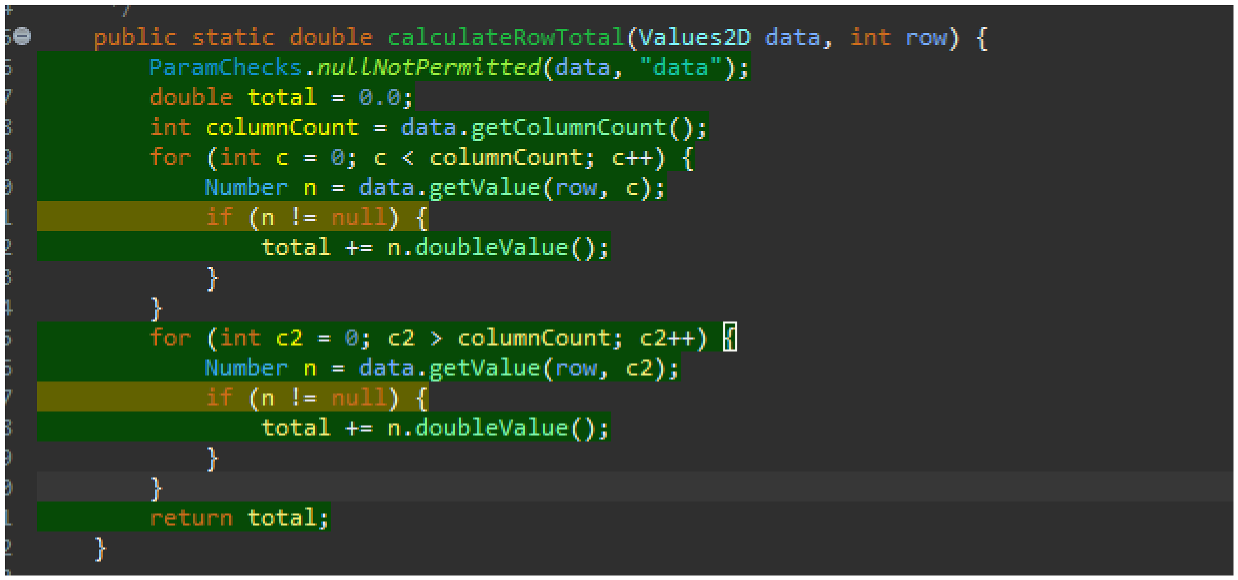


```java
    public static double calculateColumnTotal(Values2D data, int column) {
        ParamChecks.nullNotPermitted(data, "data");
        double total = 0.0;
        int rowCount = data.getRowCount();
        for (int r = 0; r < rowCount; r++) {
            Number n = data.getValue(r, column);
            if (n != null) {
                total += n.doubleValue();
            }
        }
        for (int r2 = 0; r2 > rowCount; r2++) {
            Number n = data.getValue(r2, column);
            if (n != null) {
                total += n.doubleValue();
            }
        }
        return total;
    }
```

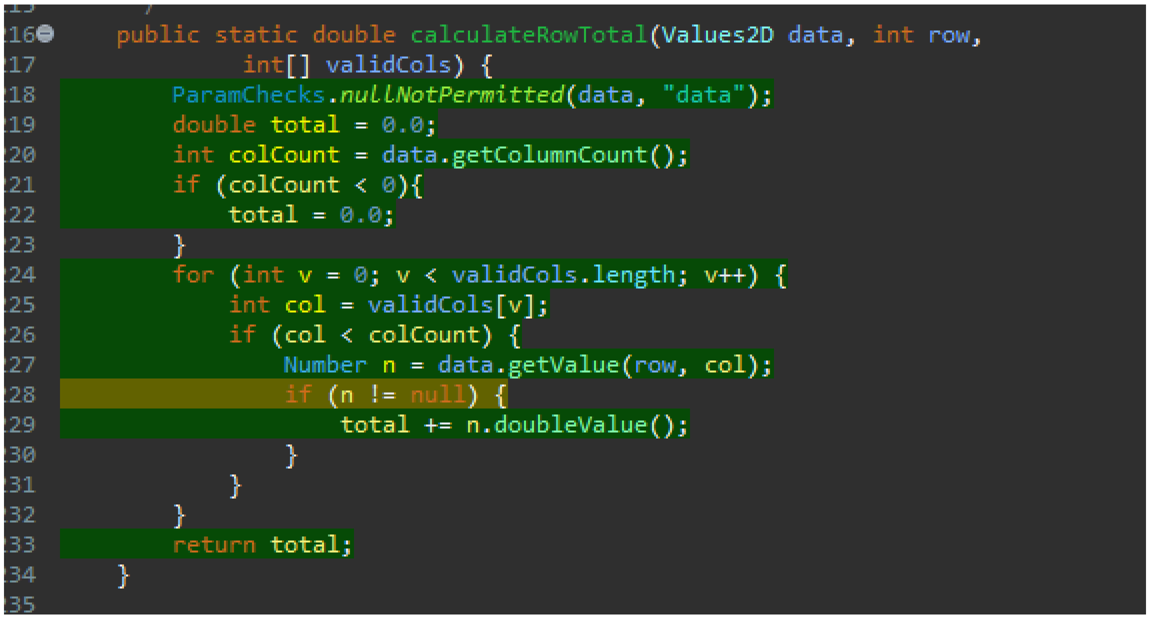


```
    public static double calculateColumnTotal(Values2D data, int column,
            int[] validRows) {
        ParamChecks.nullNotPermitted(data, "data");
        double total = 0.0;
        if (total > 0){
            total = 100;
        }

        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;
    }
```

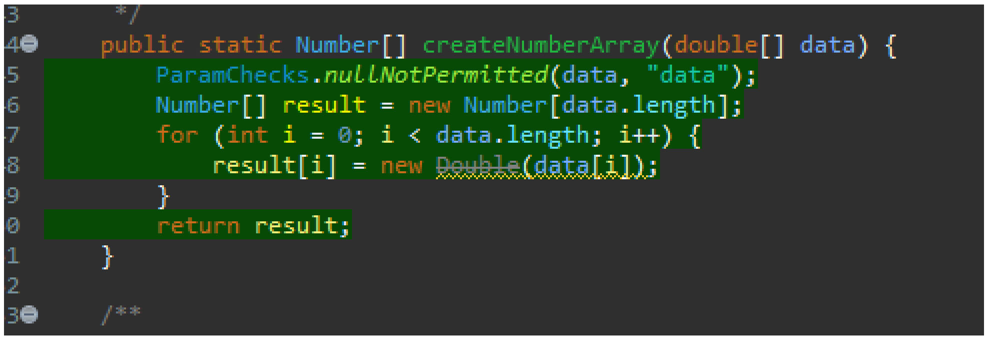
NOTE: This red line above is completely unreachable.



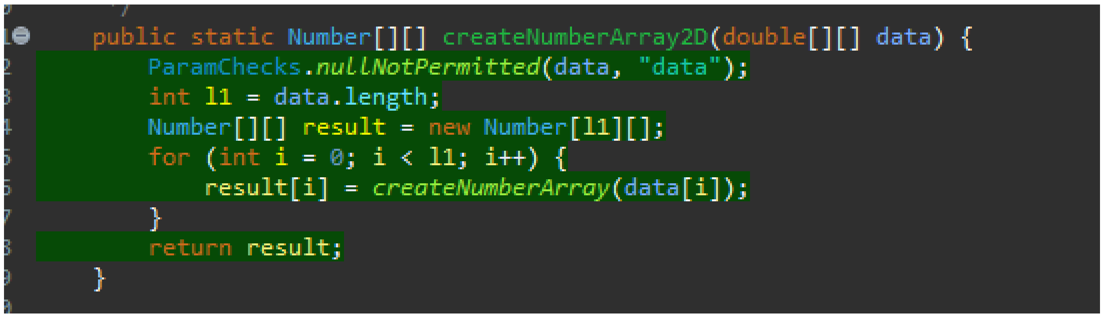
```
    public static double calculateRowTotal(Values2D data, int row) {
        ParamChecks.nullNotPermitted(data, "data");
        double total = 0.0;
        int columnCount = data.getColumnCount();
        for (int c = 0; c < columnCount; c++) {
            Number n = data.getValue(row, c);
            if (n != null) {
                total += n.doubleValue();
            }
        }
        for (int c2 = 0; c2 > columnCount; c2++) {
            Number n = data.getValue(row, c2);
            if (n != null) {
                total += n.doubleValue();
            }
        }
        return total;
    }
```



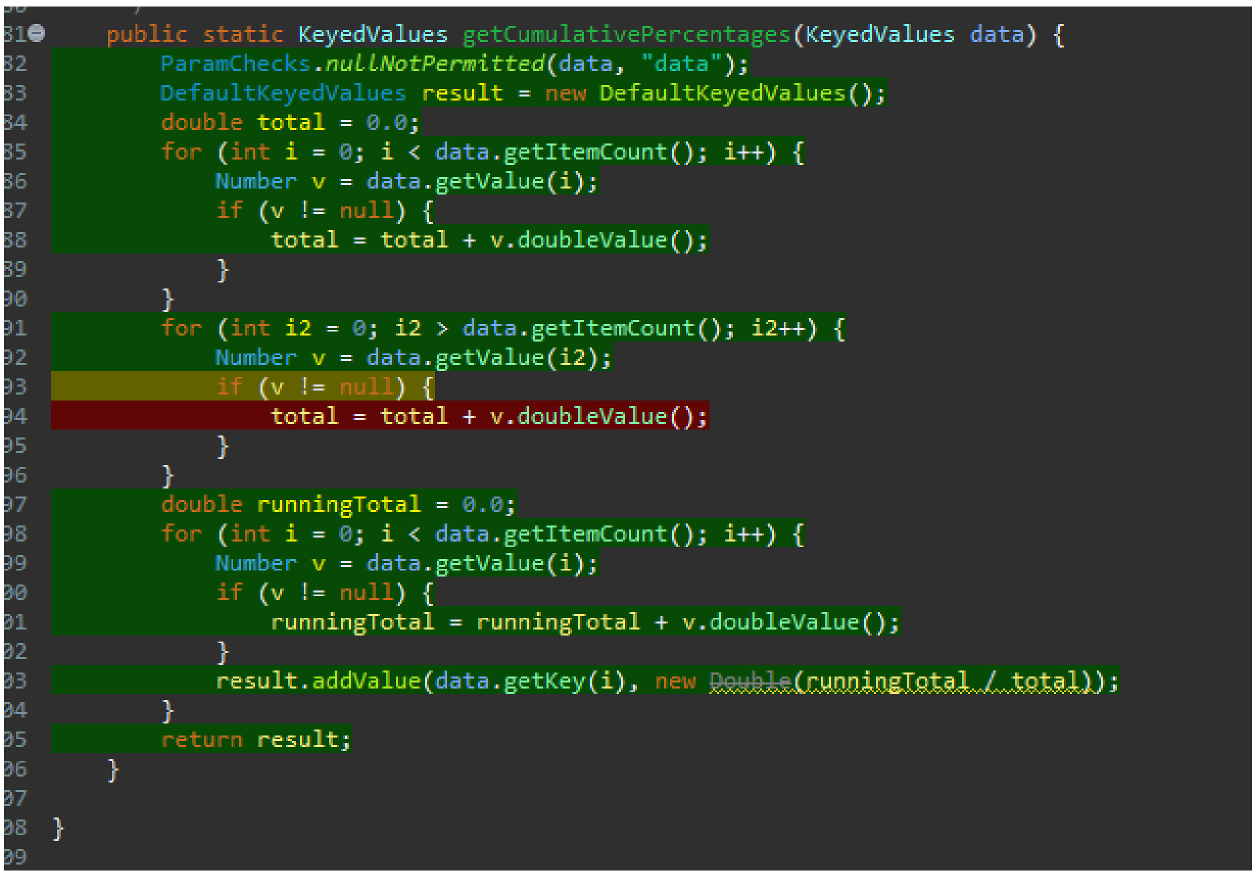
```
        */
    public static double calculateRowTotal(Values2D data, int row,
            int[] validCols) {
        ParamChecks.nullNotPermitted(data, "data");
        double total = 0.0;
        int colCount = data.getColumnCount();
        if (colCount < 0){
            total = 0.0;
        }
        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;
    }
```



```
     */
    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;
    }

    /**
```



```
     */
    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;
    }
```

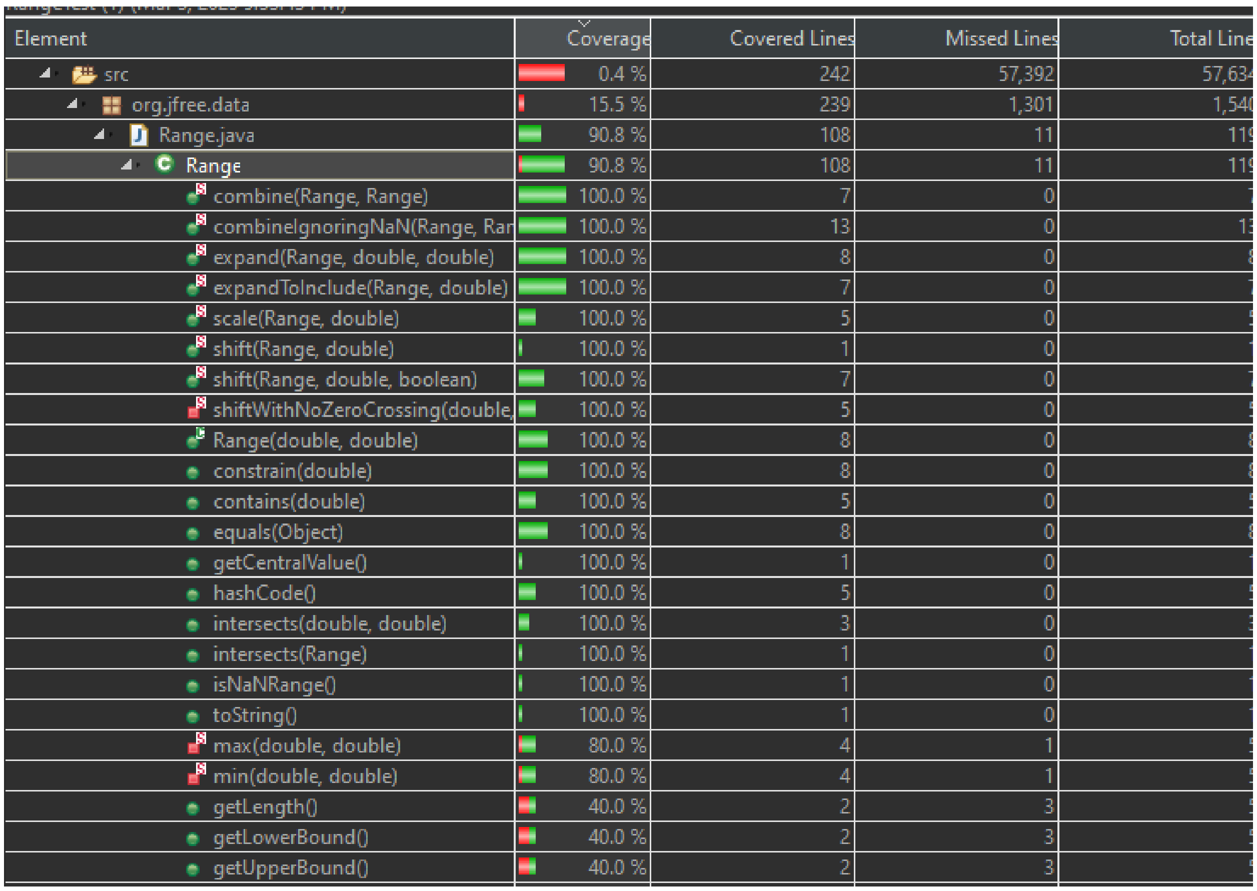


```java
    public static KeyedValues getCumulativePercentages(KeyedValues data) {
        ParamChecks.nullNotPermitted(data, "data");
        DefaultKeyedValues result = new DefaultKeyedValues();
        double total = 0.0;
        for (int i = 0; i < data.getItemCount(); i++) {
            Number v = data.getValue(i);
            if (v != null) {
                total = total + v.doubleValue();
            }
        }
        for (int i2 = 0; i2 > data.getItemCount(); i2++) {
            Number v = data.getValue(i2);
            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;
    }

}
```

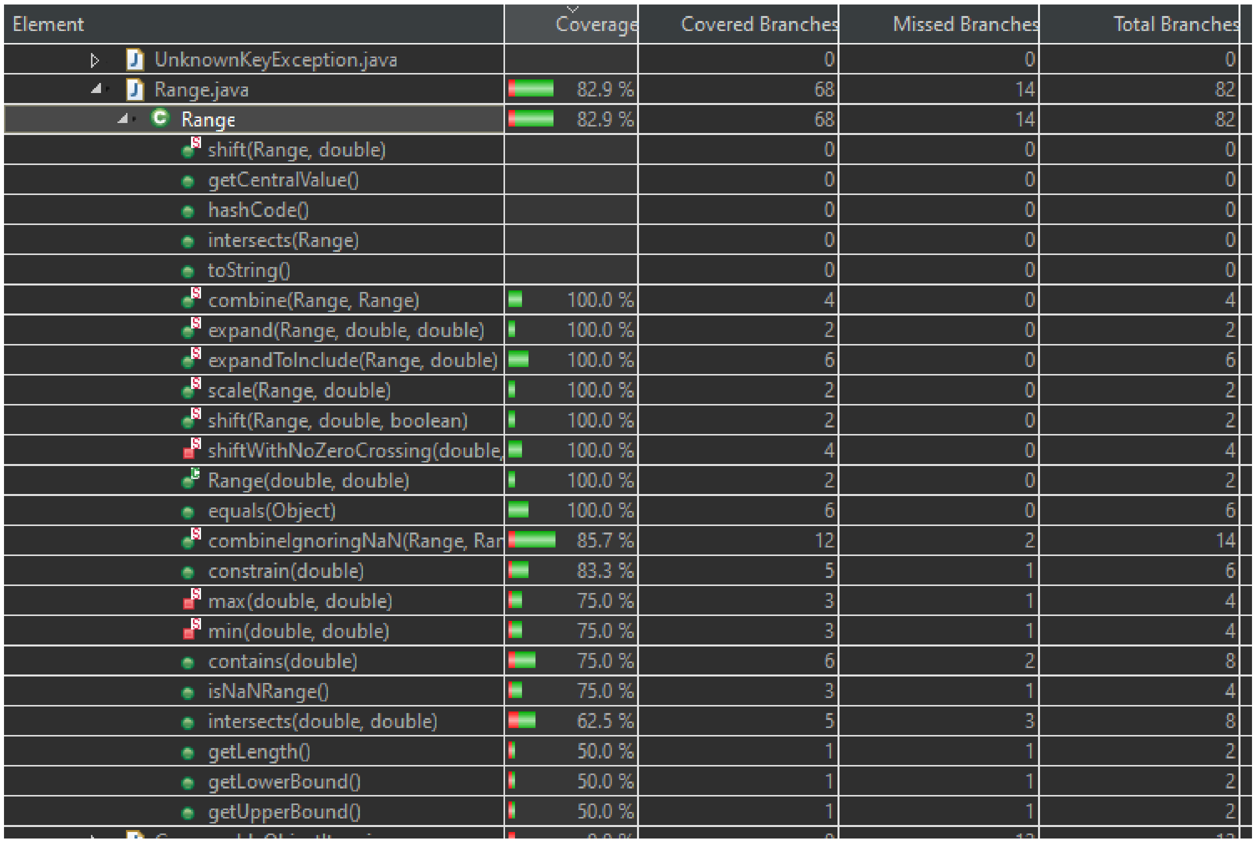
6 Pros and Cons of coverage tools used and Metrics you report

We used the built-in coverage tool, EclEmma to perform coverage calculation on our test cases. The tool has a lot of pros: very little setup is needed for it and it highlights coverage line-by-line as well as proving coverage counters for each individual method. The only con encountered is that the tool is unable to provide condition coverage, a very important metric and as such we had to replace our findings with method coverage.

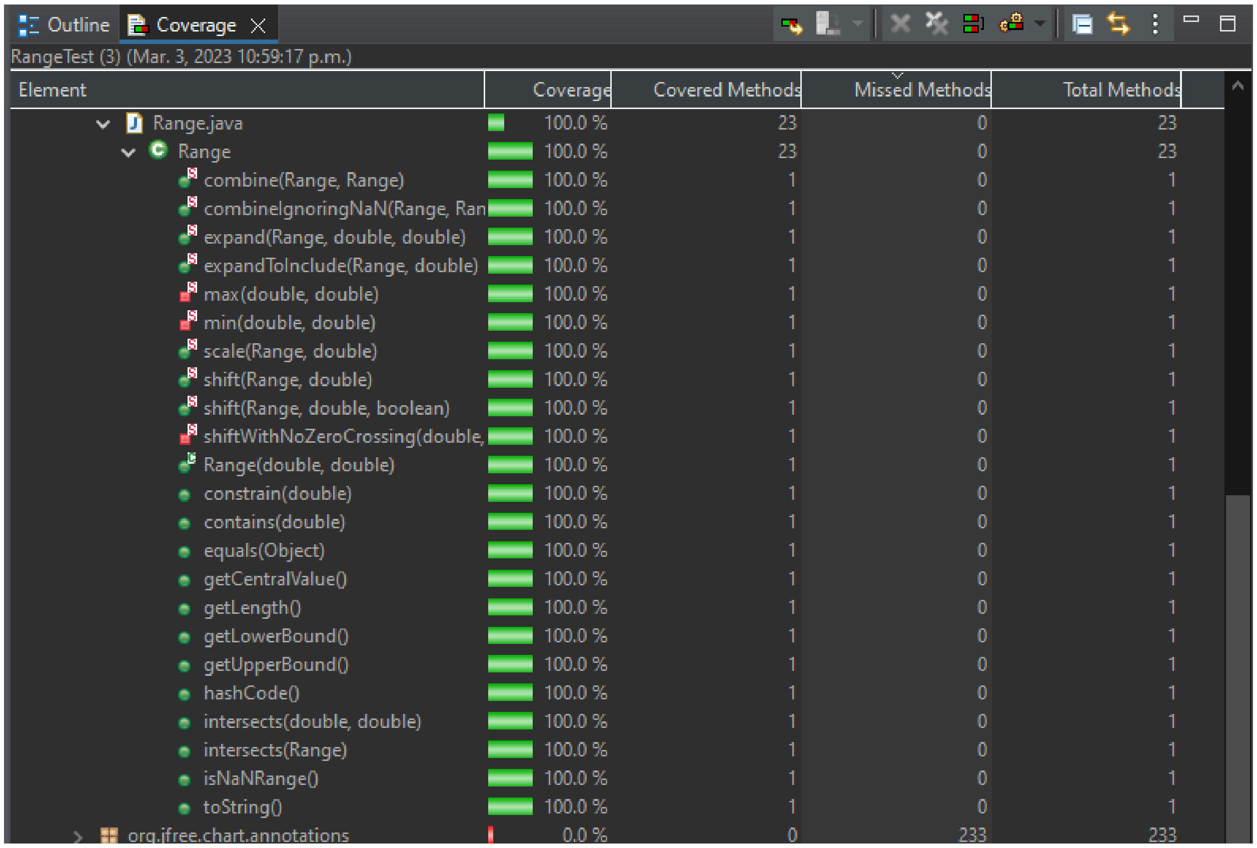
To measure coverage we used multiple metrics: line coverage, branch coverage, and method coverage as well as the DU coverage seen in Part 2. The control flow coverages are as follows:



Element	Coverage	Covered Lines	Missed Lines	Total Line
src	0.4 %	242	57,392	57,63
org.jfree.data	15.5 %	239	1,301	1,54
Range.java	90.8 %	108	11	11
Range	90.8 %	108	11	11
combine(Range, Range)	100.0 %	7	0	[illegible]
combineIgnoringNaN(Range, Rar	100.0 %	13	0	1
expand(Range, double, double)	100.0 %	8	0	[illegible]
expandToInclude(Range, double)	100.0 %	7	0	[illegible]
scale(Range, double)	100.0 %	5	0	[illegible]
shift(Range, double)	100.0 %	1	0	[illegible]
shift(Range, double, boolean)	100.0 %	7	0	[illegible]
shiftWithNoZeroCrossing(double,	100.0 %	5	0	[illegible]
Range(double, double)	100.0 %	8	0	[illegible]
constrain(double)	100.0 %	8	0	[illegible]
contains(double)	100.0 %	5	0	[illegible]
equals(Object)	100.0 %	8	0	[illegible]
getCentralValue()	100.0 %	1	0	[illegible]
hashCode()	100.0 %	5	0	[illegible]
intersects(double, double)	100.0 %	3	0	[illegible]
intersects(Range)	100.0 %	1	0	[illegible]
isNaNRange()	100.0 %	1	0	[illegible]
toString()	100.0 %	1	0	[illegible]
max(double, double)	80.0 %	4	1	[illegible]
min(double, double)	80.0 %	4	1	[illegible]
getLength()	40.0 %	2	3	[illegible]
getLowerBound()	40.0 %	2	3	[illegible]
getUpperBound()	40.0 %	2	3	[illegible]



Element	Coverage	Covered Branches	Missed Branches	Total Branches
UnknownKeyException.java		0	0	0
Range.java	82.9 %	68	14	82
Range	82.9 %	68	14	82
shift(Range, double)		0	0	0
getCentralValue()		0	0	0
hashCode()		0	0	0
intersects(Range)		0	0	0
toString()		0	0	0
combine(Range, Range)	100.0 %	4	0	4
expand(Range, double, double)	100.0 %	2	0	2
expandToInclude(Range, double)	100.0 %	6	0	6
scale(Range, double)	100.0 %	2	0	2
shift(Range, double, boolean)	100.0 %	2	0	2
shiftWithNoZeroCrossing(double,	100.0 %	4	0	4
Range(double, double)	100.0 %	2	0	2
equals(Object)	100.0 %	6	0	6
combineIgnoringNaN(Range, Ran	85.7 %	12	2	14
constrain(double)	83.3 %	5	1	6
max(double, double)	75.0 %	3	1	4
min(double, double)	75.0 %	3	1	4
contains(double)	75.0 %	6	2	8
isNaNRange()	75.0 %	3	1	4
intersects(double, double)	62.5 %	5	3	8
getLength()	50.0 %	1	1	2
getLowerBound()	50.0 %	1	1	2
getUpperBound()	50.0 %	1	1	2

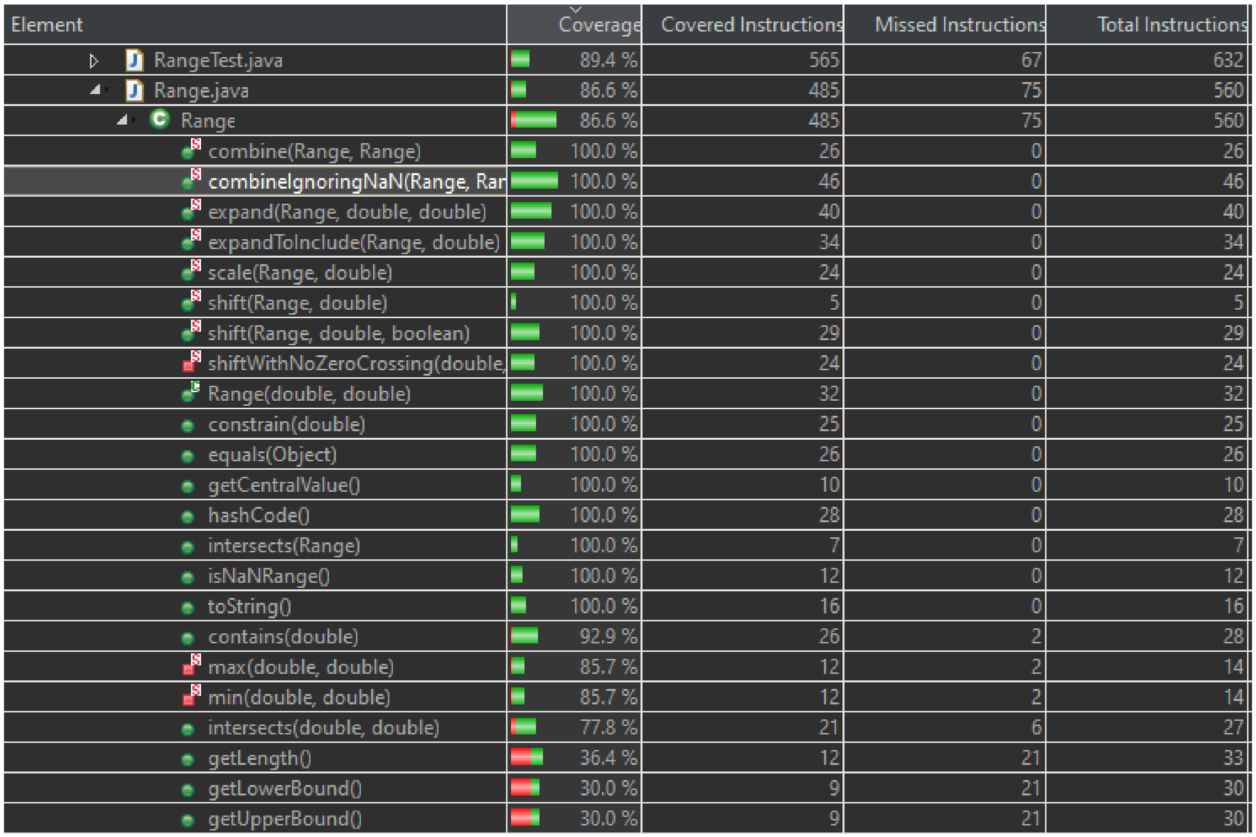


Outline Coverage

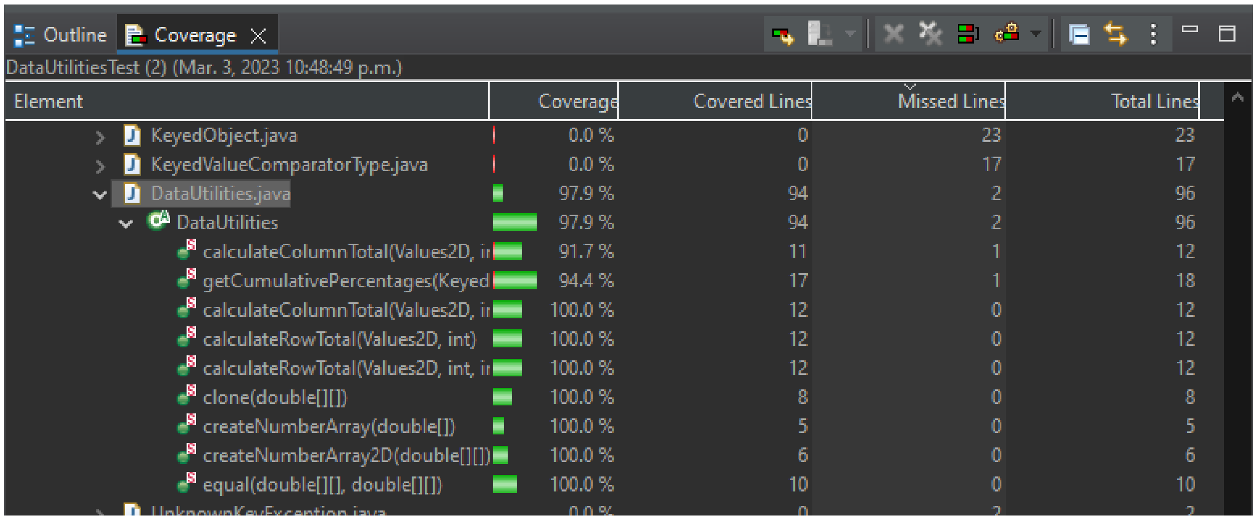
RangeTest (3) (Mar. 3, 2023 10:59:17 p.m.)

Element	Coverage	Covered Methods	Missed Methods	Total Methods
Range.java	100.0 %	23	0	23
Range	100.0 %	23	0	23
combine(Range, Range)	100.0 %	1	0	1
combineIgnoringNaN(Range, Ran	100.0 %	1	0	1
expand(Range, double, double)	100.0 %	1	0	1
expandToInclude(Range, double)	100.0 %	1	0	1
max(double, double)	100.0 %	1	0	1
min(double, double)	100.0 %	1	0	1
scale(Range, double)	100.0 %	1	0	1
shift(Range, double)	100.0 %	1	0	1
shift(Range, double, boolean)	100.0 %	1	0	1
shiftWithNoZeroCrossing(double,	100.0 %	1	0	1
Range(double, double)	100.0 %	1	0	1
constrain(double)	100.0 %	1	0	1
contains(double)	100.0 %	1	0	1
equals(Object)	100.0 %	1	0	1
getCentralValue()	100.0 %	1	0	1
getLength()	100.0 %	1	0	1
getLowerBound()	100.0 %	1	0	1
getUpperBound()	100.0 %	1	0	1
hashCode()	100.0 %	1	0	1
intersects(double, double)	100.0 %	1	0	1
intersects(Range)	100.0 %	1	0	1
isNaNRange()	100.0 %	1	0	1
toString()	100.0 %	1	0	1
org.jfree.chart.annotations	0.0 %	0	233	233

Instruction coverage, for extra:



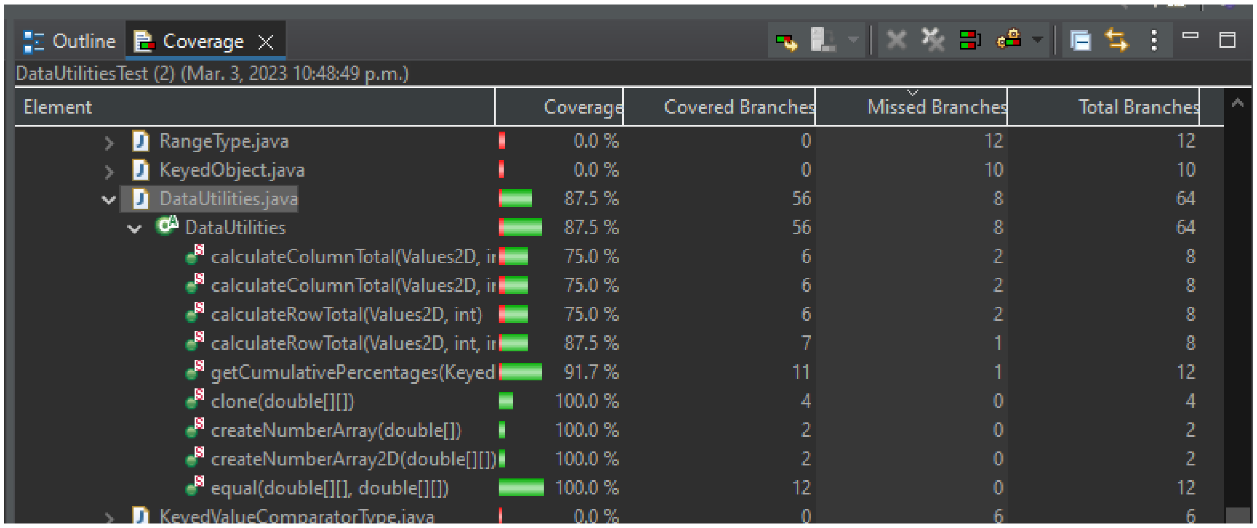
Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
RangeTest.java	89.4 %	565	67	632
Range.java	86.6 %	485	75	560
Range	86.6 %	485	75	560
combine(Range, Range)	100.0 %	26	0	26
combineIgnoringNaN(Range, Rar	100.0 %	46	0	46
expand(Range, double, double)	100.0 %	40	0	40
expandToInclude(Range, double)	100.0 %	34	0	34
scale(Range, double)	100.0 %	24	0	24
shift(Range, double)	100.0 %	5	0	5
shift(Range, double, boolean)	100.0 %	29	0	29
shiftWithNoZeroCrossing(double,	100.0 %	24	0	24
Range(double, double)	100.0 %	32	0	32
constrain(double)	100.0 %	25	0	25
equals(Object)	100.0 %	26	0	26
getCentralValue()	100.0 %	10	0	10
hashCode()	100.0 %	28	0	28
intersects(Range)	100.0 %	7	0	7
isNaNRange()	100.0 %	12	0	12
toString()	100.0 %	16	0	16
contains(double)	92.9 %	26	2	28
max(double, double)	85.7 %	12	2	14
min(double, double)	85.7 %	12	2	14
intersects(double, double)	77.8 %	21	6	27
getLength()	36.4 %	12	21	33
getLowerBound()	30.0 %	9	21	30
getUpperBound()	30.0 %	9	21	30



Outline | Coverage

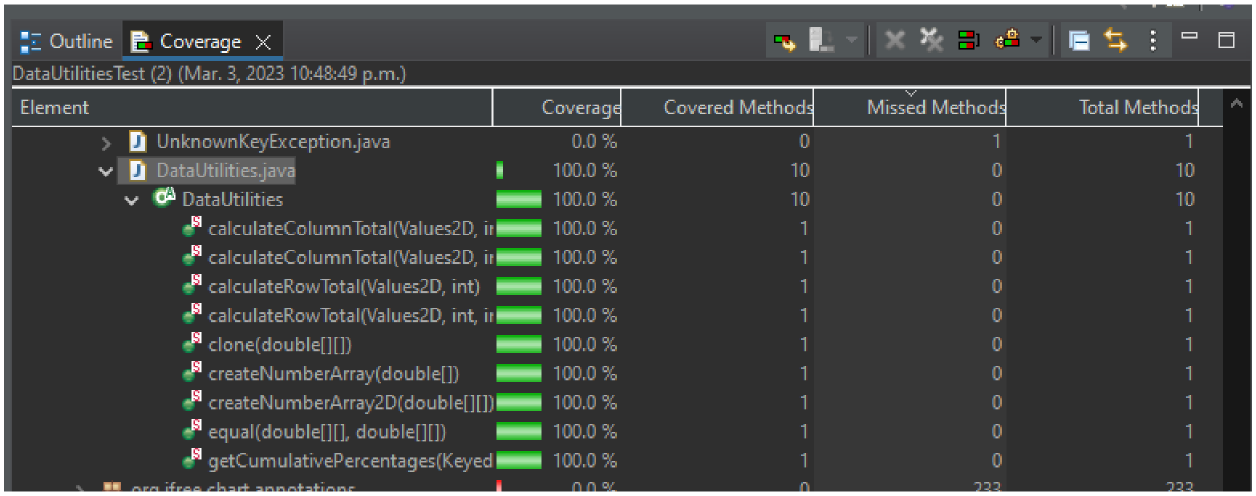
DataUtilitiesTest (2) (Mar. 3, 2023 10:48:49 p.m.)

Element	Coverage	Covered Lines	Missed Lines	Total Lines
KeyedObject.java	0.0 %	0	23	23
KeyedValueComparatorType.java	0.0 %	0	17	17
DataUtilities.java	97.9 %	94	2	96
DataUtilities	97.9 %	94	2	96
calculateColumnTotal(Values2D, ir	91.7 %	11	1	12
getCumulativePercentages(Keyed	94.4 %	17	1	18
calculateColumnTotal(Values2D, ir	100.0 %	12	0	12
calculateRowTotal(Values2D, int)	100.0 %	12	0	12
calculateRowTotal(Values2D, int, ir	100.0 %	12	0	12
clone(double[][])	100.0 %	8	0	8
createNumberArray(double[])	100.0 %	5	0	5
createNumberArray2D(double[][])	100.0 %	6	0	6
equal(double[][], double[][])	100.0 %	10	0	10
UnknownKeyException.java	0.0 %	0	2	2



DataUtilitiesTest (2) (Mar. 3, 2023 10:48:49 p.m.)

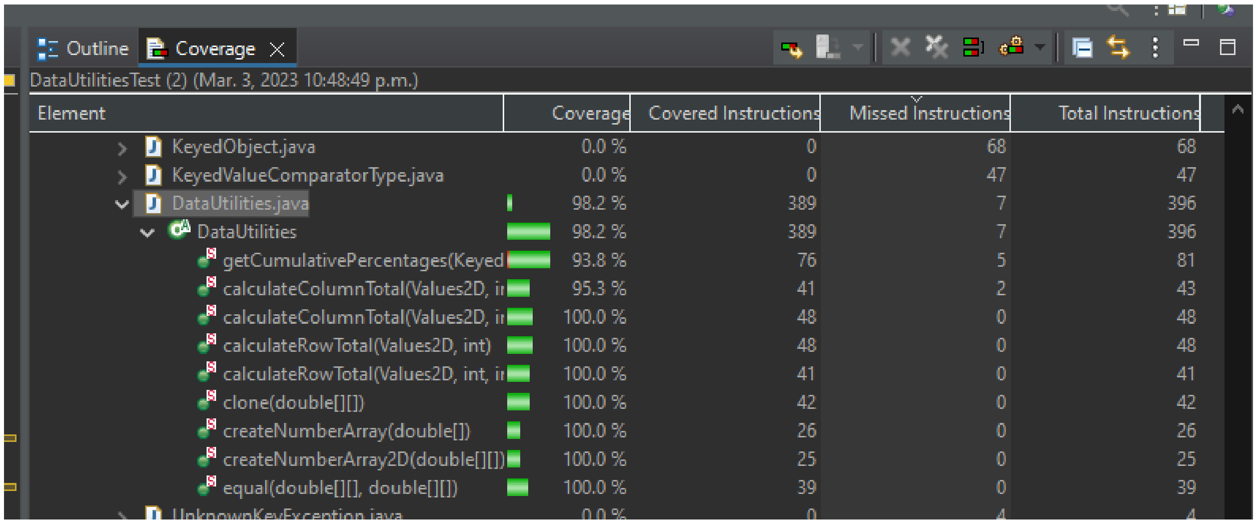
Element	Coverage	Covered Branches	Missed Branches	Total Branches
RangeType.java	0.0 %	0	12	12
KeyedObject.java	0.0 %	0	10	10
DataUtilities.java	87.5 %	56	8	64
DataUtilities	87.5 %	56	8	64
calculateColumnTotal(Values2D, ir	75.0 %	6	2	8
calculateColumnTotal(Values2D, ir	75.0 %	6	2	8
calculateRowTotal(Values2D, int)	75.0 %	6	2	8
calculateRowTotal(Values2D, int, ir	87.5 %	7	1	8
getCumulativePercentages(Keyed	91.7 %	11	1	12
clone(double[][])	100.0 %	4	0	4
createNumberArray(double[])	100.0 %	2	0	2
createNumberArray2D(double[][])	100.0 %	2	0	2
equal(double[][], double[][])	100.0 %	12	0	12
KeyedValueComparatorType.java	0.0 %	0	6	6



DataUtilitiesTest (2) (Mar. 3, 2023 10:48:49 p.m.)

Element	Coverage	Covered Methods	Missed Methods	Total Methods
UnknownKeyException.java	0.0 %	0	1	1
DataUtilities.java	100.0 %	10	0	10
DataUtilities	100.0 %	10	0	10
calculateColumnTotal(Values2D, ir	100.0 %	1	0	1
calculateColumnTotal(Values2D, ir	100.0 %	1	0	1
calculateRowTotal(Values2D, int)	100.0 %	1	0	1
calculateRowTotal(Values2D, int, ir	100.0 %	1	0	1
clone(double[][])	100.0 %	1	0	1
createNumberArray(double[])	100.0 %	1	0	1
createNumberArray2D(double[][])	100.0 %	1	0	1
equal(double[][], double[][])	100.0 %	1	0	1
getCumulativePercentages(Keyed	100.0 %	1	0	1
org.jfree.chart.annotations	0.0 %	0	233	233

Instruction coverage, for extra:



DataUtilitiesTest (2) (Mar. 3, 2023 10:48:49 p.m.)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
KeyedObject.java	0.0 %	0	68	68
KeyedValueComparatorType.java	0.0 %	0	47	47
DataUtilities.java	98.2 %	389	7	396
DataUtilities	98.2 %	389	7	396
getCumulativePercentages(Keyed	93.8 %	76	5	81
calculateColumnTotal(Values2D, ir	95.3 %	41	2	43
calculateColumnTotal(Values2D, ir	100.0 %	48	0	48
calculateRowTotal(Values2D, int)	100.0 %	48	0	48
calculateRowTotal(Values2D, int, ir	100.0 %	41	0	41
clone(double[][])	100.0 %	42	0	42
createNumberArray(double[])	100.0 %	26	0	26
createNumberArray2D(double[][])	100.0 %	25	0	25
equal(double[][], double[][])	100.0 %	39	0	39
UnknownKeyException.java	0.0 %	0	4	4

7 A comparison on the advantages and disadvantages of requirements-based test generation and coverage-based test generation.

Requirements-based test generation:

Requirements-based test generation's advantages are that it ensures completeness such that all requirements of the software are covered by the tests, which reduces the risk of missing any important functionality. It also focuses on testing the functionality of the software, which makes sure that the software meets all requirements. Moreover, early detection of defects helps to reduce cost of fixing defects in later stages.

The disadvantages of requirements-based test generation is that it is not very flexible. It is based on the requirements specification, so if the requirements change, the tests need to be updated accordingly, which can be very time consuming. It also has a limited scope, which may not cover all the possible scenarios that can occur during software execution.

Coverage-based test generation:

Coverage-based test generation's advantages are that it is focused on the source code which can help the tester to understand how the code works unlike requirements-based testing which is focused on outcomes. It is also more flexible than requirements-based testing, which means it is not dependent on the requirements specification. If the code changes, the tests are updated accordingly, which provides more flexibility.

The disadvantages of coverage-based test generation is that it has a limited focus on functionality, which means that it is possible that some scenarios are not convered during the execution of the software. Moreover, it takes more effort compared to requirements-based testing because all parts of the code must be tested.

8 A discussion on how the team work/effort was divided and managed

We originally planned on doing two pairs performing pair programming and review each other’s work afterwards but due to unexpected circumstances we had to adjust our plan a bit: two members performed the pair programming and the now-broken pair each completed their work individually. Regardless we were able to perform the tasks successfully in which all members contributed at all stages of the assignment and work was distributed equally.

9 Any difficulties encountered, challenges overcome, and lessons learned from performing the lab

We encountered quite a few difficulties in our lab but most of them came from factors external to the assignment itself: in fact our workflow was extremely good. After our experiences from Assignment 2 the objective and how to achieve it were much clearer this time around so it was much less difficult. However one of our group members ended up with really bad migraines throughout the week we were planning to work on the assignment which threw things off a bit. We managed to adapt and accommodate and they managed to get their work in regardless, albeit a bit close to deadline. We were able to problem solve along the way and it ended up working out thankfully.

The first problem that was encountered during this project was in the setup of our Eclipse project. We found that after creating the new project and following the steps described in the assignment instructions many of our tests that worked fine previously were now throwing exceptions. This was a result of certain library dependencies not being found which was fixed by including hamcrest-library-1.1.jar from assignment 2.

Overall the lab was a good way to demonstrate the importance of combining both white-box and black-box testing and how to optimally cover each metric. We were under a tighter schedule than anticipated so we had to work smart and work efficient, and in doing so learned a lot more than expected.

10 Comments/feedback on the lab itself

This lab was an insightful look into the industry standards of white-box software testing. the instructions were easy to follow and showed great clarity. The scope of white-box testing that the lab provided helped us understand white-box testing technique without being overbearing or overly time consuming. Overall, it was interesting to develop tests using a technique we had yet to encounter, and learning about it in the process of testing it to better develop test cases as we went along was very good practice which can help us in the industry as software engineers.

Please include the hamcrest-lib jar next time, it took us way too long to figure out it was missing. :(