Introduction (Test Plan):

We will begin by examining the methods to be tested. We will then review the JavaDoc index.html file to read up on the documentation and understand how the methods are expected to work. After examination, we will choose the methods we believe are most well-suited to showcase our knowledge of testing methods. With our methods chosen, we plan to examine each one in more depth, coming up with equivalence classes and potential boundary tests.

A detailed description of the unit testing strategy:

Returns the lower bound for the range.
Range - getLowerBound():
Input Domain:

- lower - any double <= upper bound within allowable values.

Equivalence Classes:

- Valid lower bound:
    - Lower bound is a valid value within a reasonable range (e.g., within the limits of double precision).
- Lower bound is at the minimum allowable value:
    - Lower bound is the smallest valid double value.
- Lower bound is just above the minimum allowable value:
    - Lower bound is a value just above the smallest valid double value.

Lower bound is at the maximum allowable value:
Lower bound is the largest valid double value.

^^^This cannot be tested since the lower bound must be less than the upper bound but the upper bound cannot be larger than the maximum allowed value.^^^

- Lower bound is just below the maximum allowable value:
    - Lower bound is a value just below the largest valid double value.
- Invalid lower bound:
    - Lower bound is NaN (Not a Number).

Returns the upper bound for the range.
Range - getUpperBound():
Input Domain:

- upper - any double >= lower bound within allowable values.

Equivalence Classes:

- Valid upper bound:
    - Upper bound is a valid value within a reasonable range (e.g., within the limits of double precision).
- Upper bound is at the maximum allowable value:
    - Upper bound is the largest valid double value.
- Upper bound is just below the maximum allowable value:
    - Upper bound is a value just below the largest valid double value.

- **Upper bound is at the minimum allowable value:**
    - **Upper bound is the smallest valid double value.**

^^^This cannot be tested since the upper bound must be greater than the lower bound but the lower bound cannot be less than the minimum allowed value.^^^

- Upper bound is just above the minimum allowable value:
    - Upper bound is a value just above the smallest valid double value.
- Invalid upper bound:
    - Upper bound is NaN (Not a Number).

Returns the length of the range.

Range - getLength():

Input Domain:

- lower - any double <= upper bound within allowable values.
- upper - any double >= lower bound within allowable values.

Equivalence Classes:

- Invalid range length (negative infinity):
    - The range length is negative infinity.
- Invalid range length (positive infinity)
    - The range length is positive infinity
- Invalid range length (NaN):
    - The range length is NaN (Not a Number).
- Valid range with positive length:
    - The range has a valid positive length (lower bound is less than upper bound).
- Single-point range length:
    - The range consists of a single point (lower bound equals upper bound).
- Negative range length:
    - The range has a negative length (lower bound greater than upper bound).

Returns true if the specified value is within the range and false otherwise

Range - contains(double value):

Input Domain:

- value - any double within allowable values.

Equivalence Classes:

- Value within Range Bounds:
  - Value less than Range upper bound and Value greater than Range lower bound.
- Value at Lower Bound
  - Value is equal to Range Lower Bound.
- Value at Upper Bound
  - Value is equal to Range Upper Bound.
- Value is Less than Lower Bound
  - Value is less than Range Lower and Upper Bound.
- Value is Greater than Upper Bound
  - Value is greater than Upper and Lower Bound.
- Value is Greater than Point Range
  - Value is greater than the Upper and Lower Bound where both bounds are equal.
- Value is Less than Point Range
  - Value is less than the Upper and Lower Bound where both bounds are equal.
- Value is equal to Point Range
  - Value is equal to the Upper and Lower Bound where both bounds are equal.
- Value is at Upper Bound at Double.MAX_VALUE
  - Value is equal to the Upper Bound where the Upper Bound is at Double.MAX_VALUE.
- Value is at Lower Bound at - Double.MAX_VALUE
  - Value is equal to the Lower Bound where the Lower Bound is at Double.MAX_VALUE
- Value within Max Size Range
  - Value is greater than lower bound and less than upper bound, where each are at Double.MAX_VALUE and Double.MAX_VALUE respectively.
- Invalid Value
  - Value passed to method not a valid double.

Returns true if the range intersects with the specified lower and upper bounds

Range - intersects(double lower, double upper):

Input Domain:

- lower - any double <= upper bound within allowable values.

- upper - any double >= lower bound within allowable values.

Equivalence Classes:

Input Range Enclosed within Range Instance
- Range.upper > Input.upper && Range.lower < Input.lower

Range Instance Enclosed within Input Range
- Range.lower > Input.lower && Range.upper < Input.Upper

Range Shares Lower Boundary with Input Upper Boundary
- Range.Lower == Input.upper

Range Shares Upper Boundary with Input Lower Boundary
- Range.Upper == Input.Lower

Input Range Above Range Instance
- Input.Lower > Range.Upper

Input Range Below Range Instance
- Input.Upper < Range.Lower

Partial Overlap to the Left of Range Instance
- Input.Lower < Range.Lower < Input.Upper < Range.Upper

Partial Overlap to the Right of Range Instance
- Range.Lower < Input.Lower < Range.Upper < Input.Upper

Point Range Inside Input Range
- Input.Lower < Range.Lower == Range.Upper < Input.Upper

Range Encloses Input Point Range
- Range.Lower < Input.Lower == Range.Upper < Input.Upper

Point Range Equivalent to Input Point Range
- Range.Lower == Range.Upper == Input.Lower == Input.Upper

Point Range Greater than Input Point Range
- Range.Lower == Range.Upper > Input.Lower == Input.Upper

Point Range Less than Input Point Range
- Range.Lower == Range.Upper < Input.Lower == Input.Upper

Invalid Input Range - Wrong Type
- Input into method of the wrong type

Invalid Input Range - Invalid Range
- Input.Lower > Input.Upper

Returns the sum of the values in one column of the supplied data table.

DataUtilities - calculateColumnTotal(Values2D data, int column):

Input Domains:
- data - all possible non-null, two-dimensional tables.
- column - any double within allowable values.

Equivalence Classes:
Valid data table and valid column index:
- Data table not null
- 0 < column-index < numColumns - 1

Valid data table and invalid column index:
- Data table not null
- 0 > column-index || column-index >= numColumns

Null data table:
- Data table is null

Empty data table:
- Data table not null but has no data

Valid data table but empty column:
- Data table not null
- Column specified empty

Valid data table with non-numeric values:
- Data table not null
- Column contains non-numeric values

Returns the sum of the values in one row of the supplied data table.
DataUtilities - calculateRowTotal(Values2D data, int row):
Input Domains:
- data - all possible non-null, two-dimensional tables.
- row - any double within allowable values.

Equivalence Classes:
Valid data table and valid row index:
- Data table not null
- 0 < row-index < numRows - 1

Valid data table and invalid row index:
- Data table not null
- 0 > row-index || row-index >= numrows

Null data table:
- Data table is null

Empty data table:
- Data table not null but has no data

Valid data table but empty row:
- Data table not null
- Row specified empty

Valid data table with non-numeric values:
- Data table not null

- Row contains non-numeric values

Constructs an array of Number objects from an array of double primitives.
DataUtilities - createNumberArray(double[] data):
Input Domain:
- data - any non-null array of type double.

Equivalence Classes:
Valid array
- Valid data

Empty array
- No data in array

Array with boundary values:
- Data contains maximum and minimum values

Constructs an array of arrays of Number objects from a corresponding structure containing double primitives.
DataUtilities - createNumberArray2D(double[][] data):
Input Domain:
- data - any two-dimensional, non-null array of type double.

Equivalence Classes:
Valid array and valid subarray
- Data is not null
- Data contains at least one non-null sub-array

Valid array with empty subarray
- Data is not null
- Data contains only empty subarray

Empty array
- Data is empty

Returns a KeyedValues instance that contains the cumulative percentage values for the data in another KeyedValues instance.
DataUtilities - getCumulativePercentages(KeyedValues data):
Input Domain:
- data - non-null data of type KeyedValue

Equivalence Classes:
Non-empty KeyedValues
- Data is not null

- Data contains at least one key-value pair.

Empty KeyedValues
- Data is not null
- Data has no key-value pairs

KeyValues with negative values:
- Data is not null
- Data has negative key values

KeyValues with zero values:
- Data is not null
- Data has values of zero

KeyValues with boundary values:
- Data is not null
- Data has values equating to boundary values

KeyValues with null values
- KeyValues object not null
- All values in KeyValue are set to null

Test cases developed:

Equivalence Class Partitioning:
- We identified different equivalence classes for each method based on the expected behaviour and the range of valid input values.
- Each equivalence class represented a distinct category of input values that should produce similar behaviour from the method under test.

Boundary Value Analysis:
- We selected test cases at the boundaries of the equivalence classes to ensure thorough testing of edge cases.
- Boundary values often represent points where the behaviour of the method may change, making them critical for detecting potential issues.

Test Case Design:
- We designed test cases to cover both normal and boundary conditions for each method.
- Test cases were named descriptively to indicate the scenario being tested and the method under test.

Benefits of Using Mocking:

Isolation:
- Mocking allows us to isolate the unit under test from its dependencies, enabling focused testing on specific components.

Control:

- With mocking, we can control the behaviour of dependencies to simulate different scenarios and edge cases.

Efficiency:

- Mocking reduces the need for extensive setup and teardown procedures, making tests more efficient to write and execute.

Drawbacks of Using Mocking:

Complexity:

- Mocking frameworks can introduce complexity, especially for inexperienced developers, leading to difficult-to-understand tests.

Maintenance Overhead:

- Mocks need to be kept up-to-date with changes in the system under test, which can introduce maintenance overhead.

Risk of Over-Mocking:

- Over-reliance on mocking may lead to tests that are tightly coupled to implementation details, making them brittle and prone to breakage with refactoring.

How the teamwork/effort was divided and managed:

We organized our workload by assigning responsibilities between the Range and DataUtilities classes. Each team member took ownership of writing test cases and defining equivalence classes for either two or three methods within their assigned class. We collaborated, offering support to group members facing challenges and providing assistance when needed. Additionally, we engaged in peer reviews, thoroughly examining each other's test cases and offering constructive feedback to ensure the quality and effectiveness of our testing efforts

Difficulties encountered, challenges overcome, and lessons learned during the lab:

One challenge we collectively faced was getting used to the Eclipse IDE as we had mostly used VScode before. After reviewing lab instructions and through the process of completing the lab itself, we were able to better understand the integration of JUnit in the Eclipse IDE.

Comments/feedback on the lab itself:

As previously stated, our group was much more familiar with VScode and it is our preferred IDE and we have used it for JUnit testing before. Consequently, we would have much preferred the

use of VSCode for this lab as we feel it is a more modern IDE and functions identically to Eclipse in terms of testing.