ECSE 429: Project Part 1

Group 10

Michael MAATOUK	260554267
Lucien GEORGE	260554267
Alexander BRATYSHKIN	260684228
Nathan LAFRANCE-BERGER	260686487
Benjamin WILLMS	260690005

Table of Contents

1. Introduction 2

2. Testing Strategy 2

3. Coverage 6

4. Test Descriptions 8

List Implementations 8

Regular Iterators 25

Map Iterators 32

Ordered Maps 42

Adapters 56

Decorators 58

1. Introduction

The Apache Commons Collections library is centered around extending the java.util.collection package with more specific collections. This is achieved by providing additional interfaces fulfilling different niche, albeit useful, contracts on one hand, along with a choice of corresponding implementations on the other hand. The goal here is to provide value to developers by reducing source code length and complexity.

Our goal in this report was to deduce the core functionality of the Apache Commons Collections, decide upon an appropriate Acceptance Testing technique, design a comprehensive test suite using the decided upon testing technique, and approximate the test suite's coverage.

2. Testing Strategy

Originally, when analyzing the Apache Commons Collections, we were considering using **category partitioning** to derive our test suite. It was easy to see how the functions could be split into categories – most importantly: create, read, update, and delete - and then those categories could have choices associated with them which correspond to the different specific methods that fall under the categories. For example, the choices associated with "create" for lists could be: the basic constructor (possibly calling new List()), the constructor with inputs (maybe the input is length: new List(length)), and fully specified list creation (List k = {23, 20, 3}). In this way we could have broken down all the core functionality of the Collections into several easily testable choices. However, we also saw that we could use **boundary value analysis** to derive our tests. We saw that the inputs to these functions could be partitioned despite the Collections' general use of input (any number of data types can be used as input). For example, for the create specified list, we could test an empty list (List k = {}), a non-empty list (List k = {23, 20, 3}), and even a null list (List k = null). In the same manner, we could use **equivalence partitioning**, but instead of testing boundaries, we could simply test a valid and an invalid input.

With these three possibilities in mind, we were required to choose the one that best suits our needs. To do this, we considered code complexity (we were among the groups who believed that we were required to implement the test cases), possible coverage, and even documentation complexity.

Documentation Complexity

Category Partitioning	Requires extensive documentation to fully explain and demonstrate the process of creating the test suite (showing category and choice derivation, as well as constraints, and final criterion used to determine the test cases)
Boundary Value Analysis	Can be documented directly in the test case descriptions by simply including the inputs (the partitions used are implied by the inputs, and are easy to see and understand).

Equivalence Partitioning	Can be documented directly in the test case descriptions by including the inputs and showing validity of the input (possibly in the expected result).

Possible Coverage

Category Partitioning	· The least that can be done is covering each choice. This, however would simply test each method (that we had deemed to be "core functionality") a single time, to see if it worked as expected with one input. · The coverage could be extended by using all valid combinations of choices, but assuming each of the 4 CRUD categories has only 3 core methods, this would be 81 test cases (before constraints). The problem here is that there are not only 3 core methods per category, and there are a few extra categories for some of the data structures. The number of tests would be huge, and as we are simply performing acceptance testing, many of the tests would show the same function performing in the same way, which is redundant.
Boundary Value Analysis	· This would still need to test each of the "core" methods, so it would cover each choice from category partitioning, except it would also cover a range of possible inputs to these methods. Each choice would be covered in a more complete way, covering many high-risk boundary values on the inputs.
Equivalence Partitioning	· This would cover the same methods as boundary value analysis, but would cut down on the number of test cases by considering only one invalid and one valid input. This works off the assumption that all invalid inputs are treated the same, and all valid inputs are treated the same.

Code Complexity

Category Partitioning	Each test case would include calls to all categories of methods, meaning the cases would be complex in structure.
Boundary Value Analysis	Each test case would test a single method in a single way, meaning the structure of the cases would be extremely simple. This does mean that there would be several cases needed to cover all the methods that one case of category partitioning would cover.
Equivalence Partitioning	The code complexity would be the same as that of boundary value analysis.

We decided to use **equivalence partitioning**. Although it requires more test cases than using each choice for category partitioning (as it tests one method per test instead of 4 or more per test, and it also requires the testing of valid and invalid inputs), it also covers far more possible errors, due to the testing being based on validity of input. Equivalence partitioning also requires far less test cases than basic boundary value analysis (min, min+, nom, max-, max, as compared to simply valid, and invalid), while minimizing possible redundant test cases. In the case of the Apache Commons Collections, it has been extensively unit tested beforehand. This means that most of the boundaries have likely already been tested, and we are only interested in testing if the basic advertised functionality is delivered by the code package. This makes us believe that we can safely assume that all valid inputs and all invalid inputs are dealt with in the same manner, and many of the test cases included in boundary value analysis would be redundant. Further, the code complexity is extremely low, which should theoretically help cut down on coding time per test case, allowing more to be written in a short time. This should help offset the large number of test cases required as compared to the number needed if each choice for category partitioning was used. Finally, equivalence partitioning does not require complex documentation to show and explain.

To decide upon the core functionality that we would test, we read the overview of the Apache Commons Collections web page. The most important section we settled on was as follows:

> *"Commons-Collections seek to build upon the JDK classes by providing new interfaces, implementations and utilities. There are many features, including:*
>
> · *Bag interface for collections that have a number of copies of each object*

- *BidiMap interface for maps that can be looked up from value to key as well and key to value*
- *MapIterator interface to provide simple and quick iteration over maps*
- *Transforming decorators that alter each object as it is added to the collection*
- *Composite collections that make multiple collections look like one*
- *Ordered maps and sets that retain the order elements are added in, including an LRU based map*
- *Reference map that allows keys and/or values to be garbage collected under close control*
- *Many comparator implementations*
- *Many iterator implementations*
- *Adapter classes from array and enumerations to collections*
- *Utilities to test or create typical set-theory properties of collections such as union, intersection, and closure"*

We used this as a form of requirements document, letting each point (including the first sentence) be a general requirement. We then dove deeper into the documentation for each of these points to decide on the main data structures and methods to be tested. For example, once we dug deeper into the documentation, lists were deemed to be included in the first sentence ("*Commons-Collections seek to build upon the JDK classes by providing new interfaces, implementations and utilities.*"). We decided upon testing the main CRUD operations, simply because their functionality is imperative for all other possible operations. For example, a sort operation requires reading and updating of a resource. CRUD operations are the most basic actions one can take on a resource, and all other operations require some combination of them. We also decided to test a few other useful methods. For these we tried to guess which would be used the most, which allowed us to gauge their "usefulness". For example, sorting is used for many applications, so we decided to add tests for sorting. Also, the retain methods allow you to quickly keep only desired elements in a collection, while disposing of the rest. This is an extremely useful function, as in a large collection with few necessary values, you can "retain" the important values instead of simply calling "remove" over and over again. To test the vaguer points, such as "*Commons-Collections seek to build upon the JDK classes,"* we decided to even test some performance, which would allow us to show that the Commons Collections improve upon the Java JDK in a measurable way (Test #17 of the List Implementations Section). There are also many similar implementations of data structures, such as linked lists and array lists. These can (for the most part) be tested in the exact same way, so copying the tests repeatedly to test each with the different implementations seemed inefficient.

For this we found **Jukito**. Jukito is a framework that allows us to define a list of instances of classes, and then run a single test on each of these instances. This means that we could define

a list of different types of lists, as well as different situations in each of these list types (lists with repeated parts, all unique parts, empty, etc.), and then run a single test case with each of these defined lists. This allowed us to cut down on the amount of necessary code to test all the different implementations of lists (as well as other classes with core similarities), and also allowed us to quickly throw in extra situations that could be useful to test that do not fall under the equivalence partitions (for example, retaining x in a list that contains x more than once, doesn't contain it at all, etc.).

3. Coverage

Coverage in a typical, concrete sense was impossible to measure for this project, as the Apache Commons Collections are an external dependency; we do not know of any real static analysis tool for checking test targets of external dependencies. Because of this, coverage was a difficult thing to gauge, so for this discussion we will mention multiple different types. These will be requirements coverage (how many of the aforementioned points on the overview page were covered in a significant way), method coverage (how many of the methods were covered), and partition coverage (how many equivalence partitions were covered, and at what level).

From the beginning, we decided to aim for complete coverage of the 12 points (requirements) on the overview page. That isn't to say that we aimed to test if the Collections expanded upon (included all functionality, plus extra) the JDK in all cases, but instead that we wanted to cover each point in terms of acceptability. This means we tried to check if the basic functionality was there, and if each point was delivered upon in a way that could be used by future developers. For this goal, we believe that our test cases have achieved this desired level of coverage. We have a significant number of test cases dedicated to each of the 12 points, and they cover basic functionality, as well as some extras.

Moving on to method coverage, we aimed for coverage of the most important methods. As there are a huge number of methods available, testing them all was simply not feasible. Instead, we tried to focus on those that seem most important. This includes all the basic CRUD operations, as well as some more focused and specific methods that we have seen many uses for in the past (such as in past courses like COMP 250, where sort and search methods are extremely prevalent). This was an ad-hoc process that was far from perfect; importance of the methods is truly based on the use case, as each one will be used in different ways and with different frequencies under various scenarios. We still attempted to determine general levels of importance to give us a non-random way of determining which secondary methods to test. We believe that leaving them all untested is not acceptable, and testing them all is unfeasible, therefore having a way to choose which to test is imperative. In the end, we believe that our test suite has met our desired level of coverage in an acceptable manner. Our tests cover all CRUD methods and numerous secondary methods for each class. While this is far from complete

method coverage (the percentage of method coverage achieved could be easily – read "tediously" – be calculated, but this was not done), we believe it covers them adequately, as many programming scenarios will never require the use of the untested methods.

For partition coverage, we aimed to partition all inputs to methods, and cover each partition at least once. In reality, not all methods could have their inputs properly partitioned. For example, in the sort method, the input is simply a list. This could technically be null, but we decided not to test null instances, as they would require some exceptional circumstances to occur in practice. That leaves only one partition; namely, a list of any type. There is only a valid input in this case, and so that is all that was tested. We deemed this level of coverage to be acceptable, as the Collections have already been extensively unit tested, so each valid input should be treated the same, and each invalid input should be treated the same. This means that we assumed boundary value analysis to be redundant, and therefore didn't need its much higher level of coverage. Our test suite fulfills our partition coverage goals in almost all cases, with rare circumstances such as the sort method being exceptions.

4. Test Descriptions

List Implementations

Test 1 - Regular addition to list instance

Inputs:

Instance(s)	Parameter(s)
- TreeList<>(Arrays.asList("xyz", "abc", "cbd")) - CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - NodeCachingLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - TreeList<>() - CursorableLinkedList<>() - NodeCachingLinkedList<>()	- String: “test”

Steps:

1. Store the length of the list in a buffer variable.
2. Add “test” string element to the list instance using the .add() method provided by the common List interface.

Expected Result:

Length of list instance should have increased by one.

Actual Result:

Buffer storing size of list instance prior to the addition plus one is equal to the size of array after new element’s addition.

Verdict:

Concrete List implementations in the Apache Commons Collections work for addition of single items for both empty and non-empty instances of TreeList, CursorableLinkedList and NodeCachingLinkedList.

Test 2 - Addition of multiple items to list instance

Inputs:

Instance(s)	Parameter(s)
- TreeList<>(Arrays.asList("xyz", "abc", "cbd")) - CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - NodeCachingLinkedList<>(Arrays.asList("xyz", "abc", "cbd"))	- String: “test1” - String: “test2”

Steps:

1. Store the length of the list in a buffer variable.
2. Add “test1” and “test2” strings elements (wrapped in an Array.asLists method) to the list using the .addAll() method provided by the common List interface.

Expected Result:

Length of list instance should have increased by two.

Actual Result:

Buffer storing size of length of the instance prior to the addition plus two is equal to the size of array after new element’s addition.

Verdict:

Concrete List implementations in the Apache Commons Collections work for addition of multiple items.

Test 3 - Addition of items to list instance at given index

Inputs:

Instance(s)	Parameter(s)
- TreeList<>(Arrays.asList("xyz", "abc",	- String: “test1”

"cbd")) - CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - NodeCachingLinkedList<>(Arrays.asList("xyz", "abc", "cbd"))	- String: “test2” - Integer: 1 - Integer: 90

Steps:

1. Add “test1” and “test2” strings elements (wrapped in an Array.asLists method) to the list using the .addAll() method provided by the common List interface, and by passing “1” as the index at which we want to add the elements
2. Add “test1” and “test2” strings elements (wrapped in an Array.asLists method) to the list using the .addAll() method provided by the common List interface, and by passing “90” as the index at which we want to add the elements

Expected Result:

1. Item at index 1 should be “test1”, item at index 2 should be “test2”
2. Addition should fail with an IndexOutOfBounds exception

Actual Result:

New elements “test1” and “test2” have been added to the lists’ instances at indices 1 and 2. IndexOutOfBounds exception thrown while adding at an index that is outside of the list's length boundary.

Verdict:

Concrete List implementations in the Apache Commons Collections work for addition of multiple items at given indices.

Test 4 - Removal of items from list instance at given index

Inputs:

Instance(s)	Parameter(s)
- TreeList<>(Arrays.asList("xyz", "abc", "cbd")) - CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd"))	- Integer: 1 - Integer: 15

- NodeCachingLinkedList<>(Arrays.asList ("xyz", "abc", "cbd"))	

Steps:

1. Remove “xyz” element from the list using the .remove() method provided by the common List interface, and by passing “1” as the index from which we want to remove “xyz.”
2. Store return element from .remove() method in temporary buffer variable.
3. Remove “xyz” element from the list using the .remove() method provided by the common List interface, and by passing “15” as the index from which we want to remove “xyz”

Expected Result:

1. Removed element buffer should be equal to the element that we just removed from the list.
2. Removal should fail with an IndexOutOfBounds exception

Actual Result:

Element “xyz” has been removed from the list since buffer string is equal to the value of the element we wished to remove.

Verdict:

Concrete List implementations in the Apache Commons Collections work for removal of single items at given indices.

Test 5 - Removal of items from list instance by specifying concrete objects

Inputs:

Instance(s)	Parameter(s)
- TreeList<>(Arrays.asList("xyz", "abc", "cbd")) - CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - NodeCachingLinkedList<>(Arrays.asList ("xyz", "abc", "cbd"))	- String: “cbd” - String: “testWrong”

Steps:

1. Remove “cbd” element from the list using the .remove() method provided by the common List interface, and by passing “cbd” as a parameter to the function.
2. Check that function returns true if element was successfully removed.
3. Attempt to remove “testWrong” element from the list using the .remove() method provided by the common List interface, and by passing “testWrong” as a parameter to the function.
4. Check that function returns false if element wasn’t successfully removed or wasn’t present in the list.

Expected Result:

1. .remove(“cbd”) should return true when removing “cbd” and no instance should not contain the “cbd” element
2. .remove(“testWrong”) should return false due to the fact that element “testWrong” is not present in any of the instance

Actual Result:

Element “cbd” has been removed from the list instances.

Verdict:

Concrete List implementations in the Apache Commons Collections work for removal of single specified items at given indices.

Test 6 - Remove all items from list

Inputs:

Instance(s)	Parameter(s)
- TreeList<>(Arrays.asList("xyz", "abc", "cbd")) - CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - NodeCachingLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - TreeList<>() - CursorableLinkedList<>() - NodeCachingLinkedList<>()	

- TreeList<>(Arrays.asList("xyz", "abc", "cbd", "abc")), - CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd", "abc")), - NodeCachingLinkedList<>(Arrays.asList("xyz", "abc", "cbd", "abc"))	

Steps:

1. Call .clear() method on given instances of list.
2. Check that size of the list instance is 0 after call of method above.

Expected Result:

1. Empty instances of the lists should return 0 and still remain empty, with no exception thrown.
2. Non-empty instances without duplicate items should have size 0.
3. Non-empty instances with duplicate items should have size 0.

Actual Result:

All elements have been removed from non-empty with duplicates and non-empty without duplicate instances. Empty instances of the lists remained empty.

Verdict:

Concrete List implementations in the Apache Commons Collections work for full clearance.

Test 7 - Remove multiple concrete items from the list

Inputs:

Instance(s)	Parameter(s)
- TreeList<>(Arrays.asList("xyz", "abc", "cbd")) - CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - NodeCachingLinkedList<>(Arrays.asList	- String: “abc” - String: “cbd” - Null

("xyz", "abc", "cbd")) - TreeList<>(Arrays.asList("xyz", "abc", "cbd", "abc")), - CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd", "abc")), - NodeCachingLinkedList<>(Arrays.asList("xyz", "abc", "cbd", "abc"))	

Steps:

1. Remove “abc” and “cbd” elements from the list instance by wrapping them in an Array.asList() call to then pass the wrapped collection to the .remove() method provided by the common List interface.
2. Check that the returned value from the method containsAll() returns false when queried for elements “abc” and “cbd”
3. Call .remove() method one more time by passing “null” as a parameter.

Expected Result:

1. Non-empty instances of lists without duplicates should only contain item “xyz”
2. Non empty instances of lists with duplicate items should only contain item “xyz”
3. Removal of null item should return NullPointerExpection

Actual Result:

Elements “abc” and “cbd” are not contained within the instances of the given elements, containing only item “xyz.” Removal of null throws a NullPointerException.

Verdict:

Concrete List implementations in the Apache Commons Collections work for removal of multiple concrete items.

Test 8 - Update item at a given index

Inputs:

Instance(s)	Parameter(s)
- TreeList<>(Arrays.asList("xyz", "abc", "cbd")) - CursorableLinkedList<>(Arrays.asList("x	- Integer: 0 - String “test” - Integer: 8

yz", "abc", "cbd")) - NodeCachingLinkedList<>(Arrays.asList ("xyz", "abc", "cbd"))	

Steps:

1. Call .set() method with 0 as first parameter to insert item “test” in place of already present item “xyz” in the item’s instance.
2. Verify that item at index 0 is now equal to “test” with a .get() call.
3. Call .set() method with 8 as first parameter to attempt to insert item “test” outside of the boundaries of the list’s length.

Expected Result:

1. Item “xyz” should be replaced by value “test” at index 0 of list’s instance.
2. IndexOutOfBounds exception should be thrown when calling the .set() method with an index outside of the list’s size.

Actual Result:

Element “xyz” has been replaced by element “test.” Exception was thrown as expected.

Verdict:

Concrete List implementations in the Apache Commons Collections work for update of items inside of an already existing instance.

Test 9 - Update item at a given index

Inputs:

Instance(s)	Parameter(s)
- TreeList<>(Arrays.asList("xyz", "abc", "cbd")) - CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - NodeCachingLinkedList<>(Arrays.asList ("xyz", "abc", "cbd"))	- Integer: 0 - Integer: 10

Steps:

1. Call .get() with parameter 0 on instance of list
2. Verify that the method returns item “xyz”
3. Call .get() with parameter 8 on instance of list, which is outside of the list’s boundary length.

Expected Result:

1. The first call should return item “xyz”
2. IndexOutOfBounds exception should be thrown when calling the method outside of the lists’ instances’ length boundary.

Actual Result:

Return value from .get() call at 0 returns “xyz,” throws an exception when called with parameter 10.

Verdict:

Concrete List implementations in the Apache Commons Collections allows retrieval of items from list.

Test 10 - Emptiness of lists

Inputs:

Instance(s)	Parameter(s)
- TreeList<>(Arrays.asList("xyz", "abc", "cbd")) - CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - NodeCachingLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - TreeList<>() - CursorableLinkedList<>() - NodeCachingLinkedList<>()	

Steps:

1. Call .isEmpty() method on empty instance of list
2. Verify that call returns true.
3. Call .isEmpty() method on non-empty instance of list

4. Verify that call returns false.

Expected Result:

1. First call to empty instances of lists should return true
3. Second call to non-empty instances of lists should return false

Actual Result:

Non-empty lists false when queried about their emptiness, empty lists return true.

Verdict:

Concrete List implementations in the Apache Commons Collections provide a working way of checking for the emptiness of a list.

Test 11 - Presence of single items in lists

Inputs:

Instance(s)	Parameter(s)
- TreeList<>(Arrays.asList("xyz", "abc", "cbd")) - CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - NodeCachingLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - TreeList<>() - CursorableLinkedList<>() - NodeCachingLinkedList<>()	- String: “xyz”

Steps:

1. Use .contains() method by passing “xyz” as its parameter on non-empty list instances that do contain “xyz”
2. Check if return value of method is true
3. Use .contains() method by passing “xyz” as its parameter on empty list instances that do not contain “xyz”
4. Check if return value of second call is false

Expected Result:

1. First method call on non-empty instances should return true

2. Second method call on empty instances should return false

Actual Result:

List instances that do contain item “xyz” get a return value of true and those that don’t get a return value of false.

Verdict:

Concrete List implementations in the Apache Commons Collections provide a working way of checking for presence of a single item in a list.

Test 12 - Presence of multiple items in lists

Inputs:

Instance(s)	Parameter(s)
- TreeList<>(Arrays.asList("xyz", "abc", "cbd")) - CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - NodeCachingLinkedList<>(Arrays.asList("xyz", "abc", "cbd"))	- String: “xyz” - String: “abc” - String: “test1”

Steps:

1. Use .contains() method by passing “xyz” and “abc” wrapped in Array.asList() as its parameter on non-empty list instances that do contain “xyz”
2. Check if return value of method is true
3. Use .contains() method by passing “xyz” and “test2” wrapped in Array.asList()its parameter on non-empty list instances that do contain “xyz” but not “test1”
4. Check if return value of second call is false

Expected Result:

1. First method call on non-empty instances should return true
2. Second method call on non-empty instances not containing “test1” instances should return false

Actual Result:

List instances that do contain items “xyz” and “abc” get a return value of true and those that

don’t have “test1” get a return value of false.

Verdict:

Concrete List implementations in the Apache Commons Collections provide a working way of checking for presence of multiple item in a list.

Test 13 - Finding index of an element in a list

Inputs:

Instance(s)	Parameter(s)
- TreeList<>(Arrays.asList("xyz", "abc", "cbd")) - CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - NodeCachingLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - TreeList<>() - CursorableLinkedList<>() - NodeCachingLinkedList<>()	- String: “xyz”

Steps:

1. Find index of “xyz” element in non-empty array by calling .indexOf() method on instances of lists and passing “xyz” as a parameter.
2. Verify that it returns the right index at which the element “xyz” is present in the instances of the list, i.e. 0.
3. Attempt to find index of “xyz” element in empty array by calling .indexOf() method on instances of lists and passing “xyz” as a parameter.
4. Verify that the return value of the second call is -1, meaning that no element “xyz” has been found in the list.

Expected Result:

1. First method call should return index 0 for non-empty instances
2. Second method call should return -1 to indicate that element “xyz” was not found on non-empty instances of lists.

Actual Result:

List instances with “xyz” present in the array return the given index at which the element is stored and empty list instances which do not contain “xyz” return -1.

Verdict:

Concrete List implementations in the Apache Commons Collections provide a working way of finding the index of an element in a list instance.

Test 14 - Retain selected elements

Inputs:

Instance(s)	Parameter(s)
- TreeList<>(Arrays.asList("xyz", "abc", "cbd")) - CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - NodeCachingLinkedList<>(Arrays.asList("xyz", "abc", "cbd")) - TreeList<>(Arrays.asList("xyz", "abc", "cbd", "abc")), - CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd", "abc")), - NodeCachingLinkedList<>(Arrays.asList("xyz", "abc", "cbd", "abc"))	- String: “xyz” - String: “cbd” - String: “willNotRetain”

Steps:

1. Create a list from instances above which only contains element “abc” by using the .removeAll() method to remove elements “xyz” and “cbd”
2. Call .retainAll() method on instances with duplicates of lists by passing “xyz” and “cbd” wrapped in Array.asList() as a parameter.
3. Compare lists created in step 1 and list obtained in step 2.
4. Call .retainAll() method on instances of lists by passing “willNotRetain” wrapped in Array.asList() as a parameter.
5. Compare list obtained in step 4 to generic empty list.

Expected Result:

1. First comparison between list containing elements “abc” and “abc” and list instances affected by .retainAll() should return true.

2. Second comparison between list containing no elements and list instances affected by .retainAll() should return true.

Actual Result:

Both assertions returned true, meaning that the list instances were able to keep only the requested elements inside of them.

Verdict:

Concrete List implementations in the Apache Commons Collections provide a working way of filtering the undesired values from all of its list instances.

Test 15 - Sorting elements of a list

Inputs:

Instance(s)	Parameter(s)
- TreeList<>(Arrays.asList("xyz", "abc", "cbd", "abc")), - CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd", "abc")), - NodeCachingLinkedList<>(Arrays.asList("xyz", "abc", "cbd", "abc"))	

Steps:

1. Create a list and insert items in the following alphabetical order: “abc”, “abc”, “cbd”, “xyz”
2. Call the .sort() method on instances of list by passing comparator String::compareToIgnoreCase as a parameter so as to sort the array alphabetically without taking into account the capitalization of the string elements.
3. Compare list created on step 1 and newly sorted array

Expected Result:

1. First list should match sorted list

Actual Result:

List has been sorted since sorted list matches comparison list with preordered elements.

Verdict:

Concrete List implementations in the Apache Commons Collections provide a working way of sorting elements.

Test 16 - Concurrent modification of cursor and iterator

Inputs:

Instance(s)	Parameter(s)
- CursorableLinkedList<>(Arrays.asList("xyz", "abc", "cbd", "abc"))	- String: “test”

Steps:

1. Initialize iterator and cursor from instance of CursorableLinkedList.
2. Check that element returned by the iterator when .next() is called on it is “xyz”
3. Remove element at iterator’s current position.
4. Check that element was removed by asserting that iterator.next() is equal to “abc”
5. Check that the cursor skips element removed by iterator by calling .next() on it
6. Call .add() method with “test” as parameter on the cursor.
7. Call cursor’s .previous() method and check that element “test” is returned by cursor’s .previous() method
8. Call cursor’s .next() method and check that element “test” is returned by cursor’s .next() method

Expected Result:

1. Check at step 2 should return “xyz”
2. Check at step 4 should return “abc”
3. Check at step 5 should return “abc”
4. Check at step 7 should return “test”
5. Check at step 8 should return “test”

Actual Result:

All of the assertions described above returned true. Therefore, both the cursor and the iterator of the CursorableLinkedList's instance exhibited expected behaviour, which is support of concurrent operations & modifications on the list through both its iterator and its cursor.

Verdict:

The Apache Commons Collections provides an implementation of List which indeed leverages the functionality of standard Java data structures by allowing concurrent modifications of both the list's cursor and iterator.

Test 17 - List implementation which supports faster insertion and removal than typical utils classes such as ArrayList and LinkedList

Inputs:

Instance(s)	Parameter(s)
- TreeList() - ArrayList() - LinkedList()	- Integers from 1 to 10000. - String: "test"

Steps:

1. Initialize variable holding start time.
2. Iterate from 0 to 10000. Inside the loop, add each element of the iteration inside of the TreeList instance with the .add() method.
3. After finalization of the loop, initialize variable holding the stop time.
4. Subtract the stop time from the start time to obtain the total duration of the addition operation for a TreeList instance.
5. Initialize variable holding start time.
6. Iterate from 0 to 10000. Inside the loop, add each element of the iteration inside of the ArrayList instance with the .add() method.
7. After finalization of the loop, initialize variable holding the stop time.
8. Subtract the stop time from the start time to obtain the total duration of the addition operation for an ArrayList instance.
9. Initialize variable holding start time.
10. Iterate from 0 to 10000. Inside the loop, add each element of the iteration inside of the LinkedList instance with the .add() method.
11. After finalization of the loop, initialize variable holding the stop time.
12. Subtract the stop time from the start time to obtain the total duration of the addition operation for a LinkedList instance.
13. Check that the duration of the addition operation for the TreeList is greater than that for

the ArrayList and the LinkedList.
14. Initialize variable holding start time.
15. Iterate from 0 to 10000. Inside the loop, insert an element of the iteration inside of the TreeList instance with the .add() method by passing every second value of “i” as the index parameter at which we wish to do the insertion and by passing the string “test” as the actual value we want to insert.
16. After finalization of the loop, initialize variable holding the stop time.
17. Subtract the stop time from the start time to obtain the total duration of the insertion operation for a TreeList instance.
18. Initialize variable holding start time.
19. Iterate from 0 to 10000. Inside the loop, insert an element of the iteration inside of the ArrayList instance with the .add() method by passing every second value of “i” as the index parameter at which we wish to do the insertion and by passing the string “test” as the actual value we want to insert.
20. After finalization of the loop, initialize variable holding the stop time.
21. Subtract the stop time from the start time to obtain the total duration of the insertion operation for an ArrayList instance.
22. Initialize variable holding start time.
23. Iterate from 0 to 10000. Inside the loop, insert an element of the iteration inside of the LinkedList instance with the .add() method by passing every second value of “i” as the index parameter at which we wish to do the insertion and by passing the string “test” as the actual value we want to insert.
24. After finalization of the loop, initialize variable holding the stop time.
25. Subtract the stop time from the start time to obtain the total duration of the insertion operation for a LinkedList instance.
26. Check that the duration of the insertion operations for the TreeList is lesser than that for the ArrayList and the LinkedList.
27. Initialize variable holding start time.
28. Iterate from 10000 to 0. Inside the loop, remove each element of the iteration inside of the TreeList instance with the .remove() method.
29. After finalization of the loop, initialize variable holding the stop time.
30. Subtract the stop time from the start time to obtain the total duration of the remove operation for a TreeList instance.
31. Initialize variable holding start time.
32. Iterate from 10000 to 0. Inside the loop, remove each element of the iteration inside of the ArrayList instance with the .remove() method.
33. After finalization of the loop, initialize variable holding the stop time.
34. Subtract the stop time from the start time to obtain the total duration of the remove operation for an ArrayList instance.
35. Initialize variable holding start time.
36. Iterate from 10000 to 0. Inside the loop, remove each element of the iteration inside of the LinkedList instance with the .remove() method.
37. After finalization of the loop, initialize variable holding the stop time.
38. Subtract the stop time from the start time to obtain the total duration of the remove operation for a LinkedList instance.
39. Check that the duration of the remove operation for the TreeList is lesser than that for the

ArrayList and the LinkedList.

Expected Result:

1. Duration of addition of element to the end of TreeList should be higher than that of ArrayList and LinkedList.
2. Duration of insertion of element at a given index of TreeList should be less than that of Array and LinkedList.
3. Duration of deletion of element at a given index of TreeList should be less than that of Array and LinkedList.

Actual Result:

Duration of addition of element to the end of TreeList is higher than that of ArrayList and LinkedList. Duration of insertion of element at a given index of TreeList is less than that of Array and LinkedList. Duration of deletion of element at a given index of TreeList is less than that of Array and LinkedList.

Verdict:

The Apache Commons Collections provides an implementation of the List interface which indeed leverages the functionality of standard Java data structures by providing an implementation which has faster deletion and insertion complexities.

Regular Iterators

Test 1 - Iteration through all items

Inputs:

Instance(s)	Parameter(s)
- ArrayIterator<>(new String[]{"abc", "cbd", "xyz"}) - CollatingIterator(String::compareToIgnoreCase, Arrays.asList("abc", "cbd", "xyz")) - LoopingIterator<>(Arrays.asList("abc", "cbd", "xyz")),	- String: “abc” - String: “cbd” - String: “xyz”

- LoopingListIterator<>(Arrays.asList("abc", "cbd", "xyz")), - ArrayListIterator(new String[]{"abc", "cbd", "xyz"}), - ObjectArrayIterator(new String[]{"abc", "cbd", "xyz"}), - UniqueFilterIterator<>(Arrays.asList("abc", "cbd", "xyz").iterator() - ArrayIterator<>(new String[]{}) - CollatingIterator(String::compareToIgnoreCase, Arrays.asList()) - LoopingIterator<>(Arrays.asList()) - LoopingListIterator<>(Arrays.asList()) - ArrayListIterator(new String[]{}) - ObjectArrayIterator(new String[]{}) - UniqueFilterIterator<>(Arrays.asList().iterator()	

Steps:

1. Check that iterator's instance's pointer points to "abc" after first call of .next().
2. Check that iterator's instance's pointer points to "cbd" after second call of .next().
3. Check that iterator's instance's pointer points to "xyz" after third call of .next().
4. Attempt to iterate to next element of non-looping iterators with a fourth call of .next().
5. Attempt to iterate to next element of iterators' instances holding empty instances of collections with a call to .next().

Expected Result:

1. Iterator should point to first element "abc" after first .next() call.
2. Iterator should point to second element "cbd" after second .next() call.
3. Iterator should point to third element "xyz" after third .next() call.
4. NoSuchElementException should be thrown for non-looping instances of iterators.
5. NoSuchElementException should be thrown for empty instances of iterators.

Actual Result:

The pointer always pointed to the right element after each subsequent call to .next() on the iterator's instance. NoSuchElementException was thrown after attempting to call .next() on the fourth element of non-looping iterators. NoSuchElementException was thrown after attempting to call .next() on empty instances of iterators.

Verdict:

The Apache Commons Collections provides multiple implementations of Iterators which offer an efficient way of iterating through all of a collection's elements.

Test 2 - Looping iteration through all items

Inputs:

Instance(s)	Parameter(s)
- LoopingIterator<>(Arrays.asList("abc", "cbd", "xyz")), - LoopingListIterator<>(Arrays.asList("abc", "cbd", "xyz")),	- String: “abc” - String: “cbd” - String: “xyz”

Steps:

1. Check that iterator’s instance’s pointer points to “abc” after first call of .next().
2. Check that iterator’s instance’s pointer points to “cbd” after second call of .next().
3. Check that iterator’s instance’s pointer points to “xyz” after third call of .next().
4. Check that iterator’s instance’s pointer points back to “abc” after fourth call of .next().

Expected Result:

1. Iterator should point to first element “abc” after first .next() call.
2. Iterator should point to second element “cbd” after second .next() call.
3. Iterator should point to third element “xyz” after third .next() call.
4. Iterator should point back to first element “abc” after fifth .next() call.

Actual Result:

The pointer always pointed to the right element after each subsequent call to .next() on the looping iterator’s instance.

Verdict:

The Apache Commons Collections provides multiple implementations of looping Iterators which offer an efficient way of iterating through all of a collection’s elements and come back to the beginning of the collection again.

Test 3 - Check for presence of next item in iteration

Inputs:

Instance(s)	Parameter(s)
- ArrayIterator<>(new String[]{"abc", "cbd", "xyz"}) - CollatingIterator(String::compareToIgnoreCase, Arrays.asList("abc", "cbd", "xyz")) - LoopingIterator<>(Arrays.asList("abc", "cbd", "xyz")), - LoopingListIterator<>(Arrays.asList("abc", "cbd", "xyz")), - ArrayListIterator(new String[]{"abc", "cbd", "xyz"}), - ObjectArrayIterator(new String[]{"abc", "cbd", "xyz"}), - UniqueFilterIterator<>(Arrays.asList("abc", "cbd", "xyz").iterator()	- String: “abc” - String: “cbd” - String: “xyz”

Steps:

1. Check if the iteration has more elements by calling .hasNext() on the iterator’s instance.
2. Move iterator’s pointer with a call of .next().
3. Check if the iteration has more elements by calling .hasNext() on the iterator’s instance.
4. Move iterator’s pointer with a call of .next().
5. Check if the iteration has more elements by calling .hasNext() on the iterator’s instance.
6. Move iterator’s pointer with a call of .next().
7. Check if the iteration has more elements by calling .hasNext() on the iterator’s instance.

Expected Result:

1. First call to .hasNext() should return true.
2. Second call to .hasNext() should return true.
3. Third call to .hasNext() should return true.
4. Fourth call to .hasNext() should return false.

Actual Result:

Iterator indicated the presence of more elements in the collection at the beginning of the test. Iterator indicated the presence of more elements in the collection after being moved by one element. Iterator indicated the presence of more elements in the collection after being moved by two elements. Iterator indicated no presence of more elements in the collection after being moved by three elements.

Verdict:

The Apache Commons Collections provides multiple implementations of looping Iterators which offer an efficient way of checking for the presence of remaining elements at any given point of an iteration.

Test 4 - Remove current item of iterator

Inputs:

Instance(s)	Parameter(s)
- ArrayIterator<>(new String[]{"abc", "cbd", "xyz"}) - CollatingIterator(String::compareToIgnoreCase, Arrays.asList("abc", "cbd", "xyz")) - LoopingListIterator<>(Arrays.asList("abc", "cbd", "xyz")), - ArrayListIterator(new String[]{"abc", "cbd", "xyz"}), - ObjectArrayIterator(new String[]{"abc", "cbd", "xyz"}), - UniqueFilterIterator<>(Arrays.asList("abc", "cbd", "xyz").iterator() - ZippingIterator(new LinkedList<>(Arrays.asList("abc", "cbd", "xyz")).listIterator(), new ArrayList<>(Arrays.asList("abc", "cbd", "xyz")).listIterator()); - LoopingIterator<>(new LinkedList<>(Arrays.asList("abc", "cbd", "xyz"))	

Steps:

1. Move iterator's pointer to next element by calling .next() method on iterator.
2. Attempt to call .remove() method on iterator instances which are not ZippingIterator or Looping Iterator at the element to which the pointer is currently pointing to.
3. Call .next() on ZippingIterator instance.
4. Call .remove() on ZippingIterator instance.
5. Call .next() on LoopingIterator instance.
6. Call .remove() on LoopingIterator instance.
7. Check that list through which the ZippingIterator was iterating has decreased in size by 1.

8. Check that list through which the LoopingIterator was iterating has decreased in size by 1.
9. Attempt to call .remove() again on ZippingIterator instance.

Expected Result:

1. Should throw an UnsupportedOperationException exception when calling .remove() on instances of Iterators which are not ZippingIterator or LoopingIterator.
2. Should return 2 when checking size of list through which the ZippingIterator was iterating.
3. Should return 2 when checking size of list through which the LoopingIterator was iterating.
4. Should throw IllegalStateException after second call to .remove() on ZippingIterator instance.

Actual Result:

A UnsupportedOperationException is thrown when calling the instances of Iterators which are not ZippingIterator or LoopingIterator. Item “abc” was successfully removed from both the ZippingIterator and LoopingIterator instances. An IllegalStateException was thrown after second call to .remove() on ZippingIterator instance.

Verdict:

The Apache Commons Collections provides certain implementations of Iterators which support removal of elements at the position to which the pointer is pointing to, albeit not for all of its implementations of iterators.

Test 5 - Iterator providing iteration through unique items only

Inputs:

Instance(s)	Parameter(s)
- UniqueFilterIterator<>(Arrays.asList("abc", "cbd", "xyz", "abc").iterator())	

Steps:

1. Call .next() method on iterator for first time.
2. Call .next() method on iterator for second time.

3. Call .next() method on iterator for third time.
4. Attempt to call .next() method on iterator for fourth time.

Expected Result:

1. Should throw exception NoSuchElementException exception.

Actual Result:

A NoSuchElementException was thrown to indicate that no element was present after third iteration, meaning that “abc” was filtered from the original collection.

Verdict:

The Apache Commons Collections provides an implementations of Iterators which supports a working iteration through only the unique items inside of a collection.

Test 6 - Iterator providing alternate iteration through multiple iterators

Inputs:

Instance(s)	Parameter(s)
- ZippingIterator(new LinkedList<>(Arrays.asList("abc", "cbd", "xyz")).listIterator(), new ArrayList<>(Arrays.asList("abc", "cbd", "xyz")).listIterator());	- String: “abc”

Steps:

1. Check that return value from first call to zippingIterator.next() is equal to “abc”
2. Check that return value from second call to zippingIterator.next() is equal to “abc”

Expected Result:

1. Return value from first call should be “abc”
2. Return value from second call should be “abc”

Actual Result:

Both calls to .next() indeed return “abc,” meaning that the iterator does jump between both iterators that were passed to its constructor during its instantiation after each call to .next().

Verdict:

The Apache Commons Collections provides an implementations of Iterators which supports alternate iteration between different iterators, leveraging once again the functionalities provided by Java’s standard Utils library.

Map Iterators

Test 1 – Test that there is no next element in an empty map.

Inputs:

Instances	Parameter(s)

– ListOrderedMap()
– LinkedMap()
– TreeBidiMap()
– ListOrderedMap()
{{
put(“ONE” , 1),
put(“TWO” , 2),
}}
– LinkedMap()
{
put(“ONE” , 1),
put(“TWO” , 2),
}}
– TreeBidiMap()
{
put(“ONE” , 1),
put(“TWO” , 2),
}}
– ListOrderedMap()
	– Iterator iterator = map.mapIterator()

{{
put(“ONE” , 1),
}}
– LinkedMap()
{
put(“ONE” , 1),
}}
– TreeBidiMap()
{
put(“ONE” , 1),
}}	

Steps

1. Create a new mapIterator on the iterable maps.
2. Call the next() method to iterate through the maps until it calls the method on the last element of the maps.

Expected Result:

Should throw an exception.

Actual Result:

When next() is called on the last element of the iterable map, it finds that there is no next element. The exception is thrown.

Verdict:

The next() method for iterators in the Apache Commons Collections handles well Exception on non-existing elements in maps.

Test 2 - Test that getKey() returns the correct key.

Inputs:

Instances	Parameter(s)
– ListOrderedMap() – LinkedMap() – TreeBidiMap() – ListOrderedMap() {{	– Iterator iterator = map.mapIterator()

put("ONE" , 1),
put("TWO" , 2),
}}
– LinkedMap()
{
put("ONE" , 1),
put("TWO" , 2),
}}
– TreeBidiMap()
{
put("ONE" , 1),
put("TWO" , 2),
}}
– ListOrderedMap()
{{
put("ONE" , 1),
}}
– LinkedMap()
{
put("ONE" , 1),
}}
– TreeBidiMap()
{
put("ONE" , 1),
}}	

Steps:

1. Create a new mapIterator on the iterable maps.
2. The next() method to get to the first element of the maps.
3. Call assertEquals() on the expected key and the actual key by calling iterator.getKey().

Expected *Result:*

1. Should get the appropriate key assigned to the value the iterator is located at when the maps are filled with elements.
2. Should throw an exception when the map is empty.

Actual *Result:*

1. getKey() successfully gets the appropriate key, in this case 1, assigned to the value "ONE" the iterator is located at when the map is filled with elements.
2. getKey() throws a NoSuchElementException.

Verdict:

The getKey() method for iterators in the Apache Commons Collection returns the correct key when available and throws an exception otherwise.

Test 3 - Test that next() actually iterates through the maps.

Inputs:

Instances	Parameter(s)

– ListOrderedMap()
– LinkedMap()
– TreeBidiMap()
– ListOrderedMap()
{{
put(“ONE” , 1),
put(“TWO” , 2),
}}
– LinkedMap()
{
put(“ONE” , 1),
put(“TWO” , 2),
}}
– TreeBidiMap()
{
put(“ONE” , 1),
put(“TWO” , 2),
}}
– ListOrderedMap()
{{
put(“ONE” , 1),
}}
– LinkedMap()
{
put(“ONE” , 1),
}}
– TreeBidiMap()
{
put(“ONE” , 1),
}}
	– Iterator iterator = map.mapIterator()
	– Object key

Steps:

1. Create a new mapIterator on the iterable maps.
2. Call the next() method to get to the first element of the maps and assign the value to an object variable key.
3. Call assertEquals() on the expected key and the actual key variable.
4. Call assertEquals() on the expected value and the actual value by calling iterator.getValue().

Expected *Result:*

1. Should get the appropriate key assigned to the value the iterator is located at when the maps are filled with elements.
2. Should get the appropriate value assigned to the key located at the iterator's position.

Actual *Result:*

1. Iterator.next() successfully stores the appropriate key in the created variable, in this case "ONE".
2. getValue() successfully returns the value 1.

Verdict:

The next() method iterates as expected through the maps using iterators from the Apache Commons Collections.

Test 4 - Test that hasNext() called on an empty map or on the last element of a map.

Inputs:

Instances	Parameter(s)

– ListOrderedMap()
– LinkedMap()
– TreeBidiMap()
– ListOrderedMap()
{{
put("ONE" , 1),
put("TWO" , 2),
}}
– LinkedMap()
{
put("ONE" , 1),
put("TWO" , 2),
}}
	– Iterator iterator = map.mapIterator()

– TreeBidiMap()
{
put(“ONE” , 1),
put(“TWO” , 2),
}}
– ListOrderedMap()
{{
put(“ONE” , 1),
}}
– LinkedMap()
{
put(“ONE” , 1),
}}
– TreeBidiMap()
{
put(“ONE” , 1),
}}	

Steps:

1. Create a new mapIterator on the iterable maps.
2. Call the next() method several times to get to the last element of the maps.
3. Call assertFalse() on the last element of the maps.

Expected Result:

When the iterator is on the last element of the map and the hasNext() method is called the expected returned value should be false.

Actual Result:

assertFalse(iterator.hasNext()) returns the expected false value.

Verdict:

The hasNext() method handles properly cases when the iterator is on the last element of the maps.

Test 5 - Test that hasNext() called when not on the last element of a map.

Inputs:

Instances	Parameter(s)

– ListOrderedMap()
– LinkedMap()
– TreeBidiMap()
– ListOrderedMap()
{{
put(“ONE” , 1),
put(“TWO” , 2),
}}
– LinkedMap()
{
put(“ONE” , 1),
put(“TWO” , 2),
}}
– TreeBidiMap()
{
put(“ONE” , 1),
put(“TWO” , 2),
}}
– ListOrderedMap()
{{
put(“ONE” , 1),
}}
– LinkedMap()
{
put(“ONE” , 1),
}}
– TreeBidiMap()
{
put(“ONE” , 1),
}}
	– Iterator iterator = map.mapIterator()

Steps:

1. Create a new mapIterator on the iterable maps.
2. Call assertTrue(iterator.hasNext()).

Expected Result:

1. When the iterator is not on the last element of the map and the hasNext() method is called the expected returned value should be true.

Actual Result:

assertTrue(iterator.hasNext()) returns the expected true value.

Verdict:

The hasNext() method works as expected when the iterator is not on the last element of the maps.

Test 6 - Test that remove() actually removes an element when the map is not empty and throws an exception when it is empty

Inputs:

Instances	Parameter(s)

– ListOrderedMap()
– LinkedMap()
– TreeBidiMap()
– ListOrderedMap()
{{
put(“ONE” , 1),
put(“TWO” , 2),
}}
– LinkedMap()
{
put(“ONE” , 1),
put(“TWO” , 2),
}}
– TreeBidiMap()
{
put(“ONE” , 1),
put(“TWO” , 2),
}}
– ListOrderedMap()
{{
put(“ONE” , 1),
}}
– LinkedMap()
{
put(“ONE” , 1),
}}
– TreeBidiMap()
{
put(“ONE” , 1),
}}
	– Iterator iterator = map.mapIterator()

Steps:

1. Create a new mapIterator on the iterable maps.
2. Call iterator.next() in order to position the iterator on the first element of the map
3. Call iterator.remove() to remove the first element of the map
4. Call assertEquals(expectedMapSize , map.size()).

Expected Result:

1. We expect that: new size = previous size – 1 if the map is not empty
2. We expect to have an exception thrown if the map is empty

Actual Result:

1. The actual result shows that a map that contained one element is now empty and that a map containing multiple elements has now one less element.
2. Also, we were not able to remove an element from an empty map and an IllegalStateException was thrown.

Verdict:

The remove() method works as expected with map iterators from the Apache Commons Collections.

Test 7 - Test that setValue() changes the value associated to a specific key

Inputs:

Instances	Parameter(s)

– ListOrderedMap()
– LinkedMap()
– TreeBidiMap()
– ListOrderedMap()
{{
put(“ONE” , 1),
put(“TWO” , 2),
}}
– LinkedMap()
{
put(“ONE” , 1),
put(“TWO” , 2),
}}
– TreeBidiMap()
	– Iterator iterator = map.mapIterator()

{
put("ONE" , 1),
put("TWO" , 2),
}}
– ListOrderedMap()
{{
put("ONE" , 1),
}}
– LinkedMap()
{
put("ONE" , 1),
}}
– TreeBidiMap()
{
put("ONE" , 1),
}}	

Steps:

1. Create a new mapIterator on the iterable maps.
2. Call iterator.next() to put the iterator on the first element of the map.
3. Call iterator.setValue(value).
4. Call assertEquals(value , iterator.getValue()).

Expected *Result:*

1. We expect that when setValue(value) is called on a non-empty map, the value associated with specific the key the iterator is located at will get change to value.
2. We also expect that an exception is thrown when the map is empty or when the setValue() method is called on a TreeBidiMap

Actual *Result:*

1. The value of the first element was changed to value = 4 on non-empty maps
2. A NoSuchElementException was thrown when the map was empty
3. An UnsupportedOperationException was thrown when setValue() was called on a TreeBidiMap

Verdict:

setValue() sets the value correctly on iterable maps. However, on TreeBidiMaps the method is not supported.

Ordered Maps

Test 1 - Test firstKey method on empty maps.

Inputs:

Instances	Parameter(s)
- ListOrderedMap() - LinkedMap() - ListOrderedMap()	- OrderedMap map

Steps:

Call map.firstKey() on an empty map.

Expected Result:

Exception is thrown.

Actual Result:

A NoSuchElementException is thrown.

Verdict:

firstKey() handles the cases when the map is empty.

Test 2 - Test firstKey method on nonempty maps.

Inputs:

Instances	Parameter(s)
- ListOrderedMap() {{ put(“ONE” , 1), }} - LinkedMap() {{	- OrderedMap map - Object key

put(“ONE” , 1), }} - ListOrderedMap() {{ put(“ONE” , 1), put(“TWO” , 2), }} - LinkedMap() {{ put(“ONE” , 1), put(“TWO” , 2), }}	

Steps:

1. Assign map.firstKey() to the Object key.
2. Call assertEquals(“ONE” , key).

Expected Result:

We expect the assertion to return true.

Actual Result:

Key = “ONE” and the assertion returns true.

Verdict:

The firstKey method of the Apache Commons Collections returns the first element of a map as expected.

Test 3 - Test lastKey method on empty maps.

Inputs:

Instances	Parameter(s)
- ListOrderedMap() - LinkedMap() - ListOrderedMap()	- OrderedMap map

Steps:

Call map.lastKey() on an empty map.

Expected Result:

Exception is thrown.

Actual Result:

A NoSuchElementException is thrown.

Verdict:

lastKey() handles the cases when the map is empty.

Test 4 - Test lastKey method on nonempty maps.

Inputs:

Instances	Parameter(s)
- ListOrderedMap() {{ put(“ONE” , 1), }} - LinkedMap() {{ put(“ONE” , 1), }} - ListOrderedMap() {{ put(“ONE” , 1), put(“TWO” , 2), }} - LinkedMap() {{ put(“ONE” , 1), put(“TWO” , 2), }}	- OrderedMap map - Object key

Steps:

1. Assign map.lastKey() to the Object key.

2. Call assertEquals(“ONE” , key) for the map of size 1 and call assertEquals(“TWO” , key) for the map of size 2.

Expected Result:

We expect the assertion to return true.

Actual Result:

1. Key = “ONE” and the assertion returns true in the case of the map of size 1.
2. Key = “TWO” and the assertion returns true in the case of the map of size 2.

Verdict:

The lastKey method of the Apache Commons Collections returns the first element of a map as expected.

Test 5 - Test nextKey method on nonempty maps.

Inputs:

Instances	Parameter(s)
- ListOrderedMap() {{ put(“ONE” , 1), put(“TWO” , 2), }} - LinkedMap() {{ put(“ONE” , 1), put(“TWO” , 2), }}	- OrderedMap map - Object key

Steps:

1. Assign map.nextKey(“ONE”) to the Object key.
2. Call assertEquals(“TWO” , key).

Expected Result:

We expect the assertion to return true.

Actual Result:

Key = “TWO” and the assertion returns true.

Verdict:

The nextKey method of the Apache Commons Collections returns the next element of a map from a specified index.

Test 6 - Test previousKey method on nonempty maps.

Inputs:

Instances	Parameter(s)
- ListOrderedMap() {{ put(“ONE” , 1), put(“TWO” , 2), }} - LinkedMap() {{ put(“ONE” , 1), put(“TWO” , 2), }}	- OrderedMap map - Object key

Steps:

1. Assign map.previousKey(“TWO”) to the Object key.
2. Call assertEquals(“ONE” , key).

Expected Result:

We expect the assertion to return true.

Actual Result:

Key = “ONE” and the assertion returns true.

Verdict:

The previousKey method of the Apache Commons Collections returns the next element of a map from a specified index.

Test 7 - Test adding elements to a map.

Inputs:

Instances	Parameter(s)
- ListOrderedMap()	- ListOrderedMap map - (“ONE” , 1)

Steps:

1. Initialize the map: map = new ListOrderedMap().
2. Add the element (“ONE” , 1) to the map by calling map.put(“ONE” , 1).
3. Check that the size of the map has increased by 1 (or is equal to 1 since the size was previously 0) by calling assertEquals(1 , map.size()).
4. Check that the value of the first element of the map is equal to 1 by calling assertEquals(1 , map.getValue(0)).
5. Check that the key is equal to “ONE” by calling assertEquals(“ONE” , map.get(0)).

Expected Result:

We expect all assertions to return true. This would validate that the element has properly been added to the map.

Actual Result:

All assertions do return true which confirms our expectation.

Verdict:

Adding an element to a map works perfectly by using put(key , value).

Test 8 - Test adding elements to a map at a specified index.

Inputs:

Instances	Parameter(s)

- ListOrderedMap()	- ListOrderedMap map - (“TWO” , 2) - (“THREE” , 3) - (“ONE” , 1)

Steps:

1. Initialize the map: map = new ListOrderedMap().
2. Add the element (“TWO” , 2) to the map by calling map.put(“TWO” , 2).
3. Add the element (“THREE” , 3) to the map by calling map.put(“THREE” , 3).
4. Add the element (“ONE” , 1) to the map by calling map.put(0 , “ONE” , 1).
5. Check that the size of the map is equal to 3 by calling assertEquals(3 , map.size()).
6. Check that the value of the first element of the map is equal to 1 by calling assertEquals(1 , map.getValue(0)).
7. Check that the key is equal to “ONE” by calling assertEquals(“ONE” , map.get(0)).

Expected Result:

We expect all assertions to return true. This would validate that the element has properly been added to the map at the specified index

Actual Result:

All assertions do return true which confirms our expectation.

Verdict:

Adding an indexed element to a map works perfectly by using put(index , key , value).

Test 9 - Test adding multiple elements to a map all at once.

Inputs:

Instances	Parameter(s)
- ListOrderedMap()	- ListOrderedMap map - ListOrderedMap add - (“ONE” , 1) - (“TWO” , 2) - (“THREE” , 3)

Steps:

1. Initialize the map: map = new ListOrderedMap().
2. Initialize the map: add = new ListOrderedMap().
3. Add the element (“ONE” , 1) to the map by calling add.put(“ONE” , 1).
4. Add the element (“TWO” , 2) to the map by calling add.put(“TWO” , 2).
5. Add the element (“THREE” , 3) to the map by calling add.put(“THREE” , 3).
6. Add all elements to the second map by calling map.putAll(add).
7. Check that the size of the map is equal to 3 by calling assertEquals(3 , map.size()).
8. Check that the value of the first element of the map is equal to 1 by calling assertEquals(1 , map.getValue(0)).
9. Check that the key is equal to “ONE” by calling assertEquals(“ONE” , map.get(0)).
10. Check that the value of the second element of the map is equal to 2 by calling assertEquals(2 , map.getValue(1)).
11. Check that the key is equal to “TWO” by calling assertEquals(“TWO” , map.get(1)).
12. Check that the value of the third element of the map is equal to 3 by calling assertEquals(3 , map.getValue(2)).
13. Check that the key is equal to “THREE” by calling assertEquals(“THREE” , map.get(2)).

Expected Result:

We expect all assertions to return true. This would validate that all the elements have properly been added to the map all at once.

Actual Result:

All assertions do return true which confirms our expectation.

Verdict:

Adding an indexed element to a map works perfectly by using put(index , key , value).

Test 10 - Test removing an element from a map at a specified index.

Inputs:

Instances	Parameter(s)
- ListOrderedMap()	- ListOrderedMap map - (“ONE” , 1) - (“TWO” , 2)

Steps:

1. Initialize the map: map = new ListOrderedMap().
2. Add the element (“ONE” , 1) to the map by calling map.put(“ONE” , 1).
3. Add the element (“TWO” , 2) to the map by calling map.put(“TWO” , 2).
4. Remove the element (“TWO” , 2) from the map by calling map.remove(1).
5. Check that the size of the map is equal to 1 by calling assertEquals(1 , map.size()).
6. Check that the element (“ONE” , 1) is still in the map by calling assertEquals(1 , map.getValue(0)), and assertEquals(“ONE” , map.get(0)).

Expected Result:

We expect all assertions to return true. This would validate that the correct element has been removed from the map

Actual Result:

All assertions do return true which confirms our expectation.

Verdict:

Removing an indexed element from a map works perfectly by using remove(index).

Test 11 - Test removing an element from an empty map at a specified index.

Inputs:

Instances	Parameter(s)
- ListOrderedMap()	- ListOrderedMap map

Steps:

1. Initialize the map: map = new ListOrderedMap().
2. Try removing an element from the map by calling map.remove(0).

Expected Result:

We expect an exception to be thrown since such element does not exist

Actual Result:

An IndexOutOfBoundsException was thrown

Verdict:

Removing an indexed element from an empty map is handled correctly by throwing an IndexOutOfBoundsException.

Test 12 - Test reading a key from an empty map at a specified index.

Inputs:

Instances	Parameter(s)
- ListOrderedMap()	- ListOrderedMap map

Steps:

1. Initialize the map: map = new ListOrderedMap().
2. Try reading a key from the map by calling map.get(0).

Expected Result:

We expect an exception to be thrown since such element does not exist

Actual Result:

An IndexOutOfBoundsException was thrown.

Verdict:

Reading a key from an empty map is handled correctly by throwing an IndexOutOfBoundsException.

Test 13 - Test reading a key from a map at a specified index.

Inputs:

Instances	Parameter(s)
- ListOrderedMap()	- ListOrderedMap map - (“ONE” , 1)

Steps:

1. Initialize the map: map = new ListOrderedMap().
2. Add element (“ONE” , 1) to the map by calling map.put(“ONE” , 1).
3. Read the element (“ONE” , 1) from the map by calling map.get(0).
4. Check that the element has correctly been read by calling assertEquals(“ONE” , map.get(0)

Expected Result:

We expect the assertion to return true.

Actual Result:

The assertion is true.

Verdict:

Reading an value from a map works correctly.

Test 14 - Test reading a key from a map at an invalid index.

Inputs:

Instances	Parameter(s)
- ListOrderedMap()	- ListOrderedMap map

Steps:

3. Initialize the map: map = new ListOrderedMap().
4. Add one element to the map by calling map.put(“ONE” , 1).
5. Try reading a key from the map by calling map.get(1).

Expected Result:

We expect an exception to be thrown since such element does not exist

Actual Result:

An IndexOutOfBoundsException was thrown.

Verdict:

Reading a key from a map is handled correctly by throwing an IndexOutOfBoundsException.

Test 15 - Test reading a value from an empty map at a specified index.

Inputs:

Instances	Parameter(s)
- ListOrderedMap()	- ListOrderedMap map

Steps:

1. Initialize the map: map = new ListOrderedMap().
2. Try reading a value from the map by calling map.getValue(0).

Expected Result:

We expect an exception to be thrown since such element does not exist

Actual Result:

An IndexOutOfBoundsException was thrown.

Verdict:

Reading a value from an empty map is handled correctly by throwing an IndexOutOfBoundsException.

Test 16 - Test reading a value from a map at a specified index.

Inputs:

Instances	Parameter(s)
- ListOrderedMap()	- ListOrderedMap map - (“ONE” , 1)

Steps:

1. Initialize the map: map = new ListOrderedMap().
2. Add element (“ONE” , 1) to the map by calling map.put(“ONE” , 1).
3. Read the element (“ONE” , 1) from the map by calling map.getValue(0).
4. Check that the value has correctly been read by calling assertEquals(“ONE” , map.getValue(0)

Expected Result:

We expect the assertion to return true.

Actual Result:

The assertion is true.

Verdict:

Reading a value from a map works correctly.

Test 17 - Test reading a value from a map at an invalid index.

Inputs:

Instances	Parameter(s)
- ListOrderedMap()	- ListOrderedMap map

Steps:

1. Initialize the map: map = new ListOrderedMap().
2. Add one element to the map by calling map.put(“ONE” , 1).
3. Try reading a key from the map by calling map.getValue(1).

Expected Result:

We expect an exception to be thrown since such element does not exist

Actual Result:

An IndexOutOfBoundsException was thrown.

Verdict:

Reading a value from a map is handled correctly by throwing an IndexOutOfBoundsException.

Test 18 - Test modifying a value from an empty map at a specified index.

Inputs:

Instances	Parameter(s)
- ListOrderedMap()	- ListOrderedMap map

Steps:

1. Initialize the map: map = new ListOrderedMap().
2. Try modifying a value from the map by calling map.setValue(0 , 0).

Expected Result:

We expect an exception to be thrown since such element does not exist

Actual Result:

An IndexOutOfBoundsException was thrown.

Verdict:

Modifying a value from an empty map is handled correctly by throwing an IndexOutOfBoundsException.

Test 19 - Test modifying a value from a map at a specified index.

Inputs:

Instances	Parameter(s)
- ListOrderedMap()	- ListOrderedMap map - (“ONE” , 1)

Steps:

5. Initialize the map: map = new ListOrderedMap().
6. Add element (“ONE” , 1) to the map by calling map.put(“ONE” , 1).
7. Modify the element (“ONE” , 1) from the map by calling map.setValue(0 , 0).
8. Check that the value has correctly been modified by calling assertEquals(0 , map.getValue(0)

Expected Result:

We expect the assertion to return true.

Actual Result:

The assertion is true.

Verdict:

Modifying a value from a map works correctly.

Adapters

Test 1: Test if the adapter was successful in rendering an enumeration from an iterator

Instance(s)	Parameter(s)
● IteratorEnumeration(Iterator iterator)	· Iterator iterator · Enumeration enumeration

Steps:

1. *Create an iterator and instantiate it with values*
2. *Pass the iterator into the* IteratorEnumeration method
3. *Check if the rendered object is indeed an enumeration with getClass() method*

Expected Result:

1. We expect to get an enumeration object from IteratorEnumeration,

Actual Result:

1. We got an enumeration object from the IteratorEnumeration

Verdict:

The IteratorEnumeration()method works

Test 2: Test if the adapter was successful in rendering an ietrable and from an iterator

Instance(s)	Parameter(s)
● IteratorIterable(Iterator iterator)	· Iterator iterator · Iterable iterable

Steps:

1. *Create an iterator and instantiate it with values*
2. *Pass the iterator into the* IteratorIterable method
3. *Check if the rendered object is indeed an iterable with getClass() method*

Expected Result:

We expect to get the proper an *iterable* object from IteratorIterable,

Actual Result:

We got an iterable object from the IteratorIterable

Verdict:

The IteratorIterable() method works

Test 3: Test if the adapter was successful in rendering an iterator from an enumeration

Instance(s)	Parameter(s)
· EnumerationIterator(Enumeration enumeration)	· Enumeration enumeration · Iterator iterator

Steps:

1. *Create an enumeration and instantiate it with values*
2. *Pass the enumeration into the* EnumerationIterator method
3. *Check if the rendered object is indeed an iterator with getClass() method*

Expected Result:

We expect to get an *iterator* object from EnumerationIterator,

Actual Result:

We got an *iterator* object from the EnumerationIterator

Verdict:

The EnumerationIterator () method works

Decorators

Test 1: add elements to decorators

Instance(s)	Parameter(s)
● AbstractCollectionDecorator(Collection coll) ● AbstractBagDecorator(Bag bag) ● AbstractSetDecorator(Set set) ● AbstractIteratorDecorator(Iterator iterator) ● AbstractListDecorator(List list) ● AbstractQueueDecorator(Queue queue)	· Collection coll · Bag bag · Set set · Iterator iterator · List list · Queue queue

Steps:

1. *Store the current size of the decorator with size() method*
2. *Add one element for each decorator with add() method*
3. *Check if the current decorator size is bigger than its previous size*
4. *Use decorated() to get the object that's being decorated*
5. *Check if the object has been decorated properly*

Expected Result:

1. We expect to have the sizeof the decorator to increase by one
2. We expect to have an object that has been changed correspondingly to the added behavior of the decorator

Actual Result:
Size increased by one and object modified according to the added behavior

Verdict:
Decorators alter each object as it is added to their corresponding collection

BAG

Test 0 – Create Bag instances

Inputs:

Instance(s)	Parameter(s)
	Arrays.asList("xyz", "abc", "cbd")

Steps:

1. Create four instances of different Bag implementations with creator (new Bag(Collection<?> coll) & new Bag()) both with collection as parameter and without any parameters and save instances in variables
   - Bag<> (Arrays.asList("xyz", "abc", "cbd"))
   - AbstractMapBag<>(Arrays.asList("xyz", "abc", "cbd"))
   - Hashbag<>(Arrays.asList("xyz", "abc", "cbd"))
   - SortedBag<>(Arrays.asList("xyz", "abc", "cbd"))
   - Bag<>()
   - AbstractMapBag<>()
   - Hashbag<>()
   - SortedBag<>()
2. Check that variables are not null

Expected Result:

Variables are not null

Actual Result:

Variables are not null

Verdict:

Concrete Bag implementations in the Apache Commons Collections work for creation of both empty and non-empty instances of Bag, AbstractMapBag, Hashbag and SortedBag.

Test 1 - Regular addition to Bag instance

Inputs:

Instance(s)	Parameter(s)
• Bag<> (Arrays.asList("xyz", "abc", "cbd"))	• String: “test”

• AbstractMapBag<>(Arrays.asList("xyz", "abc", "cbd")) • Hashbag<>(Arrays.asList("xyz", "abc", "cbd")) • SortedBag<>(Arrays.asList("xyz", "abc", "cbd")) • Bag<>() • AbstractMapBag<>() • Hashbag<>() • SortedBag<>()	

Steps:

1. Store the size of the bag in a buffer variable.
2. Add "test" string element to the bag instance using the .add() method provided by the common Bag interface.
3. Compare new size to one in buffer.

Expected Result:

Returned true
Size of Bag instance should have increased by one.

Actual Result:

Returned true
Buffer storing size of Bag instance prior to the addition plus one is equal to the size of Bag after new element's addition.

Verdict:

Concrete Bag implementations in the Apache Commons Collections work for addition of single items for both empty and non-empty instances of Bag, AbstractMapBag, Hashbag and SortedBag.

Test 2 – Addition of n copies of an object to Bag instance

Inputs:

Instance(s)	Parameter(s)

• Bag<> (Arrays.asList("xyz", "abc", "cbd")) • AbstractMapBag<>(Arrays.asList("xyz", "abc", "cbd")) • Hashbag<>(Arrays.asList("xyz", "abc", "cbd")) • SortedBag<>(Arrays.asList("xyz", "abc", "cbd")) • Bag<>() • AbstractMapBag<>() • Hashbag<>() • SortedBag<>()	• String: “test” • Int: 4

Steps:

1. Store the size of the bag in a buffer variable.
2. Add 4 instances of the “test” string element to the bag instance using the secondary .add() method provided by the common Bag interface.
3. Check Boolean value returned by method call
4. Compare new size to one in buffer.

Expected Result:

Size of Bag instance should have increased by four.
Four instances of the “test” string should be present in bag

Actual Result:

Buffer storing size of Bag instance prior to the addition plus four is equal to the size of Bag after new element’s addition.
Four instances of “test” are found

Verdict:

Concrete Bag implementations in the Apache Commons Collections work for addition of several instances of same object for both empty and non-empty instances of Bag, AbstractMapBag, Hashbag and SortedBag.

Test 3 – Check if bag contains all elements in specified collection

Inputs:

Instance(s)	Parameter(s)

• Bag<> (Arrays.asList("xyz", "abc", "cbd")) • AbstractMapBag<>(Arrays.asList("xyz", "abc", "cbd")) • Hashbag<>(Arrays.asList("xyz", "abc", "cbd")) • SortedBag<>(Arrays.asList("xyz", "abc", "cbd")) • Bag<>((Arrays.asList("xyz")) • AbstractMapBag<>((Arrays.asList("xyz")) • Hashbag<>((Arrays.asList("xyz")) • SortedBag<>((Arrays.asList("xyz")) • Bag<>() • AbstractMapBag<>() • Hashbag<>() • SortedBag<>()	• Arrays.asList("xyz", "abc", "cbd")

Steps:

1. Check for collection elements with .containsAll() method provided by the common Bag interface
2. Check Boolean value returned by method call

Expected Result:

Returns true for bags that contained all elements in parameter list
Returns false for empty bags and those containing only one element from the parameter list

Actual Result:

Returns true for bags that contained all elements in parameter list
Returns false for empty bags and those containing only one element from the parameter list

Verdict:

Concrete Bag implementations in the Apache Commons Collections work for content verification from collection for both empty and non-empty instances of Bag, AbstractMapBag, Hashbag and SortedBag.

Test 4 – Remove all instances of an item from Bag instance

Inputs:

Instance(s)	Parameter(s)

• Bag<> (Arrays.asList("abc", "abc", "cbd")) • AbstractMapBag<>(Arrays.asList("abc", "abc", "cbd")) • Hashbag<>(Arrays.asList("abc", "abc", "cbd")) • SortedBag<>(Arrays.asList("abc", "abc", "cbd")) • Bag<>() • AbstractMapBag<>() • Hashbag<>() • SortedBag<>()	• String “abc” • String “foo”

Steps:

1. Save bag size in buffer variable
2. Remove “abc” string from bag with .remove(Object object) method
3. Check Boolean value returned by method call
4. Compare bag size with one in buffer
5. Remove “foo” string from bag with .remove(Object object) method
6. Check Boolean value returned by method call
7. Compare bag size with one in buffer

Expected Result:

Returns true for bags that contained element and size decreased by number of occurrences.

Returns false for empty bags and those containing no instances of parameter string and size unchanged.

Actual Result:

Returns true for bags that contained element and size decreased by number of occurrences.

Returns false for empty bags and those containing no instances of parameter string.

Verdict:

Concrete Bag implementations in the Apache Commons Collections work for content removal from collection for both empty and non-empty instances of Bag, AbstractMapBag, Hashbag and SortedBag.

Test 5 – Remove n instances of an item from Bag instance

Inputs:

Instance(s)	Parameter(s)

• Bag<> (Arrays.asList("abc", "abc", "abc", "cbd")) • AbstractMapBag<>(Arrays.asList("abc", "abc", "abc", "cbd")) • Hashbag<>(Arrays.asList("abc", "abc", "abc", "cbd")) • SortedBag<>(Arrays.asList("abc", "abc", "abc", "cbd")) • Bag<>() • AbstractMapBag<>() • Hashbag<>() • SortedBag<>()	• String “abc” • String “foo” • Int: 2

Steps:

1. Save bag size in buffer variable
2. Remove 2 instances of “abc” string from bag with .remove(Object object, int nCopies) method
3. Check Boolean value returned by method call
4. Compare bag size with one in buffer
5. Remove “foo” string from bag with .remove(Object object, int nCopies) method
6. Check Boolean value returned by method call
7. Compare bag size with one in buffer

Expected Result:

Returns true for bags that contained element and size decreased by number specified as parameter.

Returns false for empty bags and those containing no instances of parameter string and size unchanged.

Actual Result:

Returns true for bags that contained element and size decreased by number specified as parameter.

Returns false for empty bags and those containing no instances of parameter string.

Verdict:

Concrete Bag implementations in the Apache Commons Collections work for content removal from collection for both empty and non-empty instances of Bag, AbstractMapBag, Hashbag and SortedBag.

Test 6 – Remove all elements specified in parameter list from Bag

Inputs:

Instance(s)	Parameter(s)

• Bag<> (Arrays.asList("xyz", "abc", "cbd")) • AbstractMapBag<>(Arrays.asList("xyz", "abc", "cbd")) • Hashbag<>(Arrays.asList("xyz", "abc", "cbd")) • SortedBag<>(Arrays.asList("xyz", "abc", "cbd")) • Bag<>((Arrays.asList("xyz")) • AbstractMapBag<>((Arrays.asList("xyz")) • Hashbag<>((Arrays.asList("xyz")) • SortedBag<>((Arrays.asList("xyz")) • Bag<>() • AbstractMapBag<>() • Hashbag<>() • SortedBag<>()	• Arrays.asList("xyz", "abc", "cbd")

Steps:

1. Save bag size in buffer variable
2. Remove all elements in param list from bag with .removeAll(Collection<?> coll) method
3. Check Boolean value returned by method call
4. Compare bag size with one in buffer

Expected Result:

Returns true for bags that contained elements and size decreased by number of occurences of each element in param list.
Returns false for empty bags and those not containing any elements in param list and size unchanged.

Actual Result:

For Bags with three elements:
Returned true. Size after = 0
For Bags with one element:
Returned true. Size after = 0
For empty Bags:
Returned false. Size after = 0

Returns true for bags that contained elements and size decreased by number of occurences of each element in param list.
Returns false for empty bags and those not containing any elements in param list and size unchanged.

Verdict:

Concrete Bag implementations in the Apache Commons Collections work for content removal from collection for both empty and non-empty instances of Bag, AbstractMapBag, Hashbag and SortedBag.

Test 7 – Retain all instances specified in parameter list from Bag and remove others

Inputs:

Instance(s)	Parameter(s)
• Bag<> (Arrays.asList("xyz", "abc", "cbd")) • AbstractMapBag<>(Arrays.asList("xyz", "abc", "cbd")) • Hashbag<>(Arrays.asList("xyz", "abc", "cbd")) • SortedBag<>(Arrays.asList("xyz", "abc", "cbd")) • Bag<>((Arrays.asList("xyz")) • AbstractMapBag<>((Arrays.asList("xyz")) • Hashbag<>((Arrays.asList("xyz")) • SortedBag<>((Arrays.asList("xyz")) • Bag<>() • AbstractMapBag<>() • Hashbag<>() • SortedBag<>()	• Arrays.asList("xyz", "abc")

Steps:

1. Save bag size in buffer variable
2. Retain all elements in param list from bag with .retainAll(Collection<?> coll) method
3. Check Boolean value returned by method call
4. Compare bag size with one in buffer

Expected Result:

Returns true for bags that contained elements not specified by param list.
Size unchanged for empty bags and bags only containing elements specified in param list.
Size should be equal to number of occurrence of each element in the specified param list.

Actual Result:

For Bags with three elements:
    Returned true. Size after = 1
For Bags with one element:
    Returned false. Size after = 1
For empty Bags:

Returned false. Size after = 0

Returns true for bags that contained elements not specified by param list.
Size unchanged for empty bags and bags only containing elements specified in param list.
Size should be equal to number of occurrence of each element in the specified param list.

Verdict:

Concrete Bag implementations in the Apache Commons Collections work for content removal from collection through retainment for both empty and non-empty instances of Bag, AbstractMapBag, Hashbag and SortedBag.

Test 8 – Get Bag size

Inputs:

Instance(s)	Parameter(s)
• Bag<> (Arrays.asList("xyz", "abc", "cbd")) • AbstractMapBag<>(Arrays.asList("xyz", "abc", "cbd")) • Hashbag<>(Arrays.asList("xyz", "abc", "cbd")) • SortedBag<>(Arrays.asList("xyz", "abc", "cbd")) • Bag<>() • AbstractMapBag<>() • Hashbag<>() • SortedBag<>()	

Steps:

1. Check size of bag after creation

Expected Result:

Size of Bag instance equal to 3 for nonempty bags
Size of Bag instance equal to 0 for empty bags

Actual Result:

Size of Bag instance equal to 3 for nonempty bags
Size of Bag instance equal to 0 for empty bags

Verdict:

Concrete Bag implementations in the Apache Commons Collections work for size retrieval for both empty and non-empty instances of Bag, AbstractMapBag, Hashbag and SortedBag.

BIDIRECTIONAL MAP

Test 0 – Create Bidirectional Map instances

Inputs:

Instance(s)	Parameter(s)

Steps:

1. Create four instances of different BidiMap implementations with creator (new BidiMap()) and save instances in variables
   - DualHashBidiMap<>()
   - DualLinkedHashBidiMap<>()
   - DualTreeBidiMap<>()
   - TreeBidiMap<>()

2. Check that variables are not null

Expected Result:

Variables are not null

Actual Result:

Variables are not null

Verdict:

Concrete BidiMap implementations in the Apache Commons Collections work for creation of instances of DualHashBidiMap, DualLinkHashBidiMap, DualTreeBidiMap, TreeBidiMap.

Test 1 – Add element to bidirectional map

Inputs:

Instance(s)	Parameter(s)
• DualHashBidiMap<>() • DualLinkedHashBidiMap<>() • DualTreeBidiMap<>() • TreeBidiMap<>()	String: “key” String: “value”

Steps:

1. Save map size in buffer variable
2. Add new item to map with .put(K key, V value) method
3. Compare map size with one in buffer

Expected Result:

Size increased by one

Actual Result:

Size increase by one

Verdict:

Concrete BidiMap implementations in the Apache Commons Collections work for adding elements to instances of DualHashBidiMap, DualLinkHashBidiMap, DualTreeBidiMap, TreeBidiMap.

Test 2 – Add several elements to bidirectional map

Inputs:

Instance(s)	Parameter(s)
• DualHashBidiMap<>() • DualLinkedHashBidiMap<>() • DualTreeBidiMap<>() • TreeBidiMap<>()	String: “key” String: “value” String: “k2” String: “v2” String: “k3”

Steps:

1. Save map size in buffer variable
2. Add new item to map with .put(“key”, “value”) method
3. Compare map size with one in buffer
4. Add new item to map with .put(“k2”, “v2”) method
5. Compare map size with one in buffer
6. Add new item to map with .put(“k3”, “value”) method
7. Compare map size with one in buffer

Expected Result:

Size increased by one after step 3.
Size increased by two after step 5.
Size unchanged after step 7 as “value” will be remapped to “k3” and old instance removed

Actual Result:

Size increased by one after step 3.
Size increased by two after step 5.
Size unchanged after step 7.

Verdict:

Concrete BidiMap implementations in the Apache Commons Collections work for adding elements to instances of DualHashBidiMap, DualLinkHashBidiMap, DualTreeBidiMap, TreeBidiMap.

Test 3 – Remove element from bidirectional map

Inputs:

Instance(s)	Parameter(s)
• DualHashBidiMap<>() • DualLinkedHashBidiMap<>() • DualTreeBidiMap<>() • TreeBidiMap<>() ▪ .put("k","v")	String: "k" String: "v"

Steps:

1. Save map size in buffer variable
2. Remove pair with .removeValue("v")
3. Compare returned key with "k"
4. Compare map size with one in buffer
5. Remove pair with removeValue("v")
6. Check return value
7. Compare map size with one in buffer

Expected Result:

Size decreased by one (thus empty) after step 3.
Returned key equal to "k"
Size remains the same as map is empty
Returned value equal to null

Actual Result:

Size decreased by one (thus empty) after step 3.
Returned key equal to "k"
Size remains the same as map is empty
Returned value equal to null

Verdict:

Concrete BidiMap implementations in the Apache Commons Collections work for removing elements to instances of DualHashBidiMap, DualLinkHashBidiMap, DualTreeBidiMap, TreeBidiMap.

Test 4 – Get value from map with key

Inputs:

Instance(s)	Parameter(s)
• DualHashBidiMap<>() • DualLinkedHashBidiMap<>() • DualTreeBidiMap<>() • TreeBidiMap<>() ▪ .put(“k”,”v”)	String: “foo”

Steps:

1. Get value from map with .get(“k”)
2. Compare value with “v”
3. Get value from map with .get(“foo”)
4. Check returned value

Expected Result:

Returned value equal “v”
Returned value equal null

Actual Result:

Returned value equal “v”
Returned value equal null

Verdict:

Concrete BidiMap implementations in the Apache Commons Collections work for removing elements to instances of DualHashBidiMap, DualLinkHashBidiMap, DualTreeBidiMap, TreeBidiMap.

Test 5 – Get key from map with value

Inputs:

Instance(s)	Parameter(s)
• DualHashBidiMap<>() • DualLinkedHashBidiMap<>() • DualTreeBidiMap<>() • TreeBidiMap<>() ▪ .put(“k”,”v”)	String: “foo” String: “v”

Steps:

1. Get value from map with .get(“v”)
2. Compare value with “k”
3. Get value from map with .get(“foo”)
4. Check returned value

Expected Result:

Returned value equal “k”
Returned value equal null

Actual Result:

Returned value equal “k”
Returned value equal null

Verdict:

Concrete BidiMap implementations in the Apache Commons Collections work for retrieving elements from instances of DualHashBidiMap, DualLinkHashBidiMap, DualTreeBidiMap, TreeBidiMap.

Test 6 – Get key from map with value

Inputs:

Instance(s)	Parameter(s)
• DualHashBidiMap<>() • DualLinkedHashBidiMap<>() • DualTreeBidiMap<>() • TreeBidiMap<>() ▪ .put(“k”,”v”)	String: “foo” String: “v”

Steps:

1. Get value from map with .get(“v”)
2. Compare value with “k”
3. Get value from map with .get(“foo”)
4. Check returned value

Expected Result:

Returned key equal “k”
Returned null

Actual Result:

Returned key equal “k”
Returned null

Verdict:

Concrete BidiMap implementations in the Apache Commons Collections work for retrieving elements from instances of DualHashBidiMap, DualLinkHashBidiMap, DualTreeBidiMap, TreeBidiMap.

Test 6 – Get a view of a BidiMap with keys and values reversed

Inputs:

Instance(s)	Parameter(s)
• DualHashBidiMap<>() • DualLinkedHashBidiMap<>() • DualTreeBidiMap<>() • TreeBidiMap<>() ▪ .put(“k”,”v”)	String: “k” String: “v”

Steps:

1. Inverse BidiMap with .inverseBidiMap method and save object returned in variable
2. Run .getKey(“k”) on inversed map
3. Compare returned key with “v”
4. Run .getValue(“v”) on inversed map
5. Compare returned value with “k”

Expected Result:

Returned key is equal to value from initial bidimap “v”
Returned value is equal to key from initial bidimap “k”

Actual Result:

Returned key is equal to value from initial bidimap “v”
Returned value is equal to key from initial bidimap “k”

Verdict:

Concrete BidiMap implementations in the Apache Commons Collections work for creating inverted version of instances of DualHashBidiMap, DualLinkHashBidiMap, DualTreeBidiMap, TreeBidiMap.

COMPARATORS

Test 0 – Get a view of a BidiMap with keys and values reversed

Inputs:

Instance(s)	Parameter(s)

Steps:

1. Create instanced of all 7 types of comparators with constructors and save instances to variables
    - BooleanComparator<>()
    - ComparableComparator<>()
    - ComparatorChain<>()
    - NullComparator<>()
    - ReverseComparator<>()
    - TransformingComparator<>()
2. Check that instances aren't null

Expected Result:

Instances are not null

Actual Result:

Instances are not null

Verdict:

Concrete Comparator implementations in the Apache Commons Collections work for creating instances of BooleanComparator, ComparableComparator, ComparatorChain, NullComparator, ReverseComparator, TransformingComparator.

BooleanComparator

Test 1 – Compare boolean objects with property sortsTrueFirst set to false

Inputs:

Instance(s)	Parameter(s)
booleanComparator<>(false)	Boolean: true Boolean: false null

Steps:

1. Run .compare(true, false)
2. Check return value
3. Run .compare(false, true)
4. Check return value
5. Run .compare(null, true)
6. Check return value

Expected Result:

False is returned
True is returned
NullPointerException is thrown

Actual Result:

False is returned
True is returned
NullPointerException is thrown

Verdict:

Concrete Comparator implementations in the Apache Commons Collections work for comparing boolean variables with BooleanComparator

---

Test 2 – Compare boolean objects with property sortsTrueFirst set to true

Inputs:

Instance(s)	Parameter(s)
booleanComparator<>(true)	Boolean: true Boolean: false null

Steps:

1. Run .compare(true, false)
2. Check return value
3. Run .compare(false, true)
4. Check return value
5. Run .compare(null, true)
6. Check return value

Expected Result:

True is returned
False is returned
NullPointerException is thrown

Actual Result:

True is returned
False is returned
NullPointerException is thrown

Verdict:

Concrete Comparator implementations in the Apache Commons Collections work for comparing boolean variables with BooleanComparator

---

Test 3 – Compare Comparator ordering

Inputs:

Instance(s)	Parameter(s)
booleanComparator<>(true) booleanComparator<>(false)	booleanComparator<>(true) booleanComparator<>(false)

Steps:

1. Run .equals(Object object) from true comparator with false comparator as parameter
2. Check return value
3. Run .equals(Object object from false comparator with false comparator as parameter
4. Check return value
5. Run sortsTrueFirst() from false comparator
6. Run step 1. Again
7. Check return value

Expected Result:

False is returned
False is returned
True is returned

Actual Result:

False is returned
False is returned
True is returned

Verdict:

Concrete Comparator implementations in the Apache Commons Collections work for comparing ordering for BooleanComparator,

---

ComparableComparator

Test 1 – Compare comparables

Inputs:

Instance(s)	Parameter(s)
comparableComparator<>(false)	String<Comparable> : “abc” String<Comparable> : “def”

Steps:

1. Run .compare(true, false)
2. Check return value
3. Run .compare(false, true)
4. Check return value
5. Run .compare(null, true)
6. Check return value

Expected Result:

False is returned
True is returned
NullPointerException is thrown

Actual Result:

False is returned
True is returned
NullPointerException is thrown

Verdict:

Concrete ComparableComparator implementations in the Apache Commons Collections work for comparing boolean variables

---

FixedOrderComparator

Test 1 – Add item to Comparator

Inputs:

Instance(s)	Parameter(s)
FixedOrderComparator<>({“first”})	String: “second”

Steps:

1. Add “second” to items known by comparator with .add(“second”)
2. Check return value
3. Run toString()
4. Verify that new entry is known to comparator and sits after “first”
5. Add “first” to items known by the comparator with .add(“first”)
6. Check return value

Expected Result:

Return value should be true at step 2.
Order of items known to comparator is {“first”,”second”}
Return value should be false at step 5.

Actual Result:

Order of items known to comparator is {“first”,”second”}

Verdict:

Concrete FixedOrderComparator implementations in the Apache Commons Collections work for adding elements to known ordering

---

Test 2 – Add item to Comparator equal to existing item

Inputs:

Instance(s)	Parameter(s)
FixedOrderComparator<>({“first”, “second”})	String: “third”

Steps:

1. Add “third” to items known by comparator with .addAsEqual(“first”, “third”)
2. Check return value
3. Run .compare(“first”, “third”)
4. Check return value
5. Add “first” to items known by the comparator with .add(“first”)
6. Check return value

Expected Result:

Return value should be true at step 2.
Return value should be 0
UnsupportedOpertaionsException thrown at step 6.

Actual Result:

Return value should be true at step 2.
Return value should be 0
UnsupportedOpertaionsException thrown at step 6.

Verdict:

Concrete FixedOrderComparator implementations in the Apache Commons Collections work for adding elements to known ordering

---

Test 3 – Compare items

Inputs:

Instance(s)	Parameter(s)
FixedOrderComparator<>({“first”, “second”})	String: “first” String: “second”

Steps:

1. Run .compare(“first”, “second”)
2. Check return value
3. Run .compare(“second, “first”)
4. Check return value
5. Run .compare(“first, “first”)
6. Check return value
7. Run .compare(“third”, “first”)

Expected Result:

Return value is -1 at step 2.
Return value is 1 at step 4.
Return value is 0 at step 6
IllegalArgumentException thrown at step 7

Actual Result:

Return value is -1 at step 2.
Return value is 1 at step 4.
Return value is 0 at step 6
IllegalArgumentException thrown at step 7

Verdict:

Concrete FixedOrderComparator implementations in the Apache Commons Collections work for comparing elements in known ordering

---

NullComparator

Test 1 – Compare items with nullsAreHigh set to true

Inputs:

Instance(s)	Parameter(s)
NullComparator<>()	Int: 1 Int: 34 null

Steps:

1. Compare null and 1 with .compare(null, 1) method
2. Check return value
3. Compare 34 and null with .compare(34, null) method
4. Check return value

Expected Result:

Return value is 1 at step 2.
Return value is -1 at step 4.

Actual Result:

Return value is 1 at step 2.
Return value is -1 at step 4.

Verdict:

Concrete NullComparator implementation in the Apache Commons Collections work for comparing null and non-null elements

---

Test 2 – Compare items with nullsAreHigh set to false

Inputs:

Instance(s)	Parameter(s)
NullComparator<>(false)	Int: 1 Int: 34 null

Steps:

1. Compare null and 1 with .compare(null, 1) method
2. Check return value
3. Compare 34 and null with .compare(34, null) method
4. Check return value

Expected Result:

Return value is -1 at step 2.
Return value is 1 at step 4.

Actual Result:

Return value is -1 at step 2.
Return value is 1 at step 4.

Verdict:

Concrete NullComparator implementation in the Apache Commons Collections work for comparing null and non-null elements

---

ReverseComparator

Test 1 – Compare items

Inputs:

Instance(s)	Parameter(s)
Comparator<>()	Int: 1 Int: -34

Steps:

1. Compare -34 and 1 with .compare(-34, 1) method
2. Save return value to buffer integer variable
3. Create instance of ReverseComparator with Comparator instance as parameter
4. Repeat step 1.
5. Compare return value with buffer variable

Expected Result:

Comparison at step 5 shows that the two return values are opposite as order of comparison has been reversed.

Actual Result:

Comparison at step 5 shows that the two return values are opposite as order of comparison has been reversed.

Verdict:

Concrete ReverseComparator implementation in the Apache Commons Collections work for comparing elements in reverse order

---