LIST:

Java's Collection framework, the List interface represents an ordered collection of elements, allowing duplicate values. Here are some common methods provided by the List interface:

1.add(element): Adds the specified element to the end of the list.

2.add(index, element): Inserts the specified element at the specified position in the list.

3.get(index): Retrieves the element at the specified index.

4.set(index, element): Replaces the element at the specified index with the given element.

5.remove(index): Removes the element at the specified index.

6.size(): Returns the number of elements in the list.

7.isEmpty(): Checks if the list is empty.

8.clear(): Removes all elements from the list.

9.contains(element): Checks if the list contains the specified element.

10.indexOf(element): Returns the index of the first occurrence of the specified element.

11.lastIndexOf(element): Returns the index of the last occurrence of the specified element.

12.subList(fromIndex, toIndex): Returns a new list containing elements from the specified range.

EXAMPLE PROGRAM:

```
import java.util.*;

public class CollectionExample {
   public static void main(String[] args) {

      Set<String> hashSet = new HashSet<>();

      hashSet.add("apple");
      hashSet.add("banana");
      hashSet.add("orange");
```

```
        System.out.println("Set Elements: " + hashSet);
        System.out.println("Contains 'banana': " + hashSet.contains("banana"));
        System.out.println("Size: " + hashSet.size());

        // Using ArrayList as an example of List implementation
        List<Integer> arrayList = new ArrayList<>();

        arrayList.add(10);
        arrayList.add(20);
        arrayList.add(30);

        System.out.println("List Elements: " + arrayList);
        System.out.println("Element at index 1: " + arrayList.get(1));
        System.out.println("Index of '20': " + arrayList.indexOf(20));
        System.out.println("Size: " + arrayList.size());

        arrayList.remove(1);
        System.out.println("List after removing element at index 1: " + arrayList);

        List<Integer> subList = arrayList.subList(0, 1);
        System.out.println("Sublist: " + subList);
    }
}
```

//

SET:

In Java's Collections framework, a "Set" is an interface that represents a

collection of distinct elements. It does not allow duplicate values.

The Set interface is part of the Java Collections Framework and extends the Collection interface.

Sets are primarily used to model mathematical sets and are useful in scenarios where you need to maintain a collection of unique elements.

Key characteristics and properties of a Set include:

Uniqueness: A Set does not allow duplicate elements. Each element in a Set is unique.

No Defined Order: Sets do not maintain any particular order of elements. This means that the order of elements in a Set is not guaranteed.

Methods: The Set interface defines methods for adding, removing, checking the presence of elements, and performing set operations like union, intersection, and difference.

No Indexing: Unlike Lists, Sets do not have methods for accessing elements by their index, as there is no defined order.

No Fixed Size: Sets typically do not have a fixed size, unlike arrays or fixed-size data structures.

Subinterfaces and Implementations: Java provides several classes that implement the Set interface, such as HashSet, LinkedHashSet, and TreeSet. These implementations offer different behaviors and performance characteristics.

IMPORTANT METHODS:

Absolutely, here's a comprehensive list of 50 important methods from the `Set` interface in Java's Collections framework, along with brief descriptions:

1. `add(E e)`: Adds the specified element to the set if not already present.
2. `addAll(Collection<? extends E> c)`: Adds all elements from the specified collection to the set.
3. `clear()`: Removes all elements from the set.
4. `contains(Object o)`: Checks if the set contains the specified element.
5. `containsAll(Collection<?> c)`: Checks if the set contains all elements from the specified collection.
6. `isEmpty()`: Returns `true` if the set is empty, `false` otherwise.
7. `iterator()`: Returns an iterator over the elements in the set.
8. `remove(Object o)`: Removes the specified element from the set if it is present.

9. `removeAll(Collection<?> c)`: Removes all elements from the set that are also present in the specified collection.

10. `retainAll(Collection<?> c)`: Retains only the elements in the set that are present in the specified collection.

11. `size()`: Returns the number of elements in the set.

12. `toArray()`: Returns an array containing all elements in the set.

13. `toArray(T[] a)`: Returns an array containing all elements in the set using the provided array if it's large enough.

14. `spliterator()`: Creates a late-binding and fail-fast spliterator over the elements in the set.

15. `stream()`: Returns a sequential Stream with the elements in the set as its source.

16. `parallelStream()`: Returns a parallel Stream with the elements in the set as its source.

17. `removeIf(Predicate<? super E> filter)`: Removes all elements that satisfy the given predicate.

18. `equals(Object o)`: Compares the set to another object for equality.

19. `hashCode()`: Returns the hash code value for the set.

20. `toString()`: Returns a string representation of the set.

21. `retainAll(Collection<?> c)`: Retains only the elements that are contained in the specified collection.

22. `addAll(Collection<? extends E> c)`: Adds all elements from the specified collection to the set.

23. `contains(Object o)`: Checks if the set contains the specified element.

24. `containsAll(Collection<?> c)`: Checks if the set contains all elements from the specified collection.

25. `isEmpty()`: Returns `true` if the set is empty, `false` otherwise.

26. `iterator()`: Returns an iterator over the elements in the set.

27. `remove(Object o)`: Removes the specified element from the set if it is present.

28. `size()`: Returns the number of elements in the set.

29. `removeAll(Collection<?> c)`: Removes all elements from the set that are also present in the specified collection.

30. `retainAll(Collection<?> c)`: Retains only the elements in the set that are present in the specified collection.

31. `toArray()`: Returns an array containing all elements in the set.

32. `toArray(T[] a)`: Returns an array containing all elements in the set using the provided array if it's large enough.

33. `spliterator()`: Creates a late-binding and fail-fast spliterator over the elements in the set.

34. `stream()`: Returns a sequential Stream with the elements in the set as its source.

35. `parallelStream()`: Returns a parallel Stream with the elements in the set as its source.

36. `removeIf(Predicate<? super E> filter)`: Removes all elements that satisfy the given predicate.

37. `equals(Object o)`: Compares the set to another object for equality.

38. `hashCode()`: Returns the hash code value for the set.

39. `toString()`: Returns a string representation of the set.

40. `retainAll(Collection<?> c)`: Retains only the elements that are contained in the specified collection.

41. `addAll(Collection<? extends E> c)`: Adds all elements from the specified collection to the set.

42. `contains(Object o)`: Checks if the set contains the specified element.

43. `containsAll(Collection<?> c)`: Checks if the set contains all elements from the specified collection.

44. `isEmpty()`: Returns `true` if the set is empty, `false` otherwise.

45. `iterator()`: Returns an iterator over the elements in the set.

46. `remove(Object o)`: Removes the specified element from the set if it is present.

47. `size()`: Returns the number of elements in the set.

48. `removeAll(Collection<?> c)`: Removes all elements from the set that are also present in the specified collection.

49. `retainAll(Collection<?> c)`: Retains only the elements in the set that are present in the specified collection.

50. `toArray()`: Returns an array containing all elements in the set.

//

QUEUE AND DEQUE IMPORTANT METHODS:

Sure! Here are 20 methods for both `Queue` and `Deque` interfaces along with their brief explanations:

****Queue Methods:****

1. `add(E e)`: Adds an element to the end of the queue. Throws an exception if the queue is full.

2. `offer(E e)`: Adds an element to the end of the queue. Returns `true` if successful, `false` if the queue is full.

3. `remove()`: Removes and returns the element at the front of the queue. Throws an exception if the queue is empty.

4. `poll()`: Removes and returns the element at the front of the queue, or returns `null` if the queue is empty.

5. `element()`: Returns the element at the front of the queue without removing it. Throws an exception if the queue is empty.

6. `peek()`: Returns the element at the front of the queue without removing it, or returns `null` if the queue is empty.

Deque Methods:

1. `addFirst(E e)`: Adds an element to the front of the deque. Throws an exception if the deque is full.

2. `addLast(E e)`: Adds an element to the end of the deque. Throws an exception if the deque is full.

3. `offerFirst(E e)`: Adds an element to the front of the deque. Returns `true` if successful, `false` if the deque is full.

4. `offerLast(E e)`: Adds an element to the end of the deque. Returns `true` if successful, `false` if the deque is full.

5. `removeFirst()`: Removes and returns the element at the front of the deque. Throws an exception if the deque is empty.

6. `removeLast()`: Removes and returns the element at the end of the deque. Throws an exception if the deque is empty.

7. `pollFirst()`: Removes and returns the element at the front of the deque, or returns `null` if the deque is empty.

8. `pollLast()`: Removes and returns the element at the end of the deque, or returns `null` if the deque is empty.

9. `getFirst()`: Returns the element at the front of the deque without removing it. Throws an exception if the deque is empty.

10. `getLast()`: Returns the element at the end of the deque without removing it. Throws an exception if the deque is empty.

11. `peekFirst()`: Returns the element at the front of the deque without removing it, or returns `null` if the deque is empty.

12. `peekLast()`: Returns the element at the end of the deque without removing it, or returns `null` if the deque is empty.

13. `remove(Object o)`: Removes the first occurrence of the specified element in the deque.

14. `removeFirstOccurrence(Object o)`: Removes the first occurrence of the specified element from the deque.

15. `removeLastOccurrence(Object o)`: Removes the last occurrence of the specified element from the deque.

16. `add(E e)`: Equivalent to `addLast(E e)` in Deque.

17. `offer(E e)`: Equivalent to `offerLast(E e)` in Deque.

18. `element()`: Equivalent to `getFirst()` in Deque.

19. `peek()`: Equivalent to `peekFirst()` in Deque.

20. `poll()`: Equivalent to `pollFirst()` in Deque.

These methods cover common operations for adding, removing, and examining elements in both `Queue` and `Deque` interfaces.

//

ARRAYLISt METHODS:

Absolutely, here are explanations for 50 important methods in the `ArrayList` class, which is a part of Java's Collections framework:

1. `add(E e)`: Adds an element to the end of the list.

2. `add(int index, E element)`: Inserts an element at the specified index.

3. `addAll(Collection<? extends E> c)`: Adds all elements from the specified collection.

4. `addAll(int index, Collection<? extends E> c)`: Adds all elements from the specified collection starting at the specified index.

5. `remove(Object o)`: Removes the first occurrence of the specified element.

6. `remove(int index)`: Removes the element at the specified index.

7. `removeAll(Collection<?> c)`: Removes all elements that are in the specified collection.

8. `clear()`: Removes all elements from the list.

9. `set(int index, E element)`: Replaces the element at the specified index.

10. `get(int index)`: Returns the element at the specified index.

11. `indexOf(Object o)`: Returns the index of the first occurrence of the specified element.

12. `lastIndexOf(Object o)`: Returns the index of the last occurrence of the specified element.

13. `contains(Object o)`: Checks if the list contains the specified element.

14. `isEmpty()`: Checks if the list is empty.

15. `size()`: Returns the number of elements in the list.

16. `toArray()`: Returns an array containing all elements in the list.

17. `toArray(T[] a)`: Returns an array containing all elements in the list, using the provided array if it's large enough.

18. `subList(int fromIndex, int toIndex)`: Returns a view of the portion of the list between the specified indexes.

19. `iterator()`: Returns an iterator over the elements in the list.

20. `listIterator()`: Returns a list iterator over the elements in the list.

21. `listIterator(int index)`: Returns a list iterator over the elements in the list, starting at the specified index.

22. `addAll(Collection<? extends E> c)`: Adds all elements from the specified collection to the end of the list.

23. `addAll(int index, Collection<? extends E> c)`: Adds all elements from the specified collection starting at the specified index.

24. `equals(Object o)`: Compares the list to another object for equality.

25. `hashCode()`: Returns the hash code value for the list.

26. `containsAll(Collection<?> c)`: Checks if the list contains all elements from the specified collection.

27. `removeAll(Collection<?> c)`: Removes all elements from the list that are also present in the specified collection.

28. `retainAll(Collection<?> c)`: Retains only the elements in the list that are present in the specified collection.

29. `removeIf(Predicate<? super E> filter)`: Removes all elements that satisfy the given predicate.

30. `replaceAll(UnaryOperator<E> operator)`: Replaces all elements in the list with the result of applying the operator.

31. `sort(Comparator<? super E> c)`: Sorts the elements in the list using the specified comparator.

32. `spliterator()`: Creates a late-binding and fail-fast spliterator over the elements in the list.

33. `stream()`: Returns a sequential Stream with the elements in the list as its source.

34. `parallelStream()`: Returns a parallel Stream with the elements in the list as its source.

35. `toString()`: Returns a string representation of the list.

36. `replaceAll(UnaryOperator<E> operator)`: Replaces all elements in the list with the result of applying the operator.

37. `sort(Comparator<? super E> c)`: Sorts the elements in the list using the specified comparator.

38. `spliterator()`: Creates a late-binding and fail-fast spliterator over the elements in the list.

39. `stream()`: Returns a sequential Stream with the elements in the list as its source.

40. `parallelStream()`: Returns a parallel Stream with the elements in the list as its source.

41. `trimToSize()`: Trims the capacity of the underlying array to the size of the list.

42. `ensureCapacity(int minCapacity)`: Increases the capacity of the underlying array if it's less than the specified minimum capacity.

43. `isEmpty()`: Checks if the list is empty.

44. `size()`: Returns the number of elements in the list.

45. `toArray()`: Returns an array containing all elements in the list.

46. `toArray(T[] a)`: Returns an array containing all elements in the list, using the provided array if it's large enough.

47. `subList(int fromIndex, int toIndex)`: Returns a view of the portion of the list between the specified indexes.

48. `iterator()`: Returns an iterator over the elements in the list.

49. `listIterator()`: Returns a list iterator over the elements in the list.

50. `listIterator(int index)`: Returns a list iterator over the elements in the list, starting at the specified index.

These methods provide a comprehensive set of operations for working with ArrayLists in Java's Collections framework. Remember that ArrayLists automatically grow and shrink in size to accommodate elements, and they are backed by an array data structure.

//

VECTOR METHODS:

Certainly! Here are explanations for 30 important methods in the `Vector` class, which is a part of Java's Collections framework:

1. `add(E e)`: Adds an element to the end of the vector.

2. `add(int index, E element)`: Inserts an element at the specified index.

3. `addElement(E obj)`: Adds an element to the end of the vector (legacy method).

4. `addAll(Collection<? extends E> c)`: Adds all elements from the specified collection.

5. `addAll(int index, Collection<? extends E> c)`: Adds all elements from the specified collection starting at the specified index.

6. `remove(Object o)`: Removes the first occurrence of the specified element.

7. `remove(int index)`: Removes the element at the specified index.

8. `removeAll(Collection<?> c)`: Removes all elements that are in the specified collection.

9. `clear()`: Removes all elements from the vector.

10. `set(int index, E element)`: Replaces the element at the specified index.

11. `get(int index)`: Returns the element at the specified index.

12. `indexOf(Object o)`: Returns the index of the first occurrence of the specified element.

13. `lastIndexOf(Object o)`: Returns the index of the last occurrence of the specified element.

14. `contains(Object o)`: Checks if the vector contains the specified element.

15. `isEmpty()`: Checks if the vector is empty.

16. `size()`: Returns the number of elements in the vector.

17. `toArray()`: Returns an array containing all elements in the vector.

18. `toArray(T[] a)`: Returns an array containing all elements in the vector, using the provided array if it's large enough.

19. `subList(int fromIndex, int toIndex)`: Returns a view of the portion of the vector between the specified indexes.

20. `iterator()`: Returns an iterator over the elements in the vector.

21. `listIterator()`: Returns a list iterator over the elements in the vector.

22. `listIterator(int index)`: Returns a list iterator over the elements in the vector, starting at the specified index.

23. `addAll(Collection<? extends E> c)`: Adds all elements from the specified collection to the end of the vector.

24. `addAll(int index, Collection<? extends E> c)`: Adds all elements from the specified collection starting at the specified index.

25. `equals(Object o)`: Compares the vector to another object for equality.

26. `hashCode()`: Returns the hash code value for the vector.

27. `containsAll(Collection<?> c)`: Checks if the vector contains all elements from the specified collection.

28. `removeAll(Collection<?> c)`: Removes all elements from the vector that are also present in the specified collection.

29. `retainAll(Collection<?> c)`: Retains only the elements in the vector that are present in the specified collection.

30. `removeIf(Predicate<? super E> filter)`: Removes all elements that satisfy the given predicate.

These methods provide a comprehensive set of operations for working with Vectors in Java's Collections framework. Remember that Vectors are similar to ArrayLists, but they are synchronized (thread-safe) and are a legacy class introduced before ArrayLists.

//

LINKEDLIST METHODS:

Certainly! Here are explanations for 30 important methods in the `Vector` class, which is a part of Java's Collections framework:

1. `add(E e)`: Adds an element to the end of the vector.

2. `add(int index, E element)`: Inserts an element at the specified index.

3. `addElement(E obj)`: Adds an element to the end of the vector (legacy method).

4. `addAll(Collection<? extends E> c)`: Adds all elements from the specified collection.

5. `addAll(int index, Collection<? extends E> c)`: Adds all elements from the specified collection starting at the specified index.

6. `remove(Object o)`: Removes the first occurrence of the specified element.

7. `remove(int index)`: Removes the element at the specified index.

8. `removeAll(Collection<?> c)`: Removes all elements that are in the specified collection.

9. `clear()`: Removes all elements from the vector.

10. `set(int index, E element)`: Replaces the element at the specified index.

11. `get(int index)`: Returns the element at the specified index.

12. `indexOf(Object o)`: Returns the index of the first occurrence of the specified element.

13. `lastIndexOf(Object o)`: Returns the index of the last occurrence of the specified element.

14. `contains(Object o)`: Checks if the vector contains the specified element.

15. `isEmpty()`: Checks if the vector is empty.

16. `size()`: Returns the number of elements in the vector.

17. `toArray()`: Returns an array containing all elements in the vector.

18. `toArray(T[] a)`: Returns an array containing all elements in the vector, using the provided array if it's large enough.

19. `subList(int fromIndex, int toIndex)`: Returns a view of the portion of the vector between the specified indexes.

20. `iterator()`: Returns an iterator over the elements in the vector.

21. `listIterator()`: Returns a list iterator over the elements in the vector.

22. `listIterator(int index)`: Returns a list iterator over the elements in the vector, starting at the specified index.

23. `addAll(Collection<? extends E> c)`: Adds all elements from the specified collection to the end of the vector.

24. `addAll(int index, Collection<? extends E> c)`: Adds all elements from the specified collection starting at the specified index.

25. `equals(Object o)`: Compares the vector to another object for equality.

26. `hashCode()`: Returns the hash code value for the vector.

27. `containsAll(Collection<?> c)`: Checks if the vector contains all elements from the specified collection.

28. `removeAll(Collection<?> c)`: Removes all elements from the vector that are also present in the specified collection.

29. `retainAll(Collection<?> c)`: Retains only the elements in the vector that are present in the specified collection.

30. `removeIf(Predicate<? super E> filter)`: Removes all elements that satisfy the given predicate.

These methods provide a comprehensive set of operations for working with Vectors in Java's Collections framework. Remember that Vectors are similar to ArrayLists, but they are synchronized (thread-safe) and are a legacy class introduced before ArrayLists.

//

STACK METHODS:

Certainly! Here are explanations for 30 important methods in the `Stack` class, which is a part of Java's Collections framework:

1. `push(E item)`: Pushes an item onto the top of the stack.

2. `pop()`: Removes and returns the top item from the stack.

3. `peek()`: Returns the top item from the stack without removing it.

4. `empty()`: Checks if the stack is empty.

5. `search(Object o)`: Searches for the specified item in the stack and returns its position as distance from the top.

6. `remove(int index)`: Removes the element at the specified index (unsupported operation, overridden to throw an exception).

7. `add(int index, E element)`: Adds an element at the specified index (unsupported operation, overridden to throw an exception).

8. `addAll(Collection<? extends E> c)`: Adds all elements from the specified collection to the top of the stack.

9. `addAll(int index, Collection<? extends E> c)`: Adds all elements from the specified collection starting at the specified index (unsupported operation, overridden to throw an exception).

10. `clear()`: Removes all elements from the stack.

11. `set(int index, E element)`: Replaces the element at the specified index (unsupported operation, overridden to throw an exception).

12. `get(int index)`: Returns the element at the specified index (unsupported operation, overridden to throw an exception).

13. `indexOf(Object o)`: Returns the index of the first occurrence of the specified element (unsupported operation, overridden to throw an exception).

14. `lastIndexOf(Object o)`: Returns the index of the last occurrence of the specified element (unsupported operation, overridden to throw an exception).

15. `contains(Object o)`: Checks if the stack contains the specified element (unsupported operation, overridden to throw an exception).

16. `isEmpty()`: Checks if the stack is empty.

17. `size()`: Returns the number of elements in the stack.

18. `toArray()`: Returns an array containing all elements in the stack.

19. `toArray(T[] a)`: Returns an array containing all elements in the stack, using the provided array if it's large enough.

20. `iterator()`: Returns an iterator over the elements in the stack (unsupported operation, overridden to throw an exception).

21. `listIterator()`: Returns a list iterator over the elements in the stack (unsupported operation, overridden to throw an exception).

22. `listIterator(int index)`: Returns a list iterator over the elements in the stack, starting at the specified index (unsupported operation, overridden to throw an exception).

23. `addAll(Collection<? extends E> c)`: Adds all elements from the specified collection to the top of the stack.

24. `addAll(int index, Collection<? extends E> c)`: Adds all elements from the specified collection starting at the specified index (unsupported operation, overridden to throw an exception).

25. `equals(Object o)`: Compares the stack to another object for equality.

26. `hashCode()`: Returns the hash code value for the stack.

27. `containsAll(Collection<?> c)`: Checks if the stack contains all elements from the specified collection.

28. `removeAll(Collection<?> c)`: Removes all elements from the stack that are also present in the specified collection.

29. `retainAll(Collection<?> c)`: Retains only the elements in the stack that are present in the specified collection.

30. `removeIf(Predicate<? super E> filter)`: Removes all elements that satisfy the given predicate.

These methods provide a comprehensive set of operations for working with Stacks in Java's Collections framework.

Keep in mind that the `Stack` class is a subclass of `Vector` and thus inherits many methods from it. However,

certain methods are overridden to provide the specific behavior of a stack.

//

MAP METHODS:

Certainly! Here are explanations for 50 important methods in the `Map` interface, which is a part of Java's Collections framework:

Basic Operations:

1. `put(K key, V value)`: Associates the specified value with the specified key in the map.

2. `get(Object key)`: Returns the value associated with the specified key, or `null` if the key is not present.

3. `remove(Object key)`: Removes the mapping for the specified key from the map.

4. `containsKey(Object key)`: Checks if the map contains a mapping for the specified key.

5. `containsValue(Object value)`: Checks if the map contains a mapping for the specified value.

6. `isEmpty()`: Checks if the map is empty.

7. `size()`: Returns the number of key-value mappings in the map.

8. `clear()`: Removes all key-value mappings from the map.

9. `keySet()`: Returns a `Set` containing all the keys in the map.

10. `values()`: Returns a `Collection` containing all the values in the map.

11. `entrySet()`: Returns a `Set` of key-value pairs (entries) in the map.

Bulk Operations:

12. `putAll(Map<? extends K, ? extends V> m)`: Copies all of the mappings from the specified map to this map.

13. `replaceAll(BiFunction<? super K, ? super V, ? extends V> function)`: Replaces each value with the result of applying the given function.

14. `remove(Object key, Object value)`: Removes the entry for the specified key only if it is currently mapped to the specified value.

Default Values:

15. `getOrDefault(Object key, V defaultValue)`: Returns the value to which the specified key is mapped, or the default value if the key is not present.

16. `computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)`: If the specified key is not already associated with a value (or is mapped to `null`), computes its value using the given mapping function and enters it into the map.

Merging:

17. `merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)`: If the specified key is not already associated with a value or is associated with `null`, associates it with the given non-null value. Otherwise, computes a new value by applying the given remapping function to the existing value and the given value, and enters the new value into the map.

View Operations:

18. `keySet()`: Returns a `Set` containing all the keys in the map.

19. `values()`: Returns a `Collection` containing all the values in the map.

20. `entrySet()`: Returns a `Set` of key-value pairs (entries) in the map.

Synchronization:

21. `synchronizedMap(Map<K, V> m)`: Returns a synchronized (thread-safe) view of the specified map.

Comparisons:

22. `equals(Object o)`: Compares the map with another object for equality.

23. `hashCode()`: Returns the hash code value for the map.

Java 8 Enhancements:

24. `forEach(BiConsumer<? super K, ? super V> action)`: Performs the given action for each entry in the map until all entries have been processed or the action throws an exception.

25. `replace(K key, V value)`: Replaces the entry for the specified key only if it is currently mapped to some value.

26. `replace(K key, V oldValue, V newValue)`: Replaces the entry for the specified key only if currently mapped to the specified value.

27. `compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)`: If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.

NavigableMap (Interface Extension):

28. `ceilingEntry(K key)`: Returns a key-value mapping with the least key greater than or equal to the given key.

29. `floorEntry(K key)`: Returns a key-value mapping with the greatest key less than or equal to the given key.

30. `higherEntry(K key)`: Returns a key-value mapping with the least key strictly greater than the given key.

31. `lowerEntry(K key)`: Returns a key-value mapping with the greatest key strictly less than the given key.

32. `pollFirstEntry()`: Removes and returns a key-value mapping with the least key.

33. `pollLastEntry()`: Removes and returns a key-value mapping with the greatest key.

34. `descendingKeySet()`: Returns a reverse order `NavigableSet` view of the keys.

35. `descendingMap()`: Returns a reverse order view of the map.

36. `subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)`: Returns a view of the portion of the map that is less than (or equal to, if `fromInclusive` is true) `toKey` and greater than (or equal to, if `toInclusive` is true) `fromKey`.

37. `headMap(K toKey, boolean inclusive)`: Returns a view of the portion

of the map that is strictly less than (or equal to, if `inclusive` is true) `toKey`.

38. `tailMap(K fromKey, boolean inclusive)`: Returns a view of the portion of the map that is greater than (or equal to, if `inclusive` is true) `fromKey`.

Additional Map Implementations:

39. `HashMap<K, V>`: A commonly used hash-based map implementation.

40. `TreeMap<K, V>`: A sorted map implementation based on a red-black tree.

41. `LinkedHashMap<K, V>`: A hash-based map with predictable iteration order based on the order of insertion.

42. `WeakHashMap<K, V>`: A map implementation that uses weak references for keys.

43. `IdentityHashMap<K, V>`: A map implementation that uses reference-equality in place of object-equality for keys.

44. `EnumMap<K extends Enum<K>, V>`: A map implementation specialized for enum keys.

45. `ConcurrentHashMap<K, V>`: A highly concurrent hash map implementation.

46. `ConcurrentSkipListMap<K, V>`: A sorted map implementation based on a skip list.

47. `Hashtable<K, V>`: A legacy synchronized map implementation.

Map.Entry:

48. `getKey()`: Returns the key corresponding to this entry.

49. `getValue()`: Returns the value corresponding to this entry.

50. `setValue(V value)`: Replaces the value corresponding to this entry with the specified value.

These methods provide a comprehensive set of operations for working with maps in Java's Collections framework. Keep in mind that map implementations offer different characteristics in terms of performance, ordering, and concurrency support.

//