

**Vector Methods:**

**To add objects**

add(Object o) method from Collection(I)

add(int index, Object o) method from List(I)

addElement(Object o) method from Vector(C)

**To remove objects**

remove(Object o) method from Collection(I)

clear() method from Collection(I)

remove(int index) method from List(I)

removeElement(Object o) method from Vector(C)

removeElementAt(int index) method from Vector(C)

removeAllElements() method from Vector(C)

**To get objects**

Object get(int index) method from List(I)

Object elementAt(int index) method from Vector(C)

Object firstElement() method from Vector(C)

Object lastElement() method from Vector(C)

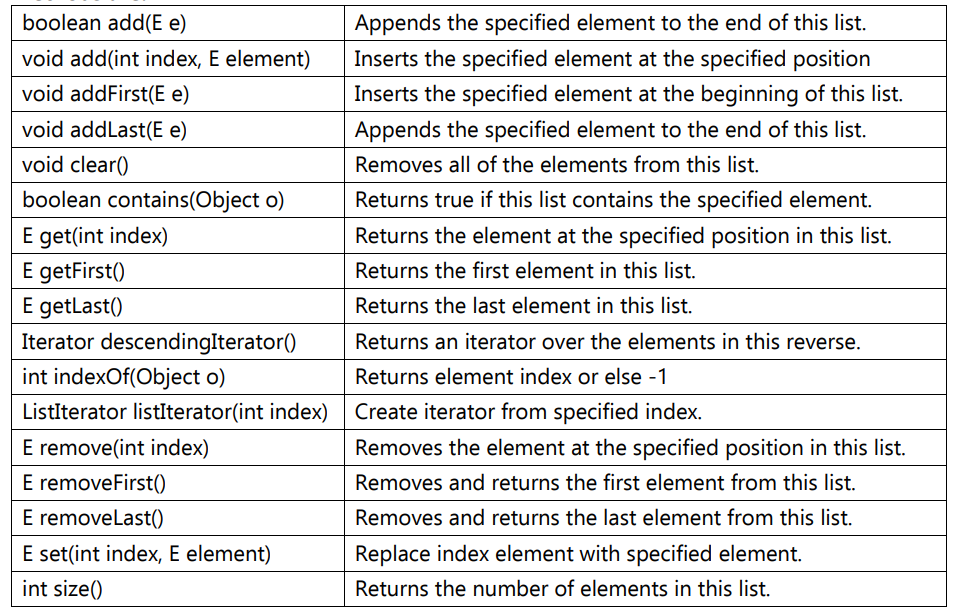
**Other Methods**

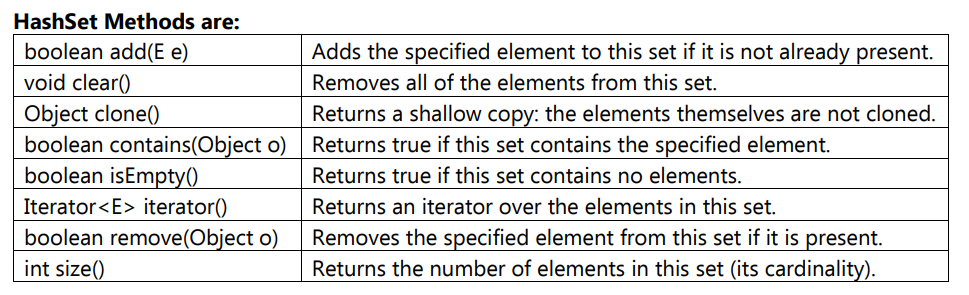
int size() – how many elements are present in the vector

int capacity() – how many objects the vector can hold

Enumeration elements() – to get the objects one by one from the collection

**LinkedList:**





SortedSet interface defines the following specific methods.

(a) Object first() - returns first element of the SortedSet

(b) Object last() - returns last element of the SortedSet

(c) SortedSet headSet(Object obj) – returns SortedSet whose elements are < obj

(d) SortedSet tailSet(Object obj) - returns SortedSet whose elements are >= obj

(e) SortedSet subSet(Object obj1, Object obj2) - returns SortedSet whose elements are >= obj1

and < obj2

(f) Comparator comparator() – return Comparator object that describes underlying sorting

technique like ascending, descending etc. If we are using default natural sorting order, then we

will get null.

**NavigableSet Interface:**

NavigableSet interface defines the following specific methods.

(a) floor(e) - returns highest element which is <= e

(b) lower(e) - returns highest element which is > e

(c) ceiling(e) - returns lowest element which is >= e

(d) higher(e) - returns lowest element which is > e

(e) pollFirst() – remove and return first element

(f) pollLast() – remove and return last element

(g) descendingSet() – returns NavigableSet in reverse order

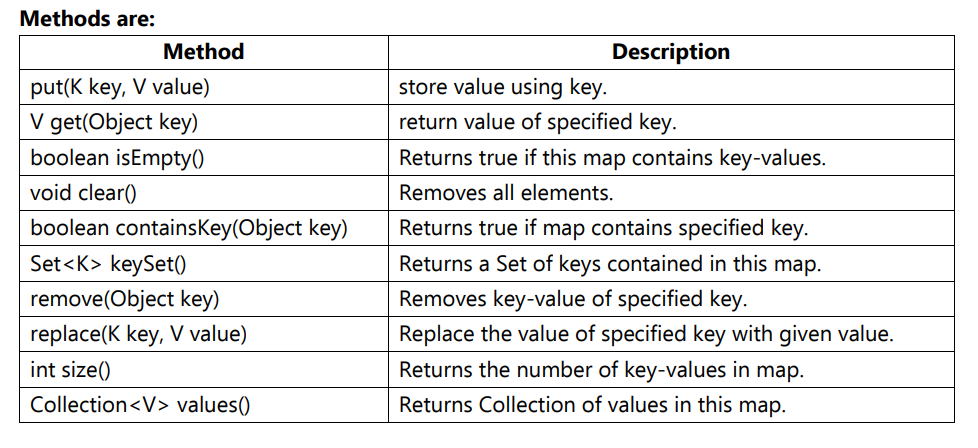
1. Write a Java Program
2. Insert the Specified Element at the Specified Position in the LinkedList
3. insert some elements at the specified position into a linked list
4. insert elements into the linked list at the first and last position
5. append the specified element to the end of a linked list
6. To insert the specified element at the front of a linked list
7. To iterate through all elements in a linked list
8. To get the first and last occurrence of the specified elements in a linked list
9. To iterate a linked list in reverse order
10. to iterate through all elements in a linked list starting at the specified position
11. to check if a particular element exists in a linked list
12. to convert a linked list to array list
13. to compare two linked lists
14. to test an linked list is empty or not
15. to replace an element in a linked list
16. to remove and return the first element of a linked list
17. to clone an linked list to another linked list
18. to join two linked lists
19. to display the elements and their positions in a linked list
20. swap two elements in a linked list

Collections.swap(list,source,destination);

1. to shuffle the elements in a linked list

Collections.shuffle(linked\_list);

**Map Interface:**



**Methods:**

(a) Object put(Object key, Object value) – to add one key value pair to the Map. If the key is

already present, then old value will be replaced with new value and returns old value.

Ex: m.put(101, “A”); // Add this entry to the Map and returns null

m.put(101, “B”);// Replace ‘A’ with ‘B’, has same key and returns ‘A’

(b) void putAll(Map p) - Add the specified Map to current Map /Ex: m1.add(m2)

(c) Object get(Object key) – Returns the value associated with specified key

(d) Object remove(Object key) – Removes the entry associated with specified key

(e) boolean containsKey(Object key) - Search the specified key in the Map

(f) boolean containsValue(Object value) - Search the specified value in the Map

(g) boolean isEmpty()

(h) int size()

(i) void clear() – All key value pairs will be removed from the Map

**Collection Views of Map Methods:**

(j) Set keySet() - returns the Set containing all the keys

(k) Collection values() – returns the Set containing all the values

(l) Set entrySet() - return the Set view containing all the keys and values

**Entry Interface:**

**Methods:**

**(a) Object getKey() – used to obtain key**

**(a) Object getValue() - used to obtain value**

**(a) Object setValue(Object o) - used to replace the old value with new value**

**SortedMap Interface:**

(a) Object firstKey() - returns first element of the SortedMap

(b) Object lastKey() - returns last element of the SortedMap

(c) SortedSet headMap(Object key) – returns SortedMap whose elements are < key

(d) SortedSet tailMap(Object key) - returns SortedMap whose elements are >= key

(e) SortedSet subMap(Object key1, Object key2) - returns SortedMap whose elements are >=

key1 and < key2

(f) Comparator comparator() – return Comparator object that describes underlying sorting

technique like ascending, descending etc. If we are using default natural sorting order, then we

will get null.

**Navigable Map Interface:**

(a) floorKey(e) - returns highest element which is <= e

(b) lowerKey(e) - returns highest element which is > e

(c) ceilingKey(e) - returns lowest element which is >= e

(d) higherKey(e) - returns lowest element which is > e

(e) pollFirstEntry() – remove and return first element

(f) pollLastEntry() – remove and return last element

(g) descendingMap() – returns NavigableMap in reverse order

1. You are given a list of students, and you need to find all the students who have scored above 90% in their exams. You can use the **ArrayList** class to store the list of students and the **Iterator** interface to iterate over the list.
2. You are given a set of employees, and you need to find all the employees who work in the same department. You can use the **HashSet** class to store the set of employees and the **contains()** method to check if an employee works in a particular department.
3. You are given a map of countries and their capitals, and you need to find all the countries that have a capital city that starts with the letter 'A'. You can use the **HashMap** class to store the map and the **keySet()** method to get the set of all the countries.
4. You are given a queue of tasks, and you need to process the tasks in the order in which they were added to the queue. You can use the **PriorityQueue** class to store the queue and the **poll()** method to process the next task in the queue.
5. You are given a stack of elements, and you need to reverse the order of the elements in the stack. You can use the **Stack** class to store the stack and the **pop()** and **push()** methods to reverse the order of the elements.

Scenario: You are working on a project that requires efficient retrieval of elements based on their insertion order. Which collection class from the Java Collection Framework would you choose and why?

The LinkedHashSet class is suitable for this scenario. It maintains the order of insertion while ensuring uniqueness.

Scenario: Your application needs to store unique elements, and you also want to perform operations like adding, removing, and checking for the presence of an element efficiently. Which collection class would be the most suitable for this requirement?

The HashSet class would be appropriate for this requirement. It allows for efficient addition, removal, and checking for the presence of elements while ensuring uniqueness.

Scenario: In a project, you need to sort a collection of objects based on their natural order or a custom order defined by a comparator. Which collection class or classes would you use to achieve this functionality?

The TreeSet class is suitable for sorting elements based on their natural order or a custom order defined by a comparator.

Scenario: You are designing a system where multiple threads will be accessing and modifying a shared collection. What measures or classes from the Collection Framework would you use to ensure thread safety in this scenario?

To ensure thread safety in a shared collection, you can use the Collections.synchronizedList or Collections.synchronizedSet methods to wrap the respective collection class (e.g., ArrayList or HashSet) and make it thread-safe.

Scenario: Your application needs to associate a unique key with a value and efficiently retrieve the value based on the key. Which collection class would you choose for this requirement?

The HashMap class is suitable for associating a unique key with a value and efficiently retrieving the value based on the key.

Scenario: In a performance-critical application, you need to frequently add and remove elements from the beginning and end of a collection. Which collection class would provide the most efficient operations for this use case?

The LinkedList class provides efficient operations for adding and removing elements from the beginning and end of the list

Scenario: You are developing a system where memory efficiency is crucial, and you need to store a large number of elements. Which collection class would you choose to minimize memory overhead?

The ArrayList class would be appropriate for scenarios where memory efficiency is crucial due to its compact storage of elements in a dynamic array.

Scenario: Your application requires a dynamic, resizable array that can grow or shrink as needed. Which collection class would be most suitable for maintaining such a data structure?

The ArrayList class is suitable for maintaining a dynamic, resizable array that can grow or shrink as needed.

Scenario: In a scenario where you need to store elements as key-value pairs and efficiently retrieve values based on keys, but the keys should be ordered either in natural order or based on a custom comparator. Which collection class would be appropriate?

The TreeMap class would be appropriate for storing elements as key-value pairs and ordering the keys either in natural order or based on a custom comparator.

Scenario: You are working on a project that involves frequently checking if a particular element exists in a collection and efficiently removing it if present. Which collection class would provide the most efficient solution for this use case?

The HashSet class provides an efficient solution for checking the presence of an element and removing it if present.

**Real Time Applications on Collection Framework:**

1. **Inventory Management System:**

**Design a simple inventory management system using Java Collections. Consider storing product information, such as name, quantity, and price. Implement functionalities to add, remove, and update product details.**

1. **Library Catalog:**

**Create a library catalog system using Java Collections. Define classes for books, authors, and categories. Implement functionalities to add books, search for books based on various criteria, and display information about available books.**

1. **Social Media Feed:**

**Design a social media feed system where each user can post messages. Use Java Collections to store posts, and implement functionalities to display a user's feed, filter posts based on criteria, and allow users to like or comment on posts.**

1. **Student Grading System:**

**Implement a student grading system using Java Collections. Store student information and their corresponding grades. Provide functionalities to calculate average grades, identify the top-performing students, and display individual student details.**

1. **Online Shopping Cart:**

**Develop an online shopping cart system using Java Collections. Store information about products, manage a user's shopping cart, and calculate the total cost. Implement functionalities for adding/removing items from the cart and generating the final invoice.**

1. **Employee Directory:**

**Design an employee directory using Java Collections. Store employee details such as name, ID, and department. Implement functionalities to search for employees based on criteria, add new employees, and update employee information.**

1. **Chat Application:**

**Build a simple chat application using Java Collections. Use a data structure to store chat messages for different users. Implement functionalities for sending and receiving messages, as well as viewing chat history.**

1. **Banking System:**

**Create a basic banking system using Java Collections. Define classes for customers, accounts, and transactions. Implement functionalities for depositing, withdrawing, and transferring money between accounts.**

1. **Weather Data Analysis:**

**Implement a weather data analysis system using Java Collections. Store weather data for different locations and implement functionalities to calculate average temperatures, identify temperature extremes, and display weather trends.**