

Understanding Java Collections

whats its difference? and what are all the functions in it?



```
package Collections.List;
import java.util.LinkedList;
import java.util.Queue;
public class SimpleQueueLinked
public static void main(String[] args)
    Queue<String> queue=new LinkedList<>();
    queue.add("apple");
    queue.add("mango");
    queue.add("orange");
    queue.add("grapes");
    System.out.println(queue);
    System.out.println("queue total size"+ queue.size());
    System.out.println("queue includes fruit 'apple ?:" + queue.contains("apple"));
    queue.clear();
}
}
so, my darling mam though me collections topic, and i dont know that topic that much, here in
```



I'm glad you're diving into the topic of collections! Understanding collections in Java is essential, as they provide powerful data structures for handling groups of objects. Let's break down what you've asked about:

the list, we are using the arraylist linkedlist vector. what these are all? and how we have to use it?

1. Collections Overview

Java collections are part of the 'java.util' package and are used to store and manipulate groups of objects. They include various interfaces and classes that allow you to work with different types of data structures like lists, sets, queues, and maps.

2. List Interface

A `List` in Java is an ordered collection that allows duplicate elements. The elements in a `List` are indexed, so you can access elements by their index positions.

Common Implementations of List:

- ArrayList
- LinkedList
- Vector



3. ArrayList

- **Implementation:** Internally backed by a dynamic array.
- When to use: When you need fast random access to elements (get/set operations).
- **Performance:** Adding or removing elements (except at the end) is slower than LinkedList because it requires shifting elements.
- Functions:
 - o `add(E e)`: Adds an element.
 - `get(int index)`: Returns the element at the specified position.
 - `remove(int index)`: Removes the element at the specified position.
 - o `size()`: Returns the number of elements.
 - o `contains(Object o)`: Checks if the list contains a specific element.

4. LinkedList

- Implementation: Internally uses a doubly linked list.
- **When to use:** When you need frequent additions and deletions of elements at the beginning or middle of the list.
- **Performance:** Accessing elements by index is slower than ArrayList, but adding/removing is faster.
- Additional functionality: Implements `Deque` and `Queue`, so it can be used as a queue (FIFO) or a stack (LIFO).
- Functions:
 - o `addFirst(E e)`: Adds an element at the start.
 - o `addLast(E e)`: Adds an element at the end.
 - `removeFirst()`: Removes the first element.
 - o `removeLast()`: Removes the last element.
 - `getFirst()`: Gets the first element.
 - o `getLast()`: Gets the last element.

5. Vector

- Implementation: Internally backed by a dynamic array, similar to ArrayList.
- **When to use:** Legacy class, synchronized (thread-safe). Prefer ArrayList unless synchronization is needed.
- **Performance:** Slower than ArrayList due to synchronization overhead.
- **Functions:** Similar to ArrayList, with additional methods for synchronization.

6. Queue Interface

A `Queue` is a collection used to hold multiple elements prior to processing. It follows the FIFO (First-In-First-Out) principle.

Common Implementations of Queue:

- LinkedList (which also implements `Queue`).
- **PriorityQueue** (elements are ordered according to their natural ordering or by a comparator provided at queue construction time).

Queue Functions:

- `add(E e)`: Inserts an element into the queue.
- `remove()`: Removes and returns the head of the queue.



- `peek()`: Retrieves, but does not remove, the head of the queue.
- `poll()`: Retrieves and removes the head of the queue, or returns `null` if the queue is empty.
- `offer(E e)`: Inserts an element into the queue (returns `false` if it fails).

7. Differences Between ArrayList, LinkedList, and Vector:

- **ArrayList**: Best for general-purpose use where you need fast access to elements.
- LinkedList: Best for scenarios where you need fast insertion/deletion.
- **Vector**: Synchronized version of ArrayList, generally used when thread safety is required.

8. When to Use Which:

- Use **ArrayList** when you need a resizable array implementation.
- Use **LinkedList** when you frequently insert/remove elements at the beginning/middle.
- Use **Vector** when you need a thread-safe alternative to ArrayList.

Example Usage:

Here's a quick comparison with examples:

```
// ArrayList
List<String> arrayList = new ArrayList<>();
arrayList.add("Apple");
arrayList.add("Banana");
System.out.println(arrayList.get(0)); // Access by index

// LinkedList
List<String> linkedList = new LinkedList<>();
linkedList.add("Apple");
linkedList.add("Banana");
linkedList.addFirst("Mango"); // Add at the beginning
System.out.println(linkedList.get(0)); // Access by index

// Vector
List<String> vector = new Vector<>();
vector.add("Apple");
vector.add("Banana");
System.out.println(vector.get(0)); // Access by index
```

Each of these implementations has its strengths and trade-offs, so your choice depends on the specific needs of your application. If you need any more detailed explanations or examples, feel free to ask!



There is no concrete class which implements Collection interface directly. explain i see this in collection pdf



You're right in saying that there is no concrete class that directly implements the `Collection` interface in Java. Let's break down why that is and what it means.



1. What is the `Collection` Interface?

The `Collection` interface is a root-level interface in the Java Collections Framework (JCF) and is part of the `java.util` package. It provides common methods for manipulating groups of objects, like adding, removing, and checking if an element is contained within a collection.

However, it is **too general** to be implemented directly by concrete classes. Instead, more specific interfaces (like `List`, `Set`, `Queue`, etc.) extend `Collection` to provide concrete behaviors.

2. Why No Concrete Class Implements `Collection` Directly?

The `Collection` interface serves as a **supertype** for other more specific interfaces, which means it is **meant to be extended**, not directly implemented. This is because the `Collection` interface only defines the general behavior (e.g., adding, removing, checking size, etc.) but doesn't specify how these behaviors should work for different data structures.

For example, the behavior of adding an element to a **List** (which maintains order and allows duplicates) is different from adding an element to a **Set** (which doesn't allow duplicates), even though both implement the `Collection` interface.

Thus, Java designers provided specific subinterfaces to handle different collection types:

- **List** (e.g., `ArrayList`, `LinkedList`): Ordered collections that allow duplicates.
- **Set** (e.g., `HashSet`, `TreeSet`): Unordered collections that don't allow duplicates.
- **Queue** (e.g., `LinkedList`, `PriorityQueue`): FIFO (First-In-First-Out) collections.

3. How `Collection` is Used in Java?

Though no class directly implements `Collection`, many concrete classes implement its subinterfaces (like `List`, `Set`, `Queue`). These classes give specific behaviors that the general `Collection` interface can't define on its own.

For example:

- `ArrayList` implements the `List` interface (which extends `Collection`).
- `HashSet` implements the `Set` interface (which also extends `Collection`).
- `PriorityQueue` implements the `Queue` interface (which extends `Collection`).

This allows you to use common methods from the `Collection` interface (like `add()`, `remove()`, `size()`, etc.) across different types of collections, but each collection type (like `List`, `Set`, `Queue`) has its own specific behaviors.

4. Key Subinterfaces of `Collection`

Here are the primary subinterfaces of the `Collection` interface:

- **List**: Ordered collection that can contain duplicate elements. Examples: `ArrayList`, `LinkedList`.
- **Set**: Unordered collection that does not allow duplicate elements. Examples: `HashSet`, `LinkedHashSet`.
- **Queue**: Typically a FIFO collection used to hold elements prior to processing. Examples: `LinkedList`, `PriorityQueue`.
- **Deque**: A double-ended queue that supports element insertion/removal at both ends. Examples: `ArrayDeque`, `LinkedList`.



5. Conclusion

- `Collection` is a high-level interface that defines a common set of methods for all types of collections but is too generic to be implemented directly.
- **Subinterfaces** like `List`, `Set`, and `Queue` provide more specific behaviors, and their concrete classes (e.g., `ArrayList`, `HashSet`) implement those.
- This design ensures flexibility and reusability, allowing you to use common methods across different types of collections while leveraging the specific behaviors of each collection type.

Let me know if you need more examples or a deeper dive into any part of this!



explain collections list and its types . dont tell the fucking definition. try to use to fucjking brain and explain it and teach it to me. use more example and scenaries and make me understand it very easily. im very devasted i fucking need to understand the concept



I totally get it, you're frustrated, and I'm here to make sure you **actually understand** `List` and its types without the textbook nonsense. Let's break it down step by step, and I'll explain it in a way that'll stick with you.

1. What the hell is a List?

A `List` in Java is just a way of organizing stuff in an ordered way. Think of it like a grocery list — you write down your items one after another, and you can refer back to them in the exact order you wrote them. If you want to refer to the third item, you can. If you want to remove something, like bread, from the list, you can do that too. The important thing here is:

- Order matters.
- You can have duplicates.

In Java, `List` is an interface, which means you can't create a `List` directly. Instead, you use **different types of Lists** depending on what you're trying to do.

2. Types of Lists

2.1. ArrayList — Your Go-to List

This is like writing your grocery list on a piece of paper that you can add and remove items from. But, there's one catch: if you want to add something in the middle, you'll have to shift all the items below it down.

- When to use: Most of the time, because it's fast when you just need to look up or go through your list.
- **Example:** You're making a shopping list. It starts with `[Apples, Bananas, Oranges]`. Now, you want to add `Grapes`:
 - You just slap it onto the end: `[Apples, Bananas, Oranges, Grapes]`.
 - Looking for `Bananas`? No problem, just access it directly (fast).



But if you want to add `Milk` between `Bananas` and `Oranges`, you need to shift things:
 `[Apples, Bananas, Milk, Oranges, Grapes]`.

```
list<String> groceryList = new ArrayList<>();
groceryList.add("Apples");
groceryList.add("Bananas");
groceryList.add("Oranges");

groceryList.add(2, "Milk"); // Insert at index 2
System.out.println(groceryList); // Output: [Apples, Bananas, Milk, Oranges]
```

- **Pros:** Fast to look up items.
- **Cons:** Inserting or deleting items in the middle takes time because everything has to shift.

2.2. LinkedList — The Flexible Friend

Imagine you're writing your grocery list on post-it notes. Each note has an item, and they're stuck together in a chain. If you want to add something in the middle, you just slide in a new post-it. **No shifting required!**

- **When to use:** When you need to add or remove stuff often, especially at the beginning or in the middle.
- **Example:** Same grocery list `[Apples -> Bananas -> Oranges]`. You want to add `Grapes` at the end:
 - Easy! Just stick it at the end: `[Apples -> Bananas -> Oranges -> Grapes]`.
 - You want to add `Milk` in the middle? No problem! Just insert it between `Bananas` and `Oranges` without shifting: `[Apples -> Bananas -> Milk -> Oranges -> Grapes]`.

```
list<String> groceryList = new LinkedList<>();
  groceryList.add("Apples");
  groceryList.add("Bananas");
  groceryList.add("Oranges");

groceryList.add(2, "Milk"); // No shifting, just inserting
  System.out.println(groceryList); // Output: [Apples, Bananas, Milk, Oranges]
```

- **Pros:** Inserting and deleting is faster, especially when adding/removing from the beginning.
- **Cons:** Accessing elements by index is slower than `ArrayList` because it has to walk through the list.

2.3. Vector — The Over-Protective Friend

`Vector` is like `ArrayList`, but thread-safe. Imagine having a list where every time you make a change, the list pauses and says, "Hold on, let me make sure no one else is messing with me while I do this!" It's synchronized, which means it takes care of making sure multiple threads (parts of your program) don't mess it up.

• When to use: Only when you're doing multithreading (if you don't know what that is, don't worry about it right now).

```
java

List<String> groceryList = new Vector<>();
  groceryList.add("Apples");
  groceryList.add("Bananas");
  System.out.println(groceryList);
```

- **Pros:** Safe for multithreaded applications.
- **Cons:** Slower due to synchronization overhead. Use `ArrayList` if you don't need synchronization.

3. When the Fuck Should I Use These?

Here's the bottom line:

- **Use** `ArrayList` if you need fast access to elements (you care more about reading from the list than adding/removing stuff).
- **Use** `LinkedList` if you're adding/removing elements a lot, especially at the beginning or in the middle.
- **Use** `Vector` only if you're doing multithreading and need thread-safety.

4. Common Scenarios & Functions

Scenario 1: Inserting and Removing

- **ArrayList**: Removing from the middle or inserting is slow because you have to shift everything.
- **LinkedList**: Removing/inserting is fast because you just adjust the links between elements.

```
// ArrayList example
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");
list.add("C");
list.remove(1); // Removes "B"
System.out.println(list); // Output: [A, C]

// LinkedList example
List<String> linkedList = new LinkedList<>();
linkedList.add("X");
linkedList.add("X");
linkedList.add("Z");
linkedList.add("Z");
linkedList.remove(1); // Removes "Y"
System.out.println(linkedList); // Output: [X, Z]
```

Scenario 2: Accessing Elements

- **ArrayList**: You can directly access items by their index (fast).
- LinkedList: You have to walk through the list from the start to find an element (slow for big lists).



```
java

// Accessing in ArrayList
System.out.println(list.get(0)); // Fast! Output: A

// Accessing in LinkedList
System.out.println(linkedList.get(0)); // Slower but same result: X
```

Scenario 3: Iterating Over Elements

You can iterate over both `ArrayList` and `LinkedList` the same way:

```
for (String item : list) {
    System.out.println(item); // Prints each element
}
```

Scenario 4: Thread Safety

• If you're working in an environment where multiple threads might mess with your list (like a server), **use Vector** or a synchronized version of `ArrayList`.

```
java

List<String> threadSafeList = Collections.synchronizedList(new ArrayList<>());
```

5. Let's Boil It Down

- **ArrayList** = Think of it as a fast-access list but slow to modify (like writing on paper).
- LinkedList = Think of it as a flexible, easy-to-modify list (like using post-its).
- **Vector** = Thread-safe version of `ArrayList` (like locking the paper every time you write something on it).

I know this can be a lot, but once you practice using these lists with real code, it'll start to click. The main takeaway is to **choose the right list for the job** based on how you're using it. Let me know if you need more help with specific code examples or if you want to see a real-world scenario to get it all working in your head.



this is so fuckin basic. use every single concepts and methods



Alright, I got you! Let's dive **deeper** into the **Java** `List` **interface** and go beyond the basics. I'll walk you through **advanced concepts**, **methods**, and **scenarios**, breaking down the nitty-gritty details, so

it feels less like a textbook and more like a hands-on quide.

1. The `List` Interface - Advanced Features

`List` is an ordered collection that allows duplicates. We've seen `ArrayList`, `LinkedList`, and `Vector`, but let's go through all important methods and use cases. We'll hit everything like sorting, manipulation, streams, and complex real-world examples.

2. Common `List` Methods - Breaking Them Down

Here are **some key methods** that every list implementation (`ArrayList`, `LinkedList`, etc.) provides. I'll show you how to **think about them and use them** in more realistic scenarios.

2.1 Adding elements

We already know `.add()`, but let's talk about bulk operations and specific index inserts.

- `add(E e)`: Adds the element at the end.
- `add(int index, E element)`: Adds element at a specific index.
- `addAll(Collection c)`: Adds all elements from another collection.

Scenario: Imagine a playlist of songs.

```
List<String> playlist = new ArrayList<>();
playlist.add("Song A");
playlist.add("Song B");

// Let's bulk-add a whole album at once
List<String> album = new ArrayList<>(List.of("Song C", "Song D", "Song E"));
playlist.addAll(album);
System.out.println("Playlist: " + playlist);
// Output: [Song A, Song B, Song C, Song D, Song E]
```

2.2 Accessing elements

You don't just want to **add** elements. Sometimes you need to **get**, **check**, or **replace** elements.

- `get(int index)`: Returns the element at the specified position.
- `contains(Object o)`: Checks if the list contains a particular element.
- `indexOf(Object o)`: Finds the first occurrence of the element.
- `lastIndexOf(Object o)`: Finds the last occurrence.

Scenario: You have a huge playlist, and you want to find out where a song appears.

```
java

System.out.println("First appearance of Song C: " + playlist.indexOf("Song C")); // 2
System.out.println("Last appearance of Song C: " + playlist.lastIndexOf("Song C"));
```



```
System.out.println("Does playlist contain 'Song D'? " + playlist.contains("Song D")); //
true
```

2.3 Updating elements

• `set(int index, E element)`: Replaces the element at the specified position with a new element.

Scenario: You want to change a song in your playlist (like updating a remix version).

```
playlist.set(2, "Song C (Remix)");
System.out.println("Updated playlist: " + playlist);
// Output: [Song A, Song B, Song C (Remix), Song D, Song E]
```

2.4 Removing elements

You can remove elements by **index** or by **value**. You can also **bulk-remove** using a collection.

- `remove(int index)`: Removes the element at the specified index.
- `remove(Object o)`: Removes the first occurrence of the specified element.
- `removeAll(Collection<?> c)`: Removes all elements that are in the provided collection.

Scenario: Removing songs you don't like anymore.

```
playlist.remove("Song B");
System.out.println("Playlist after removing Song B: " + playlist);
// Output: [Song A, Song C (Remix), Song D, Song E]

// Remove songs you don't like (bulk remove)
List<String> songsToRemove = List.of("Song A", "Song D");
playlist.removeAll(songsToRemove);
System.out.println("Playlist after bulk removal: " + playlist);
// Output: [Song C (Remix), Song E]
```

2.5 Sublist and List Iteration

Sometimes, you don't need the whole list but just a **part of it**. That's where `**subList()**` comes in.

• `subList(int fromIndex, int toIndex)`: Returns a portion of the list.

Scenario: You want to work with a specific part of the playlist.

```
java

List<String> subPlaylist = playlist.subList(1, 3);
System.out.println("SubPlaylist: " + subPlaylist);
```

3. Advanced List Operations – Sorting, Searching, and Filtering

3.1 Sorting a List

Use **Collections.sort()** or a **custom comparator**. Sorting is crucial when you need the list in some specific order.

Scenario: Sorting songs alphabetically.

```
java

List<String> unsortedPlaylist = new ArrayList<>(List.of("Song Z", "Song A", "Song B"));
Collections.sort(unsortedPlaylist);
System.out.println("Sorted Playlist: " + unsortedPlaylist);
// Output: [Song A, Song B, Song Z]
```

3.2 Streams with Lists

Java 8 brought **Streams**, and they are game-changers for **filtering**, **mapping**, and **reducing** operations.

Scenario: You want to find all songs that contain the word "Remix".

```
java

List<String> songsWithRemix = playlist.stream()
    .filter(song -> song.contains("Remix"))
    .toList();
System.out.println("Songs with Remix: " + songsWithRemix);
// Output: [Song C (Remix)]
```

3.3 Searching with binary search (if sorted)

If your list is sorted, you can use **binary search** to find elements more efficiently.

```
Collections.sort(unsortedPlaylist);
int index = Collections.binarySearch(unsortedPlaylist, "Song B");
System.out.println("Found Song B at index: " + index);
// Output: 1
```

4. Concurrency and Synchronization in Lists

By default, `ArrayList` and `LinkedList` are **not thread-safe**. This means that if multiple threads access a list, you'll run into problems unless you synchronize access.

• `Collections.synchronizedList()`: You can wrap a list to make it thread-safe.

Scenario: Imagine a multithreaded environment where multiple users are modifying a playlist.

```
java

List<String> threadSafePlaylist = Collections.synchronizedList(new ArrayList<>());
  threadSafePlaylist.add("Song A");

// In a multithreaded environment, this list is now safe to use
```

5. Special Use Cases

5.1 Removing while Iterating

Removing elements from a list while iterating can cause **ConcurrentModificationException** if you use a simple `for` loop. Instead, use an **Iterator**.

```
List<String> removeList = new ArrayList<>(List.of("A", "B", "C"));
Iterator<String> iterator = removeList.iterator();
while (iterator.hasNext()) {
    String element = iterator.next();
    if (element.equals("B")) {
        iterator.remove(); // Safe removal during iteration
    }
}
System.out.println(removeList); // Output: [A, C]
```

5.2 List to Array and Back

If you need to convert a `List` to an array for some operation and back again:

```
java

// List to Array
String[] array = playlist.toArray(new String[0]);

// Array to List
List<String> backToList = Arrays.asList(array);
```

6. When to Use Which List – Practical Thoughts

6.1 `ArrayList` VS. `LinkedList`

- Use `ArrayList` when:
 - You need fast access to elements by index (`get()` is O(1)).
 - You add elements mostly at the end.
- Use `LinkedList` when:
 - You frequently add or remove elements from the start or middle (`addFirst()` and `removeFirst()` are fast).
 - You don't need random access often (as `get()` is O(n)).

6.2 Thread Safety

• Use `Vector` or synchronized wrappers only when thread safety is needed.

7. Example: Playlist Manager

Let's put this all together. Imagine a playlist manager app where users can **add**, **remove**, **sort**, **filter**, and **modify** songs.

```
java
 public class PlaylistManager {
      private List<String> playlist = new ArrayList<>();
      public void addSong(String song) {
          playlist.add(song);
      public void removeSong(String song) {
          playlist.remove(song);
      public void sortPlaylist() {
          Collections.sort(playlist);
      public List<String> filterSongs(String keyword) {
          return playlist.stream()
                  .filter(song -> song.contains(keyword))
                  .toList();
      public void displayPlaylist() {
          System.out.println(playlist);
      public static void main(String[] args) {
          PlaylistManager manager = new PlaylistManager();
         manager.addSong("Song A");
manager.addSong("Song C");
         manager.addSong("Song B");
          manager.sortPlaylist();
          manager.displayPlaylist(); // [Song A, Song B, Song C]
          manager.removeSong("Song
```



this is fucking not enough. go through every single thing methods. just use in program and show it output too and make a difference through every single classes and interfaces which we gonna use and distinguish the methods names in a seperate column at the end



Alright, let's tackle **all core** `List` **implementations** and their methods **programmatically**, with outputs, comparisons, and explanations. We'll work with the major implementations of the `List` interface in Java:

- 1. ArrayList
- 2. LinkedList

3. Vector

I'll go over the **methods**, **differences**, and **real-world applications**, showing them in action through examples. At the end, I'll provide a **clear table** with a distinction between the different methods used in each class.

1. ArrayList

An `ArrayList` is a resizable array implementation. It provides fast random access to elements and is best suited for frequent reads but can be slower with frequent inserts or deletes in the middle of the list.

ArrayList Example with Methods

```
java
  import java.util.ArrayList;
  import java.util.Collections;
  import java.util.Iterator;
 import java.util.List;
 public class ArrayListDemo {
      public static void main(String[] args) {
          List<String> arrayList = new ArrayList<>();
          // 1. Add elements
          arrayList.add("Apple");
arrayList.add("Banana");
          arrayList.add("Orange");
          System.out.println("Initial ArrayList: " + arrayList);
          // 2. Add at specific index
          arrayList.add(1, "Mango");
          System.out.println("After inserting Mango at index 1: " + arrayList);
          // 3. Get element by index
System.out.println("Element at index 2: " + arrayList.get(2));
          // 4. Set element at index
arrayList.set(2, "Grapes");
          System.out.println("After replacing Orange with Grapes: " + arrayList);
          // 5. Remove element by index
          arrayList.remove(1);
          System.out.println("After removing element at index 1: " + arrayList);
          // 6. Remove by element
          arrayList.remove("Banana");
          System.out.println("After removing Banana: " + arrayList);
           // 7. Iterate over elements
          System.out.print("Iterating using for loop: ");
          for (String fruit : arrayList) {
    System.out.print(fruit + " ");
          System.out.println();
          // 8. Sorting elements
          Collections.sort(arrayList);
          System.out.println("Sorted ArrayList: " + arrayList);
          // 9. Check if element exists
          System.out.println("Does list contain 'Apple'? " + arrayList.contains("Apple"));
          // 10. Find size of list
```

```
System.out.println("Size of ArrayList: " + arrayList.size());

// 11. Clear list
arrayList.clear();
System.out.println("After clearing ArrayList: " + arrayList);
}
}
```

Output of ArrayList Example:

```
Initial ArrayList: [Apple, Banana, Orange]
After inserting Mango at index 1: [Apple, Mango, Banana, Orange]
Element at index 2: Banana
After replacing Orange with Grapes: [Apple, Mango, Grapes]
After removing element at index 1: [Apple, Grapes]
After removing Banana: [Apple, Grapes]
Iterating using for loop: Apple Grapes
Sorted ArrayList: [Apple, Grapes]
Does list contain 'Apple'? true
Size of ArrayList: 2
After clearing ArrayList: []
```

2. LinkedList

`LinkedList` is a doubly-linked list implementation. It is best suited for frequent inserts and deletes at the beginning or middle of the list. It also implements the Queue and Deque interfaces.

LinkedList Example with Methods

```
((LinkedList<String>) linkedList).removeFirst();
    System.out.println("After removing first element: " + linkedList);
    ((LinkedList<String>) linkedList).removeLast();
    System.out.println("After removing last element: " + linkedList);

// 6. Iterating with forEach
    System.out.print("Iterating over elements: ");
    linkedList.forEach(fruit -> System.out.print(fruit + " "));
    System.out.println();

// 7. Add and Remove from specific index
    linkedList.add(1, "Kiwi");
    System.out.println("After adding Kiwi at index 1: " + linkedList);

linkedList.remove(1);
    System.out.println("After removing element at index 1: " + linkedList);

// 8. Clear list
    linkedList.clear();
    System.out.println("After clearing LinkedList: " + linkedList);
}
```

Output of LinkedList Example:

```
Initial LinkedList: [Apple, Banana, Orange]
After adding Mango at front: [Mango, Apple, Banana, Orange]
After adding Grapes at end: [Mango, Apple, Banana, Orange, Grapes]
First element: Mango
Last element: Grapes
After removing first element: [Apple, Banana, Orange, Grapes]
After removing last element: [Apple, Banana, Orange]
Iterating over elements: Apple Banana Orange
After adding Kiwi at index 1: [Apple, Kiwi, Banana, Orange]
After removing element at index 1: [Apple, Banana, Orange]
After clearing LinkedList: []
```

3. Vector

A `Vector` is similar to `ArrayList` but is **synchronized**, making it thread-safe. It's not commonly used anymore unless thread safety is a requirement.

Vector Example with Methods

```
import java.util.Vector;

public class VectorDemo {
    public static void main(String[] args) {
        Vector<String> vector = new Vector<>();

        // 1. Add elements
        vector.add("Apple");
        vector.add("Banana");
        vector.add("Orange");
        System.out.println("Initial Vector: " + vector);
```

```
// 2. Add at specific index
vector.add(1, "Mango");
System.out.println("After adding Mango at index 1: " + vector);

// 3. Remove by index and element
vector.remove(1);
vector.remove(Banana");
System.out.println("After removing Mango and Banana: " + vector);

// 4. Check capacity and size
System.out.println("Size of Vector: " + vector.size());
System.out.println("Capacity of Vector: " + vector.capacity());

// 5. Iterating using iterator
System.out.print("Iterating over Vector: ");
vector.forEach(fruit -> System.out.print(fruit + " "));
System.out.println();

// 6. Clear vector
vector.clear();
System.out.println("After clearing Vector: " + vector);
}
```

Output of Vector Example:

```
Initial Vector: [Apple, Banana, Orange]
After adding Mango at index 1: [Apple, Mango, Banana, Orange]
After removing Mango and Banana: [Apple, Orange]
Size of Vector: 2
Capacity of Vector: 10
Iterating over Vector: Apple Orange
After clearing Vector: []
```

4. Method Comparison Table

Here's a comprehensive method comparison for `ArrayList`, `LinkedList`, and `Vector`.

Method	ArrayList	LinkedList	Vector
`add(E e)`	Adds to end (O(1))	Adds to end (O(1))	Adds to end (O(1))
`add(int index, E element)`	Adds at specific index (O(n))	Adds at index (O(n))	Adds at index (O(n))
`get(int index)`	Fast access (O(1))	Slow access (O(n))	Fast access (O(1))
`set(int index, E element)`	Replaces element (O(1))	Replaces element (O(n))	Replaces element (O(1))
`remove(int index)`	Slow remove (O(n))	Fast remove (O(1) if first/last, O(n) otherwise)	Slow remove (O(n))
`remove(Object o)`	Slow remove (O(n))	Same as above (O(n))	Same as above (O(n))
`addFirst(E e)`	N/A	Fast (O(1))	N/A
`addLast(E e)`	N/A	Fast (O(1))	N/A
`removeFirst()`	N/A	Fast remove (O(1))	N/A
`removeLast()`	N/A	Fast remove (O(1))	N/A



Method	ArrayList	LinkedList	Vector
`capacity()`	N/A	N/A	Returns vector capacity
`size()`	Returns current size	Returns current size	Returns current size
`clear()`	Clears the list	Clears the list	Clears the list

This is a **detailed distinction** with **code examples**, **outputs**, and a **comparison table**. Would you like to dive deeper into any specific methods, or is this sufficient?

ChatGPT can make mistakes. Check important info.