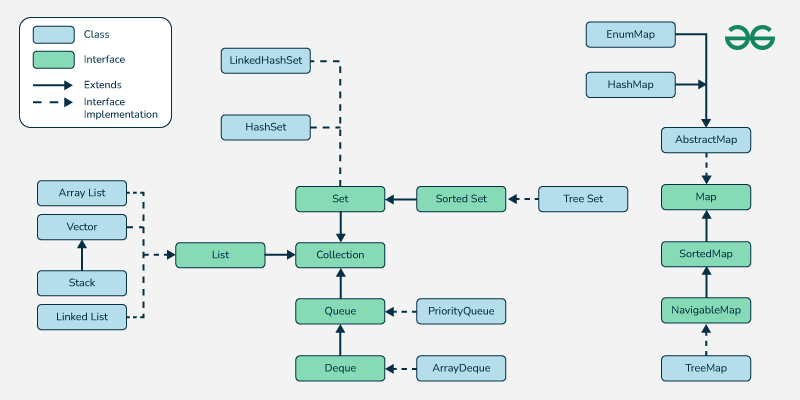
**https://www.geeksforgeeks.org/collections-class-in-java/?ref=next\_article**

**Collections in Java**

Any group of individual objects represented as a single unit is known as a Java Collection of Objects. In Java, a separate framework named the “Collection Framework” has been defined in JDK 1.2, which holds all the Java Collection Classes and Interfaces.

In Java, the Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main “root” interfaces of Java collection classes.



### What You Should Learn in Java Collections?

* [List Interface](https://www.geeksforgeeks.org/list-interface-java-examples/)
  + [Abstract List Class](https://www.geeksforgeeks.org/abstractlist-in-java-with-examples/)
  + [Abstract Sequential List Class](https://www.geeksforgeeks.org/abstractsequentiallist-in-java-with-examples/)
  + [Array List](https://www.geeksforgeeks.org/arraylist-in-java/)
  + [Vector Class](https://www.geeksforgeeks.org/java-util-vector-class-java/)
  + [Stack Class](https://www.geeksforgeeks.org/stack-class-in-java/)
  + [LinkedList Class](https://www.geeksforgeeks.org/linked-list-in-java/)
* [Queue Interface](https://www.geeksforgeeks.org/queue-interface-java/)
  + [Blocking Queue Interface](https://www.geeksforgeeks.org/blockingqueue-interface-in-java/)
  + [AbstractQueue Class](https://www.geeksforgeeks.org/abstractqueue-in-java-with-examples/)
  + [PriorityQueue Class](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/)
  + [PriorityBlockingQueue Class](https://www.geeksforgeeks.org/priorityblockingqueue-class-in-java/)
  + [ConcurrentLinkedQueue Class](https://www.geeksforgeeks.org/concurrentlinkedqueue-in-java-with-examples/)
  + [ArrayBlockingQueue Class](https://www.geeksforgeeks.org/arrayblockingqueue-class-in-java/)
  + [DelayQueue Class](https://www.geeksforgeeks.org/delayqueue-class-in-java-with-example/)
  + [LinkedBlockingQueue Class](https://www.geeksforgeeks.org/linkedblockingqueue-class-in-java/)
  + [LinkedTransferQueue](https://www.geeksforgeeks.org/linkedtransferqueue-in-java-with-examples/)
* [Deque Interface](https://www.geeksforgeeks.org/deque-interface-java-example/)
  + [BlockingDeque Interface](https://www.geeksforgeeks.org/blockingdeque-in-java/)
  + [ConcurrentLinkedDeque Class](https://www.geeksforgeeks.org/concurrentlinkeddeque-in-java-with-examples/)
  + [ArrayDeque Class](https://www.geeksforgeeks.org/arraydeque-in-java/)
* [Set Interface](https://www.geeksforgeeks.org/set-in-java/)
  + [Abstract Set Class](https://www.geeksforgeeks.org/abstractset-class-in-java-with-examples/)
  + [CopyOnWriteArraySet Class](https://www.geeksforgeeks.org/copyonwritearrayset-in-java/)
  + [EnumSet Class](https://www.geeksforgeeks.org/enumset-class-java/)
  + [ConcurrentHashMap Class](https://www.geeksforgeeks.org/concurrenthashmap-in-java/)
  + [HashSet Class](https://www.geeksforgeeks.org/hashset-in-java/)
  + [LinkedHashSet Class](https://www.geeksforgeeks.org/linkedhashset-in-java-with-examples/)
* [SortedSet Interface](https://www.geeksforgeeks.org/sortedset-java-examples/)
  + [NavigableSet Interface](https://www.geeksforgeeks.org/navigableset-java-examples/)
  + [TreeSet](https://www.geeksforgeeks.org/treeset-in-java-with-examples/)
  + [ConcurrentSkipListSet Class](https://www.geeksforgeeks.org/concurrentskiplistset-in-java-with-examples/)
* [Map Interface](https://www.geeksforgeeks.org/map-interface-java-examples/)
  + [SortedMap Interface](https://www.geeksforgeeks.org/sortedmap-java-examples/)
  + [NavigableMap Interface](https://www.geeksforgeeks.org/navigablemap-interface-in-java-with-example/)
  + [ConcurrentMap Interface](https://www.geeksforgeeks.org/concurrentmap-interface-java/)
  + [TreeMap Class](https://www.geeksforgeeks.org/treemap-in-java/)
  + [AbstractMap Class](https://www.geeksforgeeks.org/abstractmap-in-java/)
  + [ConcurrentHashMap Class](https://www.geeksforgeeks.org/concurrenthashmap-in-java/)
  + [EnumMap Class](https://www.geeksforgeeks.org/enummap-class-java-example/)
  + [HashMap Class](https://www.geeksforgeeks.org/java-util-hashmap-in-java-with-examples/)
  + [IdentityHashMap Class](https://www.geeksforgeeks.org/identityhashmap-class-java/)
  + [LinkedHashMap Class](https://www.geeksforgeeks.org/linkedhashmap-class-java-examples/)
  + [HashTable Class](https://www.geeksforgeeks.org/hashtable-in-java/)
  + [Properties Class](https://www.geeksforgeeks.org/java-util-properties-class-java/)
* Other Important Concepts
  + [How to convert HashMap to ArrayList](https://www.geeksforgeeks.org/how-to-convert-hashmap-to-arraylist-in-java/)
  + [Randomly select items from a List](https://www.geeksforgeeks.org/randomly-select-items-from-a-list-in-java/?ref=rp)
  + [How to add all items from a collection to an ArrayList](https://www.geeksforgeeks.org/how-to-add-all-items-from-a-collection-to-an-arraylist-in-java/?ref=rp)
  + [Conversion of Java Maps to List](https://www.geeksforgeeks.org/conversion-of-java-maps-to-list/)
  + [Array to ArrayList Conversion](https://www.geeksforgeeks.org/array-to-arraylist-conversion-in-java/?ref=rp)
  + [ArrayList to Array Conversion](https://www.geeksforgeeks.org/arraylist-array-conversion-java-toarray-methods/?ref=rp)
  + [Differences between Array and ArrayList](https://www.geeksforgeeks.org/array-vs-arraylist-in-java/?ref=rp)

## **What is a Framework in Java?**

A framework is a set of classes and interfaces which provide a ready-made architecture. In order to implement a new feature or a class.

### **Why Need for a Separate Collection Framework in Java?**

The need for a separate **Collection Framework** in Java arises due to several limitations of earlier approaches for handling data structures like arrays and older classes such as Vector and Hashtable. Here’s why a distinct Collection Framework was introduced in Java:

### **1. Limitations of Arrays**

* **Fixed Size:** Arrays have a fixed size that must be specified at the time of creation. This makes them inflexible for scenarios where the number of elements is dynamic.
* **Type Limitations:** Arrays are type-specific but lack robust support for generic data types, leading to potential type safety issues before Java 1.5 (before generics were introduced).
* **Lack of Utility Methods:** Arrays don't provide built-in methods for common operations like sorting, searching, or resizing. Developers had to write these utilities manually.

### **2. Lack of Unified API**

* Before the Collection Framework, classes like Vector, Hashtable, and arrays worked independently, with no standard way to handle them consistently.
* This led to redundancy, as developers had to learn different APIs and implementations for similar tasks.

### **3. Code Reusability and Maintainability**

* Without a standard framework, data structures and algorithms were often reimplemented by developers, reducing code reusability.
* Maintaining and debugging custom data structure implementations could be time-consuming and error-prone.

### **4. Improved Performance**

* The older classes (Vector, Hashtable) were synchronized by default, leading to unnecessary overhead in single-threaded environments.
* The Collection Framework provides alternatives like ArrayList and HashMap, which are more performant due to their non-synchronized nature, with thread-safe versions available when needed (e.g., Collections.synchronizedList()).

### **5. Type Safety and Generics**

* The introduction of generics in the Collection Framework (from Java 1.5) ensures type safety by allowing collections to specify the type of objects they hold, reducing runtime errors caused by typecasting.

### **6. Rich Utility Methods**

* The Collection Framework provides a rich set of utility methods through the Collections and Arrays classes for tasks like:
  + Sorting
  + Searching
  + Thread-safe wrappers
  + Bulk operations like addAll, removeAll, etc.

### **7. Support for Advanced Data Structures**

* The Collection Framework supports a wide range of data structures like:
  + Lists (ArrayList, LinkedList)
  + Sets (HashSet, TreeSet)
  + Queues (PriorityQueue, Deque)
  + Maps (HashMap, TreeMap)

These enable developers to choose the best-suited structure for their use case.

### **8. Interoperability and Polymorphism**

* The Collection Framework is built around interfaces (List, Set, Map, etc.) that allow interoperability and flexibility.
* Code written to use an interface like List can work with any implementation (ArrayList, LinkedList) without modification.

### **9. Standardized Exception Handling**

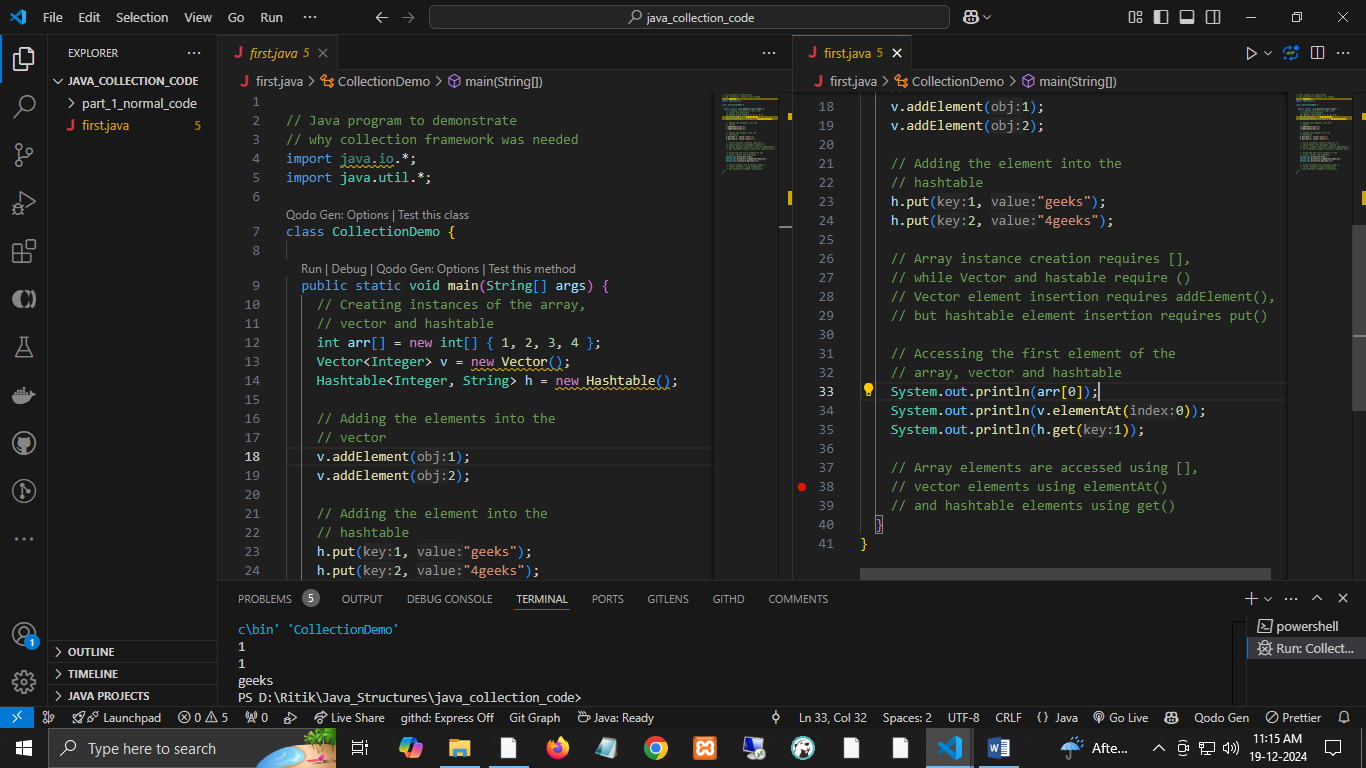
* The Collection Framework introduces standardized exceptions like ConcurrentModificationException and UnsupportedOperationException, improving debugging and error handling.

### **10. Extensibility**

* The framework is extensible, allowing developers to create their custom collection implementations by implementing the provided interfaces.

### Conclusion

The Java Collection Framework was introduced to address the limitations of older data structures and provide a **standard, reusable, and efficient API** for working with collections. It enhances productivity, performance, and maintainability by offering a unified approach to handling dynamic data.



Here in this example🡪

As we can observe, none of these collections(Array, Vector, or Hashtable) implements a standard member access interface, it was very difficult for programmers to write algorithms that can work for all kinds of Collections. Another drawback is that most of the ‘Vector’ methods are final, meaning we cannot extend the ’Vector’ class to implement a similar Collection. Therefore, Java developers decided to come up with a common interface to deal with the problems mentioned above and introduced the Collection Framework in JDK 1.2, after which both legacy Vectors and Hashtables were modified to conform to the Collection Framework.

**Is there any predefined method available for an array in java to insert data into that array or delete or update ?**

**Ans->**

In Java, arrays are fixed in size once they are created. Therefore, there are no predefined methods to directly **insert**, **delete**, or **update** elements in an array. However, you can achieve these operations using manual logic or by using more flexible data structures like ArrayList (from the Java Collections Framework), which provides methods to dynamically manage data.

Here’s an overview of your options:

### **1. Updating an Array**

You can directly update an element in the array by accessing it via its index.

int[] arr = {10, 20, 30, 40, 50};

arr[2] = 100; // Update the 3rd element to 100

System.out.println(Arrays.toString(arr)); // Output: [10, 20, 100, 40, 50]

### **2. Inserting into an Array**

To "insert" an element, you need to:

* Create a new array with a larger size.
* Copy the original array elements to the new array.
* Insert the new element at the desired position.

Example:

import java.util.Arrays;

public class Main {

public static void main(String[] args) {

int[] arr = {10, 20, 30, 40};

int newElement = 25;

int position = 2; // Insert at index 2

int[] newArr = new int[arr.length + 1]; // Create a new array with extra space

for (int i = 0, j = 0; i < newArr.length; i++) {

if (i == position) {

newArr[i] = newElement; // Insert the new element

} else {

newArr[i] = arr[j++]; // Copy old elements

}

}

System.out.println(Arrays.toString(newArr)); // Output: [10, 20, 25, 30, 40]

}

}

### **3. Deleting from an Array**

Similarly, to "delete" an element, you create a new array without the element you want to remove.

Example:

import java.util.Arrays;

public class Main {

public static void main(String[] args) {

int[] arr = {10, 20, 30, 40};

int position = 2; // Delete the element at index 2

int[] newArr = new int[arr.length - 1]; // Create a smaller array

for (int i = 0, j = 0; i < arr.length; i++) {

if (i != position) {

newArr[j++] = arr[i]; // Copy elements, skipping the one to delete

}

}

System.out.println(Arrays.toString(newArr)); // Output: [10, 20, 40]

}

}

### **4. Using** ArrayList **(Recommended for Dynamic Arrays)**

If you need flexibility, consider using ArrayList, which allows you to dynamically add, remove, and update elements without the need to manually resize arrays.

Example:

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<Integer> list = new ArrayList<>();

list.add(10); // Add elements

list.add(20);

list.add(30);

list.add(1, 15); // Insert 15 at index 1

System.out.println(list); // Output: [10, 15, 20, 30]

list.remove(2); // Remove element at index 2

System.out.println(list); // Output: [10, 15, 30]

list.set(1, 50); // Update element at index 1

System.out.println(list); // Output: [10, 50, 30]

}

}

### **Key Takeaways**

* **For arrays**: Manual logic is required for inserting or deleting elements.
* **For dynamic resizing and easy operations**: Use ArrayList or other Java Collection classes like LinkedList for convenience.

Would you like more detailed examples or clarification?

### **Advantages of the Java Collection Framework**

The **Java Collection Framework (JCF)** provides a unified architecture for managing and manipulating collections of objects. Here are the key advantages:

### **1. Unified Architecture**

* **Consistent API:** The framework provides standard interfaces like List, Set, Map, and their implementations, ensuring consistency across different types of collections.
* **Ease of Learning:** Since all collections follow a common structure, developers can easily learn and use different collection types.

### **2. Reduces Programming Effort**

* **Reusable Data Structures:** Developers don't need to implement custom data structures like linked lists or hash tables. The framework provides ready-to-use implementations.
* **Pre-built Algorithms:** Built-in methods like sort, search, and shuffle reduce the need to write complex algorithms.

### **3. Improved Performance**

* **Optimized Implementations:** The framework provides high-performance, optimized implementations of commonly used data structures and algorithms.
* **Dynamic Storage:** Collections like ArrayList automatically resize, saving the effort of managing array sizes manually.

### **4. Type Safety with Generics**

* Generics in collections ensure **type safety** at compile time, reducing runtime errors:
* List<String> list = new ArrayList<>();
* list.add("Hello");
* // list.add(123); // Compile-time error

### **5. Supports Polymorphism**

* Developers can code against interfaces (e.g., List, Set, Map) instead of specific implementations (e.g., ArrayList, HashSet), making code more flexible and extensible:
* List<String> list = new LinkedList<>();

### **6. Reduces Code Complexity**

* **Utility Classes:** The Collections and Arrays utility classes provide static methods to perform common operations (e.g., sorting, binary search) with minimal code.
* Example:
* Collections.sort(list);

### **7. Enhanced Maintainability**

* **Readability:** The framework's standardized naming conventions and interfaces improve code readability.
* **Extensibility:** Developers can extend existing classes or create their custom collections easily.

### **8. Thread-Safety Support**

* Classes like Vector and Hashtable are thread-safe, and utilities like Collections.synchronizedList() make it easy to create synchronized collections when needed.

### **9. Interoperability**

* The framework allows easy **interconversion** between different types of collections:
* Set<String> set = new HashSet<>(list);

### **10. Rich Functionality**

* **Multiple Implementations:** Offers various data structures, such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap, etc., for different use cases.
* **Sorting & Searching:** Supports custom sorting using comparators or natural ordering.

### **11. Backward Compatibility**

* JCF is compatible with older, pre-collection classes like Vector and Enumeration, ensuring legacy code can still work seamlessly.

### **12. Minimal Memory Overhead**

* The framework provides lightweight implementations, like ArrayList, which consume less memory compared to manually implemented data structures.

### **13. Supports Streams (Java 8+)**

* The framework integrates seamlessly with **Streams API** to perform functional-style operations like filtering, mapping, and reducing:
* list.stream().filter(s -> s.startsWith("A")).forEach(System.out::println);

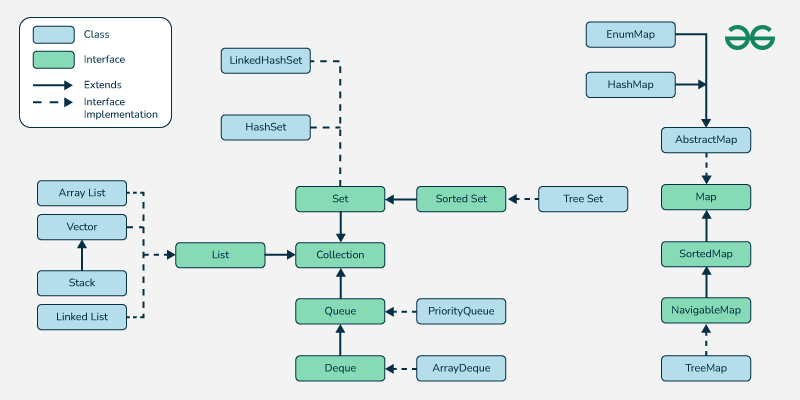
### **14. Widely Used in Libraries**

* Many Java libraries and frameworks, such as Spring and Hibernate, heavily rely on collections, making familiarity with the framework essential.

### **Conclusion**

The Java Collection Framework simplifies and standardizes how developers work with groups of objects, making code more efficient, readable, and maintainable. It provides a rich set of tools that are flexible enough to meet a wide range of application needs.

## **Hierarchy of the Collection Framework in Java**

* The utility package, (java.util) contains all the classes and interfaces that are required by the collection framework.
* The collection framework contains an interface named an iterable interface which provides the iterator to iterate through all the collections. This interface is extended by the main collection interface which acts as a root for the collection framework.
* All the collections extend this collection interface thereby extending the properties of the iterator and the methods of this interface.
* The following figure illustrates the hierarchy of the collection framework.
* 
* [**Class**](https://www.geeksforgeeks.org/classes-objects-java/)**:** A class is a user-defined blueprint from which objects are created. It represents the set of properties or methods common to all objects of one type.
* [**Interface**](https://www.geeksforgeeks.org/interfaces-in-java/)**:** Like a class, an interface can have methods and variables, but the methods declared in an interface are, by default, abstract (only method signature, nobody). Interfaces specify what a class must do and not how. It is the blueprint of the class.

## Methods of the Collection Interface

The **Collection** interface in Java is the root interface of the Java Collections Framework and provides standard methods that all collection types (like List, Set, etc.) share. While **Collection** does not have a sort() method, sorting can be achieved using the **Collections utility class**. Below is an explanation of relevant procedures for the Collection interface, followed by instructions on how to sort a collection in descending order.

### **Methods of the** Collection **Interface**

Here are the commonly used methods of the Collection interface:

| **Method** | **Description** |
| --- | --- |
| add(E e) | Adds the specified element to the collection. |
| addAll(Collection<? extends E> c) | Adds all elements of the specified collection to this collection. |
| clear() | Removes all elements from the collection. |
| contains(Object o) | Returns true if the collection contains the specified element. |
| containsAll(Collection<?> c) | Returns true if the collection contains all elements of the specified collection. |
| isEmpty() | Checks if the collection is empty (true if it is empty, false otherwise). |
| iterator() | Returns an iterator to traverse the elements in the collection. |
| remove(Object o) | Removes a single instance of the specified element from the collection. |
| removeAll(Collection<?> c) | Removes all elements in this collection that are also in the specified collection. |
| retainAll(Collection<?> c) | Retains only the elements in this collection that are contained in the specified collection. |
| size() | Returns the number of elements in the collection. |
| toArray() | Converts the collection into an array. |
| toArray(T[] a) | Converts the collection into an array of the specified type. |

### **Sorting a Collection in Descending Order**

As mentioned earlier, the Collection interface does not provide a sort() method directly. Sorting is typically performed on **lists**, which implement the List interface. You can use the **Collections.sort()** method along with a custom comparator to sort in descending order.

#### **Steps to Sort in Descending Order:**

1. Convert the collection to a List (if it's not already a List).
2. Use **Collections.sort()** with a custom comparator or use a lambda expression for descending order.

#### **Example 1: Sorting a List in Descending Order**

import java.util.\*;

public class Main {

public static void main(String[] args) {

// Create a list

List<Integer> numbers = Arrays.asList(5, 2, 9, 1, 7);

// Sort the list in descending order

Collections.sort(numbers, Collections.reverseOrder());

// Print the sorted list

System.out.println("Sorted in Descending Order: " + numbers);

}

}

**Output:**

Sorted in Descending Order: [9, 7, 5, 2, 1]

#### **Example 2: Sorting with a Custom Comparator**

import java.util.\*;

public class Main {

public static void main(String[] args) {

// Create a list of strings

List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "Dave");

// Sort in descending order using a custom comparator

Collections.sort(names, (a, b) -> b.compareTo(a));

// Print the sorted list

System.out.println("Sorted in Descending Order: " + names);

}

}

**Output:**

Sorted in Descending Order: [Dave, Charlie, Bob, Alice]

#### **Example 3: Sorting a Set in Descending Order**

Since Set does not maintain order, you need to convert it to a List first:

import java.util.\*;

public class Main {

public static void main(String[] args) {

// Create a set

Set<Integer> set = new HashSet<>(Arrays.asList(5, 2, 9, 1, 7));

// Convert to a list

List<Integer> list = new ArrayList<>(set);

// Sort the list in descending order

Collections.sort(list, Collections.reverseOrder());

// Print the sorted list

System.out.println("Sorted in Descending Order: " + list);

}

}

**Output:**

Sorted in Descending Order: [9, 7, 5, 2, 1]

#### **Sorting Summary**

* Use **Collections.reverseOrder()** for descending order.
* Ensure the collection type supports sorting (e.g., convert Set to List).
* Use custom comparators for complex sorting logic.

**Q. List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "Dave"); is this new way to create a list in java**

Yes, Arrays.asList() is a **convenient way to create a fixed-size list** in Java, but it's not a new feature. It has been part of Java since **Java 1.2**. However, it's important to understand its characteristics and limitations.

### **Key Features of** Arrays.asList()

1. **Fixed Size:**
   * The list returned by Arrays.asList() is **backed by the array**, meaning you cannot add or remove elements. Attempting to do so will throw an UnsupportedOperationException.
2. List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "Dave");
3. names.add("Eve"); // Throws UnsupportedOperationException
4. **Modifiable Elements:**
   * You can modify existing elements in the list, but you cannot change its size.
5. List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "Dave");
6. names.set(1, "Eve"); // Works
7. System.out.println(names); // Output: [Alice, Eve, Charlie, Dave]
8. **Backed by Array:**
   * Changes to the array will reflect in the list and vice versa.
9. String[] array = {"Alice", "Bob"};
10. List<String> names = Arrays.asList(array);
11. array[0] = "Eve";
12. System.out.println(names); // Output: [Eve, Bob]

### **Alternatives for Creating Lists**

Since Java 9, there are newer and more flexible ways to create lists:

#### **1.** List.of() **(Immutable Lists - Java 9+)**

* Creates an immutable list (cannot modify elements or size).

List<String> names = List.of("Alice", "Bob", "Charlie", "Dave");

names.add("Eve"); // Throws UnsupportedOperationException

names.set(1, "Eve"); // Throws UnsupportedOperationException

#### **2.** ArrayList **(Resizable and Modifiable)**

* Use ArrayList for a fully dynamic list where you can add or remove elements.

List<String> names = new ArrayList<>(Arrays.asList("Alice", "Bob", "Charlie", "Dave"));

names.add("Eve"); // Works

System.out.println(names); // Output: [Alice, Bob, Charlie, Dave, Eve]

#### **3.** Stream **API (Java 8+)**

* Use streams to collect elements into a list.

List<String> names = Stream.of("Alice", "Bob", "Charlie", "Dave")

.collect(Collectors.toList());

### **When to Use** Arrays.asList()

* **Quick Initialization:** When you need a quick, fixed-size list for reading or modifying elements but don't need to add/remove items.
* **Backed by Array:** When you want changes in the array to reflect in the list (or vice versa).

### **When NOT to Use** Arrays.asList()

* If you need a resizable list (ArrayList is better).
* If you need an immutable list (List.of() is better).

**Q. then what is ArrayList<Integer> al = new ArrayList<Integer>();**

The statement ArrayList<Integer> al = new ArrayList<Integer>(); is a way to create a **dynamic, resizable list** in Java using the ArrayList class. Unlike Arrays.asList(), which returns a fixed-size list, an ArrayList is part of the **Java Collections Framework** and allows for adding, removing, and modifying elements dynamically.

### **What is an** ArrayList**?**

* **ArrayList** is a class in Java that implements the List interface and is part of java.util package.
* It uses a **dynamic array** internally to store elements, which means its size can grow or shrink as needed.
* It provides flexibility and is commonly used when working with frequently changing data collections.

### **Key Features of** ArrayList

1. **Resizable:**
   * Unlike arrays, ArrayList can automatically resize itself when elements are added or removed.
2. ArrayList<Integer> al = new ArrayList<>();
3. al.add(10);
4. al.add(20);
5. al.add(30);
6. System.out.println(al); // Output: [10, 20, 30]
7. **Supports Duplicates:**
   * An ArrayList allows duplicate elements.
8. ArrayList<Integer> al = new ArrayList<>();
9. al.add(10);
10. al.add(10);
11. System.out.println(al); // Output: [10, 10]
12. **Zero-based Indexing:**
    * Elements in an ArrayList are indexed starting from 0, just like arrays.
13. System.out.println(al.get(0)); // Output: 10
14. **Allows Null Elements:**
    * ArrayList can store null values.
15. al.add(null);
16. System.out.println(al); // Output: [10, 10, null]
17. **Type-Safe with Generics:**
    * By specifying <Integer>, the ArrayList is restricted to storing only Integer objects, ensuring type safety at compile time.
18. ArrayList<Integer> al = new ArrayList<>();
19. al.add(10);
20. // al.add("String"); // Compilation error: incompatible types

### **Difference Between** ArrayList **and** Arrays.asList()

| **Feature** | **ArrayList** | **Arrays.asList()** |
| --- | --- | --- |
| **Size** | Resizable (can add/remove elements). | Fixed-size (cannot add/remove elements). |
| **Mutability** | Entirely mutable (add, remove, modify). | Elements are mutable, but size is not. |
| **Creation** | Created explicitly using new. | Created from an existing array or values. |
| **Use Case** | When you need a dynamic, resizable list. | When you need a quick, fixed-size list. |

### **Common Methods of** ArrayList

Here are some key methods you can use with ArrayList:

| **Method** | **Description** |
| --- | --- |
| add(E e) | Appends the specified element to the end of the list. |
| add(int index, E e) | Inserts the specified element at the specified position in the list. |
| get(int index) | Returns the element at the specified position. |
| remove(int index) | Removes the element at the specified position. |
| remove(Object o) | Removes the first occurrence of the specified element. |
| size() | Returns the number of elements in the list. |
| set(int index, E e) | Replaces the element at the specified position with the specified element. |
| isEmpty() | Returns true if the list is empty, false otherwise. |
| clear() | Removes all elements from the list. |
| contains(Object o) | Returns true if the list contains the specified element. |
| indexOf(Object o) | Returns the index of the first occurrence of the specified element, or -1. |

### **Example: Using** ArrayList

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

// Create an ArrayList of integers

ArrayList<Integer> al = new ArrayList<>();

// Add elements

al.add(10);

al.add(20);

al.add(30);

// Access elements

System.out.println("Element at index 1: " + al.get(1)); // Output: 20

// Modify an element

al.set(1, 25);

// Remove an element

al.remove(2);

// Iterate through the ArrayList

System.out.println("ArrayList elements:");

for (int num : al) {

System.out.println(num);

}

}

}

**Output:**

Element at index 1: 20

ArrayList elements:

10

25

### **Why Use** ArrayList**?**

1. When you need a **dynamic size** that can grow or shrink.
2. To perform frequent **additions or removals** at the end of the list.
3. For its support for **random access** using the get() method, as it has an O(1) time complexity for this operation.

Let me know if you'd like further clarification or more examples!

## Q. Interfaces that Extend the Java Collections Interface

### **1. [Iterable Interface](https://www.geeksforgeeks.org/iterable-interface-in-java/)**

* This is the root interface for the entire collection framework.
* The collection interface extends the iterable interface. Therefore, all interfaces and classes implement this interface inherently.
* This interface's main functionality is to provide an iterator for the collections. Therefore, it contains only one abstract method: the iterator.
* It returns the

Iterator iterator();

### **2.**[**Collection Interface**](https://www.geeksforgeeks.org/collection-interface-in-java-with-examples/)

* This interface extends the iterable interface and is implemented by all the classes in the collection framework.
* This interface contains all the basic methods every collection has, such as adding, removing, clearing, etc.
* This interface implements all these methods because they are implemented by all the classes irrespective of their implementation style. Also, having these methods in this interface ensures that the names of the methods are universal for all the collections.
* Therefore, in short, we can say that this interface builds a foundation for implementing the collection classes.

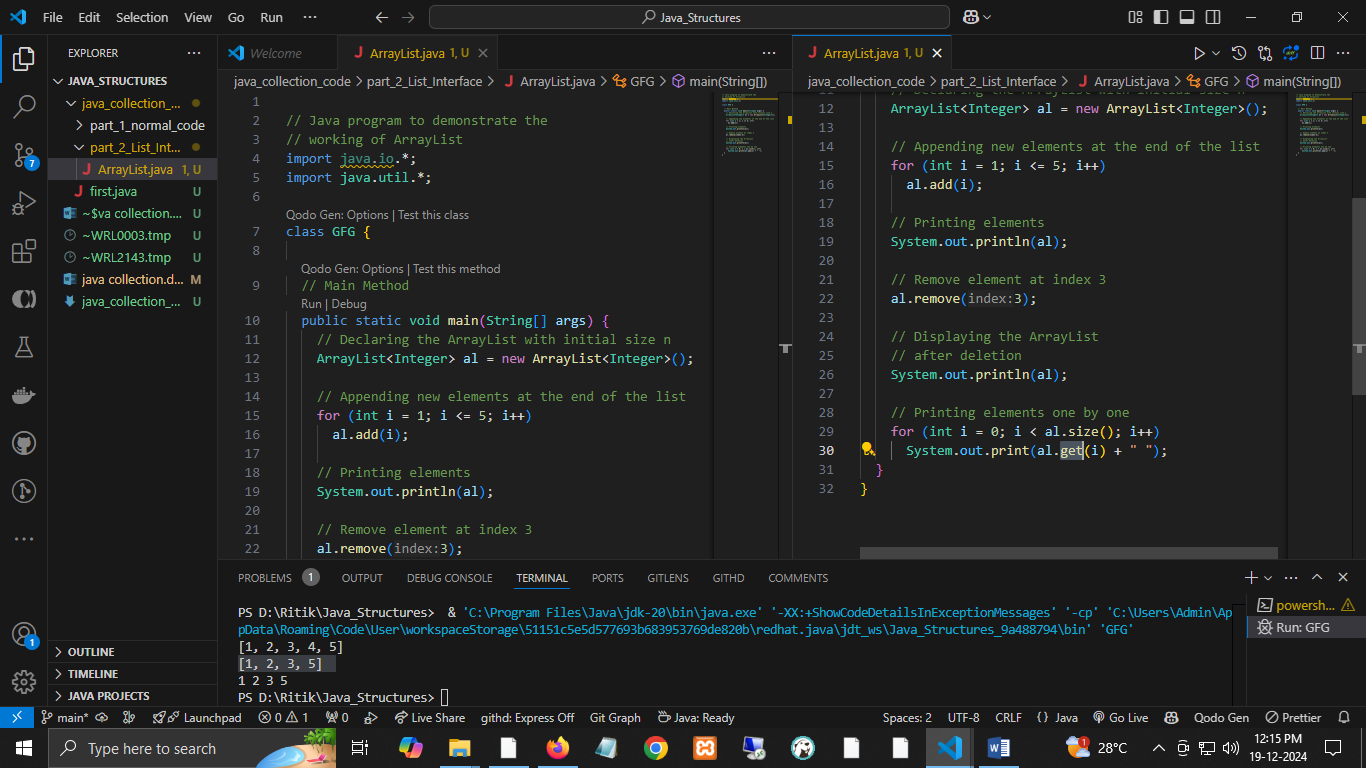
### **3.**[**List Interface**](https://www.geeksforgeeks.org/list-interface-java-examples/)

* This is a child interface of the collection interface.
* This interface is dedicated to list-type data, in which we can store all the ordered collections of objects.
* This also allows duplicate data to be present in it. Various classes like ArrayList, Vector, Stack, etc, implement this list interface. Since all the subclasses implement the list, we can instantiate a list object with any of these classes.
* **Important\* allow duplicate data and ordered data.**
* **For example:**
* List <T> al = new ArrayList<> ();   
  List <T> ll = new LinkedList<> ();   
  List <T> v = new Vector<> ();   
  Where T is the type of the object

***The classes that implement the List interface are as follows:***

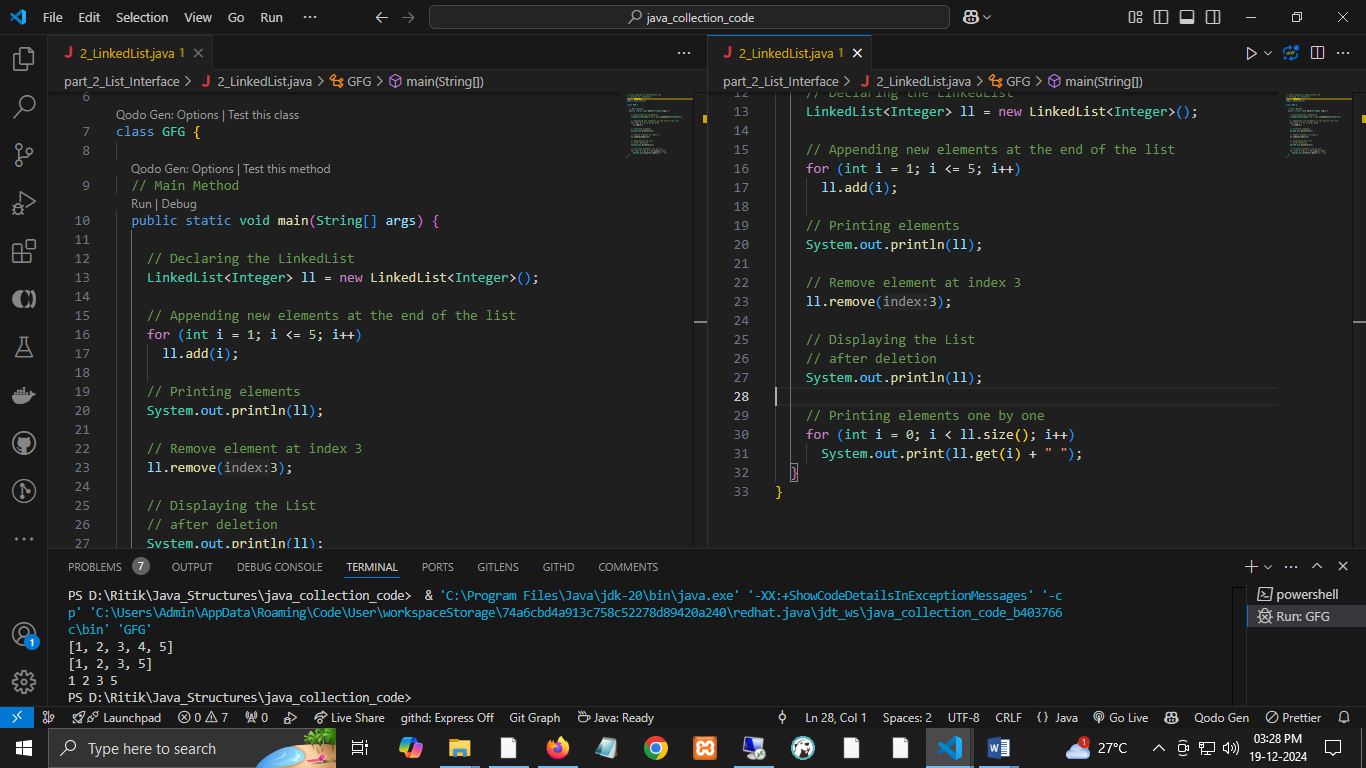
#### **i). [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/)**

* ArrayList provides us with dynamic arrays in Java. Though it may be slower than standard arrays, it can be helpful in programs where lots of manipulation in the array is needed.
* The size of an ArrayList is increased automatically if the collection grows or shrinks if the objects are removed from the collection.
* Java ArrayList allows us to access the list randomly.
* ArrayList can not be used for [primitive types](https://www.geeksforgeeks.org/primitive-data-type-vs-object-data-type-in-java-with-examples/), like int, char, etc. We will need a [wrapper class](https://www.geeksforgeeks.org/wrapper-classes-java/) for such cases.
* Let’s understand the ArrayList with the following example: in the code folder.

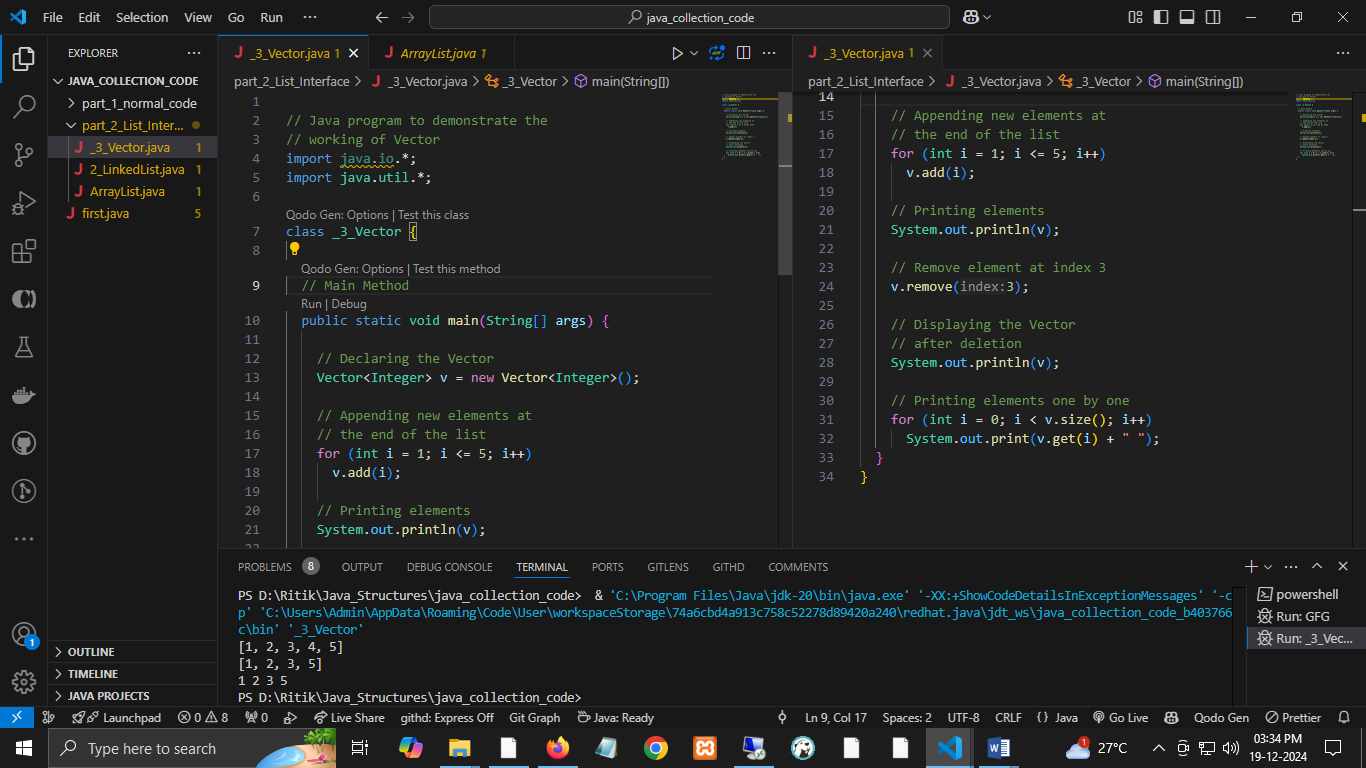


#### **ii).**[**LinkedList**](https://www.geeksforgeeks.org/linked-list-in-java/)

* The LinkedList class implements the LinkedList data structure, a linear data structure in which the elements are not stored in contiguous locations, and every element is a separate object with a data part and an address part.
* The elements are linked using pointers and addresses.
* Each element is known as a node.
* Let’s understand the LinkedList with the following example:

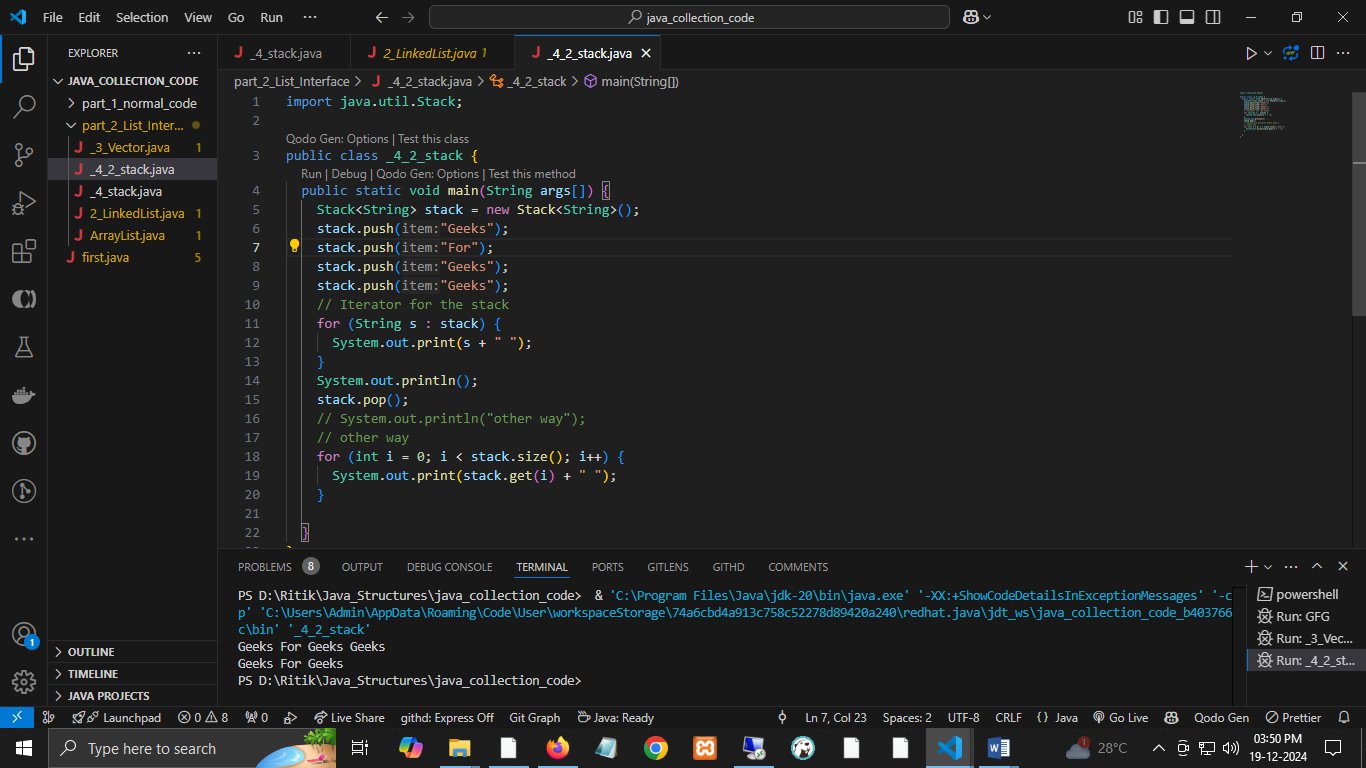


#### **iii).**[**Vector**](https://www.geeksforgeeks.org/java-util-vector-class-java/)

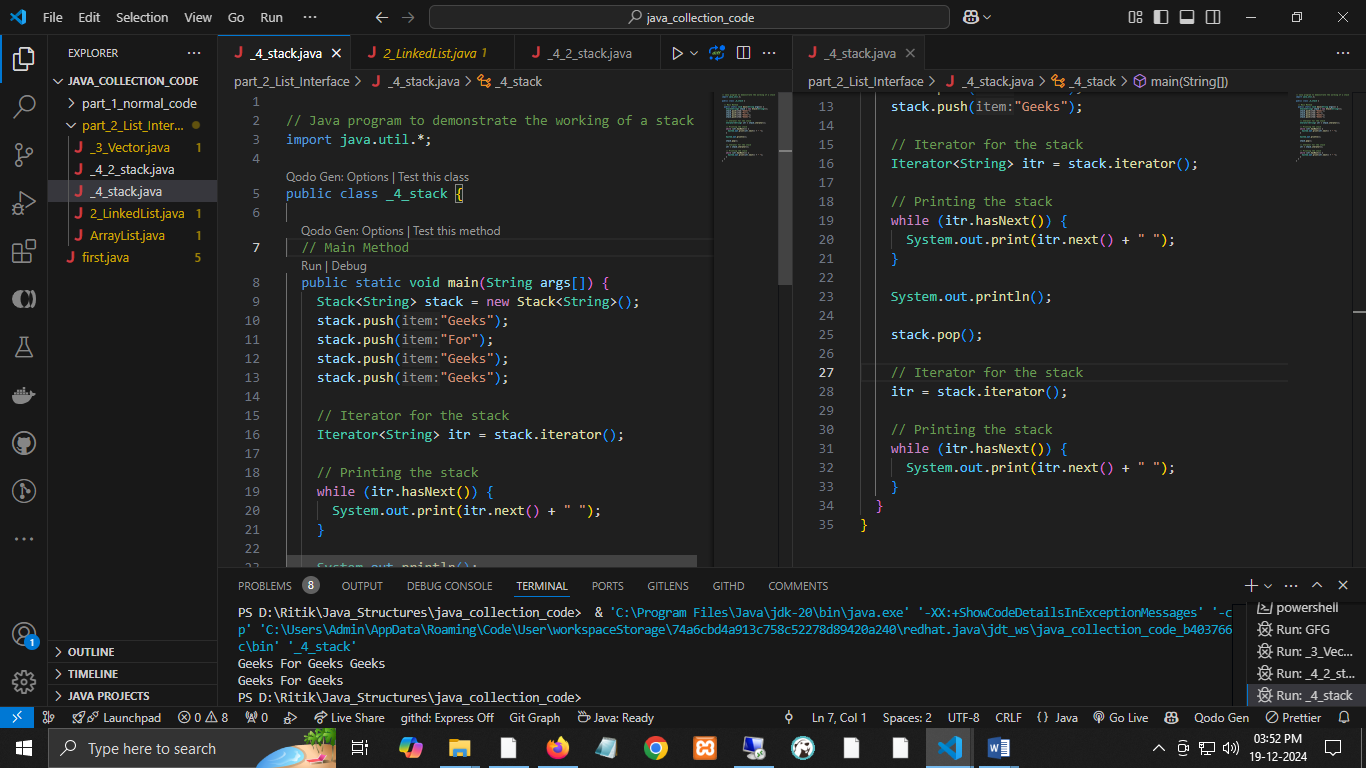
* A vector provides us with dynamic arrays in Java. Though it may be slower than standard arrays, it can be helpful in programs where lots of manipulation in the array is needed.
* This is identical to ArrayList in terms of implementation. However, the primary difference between a vector and an ArrayList is that a Vector is synchronized, and an ArrayList is non-synchronized.
* Let’s understand the Vector with an example:
* 

#### **iv).**[**Stack**](https://www.geeksforgeeks.org/stack-class-in-java/)

* Stack class models and implements the [Stack data structure](https://www.geeksforgeeks.org/stack-data-structure/).
* The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, it provides three more functions: empty, search, and peek.
* This class can also be referred to as a subclass of Vector.
* Let’s understand the stack with an example:



Using Iterator and while loop



***Note:*** *Stack is a subclass of Vector and a legacy class. It is thread-safe, which might be overhead in an environment where thread safety is not needed. An alternative to Stack is to use ArrayDequeue, which is not thread-safe but has a faster array implementation.*

### **4.**[**Queue Interface**](https://www.geeksforgeeks.org/queue-interface-java/)

* As the name suggests, a queue interface maintains the FIFO(First In First Out) order like a real-world queue line.
* This interface is dedicated to storing all the elements where the order of the elements matters. For example, whenever we try to book a ticket, the tickets are sold on a first-come, first-serve basis. Therefore, the person whose request arrives first in the queue gets the ticket.
* There are various classes like [PriorityQueue](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/), [ArrayDeque](https://www.geeksforgeeks.org/arraydeque-in-java/), etc. Since all these subclasses implement the queue, we can instantiate a queue object with any of these classes.

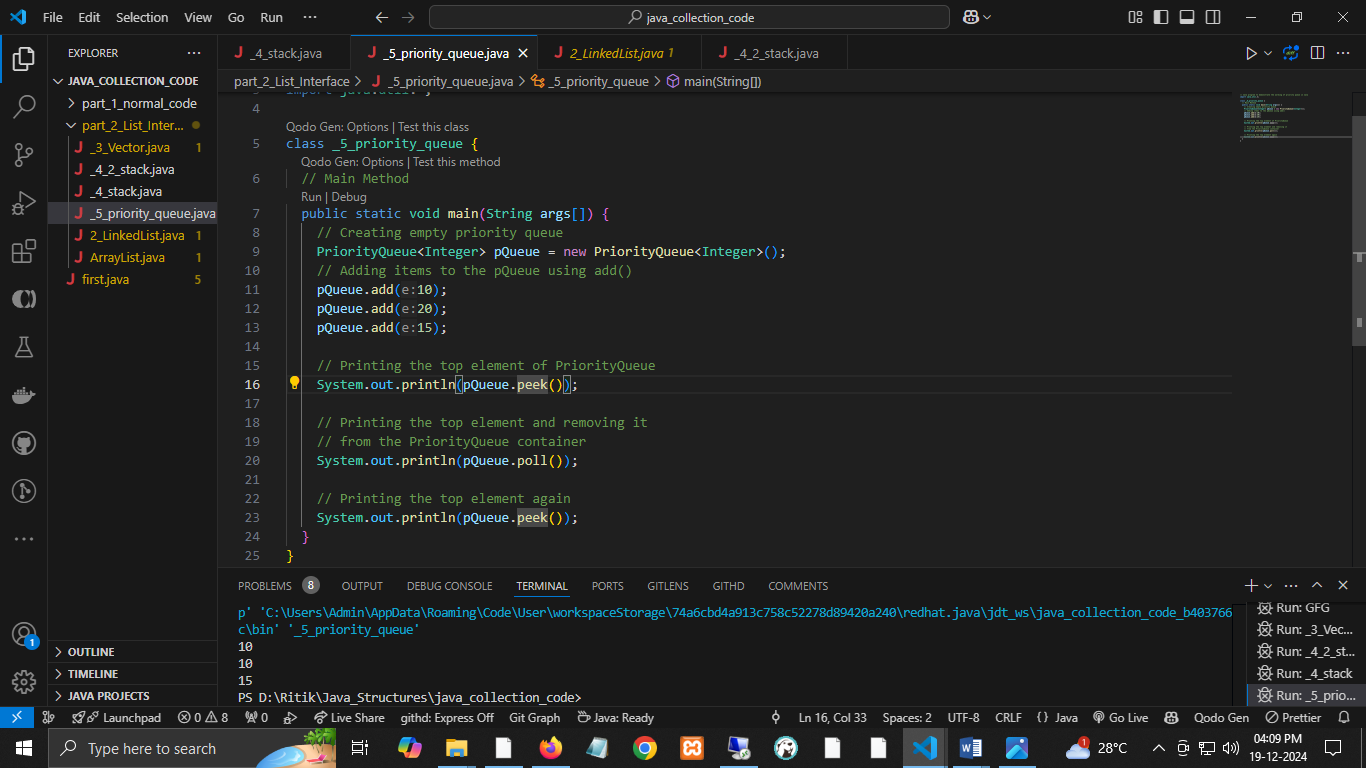
**For example:**

Queue <T> pq = new PriorityQueue<> ();   
Queue <T> ad = new ArrayDeque<> ();   
Where T is the type of the object.

***The most frequently used implementation of the queue interface is the PriorityQueue.***

#### [**Priority Queue**](https://www.geeksforgeeks.org/priority-queue-class-in-java/)

* A PriorityQueue is used when the objects are supposed to be processed based on priority.
* It is known that a queue follows the First-In-First-Out algorithm, but sometimes, the queue elements need to be processed according to priority, and this class is used in these cases.
* The Priority Queue is based on the priority heap. Depending on which constructor is used, the elements of the priority queue are ordered according to natural ordering or by a Comparator provided at queue construction time.
* Let’s understand the priority queue with an example:



### **5.**[**Deque Interface**](https://www.geeksforgeeks.org/deque-interface-java-example/)

* This is a very slight variation of the [queue data structure](https://www.geeksforgeeks.org/queue-data-structure/).
* Deque, also known as a double-ended queue, is a data structure in which elements can be added and removed from both ends.
* This interface extends the queue interface.
* The class which implements this interface is ArrayDeque. Since ArrayDeque class implements the Deque interface, we can instantiate a deque object with this class.

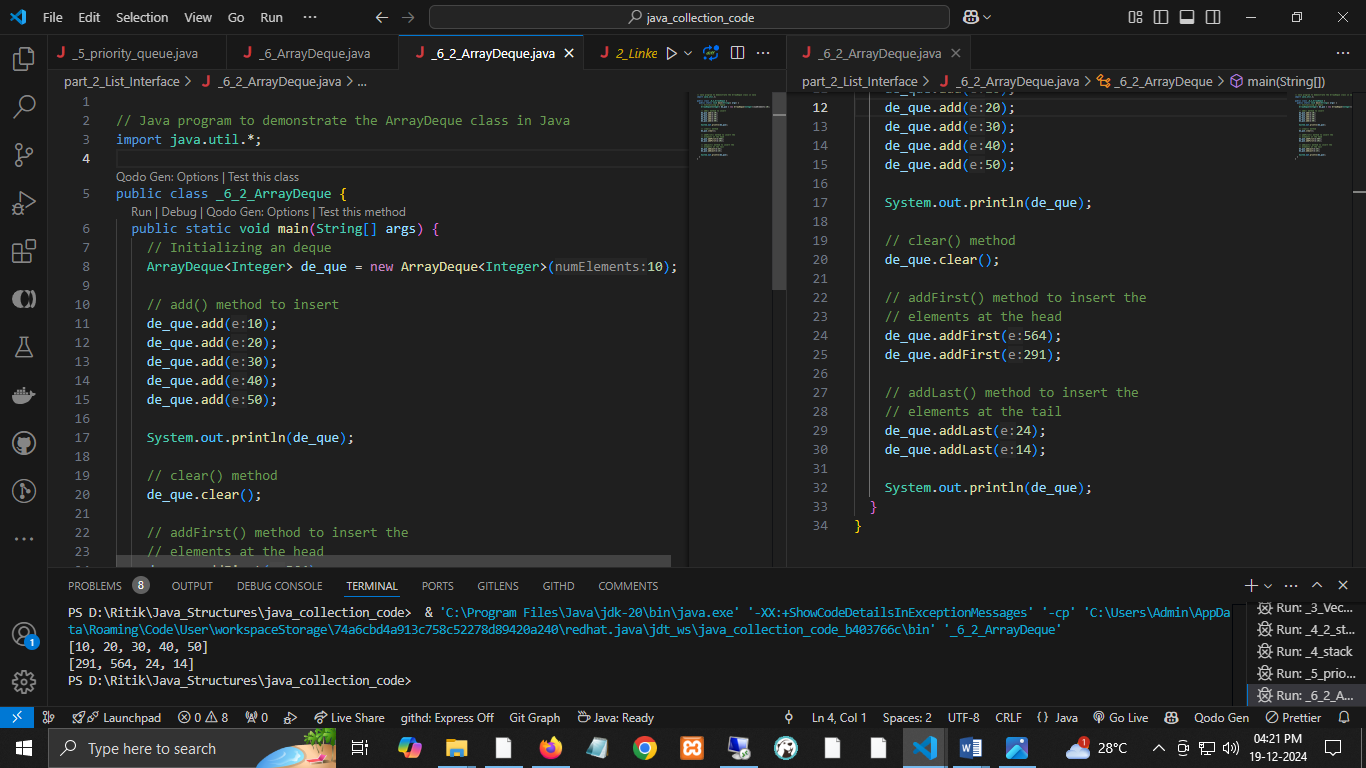
**For example:**

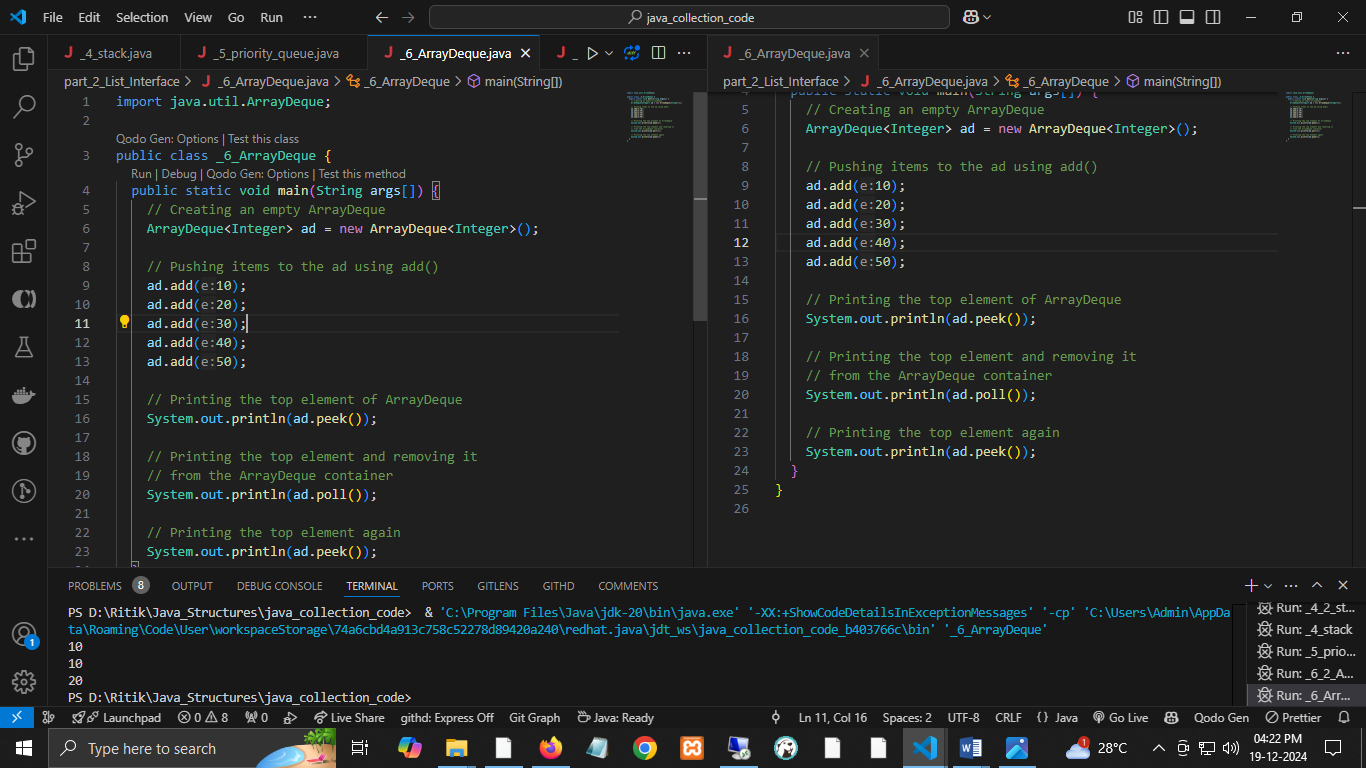
Deque<T> ad = new ArrayDeque<> ();   
Where T is the type of the object.

***The class that implements the deque interface is ArrayDeque.***

[**ArrayDeque**](https://www.geeksforgeeks.org/arraydeque-in-java/)

* The ArrayDeque class, implemented in the collection framework, allows us to apply a resizable array.
* This special array grows and allows users to add or remove an element from both sides of the queue.
* Array deques have no capacity restrictions, and they grow as necessary to support usage.
* Let’s understand ArrayDeque with an example:





### **6.**[**Set Interface**](https://www.geeksforgeeks.org/set-in-java/)

* A set is an unordered collection of objects in which duplicate values cannot be stored.
* This collection is used when we wish to avoid duplication and store only unique objects.
* This set interface is implemented by various classes, such as HashSet, TreeSet, LinkedHashSet, etc. Since all the subclasses implement the set, we can instantiate a set object with any of these classes.

For example:

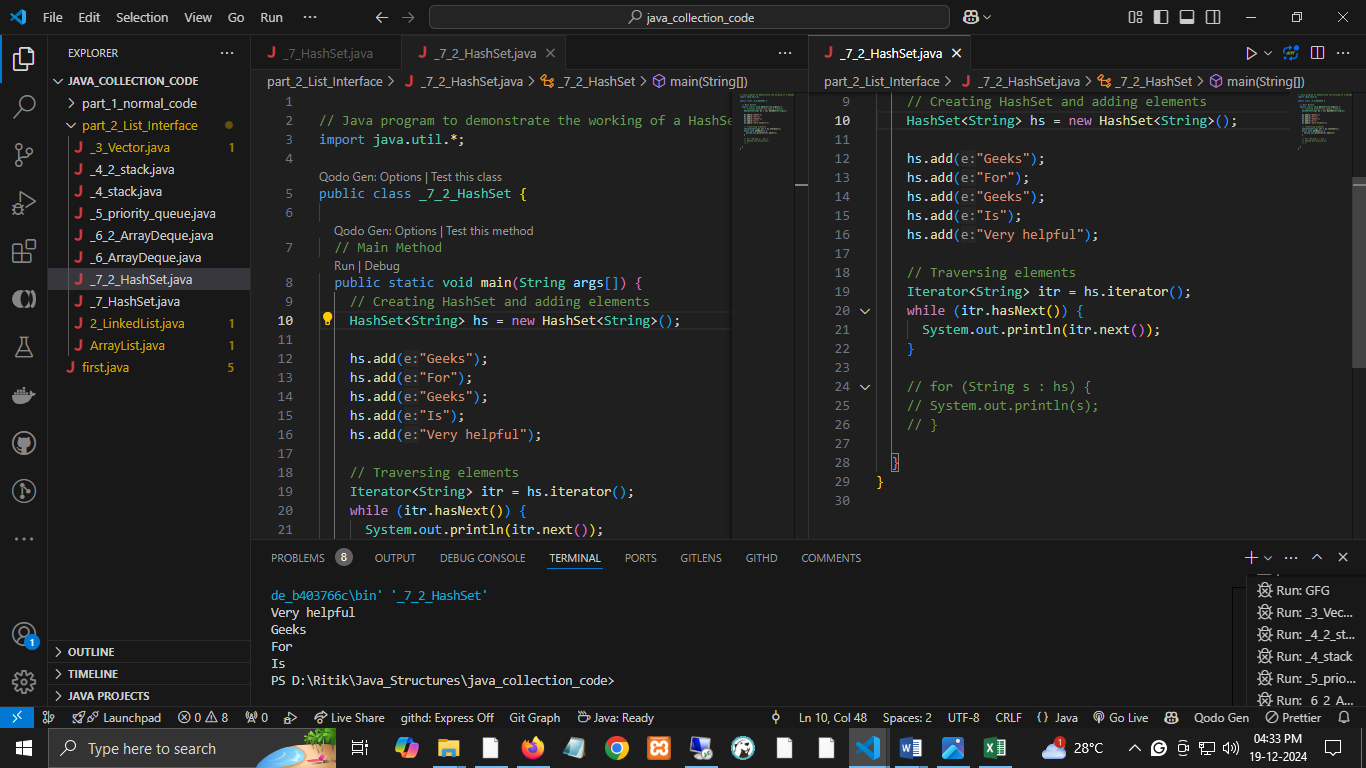
Set<T> hs = new HashSet<> ();   
Set<T> lhs = new LinkedHashSet<> ();   
Set<T> ts = new TreeSet<> ();   
Where T is the type of the object.

***The following are the classes that implement the Set interface:***

#### **i).**[**HashSet**](https://www.geeksforgeeks.org/hashset-in-java/)

* The HashSet class is an inherent hash table data structure implementation. The objects we insert into the HashSet are not guaranteed to be inserted in the same order.
* The objects are inserted based on their hashcode.
* This class also allows the insertion of NULL elements.

Let’s understand HashSet with an example:

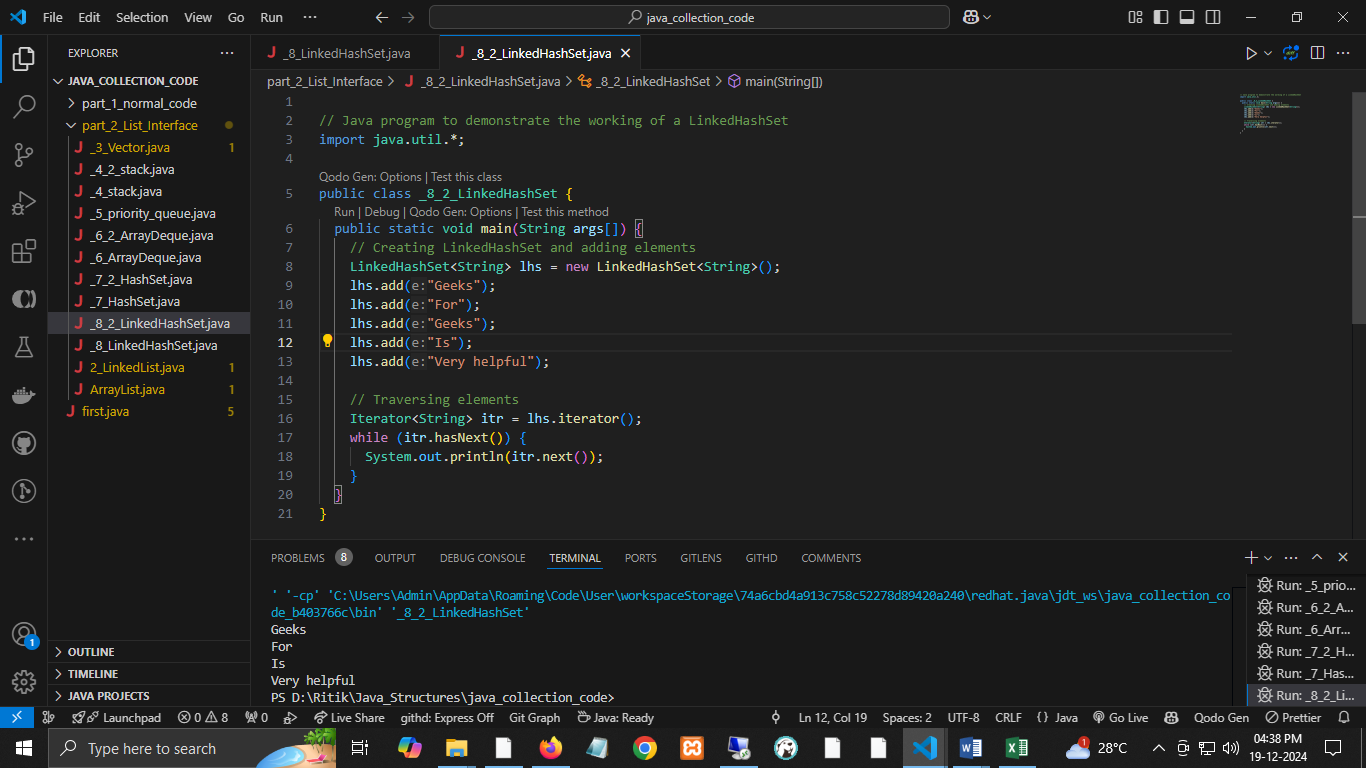


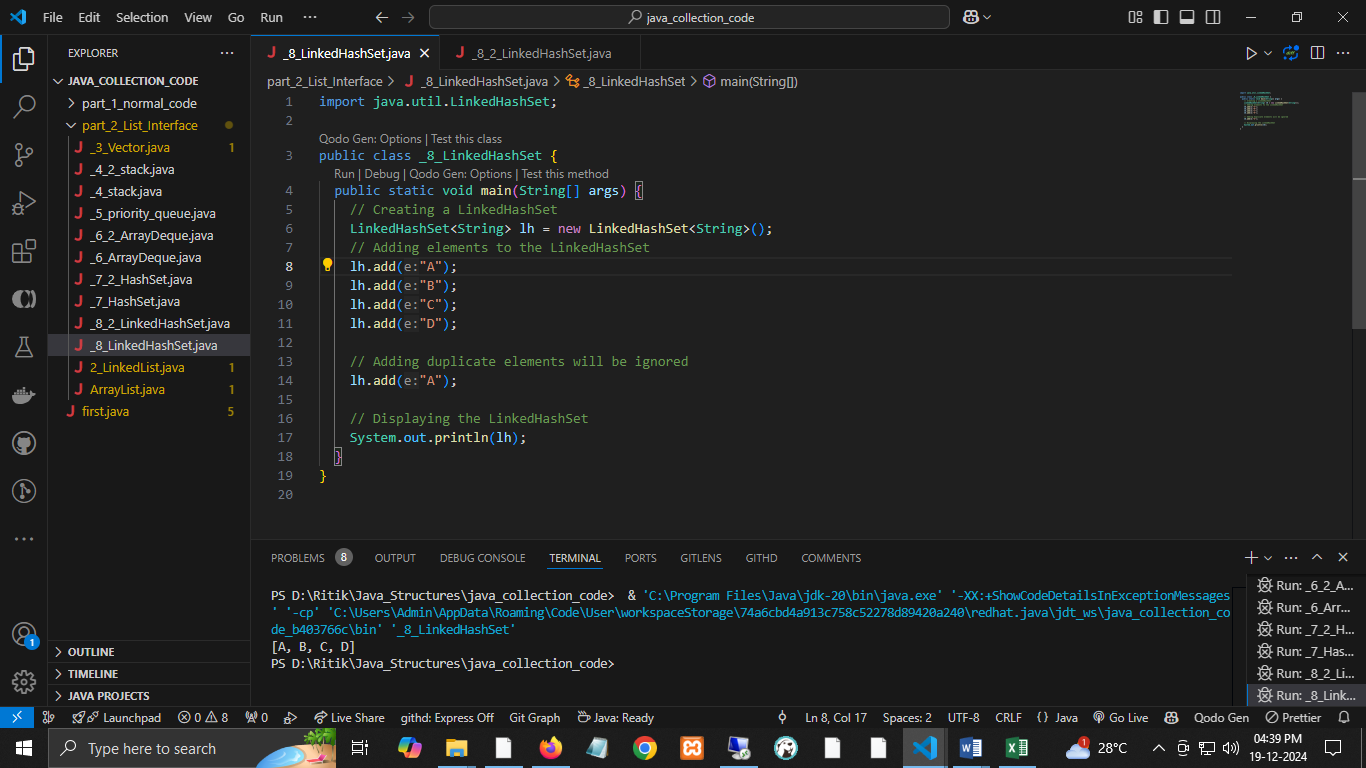


#### **ii). [LinkedHashSet](https://www.geeksforgeeks.org/linkedhashset-in-java-with-examples/)**

* A LinkedHashSet is very similar to a HashSet.
* The difference is that this uses a doubly linked list to store the data and retains the ordering of the elements.

Let’s understand the LinkedHashSet with an example:





### **7.**[**Sorted Set Interface**](https://www.geeksforgeeks.org/sortedset-java-examples/)

* This interface is very similar to the set interface.
* The only difference is that this interface has extra methods that maintain the ordering of the elements.
* The sorted set interface extends the set interface and is used to handle data that needs to be sorted.
* The class that implements this interface is TreeSet. Since this class implements the SortedSet, we can instantiate a SortedSet object with it.

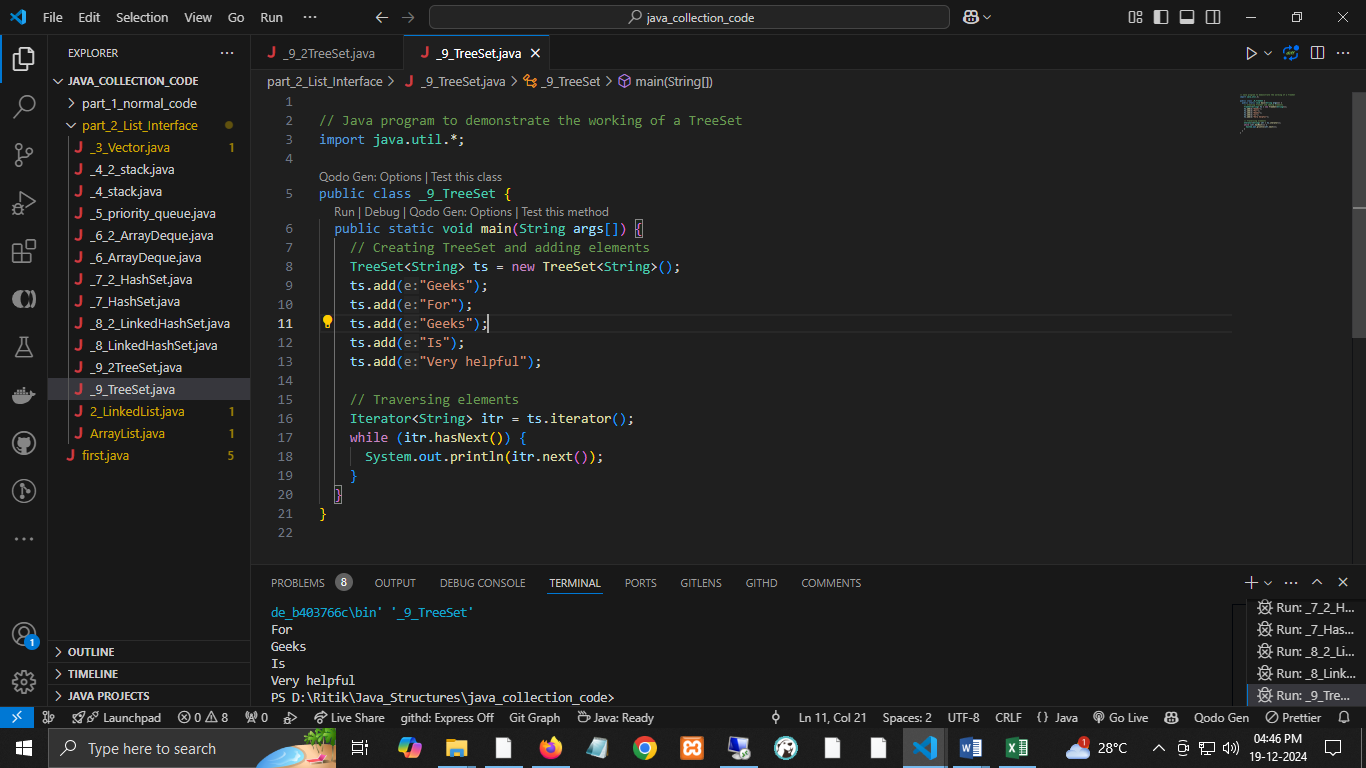
**For example:**

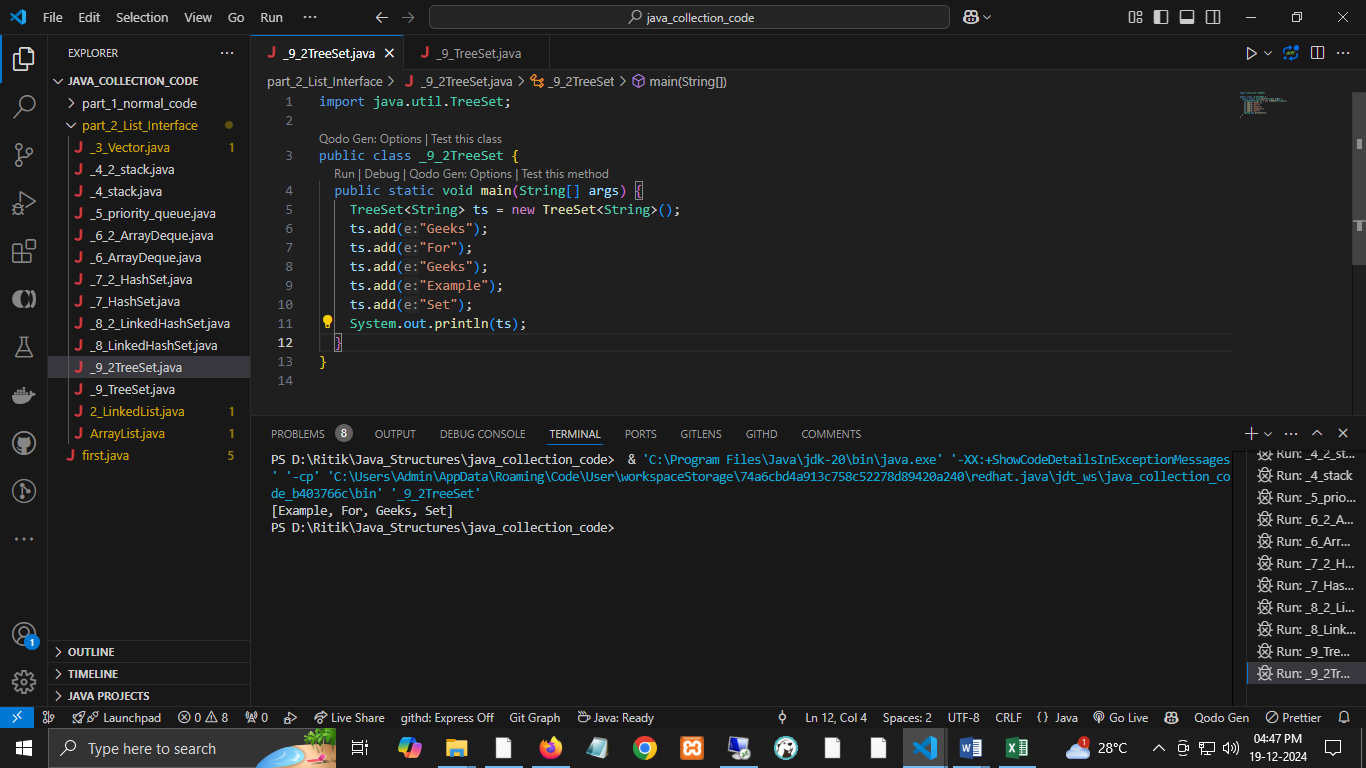
SortedSet<T> ts = new TreeSet<> ();   
Where T is the type of the object.

#### **The class which implements the sorted set interface is TreeSet.**   [**TreeSet**](https://www.geeksforgeeks.org/treeset-in-java-with-examples/)

* The TreeSet class uses a Tree for storage.
* The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided.
* This must be consistent with equals to implement the Set interface correctly.
* It can also be ordered by a Comparator provided at a set creation time, depending on the constructor used.

Let’s understand TreeSet with an example:





### **8.**[**Map Interface**](https://www.geeksforgeeks.org/map-interface-java-examples/)

* A map is a data structure that supports the key-value pair for data mapping.
* This interface doesn’t support duplicate keys because the same key cannot have multiple mappings. However, it allows duplicate values in different keys.
* A map is helpful if there is data and we wish to perform operations based on the key.
* Various classes like HashMap, TreeMap, etc implement this map interface. Since all the subclasses implement the map, we can instantiate a map object with any of these classes.

**For example:**

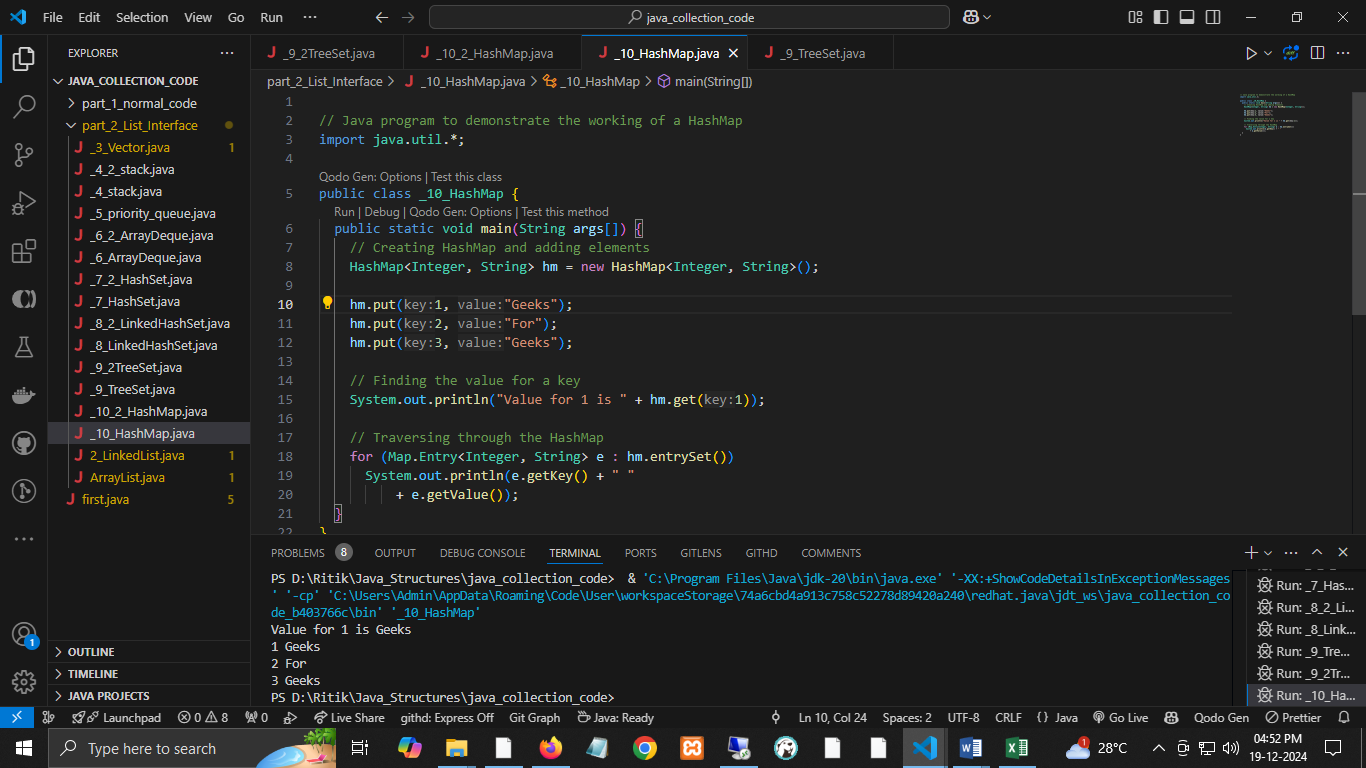
Map<T> hm = new HashMap<> ();   
Map<T> tm = new TreeMap<> ();  
   
Where T is the type of the object.

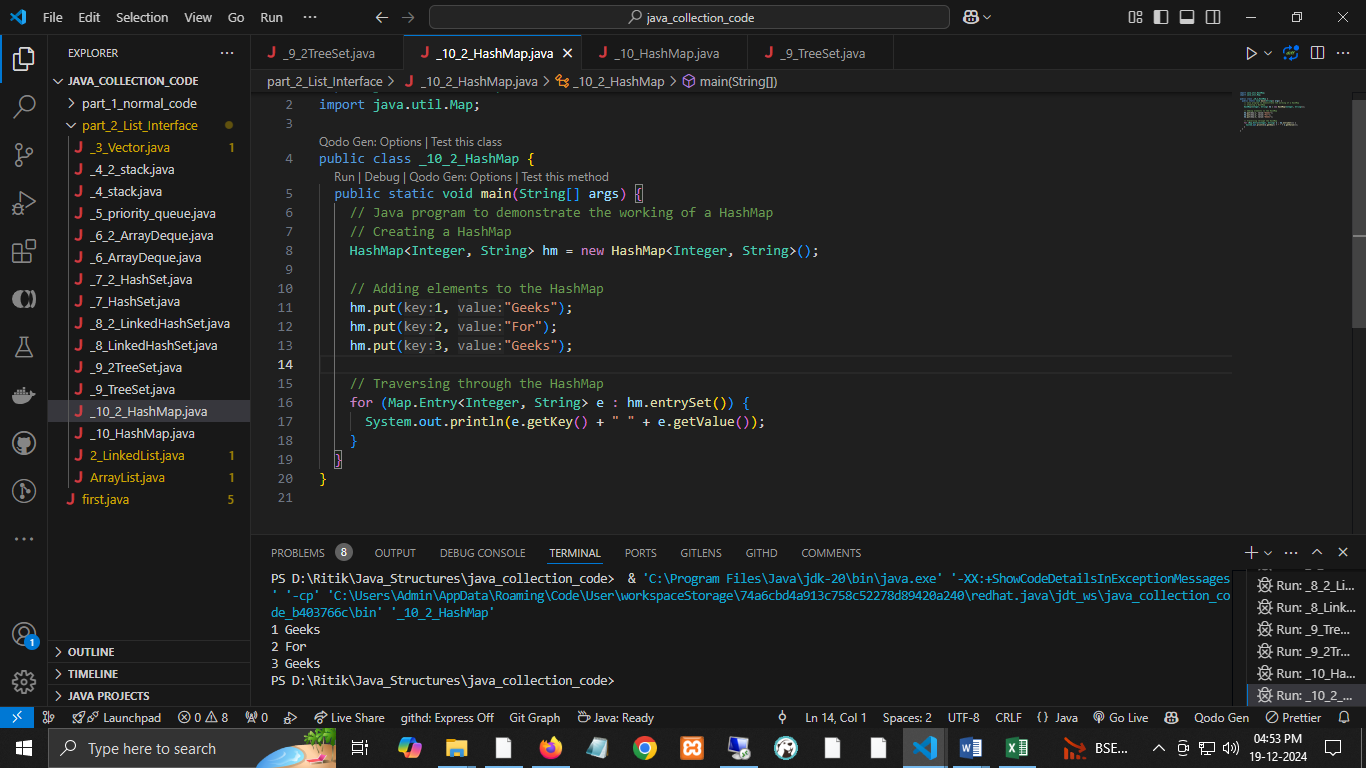
***The most frequently used implementation of a map interface is HashMap.***

#### [**HashMap**](https://www.geeksforgeeks.org/java-util-hashmap-in-java-with-examples/)

* HashMap provides the basic implementation of Java's Map interface.
* It stores data in (Key, Value) pairs. To access a value in a HashMap, we must know its key.
* HashMap uses a technique called Hashing.
* Hashing converts a large String to a small String that represents the exact String, so the indexing and search operations are faster.
* HashSet also uses HashMap internally.

Let’s understand the HashMap with an example:

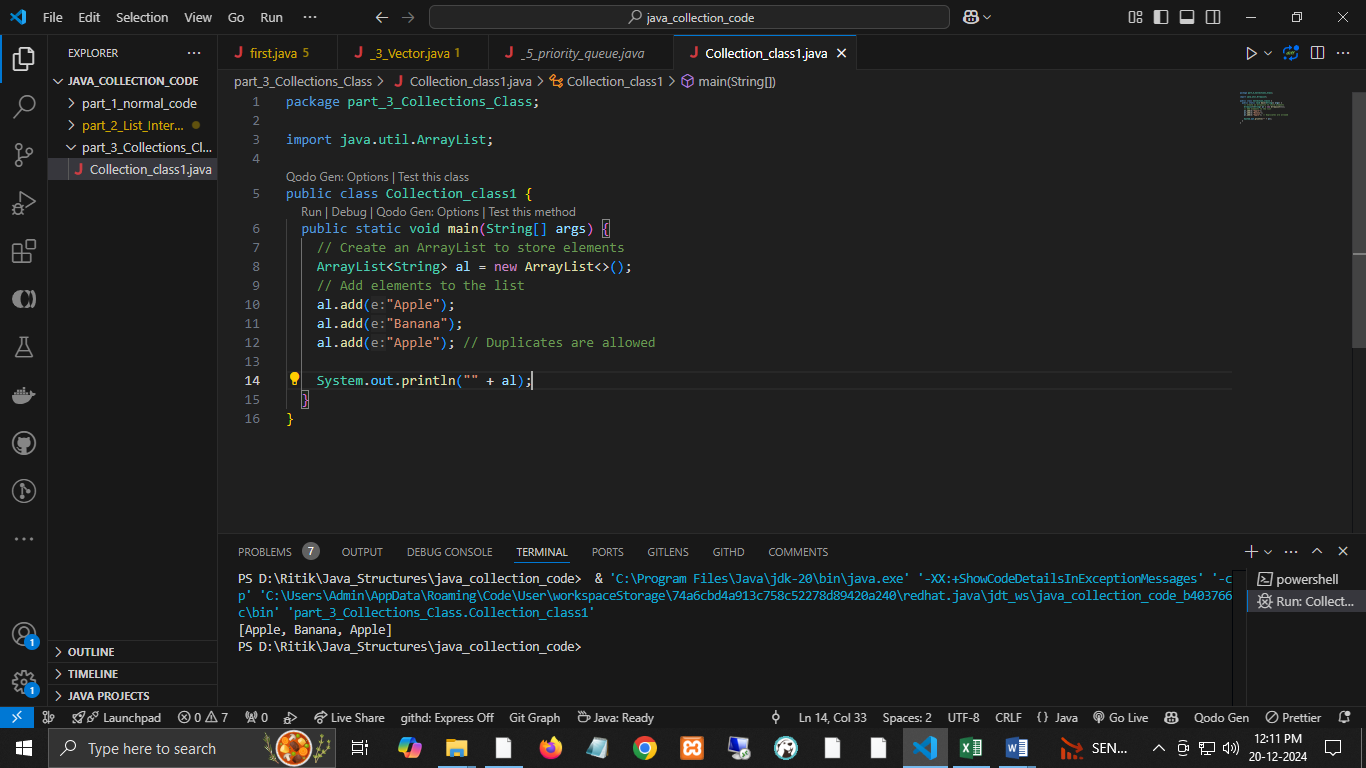




# Collections Class in Java

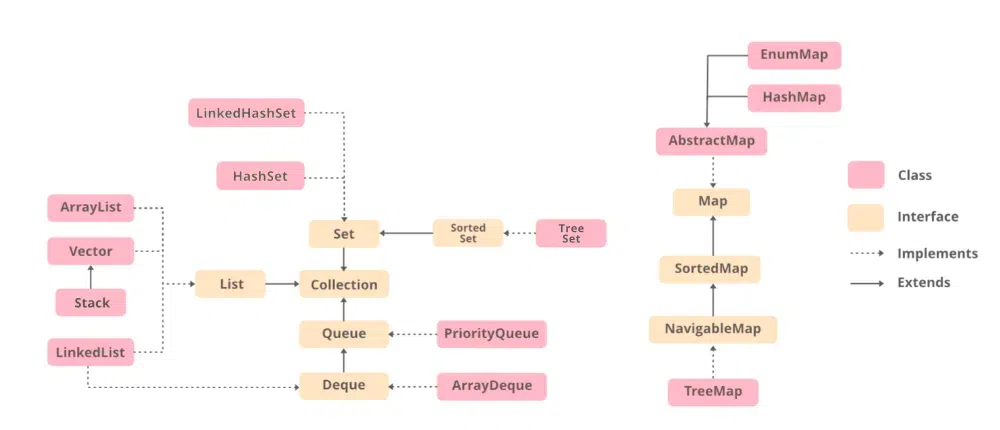
* **Collections class in Java** is one of the utility classes in Java [Collections Framework](https://www.geeksforgeeks.org/collections-in-java-2/). The **java.util** package contains the Collections class in Java.
* Java Collections class is used with static methods that operate on collections or return collections.
* All the methods of this class throw the **NullPointerException** if the collection or object passed to the methods is null.

**Example 1**: Here, we will use [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/" \t "_blank), which is a class from the Java Collections framework. It allows to store elements in a list by maintaining the insertion order and also allows duplicates.



## **Collection Class Declaration**

*public class Collections extends Object*

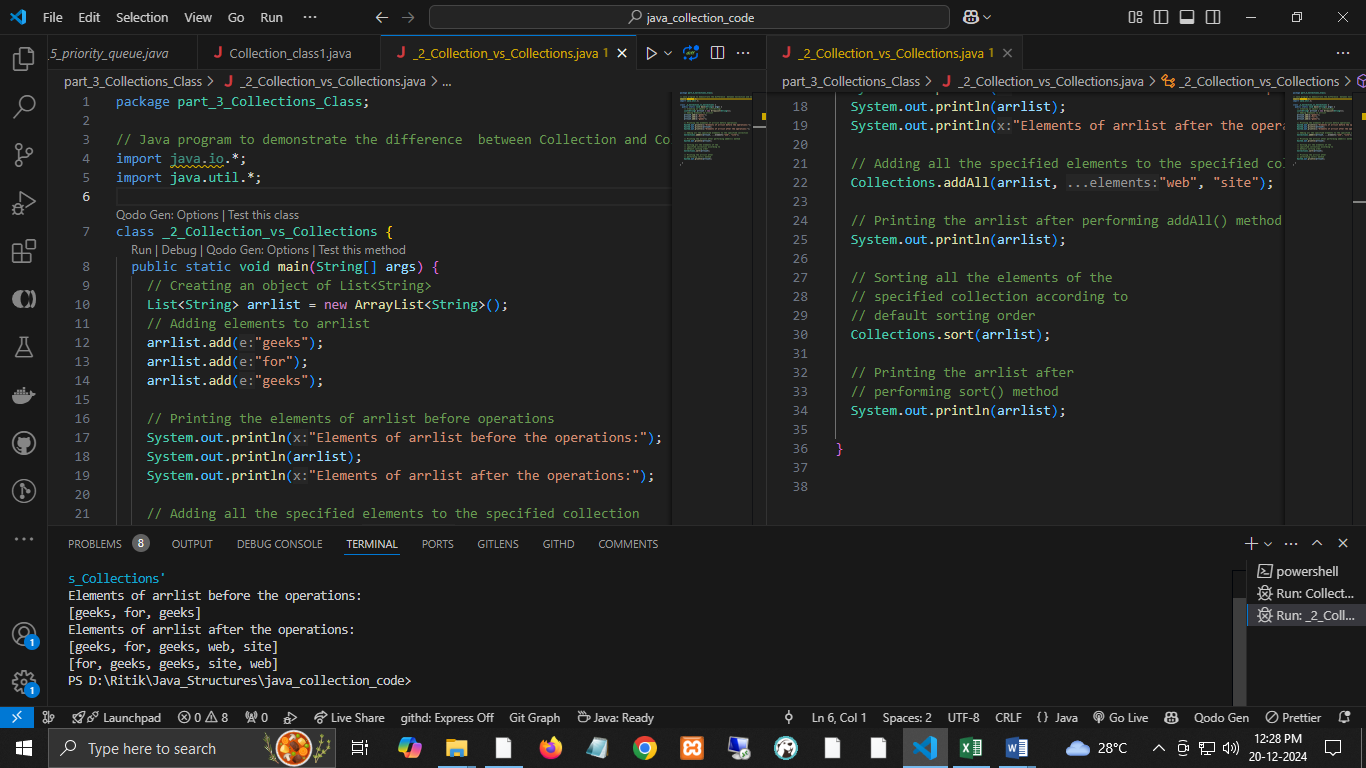


## **Java Collection Class**

**The Collection Framework contains both classes and interfaces.** Although both seem the same, there are certain differences between Collection classes and the Collections framework.

**Collection vs Collections:**

| **Collection** | **Collections** |
| --- | --- |
| It is An interface. | It is a utility class. |
| It represents a group of individual objects as a single unit. | It defines several utility methods that are used to operate on collection. |
| Since Java8, the collection has been an interface that contains a static method. The Interface can also contain abstract and default methods. | It contains only static methods. |



## **Java Collections Class Fields**

The collection class contains 3 fields as listed below which can be used to return immutable entities.

* EMPTY\_LIST to get an immutable empty List
* EMPTY\_SET to get an immutable empty Set
* EMPTY\_MAP to get an immutable empty Map

## **Java Collections Methods**

## **Java Collections Example**

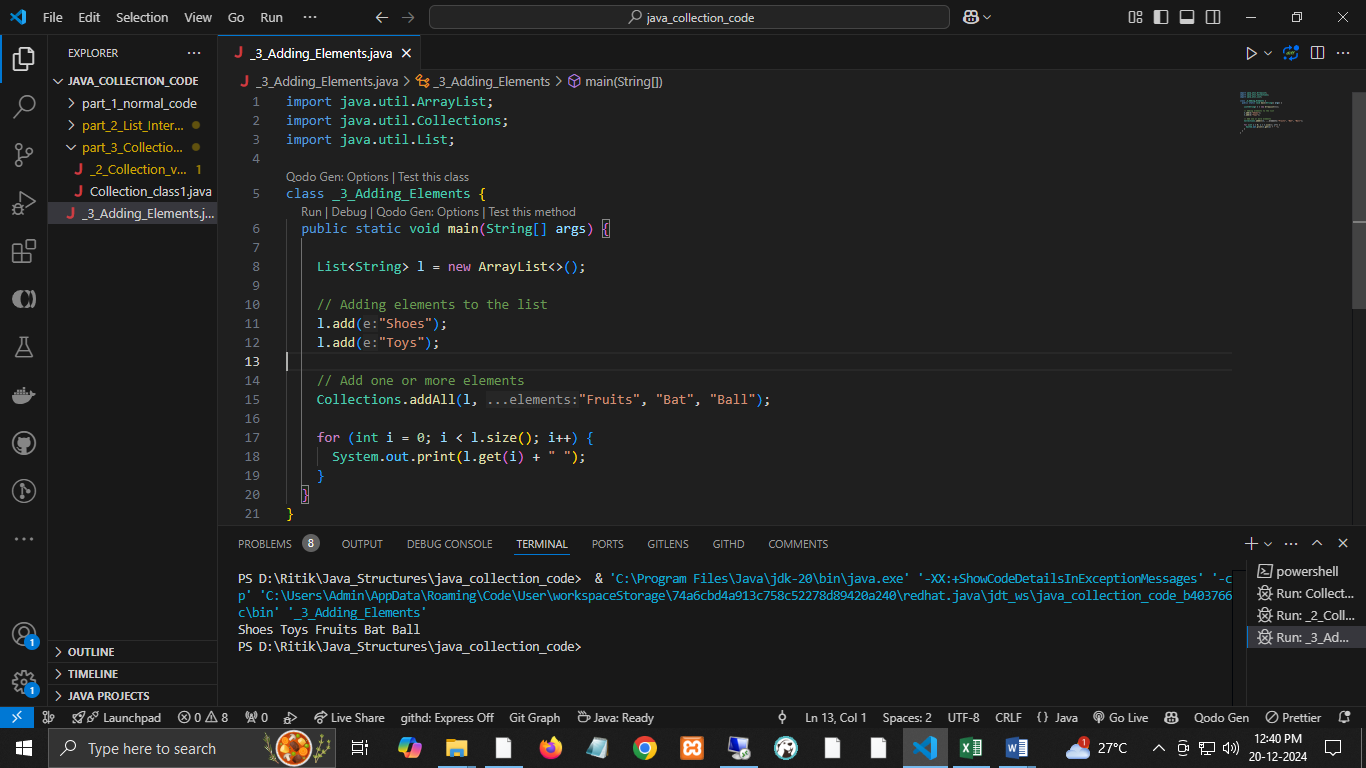
Examples of Collections Classes in Java are mentioned below:

* Adding Elements to the Collections
* Sorting a Collection
* Searching in a Collection
* Copying Elements
* Disjoint Collection

### **1.**Adding Elements to the Collections Class Object

The [addAll()](https://www.geeksforgeeks.org/collections-addall-method-in-java-with-examples/" \l ":~:text=The%20addAll()%20method%20of,individually%20or%20as%20an%20array.) method of **the Java.util.Collections class adds** all the specified elements to the specified collection. Elements to be added may be specified individually or as an array.

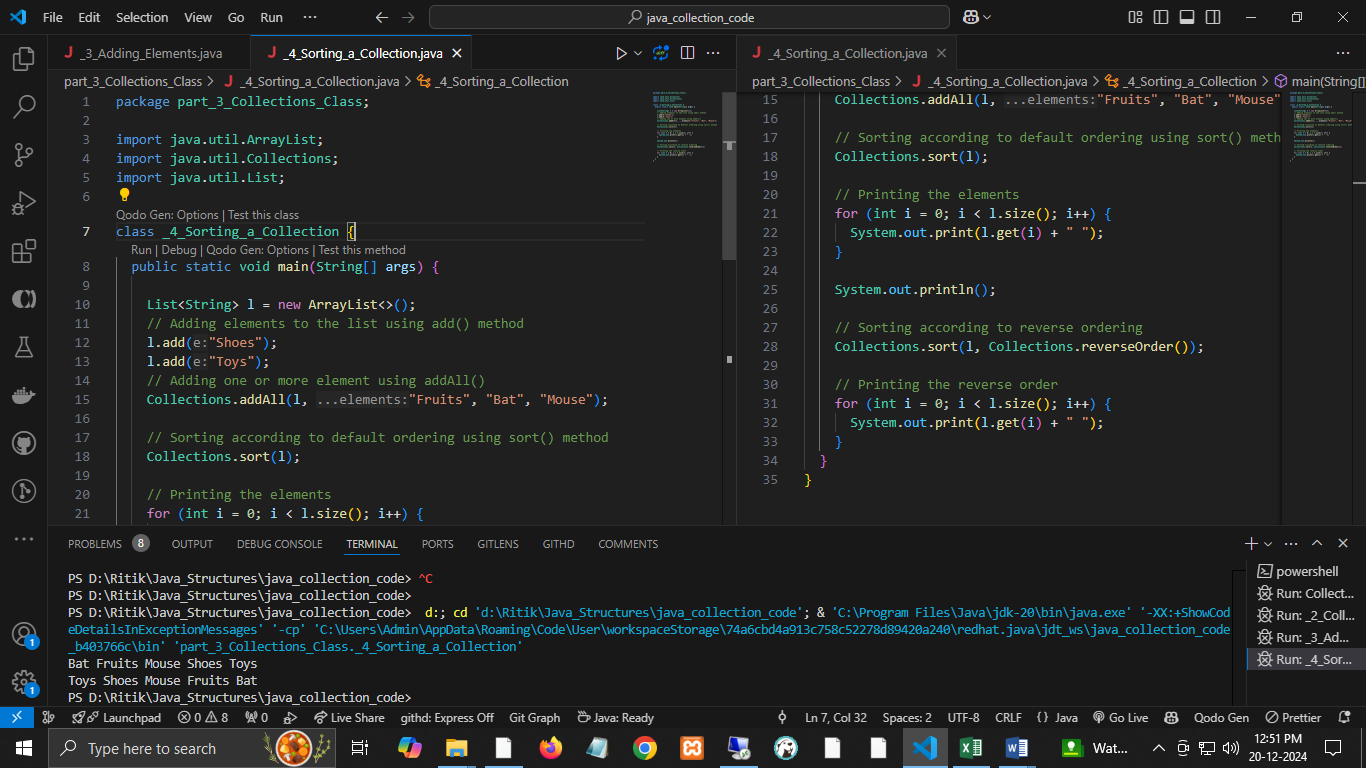
**Example:**



### **2.**Sorting a Collection

[Collections.sort()](https://www.geeksforgeeks.org/collections-sort-java-examples/) is used to sort the elements present in the specified [list](https://www.geeksforgeeks.org/list-interface-java-examples/) of Collections in ascending order. [Collections.reverseOrder()](https://www.geeksforgeeks.org/collections-reverseorder-java-examples/) is used to sort in descending order.

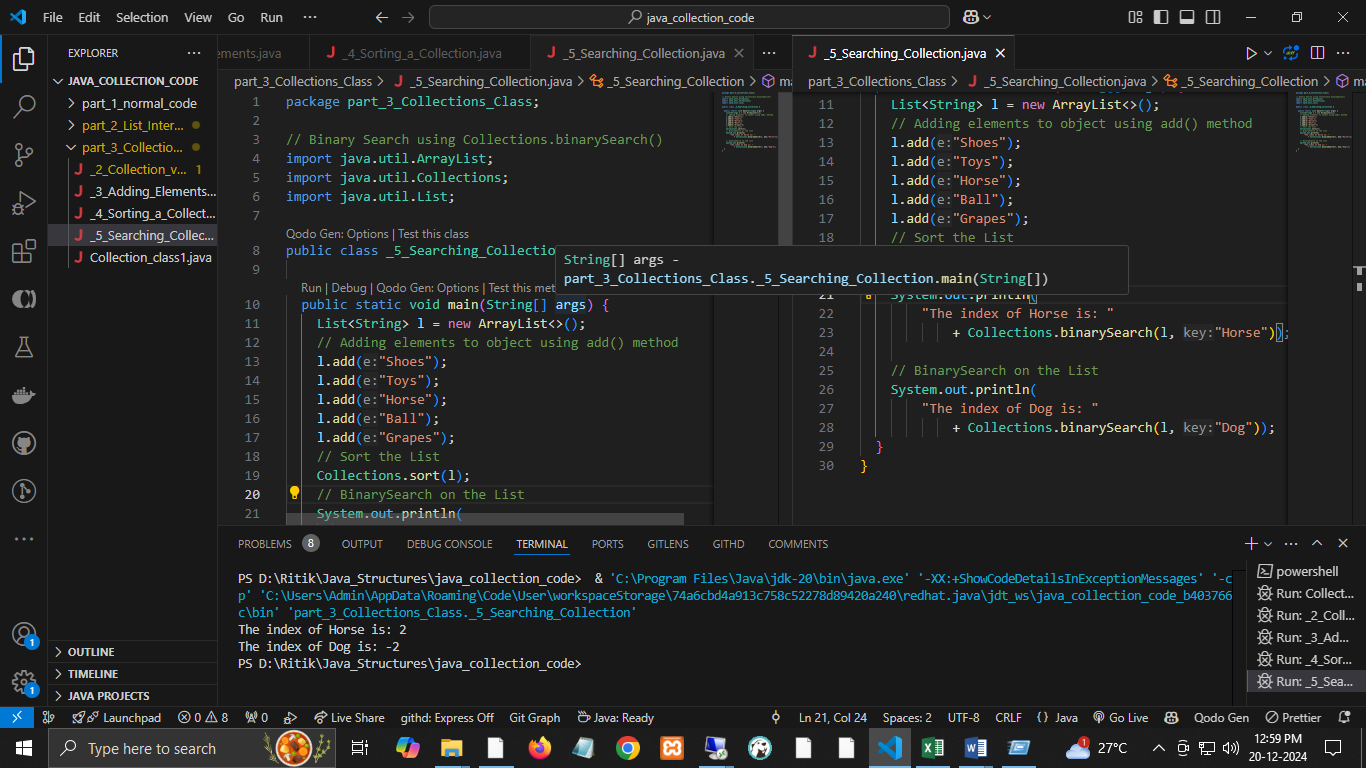
**Example:**



### **3.**Searching in a Collection

[Collections.binarySearch()](https://www.geeksforgeeks.org/collections-binarysearch-java-examples/) method returns the position of an object in a sorted list. To use this method, the list should be sorted in ascending order. Otherwise, the result returned from the process will be wrong. If the element exists in the list, the method will return the element's position in the sorted list. Otherwise, the result returned by the process would be the – (insertion point where the element should have been present if it exists)-1).

**Example:**

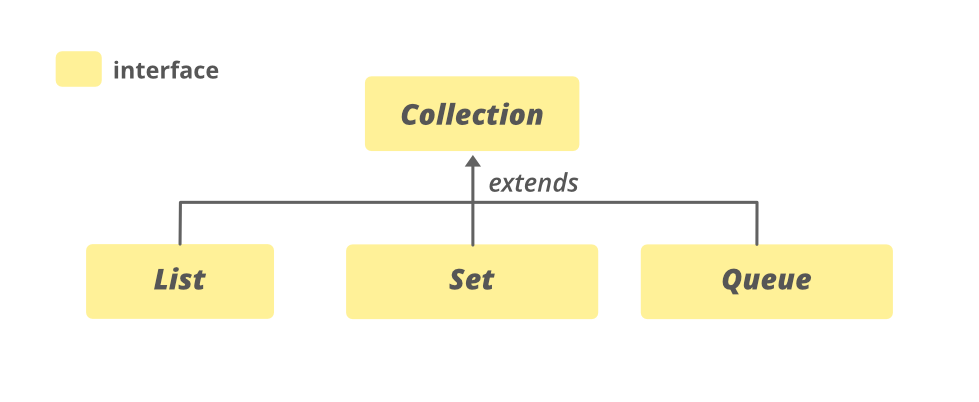


# Collection Interface in Java with Examples

* The **Collection**interface is a member of the [Java Collections Framework](https://www.geeksforgeeks.org/collections-in-java-2/).
* It is a part of **java.util** package.
* It is one of the root interfaces of the Collection Hierarchy.
* The Collection interface is not directly implemented by any class. However, it is implemented indirectly via its subtypes or subinterfaces like [List](https://www.geeksforgeeks.org/list-interface-java-examples/), [Queue](https://www.geeksforgeeks.org/queue-interface-java/), and [Set](https://www.geeksforgeeks.org/set-in-java/).

**For Example,** the [HashSet](https://www.geeksforgeeks.org/hashset-in-java/#:~:text=Last%20Updated%3A%2009%2D09%2D,class%20permits%20the%20null%20element.) class implements the Set interface which is a subinterface of the Collection interface. If a collection implementation doesn’t implement a particular operation, it should define the corresponding method to throw **UnsupportedOperationException**.

* **The Hierarchy of Collection**



**SubInterfaces of Collection Interface**

* All the Classes of the Collection Framework implement the subInterfaces of the Collection Interface.
* All the methods of Collection interfaces are also contained in its subinterfaces. These subinterfaces are sometimes called as **Collection Types** or **SubTypes of Collection.**
* These include the following:
  + List
  + Set
  + SortedSet
  + Queue
  + Deque

**List:**

* This list interface is implemented by various classes, such as [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/), [Vector](https://www.geeksforgeeks.org/java-util-vector-class-java/), [Stack](https://www.geeksforgeeks.org/stack-class-in-java/), etc.
* Since all the subclasses implement the list, we can instantiate a list object with any of these classes.

For example,

*List <T> al = new ArrayList<> ();*

*List <T> ll = new LinkedList<> ();*

*List <T> v = new Vector<> ();*

*Where T is the type of the object*

**Set:**

* This set interface is implemented by various classes like [HashSet](https://www.geeksforgeeks.org/hashset-in-java/), [TreeSet](https://www.geeksforgeeks.org/treeset-in-java-with-examples/), [LinkedHashSet](https://www.geeksforgeeks.org/linkedhashset-in-java-with-examples/), etc.
* Since all the subclasses implement the set, we can instantiate a set object with any of these classes.

For example,

*Set<T> hs = new HashSet<> ();*

*Set<T> lhs = new LinkedHashSet<> ();*

*Set<T> ts = new TreeSet<> ();*

*Where T is the type of the object.*

**SortedSet**:

* The sorted set interface extends the set interface and handles the data that needs to be sorted.
* The class that implements this interface is [TreeSet](https://www.geeksforgeeks.org/treeset-in-java-with-examples/). Since this class implements the SortedSet, we can instantiate a SortedSet object with this class.

For example,

*SortedSet<T> ts = new TreeSet<> ();*

*Where T is the type of the object.*

**Queue:**

* Queue has various classes like [PriorityQueue](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/), [Deque](https://www.geeksforgeeks.org/deque-set-1-introduction-applications/), [ArrayDeque](https://www.geeksforgeeks.org/arraydeque-in-java/), etc. Since all these subclasses implement the queue, we can instantiate a queue object with any of these classes.

For example,

*Queue <T> pq = new PriorityQueue<> ();*

*Queue <T> ad = new ArrayDeque<> ();*

*Where T is the type of the object.*

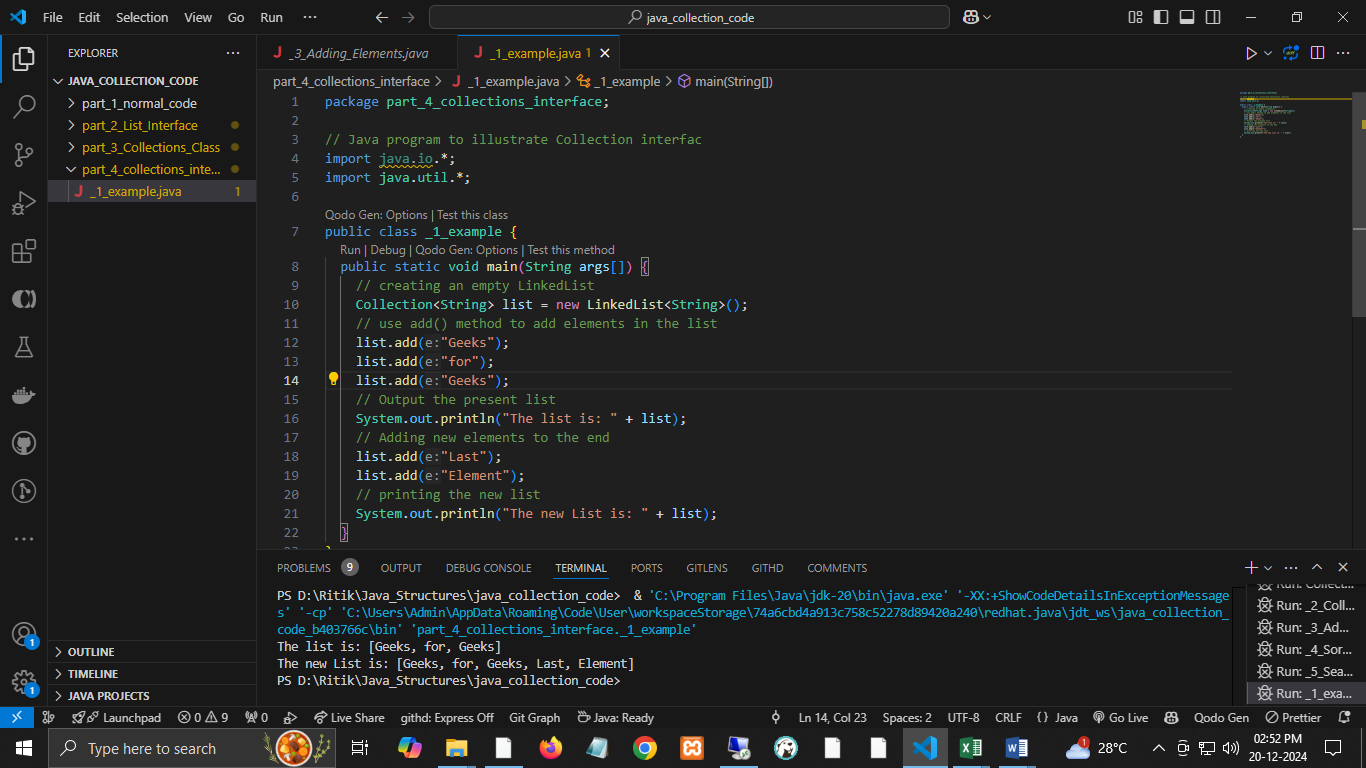
**Deque**:

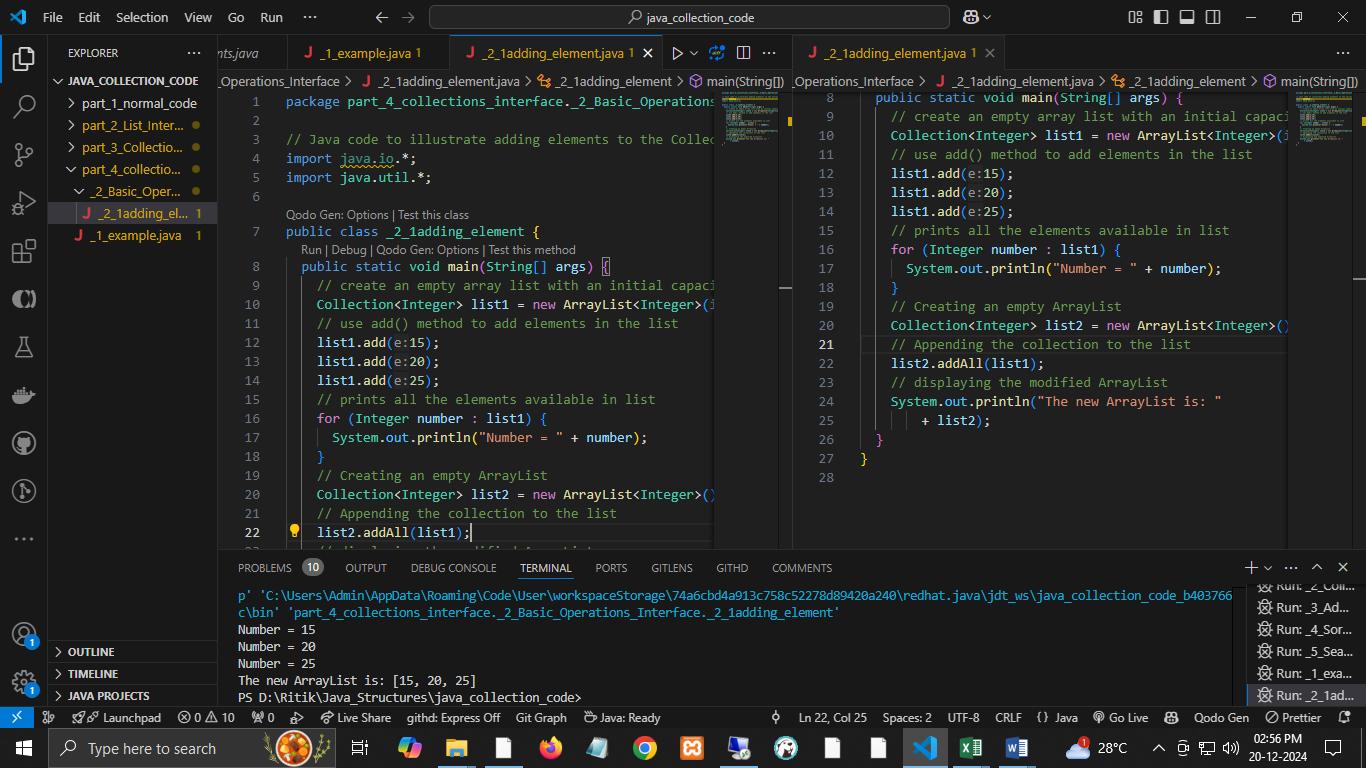
* This is a very slight variation of the [queue data structure](https://www.geeksforgeeks.org/queue-data-structure/).
* A deque, also known as a double-ended queue, is a data structure in which we can add and remove elements from both ends of the queue.
* This interface extends the queue interface.
* The class which implements this interface is [ArrayDeque](https://www.geeksforgeeks.org/arraydeque-in-java/). Since this class implements the deque, we can instantiate a deque object with this class.

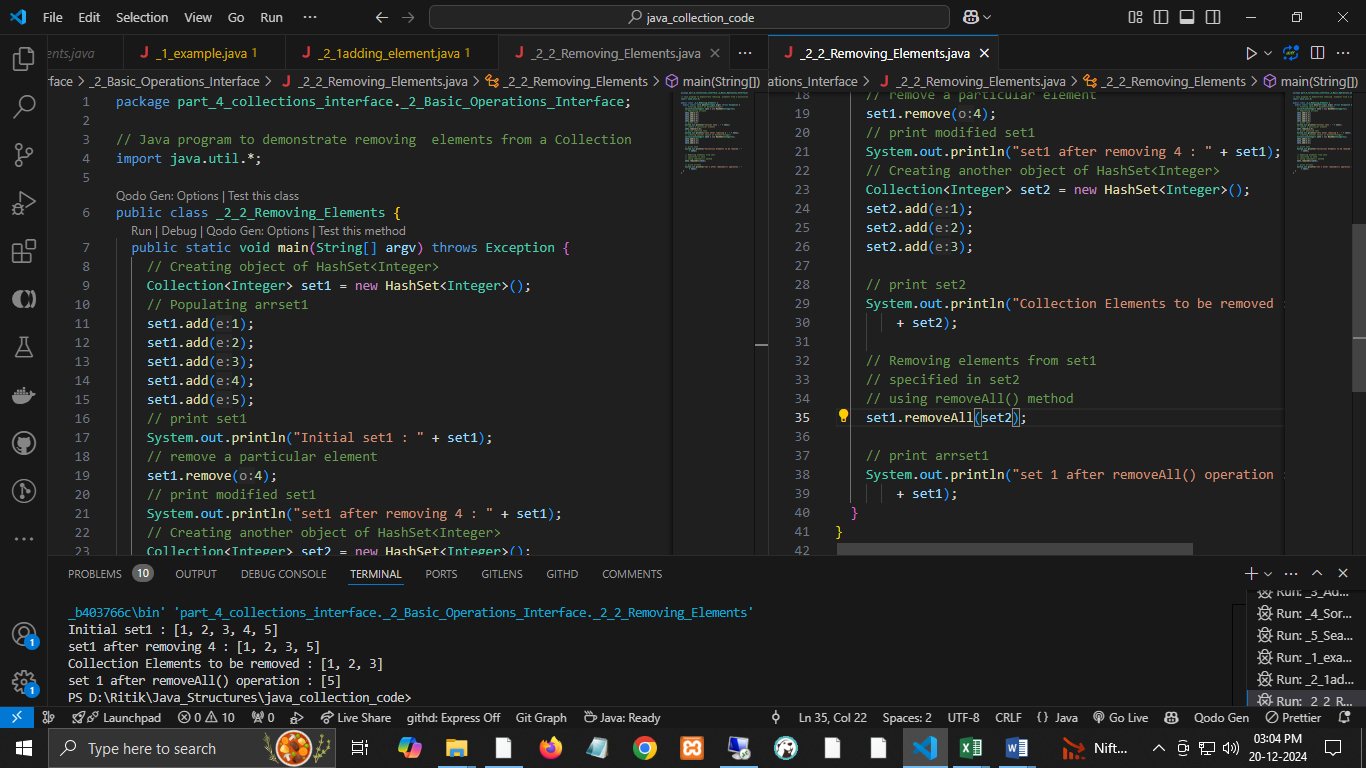
For example,

*Deque<T> ad = new ArrayDeque<> ();*

*Where T is the type of the object.*







### Methods of Collection

# Java List Interface

* The List Interface in Java extends the [**Collection Interface**](https://www.geeksforgeeks.org/collection-interface-in-java-with-examples/) and is a part of **[java.util package](https://www.geeksforgeeks.org/java-util-package-java/" \t "_blank)**.
* A Java list is used to store ordered collections of elements. Thus, it allows you to organize and manage data sequentially.
  + Maintained the order of elements in which they are added.
  + Allows the duplicate elements.
  + The implementation classes of the List interface are **[ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/" \t "_blank)**, [**LinkedList**](https://www.geeksforgeeks.org/linked-list-in-java/), [**Stack**](https://www.geeksforgeeks.org/stack-class-in-java/), and [**Vector**](https://www.geeksforgeeks.org/java-util-vector-class-java/).
  + Can add Null values that depend on the implementation.
  + The List interface offers methods to access elements by their index and includes the**listIterator() method**, which returns a **ListIterator**.
  + Using ListIterator, we can traverse the list in both forward and backward directions.

**Example:**

**Declaration of Java List Interface**

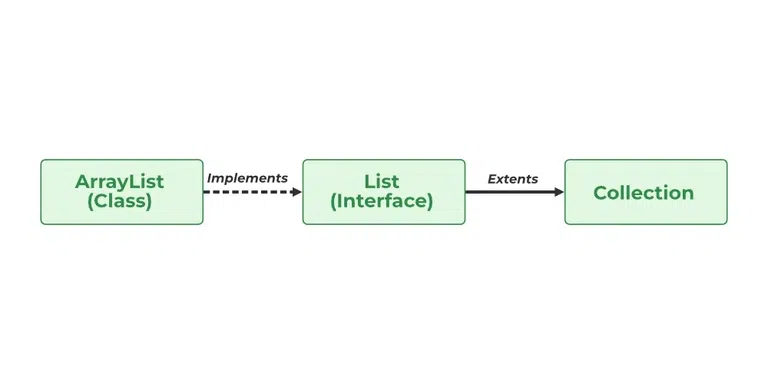
public interface List<E> extends Collection<E> ;

**Let us elaborate on creating objects or instances in a List.**Since **List** is an [interface](https://www.geeksforgeeks.org/interfaces-in-java/), objects cannot be made of the type list. We always need a class that implements this **List** to create an object. Also, after the introduction of [Generics](https://www.geeksforgeeks.org/generics-in-java/) in Java 1.5, it is possible to restrict the type of object that can be stored in the List. Just like several other user-defined ‘interfaces’ implemented by user-defined ‘classes’, **List** is an ‘interface’, implemented by the **ArrayList** class, pre-defined in **java.util** package.

**Syntax:**

List<Obj> list = new ArrayList<Obj> ();

Obj is the type of the object to be stored in List.



* **ArrayList** and **LinkedList** are the most widely used due to their dynamic resizing and efficient performance for specific operations.
* **Vector** is considered a legacy class and is rarely used in modern Java programming. ArrayList and java.util replace it. concurrent package.
* **Stack** is a subclass of Vector, designed for Last-In-First-Out (LIFO) operations.

Now let us perform various operations using List Interface to have a better understanding of the same. We will be discussing the following operations listed below and later on implementing them via clean Java codes.

**Java List – Operations**

Since List is an interface, it can be used only with a class that implements this interface. Now, let’s see how to perform a few frequently used operations on the List.

* **Operation 1:**Adding elements to List using add() method
* **Operation 2:**Updating elements in List using set() method
* **Operation 3:** Searching for elements using indexOf(), lastIndexOf methods
* **Operation 4:**Removing elements using remove() method
* **Operation 5:**Accessing Elements in List using get() method
* **Operation 6:**Checking if an element is present in the List using contains() method

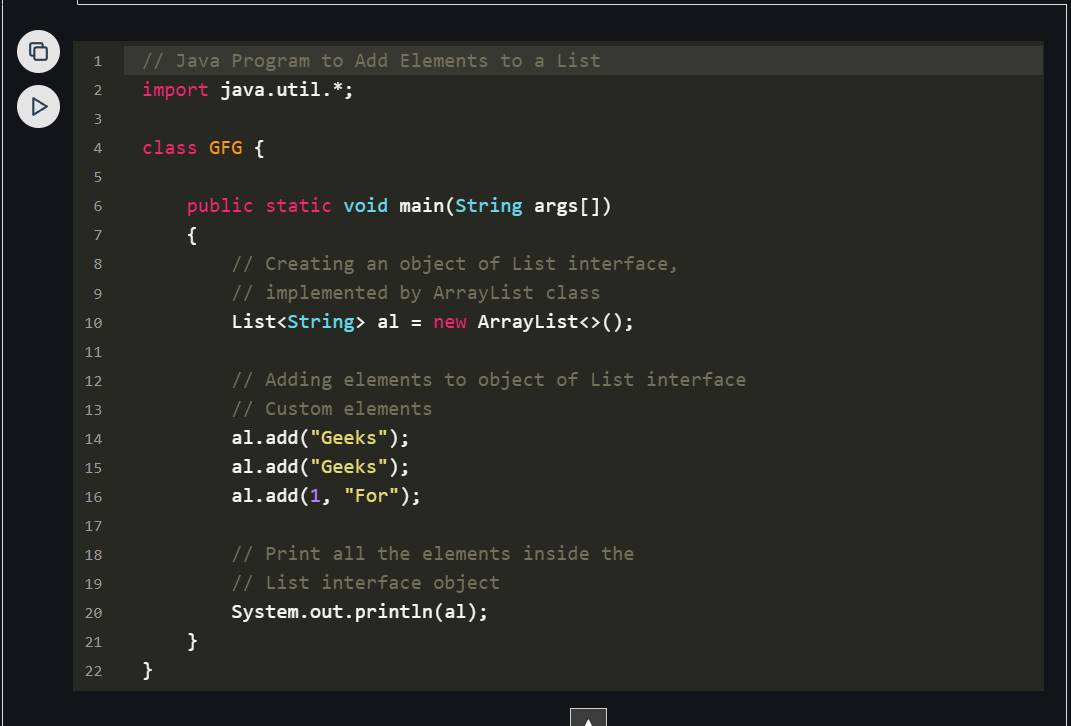
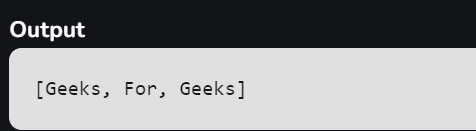
**1. Adding Elements**

In order to add an element to the list, we can use the[**add()**](https://www.geeksforgeeks.org/list-add-method-in-java-with-examples/)method. This method is overloaded to perform multiple operations based on different parameters.

**Parameters:**  It takes 2 parameters, namely:

* **add(Object o):** This method is used to add an element at the end of the List.
* **add(int index, Object o):** This method is used to add an element at a specific index in the List

**Note:**If we do not specify the length of the array in the ArrayList constructor while creating the List object, using add(int index, Object) for any index i will throw an Exception if we have not specified the values for 0 to i-1 index already.

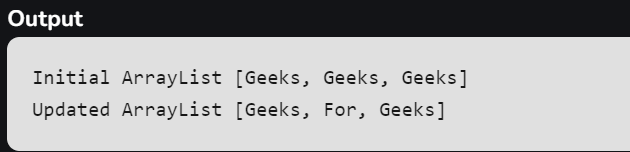
 

**2. Updating Elements**

After adding the elements, if we wish to change the element, it can be done using the[**set()**](https://www.geeksforgeeks.org/abstractlist-set-method-in-java-with-examples/)method. Since List is indexed, the element which we wish to change is referenced by the index of the element. Therefore, this method takes an index and the updated element which needs to be inserted at that index.

**Example:**





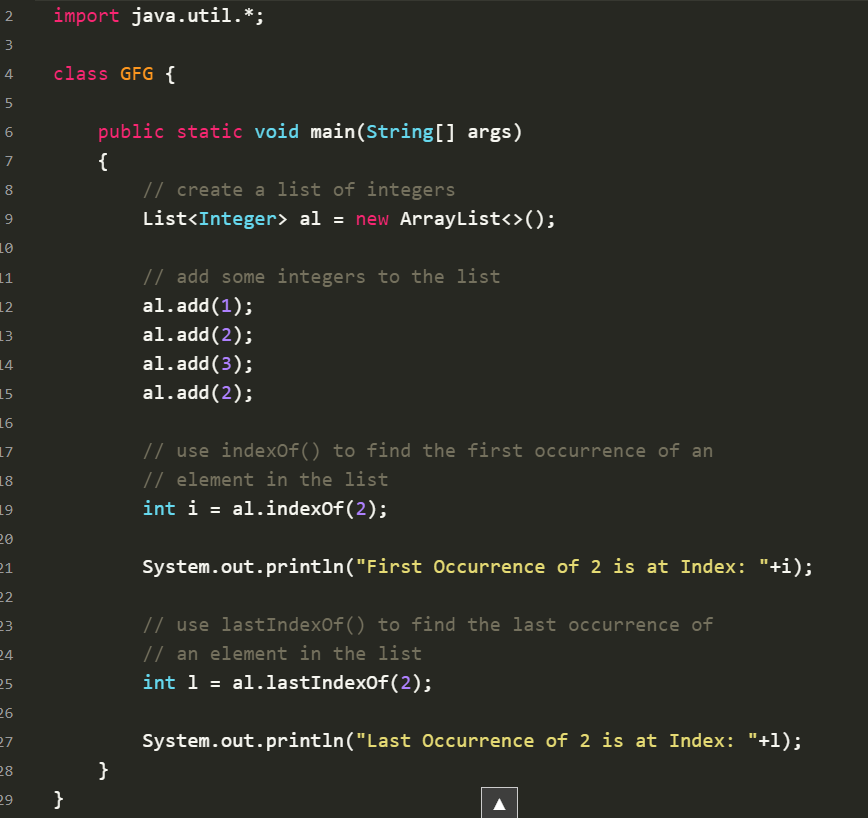
**3. Searching Elements**

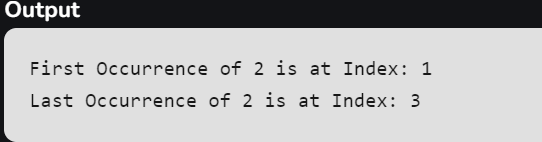
* Searching for elements in the List interface is a common operation in Java programming.
* The List interface provides several methods to search for elements, such as the[**indexOf()**](https://www.geeksforgeeks.org/list-indexof-method-in-java-with-examples/),**[lastIndexOf()](https://www.geeksforgeeks.org/list-lastindexof-method-in-java-with-examples/)** methods.
* The indexOf() method returns the index of the first occurrence of a specified element in the list, while the lastIndexOf() method returns the index of the last occurrence of a specified element.

**Parameters:**

* **indexOf(Object o):**Returns the index of the first occurrence of the specified element in the list, or -1 if the element is not found
* **lastIndexOf(Object o):**Returns the index of the last occurrence of the specified element in the list, or -1 if the element is not found

**Example:**





**4. Removing Elements**

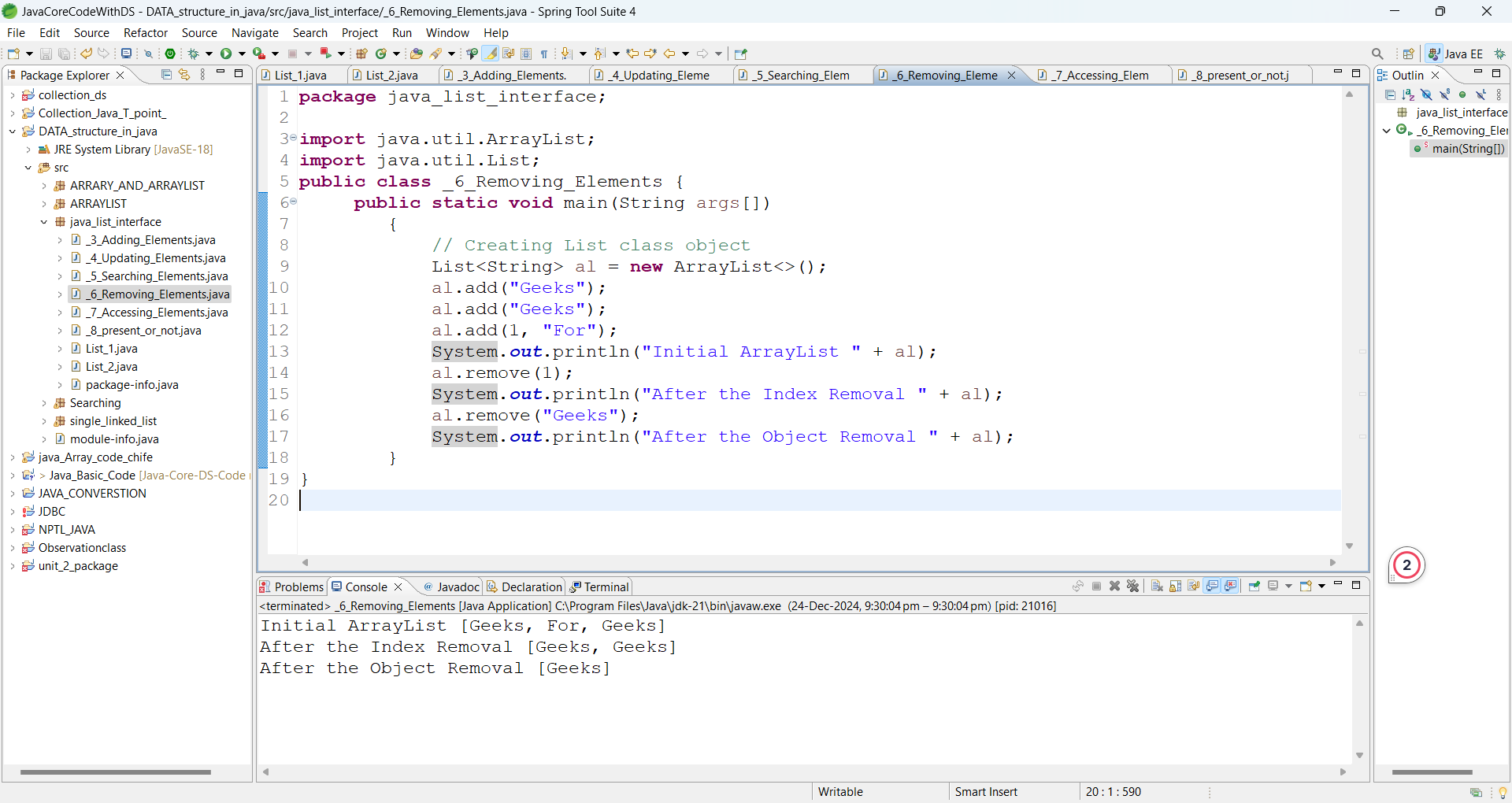
In order to remove an element from a list, we can use the[**remove()**](https://www.geeksforgeeks.org/list-removeobject-obj-method-in-java-with-examples/)method. This method is overloaded to perform multiple operations based on different parameters.

They are:

**Parameters:**

* **remove(Object o):** This method is used to simply remove an object from the List. If there are multiple such objects, then the first occurrence of the object is removed.
* **remove(int index):** Since a List is indexed, this method takes an integer value which simply removes the element present at that specific index in the List. After removing the element, all the elements are moved to the left to fill the space and the indices of the objects are updated.

**Example:**



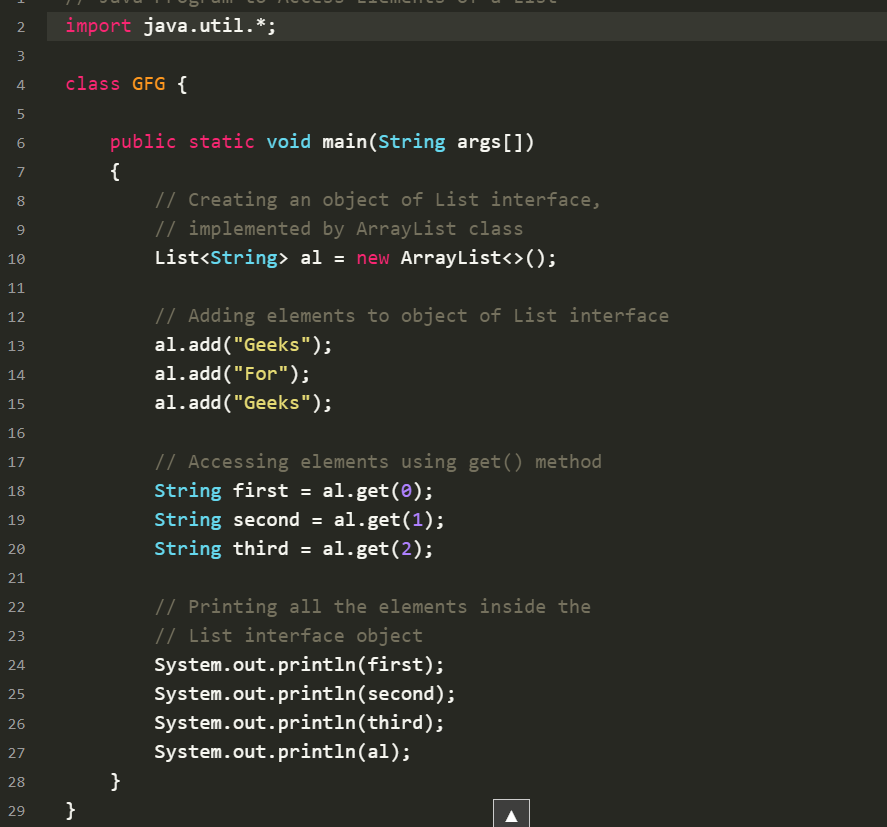
**5. Accessing Elements**

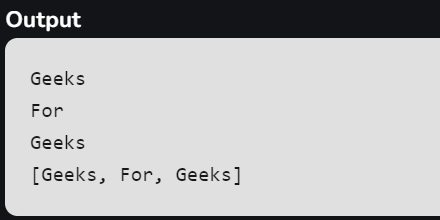
In order to access an element in the list, we can use the [**get()**](https://www.geeksforgeeks.org/list-get-method-in-java-with-examples/) method, which returns the element at the specified index

**Parameters:**

* get(int index): This method returns the element at the specified index in the list.

**Example:**





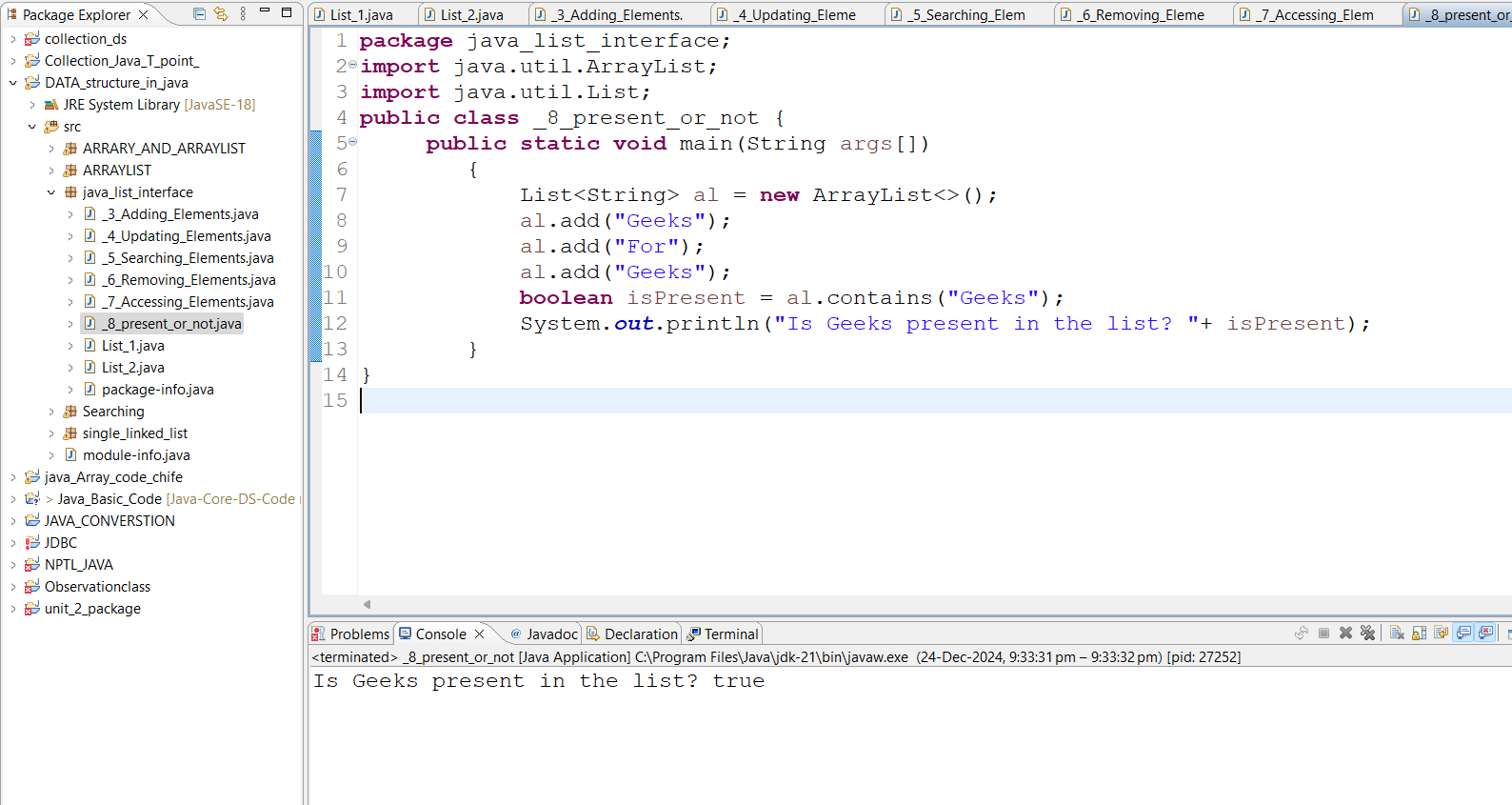
**6. Checking if an element is present or not**

In order to check if an element is present in the list, we can use the [**contains()**](https://www.geeksforgeeks.org/list-contains-method-in-java-with-examples/)method. This method returns true if the specified element is present in the list, otherwise, it returns false.

**Parameters:**

* contains(Object o): This method takes a single parameter, the object to be checked if it is present in the list.

**Example:**



**Complexity of List Interface in Java**

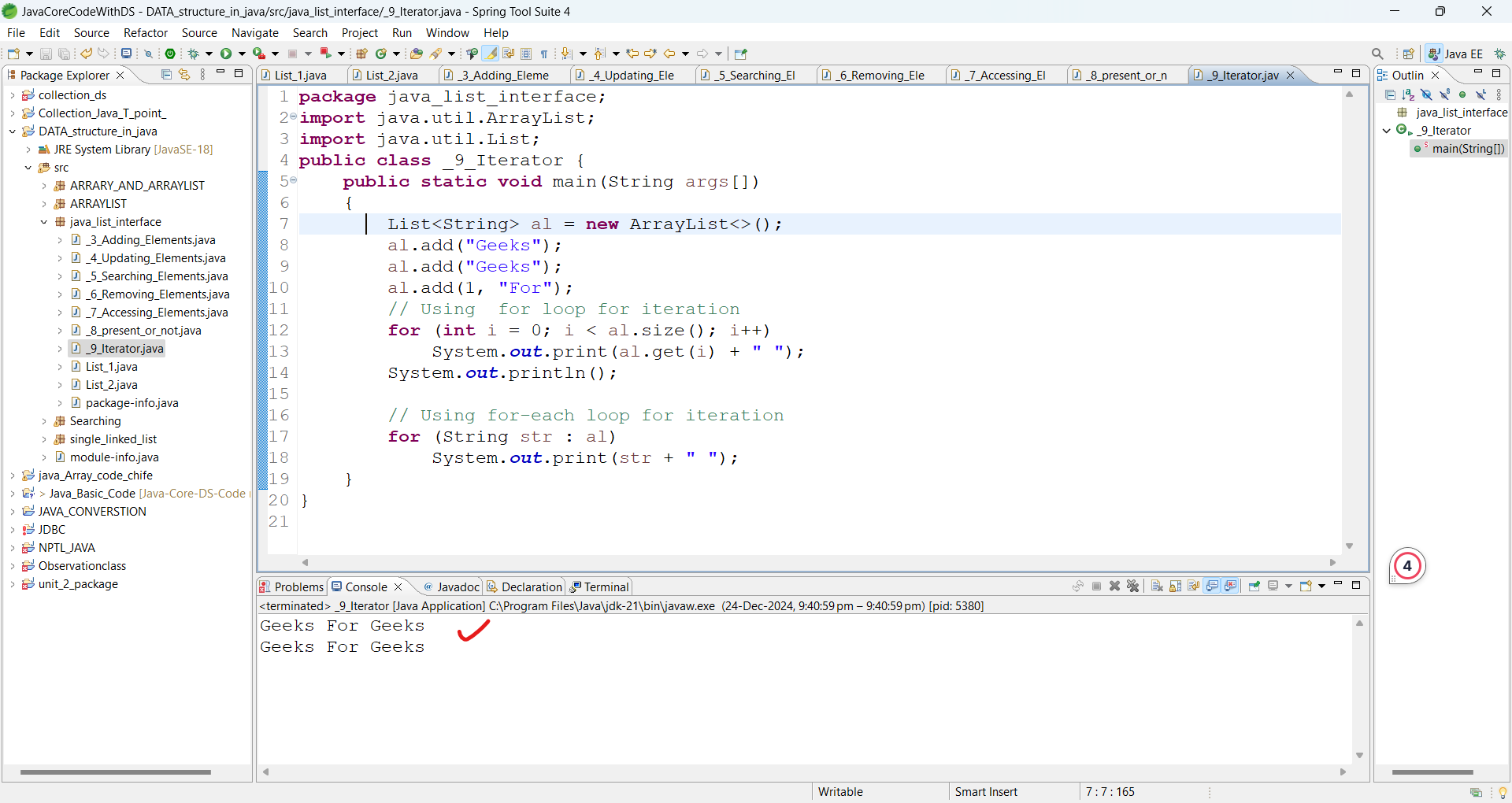
| **Operation** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- |
| **Adding Element in List Interface** | O(1) | O(1) |
| **Remove Element from List Interface** | O(N) | O(N) |
| **Replace Element in List Interface** | O(N) | O(N) |
| **Traversing List Interface** | O(N) | O(N) |

**Iterating over List Interface in Java**

Till now we are having a very small input size and we are doing operations manually for every entity. Now let us discuss various ways by which we can iterate over the list to get them working for a larger sample set.

**Methods:**There are multiple ways to iterate through the List. The most famous ways are by using the basic [for loop](https://www.geeksforgeeks.org/loop-java-important-points/) in combination with a [get() method](https://www.geeksforgeeks.org/list-get-method-in-java-with-examples/) to get the element at a specific index and the [advanced for a loop](https://www.geeksforgeeks.org/for-each-loop-in-java/).

**Example:**



**Methods of the List Interface**

Since the main concept behind the different types of lists is the same, the list interface contains the following methods:

| **Method** | **Description** |
| --- | --- |
| [**add(int index, element)**](https://www.geeksforgeeks.org/list-addint-index-e-element-method-in-java/) | This method is used with Java List Interface to add an element at a particular index in the list. When a single parameter is passed, it simply adds the element at the end of the list. |
| [**addAll(int index, Collection collection)**](https://www.geeksforgeeks.org/list-addall-method-in-java-with-examples/) | This method is used with List Interface in Java to add all the elements in the given collection to the list. When a single parameter is passed, it adds all the elements of the given collection at the end of the list. |
| [**size()**](https://www.geeksforgeeks.org/list-size-method-in-java-with-examples/) | This method is used with Java List Interface to return the size of the list. |
| [**clear()**](https://www.geeksforgeeks.org/list-clear-method-in-java-with-examples/) | This method is used to remove all the elements in the list. However, the reference of the list created is still stored. |
| [**remove(int index)**](https://www.geeksforgeeks.org/list-removeint-index-method-in-java-with-examples/) | This method removes an element from the specified index. It shifts subsequent elements(if any) to left and decreases their indexes by 1. |
| [**remove(element)**](https://www.geeksforgeeks.org/list-removeobject-obj-method-in-java-with-examples/) | This method is used with Java List Interface to remove the first occurrence of the given element in the list. |
| [**get(int index)**](https://www.geeksforgeeks.org/list-get-method-in-java-with-examples/) | This method returns elements at the specified index. |
| [**set(int index, element)**](https://www.geeksforgeeks.org/arraylist-set-method-in-java-with-examples/) | This method replaces elements at a given index with the new element. This function returns the element which was just replaced by a new element. |
| [**indexOf(element)**](https://www.geeksforgeeks.org/list-indexof-method-in-java-with-examples/) | This method returns the first occurrence of the given element or *-1* if the element is not present in the list. |
| [**lastIndexOf(element)**](https://www.geeksforgeeks.org/list-lastindexof-method-in-java-with-examples/) | This method returns the last occurrence of the given element or *-1* if the element is not present in the list. |
| [**equals(element)**](https://www.geeksforgeeks.org/list-equals-method-in-java-with-examples/) | This method is used with Java List Interface to compare the equality of the given element with the elements of the list. |
| [**hashCode()**](https://www.geeksforgeeks.org/list-hashcode-method-in-java-with-examples/) | This method is used with List Interface in Java to return the hashcode value of the given list. |
| [**isEmpty()**](https://www.geeksforgeeks.org/list-isempty-method-in-java-with-examples/) | This method is used with Java List Interface to check if the list is empty or not. It returns true if the list is empty, else false. |
| [**contains(element)**](https://www.geeksforgeeks.org/list-contains-method-in-java-with-examples/) | This method is used with List Interface in Java to check if the list contains the given element or not. It returns true if the list contains the element. |
| [**containsAll(Collection collection)**](https://www.geeksforgeeks.org/list-containsall-method-in-java-with-examples/) | This method is used with Java List Interface to check if the list contains all the collection of elements. |
| **sort(Comparator comp)** | This method is used with List Interface in Java to sort the elements of the list on the basis of the given [comparator](https://www.geeksforgeeks.org/comparator-interface-java/). |

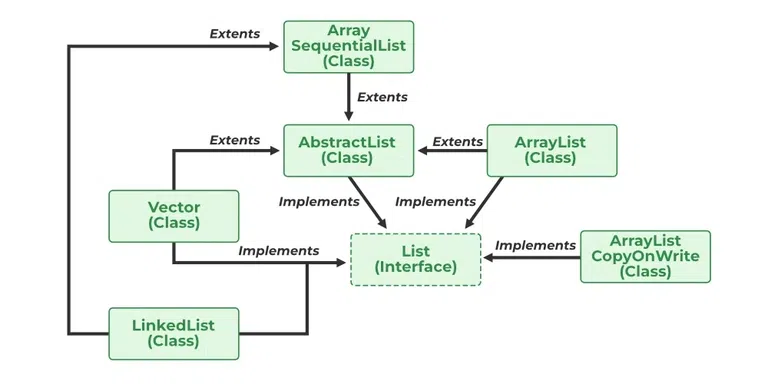
**Java List vs Set**

Both the List interface and the Set interface inherits the Collection interface. However, there exists some differences between them.

| **List** | **Set** |
| --- | --- |
| The List is an ordered sequence. | The Set is an unordered sequence. |
| List allows duplicate elements | Set doesn’t allow duplicate elements. |
| Elements by their position can be accessed. | Position access to elements is not allowed. |
| Multiple null elements can be stored. | The null element can store only once. |
| List implementations are ArrayList, LinkedList, Vector, Stack | Set implementations are HashSet, LinkedHashSet. |

**Classes Association with a Java List Interface**

Now let us discuss the classes that implement the List Interface for which first do refer to the pictorial representation below to have a better understanding of the List interface. It is as follows:

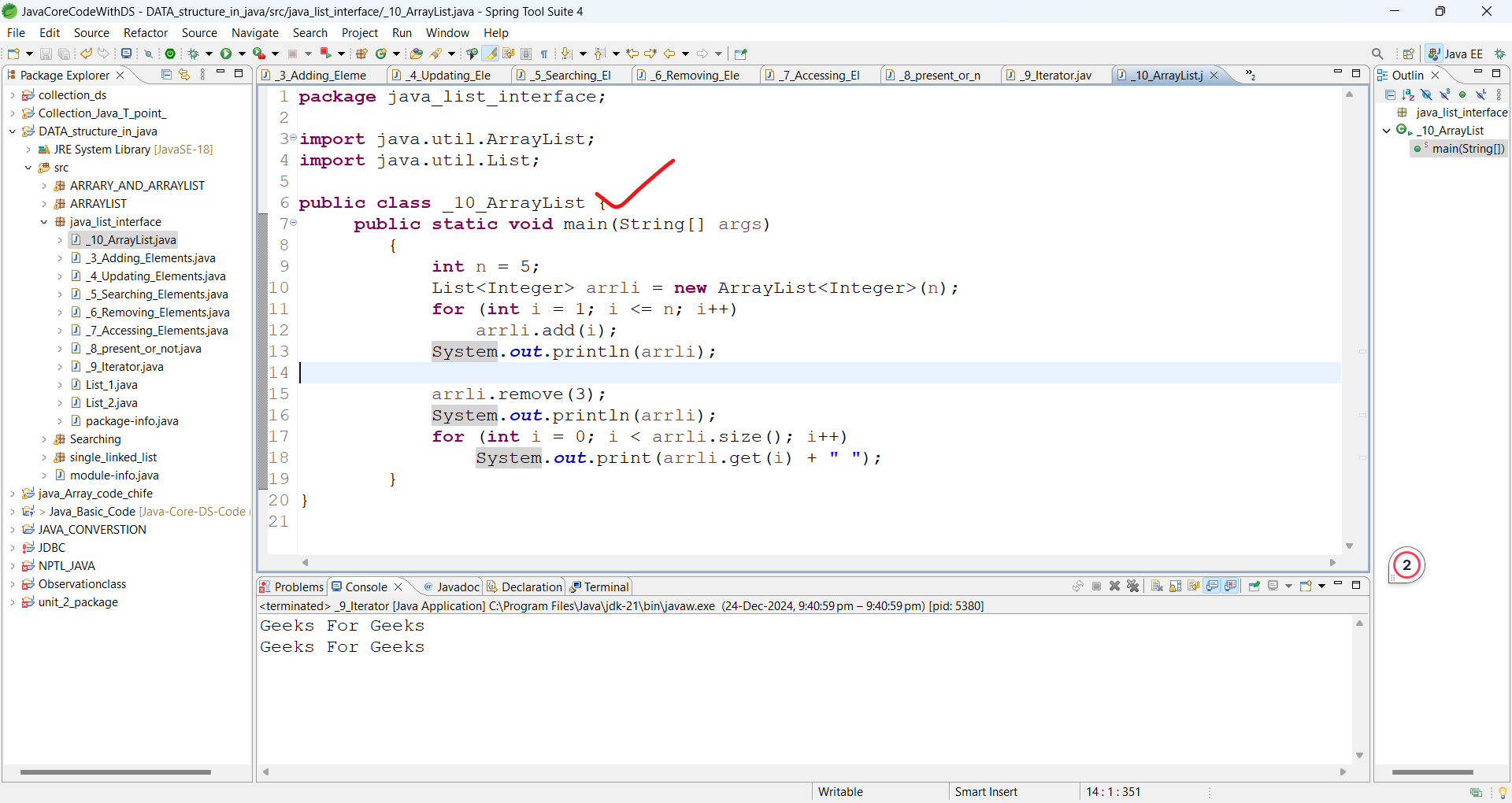


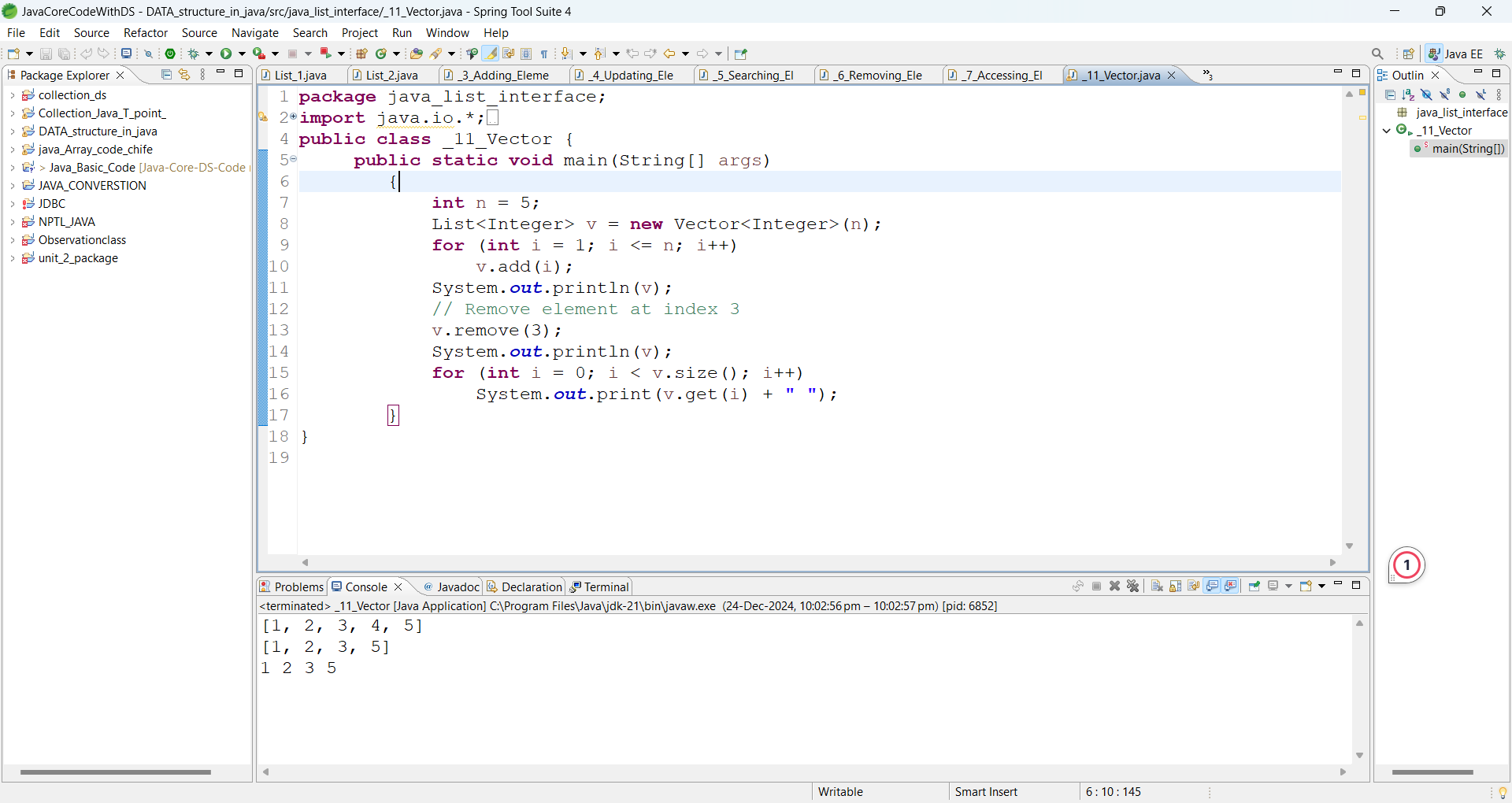
[AbstractList](https://www.geeksforgeeks.org/abstractlist-in-java-with-examples/), [CopyOnWriteArrayList](https://www.geeksforgeeks.org/copyonwritearraylist-in-java/), and the [AbstractSequentialList](https://www.geeksforgeeks.org/abstractsequentiallist-in-java-with-examples/) are the classes that implement the List interface. A separate functionality is implemented in each of the mentioned classes. They are as follows:

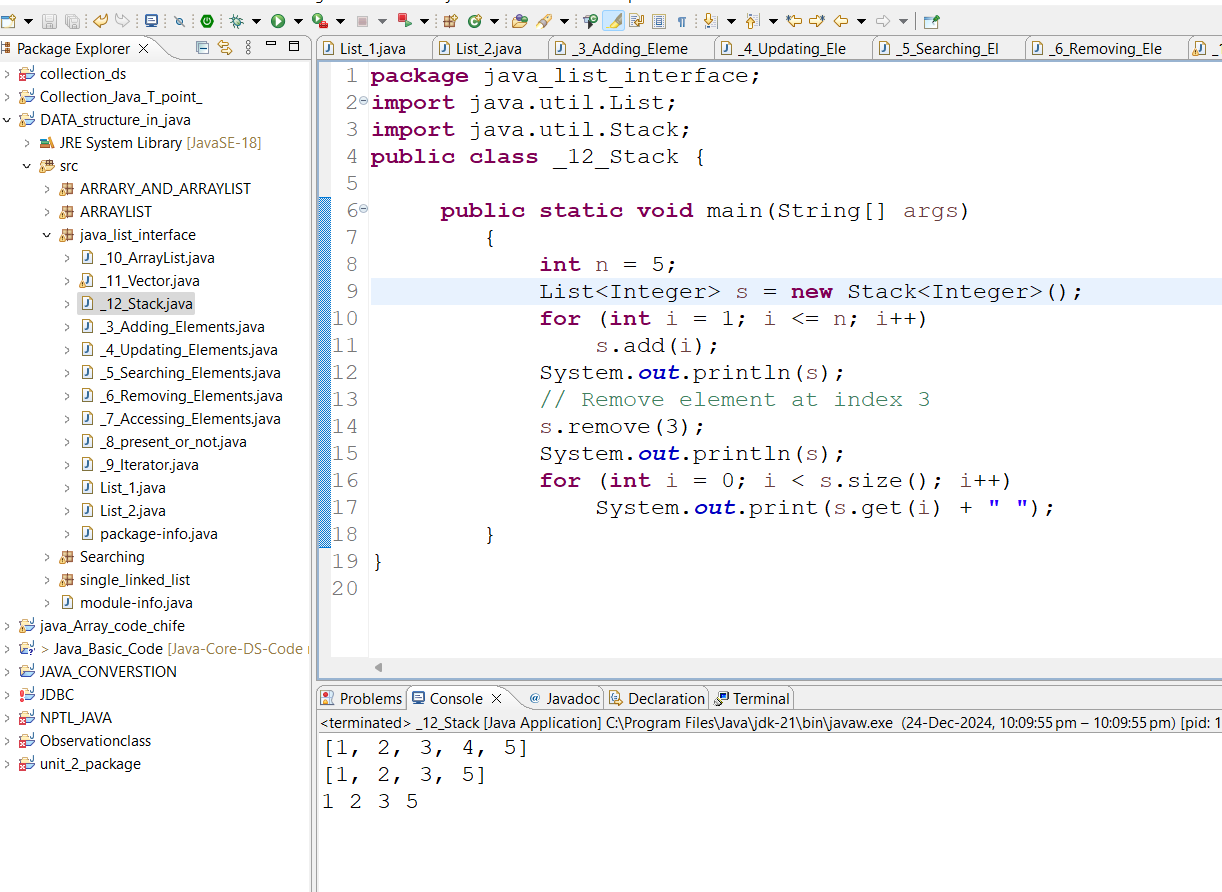
1. **AbstractList:** This class is used to implement an unmodifiable list, for which one needs to only extend this AbstractList Class and implement only the *get()* and the *size()* methods.
2. **CopyOnWriteArrayList:** This class implements the list interface. It is an enhanced version of [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/) in which all the modifications(add, set, remove, etc.) are implemented by making a fresh copy of the list.
3. **AbstractSequentialList:** This class implements the [Collection interface](https://www.geeksforgeeks.org/collections-in-java-2/) and the AbstractCollection class. This class is used to implement an unmodifiable list, for which one needs to only extend this AbstractList Class and implement only the *get()* and the *size()* methods.

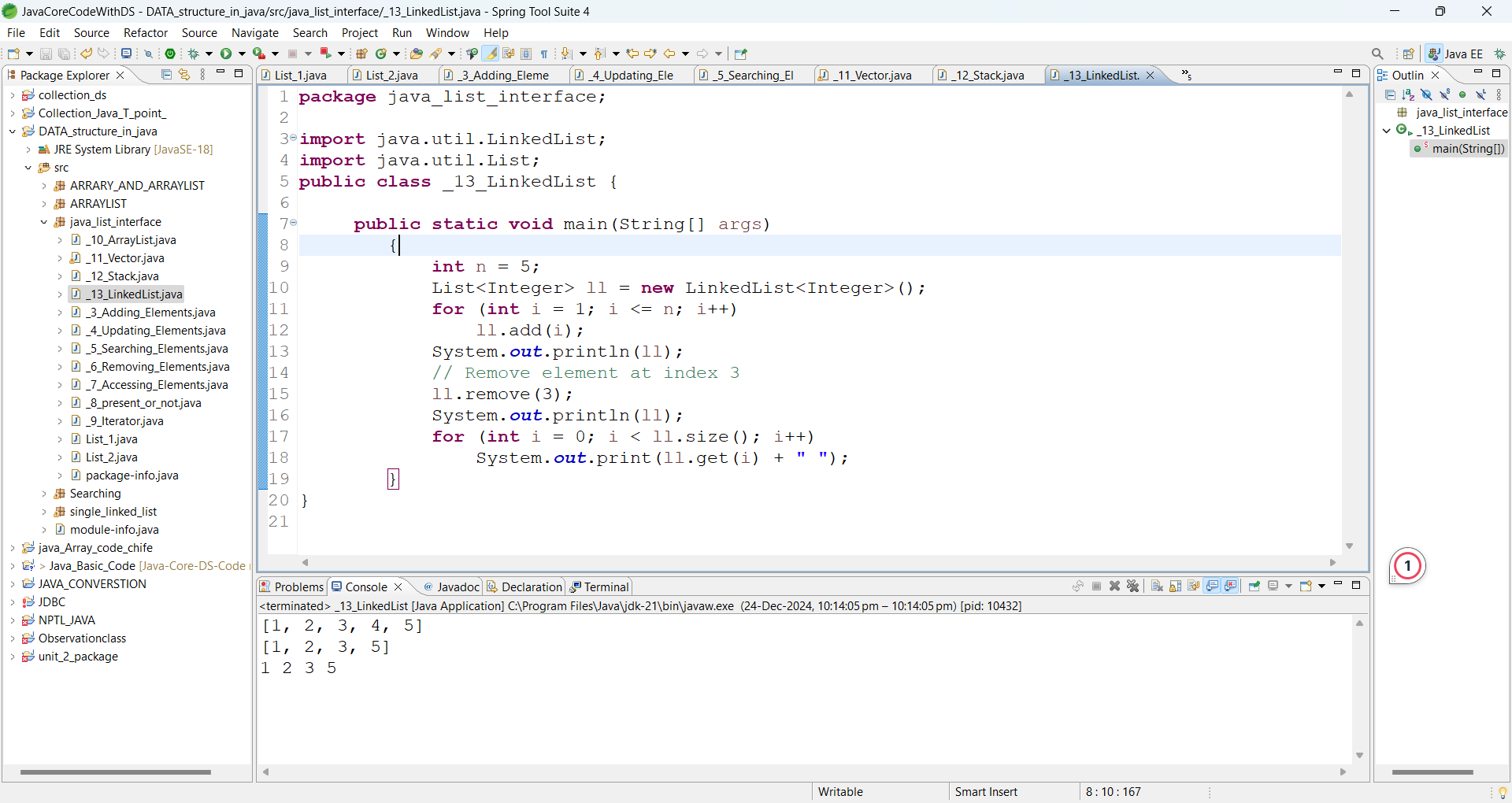
We will proceed in this manner.

* ArrayList
* Vector
* Stack
* LinkedList









**ArrayList in Java**

Java ArrayList is a part of the collections framework and it is a class of java.util package. It provides us with dynamic arrays in Java. Though, it may be slower than standard arrays it can be helpful in programs where lots of manipulation in the array is required. The main advantage of ArrayList is that, unlike normal arrays, we don’t need to mention the size when creating ArrayList. It automatically adjusts its capacity as elements are added or removed.

ArrayLists provide dynamic resizing, making them an essential tool in Java for handling collections.

ArrayList is a Java class implemented using the [List interface](https://www.geeksforgeeks.org/list-interface-java-examples/). Java ArrayList, as the name suggests, provides the functionality of a dynamic array where the size is not fixed as an array. Also, as a part of Collections framework, it has many features not available with arrays.

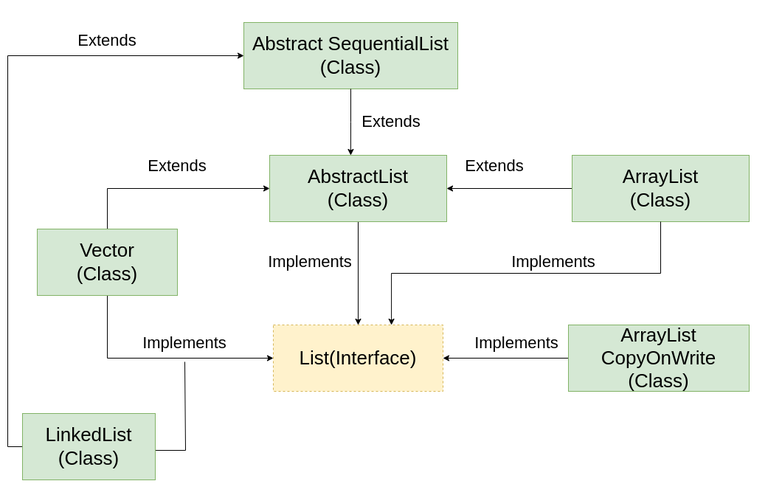
**Important Features of ArrayList in Java**

* ArrayList inherits [AbstractList](https://www.geeksforgeeks.org/abstractlist-in-java-with-examples) class and implements the [List interface](https://www.geeksforgeeks.org/list-interface-java-examples).
* ArrayList is initialized by size. However, the size is increased automatically if the collection grows or shrinks if the [objects](https://www.geeksforgeeks.org/classes-objects-java) are removed from the collection.
* Java ArrayList allows us to randomly access the list.
* ArrayList can not be used for [primitive types](https://www.geeksforgeeks.org/data-types-in-java), like int, char, etc. We need a [wrapper class](https://www.geeksforgeeks.org/wrapper-classes-java) for such cases.
* ArrayList in Java can be seen as a [vector in C++](https://www.geeksforgeeks.org/vector-in-cpp-stl).
* ArrayList is not Synchronized. Its equivalent synchronized class in Java is [Vector](https://www.geeksforgeeks.org/java-util-vector-class-java).

**Some Key Points of ArrayList in Java**

1. **ArrayList is Underlined data Structure Resizable Array or Growable Array.**
2. **ArrayList Duplicates Are Allowed.**
3. **Insertion Order is Preserved.**
4. **Heterogeneous objects are allowed.**
5. **Null insertion is possible.**

**Let’s understand the Java ArrayList in depth. Look at the below image:**



**In the above illustration, [AbstractList](https://www.geeksforgeeks.org/abstractlist-in-java-with-examples), [CopyOnWriteArrayList](https://www.geeksforgeeks.org/copyonwritearraylist-in-java), and [AbstractSequentialList](https://www.geeksforgeeks.org/abstractsequentiallist-in-java-with-examples) are the classes that implement the list interface. A separate functionality is implemented in each of the mentioned classes. They are:**

1. AbstractList: This class is used to implement an unmodifiable list, for which one needs to only extend this AbstractList Class and implement only the *get()* and the *size()* methods.
2. CopyOnWriteArrayList: This class implements the list interface. It is an **enhanced version of [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java) in which all the modifications(add, set, remove, etc.)** are implemented by making a fresh copy of the list.
3. AbstractSequentialList: This class implements the [Collection interface](https://www.geeksforgeeks.org/collections-in-java-2) and the AbstractCollection class. This class is used to implement an unmodifiable list, for which one needs to only extend this AbstractList Class and implement only the *get()* and the *size()* methods.

# Table of Content

# [Syntax of ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/?ref=next_article#syntax-of-arraylist)

# [Constructors in ArrayList in Java](https://www.geeksforgeeks.org/arraylist-in-java/?ref=next_article#constructors-in-arraylist-in-java)

# [Operations in ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/?ref=next_article#operations-in-arraylist)

# [Java ArrayList Methods](https://www.geeksforgeeks.org/arraylist-in-java/?ref=next_article#arraylist-in-java-methods)

### Syntax of ArrayList

*ArrayList<Integer> arr = new ArrayList<Integer>();*

Constructors in ArrayList in Java

* In order to Create an ArrayList, we need to create an object of the ArrayList class.
* The ArrayList class consists of various [constructors](https://www.geeksforgeeks.org/constructors-in-java) which allow the possible creation of the array list.
* The following are the constructors available in this class:

| **Constructor** | **Description** | **Initialize and Declare ArrayList** |
| --- | --- | --- |
| **ArrayList()** | This constructor is used to build an empty array list. | ArrayList arr = new ArrayList(); |
| **ArrayList(Collection c)** | This constructor is used to build an array list initialized with the elements from the collection c. | ArrayList arr = new ArrayList(c); |
| **ArrayList(int capacity)** | This constructor is used to build an array list with the initial capacity being specified. | ArrayList arr = new ArrayList(N); |

**Operations in ArrayList**

Now, Using the constructors we have got ArrayList for further operations like Insertion , Deletion and Updation of the elements in ArrayList.

**1. Adding Elements in ArrayList**

Adding elements seems bit complex when the size of ArrayList is not defined:

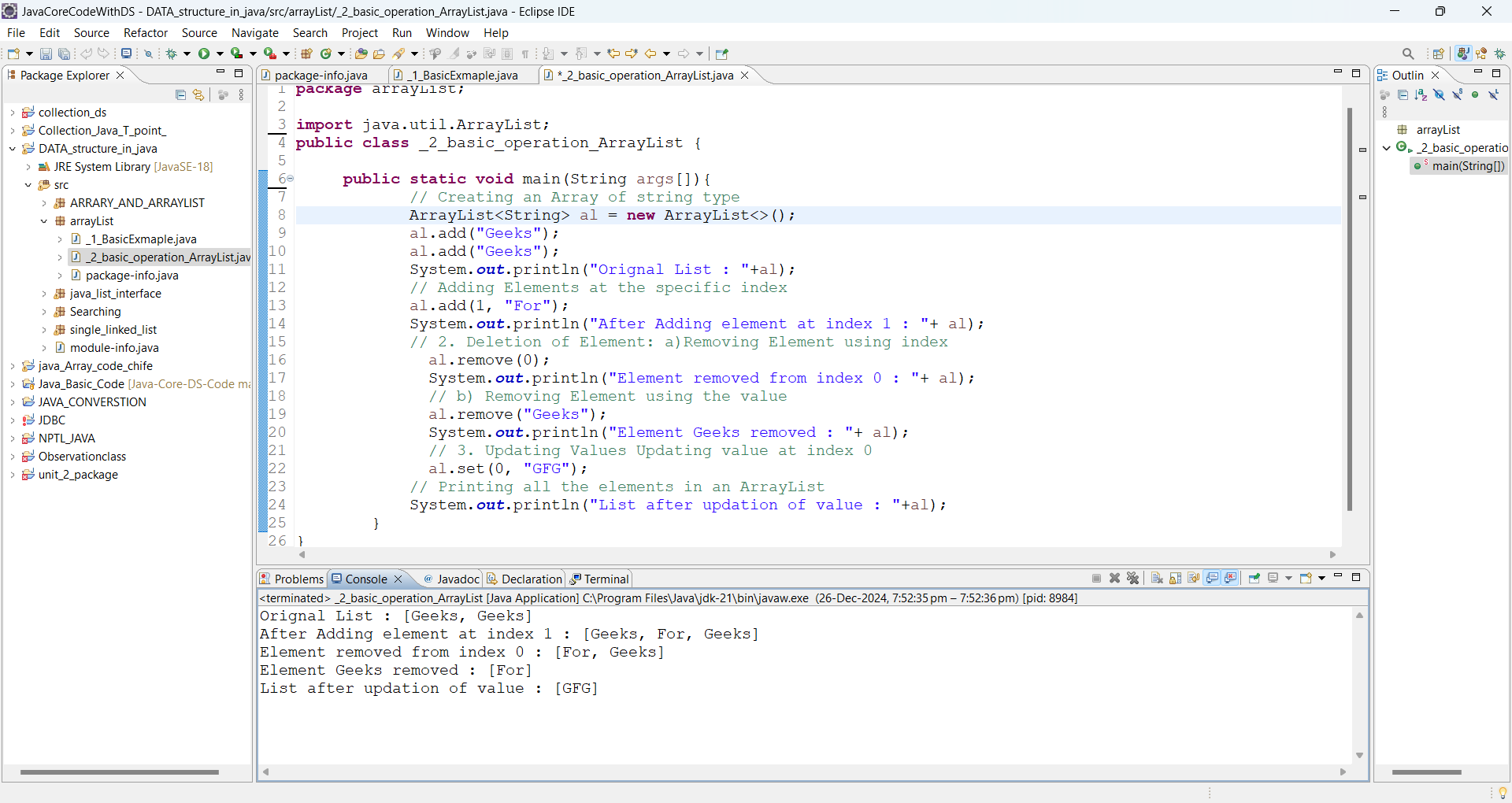
* Creates a bigger-sized memory on heap memory (for example memory of double size).
* Copies the current memory elements to the new memory.
* The new item is added now as there is bigger memory available now.
* Delete the old memory.

2. **Changing Elements in ArrayList**

* After adding the elements, if we wish to change the element, it can be done using the [set()](https://www.geeksforgeeks.org/arraylist-set-method-in-java-with-examples) method.
* Since an ArrayList is indexed, the element which we wish to change is referenced by the index of the element. Therefore, this method takes an index and the updated element which needs to be inserted at that index.

3. **Removing Elements in ArrayList**

* In order to remove an element from an ArrayList, we can use the [remove() method](https://www.geeksforgeeks.org/arraylist-linkedlist-remove-methods-java-examples).
* This method is overloaded to perform multiple operations based on different parameters.



**Complexity of Java ArrayList**

| **Operation** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- |
| **Inserting Element in ArrayList** | **O(1)** | **O(N)** |
| **Removing Element from ArrayList** | **O(N)** | **O(1)** |
| **Traversing Elements in ArrayList** | **O(N)** | **O(N)** |
| **Replacing Elements in ArrayList** | **O(1)** | **O(1)** |

**Advantages of Java ArrayList**

* **Dynamic size:**ArrayList can dynamically grow and shrink in size, making it easy to add or remove elements as needed.
* **Easy to use**: ArrayList is simple to use, making it a popular choice for many Java developers.
* **Fast access**: ArrayList provides fast access to elements, as it is implemented as an array under the hood.
* **Ordered collection**: ArrayList preserves the order of elements, allowing you to access elements in the order they were added.
* **Supports null values**: ArrayList can store null values, making it useful in cases where the absence of a value needs to be represented.

**Disadvantages of Java ArrayList**

* **Slower than arrays**: ArrayList is slower than arrays for certain operations, such as inserting elements in the middle of the list.
* **Increased memory usage**: ArrayList requires more memory than arrays, as it needs to maintain its dynamic size and handle resizing.
* **Not thread-safe:** ArrayList is not thread-safe, meaning that multiple threads may access and modify the list concurrently, leading to potential race conditions and data corruption.
* **Performance degradation**: ArrayList’s performance may degrade as the number of elements in the list increases, especially for operations such as searching for elements or inserting elements in the middle of the list.

**Java ArrayList Methods**

| **Method** | **Description** |
| --- | --- |
| [add(int index, Object element)](https://www.geeksforgeeks.org/java-util-arraylist-add-method-java) | This method is used to insert a specific element at a specific position index in a list. |
| [add(Object o)](https://www.geeksforgeeks.org/java-util-arraylist-add-method-java) | This method is used to append a specific element to the end of a list. |
| [addAll(Collection C)](https://www.geeksforgeeks.org/java-util-arraylist-addall-method-java) | This method is used to append all the elements from a specific collection to the end of the mentioned list, in such an order that the values are returned by the specified collection’s iterator. |
| [addAll(int index, Collection C)](https://www.geeksforgeeks.org/java-util-arraylist-addall-method-java) | Used to insert all of the elements starting at the specified position from a specific collection into the mentioned list. |
| [clear()](https://www.geeksforgeeks.org/arraylist-clear-java-examples) | This method is used to remove all the elements from any list. |
| [clone()](https://www.geeksforgeeks.org/clone-method-in-java-2) | This method is used to return a shallow copy of an ArrayList in Java. |
| [contains(Object o)](https://www.geeksforgeeks.org/arraylist-contains-java) | Returns true if this list contains the specified element. |
| [ensureCapacity(int minCapacity)](https://www.geeksforgeeks.org/arraylist-ensurecapacity-method-in-java-with-examples) | Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument. |
| [forEach(Consumer<? super E> action)](https://www.geeksforgeeks.org/arraylist-foreach-method-in-java) | Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception. |
| [get(int index)](https://www.geeksforgeeks.org/arraylist-get-method-java-examples) | Returns the element at the specified position in this list. |
| [indexOf(Object O)](https://www.geeksforgeeks.org/java-util-arraylist-indexof-java) | The index the first occurrence of a specific element is either returned or -1 in case the element is not in the list. |
| [isEmpty()](https://www.geeksforgeeks.org/arraylist-isempty-java-example) | Returns true if this list contains no elements. |
| [lastIndexOf(Object O)](https://www.geeksforgeeks.org/arraylist-lastindexof-java-example) | The index of the last occurrence of a specific element is either returned or -1 in case the element is not in the list. |
| [listIterator()](https://www.geeksforgeeks.org/arraylist-listiterator-method-in-java-with-examples) | Returns a list iterator over the elements in this list (in proper sequence). |
| [listIterator(int index)](https://www.geeksforgeeks.org/arraylist-listiterator-method-in-java-with-examples) | Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list. |
| [remove(int index)](https://www.geeksforgeeks.org/arraylist-linkedlist-remove-methods-java-examples) | Removes the element at the specified position in this list. |
| [remove(Object o)](https://www.geeksforgeeks.org/arraylist-linkedlist-remove-methods-java-examples) | Removes the first occurrence of the specified element from this list, if it is present. |
| [removeAll(Collection c)](https://www.geeksforgeeks.org/arraylist-removeall-method-in-java-with-examples) | Removes from this list all of its elements that are contained in the specified collection. |
| [removeIf(Predicate filter)](https://www.geeksforgeeks.org/arraylist-removeif-method-in-java) | Removes all of the elements of this collection that satisfy the given predicate. |
| [removeRange(int fromIndex, int toIndex)](https://www.geeksforgeeks.org/arraylist-removerange-java-examples) | Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. |
| [retainAll(Collection<?> c)](https://www.geeksforgeeks.org/arraylist-retainall-method-in-java) | Retains only the elements in this list that are contained in the specified collection. |
| [set(int index, E element)](https://www.geeksforgeeks.org/arraylist-set-method-in-java-with-examples) | Replaces the element at the specified position in this list with the specified element. |
| [size()](https://www.geeksforgeeks.org/arraylist-size-method-in-java-with-examples) | Returns the number of elements in this list. |
| [spliterator?()](https://www.geeksforgeeks.org/arraylist-spliterator-method-in-java) | Creates a late-binding and fail-fast Spliterator over the elements in this list. |
| [subList(int fromIndex, int toIndex)](https://www.geeksforgeeks.org/arraylist-sublist-method-in-java-with-examples) | Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive. |
| [toArray()](https://www.geeksforgeeks.org/arraylist-array-conversion-java-toarray-methods) | This method is used to return an array containing all of the elements in the list in the correct order. |
| [toArray(Object[] O)](https://www.geeksforgeeks.org/arraylist-array-conversion-java-toarray-methods) | It is also used to return an array containing all of the elements in this list in the correct order same as the previous method. |
| [trimToSize()](https://www.geeksforgeeks.org/arraylist-trimtosize-java-example) | This method is used to trim the capacity of the instance of the ArrayList to the list’s current size. |

**FAQs of ArrayList in Java**

**How is ArrayList different from an Array in Java?**

*An ArrayList can resize dynamically, while a traditional array has a fixed size. ArrayList also provides many useful methods like add(), remove(), and size().*

**How to Access elements in an ArrayList?**

*Elements can be accessed using the get() method:  
String element = list.get(0);*

**How to Remove an element from an ArrayList?**

*Use the remove() method to remove elements by index:  
list.remove(0);*

**Is ArrayList Synchronized?**

*No, ArrayList is not synchronized. Use Collections.synchronizedList(new ArrayList<>()) for thread-safe operations.*

**Can we Store null elements in an ArrayList?**

*Yes, ArrayList can store null elements.*

**How is data stored in ArrayList?**

*ArrayList can store data till the ArrayList size is full, after that the size of ArrayList is doubled if we want to store any more elements.*

# Vector Class in Java

The Vector class implements a growable array of objects.

Vectors fall in legacy classes, but now it is fully compatible with collections.

It is found in[java.util package](https://www.geeksforgeeks.org/java-util-package-java/) and implement the [List](https://www.geeksforgeeks.org/list-interface-java-examples/) interface.

* **Thread-Safe**: All methods are synchronized, making it suitable for multi-threaded environments. However, this can lead to performance overhead in single-threaded scenarios.
* **Allows Nulls**: Can store null elements.
* **Enumeration Support**: Provides backward compatibility with Enumeration, a legacy way of iterating over elements.

In Java, the Vector class is a part of the Java Collections Framework and provides a dynamic array implementation of the List interface. It was added in the original release of Java (Java 1.0) and provides many methods for manipulating the elements of a vector, including adding, inserting, and removing elements.

Note that the Vector class is synchronized, meaning that multiple threads can access the same vector without causing problems. However, this synchronization comes at the cost of performance, so if you don’t need to share a vector between multiple threads, it’s generally better to use an alternative class like ArrayList which is not synchronized.

**Example:**

**Few more points about Vector are mentioned below:**

* They are very similar to [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/), but Vector is synchronized and has some legacy methods that the collection framework does not contain.
* It also maintains an insertion order like an ArrayList. Still, it is rarely used in a non-thread environment as it is **synchronized**, and due to this, it gives a poor performance in adding, searching, deleting, and updating its elements.
* The Iterators returned by the Vector class are fail-fast. In the case of concurrent modification, it fails and throws the **ConcurrentModificationException.**

We can use all the methods of the List interface as shown below as follows:



**Syntax:**

*public class Vector<E> extends AbstractList<E>   
implements List<E>, RandomAccess, Cloneable, Serializable*

Here,**E** is the type of element.

* It extends [AbstractList](https://www.geeksforgeeks.org/abstractlist-in-java-with-examples/) and implements [List](https://www.geeksforgeeks.org/list-interface-java-examples/) interfaces.
* It implements Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess interfaces.
* The directly known subclass is [Stack](https://www.geeksforgeeks.org/stack-class-in-java/).

**Important points regarding the Increment of vector capacity are as follows:**

If the increment is specified, Vector will expand according to it in each allocation cycle. Still, if the increment is not specified, the vector’s capacity doubles each allocation cycle. Vector defines three protected data members:

* ***int capacityIncrement:*** Contains the increment value.
* ***int elementCount:*** Number of elements currently in vector stored in it.
* ***Object elementData[]:*** Array that holds the vector is stored in it.

Common Errors in the declaration of Vectors are asfollows**:**

* Vector throws an **IllegalArgumentException** if the InitialSize of the vector defined is negative.
* If the specified collection is null, It throws **NullPointerException**.

### Constructors

**1. Vector():** Creates a default vector of the initial capacity is 10.

*Vector<E> v = new Vector<E>();*

**2. Vector(int size):**Creates a vector whose initial capacity is specified by size.

*Vector<E> v = new Vector<E>(int size);*

**3. Vector(int size, int incr):** Creates a vector whose initial capacity is specified by size and increment is specified by incr. It specifies the number of elements to allocate each time a vector is resized upward.

*Vector<E> v = new Vector<E>(int size, int incr);*

**4. Vector(Collection c):** Creates a vector that contains the elements of collection c.

*Vector<E> v = new Vector<E>(Collection c);*

### Methods in Vector Class

***Note:***

* *If the vector increment is not specified then it’s capacity will be doubled in every increment cycle.*
* *The capacity of a vector cannot be below the size, it may equal to it.*

### Performing Various Operations on Vector class in Java

Let us discuss various operations on Vector class that are listed as follows:

* Adding elements
* Updating elements
* Removing elements
* Iterating over elements

Program:

### **Advantages of using Vector in Java**

* Synchronization: As mentioned before, Vector is synchronized, making it safe to use in a multi-threaded environment.
* Dynamic Size: The size of a Vector can grow or shrink dynamically as elements are added or removed, so you don’t have to worry about setting an initial size that will accommodate all elements.
* Legacy support: Vector has been part of Java since its inception and is still supported, so it’s a good option if you need to work with older Java code that uses Vector.

### **Disadvantages of using Vector in Java**

* Performance: The synchronization in Vector can lead to slower performance compared to other collection classes, such as ArrayList.
* Legacy Code: While Vector is still supported, newer Java code is often written using the more modern collection classes, so it may be harder to find examples and support for Vector.
* Unnecessary overhead: If you don’t need the synchronization features of Vector, using it will add unnecessary overhead to your code.

# LinkedList in Java

Linked List is a part of the [Collection framework](https://www.geeksforgeeks.org/collections-in-java-2/) present in [java.util package](https://www.geeksforgeeks.org/java-util-package-java/). This class is an implementation of the [LinkedList data structure](https://www.geeksforgeeks.org/data-structures/linked-list/) which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses and each element is known as a node.

**Example:**

**Note:** It also has a few disadvantages like the nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach a node we wish to access.

The LinkedList is a versatile data structure, especially useful when you need constant-time insertions and deletions.

## **How Does LinkedList work Internally?**

Since a LinkedList acts as a dynamic array and we do not have to specify the size while creating it, the size of the list automatically increases when we dynamically add and remove items. And also, the elements are not stored in a continuous fashion. Therefore, there is no need to increase the size. Internally, the LinkedList is implemented using the [doubly linked list data structure](https://www.geeksforgeeks.org/doubly-linked-list/).

The main difference between a normal linked list and a doubly LinkedList is that a doubly linked list contains an extra pointer, typically called the previous pointer, together with the next pointer and data which are there in the singly linked list.

## **Constructors in the LinkedList**

In order to create a LinkedList, we need to create an object of the LinkedList class. The LinkedList class consists of various constructors that allow the possible creation of the list. The following are the constructors available in this class:

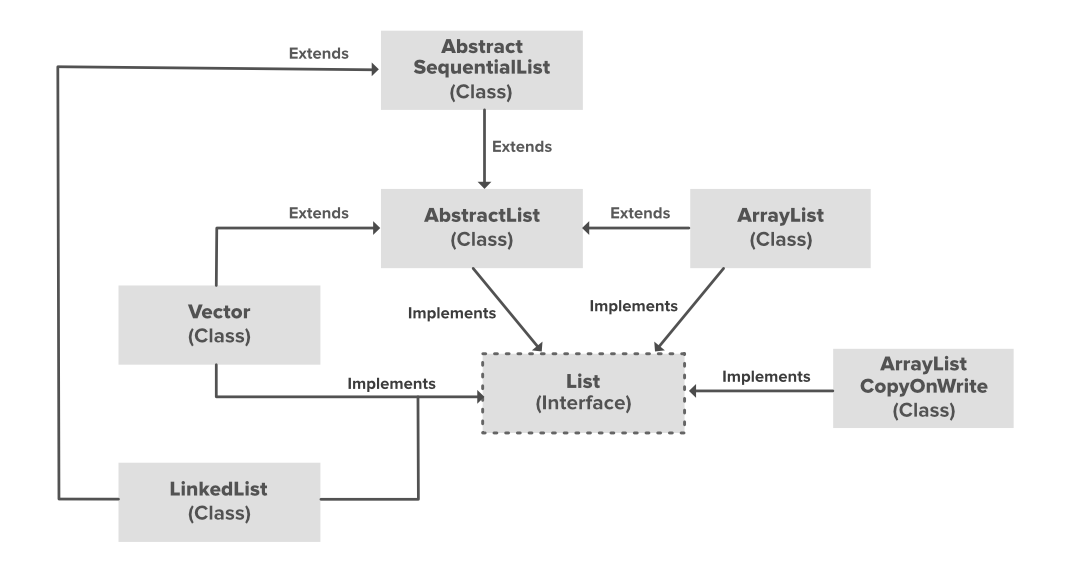
1. **LinkedList():** This constructor is used to create an empty linked list. If we wish to create an empty LinkedList with the name ll, then, it can be created as:

*LinkedList ll = new LinkedList();*

2. **LinkedList(Collection C):** This constructor is used to create an ordered list that contains all the elements of a specified collection, as returned by the collection’s iterator. If we wish to create a LinkedList with the name ll, then, it can be created as:

*LinkedList ll = new LinkedList(C);*

## **Methods for Java LinkedList**



### Performing Various Operations on LinkedList

* Adding elements
* Updating elements
* Removing elements
* Iterating over elements
* To Array();
* Size();
* remove First();
* remove last();

examples:

### Advantages of using LinkedList in Java

* Dynamic size: As with Vector, the size of a LinkedList can grow or shrink dynamically, so you don’t have to worry about setting an initial size.
* Efficient Insertions and Deletions: LinkedList is an efficient data structure for inserting or deleting elements in the middle of the list because you only need to change the links between elements, rather than shifting all elements after the insertion or deletion point.
* Flexible Iteration: With a linked list, you can efficiently iterate through the list in either direction, since each element has a reference to both its predecessor and successor elements.

### Disadvantages of using LinkedList in Java

* Performance: LinkedList has a slower performance than ArrayList when it comes to accessing individual elements. This is because you need to traverse the list to reach the desired element, whereas with ArrayList, you can simply access the desired element using an index.
* Memory overhead: LinkedList requires more memory than ArrayList because each element requires additional memory for the links to its predecessor and successor elements.

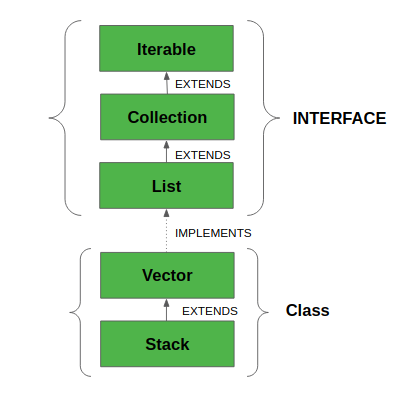
# Stack Class in Java

Java [Collection framework](https://www.geeksforgeeks.org/collections-in-java-2/) provides a Stack class that models and implements a [**Stack data structure**](https://www.geeksforgeeks.org/stack-data-structure/). The class is based on the basic principle of **LIFO**(last-in-first-out). In addition to the basic push and pop operations, the class provides three more functions of empty, search, and peek.

* The Stack class extends Vector and provides additional functionality specifically for stack operations, such as **push, pop, peek, empty,**and **search**.
* The Stack class can indeed be referred to as a subclass of Vector, inheriting its methods and properties.

**Example:**

The below diagram shows the **hierarchy of the Stack class**:



The class supports one default constructor **Stack()** which is used to create an empty stack.

### **Declaration of Stack**

*public class Stack<E> extends Vector<E>*

**All Implemented Interfaces:**

* **Serializable:** It is a marker interface that classes must implement if they are to be serialized and deserialized.
* **Cloneable:** This is an interface in Java which needs to be implemented by a class to allow its objects to be cloned.
* **Iterable<E>:** This interface represents a collection of objects which is iterable — meaning which can be iterated.
* **Collection<E>:** A Collection represents a group of objects known as its elements. The Collection interface is used to pass around collections of objects where maximum generality is desired.
* [**List<E>:**](https://www.geeksforgeeks.org/list-interface-java-examples/)The List interface provides a way to store the ordered collection. It is a child interface of Collection.
* **RandomAccess:** This is a marker interface used by List implementations to indicate that they support fast (generally constant time) random access.

### **How to Create a Stack?**

In order to create a stack, we must import **java.util.stack** package and use the Stack() constructor of this class. The below example creates an empty Stack.

*Stack<E> stack = new Stack<E>();*

Here E is the type of Object.

**Example:**

### Performing various operations on Stack class

1. **Adding Elements:**
2. **Accessing the Element:**
3. **Removing Elements:**

## **Methods in Stack Class**

**Set in Java**