Chapter 1 Collections

A collection in Java is a group of individual objects that are treated as a single unit

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

Needed for a Collection Framework

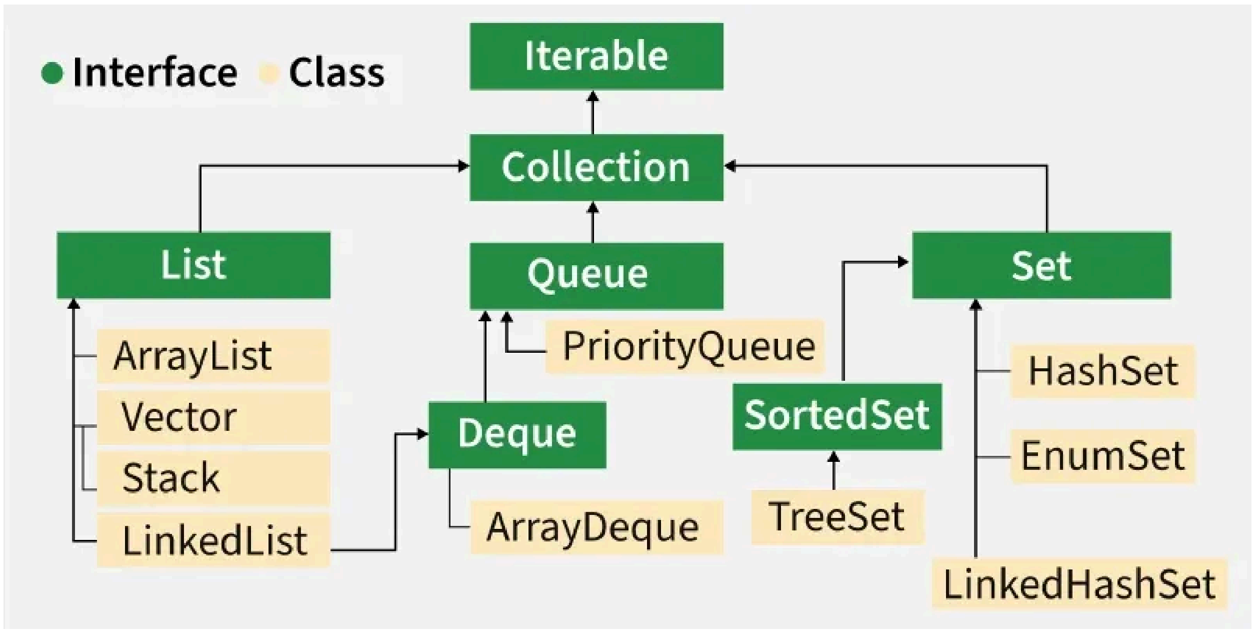
Before the Collection Framework (before JDK 1.2), Java used Arrays, Vectors and Hashtables to group objects, but they lacked a common interface. Each had a separate implementation, making usage inconsistent and harder for developers to learn and maintain.

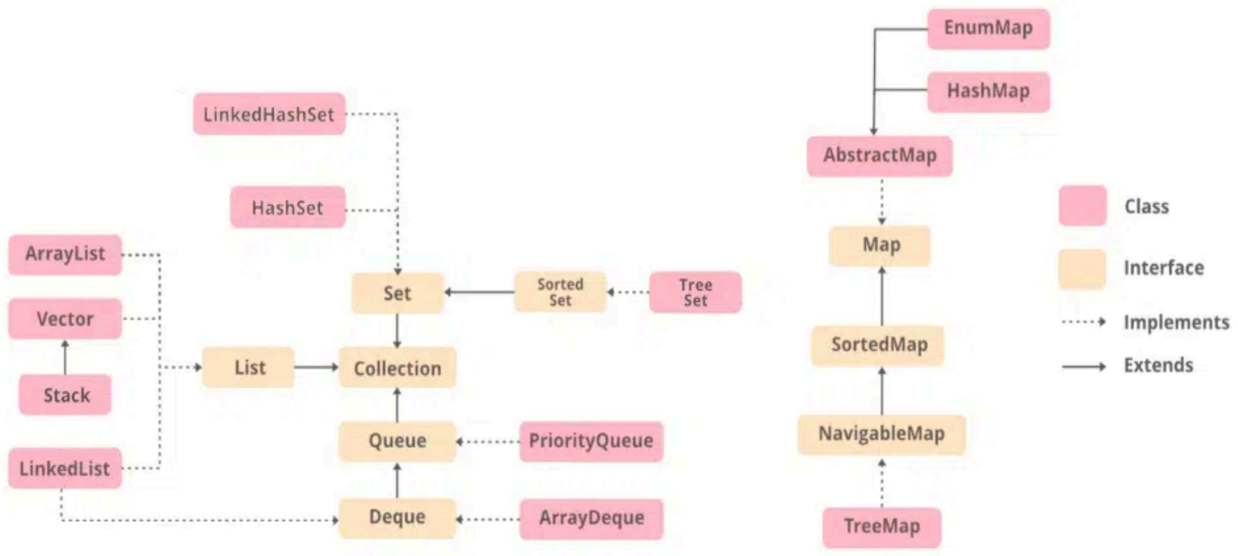
```
import java.io.*;
import java.util.*;
class CollectionDemo {
    public static void main(String[] args)
    {
        // Creating instances of the array, vector and hashtable
        int arr[] = new int[] { 1, 2, 3, 4 };
        Vector<Integer> v = new Vector();
        Hashtable<Integer, String> h = new Hashtable();
        // Adding the elements into the vector
        v.addElement(1);
        v.addElement(2);
        // Adding the element into the hashtable
        h.put(1, "geeks");
        h.put(2, "4geeks");
        // Accessing the first element of the array, vector and hashtable
        System.out.println(arr[0]);
        System.out.println(v.elementAt(0));
        System.out.println(h.get(1));
    }
}
```

Advantages of the Java Collection Framework

Since the lack of a collection framework gave rise to the above set of disadvantages, the following are the advantages of the collection framework.

- **Consistent API:** Interfaces like List, Set, and Map have common methods across classes (ArrayList, LinkedList, etc.)**.**
- **Less Coding Effort:** Developers focus on usage, not designing data structures—supports OOP abstraction.
- **Better Performance:** Offers fast, reliable implementations of data structures, improving speed and quality of code.





Interfaces that Extend the Java Collections Interface

The collection framework contains multiple interfaces where every interface is used to store a specific type of data. The following are the interfaces present in the framework.

1. Iterable Interface

Iterable interface is the root of the Collection Framework. It is extended by the Collection interface, making all collections inherently iterable. Its primary purpose is to provide an Iterator to traverse elements, defined by its single abstract method iterator().

Iterator iterator();

2. Collection Interface

Collection interface extends Iterable and serves as the foundation of the Collection Framework. It defines common methods like add(), remove(), and clear(), ensuring consistency and reusability across all collection implementations.

3. List Interface

List interface extends the Collection interface and represents an ordered collection that allows duplicate elements. It is implemented by classes like ArrayList, Vector, and Stack. Since all these classes implement List, a list object can be instantiated using any of them

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 which implement the List interface are as follows:

ArrayList

ArrayList provides a dynamic array in Java that resizes automatically as elements are added or removed. Although slower than standard arrays, it is efficient for frequent modifications. It supports random access but cannot store primitive types directly—wrapper classes like Integer or Character are required

Let's understand the ArrayList with the following example:

```
import java.io.*;

import java.util.*;

class GFG {

    // Main Method

    public static void main(String[] args)

    {
}

        // Declaring the ArrayList with initial size n

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



        // Appending new elements at the end of the list

        for (int i = 1; i <= 5; i++)

            al.add(i);

        // Printing elements

        System.out.println(al);

        // Remove element at index 3

        al.remove(3);

        // Displaying the ArrayList after deletion

        System.out.println(al);

        // Printing elements one by one

        for (int i = 0; i < al.size(); i++)

            System.out.print(al.get(i) + " ");

    }

}
```

LinkedList:

LinkedList is a linear data structure where elements (nodes) are stored non-contiguously. Each node contains data and a reference to the next (and optionally previous) node, forming a chain of elements linked by pointers.

Let's understand the LinkedList with the following example:

```
import java.io.*;
import java.util.*;

class GFG {

   // Main Method
   public static void main(String[] args)
   {

       // Declaring the LinkedList
       LinkedList<Integer> ll = new LinkedList<Integer>();

       // Appending new elements at
       // the end of the list
       for (int i = 1; i <= 5; i++)
           ll.add(i);

       // Printing elements
       System.out.println(ll);

       // Remove element at index 3
       ll.remove(3);

       // Displaying the List
       // after deletion
       System.out.println(ll);

       // Printing elements one by one
       for (int i = 0; i < ll.size(); i++)
```

```
            System.out.print(ll.get(i) + " ");
    }
}
```

Output

```
[1, 2, 3, 4, 5]
```

```
[1, 2, 3, 5]
```

```
1 2 3 5
```

Vector:

Vector provides a dynamic array in Java, similar to ArrayList, but with synchronized methods for thread safety. While slower due to synchronization overhead, it is useful in multi-threaded environments. Like ArrayList, it resizes automatically during element manipulation.

Let's understand the Vector with an example:

```
import java.io.*;
import java.util.*;

class GFG {

    // Main Method
    public static void main(String[] args)
    {

        // Declaring the Vector
        Vector<Integer> v = new Vector<Integer>();

        // Appending new elements at the end of the list
        for (int i = 1; i <= 5; i++)
```

```
            v.add(i);

        // Printing elements
        System.out.println(v);

        // Remove element at index 3
        v.remove(3);

        // Displaying the Vector after deletion
        System.out.println(v);

        // Printing elements one by one
        for (int i = 0; i < v.size(); i++)
            System.out.print(v.get(i) + " ");
    }
}
```

Output

```
[1, 2, 3, 4, 5]

[1, 2, 3, 5]

1 2 3 5
```

Stack

Stack class implements the LIFO (last-in-first-out) data structure. It supports core operations like push() and pop(), along with peek(), empty(), and search(). Stack is a subclass of Vector and inherits its properties.

Let's understand the stack with an example:

```
import java.util.*;
public class GFG {
   // Main Method
   public static void main(String args[])
   {
       Stack<String> stack = new Stack<String>();
       stack.push("Geeks");
       stack.push("For");
       stack.push("Geeks");
       stack.push("Geeks");

       // Iterator for the stack
       Iterator<String> itr = stack.iterator();

       // Printing the stack
       while (itr.hasNext()) {
           System.out.print(itr.next() + " ");
       }

       System.out.println();

       stack.pop();

       // Iterator for the stack
       itr = stack.iterator();

       // Printing the stack
       while (itr.hasNext()) {
           System.out.print(itr.next() + " ");
       }
   }
}
```

Output

```
Geeks For Geeks Geeks

Geeks For Geeks
```

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 alternate to Stack is to use ArrayDequeue which is not thread-safe and has faster array implementation.*

4. Queue Interface

The Queue interface follows the FIFO (First-In, First-Out) principle, where elements are processed in the order they are added—similar to a real-world queue (e.g., ticket booking). It is used when order matters. Classes like PriorityQueue and ArrayDeque implement this interface, allowing queue objects to be instantiated accordingly.

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

PriorityQueue processes elements based on their priority rather than insertion order. It uses a priority heap for internal storage. Elements are ordered either by their natural ordering or by a custom Comparator provided at construction.

Let's understand the priority queue with an example:

```java
import java.util.*;

class GfG {

   // Main Method
   public static void main(String args[])
   {
       // Creating empty priority queue
       PriorityQueue<Integer> pQueue
           = new PriorityQueue<Integer>();

       // Adding items to the pQueue using add()
       pQueue.add(10);
       pQueue.add(20);
       pQueue.add(15);

       // Printing the top element of PriorityQueue
       System.out.println(pQueue.peek());

       // Printing the top element and removing it
       // from the PriorityQueue container
       System.out.println(pQueue.poll());

       // Printing the top element again
       System.out.println(pQueue.peek());
   }
}
```

Output

```
10

10

15
```

5. Deque Interface

Deque interface extends Queue and allows insertion and removal of elements from both ends. It is implemented by classes like ArrayDeque, which can be used to instantiate a Deque object.

For example:

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

Where T is the type of the object.

The class which implements the deque interface is ArrayDeque.

ArrayDeque

ArrayDeque class implements a resizable, double-ended queue that allows insertion and removal from both ends. It has no capacity restrictions and grows automatically as needed.

Let's understand ArrayDeque with an example:

```java
import java.util.*;
public class ArrayDequeDemo {
   public static void main(String[] args)
   {
       // Initializing an deque
       ArrayDeque<Integer> de_que
           = new ArrayDeque<Integer>(10);

       // add() method to insert
       de_que.add(10);
       de_que.add(20);
       de_que.add(30);
       de_que.add(40);
       de_que.add(50);

       System.out.println(de_que);

       // clear() method
       de_que.clear();

       // addFirst() method to insert the
       // elements at the head
       de_que.addFirst(564);
       de_que.addFirst(291);

       // addLast() method to insert the
       // elements at the tail
       de_que.addLast(24);
       de_que.addLast(14);

       System.out.println(de_que);
   }
}
```

Output

```
[10, 20, 30, 40, 50]
```

```
[291, 564, 24, 14]
```

6. Set Interface

Set interface represents an unordered collection that stores only unique elements (no duplicates). It's implemented by classes like HashSet, TreeSet, and LinkedHashSet, and can be instantiated using any of these
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:

HashSet

HashSet class implements a hash table and stores elements based on their hash codes. It does not guarantee insertion order and allows one null element.

Example:

```java
import java.util.*;

public class HashSetDemo {

    // Main Method
    public static void main(String args[])
    {
        // Creating HashSet and
        // adding elements
        HashSet<String> hs = new HashSet<String>();

        hs.add("Geeks");
        hs.add("For");
        hs.add("Geeks");
        hs.add("Is");
        hs.add("Very helpful");

        // Traversing elements
        Iterator<String> itr = hs.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

Output

```
Very helpful

Geeks

For

Is
```

LinkedHashSet

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:

```java
import java.util.*;

public class LinkedHashSetDemo {

   // Main Method
   public static void main(String args[])
   {
       // Creating LinkedHashSet and adding elements
       LinkedHashSet<String> lhs
           = new LinkedHashSet<String>();

       lhs.add("Geeks");
       lhs.add("For");
       lhs.add("Geeks");
       lhs.add("Is");
       lhs.add("Very helpful");

       // Traversing elements
       Iterator<String> itr = lhs.iterator();
       while (itr.hasNext()) {
           System.out.println(itr.next());
       }
   }
}
```

Output

```
Geeks

For

Is
```

```
Very helpful
```

7. Sorted Set Interface

Sorted Set interface extends Set and maintains elements in sorted order. It includes additional methods for range views and ordering. It is implemented by the TreeSet class.

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

TreeSet uses a self-balancing tree (Red-Black Tree) to store elements in sorted order. It maintains natural ordering or uses a custom Comparator if provided during creation. Ordering must be consistent with equals to ensure proper Set behavior.

Let's understand TreeSet with an example:

```
import java.util.*;

public class TreeSetDemo {
   // Main Method
   public static void main(String args[])
   {
       // Creating TreeSet and
       // adding elements
       TreeSet<String> ts = new TreeSet<String>();

       ts.add("Geeks");
       ts.add("For");
       ts.add("Geeks");
       ts.add("Is");
       ts.add("Very helpful");

       // Traversing elements
       Iterator<String> itr = ts.iterator();
       while (itr.hasNext()) {
           System.out.println(itr.next());
       }
   }
}
```

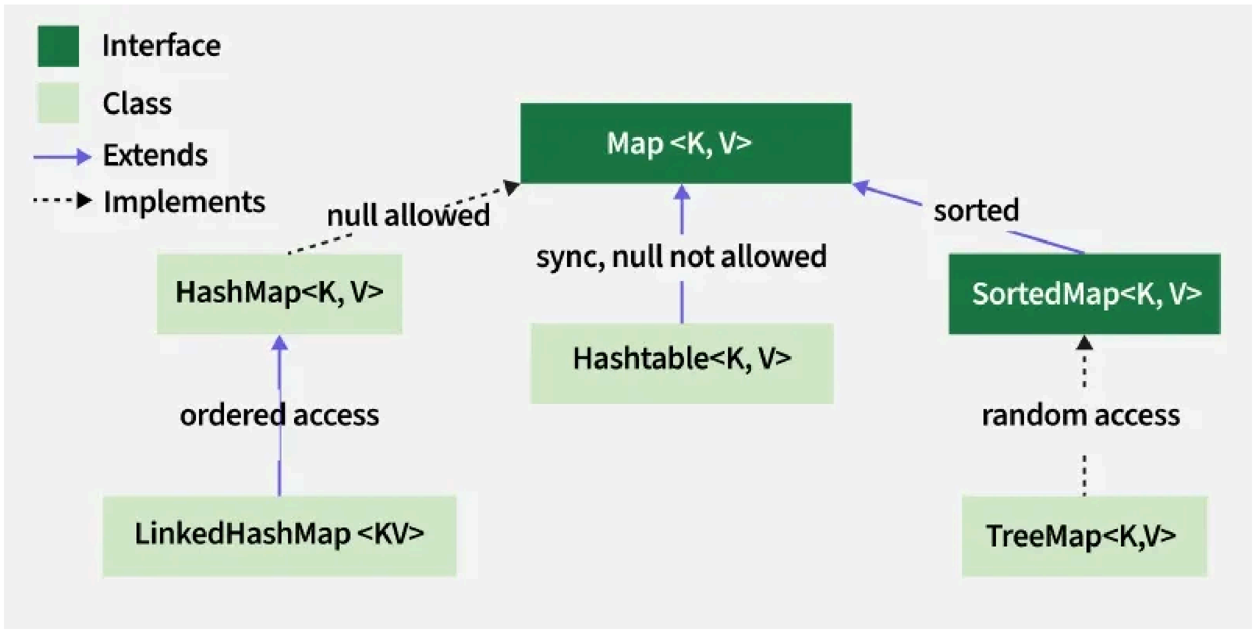
Output

```
For

Geeks

Is

Very helpful
```

Map Interface

Map is a data structure that supports the key-value pair for mapping the data. 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 useful if there is data and we wish to perform operations on the basis of the key. This map interface is implemented by various classes like HashMap, TreeMap, etc. Since all the subclasses implement the map, we can instantiate a map object with any of these classes.

Hierarchy of Map Interface



The frequently used implementation of a Map interface is a HashMap.

HashMap

HashMap is a basic implementation of the Map interface that stores data as key-value pairs. It uses hashing for fast access, converting keys into hash codes to index values efficiently. To retrieve a value, the corresponding key is required. Internally, HashSet is also backed by a HashMap.

Let's understand the HashMap with an example:

```
import java.util.*;
public class HashMapDemo {

   // Main Method
   public static void main(String args[])
   {
       // Creating HashMap and
       // adding elements
       HashMap<Integer, String> hm
           = new HashMap<Integer, String>();

       hm.put(1, "Geeks");
       hm.put(2, "For");
       hm.put(3, "Geeks");

       // Finding the value for a key
       System.out.println("Value for 1 is " + hm.get(1));

       // Traversing through the HashMap
       for (Map.Entry<Integer, String> e : hm.entrySet())
           System.out.println(e.getKey() + " "
                              + e.getValue());
   }
}
```

Output

```
Value for 1 is Geeks
```

```
1 Geeks
2 For
3 Geeks
```

Interface

1.Comparator

Used to order the object of user defined class.Comparator is an interface in java.util package.It is used to define custom sorting rules for objects.Useful when

a.You cannot modify the class .

b.You want multiple ways to sort the same objects (e.g., by name, by age).

The Comparator interface define 2 methods

i.compare()-

here compare 2 objects or elements

a. int compare(Object obj1.Object 2)

return 0;// both object are same

return 1;//positive value if obj1 is greater than obj2

return -1;//other wise negative value

b.list.sort(Comparator.comparingInt(String::length));

(String::length)-"Take a String and get its length"

Comparator.comparingInt(...)-"Compare strings using their length (an int)"

c.list.sort(Comparator.comparingInt(String::length).reversed());

.reversed()-in reverse order**.**

Example:

```
 // using Compare()method the elements from collections
import java.util.*;

import java.util.List;

import java.util.ArrayList;

class Main {

    public static void main(String[] args) {

        List<String> list = new ArrayList();

        Collections.addAll(list, "Banana", "Apple", "Kiwi", "Mango");

/*        Collections.sort(list, new Comparator<String>(){

            public int compare(String a, String b) {

                return a.length() - b.length();

                    } });*/

        list.sort(Comparator.comparingInt(String::length));

             System.out.println(list);

    }  }
```

2.Iterator

Using iterator traverse in one direction (forward).

Easiest way to display all elements of collection

Which object implements either the Iterator or ListIterator interface .iterator enables us through collection ,display or removing elements ,traversing list ,modification in list.

Only one element of collection can access at time By using iterator object.Collection class provides iterator()-return start of collection

ListIterator extends to Iterator .

Steps of how we access

1.Obtain iterator to start Collection by calling the collection’s iterator method.

2.set up loop that makes call to hasNext().have the loop iterator as long as hasNext()return true.

3.Within the loop obtain each element by calling next();

Example:

```
import java.util.*;

public class Main {

    public static void main(String[] args) {

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

        Collections.addAll(list, "Apple", "Banana", "Mango");

        Iterator<String> it = list.iterator(); // create iterator

        while (it.hasNext()) {              // check if more elements

                    System.out.println( it.next());  // get next element
        }
   } }
```

3.ListIterator

Is used to iterate through elements of collection class

Using this traverse the collection class both in direction backward and forward

Method

1.add(object obj)- adding/inserting element in to list returned by next().

2. hasNext()-return true if there is next element o.w false.

3.next()-return next element .

4.previous()- for backward traversal.

4. nextIndex() and previousIndex()- You can get the index of the next or previous element using For modifying the list.

5.remove()- remove current element from list.

6.set()- assign obj to current element and returned.

Example:

```
import java.util.ArrayList;

import java.util.ListIterator;

public class ListIteratorExample {

    public static void main(String[] args) {
```

```
//1. Create a List collection (ArrayList) and add elements

ArrayList<String> cars = new ArrayList<>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

// 2. Obtain a ListIterator from the list

ListIterator<String> it = cars.listIterator();

System.out.println("--- Traversing forward ---");

// 3. Loop forward using hasNext() and next()

while (it.hasNext()) {

    String element = it.next();

    System.out.println(element);

    // Example of modifying the list: replace "Ford" with "Honda"

    if (element.equals("Ford")) {

        it.set("Honda");

    }

}

System.out.println("\n--- Traversing backward (after reaching the end) ---");
```

```
        // 4. Loop backward using hasPrevious() and previous()

        // The iterator is now at the end of the list after the forward traversal.

        while (it.hasPrevious()) {

          System.out.println(it.previous());

        }

        System.out.println("\n--- List after modification ---");

        // 5. Print the final list to see the modification

        System.out.println(cars);

    }}
```

4.Enumeration Interface

Used java.util.Enumeration interface . and defines the method which you can enumerate the element in collection of objects one at time .

Methods

The Enumeration interface provides two methods for iteration:

- boolean hasMoreElements()- Returns true if there are more elements to be extracted, false otherwise.
- Object nextElement()- Returns the next element in the sequence and advances the cursor.

Example:

```
import java.util.Enumeration;
import java.util.Vector;

public class EnumerationExample {

    public static void main(String[] args) {

        Vector<String> animals = new Vector<>();

        animals.add("Dog");

        animals.add("Cat");

        animals.add("Elephant");

        // Get the enumeration object

        Enumeration<String> e = animals.elements();

        // Iterate through the elements

        while (e.hasMoreElements()) {

            System.out.println(e.nextElement());

        }   } }
```

Methods of the Collection Interface

This interface contains various methods which can be directly used by all the collections which implement this interface. They are:

Methods of the Collection Interface

This interface contains various methods which can be directly used by all the collections which implement this interface. They are:

Method	Description
add(Object)	This method is used to add an object to the collection.
addAll(Collection c)	This method adds all the elements in the given collection to this collection.
clear()	This method removes all of the elements from this collection.
contains(Object o)	This method returns true if the collection contains the specified element.

containsAll(Collection c)	This method returns true if the collection contains all of the elements in the given collection.
equals(Object o)	This method compares the specified object with this collection for equality.
hashCode()	This method is used to return the hash code value for this collection.
isEmpty()	This method returns true if this collection contains no elements.

iterator()	This method returns an iterator over the elements in this collection.
parallelStream()	This method returns a parallel Stream with this collection as its source.
remove(Object o)	This method is used to remove the given object from the collection. If there are duplicate values, then this method removes the first occurrence of the object.

removeAll(Collection c)	This method is used to remove all the objects mentioned in the given collection from the collection.
removeIf(Predicate filter)	This method is used to remove all the elements of this collection that satisfy the given predicate.
retainAll(Collection c)	This method is used to retain only the elements in this collection that are contained in the specified collection.

size()	This method is used to return the number of elements in the collection.
spliterator()	This method is used to create a Spliterator over the elements in this collection.
stream()	This method is used to return a sequential Stream with this collection as its source.
toArray()	This method is used to return an array containing all of the elements in this collection.

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() method of **java.util.Collections** class is used to add all the specified elements to the specified collection. Elements to be added may be specified individually or as an array.

Example:

```
// Adding Elements
// Using addAll() method

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Geeks {

   public static void main(String[] args) {

       List<String> l = new ArrayList<>();

       // Adding elements to the list
       l.add("Shoes");
       l.add("Toys");

       // Add one or more elements
```

```
        Collections.addAll(l, "Fruits", "Bat", "Ball");

        for (int i = 0; i < l.size(); i++) {
            System.out.print(l.get(i) + " ");
        }
    }
}
```

Output

```
Shoes Toys Fruits Bat Ball
```

2. Sorting a Collection

Collections.sort() is used to sort the elements present in the specified list of Collections in ascending order. Collections.reverseOrder() is used to sort in descending order.

Example:

```
// Sorting a Collections using sort() method
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Geeks {

   public static void main(String[] args) {

        List<String> l = new ArrayList<>();

        // Adding elements to the list
        // using add() method
        l.add("Shoes");
        l.add("Toys");

        // Adding one or more
        // element using addAll()
        Collections.addAll(l, "Fruits", "Bat", "Mouse");

        // Sorting according to default ordering
        // using sort() method
        Collections.sort(l);

        // Printing the elements
        for (int i = 0; i < l.size(); i++) {
            System.out.print(l.get(i) + " ");
        }

        System.out.println();

        // Sorting according to reverse ordering
        Collections.sort(l, Collections.reverseOrder());

        // Printing the reverse order
        for (int i = 0; i < l.size(); i++) {
            System.out.print(l.get(i) + " ");
        }
    }
}
```

Output

```
Bat Fruits Mouse Shoes Toys

Toys Shoes Mouse Fruits Bat
```

3. Searching in a Collection

Collections.binarySearch() 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 method will be wrong. If the element exists in the list, the method will return the position of the element in the sorted list, if the element does not exist in the list then this method will return a negative number that shows where the item would be inserted in the list - 1.

Example:

```
// Binary Search using Collections.binarySearch()
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Geeks {

   public static void main(String[] args) {

        List<String> l = new ArrayList<>();

        // Adding elements to object
        // using add() method
        l.add("Shoes");
        l.add("Toys");
        l.add("Horse");
        l.add("Ball");
        l.add("Grapes");

        // Sort the List
        Collections.sort(l);

        // BinarySearch on the List
        System.out.println(
                "The index of Horse is: "
                + Collections.binarySearch(l, "Horse"));

            // BinarySearch on the List
            System.out.println(
                "The index of Dog is: "
                + Collections.binarySearch(l, "Dog"));
    }
}
```

Output

```
The index of Horse is: 2

The index of Dog is: -2
```

Note: The list must be sorted before using binarySearch to get the correct results.

4. Copying Elements

The copy() method of Collections class is used to copy all the elements from one list into another. After the operation, the index of each copied element in the destination list will be identical to its index in the source list. The destination list must be at least as long as the source list. If it is longer, the remaining elements in the destination list are unaffected.

Example:

```java
// Copying Elements using copy() method
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Geeks {

   public static void main(String[] args) {

       List<String> l1 = new ArrayList<>();

       // Add elements
       l1.add("Shoes");
       l1.add("Toys");
       l1.add("Horse");
       l1.add("Tiger");

       // Print the elements
       System.out.println(
           "The Original Destination list is: ");

       for (int i = 0; i < l1.size(); i++) {
           System.out.print(l1.get(i) + " ");
       }
       System.out.println();

       // Create source list
       List<String> l2 = new ArrayList<>();

       // Add elements
       l2.add("Bat");
       l2.add("Frog");
       l2.add("Lion");

       // Copy the elements from source to destination
       Collections.copy(l1, l2);
```

```
        // Printing the modified list
        System.out.println(
            "The Destination List After copying is: ");

        for (int i = 0; i < l1.size(); i++) {
            System.out.print(l1.get(i) + " ");
        }
    }
}
```

Output

```
The Original Destination list is:

Shoes Toys Horse Tiger

The Destination List After copying is:

Bat Frog Lion Tiger
```

5. Disjoint CollectionCollections.disjoint() is used to check whether two specified collections have nothing in common. It returns true if the two collections do not have any element in common.

Example:

```
// Working of Disjoint Function
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Geeks {

   public static void main(String[] args) {

        List<String> l1 = new ArrayList<>();
       // Add elements to l1
       l1.add("Shoes");
       l1.add("Toys");
       l1.add("Horse");
       l1.add("Tiger");

       List<String> l2 = new ArrayList<>();

       // Add elements to l2
       l2.add("Bat");
       l2.add("Frog");
       l2.add("Lion");

       // Check if disjoint or not
       System.out.println(
           Collections.disjoint(l1, l2));
   }
}
```

Output

```
True
```