COURSE:- Java Programming

Sem:- I

Module: 5

By:

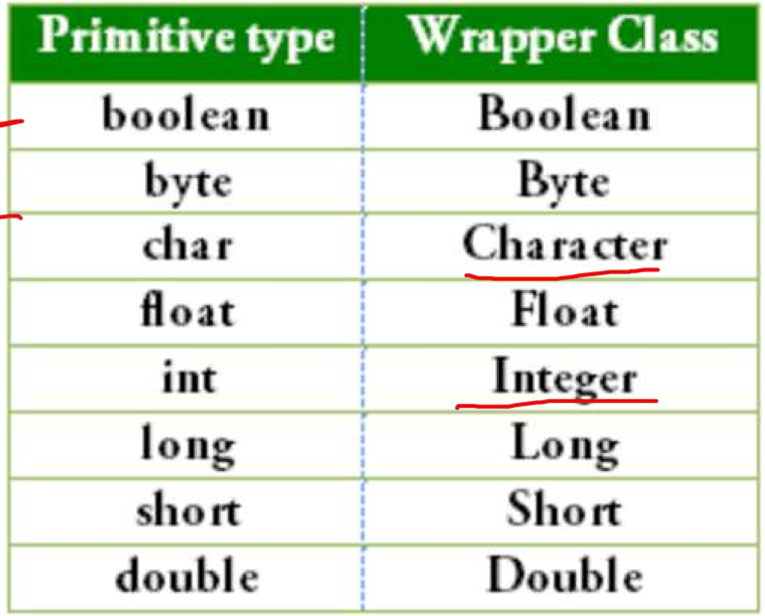
Dr. Ritu Jain,
Associate Professor

Collections Framework

- A framework is a set of classes and interfaces which provide a ready-made architecture.
- The **java.util** package contains Collections Framework.
- It is a sophisticated hierarchy of **interfaces and classes** that helps in *managing groups of objects* efficiently.
- Java Collections can achieve *all the operations that you perform* on a data such as searching, sorting, insertion, manipulation, and deletion.
- Java **Collections Framework** gives the programmer access to *pre-packaged data structures as well as to algorithms* for manipulating them .
- Collection can hold only objects, not primitives (such as int, double). For this wrapper classes are used.

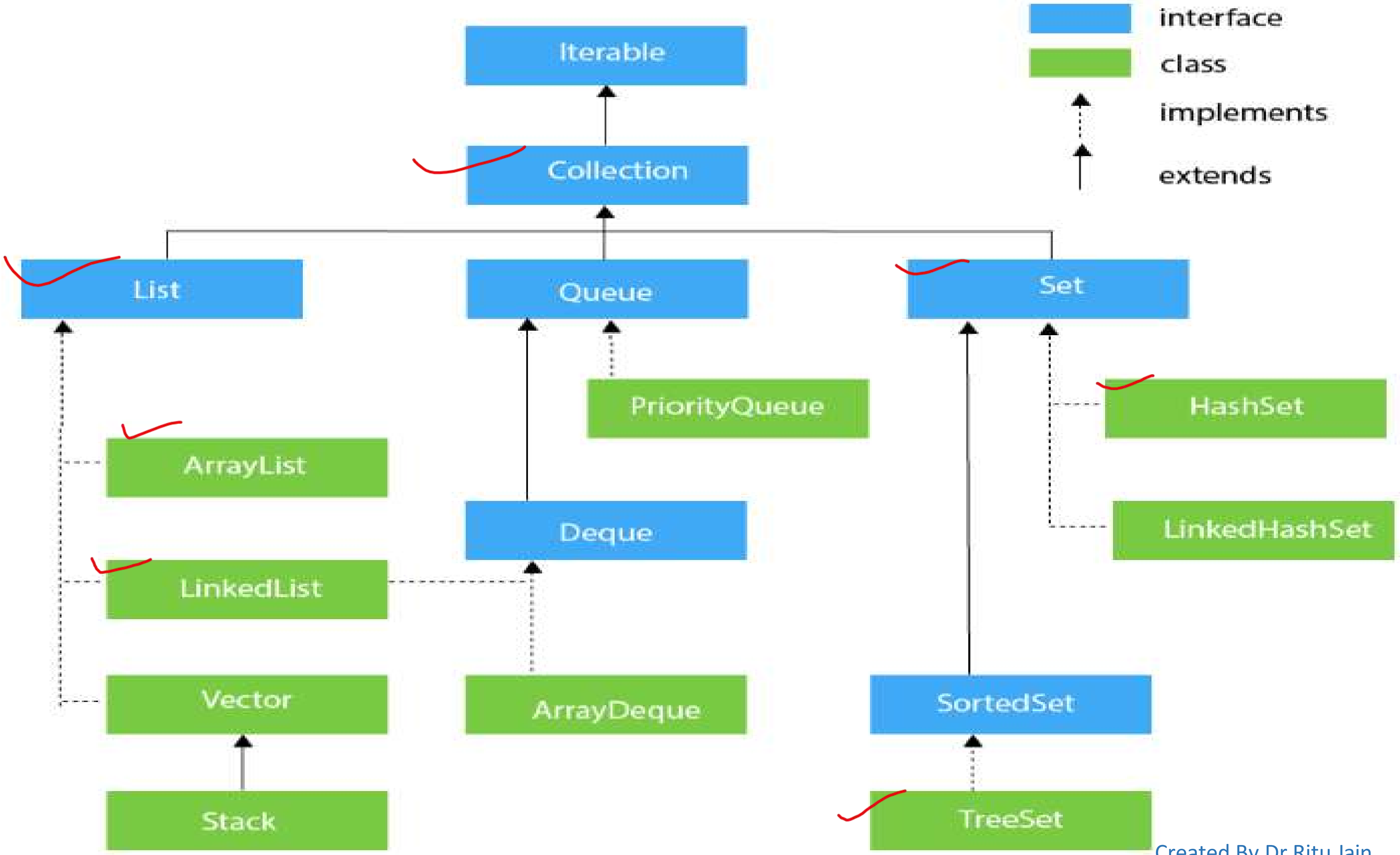
Wrapper Classes

- **Wrapper classes** are Java classes that were created to hold one primitive data value. Examples are Integer, Double, Byte, Short. Objects of these types hold one value of their corresponding primitive type (int, double, byte, short). They are used when you desire to store primitive data types in Java structures that require objects (e.g. ArrayLists).



Primitive type	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

Hierarchy of Collection Framework



Created By Dr Ritu Jain

Need for Collection Framework

- Before Java 1.2, data was stored using arrays, vectors, and hash tables, which had limitations:
    - **Fixed size** → arrays can’t grow dynamically
    - **Lack of common architecture** → each class had different methods
    - **No in-built algorithms** → sorting/searching required manual coding
    - **No type safety** (before generics)
    - The Collection Framework solved all these problems.

Features of Collection Framework

1. **Dynamic and Resizable:** Collections automatically **grow or shrink** depending on the number of elements.

2. **Unified Architecture:** All collections follow common interfaces (Collection, List, Set), making them easy to learn and use.

3. **Reusability:** Common data structures like lists, sets, stacks, and queues are readily available.

4. **Type Safety via Generics**

   Example:

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

   Only Strings allowed → no runtime ClassCastException.

5. **High Performance:** Most classes use optimized algorithms for fast insertion, deletion, and search.

6. **In-built Algorithms:** Class Collections provides Sorting, Searching, Shuffling

7. **Iterable Support:** Every collection can be traversed using Iterator, ListIterator, Enhanced for-loop

8. **Ready-to-use Data Structures:** Includes ArrayList, LinkedList, HashSet, TreeSet, PriorityQueue etc

Advantages of the Java Collection Framework

The primary advantages of a collections framework are that it:

- **Reduces programming effort** by providing data structures and algorithms so you don't have to write them yourself.
- **Increases performance** by providing high-performance implementations of data structures and algorithms.
- **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
- **Reduces the effort required to learn API:** Same method names across classes (add, remove, size)
- **Fosters software reuse** by providing a standard interface for collections and algorithms with which to manipulate them.
- **Flexibility:** Dynamic resizing
- **Maintainability:** Code is cleaner and easier to understand

Interfaces in Collection Framework

- **Iterable:**
  - super interface for all the collection classes.
  - The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement it.
  - An object that implements this **interface** allows it to be the target of the "for each" statement. The for-each loop is **used** for iterating over collections.
  - The Java Iterable interface has only one method named iterator().
    - This method return a Java Iterator which can be used to iterate the elements of the object implementing the Iterable interface.

Characteristics of Collection Interface

- It represents a general-purpose container of objects.
- Does **NOT** support indexed access (unlike List).
- Provides fundamental operations applicable to all collections.
- Subinterfaces (List, Set, Queue) add more specialized behavior.

Interfaces in Collection Framework (cont'd)

- **java.util.Collection:** root interface of the collection hierarchy. A collection represents a group of objects known as its *elements*.
- It has three sub-interfaces:
  - **List:** an ordered collection. Lists can contain duplicate elements.
  - **Queue:** Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed.
  - **Set:** collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets.
- An **ordered collection** is a data structure where the position of each element is fixed and determined by the order in which it was added or by a specific, defined criteria.

List and Set in Java

- Both of them are used to store objects and provides a convenient API to insert, remove and retrieve elements, along with to support Iteration over collection
  1. Fundamental difference between List and Set in Java is **allowing duplicate elements**. List in Java allows duplicates while **Set doesn't allow any duplicate**.
  2. **List is an Ordered Collection while Set is an unordered collection.**
     1. List maintains **insertion order of elements**, means any element which is inserted before will go on lower index than any element which is inserted after.
     2. **Set in Java doesn't maintain any order. Though Set provide another alternative called SortedSet which can store Set elements in specific Sorting order.**
  3. Popular implementation of List interface in Java includes ArrayList, Vector, and LinkedList. While popular implementation of the Set interface includes HashSet, TreeSet, and LinkedHashSet.

Differentiate between List and Set

List	Set
1. The List is an indexed sequence.	1. The Set is an non-indexed sequence.
2. List allows duplicate elements	2. Set doesn't allow duplicate elements.
3. Elements by their position can be accessed.	3. Position access to elements is not allowed.
4. Multiple null elements can be stored.	4. Null element can store only once.
5. List implementations are ArrayList, LinkedList, Vector, Stack	5. Set implementations are HashSet, LinkedHashSet.

When to use List and when to use Set

Use a List when:

- **Order matters**: You need elements in a specific sequence (e.g., steps in a recipe, a time-ordered stream of data).
- **Duplicates are allowed**: You expect or need to store the same item multiple times (e.g., multiple entries for the same score).
- **Positional access is needed**: You need to get, add, or remove elements at a specific index (e.g., the 5th item).
- **Example**: A list of steps in a tutorial, a list of songs in a playlist, or sensor readings over time.

Use a Set when:

- **Uniqueness is required**: You only want distinct items (e.g., unique participants in a survey, tags for a blog post).
- **Order doesn't matter**: The arrangement of items isn't important (though some Set implementations like LinkedHashSet maintain insertion order).
- **Fast lookups**: You frequently need to check if an item is already in the collection (e.g., contains() operation is much faster in Sets).
- **Example**: A collection of unique usernames, a bag of unique items in a game, or mathematical set operations like unions/intersections.

Collection Interface

- The **Collection** interface is the foundation upon which the Collections Framework is built because it must be implemented by any class that defines a collection.
- **Collection** is a generic interface that has this declaration: interface Collection<E>
- Here, **E** specifies the type of objects that the collection will hold.
- **Collection** extends the **Iterable** interface. This means that all collections can be cycled through by use of the for-each style **for** loop.

Collection methods

- **public boolean add(Object obj)** : This method is able to add the specified element to Collection object.
    - If added successfully: return true, else: return false

- **public boolean addAll(Collection c)** : This method can be used to add all the elements of the specified Collection to the present Collection object.
    - If collection changed(i.e. elements are added): return true, else: return false

- **public boolean remove(Object obj)**: This method can be used to remove the specified element from the Collection object.
    - If removed successfully: return true, else: return false
- **public boolean removeAll(Collection c)** : remove all the elements of the specified Collection from the invoking Collection object.
    - If collection changed(i.e. elements were removed): return true, else: return false

- **public void clear():** Remove all elements from the invoking collection.

- **public boolean contains(Object obj)** : This method will check whether the specified element exists or not in the Collection object

Collection methods

- **public boolean containsAll(Collection c)** : return true, if the invoking collection contains all element of c. Otherwise, return false.

- **public boolean retainAll(Collection c)**: remove all the elements from the invoking Collection object except the elements those in c. Return true, if the collection changed. Otherwise, return false.

- **boolean isEmpty()**: return true, if invoking collection is empty.

- **Object [] toArray():** Converts collection to array

- **Iterator iterator():** Return an iterator for the invoking collection.

- **int size():** return number of elements in the invoking collection.

List Interface

- List Interface is the sub interface of Collection.
- It contains index-based methods to insert and delete elements.
- **List** interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements.
- Elements can be inserted or accessed by their position in the list, using a zero-based index.
- Dynamic Size: List grows/shrinks automatically.
- A list may contain duplicate elements.

- **List** is a generic interface that has this declaration:

  interface List<E>

- Can use ListIterator interface also with List

Methods of List Interface

Method	**Description**
void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
boolean addAll(int index, Collection c)	Insert all elements of c into invoking list at the specified index. Preexisting elements are shifted up.
E get(int index)	It is used to fetch the element from the particular position of the list.
int indexOf(Object o)	returns the index of the **first instance** of obj in the invoking list. or **-1** if this list does not contain the element.
int lastIndexOf(Object 0)	returns the index of the la**st instance** of obj in the invoking list. , or **-1** if this list does not contain the element.

Methods of List Interface (cont’d)

Methods	Description
boolean contains(Object o)	It returns true if the list contains the specified element
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
ListIterator <E> listIterator()	Returns an iterator to the start of the invoking list
List <E>subList(int start, int end)	Return a list that includes elements from start to end-1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

Created By Dr Ritu Jain

Advantages of List Interface

Advantages of List Interface

- Maintains insertion order
- Allows duplicates
- Provides index-based access
- Supports advanced iterators
- Flexible and easy to use

Array verses ArrayList

Aspect	Array	ArrayList
Dimensionality	An array can be single-dimensional or multi-dimensional	ArrayList can be only a single-dimensional
Traversing Elements	Uses for and foreach loop for iteration	uses for-each, Iteraor or ListIterator
Length/Size	The length keyword gives the total size of the array	size() method returns the number of elements in the ArrayList
Size	Array size is static and fixed length	ArrayList size is dynamic
Speed	Array speed is fast due to the fixed size	ArrayList speed is relatively slower due to resizing and dynamic behaviour
Primitive Data Storage	Directly stores primitive data types (e.g., int, double)	Primitive data types are not directly added to unlikely arrays, they are added indirectly with the help of autoboxing and unboxing
Generics	Not supported, making arrays type-unsafe	Supports generics, making ArrayList type-safe
Adding Elements	Uses assignment (arr[0] = value)	Uses add() method

When to use Array

When to Use an Array

Arrays are a fundamental, low-level data structure in Java and are best suited for performance-critical scenarios with static data requirements.

- **Fixed Size:** Use an array when the size of the collection is known beforehand and will not change during execution.
- **Performance is Critical:** Arrays offer better performance due to direct memory access and less overhead compared to ArrayList, which is important for high-performance applications like 3D graphics.
- **Storing Primitive Data Types:** Arrays can store both primitive types (like int, char, float) and objects directly, which avoids the overhead of Java's autoboxing and wrapper classes required by ArrayList.
- **Multi-dimensional Data:** Arrays natively support multi-dimensional structures (e.g., int[][]), which is useful for representing matrices or grids.

When to Use an ArrayList

ArrayList is part of the Java Collections Framework and provides more flexibility and convenience than a raw array.

- **Dynamic Size:** Use an ArrayList when you do not know the number of elements in advance or when the size needs to change (grow or shrink) dynamically during the program's execution.
- **Frequent Modifications:** ArrayList provides built-in methods like add(), remove(), clear(), and contains() which make adding, deleting, and searching for elements much easier than manual array manipulation.
- **Working with Objects and Generics:** ArrayList uses generics, providing type-safe collections at compile time (e.g., ArrayList<String>). It can only store objects (wrapper classes are used for primitives).
- **Convenience and Readability:** For most general-purpose use cases, ArrayList is the preferred choice as it handles many common data management tasks automatically, leading to cleaner and more maintainable code. It is good practice to refer to it using the interface, e.g., List<String> list = new ArrayList<>();.

ArrayList class

- **ArrayList** class implements the **List** interface.
- **ArrayList** is a generic class that has this declaration:
  - class ArrayList<E>
- Here, **E** specifies the type of objects that the list will hold.
- Java ArrayList class uses a dynamic array for storing the elements.
- In Java, standard arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold. But, sometimes, you may not know until run time precisely how large an array you need. To handle this situation, the Collections Framework defines **ArrayList**.
- In essence, an **ArrayList** is a variable-length array of object references. That is, an **ArrayList** can dynamically increase or decrease in size. Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk.

ArrayList class

- Its initial capacity is 10. This means that an internal array of size 10 is allocated to store elements.
- It is important to distinguish between the capacity and the size of an ArrayList:
  - **Capacity:** The total number of elements the internal array can hold before it needs to be resized.
  - **Size:** The actual number of elements currently present in the ArrayList.
- When you first create an ArrayList with the default constructor, its capacity is 10, but its size is 0 because no elements have been added yet.
- As you add elements, the size increases. If you add more than 10 elements, the ArrayList will automatically resize its internal array to accommodate the new elements, typically by increasing the capacity by 50% of the old capacity.

ArrayList class

- The important points about Java ArrayList class are:
  - can contain duplicate elements.
  - maintains insertion order.
  - non synchronized. It means ArrayList does not include any internal mechanisms (like locks or mutexes) to prevent data corruption when multiple threads access and modify it concurrently.
  - allows random access because array works at the index basis.
  - manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

Syntax: ArrayList<E> list1 = new ArrayList<E>();

Constructors of ArrayList

Constructor	Description
ArrayList()	build an empty array list.
ArrayList(Collection c)	build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	build an array list that has the specified initial capacity.

Iterator interface

- Iterator: Interface provides the facility of iterating the elements **only in forward fashion.**
- An iterator provides a means of *enumerating the contents of a collection (accessing elements one by one)*
- Helps to perform both read and remove operations.
- Iterator is used by all collection interface classes to access elements..

Using iterator in your Java Program

- If we want to use Iterator in java applications then we have to use the following steps.
  - **Create Iterator Object**: To create Iterator object we have to use the *public Iterator iterator()* method.

    Example: Iterator it = al.iterator();
  - **Retrieve Elements from Iterator:** To retrieve elements from Iterator we have to use the following steps.
    - Check whether next element exists or not from the current cursor position by using *public boolean hasNext()* method.This method will return true if next element exists otherwise **return false if no element.**
    - If next element exists then read next element and move cursor to next position by using the public Object *next()* method.
  - **Remove Element:** To remove an element available at current cursor position then we have to use the public void remove() method

Methods of iterator interface

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.

Ways to iterate the elements of the collection in java

- There are various ways to traverse the collection elements. Some of them are as follows:
  - for loop.
  - for-each loop.
  - Iterator interface.
  - ListIterator interface.

Iterable and iterator interface

- In Java, Iterable and Iterator are two distinct interfaces that work in conjunction to enable iteration over collections of elements.
- **Iterable:** The Iterable interface signifies that an object can be "iterated over," meaning it can produce an Iterator.
- It defines a single method: iterator(), which returns an Iterator instance for the collection.
- Classes that implement Iterable can be used directly in an enhanced for-each loop (e.g., for (Element e : iterableObject)), as the loop implicitly calls the iterator() method to obtain an Iterator.

ArrayList

- Although the capacity of an **ArrayList** object increases automatically as objects are stored in it, you can increase the capacity of an **ArrayList** object manually by calling **ensureCapacity( )**.

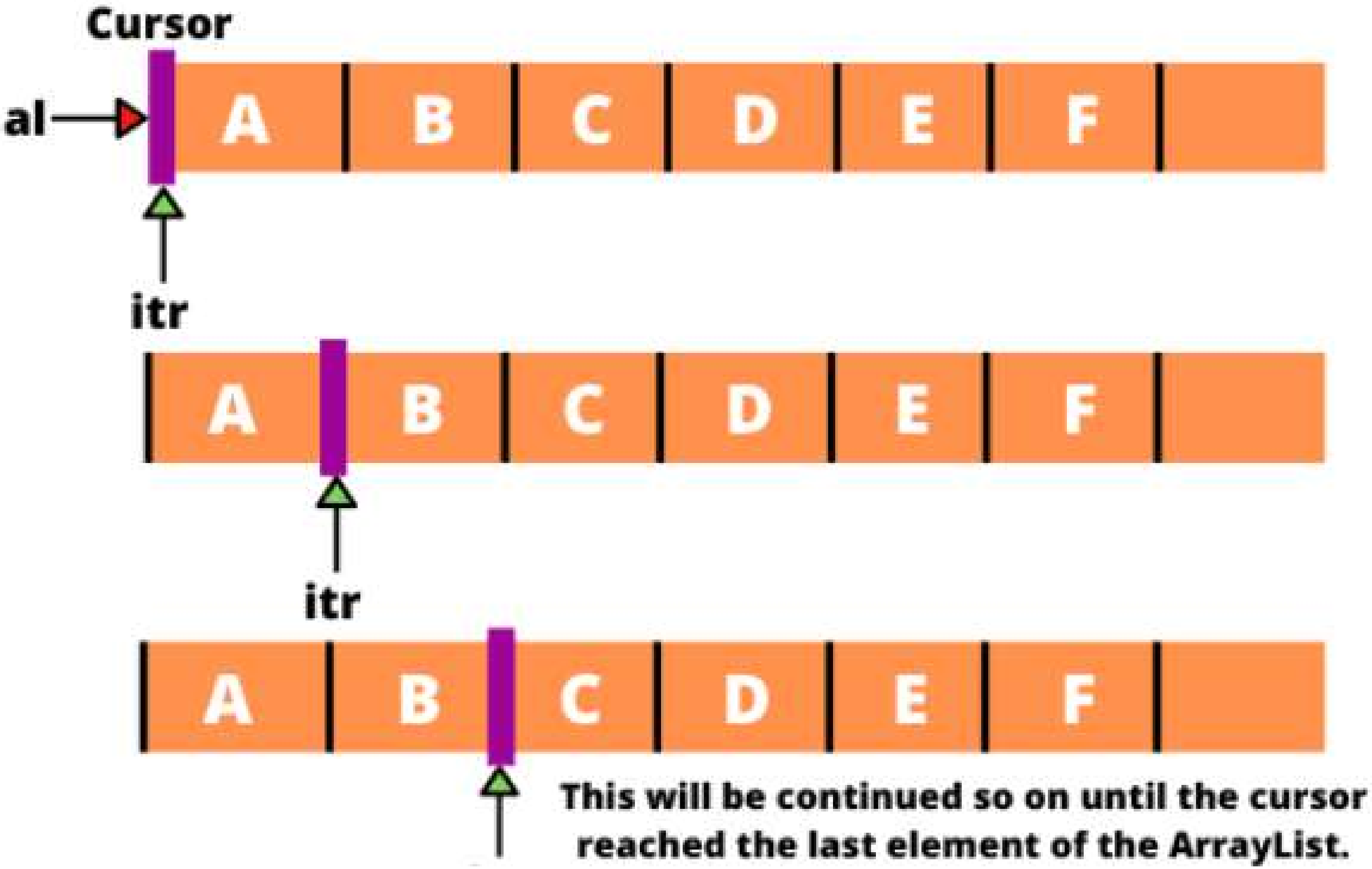
- You might want to do this if you know in advance that you will be storing many more items in the collection than it can currently hold. By increasing its capacity once, at the start, you can prevent several reallocations later. Because reallocations are costly in terms of time, preventing unnecessary ones improves performance.

- The signature for **ensureCapacity( )** is shown here:
  - void ensureCapacity(int *cap*)

- Here, *cap* specifies the new minimum capacity of the collection.

iterator

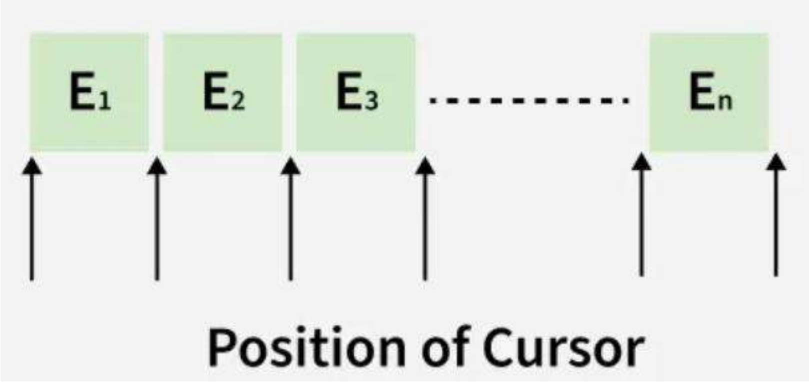
r

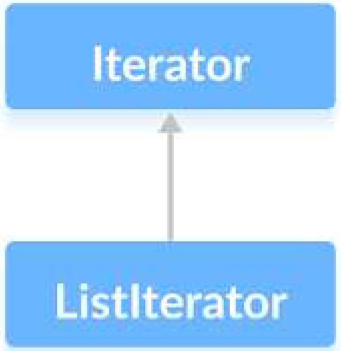


This will be continued so on until the cursor reached the last element of the ArrayList.

ListIterator

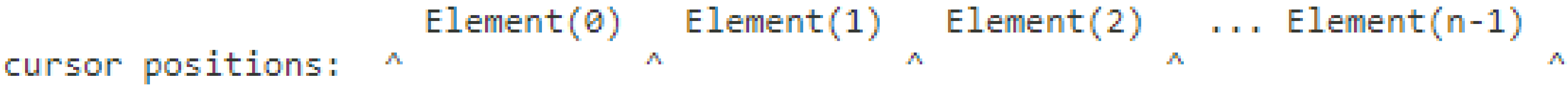
- ListIterator is a type of Java_cursor used to traverse all types of lists, such as ArrayList, LinkedList, Vector, and Stack.
- It was introduced in Java 1.2 and extends the Iterator interface.
  - It works only with list-implemented classes.
  - It supports bi-directional traversal (forward & backward).
  - Supports Create, Read, Update, Delete (CRUD) operations.





ListIterator

- There is **no current element** in ListIterator. Its cursor always lies between the previous and next elements. The **previous()** will return to the previous elements and the **next()** will return to the next element.
- An iterator for a list of length n has n+1 possible cursor positions, as illustrated by the carets (^) below:



```
                     Element(0)   Element(1)   Element(2)   ... Element(n-1)
cursor positions:  ^            ^            ^            ^                  ^
```

Forward and Backward Traversal using listIterator

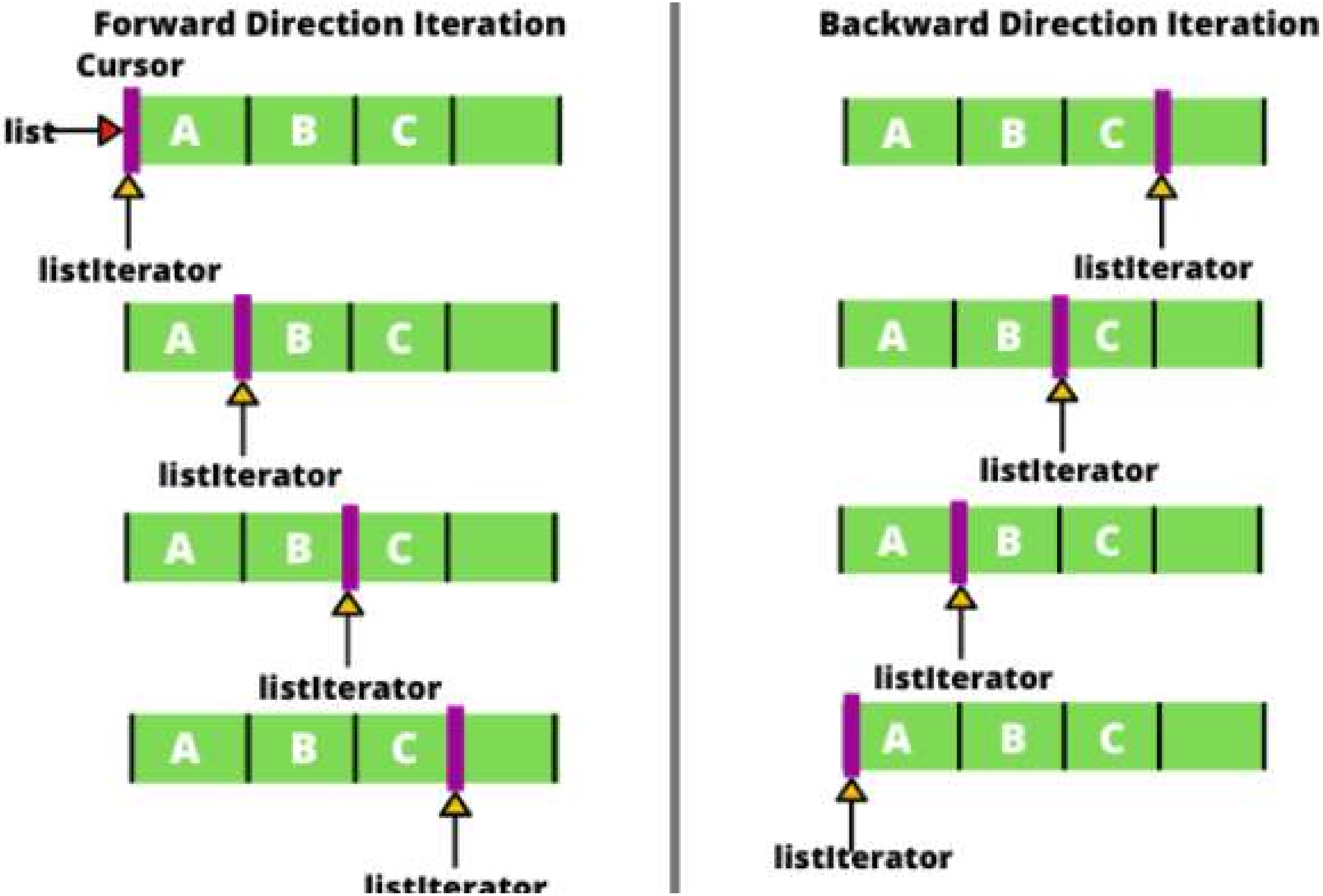
1. Forward direction iteration

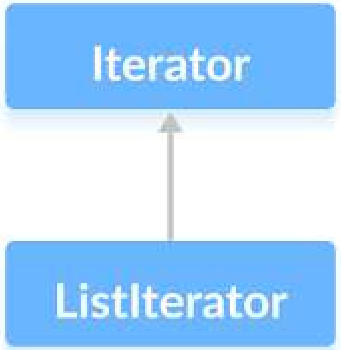
- **hasNext():** This method returns true when the list has more elements to traverse while traversing in the forward direction
- **next():** This method returns the next element of the list and advances the position of the cursor.
- **nextIndex():** This method returns the index of the element that would be returned on calling the *next()* method.

2. Backward direction iteration

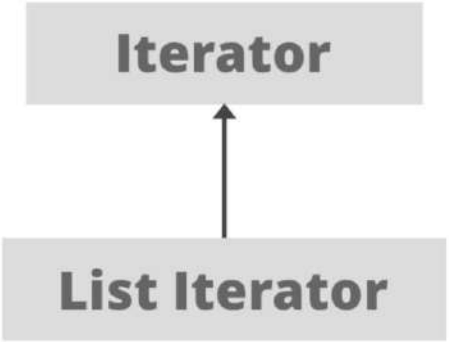
- **hasPrevious():** This method returns true when the list has more elements to traverse while traversing in the reverse direction
- **previous():** This method returns the previous element of the list and shifts the cursor one position backward.
- **previousIndex():** This method returns the index of the element that would be returned on calling the *previous()* method.

listiterator





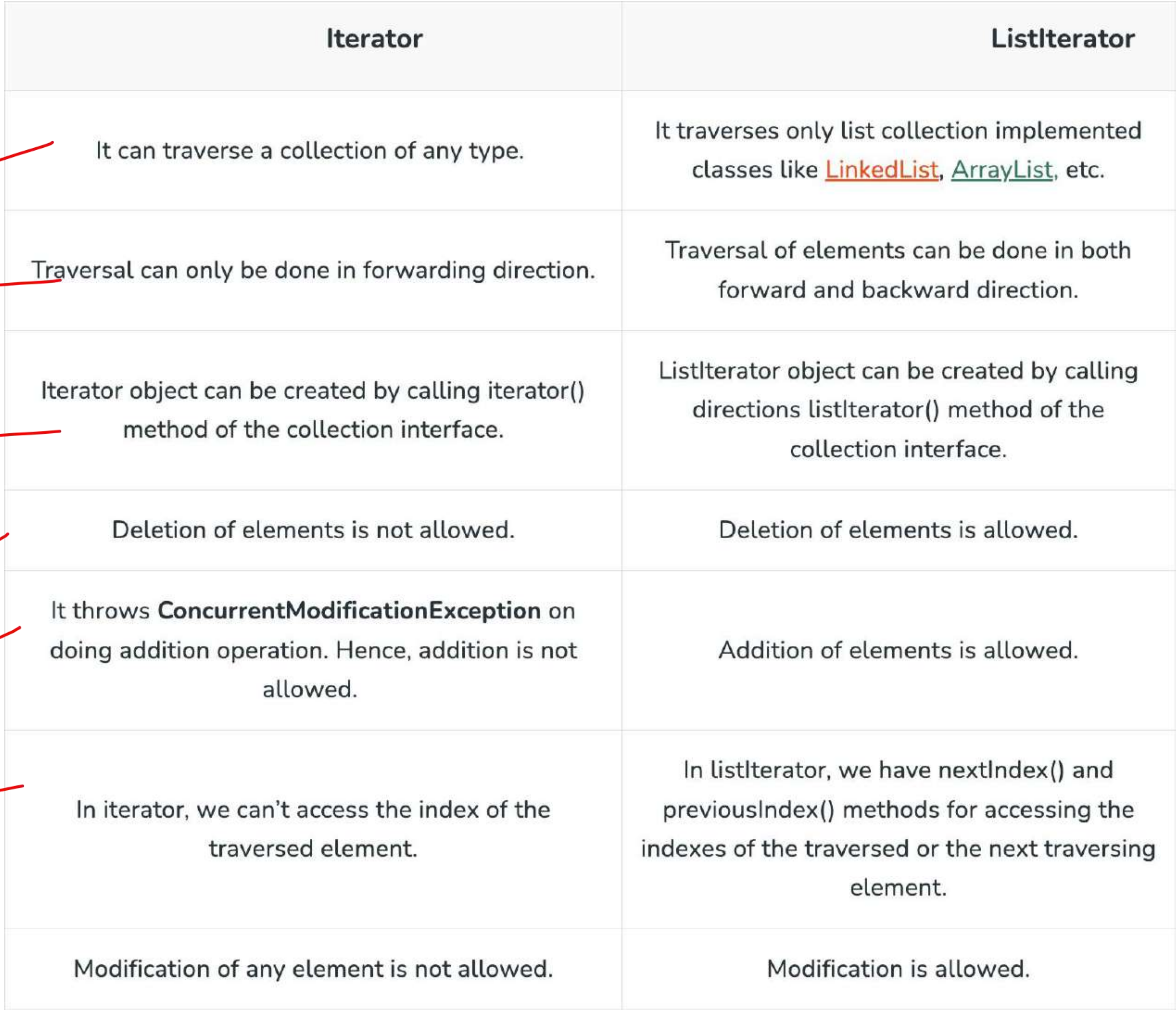
Differences between Iterator and ListIterator?



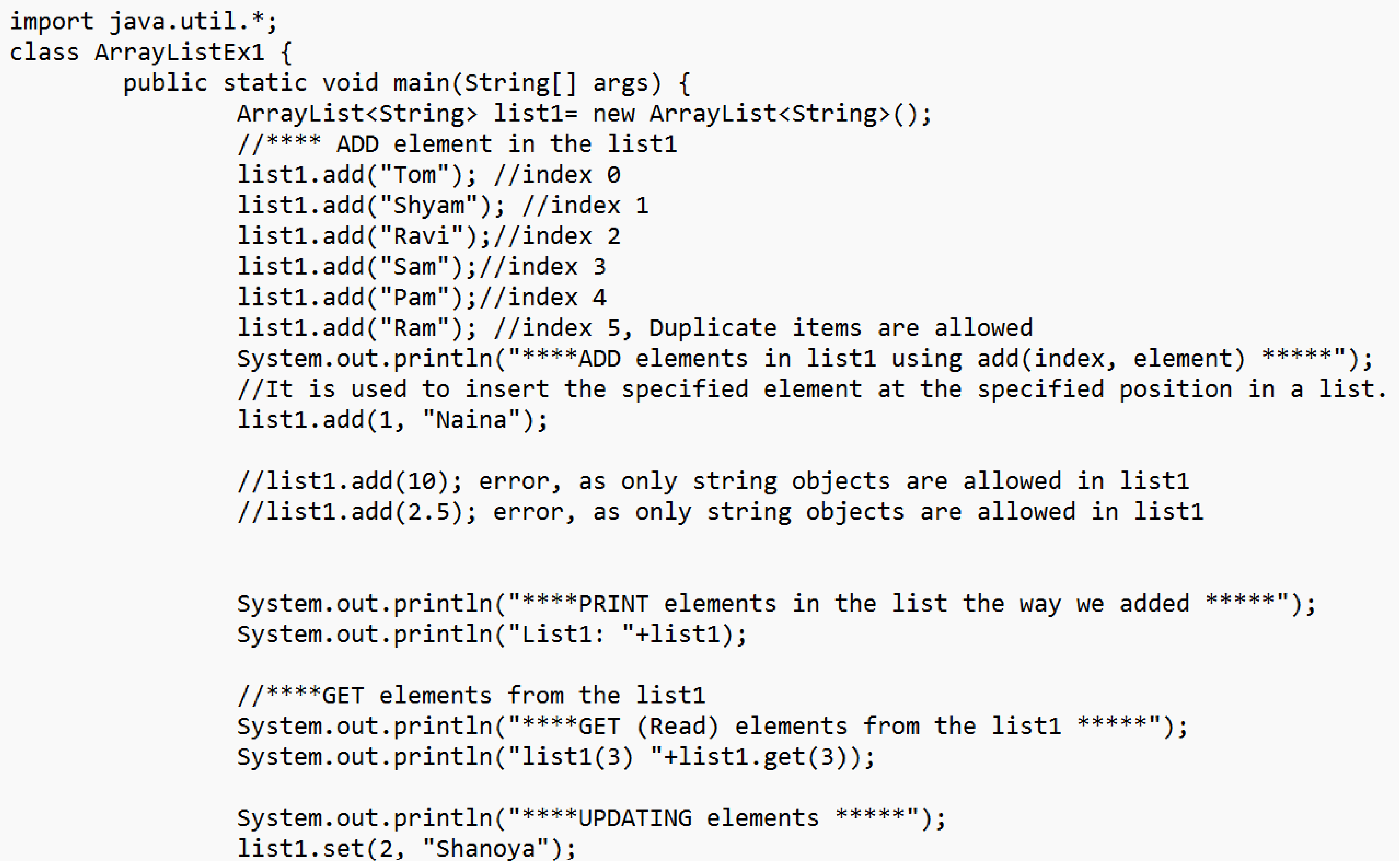
- Iterator is applicable for all Collection ımplementatıons(universal iterator). ListIterator is applicable for only List implementations.

- Iterator are allowed to iterate elements in only Forward direction. ListIterator is able to allow to iterate elements in both Forward direction and backward direction.

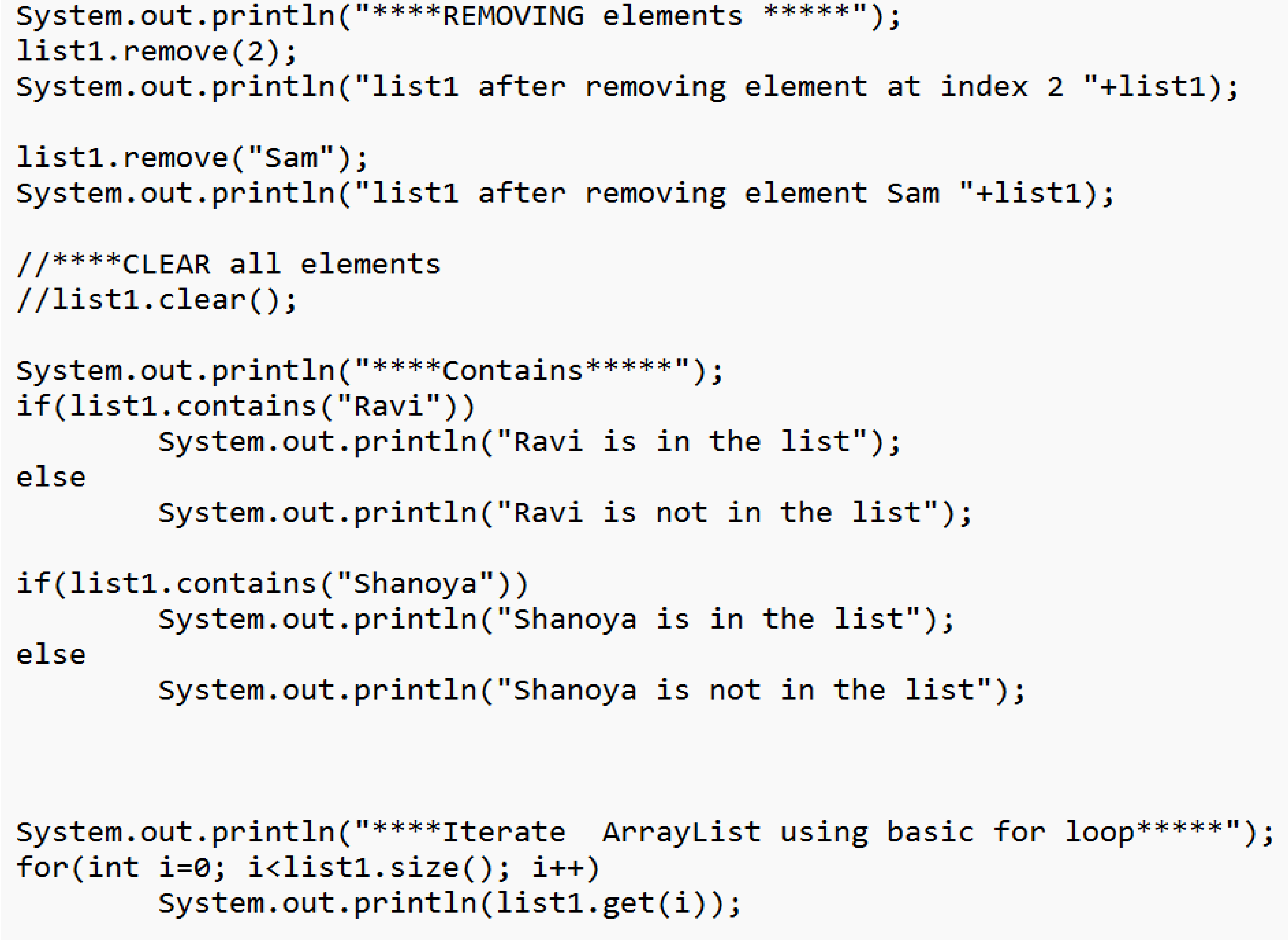
- Iterator is able to allow both read and remove operations while iterating elements. ListIterator supports all four CRUD (Create, Read, Update, Delete) operations.



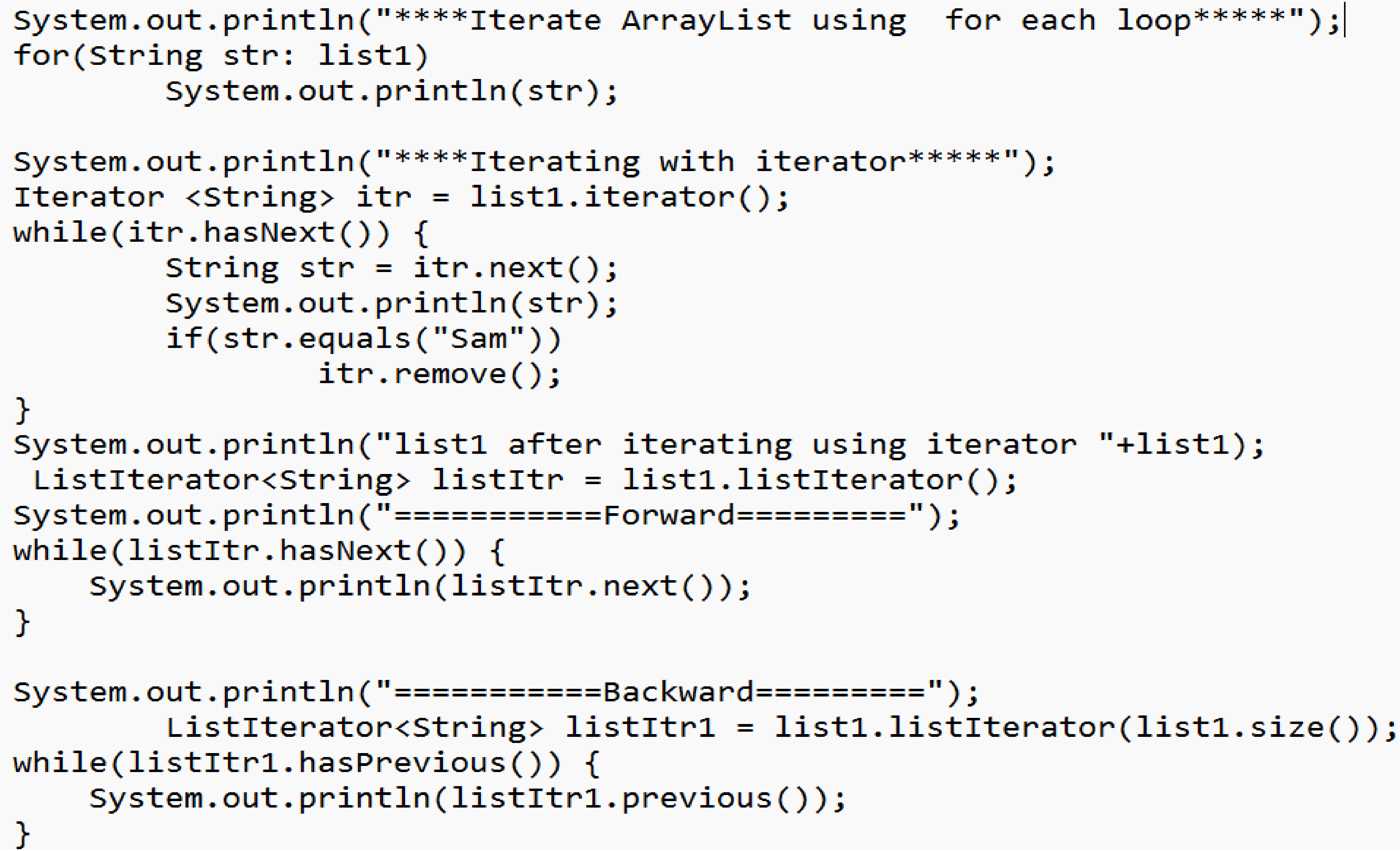
Iterator	ListIterator
It can traverse a collection of any type.	It traverses only list collection implemented classes like LinkedList, ArrayList, etc.
Traversal can only be done in forwarding direction.	Traversal of elements can be done in both forward and backward direction.
Iterator object can be created by calling iterator() method of the collection interface.	ListIterator object can be created by calling directions listIterator() method of the collection interface.
Deletion of elements is not allowed.	Deletion of elements is allowed.
It throws **ConcurrentModificationException** on doing addition operation. Hence, addition is not allowed.	Addition of elements is allowed.
In iterator, we can't access the index of the traversed element.	In listIterator, we have nextIndex() and previousIndex() methods for accessing the indexes of the traversed or the next traversing element.
Modification of any element is not allowed.	Modification is allowed.



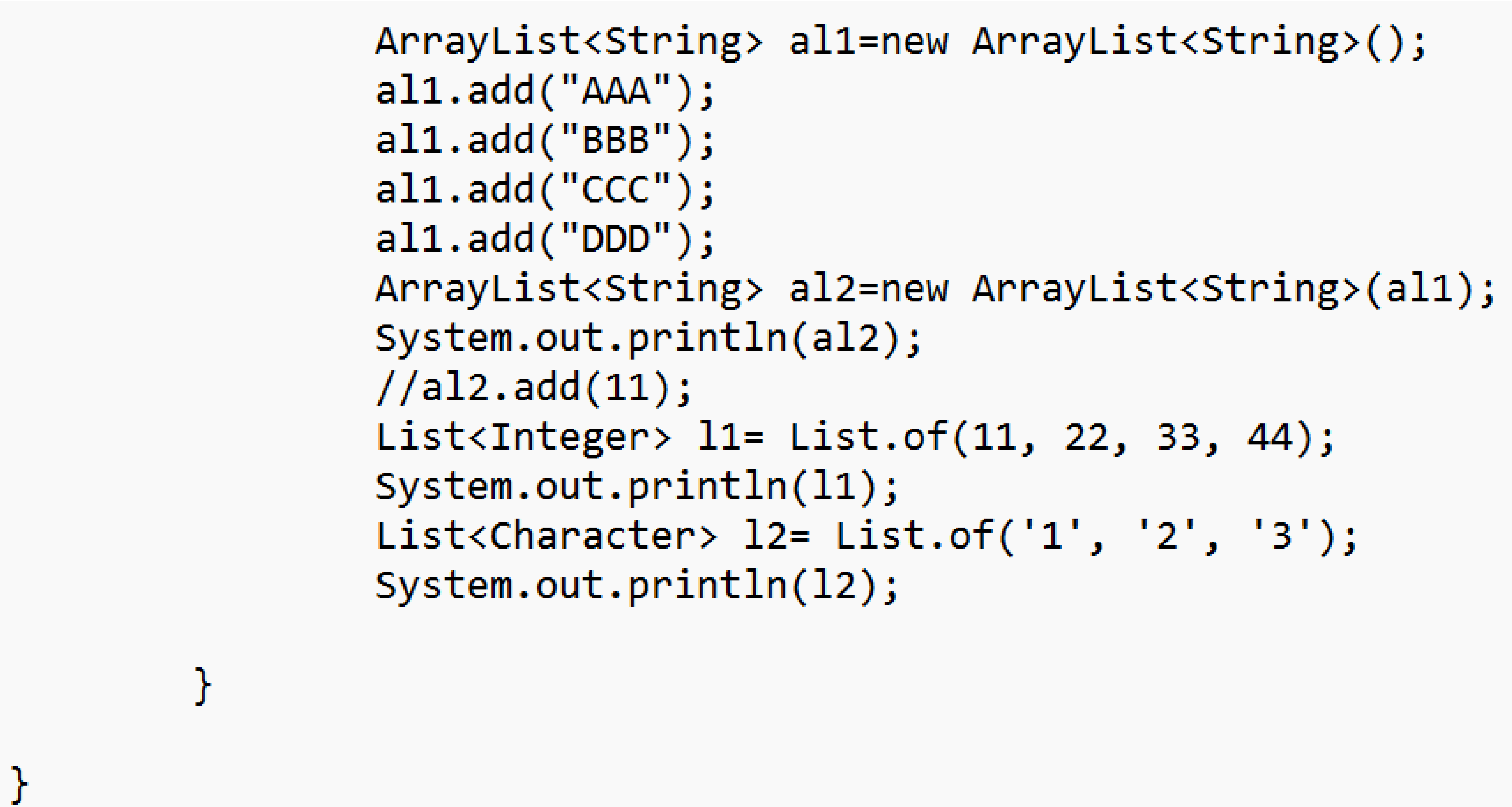
```
import java.util.*;
class ArrayListEx1 {
        public static void main(String[] args) {
                ArrayList<String> list1= new ArrayList<String>();
                //**** ADD element in the list1
                list1.add("Tom"); //index 0
                list1.add("Shyam"); //index 1
                list1.add("Ravi");//index 2
                list1.add("Sam");//index 3
                list1.add("Pam");//index 4
                list1.add("Ram"); //index 5, Duplicate items are allowed
                System.out.println("****ADD elements in list1 using add(index, element) *****");
                //It is used to insert the specified element at the specified position in a list.
                list1.add(1, "Naina");

                //list1.add(10); error, as only string objects are allowed in list1
                //list1.add(2.5); error, as only string objects are allowed in list1


                System.out.println("****PRINT elements in the list the way we added *****");
                System.out.println("List1: "+list1);

                //****GET elements from the list1
                System.out.println("****GET (Read) elements from the list1 *****");
                System.out.println("list1(3) "+list1.get(3));

                System.out.println("****UPDATING elements *****");
                list1.set(2, "Shanoya");
```



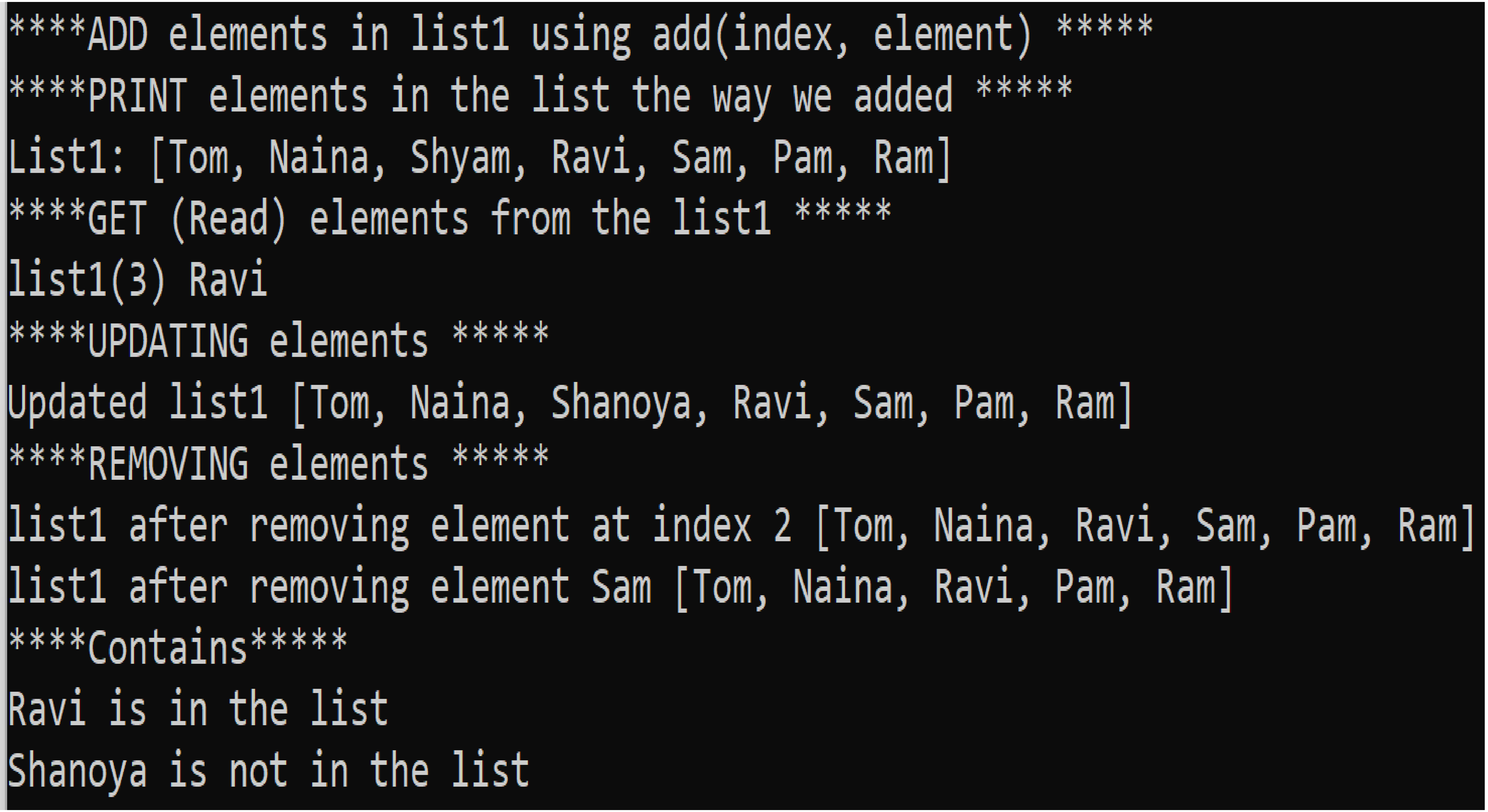
```
System.out.println("****REMOVING elements *****");
list1.remove(2);
System.out.println("list1 after removing element at index 2 "+list1);

list1.remove("Sam");
System.out.println("list1 after removing element Sam "+list1);

//****CLEAR all elements
//list1.clear();

System.out.println("****Contains*****");
if(list1.contains("Ravi"))
        System.out.println("Ravi is in the list");
else
        System.out.println("Ravi is not in the list");

if(list1.contains("Shanoya"))
        System.out.println("Shanoya is in the list");
else
        System.out.println("Shanoya is not in the list");



System.out.println("****Iterate  ArrayList using basic for loop*****");
for(int i=0; i<list1.size(); i++)
        System.out.println(list1.get(i));
```



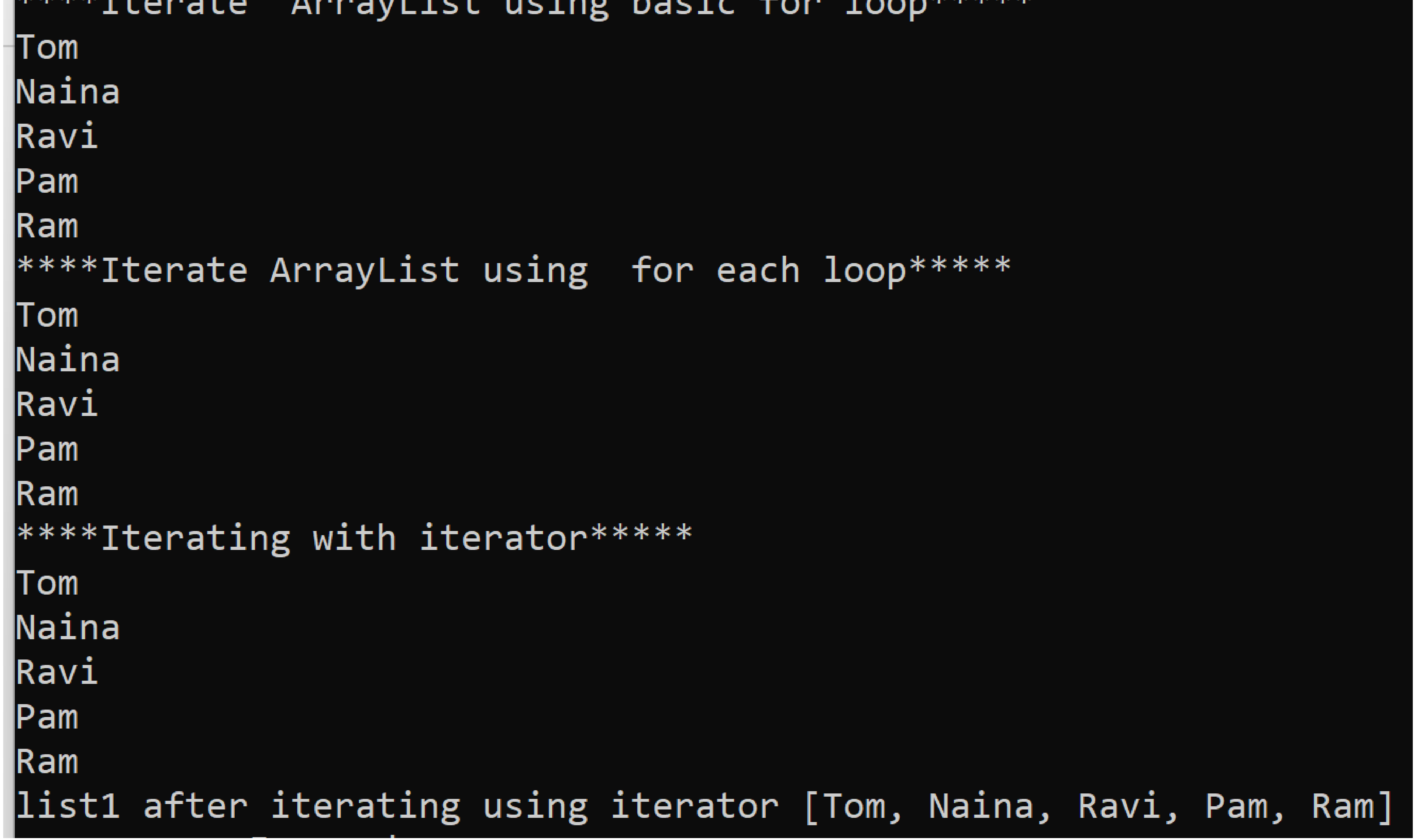
```
System.out.println("****Iterate ArrayList using  for each loop*****");
for(String str: list1)
        System.out.println(str);

System.out.println("****Iterating with iterator*****");
Iterator <String> itr = list1.iterator();
while(itr.hasNext()) {
        String str = itr.next();
        System.out.println(str);
        if(str.equals("Sam"))
                itr.remove();
}
System.out.println("list1 after iterating using iterator "+list1);
 ListIterator<String> listItr = list1.listIterator();
System.out.println("===========Forward=========");
while(listItr.hasNext()) {
    System.out.println(listItr.next());
}

System.out.println("===========Backward=========");
        ListIterator<String> listItr1 = list1.listIterator(list1.size());
while(listItr1.hasPrevious()) {
    System.out.println(listItr1.previous());
}
```



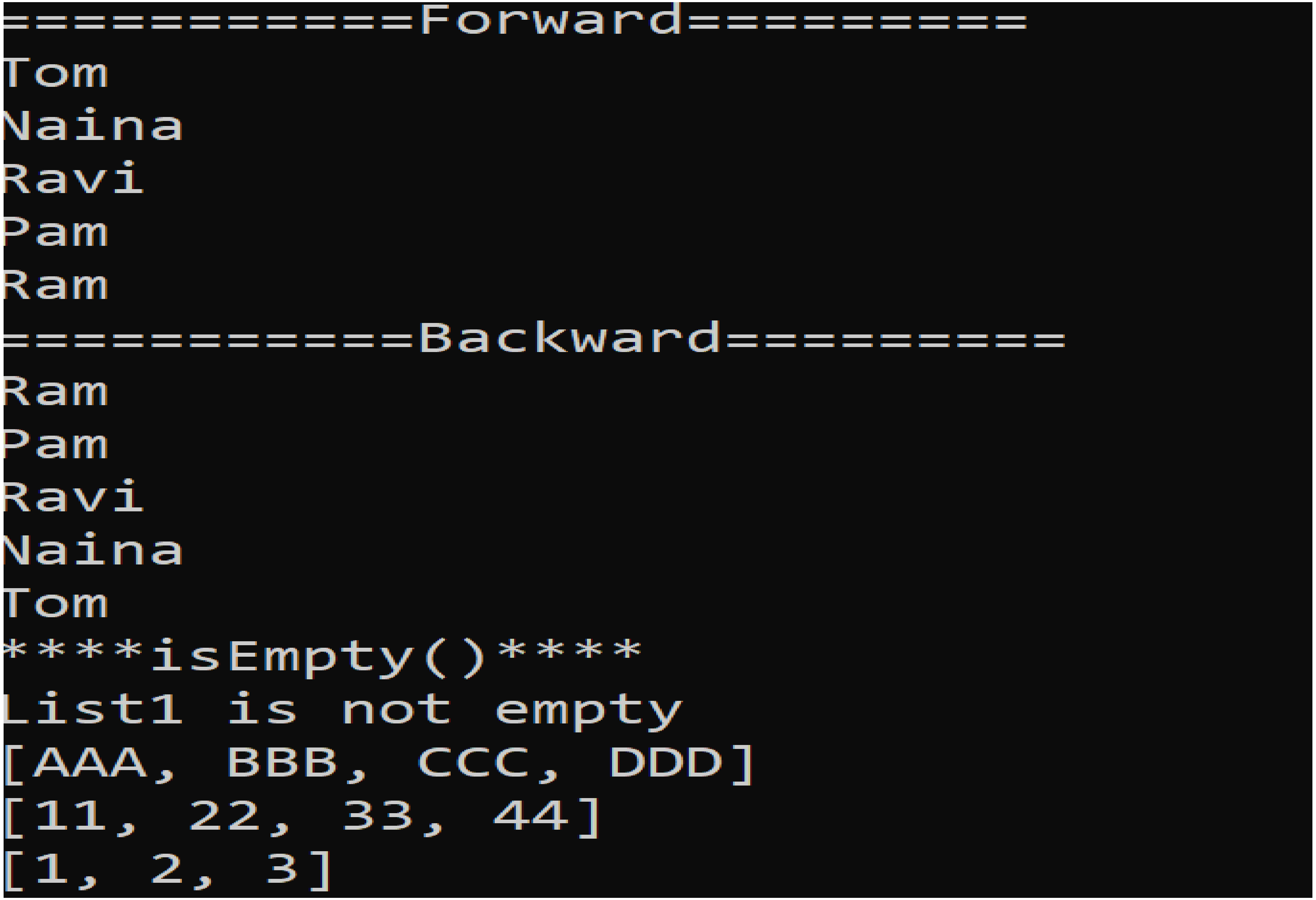
```
                ArrayList<String> al1=new ArrayList<String>();
                al1.add("AAA");
                al1.add("BBB");
                al1.add("CCC");
                al1.add("DDD");
                ArrayList<String> al2=new ArrayList<String>(al1);
                System.out.println(al2);
                //al2.add(11);
                List<Integer> l1= List.of(11, 22, 33, 44);
                System.out.println(l1);
                List<Character> l2= List.of('1', '2', '3');
                System.out.println(l2);

        }

}
```



```
****ADD elements in list1 using add(index, element) *****
****PRINT elements in the list the way we added *****
List1: [Tom, Naina, Shyam, Ravi, Sam, Pam, Ram]
****GET (Read) elements from the list1 *****
list1(3) Ravi
****UPDATING elements *****
Updated list1 [Tom, Naina, Shanoya, Ravi, Sam, Pam, Ram]
****REMOVING elements *****
list1 after removing element at index 2 [Tom, Naina, Ravi, Sam, Pam, Ram]
list1 after removing element Sam [Tom, Naina, Ravi, Pam, Ram]
****Contains*****
Ravi is in the list
Shanoya is not in the list
```



```
****Iterate  ArrayList using basic for loop*****
Tom
Naina
Ravi
Pam
Ram
****Iterate ArrayList using  for each loop*****
Tom
Naina
Ravi
Pam
Ram
****Iterating with iterator*****
Tom
Naina
Ravi
Pam
Ram
list1 after iterating using iterator [Tom, Naina, Ravi, Pam, Ram]
```



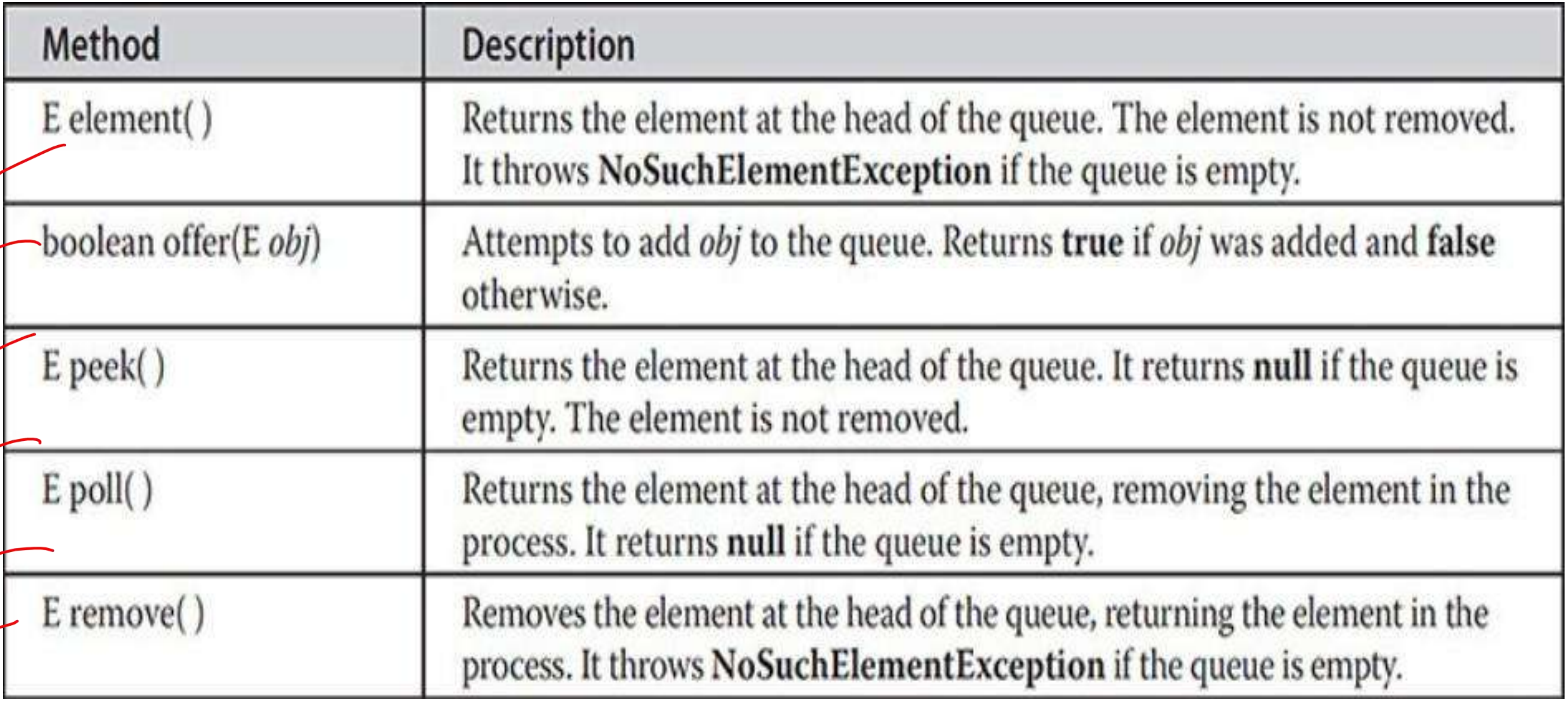
```
============Forward==========
Tom
Naina
Ravi
Pam
Ram
============Backward==========
Ram
Pam
Ravi
Naina
Tom
****isEmpty()****
List1 is not empty
[AAA, BBB, CCC, DDD]
[11, 22, 33, 44]
[1, 2, 3]
```

```
//ArrayList of objects

import java.util.ArrayList;
import java.util.List;
class User {
   private String name;
   private int age;
   public User(String name, int age) {
      this.name = name;
      this.age = age;
   }
   public String getName() {
      return name;
   }
   public int getAge() {
      return age;
   }
}
public class ArrayListObject {
   public static void main(String[] args) {
      List<User> users = new ArrayList<>();
      users.add(new User("Rajeev", 25));
      users.add(new User("John", 34));
      users.add(new User("Steve", 29));
      for(User u:users)
      {
         System.out.println("Name : " + u.getName() + ", Age : " +
         u.getAge());
      }


      });
   }
}
```

Queue interface

- The **Queue** interface extends **Collection** and declares the behavior of a queue, which is often a first-in, first-out list.
- **Queue** is a generic interface that has this declaration:
    - interface Queue<E>
  - Here, **E** specifies the type of objects that the queue will hold.

Queue interface Methods



Method	Description
E element( )	Returns the element at the head of the queue. The element is not removed. It throws **NoSuchElementException** if the queue is empty.
boolean offer(E *obj*)	Attempts to add *obj* to the queue. Returns **true** if *obj* was added and **false** otherwise.
E peek( )	Returns the element at the head of the queue. It returns **null** if the queue is empty. The element is not removed.
E poll( )	Returns the element at the head of the queue, removing the element in the process. It returns **null** if the queue is empty.
E remove( )	Removes the element at the head of the queue, returning the element in the process. It throws **NoSuchElementException** if the queue is empty.

Queue interface

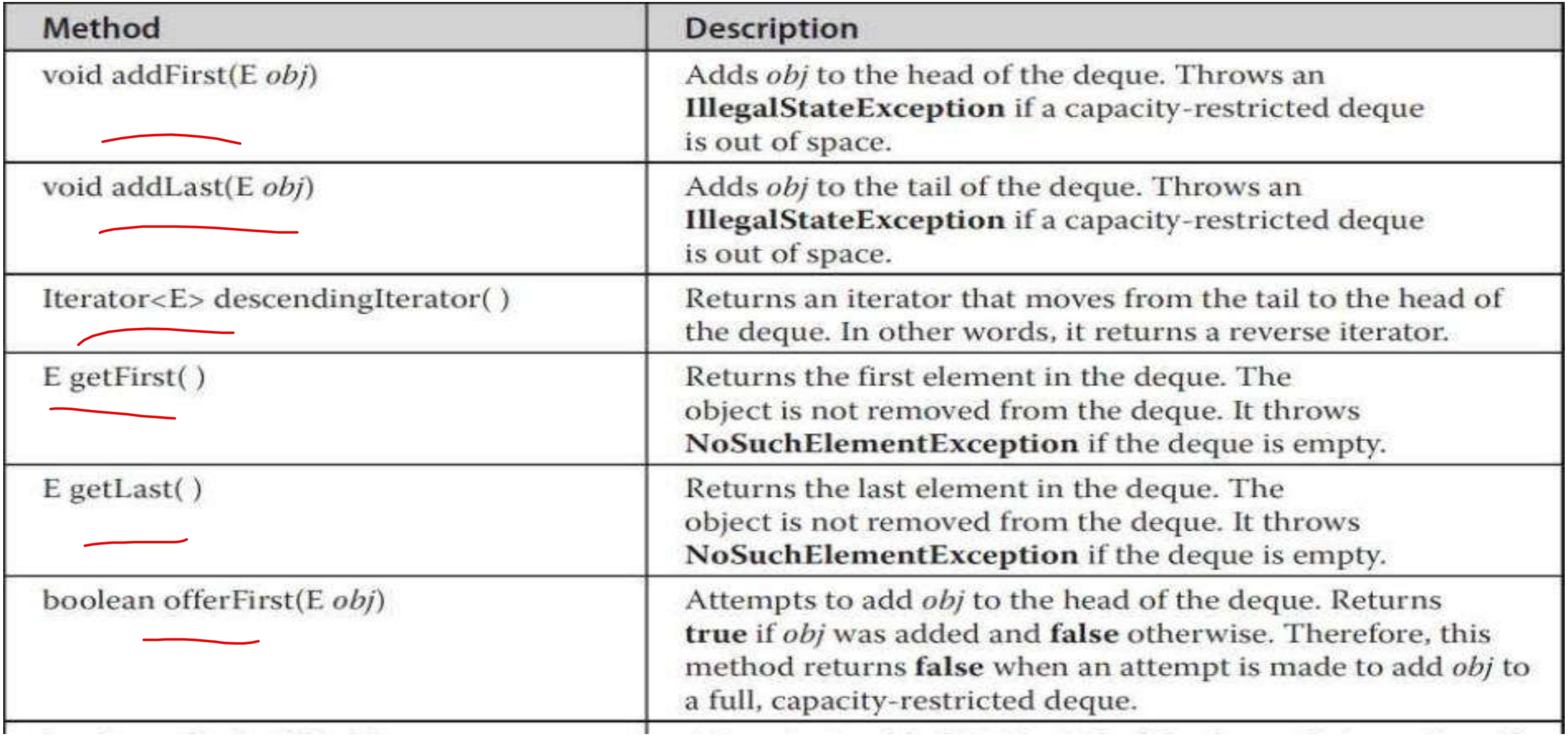
- **Elements can only be removed from head of the queue.**
  - There are two methods that obtain and remove elements: **poll( )** and **remove( )**.
    - The difference between them is that **poll()** returns **null** if the queue is empty, but **remove( )** throws an exception.
- There are two methods, **element( )** and **peek( )**, that obtain but don’t remove the element at the head of the queue.
  - They differ only in that **element( )** throws an exception if the queue is empty, but **peek( )** returns **null**.
- offer( ) only attempts to add an element to a queue. Because some queues have a fixed length and might be full, offer( ) can fail and will return false.
- add(Object o) add element to a queue, it throws IllegalStateException if not able to add element.

Deque Interface

- The **Deque** interface extends **Queue** and declares the behavior of a double- ended queue.
- Double-ended queues can function as standard, first-in, first-out queues or as last-in, first-out stacks.
- **Deque** is a generic interface that has this declaration: interface Deque<E>
- Here, **E** specifies the type of objects that the deque will hold.

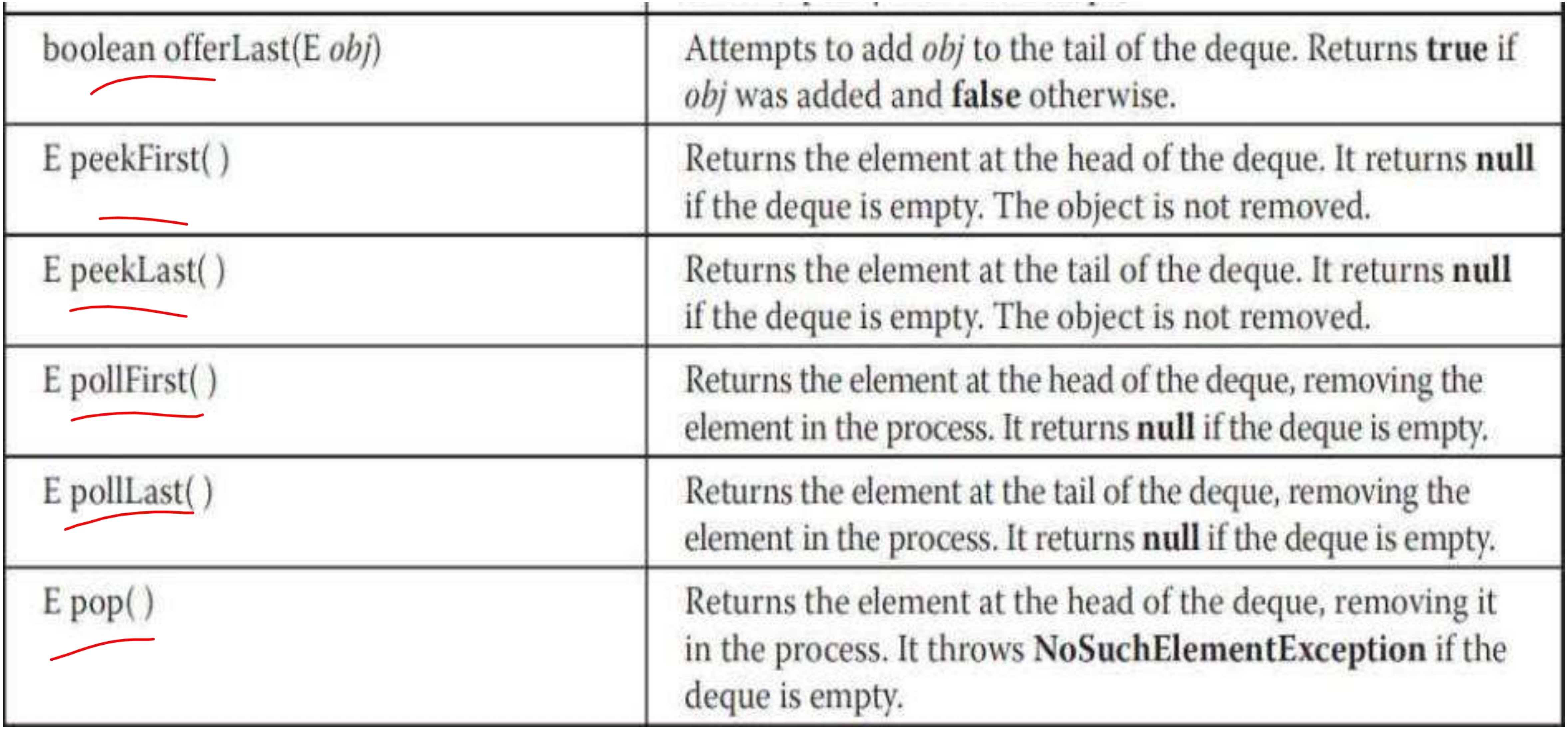
Deque interface methods

In addition to the methods that it inherits from **Queue**, **Deque** adds those methods summarized in the table:



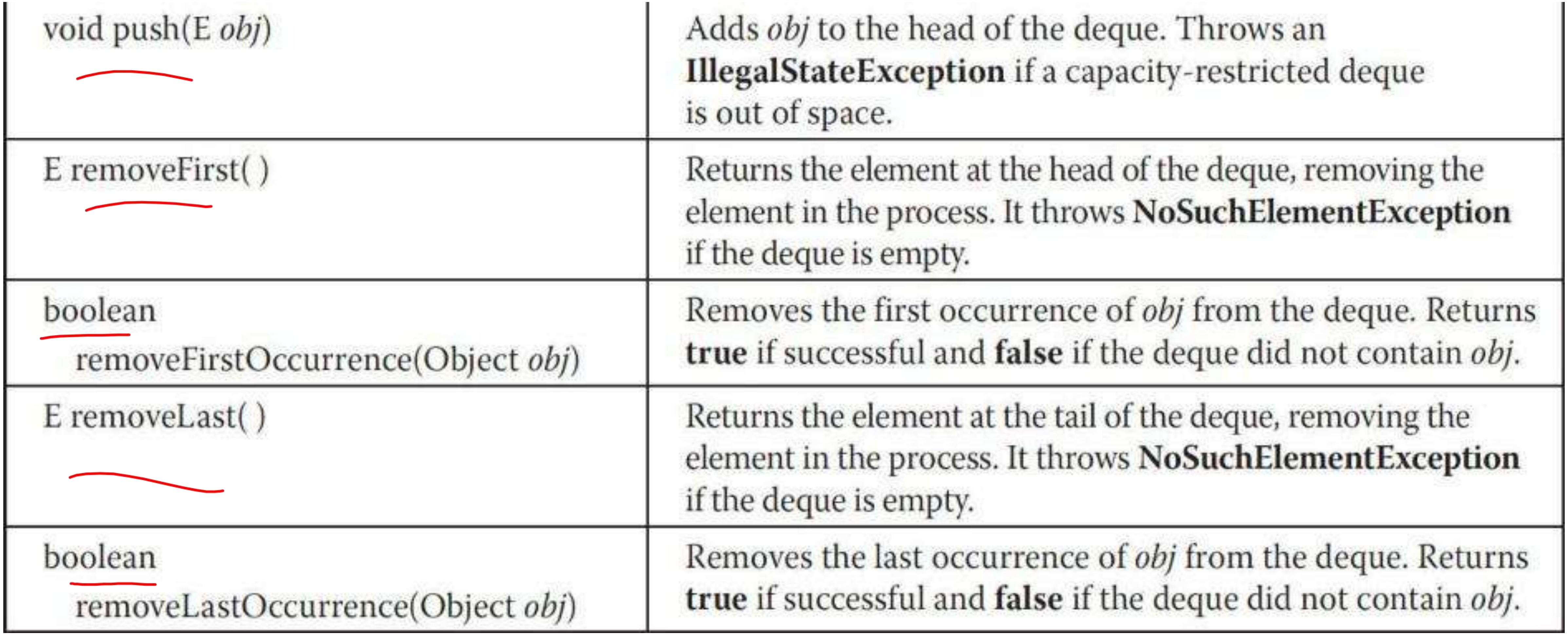
Method	Description
void addFirst(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
void addLast(E *obj*)	Adds *obj* to the tail of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the tail to the head of the deque. In other words, it returns a reverse iterator.
E getFirst( )	Returns the first element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty.
E getLast( )	Returns the last element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty.
boolean offerFirst(E *obj*)	Attempts to add *obj* to the head of the deque. Returns **true** if *obj* was added and **false** otherwise. Therefore, this method returns **false** when an attempt is made to add *obj* to a full, capacity-restricted deque.

Deque interface methods (cont'd)



boolean offerLast(E *obj*)	Attempts to add *obj* to the tail of the deque. Returns **true** if *obj* was added and **false** otherwise.
E peekFirst( )	Returns the element at the head of the deque. It returns **null** if the deque is empty. The object is not removed.
E peekLast( )	Returns the element at the tail of the deque. It returns **null** if the deque is empty. The object is not removed.
E pollFirst( )	Returns the element at the head of the deque, removing the element in the process. It returns **null** if the deque is empty.
E pollLast( )	Returns the element at the tail of the deque, removing the element in the process. It returns **null** if the deque is empty.
E pop( )	Returns the element at the head of the deque, removing it in the process. It throws **NoSuchElementException** if the deque is empty.

Deque interface methods (cont'd)

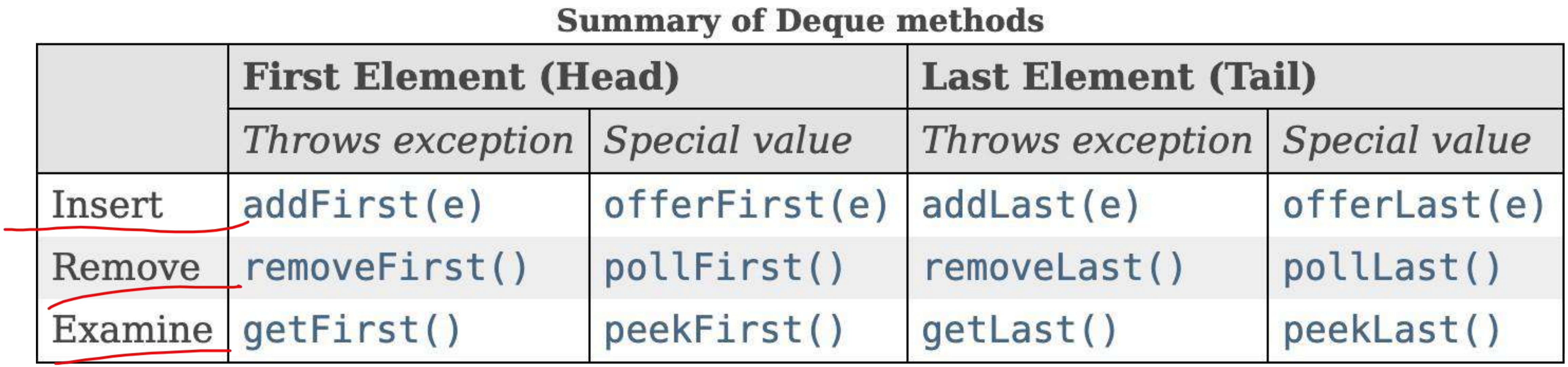


void push(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
E removeFirst( )	Returns the element at the head of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeFirstOccurrence(Object *obj*)	Removes the first occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.
E removeLast( )	Returns the element at the tail of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeLastOccurrence(Object *obj*)	Removes the last occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.

Deque interface methods (cont’d)

- **Deque** includes the methods **push( )** and **pop( )**. These methods enable a **Deque** to function as a stack.
- Also, notice the **descendingIterator**( ) method. It returns an iterator that returns elements in reverse order. In other words, it returns an iterator that moves from the end of the collection to the start.
- A **Deque** implementation can be *capacity-restricted*, which means that only a limited number of elements can be added to the deque. When this is the case, an attempt to add an element to the deque can fail.
- **Deque** allows you to handle such a failure in two ways.
  - First, methods such as **addFirst( )** and **addLast( )** throw an **IllegalStateException** if a capacity- restricted deque is full.
  - Second, methods such as **offerFirst( )** and **offerLast( )** return **false** if the element cannot be added.

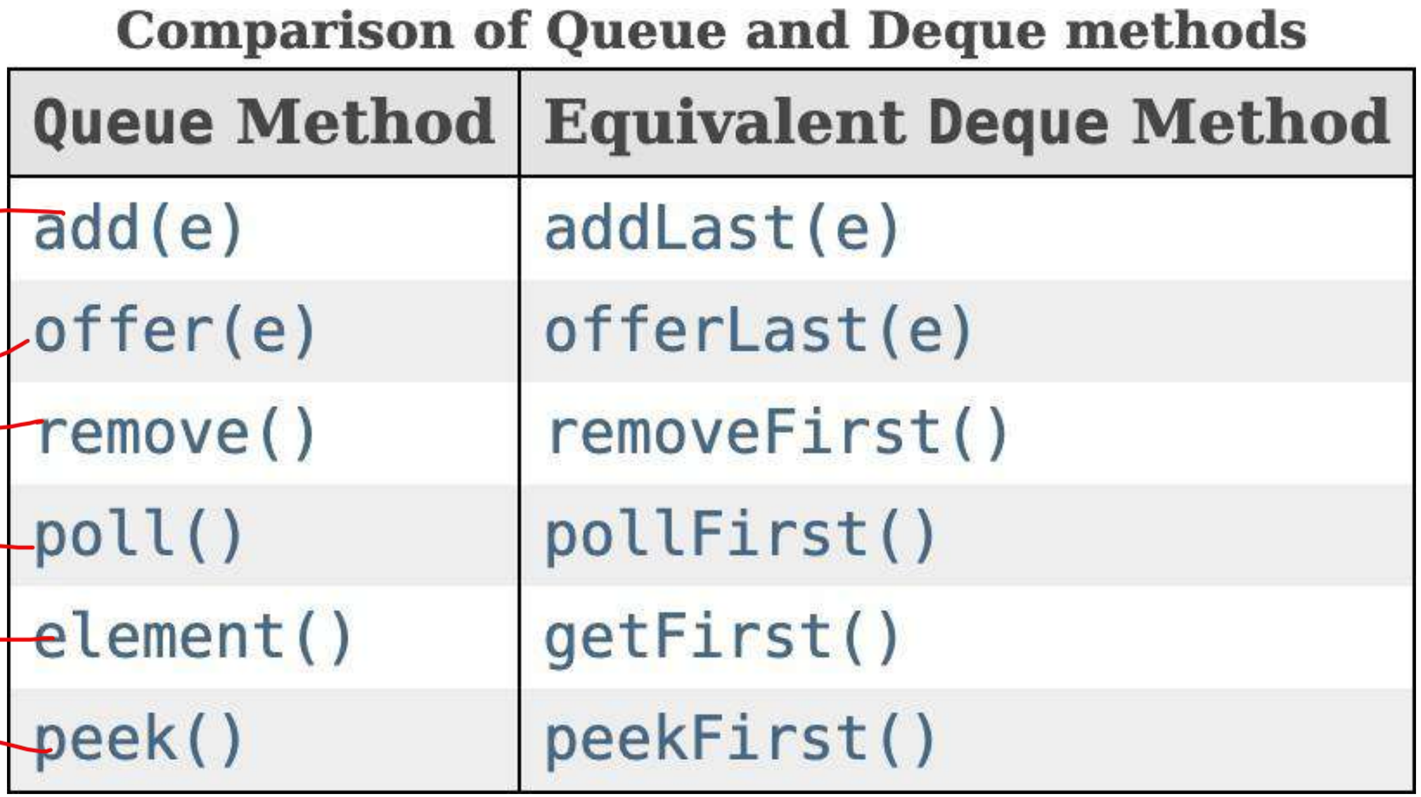
Deque methods



Summary of Deque methods

	First Element (Head)		**Last Element (Tail)**	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

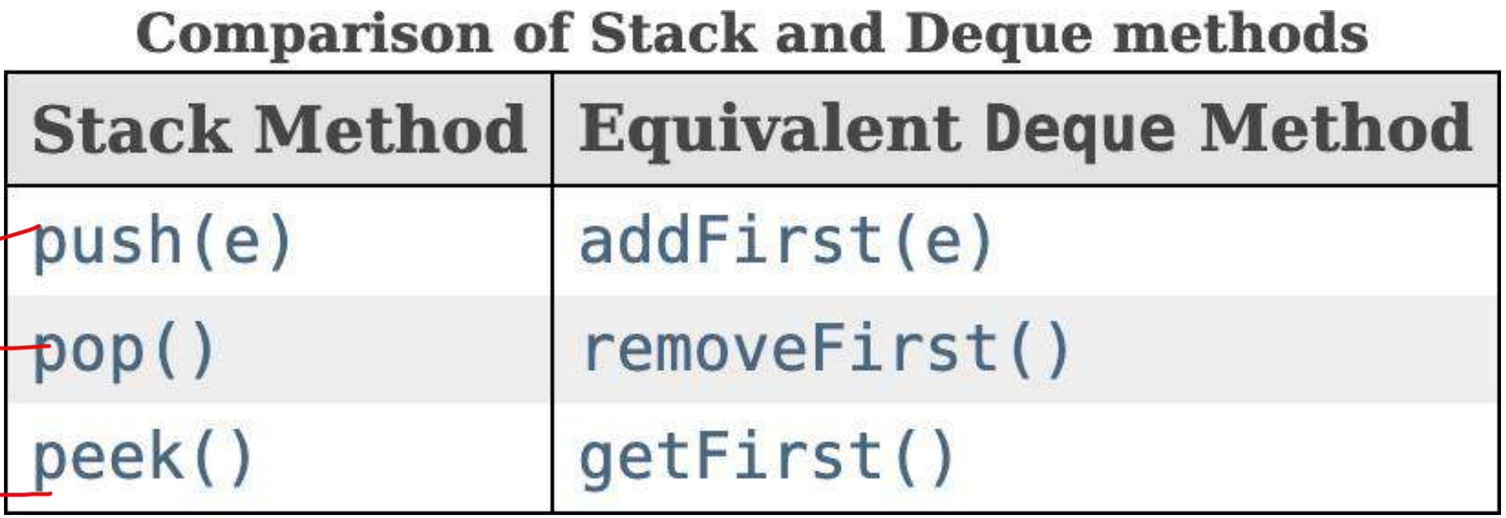
Deque methods



Comparison of Queue and Deque methods

Queue Method	Equivalent Deque Method
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

Duque methods

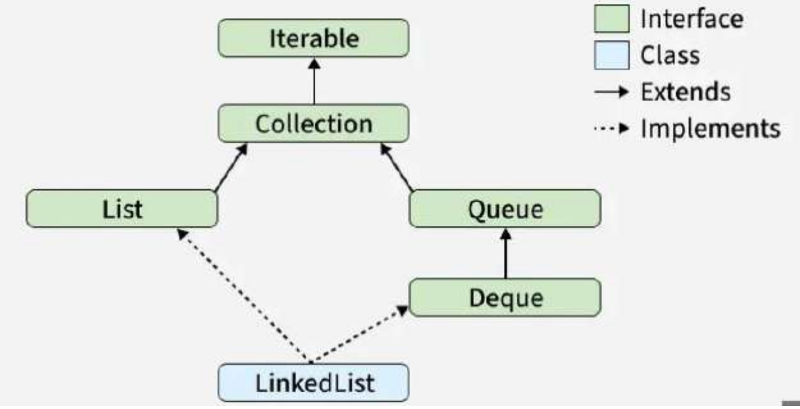


Comparison of Stack and Deque methods

Stack Method	Equivalent Deque Method
push(e)	addFirst(e)
pop()	removeFirst()
peek()	getFirst()

LinkedList

- **LinkedList** class extends **AbstractSequentialList** and implements the **List**, **Deque**, and **Queue** interfaces.
- Internally, the LinkedList is implemented using doubly linked list data structure



- It provides a linked-list data structure.
- can be used as a list, stack or queue.
- can contain duplicate elements.
- maintains insertion order.
- In the case of a doubly linked list, we can add or remove elements from both sides.
- **LinkedList** is a generic class that has this declaration:
    - class LinkedList<E>
- Here, **E** specifies the type of objects that the list will hold.
- Syntax: **LinkedList<E> list1 = new** LinkedList**<E>();**

where E is the data type of LinkedList

Constructors for Linked List

Constructor	**Description**
LinkedList()	used to construct an empty list.
LinkedList(Collection c)	builds a linked list that is initialized with the elements of the collection *c*.

Various operations on LinkedList

Because **LinkedList** implements the **Deque** interface, you have access to the methods defined by **Deque**.

- Adding elements:
  - To add elements to the start of a list, you can use **addFirst( )** or **offerFirst( )**.
  - To add elements to the end of the list, use **addLast( )** or **offerLast( )**.
- Examining/ Retrieving elements
  - To obtain the first element, you can use **getFirst( )** or **peekFirst( )**.
  - To obtain the last element, use **getLast( )** or **peekLast( )**.
- Removing Elements:
  - To remove the first element, use **removeFirst( )** or **pollFirst( )**.
  - To remove the last element, use **removeLast( )** or **pollLast( )**.
- Iterating over elements:
- toArray()
- Checking size:

Ex2: LinkedList

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

public static void main(String args[])
{
        LinkedList<String> ll1= new LinkedList <String>();
        ll1.add("AAA");
        ll1.add("BBB");
        ll1.add("CCC");
        ll1.add("DDD");
        System.out.println("ll1 "+ll1);
        ll1.addFirst("OOO");
        ll1.addLast("ZZZ");
        System.out.println("ll1 "+ll1);

        System.out.println("First "+ll1.getFirst());
        System.out.println("Last "+ll1.getLast());

        ll1.removeFirst();
        System.out.println("After Rem First "+ll1);
        ll1.removeLast();
        System.out.println("Rem Last "+ll1);

        LinkedList<String> ll2= new LinkedList <String>(ll1);
        System.out.println("ll2 “+ll2);
}}
```

Set interface

- A Set interface extends Collection interface
- It cannot contain duplicate elements. Therefore, the **add( )** method returns **false** if an attempt is made to add duplicate elements to a set.
- It models mathematical set abstraction.
- *Set interface contains only methods inherited from Collection.*
- **Set** is a generic interface that has this declaration:
        - interface Set<E>
    - Here, **E** specifies the type of objects that the set will hold.
- Two Set instances are equal if they contain the same elements.
-

Set interface

- It is not index based
- does not allow duplicate elements.
- Does not follow insertion order. **Note: LinkedHashSet will follow insertion order.**
- Does not follow Sorting order. **Note: SortedSet, NavigableSet and TreeSet are following Sorting order.**
- allow only one null value. **Note: SortedSet, NavigableSet and TreeSet does not allow even single null value.**

Set Methods

boolean	**add** (**E** e)	Adds the specified element to this set if it is not already present (optional operation).
boolean	**addAll** (**Collection** c)	Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	**clear**()	Removes all of the elements from this set (optional operation).
boolean	**contains** (**Object** o)	Returns trueif this set contains the specified element.
boolean	**containsAll** (**Collection** c)	Returns trueif this set contains all of the elements of the specified collection.
static<E> **Set**<E>	**copyOf** (**Collection** coll)	Returns an unmodifiable Set containing the elements of the given Collection.
boolean	**isEmpty**()	Returns trueif this set contains no elements.

Set Methods

boolean	**remove (Object o)**	Removes the specified element from this set if it is present
boolean	**removeAll (Collection c)**	Removes from this set all of its elements that are contained in the specified collection
boolean	**retainAll (Collection c)**	Retains only the elements in this set that are contained in the specified collection
int	**size()**	Returns the number of elements in this set (its cardinality).

HashSet class

- implements the **Set** interface, backed by a hash table (actually a `HashMap` instance).
- **HashSet** is a generic class that has this declaration:
    - class HashSet<E>
      Here, **E** specifies the type of objects that the set will hold.
- It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time.
- This class permits a single `null` element.
- **HashSet does not define any additional methods**

Created By Dr Ritu Jain

HashSet class

- It uses hashing technique to store elements. HashSet uses HashMap internally in Java.
- The advantage of hashing is that it offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets even for large sets.

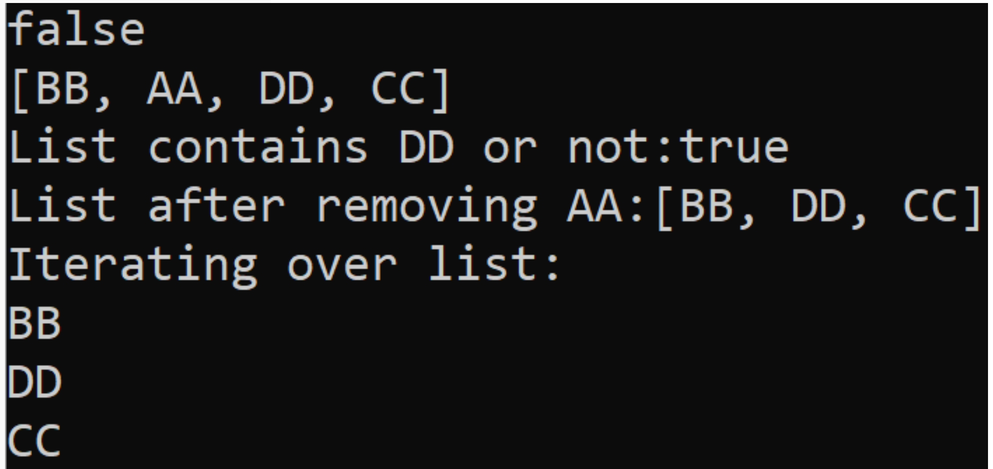
Constructors of HashSet class

Note: fillRatio is also known as load factor Default capacity:16;

Default load factor:0.75;   16*.75=12

Constructor	Description
HashSet( )	Constructs a new, empty set. Default initial capacity is 16 and load factor is 0.75. Eg: HashSet<E> hs = new HashSet<E>();
HashSet(Collection c)	Constructs a new set containing the elements in the specified collection.
HashSet(int *capacity*)	Constructs a new, empty set with specified initial capacity and default load factor (0.75).
HashSet(int *capacity*, float *fillRatio*)	initializes both the capacity and the fill ratio (also called *load factor*) of the hash set from its arguments. The fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded.

```java
import java.util.*;
public class HashSetEx1  {
    public static void main(String[] args)
    {
        HashSet<String> h = new HashSet<String>();
        // Adding elements into HashSet using add() method
        h.add("BB");
        h.add("AA");
        h.add("DD");
        h.add("CC");
        // Adding duplicate elements
        System.out.println(h.add("BB")); //return false
        System.out.println(h);
        System.out.println("Set contains DD or not:"+h.contains("DD"));
        // Removing items from HashSet using remove() method
        h.remove("AA");
        System.out.println("Set  after removing AA:"+ h);
        System.out.println("Iterating over Set :");

        Iterator<String> itr = h.iterator();
        while (itr.hasNext())
            System.out.println(itr.next());
    }
}
```



```
false
[BB, AA, DD, CC]
List contains DD or not:true
List after removing AA:[BB, DD, CC]
Iterating over list:
BB
DD
CC
```

Example 2 of HashSet storing objects of Integer class

```
import java.util.HashSet;
public class HashSetEx
{
public static void main(String[] args)
{
/*creating a HashSet */
HashSet<Integer> hashSet= new HashSet<Integer>();
/* add elements to HashSet  */
hashSet.add(5);
hashSet.add(2);
hashSet.add(3);
hashSet.add(6);
hashSet.add(13);
 System.out.println("Elements in HashSet are ");
/* iterate in hashSet */
for(int hash: hashSet){
System.out.println(hash);
}
}
}
```

Output:

```
Elements in HashSet are
2
3
5
6
13
```

SortedSet Interface

- The **SortedSet** interface extends **Set interface**
- declares the behavior of a **set sorted in ascending order.**
- **SortedSet** is a generic interface that has this declaration:
  - interface SortedSet<E>.   Here, E specifies the type of objects that the set will hold.

- **SortedSet** defines several methods that make set processing more convenient.

Methods of SortedSet Interface

E first()	Returns the first (lowest) element currently in this set.
SortedSet<E> headSet(E toElement)	Returns a view of the portion of this set whose elements are strictly less than toElement.
E last()	Returns the last (highest) element currently in this set.
SortedSet<E> subSet(E fromElement, E toElement)	Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
SortedSet<E> tailSet(E fromElement)	Returns a view of the portion of this set whose elements are greater than or equal to fromElement.

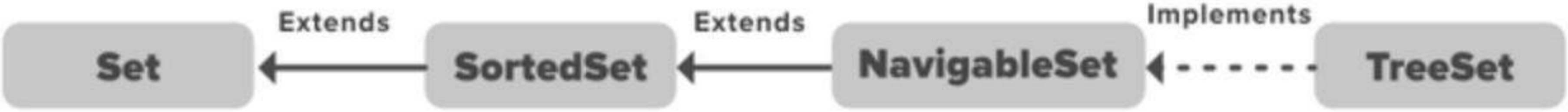
SortedSet

- To obtain the first object in the set, call **first( )**.
- To get the last element, use **last( )**.
- To obtain a subset of a sorted set, call **subSet( ) by** specifying the first and last object in the set.
- If you need the subset that starts with the first element in the set, use **headSet( )**.
- If you want the subset that ends the set, use **tailSet( )**.

TreeSet

- **TreeSet** implements the **NavigableSet** interface.
- It creates a collection that uses a tree for storage.
- Objects are stored in sorted, ascending order.
- Access and retrieval times are quite fast, which makes **TreeSet** an excellent choice when storing large amounts of sorted information that must be found quickly.
- Internally, a TreeSet is typically implemented using a self-balancing binary search tree, such as a red-black tree. This structure maintains the elements in sorted order, which allows for efficient retrieval of elements in sorted sequence. The elements are sorted based on their natural ordering (if they implement the Comparable interface) or by a specified comparator provided during construction.

- **TreeSet** is a generic class that has this declaration:
  - class TreeSet<E>



-

TreeSet Constructors

- **TreeSet** has the following constructors:
  - TreeSet( )
  - TreeSet(Collection *c*)

The first form constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements.

The second form builds a tree set that contains the elements of *c*.

Reason why SortedSet does not allow null values.

The SortedSet interface (and its primary implementation, TreeSet) in Java does not allow null values primarily because it relies on element comparison for maintaining sorted order, and a null value cannot be compared with any object.

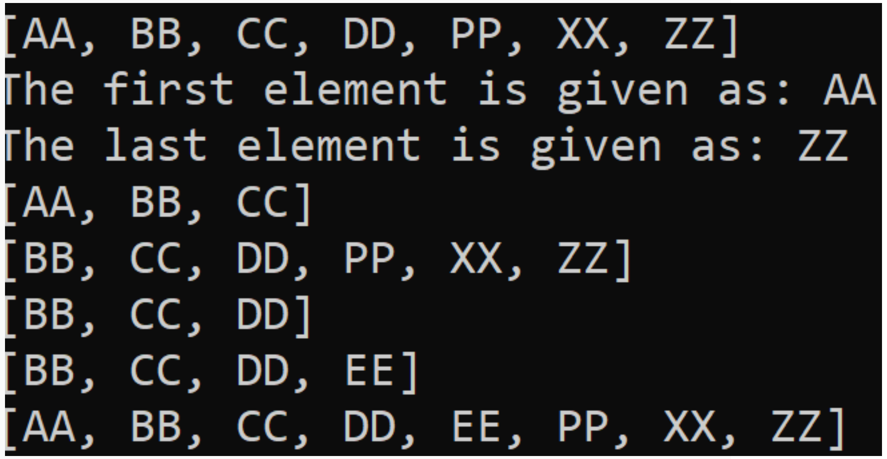
```
SortedSet <String>ss=new TreeSet<>();
ss.add(“AAA”);
ss.add(“BBB”);
ss.add(“CCC”);
ss.add(null);
 //when “BBB” is added “BBB”.compareTo(“AAA”) is called internally
// when “CCC” is added “CCC”.compareTo(“AAA”) and “CCC”.compareTo(“BBB”) is called
//null.compareTo(“AAA”); java.lang.NullPointerException is thrown
```

Reason why SortedSet does not allow null values.

- By default, the **Comparable interface** is used internally by TreeSet to sort the elements. Now in the Comparable Interface, the **compareTo() method** is used to compare one value with another to sort the elements.
- So because of this purpose, null has no value, that's why the compareTo() method cannot compare null with another value, giving a **NullPointerException**.

Example of TreeSet

```java
import java.util.SortedSet;
import java.util.TreeSet;
public class SortedSetEx2 {
   public static void main(String[] args) {
       SortedSet<String> set = new TreeSet<>();
            set.add("PP");
            set.add("DD");
            set.add("ZZ");
            set.add("AA");
            set.add("XX");
            set.add("CC");
            set.add("BB");
            System.out.println(set);
           System.out.println("The first element is given as: " + set.first());
           System.out.println("The last element is given as: " + set.last());

           //Returns a view of the portion of the given set whose elements are strictly less than the toElement.
           System.out.println(set.headSet("DD"));
           //Returns a view of the map whose keys are whose elements are greater than or equal to the parameter fromElement.
           System.out.println(set.tailSet("BB"));
          //return a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
           System.out.println(set.subSet("BB","PP"));

           SortedSet<String> s=set.subSet("BB","PP");
           s.add("EE");
           //s.add(null); //java.lang.NullPointerException
           //s.add("ff"); //java.lang.IllegalArgumentException: key out of range as last element is PP and ff is beyond it
           // s.add("YY"); //IllegalArgumentException as toElement is PP and YY is beyond PP
           System.out.println(s);
           System.out.println(set);
         }
}
```



```
[AA, BB, CC, DD, PP, XX, ZZ]
The first element is given as: AA
The last element is given as: ZZ
[AA, BB, CC]
[BB, CC, DD, PP, XX, ZZ]
[BB, CC, DD]
[BB, CC, DD, EE]
[AA, BB, CC, DD, EE, PP, XX, ZZ]
```

HashSet vs TreeSet in Java

- When it comes to discussing differences between Set the firstmost thing that comes into play is the insertion order and how elements will be processed.
- HashSet in java is a class implementing the Set interface, backed by a hash table which is actually a HashMap instance.
- This class permits the null element.
- The class also offers constant time performance for the basic operations like add, remove, contains, and size assuming the hash function disperses the elements properly among the buckets while TreeSet is an implementation of the SortedSet interface which as the name suggests uses the tree for storage purposes where here the ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided.

HashSet vs TreeSet in Java

1. Speed and internal implementation

- For operations like search, insert, and delete HashSet takes constant time for these operations on average.
- HashSet is faster than TreeSet.
- HashSet is Implemented using a hash table. TreeSet is implemented using a self-balancing binary search tree (Red-Black Tree). TreeSet is backed by TreeMap in Java.
- TreeSet takes O(Log n) for search, insert and delete which is higher than HashSet. But TreeSet keeps sorted data. Also, it supports operations like headset(). These operations are not supported not supported in HashSet.

2. Ordering

- Elements in HashSet are not ordered. TreeSet maintains objects in Sorted order defined by either Comparable or Comparator method in Java. TreeSet elements are sorted in ascending order by default. It offers several methods to deal with the ordered set like first(), last(), headSet(), tailSet(), etc.

HashSet vs TreeSet in Java

3. Null Object

HashSet allows null object. TreeSet doesn't allow null Object and throw NullPointerException, Why, because TreeSet uses compareTo() method to compare keys and compareTo() will throw java.lang.NullPointerException.

4. Comparison

HashSet uses the equals() method to compare two objects in Set and for detecting duplicates. TreeSet uses compareTo() method for same purpose. If equals() and compareTo() are not consistent, i.e. for two equal objects equals should return true while compareTo() should return zero, then it will break the contract of Set interface and will allow duplicates in Set implementations like TreeSet

when to prefer TreeSet over HashSet?

Sorted unique elements are required instead of unique elements. The sorted list given by TreeSet is always in ascending order.

TreeSet has a greater locality than HashSet.If two entries are nearby in the order, then TreeSet places them near each other in data structure and hence in memory, while HashSet spreads the entries all over memory regardless of the keys they are associated to.

TreeSet uses a **Red-Black tree algorithm** underneath to sort out the elements. When one needs to perform read/write operations frequently, then TreeSet is a good choice.

LinkedHashSet is another data structure that is between these two. It provides time complexities like HashSet and maintains the order of insertion (Note that this is not sorted order, but the order in which elements are inserted).

Differentiate between HashSet and Tree Set

Properties	HashSet	TreeSet
Implementation	It internally uses HashMap to store the elements.	Internally it uses TreeMap to store the elements.
Data Structure	The Data structure that HashSet uses is HashTable.	The Data Structure that TreeSet uses is a Red-Black Tree.
Time Complexity	To add or remove the element from HashSet, the time complexity is O(1).	For adding or removing the element from TreeSet, the time complexity is O(log(n)).
Null Values	Only one null element can be stored in the HashSet.	No null elements are allowed.
Comparison	The hashCode() or equals() method is used to compare the elements it uses.	The compare() and compareTo() method is used for comparison.
Sorted values	There is no guarantee that the elements will be stored in sorted order.	TreeSet maintains the Sorted order.
Performance	It is faster than TreeSet.	TreeSet is slower than HashSet.
Values to Store	HashSet can store Heterogenous values.	You can store only homogenous values in TreeSet.
Methods	In comparison to TreeSet, HashSet offers fewer methods.	TreeSet has more methods in comparison to HashSet.
Iteration order	To iterate HashSet elements, it uses the arbitrary method.	It has sorted values.

When to use HashSet and When to use TreeSet

You should use a **HashSet when the order of elements does not matter and you need fast performance** for basic operations (add, remove, contains). Use a **TreeSet when you need a collection of unique elements that are sorted** in a specific order.

When to use HashSet

- HashSet is ideal for scenarios where speed is the primary concern and maintaining a specific order is not necessary.
- **Fast Operations**: It offers average constant time complexity (O(1)) for add, remove, and contains operations, which is faster than TreeSet.
- **Unordered Data**: Elements are stored in an unpredictable, unordered way based on their hash codes. The order may change as the set grows or shrinks.
- **Allows one null element**: HashSet permits one null value to be added.
- **Example Use Case**: Storing a list of unique user IDs or checking the existence of an element frequently in a large dataset where order is irrelevant.

When to use TreeSet

TreeSet is suitable when you need to store unique, sorted elements and require navigation features.

- **Sorted Order**: Elements are automatically sorted in their natural ordering (e.g., alphabetically for strings, numerically for integers) or by a custom Comparator provided at creation.
- **Guaranteed Logarithmic Performance**: Operations like add, remove, and contains have a time complexity of O(log n). This is slower than HashSet but guaranteed, whereas HashSet is O(1) *on average*.
- **Navigation Methods**: It provides methods to navigate the set, such as first(), last(), headSet(), and tailSet(), due to its implementation of the NavigableSet interface.
- **Does not allow null**: Adding a null element will cause a NullPointerException because it uses the compareTo() method for ordering.
- **Example Use Case**: Maintaining a real-time leaderboard, managing a sorted list of timestamps, or performing range queries to find elements within a specific range.

Thanks!!!