JAVA COLLECTIONS

Agustin Gimenez
CIT 360

1.

INTRODUCTION

Collections & Collections Framework

COLLECTIONS

Collections are **objects** that group multiple elements into a single unit. They are used to **store**, **retrieve**, **manipulate**, and **communicate** data.

COLLECTIONS FRAMEWORK

A **framework** is an architecture that helps you represent and manipulate collections. These frameworks contain:

- **Interfaces:** abstract data types that represent collections and allowed them to be manipulated independently
- **Implementations:** reusable data structures
- **Algorithms:** methods that perform useful computations on objects that implement collection interfaces

BENEFITS OF COLLECTIONS FRAMEWORK

- **Reduces programming effort:** No need to write adapter objects or conversion code to connect APIs
- **Increases speed and quality:** High-performance and high-quality implementations of useful data structures and algorithms
- **Allows interoperability among unrelated APIs:** Different APIs can interoperate and pass collections between them without any problem
- **Reduces effort to learn and to use new APIs:** No more using sub-APIs on each API to manipulate collections
- **Reduces effort to design new APIs:** Standard collections interfaces can be used when creating a new API
- **Fosters software reuse:** Reusable data structures

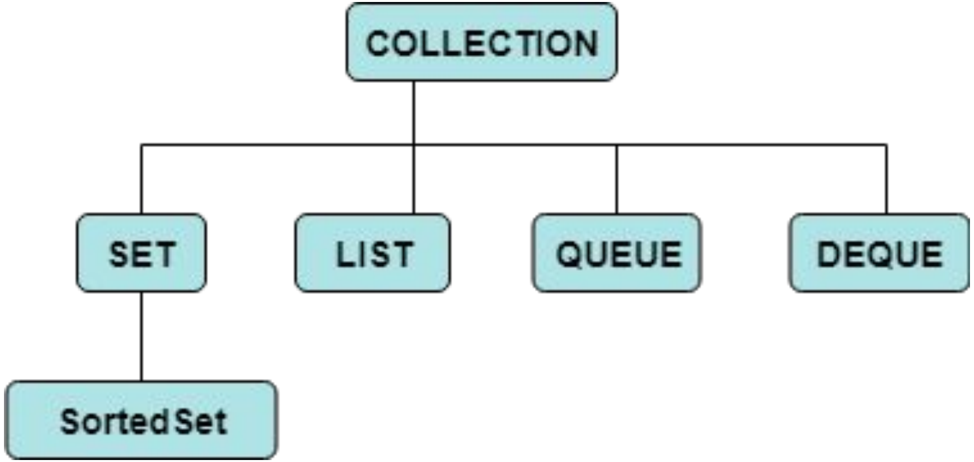
2.

INTERFACES

Different types of Collections

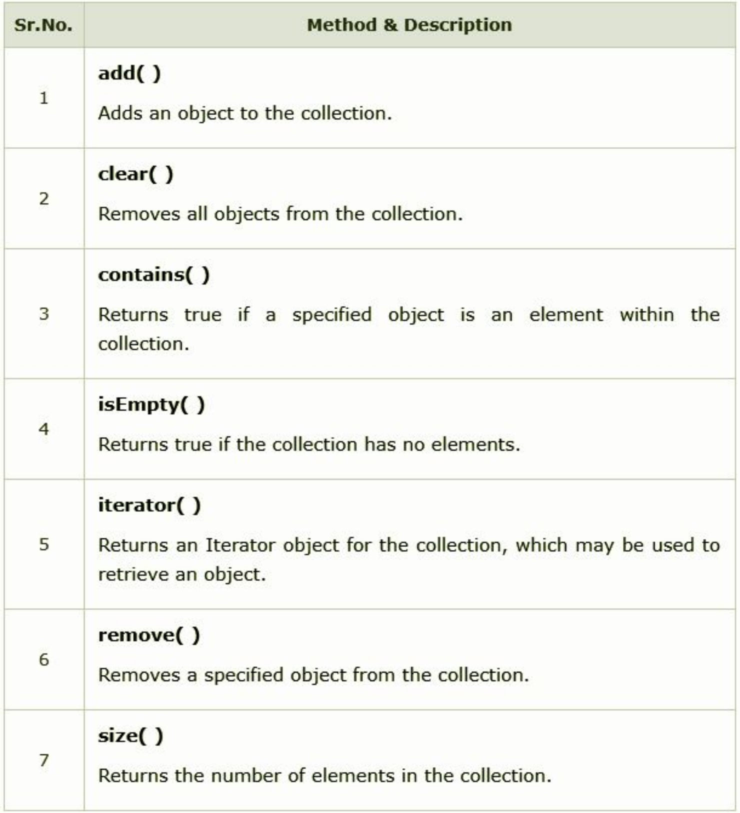
COLLECTION INTERFACE

The **Collection** interface encapsulate different types of collections that form a hierarchy. Here are the most basic ones:



SET INTERFACE

The **Set** interface is a collection that can’t have duplicated elements. It inherits methods from **Collection** and adds a restriction that duplicate elements are prohibited. Some of the methods that can be used are shown in the following table.



Sr.No.	Method & Description
1	**add( )** Adds an object to the collection.
2	**clear( )** Removes all objects from the collection.
3	**contains( )** Returns true if a specified object is an element within the collection.
4	**isEmpty( )** Returns true if the collection has no elements.
5	**iterator( )** Returns an Iterator object for the collection, which may be used to retrieve an object.
6	**remove( )** Removes a specified object from the collection.
7	**size( )** Returns the number of elements in the collection.

SET INTERFACE – Examples

Set can be implemented in different classes, such as **HashSet**, **TreeSet**, and **LinkedHashSet**.

EXAMPLE

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

  public static void main(String args[]) {
      int count[] = {34, 22,10,60,30,22};
      Set<Integer> set = new HashSet<Integer>();
      try {
         for(int i = 0; i < 5; i++) {
            set.add(count[i]);
         }
         System.out.println(set);

         TreeSet sortedSet = new TreeSet<Integer>(set);
         System.out.println("The sorted list is:");
         System.out.println(sortedSet);

         System.out.println("The First element of the set is: "+ (Integer)sortedSet.first());
         System.out.println("The last element of the set is: "+ (Integer)sortedSet.last());
      }
      catch(Exception e) {}
   }
}
```

OUTPUT

```
[34, 22, 10, 60, 30]
The sorted list is:
[10, 22, 30, 34, 60]
The First element of the set is: 10
The last element of the set is: 60
```

LIST INTERFACE

The **List** interface stores a sequence of elements. Here are some of its characteristics:

- Elements can be inserted or accessed by their position on the list
- The list can contain duplicates
- This interface defines its own methods in addition to those inherited from Collections interface

LIST INTERFACE – Examples

List can be implemented in different classes, such as **ArrayList**, or **LinkedList**.

EXAMPLE

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

   public static void main(String[] args) {
      List a1 = new ArrayList();
      a1.add("Zara");
      a1.add("Mahnaz");
      a1.add("Ayan");
      System.out.println(" ArrayList Elements");
      System.out.print("\t" + a1);

      List l1 = new LinkedList();
      l1.add("Zara");
      l1.add("Mahnaz");
      l1.add("Ayan");
      System.out.println();
      System.out.println(" LinkedList Elements");
      System.out.print("\t" + l1);
   }
}
```

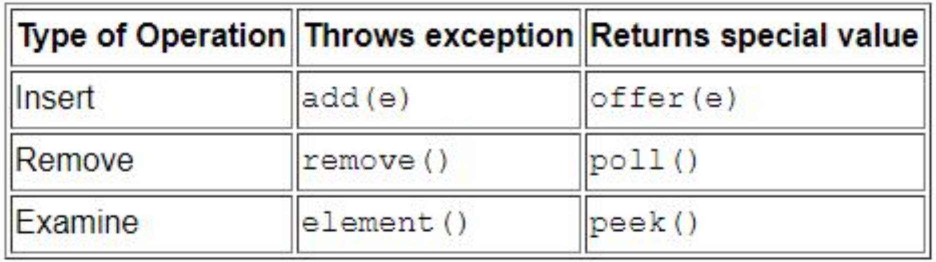
OUTPUT

```
ArrayList Elements
        [Zara, Mahnaz, Ayan]
 LinkedList Elements
        [Zara, Mahnaz, Ayan]
```

QUEUE INTERFACE

The **Queue** interface is for holding elements prior to processing. It also provides its own methods, especially **insertion**, **removal** and **inspection** operations.

Each method throws an exception if the operation fails and returns a special value, illustrated in the following table:



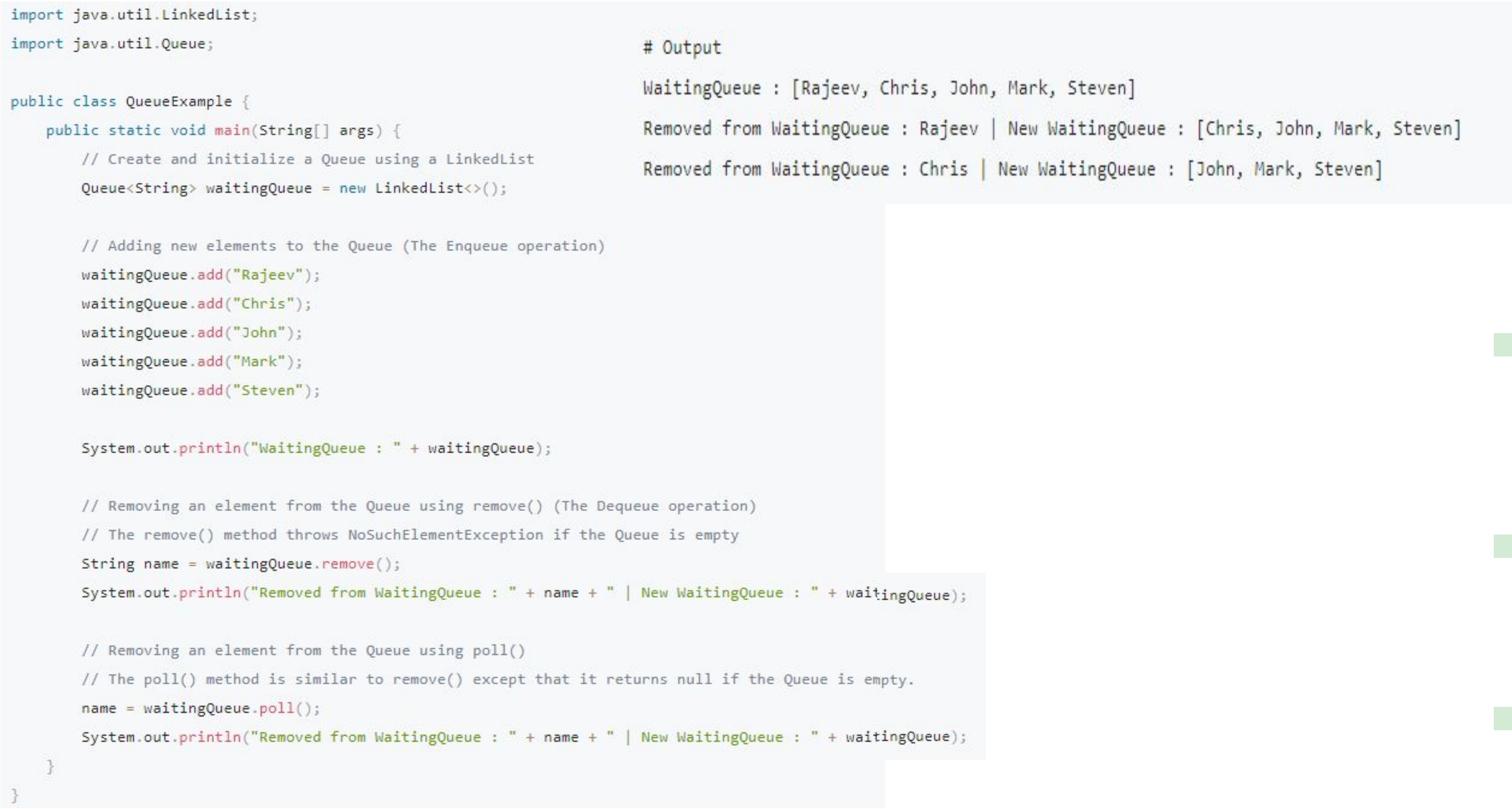
Type of Operation	Throws exception	Returns special value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

- The **add** method, which **Queue** inherits from **Collection**, inserts an element unless it would violate the queue capacity restrictions
- The **remove** and **poll** methods both remove and return the head of the queue.
- The **element** and **peek** methods return, but do not remove, the head of the queue.

QUEUE INTERFACE – Examples

EXAMPLE

OUTPUT

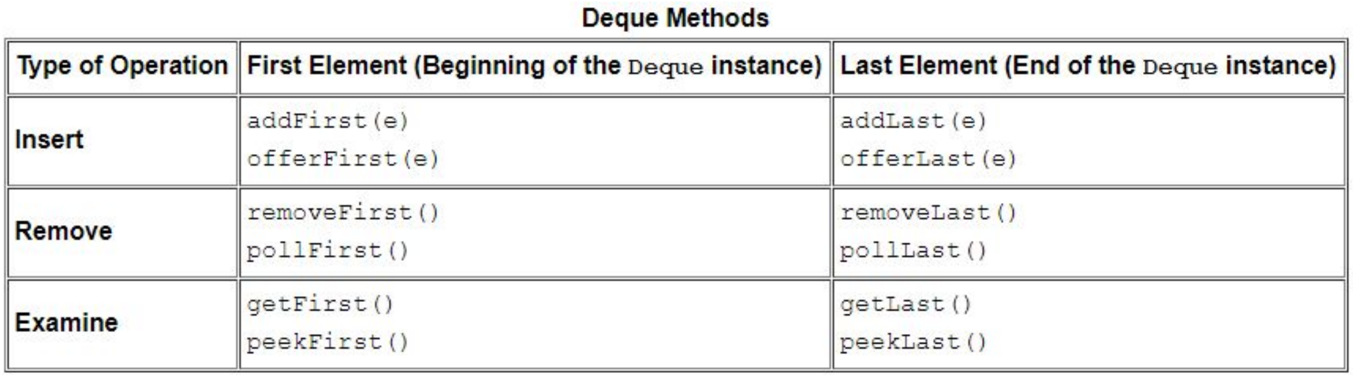


```java
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args) {
        // Create and initialize a Queue using a LinkedList
        Queue<String> waitingQueue = new LinkedList<>();

        // Adding new elements to the Queue (The Enqueue operation)
        waitingQueue.add("Rajeev");
        waitingQueue.add("Chris");
        waitingQueue.add("John");
        waitingQueue.add("Mark");
        waitingQueue.add("Steven");

        System.out.println("WaitingQueue : " + waitingQueue);

        // Removing an element from the Queue using remove() (The Dequeue operation)
        // The remove() method throws NoSuchElementException if the Queue is empty
        String name = waitingQueue.remove();
        System.out.println("Removed from WaitingQueue : " + name + " | New WaitingQueue : " + waitingQueue);

        // Removing an element from the Queue using poll()
        // The poll() method is similar to remove() except that it returns null if the Queue is empty.
        name = waitingQueue.poll();
        System.out.println("Removed from WaitingQueue : " + name + " | New WaitingQueue : " + waitingQueue);
    }
}
```

```
# Output
WaitingQueue : [Rajeev, Chris, John, Mark, Steven]
Removed from WaitingQueue : Rajeev | New WaitingQueue : [Chris, John, Mark, Steven]
Removed from WaitingQueue : Chris | New WaitingQueue : [John, Mark, Steven]
```

DEQUE INTERFACE

A **Deque** is a double-ended-queue, which is a linear collection of elements that supports the insertion and removal of elements at both end points. It has its own methods as well, which will help you **insert**, **remove** or **examine** the elements.



Deque Methods

Type of Operation	First Element (Beginning of the `Deque` instance)	Last Element (End of the `Deque` instance)

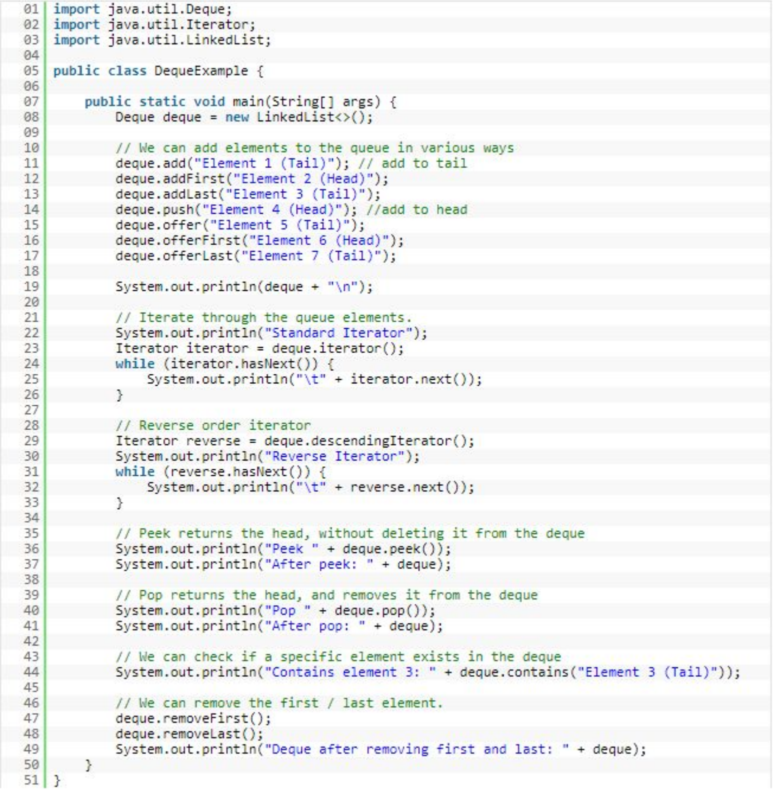
Insert	`addFirst(e)`
	`offerFirst(e)`
		`addLast(e)`
		`offerLast(e)`

Remove	`removeFirst()`
	`pollFirst()`
		`removeLast()`
		`pollLast()`

Examine	`getFirst()`
	`peekFirst()`
		`getLast()`
		`peekLast()`

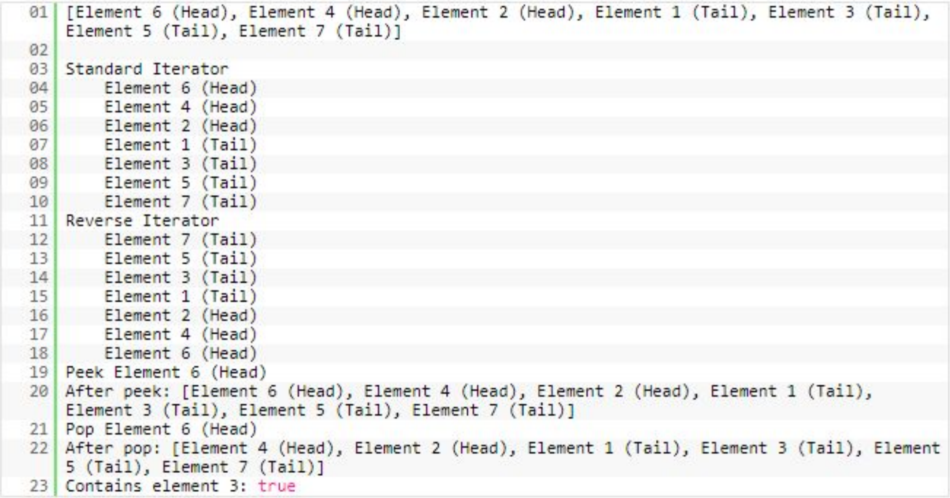
DEQUE INTERFACE – Examples

EXAMPLE



```java
import java.util.Deque;
import java.util.Iterator;
import java.util.LinkedList;

public class DequeExample {

    public static void main(String[] args) {
        Deque deque = new LinkedList<>();

        // We can add elements to the queue in various ways
        deque.add("Element 1 (Tail)"); // add to tail
        deque.addFirst("Element 2 (Head)");
        deque.addLast("Element 3 (Tail)");
        deque.push("Element 4 (Head)"); //add to head
        deque.offer("Element 5 (Tail)");
        deque.offerFirst("Element 6 (Head)");
        deque.offerLast("Element 7 (Tail)");

        System.out.println(deque + "\n");

        // Iterate through the queue elements.
        System.out.println("Standard Iterator");
        Iterator iterator = deque.iterator();
        while (iterator.hasNext()) {
            System.out.println("\t" + iterator.next());
        }

        // Reverse order iterator
        Iterator reverse = deque.descendingIterator();
        System.out.println("Reverse Iterator");
        while (reverse.hasNext()) {
            System.out.println("\t" + reverse.next());
        }

        // Peek returns the head, without deleting it from the deque
        System.out.println("Peek " + deque.peek());
        System.out.println("After peek: " + deque);

        // Pop returns the head, and removes it from the deque
        System.out.println("Pop " + deque.pop());
        System.out.println("After pop: " + deque);

        // We can check if a specific element exists in the deque
        System.out.println("Contains element 3: " + deque.contains("Element 3 (Tail)"));

        // We can remove the first / last element.
        deque.removeFirst();
        deque.removeLast();
        System.out.println("Deque after removing first and last: " + deque);
    }
}
```

OUTPUT



```
[Element 6 (Head), Element 4 (Head), Element 2 (Head), Element 1 (Tail), Element 3 (Tail), Element 5 (Tail), Element 7 (Tail)]

Standard Iterator
    Element 6 (Head)
    Element 4 (Head)
    Element 2 (Head)
    Element 1 (Tail)
    Element 3 (Tail)
    Element 5 (Tail)
    Element 7 (Tail)
Reverse Iterator
    Element 7 (Tail)
    Element 5 (Tail)
    Element 3 (Tail)
    Element 1 (Tail)
    Element 2 (Head)
    Element 4 (Head)
    Element 6 (Head)
Peek Element 6 (Head)
After peek: [Element 6 (Head), Element 4 (Head), Element 2 (Head), Element 1 (Tail), Element 3 (Tail), Element 5 (Tail), Element 7 (Tail)]
Pop Element 6 (Head)
After pop: [Element 4 (Head), Element 2 (Head), Element 1 (Tail), Element 3 (Tail), Element 5 (Tail), Element 7 (Tail)]
Contains element 3: true
```

MAP INTERFACE

The **Map** interface maps unique keys (objects) to values. A map can't contain duplicate keys since each key needs to map a value.

This interface includes methods for:

Basic operations

- put
- get
- remove
- containsKey
- containsValue
- size
- empty

Bulk operations

- putAll
- Clear

Collections view

- keySet
- entrySet
- values

MAP INTERFACE – Examples

Map can be implemented in different classes, such as **HashMap**, **TreeMap**, and **LinkedHashMap**.

EXAMPLE

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

   public static void main(String args[]) {
      // Create a hash map
      TreeMap tm = new TreeMap();

      // Put elements to the map
      tm.put("Zara", new Double(3434.34));
      tm.put("Mahnaz", new Double(123.22));
      tm.put("Ayan", new Double(1378.00));
      tm.put("Daisy", new Double(99.22));
      tm.put("Qadir", new Double(-19.08));

      // Get a set of the entries
      Set set = tm.entrySet();

      // Get an iterator
      Iterator i = set.iterator();

      // Display elements
      while(i.hasNext()) {
         Map.Entry me = (Map.Entry)i.next();
         System.out.print(me.getKey() + ": ");
         System.out.println(me.getValue());
      }
      System.out.println();

      // Deposit 1000 into Zara's account
      double balance = ((Double)tm.get("Zara")).doubleValue();
      tm.put("Zara", new Double(balance + 1000));
      System.out.println("Zara's new balance: " + tm.get("Zara"));
   }
}
```

OUTPUT

```
Ayan: 1378.0
Daisy: 99.22
Mahnaz: 123.22
Qadir: -19.08
Zara: 3434.34

Zara's new balance: 4434.34
```

“ SOURCES

- https://docs.oracle.com/javase/tutorial/collections/interfaces/index.html

- http://www.tutorialspoint.com/java/java_collections.htm

- https://examples.javacodegeeks.com/core-java/util/deque-util/java-util-deque-example/

THE END!